A Scalable Model of Distribution Promoting Autonomy of and Cooperation between PJava Object Stores

S. Spence and M.P. Atkinson

Department of Computing Science
University of Glasgow, GLASGOW, UK
{susan, mpa\@dcs.gla.ac.uk}

Abstract

Distributed persistent systems often provide transparency of distribution at the cost of autonomy of object stores, efficiency of cooperation between them and scalability of the system. We propose a flexible model of distribution to support scalable persistent application systems that allows for both autonomous stores with low-cost, limited commitments to other stores and sophisticated models of interaction on a limited scale with maintainable inter-store references. Scalability would be provided by exposing the distribution of stores, with the use of extended URLs as global names. Autonomy of a store would be preserved by making it possible for the store to retain control over the objects it makes available for remote use.

1. Introduction

A truly-scalable model of distribution allowing a large degree of autonomy of stores, but also providing support for efficient cooperation between limited numbers of stores, is required for support of large, persistent application systems.

This paper presents a model for support of persistent, distributed, object-oriented applications running on a system of nodes over local or wide area networks. It is intended to support distribution for Persistent Java [3] but the model could be generalised to other persistent programming languages such as Napier88. Whereas some systems have inflexible policies for distribution support built into them so they support only applications that make the same trade-offs, we consider flexibility of support for a wide range of applications is feasible if sufficient thought is given to the combination of supported features. The exchangeable user-level services of the well-designed, distributed support system, Horus [25], demonstrate that flexibility, support of a range of policies *and* performance is feasible.

The one-world model for persistent systems is an ideal - truly-scalable *and* location-transparent, with one consistent, logical store of unlimited size and reasonable performance for supporting a range of persistent application systems. The reality is that trade-offs are made for the provision of location-transparency, consistency and other desirable features. Requirements for distributed applications and a large logical store necessitate distribution of physical stores. Maintenance of location-transparency over the distributed stores impacts on performance and scalability. Autonomy of stores becomes desirable to control the use and evolution of persistent data since the management of dependencies between stores (e.g. for stabilisation and garbage-collection) also impacts on performance and scalability of the overall system.

As stated in [28], robustness and reliability in large-scale systems can best be supported if distribution is exposed and the programmer can address issues of latency, concurrency and partial failures. Experience with Distributed PS-Algol [2] has shown that truly-scalable distributed systems are only possible through exposure of the distribution. Thus, we propose the use of an extended URL for global naming, which can be used to name remote stores and objects within those stores in a non-location-transparent manner. Where it is a requirement for a distributed application to have efficient cooperation between limited numbers of stores, an extra level of support, built on top of the basic distribution primitives, will be provided.

Dependencies are created between stores when remote references proliferate and when code executing over one store modifies data on one or more remote stores. The cost of tracking these dependencies, in order to maintain consistency and perform distributed stabilisation and distributed garbage collection, must be considered. This cost is exacerbated by the necessity of coping with failures. It affects the performance of applications and limits the acceptable scale within which these dependencies can be realistically maintained. By allowing stores to retain control over the availability of their own objects, dependencies can be strictly controlled and restricted where

Proceedings of The Thirtieth Annual Hawwaii International Conference on System Sciences ISBN 0-8186-7862-3/97 $17.00 © 1997 IEEE

1060-3425/97 $10.00 (c) 1997 IEEE

required. Support will also be provided for stores to retain control over the use of their objects for inter-store references, copying of objects from one store to another and invocation of remote methods.

The use of extended URLs for global naming is described in section 2. This is followed by a description of how a store makes available and uses both these extended URLs and other information associated with objects on their conditions of use in section 3., section 4. and section 5. Issues associated with the usage of distributed objects and methods are also considered in those sections. Related work and technologies for supporting policies of use are then considered in section 6. and section 7.

1.1. Context

The assumptions made in this paper include these below.

- The proposed model is intended to support distribution for PJava, an implementation of the object-oriented language Java with support for orthogonal persistence provided via the class PJavaStore and supported by a stable store. Any Java object may persist if it is registered with an instance of a PJavaStore as a persistent root or if it is made reachable from a root of persistence.
- The topology of the distributed system using our model is expected to scale to a large number of machines running over a WAN with one or more stores per machine and one or more threads of computation running over each store. The interaction between two stores on one machine can be treated in the same way as interaction between remote stores, as long as global naming of these stores distinguishes between them.
- Since our work will be developed in the context of Persistent Java, issues of heterogeneity are considered in the context of the multi-platform, homogenous language environment of Java.
- In order to support the requirements of applications with a range of models of distributed computation, support is desirable for remote references between stores, querying of stores, copying of objects between stores, and remote method invocation for reading and/or updating.
- Failures of networks, machines, stores and processes are inevitable, so support for detection and reporting of common failures must be supported and an appropriate user model for handling of failure reports will be provided.

2. Global Naming of Persistent Objects

A number of examinations of the issue of scalable naming schemes [21], [7], [6], [11] have concluded that a composite naming scheme is necessary for scalability since it has been found that other types of naming policies used in distributed systems do not ultimately scale. The use of URLs for global naming of resources on the Internet is demonstrably successful, in that a URL can name any resource and the name can then (via a protocol such as HTTP) be mapped to a specific physical node on the Internet. They are also demonstrably scalable since URLs support unique naming of millions of resources throughout the Internet.

Given the current popularity of WWW and the need for storage technologies to support the ever-growing volume of information being made available on the Internet, it seems appropriate to consider the use of some sort of extended URL as a mechanism for accessing persistent stores. This allows us to take advantage of URL standardisation [4] and related work for our design of a global naming scheme for persistent objects and may enable work on provision of persistent store services on the Internet in the future.

A URL naming scheme can be specified which uses composite naming to refer to remote persistent stores and for access of persistent roