

Distribution Strategies for Persistent Java

S. Spence

Department of Computing Science
University of Glasgow, Glasgow, UK
susan@dcs.gla.ac.uk

Abstract

Some of the features of the model of distribution which has already been proposed for PJava are considered in detail in this paper. Focus is put on global naming, support for a large degree of autonomy of stores, the setting of conditions on remote use of objects and the use of timeouts in the management of remote references. Use is made of a running example to illustrate how a programmer would utilise such distribution support and to bring out some of the issues which need to be considered in order to further refine PJava's model of distribution.

1 Introduction

A model of distribution for PJava was first presented in [13]. This paper now focuses on a subset of the features described in the preceding paper, to consider them in more detail and to examine how these features could be used by applications running in a distributed persistent system.

When creating a model for a new distributed system, we would like to be able to conform as much as possible to the *ideal* model. Sape Mullender describes the state of the art for a general, distributed system as one with the accessibility, coherence and manageability advantages of a centralised system, plus the sharing, growth, cost and autonomy advantages of networked systems. Real security and high availability are also thrown in for good measure [8]. Add orthogonal persistence and, in the ideal persistent system [2], you also gain integrated support for application development and data management, consistent support for data that abstracts over its longevity, size and type and the presentation of a one-world model to the user. In this one-world model a wide range of applications should be able to run, with acceptable performance, over one consistent, logical store of data and code, of unlimited size; this should ideally be achieved with complete location transparency and unlimited scalability.

As both distributed systems and persistent systems literature acknowledge, in the real world, where current hardware technology and software support cannot meet these requirements, it is necessary to identify which aspects of the ideal system we are most interested in aspiring to and which ones we can realistically compromise on. Several criteria influence the model of distribution for PJava.

1. Store autonomy - a store should have control over the use of the objects it contains, including whether those objects can be copied, moved or referenced remotely. While it should be possible for remote stores to be able to hold remote references to objects the local store contains, the local store should not have to guarantee referential integrity forever.

2. True scalability - the mechanisms and data structures for distribution must scale not just to the scope of a Local Area Network (LAN) but also to applications interacting over a Wide Area Network (WAN), potentially involving hundreds or thousands of machines.
3. Flexibility - no single model of interaction should be imposed on applications in the distributed system; both small sets of closely-cooperating applications and larger collections of occasionally interacting applications should have suitable support with the costs of interaction being in proportion to the degree of interaction.
4. Practicality - solutions should work to the scale and with the performance required.

In the following sections, the current, non-distributed implementation of PJava is introduced, the requirements of Forest, a PJava application requiring distribution support, are examined and our current proposal for a model of distribution for PJava is described. After introducing a running example for the paper, some features of our model of distribution and the issues they raise are then considered in detail.

2 PJava

The design of PJava is described in [1]. It includes support for *orthogonal* persistence: any object can be made persistent, irrelevant of its type, size and lifetime. It also includes support for persistence by reachability: when a user makes an object persistent the underlying PJava implementation automatically ensures the persistence of not only the user-nominated object's attributes, its methods and its class and superclasses but also the attributes, methods, classes and superclasses of all objects referenced by it. PJava ensures that all new persistent objects or any updates to existing persistent objects are automatically propagated to the store (i.e. written to disk) on successful execution of code. It also ensures that no updates are propagated on failures. The use and implications of persistence are illustrated in section 2.1 below.

2.1 An Example of Persistence by Reachability

Development and maintenance of a set of components for support of work in the architect's firm "By Design" forms the basis for a running example in this paper. This simplistic example assumes that three architects Ann (A), Bob (B) and Zak (Z) each have their own development environment for designing houses. The example is first used here to illustrate how objects can be made persistent and how persistent objects are used.

Bob creates a new store, `storeB`, as his working environment and installs a library of frontage features (doors, windows, etc) to use for his design drawings of the outside of houses. The code run to achieve this task is as follows:

```
PJavaStore storeB = new PJavaStore('/local/design/stores/storeB');
LibraryIndex libraries = new LibraryIndex('Bob's libraries');
storeB.setPRoot(libraries);
Library frontageLibrary = new Library('Bob's frontage features');
// copies and adapts Features door, window and roof ...
frontageLibrary.setDefaultDoor(door);
frontageLibrary.setDefaultWindow(window);
frontageLibrary.setDefaultRoof(roof);
```

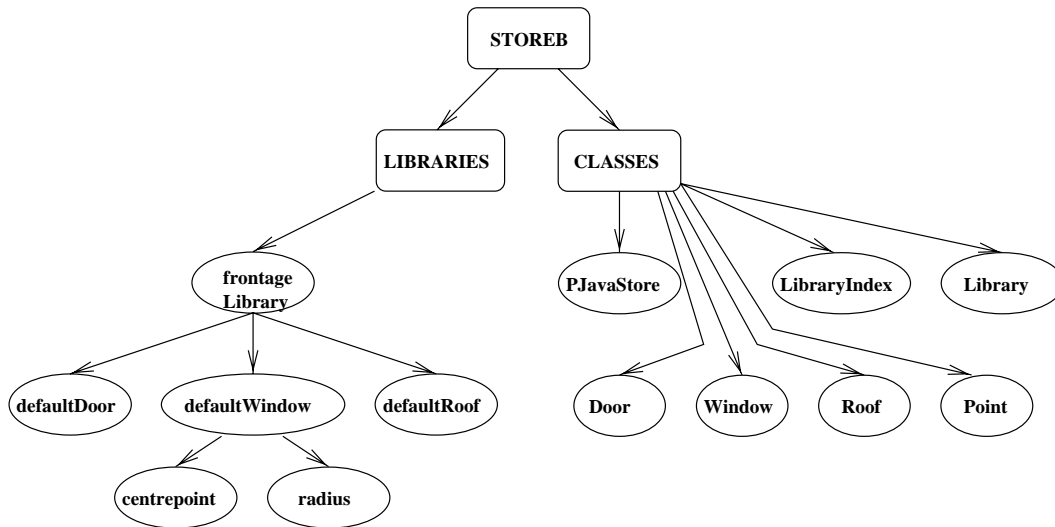


Figure 1: The Graph of storeB's Reachable Objects

```

...
Libraries.addLibrary(frontageLibrary);

```

Making the `frontageLibrary` reachable from `libraries`, which is a persistent root, results in all things reachable from `frontageLibrary` being made persistent too. Since the `door`, `window` and `roof` objects are attributes of `frontageLibrary`, the objects, including their attributes, methods and classes, will be made persistent. For example, the value of the `centrepoint` and `radius` attributes of the round window will be made persistent and the `Point` class, of which `centrepoint` is an instance, will be made persistent. Since the `frontageLibrary` is an instance of the `Library` class, the `Library` class will automatically be made persistent too.

After creating his own customised set of libraries, Bob is ready to start work on his first design. Normal interaction of a program with a store involves the setting of references to objects in the store and the calling of methods of those objects in the normal way. The following example is the code used to support Bob's use of library objects in his store for starting his design. The program is run with an option that specifies it will use the store `storeB`.

```

PjavaStore storeB = getStore();
LibraryIndex libraries = storeB.getPRoot("Bob's libraries");
Library frontageLib = libraries.getLibrary("Bob's frontage features");
HouseDesign design1 = new HouseDesign();
design1.door = frontageLib.getDefaultDoor();
design1.window1 = frontageLib.getDefaultWindow();

```

3 Forest: PJava's Killer Application

While support for persistence on the scale of a single store is a good start, support for *distributed* persistent stores is necessary to allow persistent object storage to scale beyond the capacity of a single machine and to support the use of objects in

distributed stores by distributed computations. The model for distributed PJava is intended to support a range of distributed application systems, including Forest, a software development environment currently being developed by our collaborators at Sun Microsystems Laboratories. In this section the support necessary for Forest's use of distribution is considered.

Forest is an integrated, multi-user, software development environment for medium to large-scale systems programming [5]. Its implementation of configuration management is based on the use of typed, immutable objects combined with versioning. The configuration management manipulates objects which are referred to as components. These components range from small objects that, for example, represent a fragment of a source program through to large, complex, structured objects that can themselves be composed of thousands of objects. The components are organised into packages, each of which is a tree of versions of those components. The packages are, in turn, grouped into projects. A project is shared by a group of programmers; a user in this group typically checks out the latest version of a package to work on. The user evolves (i.e. works on) a copy of the latest version of the package, since the original components are immutable. One of the benefits of the immutability of components is that it supports the sharing of common structure, such as the components representing a library of generally-useful routines for example.

Each component has a globally-unique, 128-bit identifier, part of which encodes the project that contains it. To avoid performance problems of UID generation, this is distributed among components in a project. Use of this UID frees Forest from the implementation details of the indexing system of any particular database.

Work on packages is done in the context of transactions; which should be short in order to minimise locking conflicts. Read-only transactions support the browsing of packages while update transactions support editing and building of sources.

Because a component is immutable, it can be freely copied between projects at different sites. The current theory for the use of Forest is that a group of programmers will share a single project and sharing between larger groups should be supported by copying a version of a package rather than through use of inter-project references.

Thus, the model of distribution to support Forest should support both local sharing of a store and sharing of components between stores. Interaction with the stores requires support for transactions, or at least provision of transaction primitives. The ability to copy components from one store to another is also required, but the immutability of committed components does not need to be enforced at that level since it is imposed by the configuration management of Forest itself.

4 A Model of Distribution for PJava

"A Scalable Model of Distribution Promoting Autonomy of and Cooperation between PJava Object Stores" has been proposed in [13]. This model is intended to meet the requirements of a range of persistent, distributed application systems including Forest.

In our distribution model, emphasis is put on support for both largely-autonomous stores that have only a small amount of interaction with other stores as well as *small* groups of closely cooperating stores that, for example, require to frequently share objects over a length of time in order to perform distributed computation.

Stores gain autonomy by being able to control and limit any access to and use of the objects they contain. By making it possible for a store to specify which objects may be made available for remote use and by advocating the use of timeouts on remote references, the PJava model of distribution allows a store to limit the number of objects which can be remotely referenced and to associate a timeout with these remote references. The timeout specifies the time until which a store intends to guarantee availability of an object for remote use and provides a way for the store to avoid having to try to guarantee referential integrity forever (since the latter guarantee can be considered unrealistic in the face of an evolving, long-running system with failures). Timeouts also allow the PJava distribution model to avoid the problems of supporting distributed garbage collection, since a store can use timeouts to ensure that remote references no longer exist before performing a full garbage collection of its own contents. Garbage collection of the store can still take place *before* timeouts come into effect, but they must be conservative with respect to objects that have been made available for remote use, since the store cannot tell whether they are remotely referenced or not.

As well as being able to specify which objects are available for remote use and the timeouts on references to them, a store can also set other conditions on use, thus supporting greater control and autonomy of the store containing the objects. These specify whether an object can be copied, moved or updated by a remote application and which methods of the objects can be remotely invoked. The implications of each condition are described in [13].

As required by Forest, and as it is desirable for support of scalability, distributed PJava is intended to support applications not just at the scale of a LAN but also to the scale of large numbers of machines on a WAN. This requires consideration of support for both continuous and sporadic connections and support for coping with latency. Large-scale distribution requires global naming of objects and the design of distribution management data structures which support the PJava model scalably and efficiently (e.g. with use of data structures that grow non-linearly). Support for the use and communication of bulk types is intended to be included in PJava too, in order that large-scale systems with large collections of data benefit from integrated support for them.

Having introduced the main concepts of the PJava distribution model, the rest of this paper concentrates on an examination of the issues of global naming, support for autonomy through use of conditions including timeouts and examples that illustrate how such concepts might be used.

4.1 Distribution context

The architects example is now used here to demonstrate the context of distributed PJava. It is envisaged that distribution of stores will be over a WAN, usually with one store per machine and a number of applications running over a store. It should be possible for multiple stores to be run on a single machine but interaction between these can be treated as interaction between *remote* stores as long as global naming distinguishes between them. In line with the multi-platform nature of Java¹, no restrictive assumptions should be made about the support provided by the operating system or hardware.

In figure 2, Ann and Bob both use their own machines (conveniently given their own names) at the Glasgow office for their design work. They each have their own

¹Java is a registered trademark of Sun Microsystems Inc. in the USA and other countries

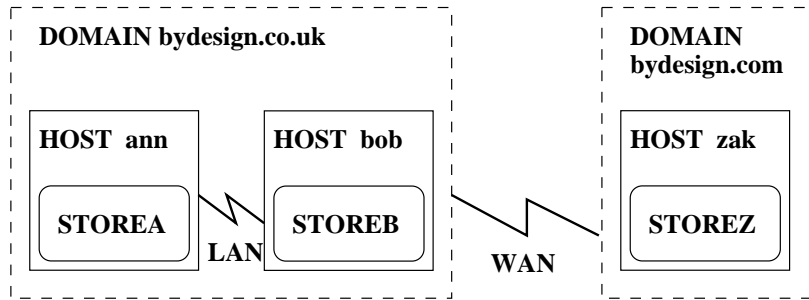


Figure 2: The Configuration of Ann, Bob and Zak's Machines

store, where they keep their current and past designs plus standard libraries and tools and they communicate via a LAN when they need to use objects located in other stores within the same domain. There is no reason why Ann and Bob's stores could not be located on the same machine. Interaction between Ann and Bob's stores, in this case, would still be modelled as sharing of remote objects, since PJava interprets remoteness in the context of stores rather than machines hosting them. Meanwhile, Zak is busy selling houses from the California office. He has his own store of information on currently-available house designs and if he needs to access information held in Glasgow he must do so via a WAN.

5 Naming

Having introduced the context of distributed PJava, support for global naming and autonomy are features of the model which are to be considered here in detail. On the subject of global naming, ideally it should be possible for each store to be able to create and name their objects independently of any global control while at the same time allowing users to share objects between stores.

In order for applications to be able to share objects between stores, it is necessary to be able to name the objects, the stores containing them and the machines hosting them. Issues of longevity, scale and transparency must be considered. Persistent applications use object names that must be valid for not just the execution time of the application, but also for the lifetime of the store containing a persistent application and the persistent objects that it uses. Consideration must be given to what naming scheme and supporting implementation is sufficiently flexible to cope with changes over such potentially long lifetimes, and whether it is also scalable. Distributed PJava should scale to a WAN with many machines, so the names used for stores and objects must work in that context and the implementation's distribution data structures must scale to be able to cope with large-scale systems. The issue of whether location transparency and location independence should be maintained also needs to be considered. In choosing whether to support such desirable features in a distribution model for an implementable and usable system, appropriate methods of naming and the costs of their implementation must also be considered.

Location-transparency can be supported to different degrees: for example, the object name at the user-level can give no hint of its physical address or it can be named relative to a location-transparent store or server name. If the name is independent, as well as location-transparent, it gives no indication of the machine

it is on, the store that contains it or the path from which it is reachable in that store. This means it can be moved from one store to another without changing names. However, although this provides an extremely simple user model and complete flexibility in terms of coping with changes to stores and objects, maintaining the consistency of the required mappings of these location-transparent, location-independent object names to their physical locations is difficult to achieve in a scalable, distributed system with acceptable performance.

Global naming can be provided over either a large, flat namespace or a hierarchical one. However, maintaining a single, flat namespace requires a way to uniquely identify objects that may have been created on different sites with the same user-level name, as well as mapping names to object locations.

Both a flat namespace and a hierarchical one can make use of a name service to map the high level object names to low level physical addresses. In a scalable, fault-tolerant, distributed system this requires multiple name servers with either the name space split up between them, a potentially expensive protocol for ensuring unique naming or use of a hierarchical naming scheme where uniqueness within a restricted scope can be left to the user.

The simplest form of hierarchical naming involves the machine, store and path of the object all being specified explicitly. This means that the physical addressing of the object is hardwired into application code. This is not transparent or flexible since moving the object would require a user to edit the code. Thus, moving of objects between stores or moving of stores is not possible.

However, hierarchical naming schemes can be a good and proven way to ensure unique naming of objects in a large, distributed system, since it is reasonable to expect a user to be able to maintain unique naming within one part of a hierarchy [10]. Since some degree of location independence is desirable to allow adaption to the inevitable changes in system configuration of systems running persistent applications over the lifetime of their associated store, it seems reasonable to use a hierarchical naming scheme that names objects relative to a location-transparent store name. This means that the store can move from one machine to another in the distributed system without the name of the store and all objects named relative to it having to be changed. At the implementation level, this would require a name server to map store names to physical locations. To ensure scalability, the name server would have to be distributed and a method of maintaining consistent information across these servers would be necessary. In a large system, the information held at each name server could be a lot but it is envisaged that stores would not move their location very often. Reasons why it might be necessary to move a store include fault-tolerance and availability or reconfiguration within an organisation. Thus, a service mapping store names to machines that is similar to the Domain Name Service (DNS), that maps machine names to IP addresses, should be suitable.

5.1 The Use of POLs and PONs for Naming

As has been proved by the wide use of WWW browsers, it is possible to name objects such as hyper-text information pages, GIF images and Java applets, made available anywhere on the Internet, through use of a URL. Demonstrably, the hierarchical form of naming provided by a URL does scale to a WAN and supports unique naming of millions of objects. A URL maps to a file in a directory hierarchy on a machine in a domain. The proposed PJava naming scheme will use

an extended form of URL, called a Persistent Object Locator (POL), which maps to an object in a persistent store on a machine in a domain.

Presentation of the real path to an object within a store is not necessary when making it available for remote use. When an object is specified as available, it could be added to an in-store index of objects that are available for remote use. If the index itself is made available to remote users, it can provide the mapping while still hiding the real path to the object in the store. This has the added benefit that if the object becomes unreachable from elsewhere in the store, it will still be reachable from that index and will not be garbage-collected as long as the store continues to make it available for remote use.

While use of a POL requires the user to know the domain in which the store to be accessed is located, it is also possible to support use of a more location-transparent name. Just as there are proposals to complement URLs with more location-transparent URNs [12], a Persistent Object Name (PON), could be supported which maps from just the name of an object within a store to its physical location. Since the maintenance of consistent, distributed name servers for support of such mapping incurs a performance cost, support for this type of naming is only intended to be supported by distributed PJava for a small set of closely-cooperating stores.

It seems reasonable that a set of closely-cooperating stores can afford to pay the cost of small-scale, highly location-transparent facilities in order to gain in the efficiency of their cooperation over a period of time. It is envisaged that PJava will specify guidelines on the number of stores that can use facilities for support of close cooperation (e.g. to ten initially, until experiments can help us to judge what is a realistic restriction on the number of closely-cooperating stores). This avoids such cooperating applications trying to scale up beyond the point where reasonable performance can be expected for the facilities supported.

5.1.1 An Example of a POL

The following POL identifies a door object in Bob's store, accessible from the machine servicing outside requests at the Glasgow branch of the company "By Design".

```
pol://bydesign.co.uk/storeB.door
```

This identifies the machine by mapping the domain name *bydesign.co.uk* to a particular machine. The port number is not specified here, so it is assumed that a service for dealing with access requests to a store has been registered at a fixed portnumber by the local administrator. The service accesses the specified store to find the object *door*.

A POL is likely to be the most suitable form of naming for an object referenced sporadically by remote, largely-independent stores, since it requires less overhead to map the POL to the actual object than the more location-transparent naming scheme would. Thus, if Zak from the sales team, who operates over a store on a different continent from Bob and only accesses the stores in By Design's Glasgow office infrequently, decides to display Bob's current door design to a customer, a POL is the most suitable form of naming of the door object for him to use. In comparison, a PON could also be used to name the same object where the setting up and maintenance of the location-transparency provided by the necessary mapping pays off. Since Ann and Bob work in the same design team and often share components when making up new designs, it makes sense for Ann to be able to use a PON to refer to Bob's door.

6 Autonomy

For reasons of protection and autonomy, it is desirable for a store to have control over the objects it contains. It is likely that only a subset of the objects in one store need to be made available for remote use because, for example, the rest are private to the applications that run over that particular store. Also, allowing any application in a distributed system to have remote references to any object in any store has a tendency to lead to a spaghetti of inter-store references which are difficult to manage and control. While it may be reasonable to expect (though not necessarily to get) users to manage the growth of inter-store references in a very small, distributed system through disciplined programming, this becomes infeasible for an evolving, long-running system, never mind at a larger scale. Another aspect of control that a store may require is the ability to say which objects can be copied to another store and which cannot. It is intended that distributed PJava will enforce a policy (e.g. through the access mechanism provided) that restricts visibility of the contents of a store to only those objects which have explicitly been made visible. Thus, if a store does not make an object visible to remote stores, they cannot copy it.

As described in [13], it is our intention to support a range of conditions on object use, including whether an object can be remotely-referenced, copied, moved and updated. If implementation of an index of remotely-available objects for each store is chosen, it seems reasonable to enforce the conditions of use, such as whether the object can be copied, by making a set of appropriate methods available. Thus if the object cannot be copied, there should be no method available for a remote store to be able to gain a copy of that object. Since the autonomy and control of the store containing a particular object is emphasised, it seems reasonable to expect that store to impose any protection necessary. Thus, if only certain remote sites should have access to a particular object, an access control list should be associated with the object in the index of remotely-available objects.

In Forest, as described in section 3, groups of programmers would normally work over one store and would access a remote store if, for example, they wished to import a copy of a package for use in their own application. However, programmers working on a package that is not yet stable are not likely to want it to be possible to import a copy of that package outside of its store. On the other hand, if a number of groups are cooperating on building a large application then it is desirable for a large degree of interaction between a small number of stores to be supported.

In general, applications in a distributed system tend to interact either intensively in small groups, where they need a lot of support for such interaction, or else sporadically, connecting just long enough to gain information or use a service. When considering how to support large scale, distributed systems, it seems desirable to support both models of interaction.

6.1 Managing references with the aid of timeouts

As described in [3], site autonomy implies that each site must be capable of managing its own data regardless of other sites. There are a number of reference management issues which must be addressed in order for one site to be largely autonomous. These include distributed garbage collection and stabilisation.

An application wishing to maintain a great deal of autonomous control over its store would ideally like to be able to ignore the problems of both distributed garbage-collection and distributed stabilisation. It could then focus on garbage collection within the one store and stabilise any updates it makes without having to worry about coordinating with or taking account of other stores. However, this is in direct conflict with a desire to share objects between stores. If an application wishes to be able to use an object from another store or make objects from its store available for remote use, it must compromise on coordinating house-keeping activities with other stores in some way. The association of timeouts with remote references is being considered for distributed PJava as a way of supporting such a compromise.

The problem with garbage collection over distributed stores is the difficulty of working out when it becomes safe to garbage-collect an object which may be remotely-referenced. One way to solve this problem is to employ reference counting: the object keeps a count of all the local and remote references to it and, when the count is zero, the object can be discarded. A number of reference counting techniques are described in [11]. Problems with distributed implementations of reference counting include the need for causal ordering of messages to implement the technique and failure to collect cyclic garbage. It is envisaged that the use of timeouts in PJava could avoid the need for distributed garbage collection protocols altogether since, once timeouts have expired on remotely visible objects, they can be collected locally.

Stabilisation is employed in PJava to write new persistent objects and updates to existing persistent objects back to stable storage (disk). In a distributed system such updates can be done across distributed stores, either implicitly when an application unknowingly works over distributed objects in a location-transparent system that presents one logical store or explicitly during a distributed transaction. Distributed stabilisation involves taking a snapshot of all the currently reachable objects and writing them back to disk; this implies “stopping the world” and traversing the graph of all reachable objects throughout the distributed system which is difficult in a small-scale, distributed system and practically impossible at a larger scale because the delay caused in application execution becomes unacceptable.

In order to support timeouts, it at first seemed necessary to consider the problems of global time in a system distributed over a WAN. A lot of research that has been done on clock synchronisation, including [6], makes assumptions about the distributed system in which it works so it is difficult to find an algorithm which scales acceptably to a WAN and which has realistic assumptions about the operation of a real, distributed system that, for example, handles communication failures. Thus, if timeouts are to be used at all, it must be with a proven and realistic protocol for applications on different machines to be able to set and use them.

The Network Time Protocol, as proposed by [7], is used as a standard for clock synchronisation throughout the Internet. Values taken from servers synchronised using this protocol have shown that the synchronisation error is below 30 milliseconds for all but one per cent of the samples taken. No bounded clock drift can be guaranteed but this does give some guidance on what can be expected. On this basis, the timeout given for a reference could be interpreted by the remote site as the given timeout minus an allowance for clock drift in order to ensure the remote site uses a reasonably conservative estimate of the timeout. Thus, if they respect this timeout when using the remote reference, there is a high probability

that they will be able to use the referenced object. Where this is not sufficient, and if the remote site is operational, they should receive an exception when the referenced object is no longer available for remote use.

However, the *use* of timeouts by remote applications is potentially much more complex. In theory, the application knows the time limit on the remote reference it is using and can make use of that remote reference until the timeout occurs. In practise, the application needs to schedule the execution of the code that uses the remote references in a way that ensures that if, for example, it is running a transaction, it has access to the remote references it needs until completion of that transaction. In a heterogeneous environment, the same transaction will take different lengths of time to complete on different architectures so knowing when the transaction will end is no trivial matter. This also takes no account of communication delays or failures.

Use of fine-grained timeouts (milliseconds, seconds) by distributed applications is likely to be impossible to implement in distributed PJava because of the difficulty of maintaining fine-grained clock synchronisation. However, this does not preclude the use of more coarse-grained timeouts, at a higher level of decision-making, that do not require such accuracy of coordination. If the timeouts are specified in hours or even days, it is likely that a user is then able to judge, given the timeout information, whether the application they wish to run is likely to finish within the time limit for remote references that it uses. Experience informs the user's decision: a small test program is likely to only take a few seconds or minutes, a compilation may take somewhere between a few minutes and a few hours and measurements could be done overnight or may take perhaps several days to execute. At these levels of granularity, the synchronisation of clocks on different machines throughout a WAN is unlikely to be an issue and the use of timeouts does become feasible.

7 Using the Distributed PJava Model

7.1 Making Objects Available for Remote Use

In PJava's distribution model, only an application running over a local store can make objects visible for use by applications running over other stores.² An index of `VisibleRoots`, registered as a persistent root in a store, has been chosen as the method for accessing all objects in a store that are visible for remote referencing. Objects of the store are made visible by being made reachable from the `VisibleRoots` index. Alternative models of visibility include making every visible object a root of persistence itself.

To conform with the normal Java programming model, when an object is made remotely visible, it should be possible for a remote site to only be able to access the public methods and, although generally inadvisable on the grounds of encapsulation, the public attributes, of the object. On specification of visible objects by a user, the underlying PJava implementation should automatically generate a stub for each public method of each visible object in order that they may be remotely invoked. It should be possible for a user to specify, when they

²The term *visibility* of objects is used below to describe those made available for remote use. The term *availability* in distributed systems has its own connotations, usually of replication for availability. The term *publish* was considered but rejected because of its usual association with a corresponding subscribe mechanism. Subscription is not supported in PJava since the stores containing referenced objects do not keep a record of the remote applications accessing those objects.

make an object visible, whether some of the otherwise public methods of the object should be invocable from a remote site or not. Refined protection mechanisms could be considered to restrict the invocation of a specified set of methods to an approved set of remote users but this will be more costly.

A set of conditions on object use are associated with each object that is made visible:

timeout This specifies the time until which the object is guaranteed to be visible to remote users. It should be possible for an extension to a timeout to be negotiated. The criteria for the setting of a default timeout and for a store's judgement over extension of timeouts needs to be considered.

copy It may be reasonable to make it possible for a remote user to invoke one or more methods on an object while not actually being able to make a copy of that object.

update If update is not allowed then the object is assumed to be accessible for reading only.

move It is reasonable to have a store enforce no movement of objects by providing no method in the store interface for a remote user to be able to access the object for such a purpose. However, where this is desirable, the use and implications of something like proxies must be considered; other remote applications may still have references to the object in its old location, while direct access to the object's new location also needs to be supported.

invoke If an object is visible but support for invocation of only a subset of its public methods is to be provided then the user must be able to specify which methods can be made visible.

The implications of dynamic changes to object conditions have still to be considered. A well-defined model must produce predictable behaviour if this is to be supported. It may be that the changes apply to all accesses to the visible object from the point when the conditions are changed but this provides no support for the remote user to find out what changes have been made to the conditions, except through reception of exceptions when an operation, which was expected to succeed, actually fails; this is likely to be unacceptable!

There are a number of issues to be considered if the conditions set on a visible object allow it to be moved. The use of timeouts on references may allow us to move the referenced object after the time of availability has expired. Issues to be considered for support of this include:

- if the object is made visible again, this time from its new location, can a remote site perceive it as the *same* object, just in a different location?
- it is not necessary to wait for a timeout to move an object if that object is visible directly (rather than reachable from another visible object) and a `VisibleRoots` set maps the object to whatever its location is within one store.
- it is not necessary to wait for a timeout to move an object if a proxy is placed in the `VisibleRoots` set to redirect use of the object to its new location. However, this amounts to supporting the user unknowingly creating lots of remote references if objects are allowed to move between stores in this way. Can the use of timeouts still be considered sufficient to control them then?

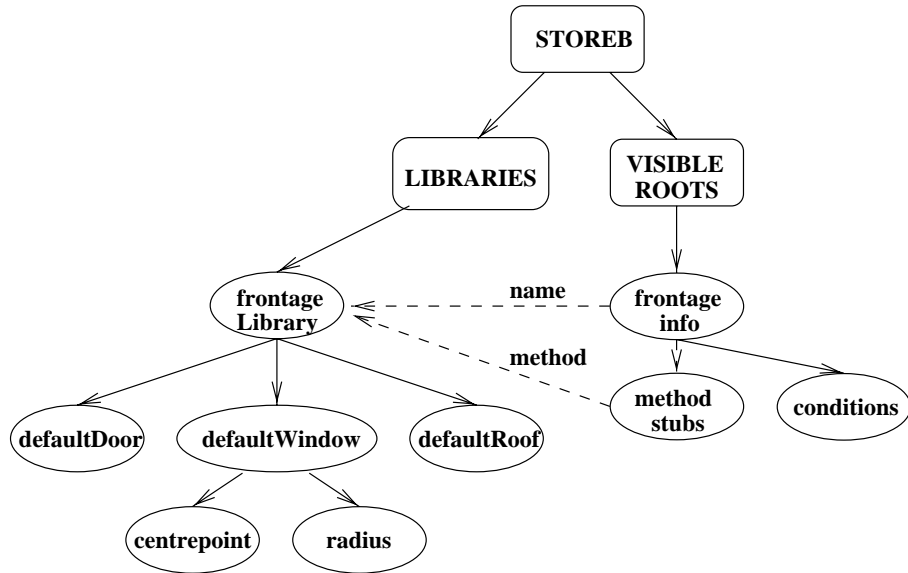


Figure 3: The Use of a VisibleRoots Object in a Store

Consideration of concurrent use of a store complicates the setting of visible objects and their conditions of use considerably. If two threads run over the same store and both want to make the same object visible but with different conditions, then what conditions are actually set on the object? It may also be the case that the two threads would like to make the object visible to two, not necessarily overlapping sets of remote users. The implications of concurrent use of conditions need further consideration.

7.1.1 An Example of Making Objects Visible

Bob, having developed his first house design, shows it to Ann. She is particularly impressed by the set of frontage features that he has used. He agrees to make his customised library of frontage features available for her to use. The following code is run to achieve this task, over the store `storeB`.

```

PjavaStore storeB = getStore();
LibraryIndex libraries = storeB.getPRoot("Bob's libraries");
Library frontageLib = libraries.getLibrary("Bob's frontage features");
storeB.setVisible(frontageLib, timeLimit, copy, update, invoke);

```

This has the effect of adding a reference to the `frontageLib` object to the `VisibleRoots` index of `storeB`, as illustrated in figure 3.

The `VisibleRoots` index contains a reference to an object containing information about the `frontageLibrary` visible object. This information includes a global name for the real location of the object (indicated by the dotted line), a reference to an object logging conditions of use and another to the object containing the stubs corresponding to each method of the real object (the relation here is also indicated by a dotted line). The value of `copy` on the `frontageLibrary` visible object is set to true, while that of `update` is set to false, so the interface generated for the object permits only copying and not updating of the object while it is in `storeB`. A stub is also generated for each public method of the object that unmarshalls parameters, calls the real method and marshals the results.

7.2 Finding Out about Remote Objects

A model of interaction between an application and the store containing objects it wishes to use is considered below.

An application wishing to find out about a remote store's visible objects must send a query to the service providing an interface to that store. It seems likely that every store should provide an interface service, whether or not it contains any objects that have been made visible for remote use. If it didn't, it would be difficult for an application to tell the difference between a store with no interface service and a store which has become unavailable because of communication or process failure. Thus a nominal service would be useful, even if it only informs remote applications about the lack of visible objects in the corresponding store.

A standard PJava class `RemoteAccessClass` could provide methods for finding out about services provided by other stores. An application could call either the `getStoreAvailability` method to find out about all the objects visible in a store or the `getObjectAvailability` method to find out whether a specific object is visible in a store. It is then possible to set up an interface to use the object by making a call to `getObject` to obtain a reference to it. It would help if the user was able to give some indication of whether frequent interactions are likely with this other store in order to know whether to establish a long-term connection or whether to make a one-off connection between the application's site and the site of the store.

The call to `getStoreAvailability` would return a vector of objects containing information on the global name and conditions of use of each visible object in the specified store. These objects are likely to be comparable to capabilities; as used in Amoeba [9] for example, a user is given access to an object by being given a capability for that object. A subsequent call to `getObject` could result in a proxy being created for that object with methods that should enforce the conditions on object use and make the public methods available for calling with support of something like RMI [4] underneath. If it is desirable that these proxy objects should persist beyond the execution of the current application or that they should tolerate crashes of the application then they should be made persistent.

7.2.1 An Example of Finding Out About Visible Objects

In order for Ann to use Bob's library of frontages, the code supporting her task uses methods of `RemoteAccessClass` which is one of the standard classes provided as a persistent root in her store. A query is sent to the service providing an interface to Bob's store, `storeB`. An object containing information on the methods of the visible `frontageLib` object in `storeB` is passed to the `RemoteAccessClass` instance. The code within `RemoteAccessClass` creates a stub object providing access to the methods of the object and returns that object as the result of the query. The code running over Ann's store sets a local reference to this object and can now use the methods of it. (The conditions of use on the object can also be queried via the `RemoteAccessClass`).

```
RemoteAccessClass remoteAccess = storeA.getPRoot('remoteAccess');
Library bobsFrontageLib =
    remoteAccess.getObject('pon:storeB.frontageLib');
```

Once a remote reference has been set up to a visible object, and stubs have implicitly been created for its methods, a call to a method of the object should appear to the user to be the same as a call to a local method within the application

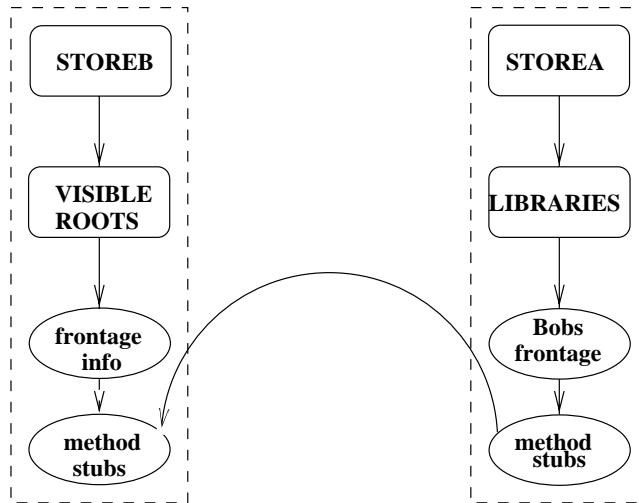


Figure 4: The Use of a Remote Method in a Store

program. Thus, if Ann decides to create a house design with a door in the style of the one in Bob’s library, the code supporting her task will use it directly.

```
PjavaStore storeA = getStore();
HouseDesign design20 = new HouseDesign();
design20.door = bobsFrontageLib.getDefaultDoor();
```

This should invoke the wrapper for the `getDefaultDoor` method in the stub `frontageLib` object. The wrapper would marshal the arguments if there were any and use something like RMI to invoke the real method at the remote store. This is illustrated in figure 4.

8 Conclusion

Some of the issues raised by the current proposed model of distributed PJava, have been examined in detail in this paper. The use of the suitable scalable hierarchical naming has been introduced and a model for making objects visible and for their remote use has been described. More consideration is needed about the implications of setting and resetting conditions on objects. Timeouts should be usable if they are set at a suitable level of coarse granularity, thus avoiding synchronisation problems. It is likely that it will only be possible to come to a reasonable and practical conclusion about some of the issues raised here after some initial experimentation has been done to test out current theories.

9 Acknowledgements

The PJava project is funded by Sun Microsystems Laboratories and the EPSRC. Thanks are due to those who reviewed this paper; especially Huw Evans, who was particularly thorough!

References

- [1] M. Atkinson, M. Jordan, L. Daynès, and S. Spence. Design issues for persistent java: a type-safe, object-oriented, orthogonally persistent system. In *Proceedings of The Seventh International Workshop on Persistent Object Systems*, Cape May, New Jersey, USA, May 1996.
- [2] M. Atkinson and R. Morrison. Orthogonal persistent object systems. *VLDB Journal*, 4(3):319–401, 1995.
- [3] S. Ceri and G. Pelagatti. *Distributed Databases: Principles and Systems*. McGraw-Hill, 1984.
- [4] Sun Microsystems Inc. Java remote method invocation specification, draft revision 0.9. <http://chatsubo.javasoft.com/current/doc/rmi-spec/rmiTOC.doc.html>, May 1996.
- [5] M. Jordan and M. Van De Vanter. Software configuration management in an object-oriented database. In *USENIX Conference on Object-Oriented Technologies (COOTS)*, Monterey, California, June 1995.
- [6] L. Lamport and P.M. Melliar-Smith. Synchronising clocks in the presence of faults. *Journal of the ACM*, 32(1):52–78, Jan 1985.
- [7] D.L. Mills. Internet time synchronisation: the network time protocol. *IEEE Transactions on Communication*, 39(10):1482–93, 1991.
- [8] S. Mullender, editor. *Distributed Systems*. Addison-Wesley, second edition, 1993.
- [9] S.J. Mullender, G. van Rossum, A.S. Tanenbaum, R. van Renesse, and H. van Stavaren. Amoeba: A distributed operating system for the 1990s. *IEEE Computer*, 23(5):44–53, 1990.
- [10] R. Needham. Names. In *Distributed Systems*. Addison-Wesley. Second Edition., 1993.
- [11] D. Plainfosse and M. Shapiro. A survey of distributed garbage collection techniques. In *Proceedings of the International Workshop on Memory Management (IWMM95)*, pages 211–250. Springer, 1995.
- [12] K. Sollins and L. Masinter. Functional requirements for uniform resource names. RFC 1737, December 1994.
- [13] S. Spence and M. Atkinson. A scalable model of distribution promoting autonomy of and cooperation between pjava object stores. In *Proceedings of the Thirtieth Hawaii International Conference on System Sciences*, Hawaii, USA, January 1997. to be published.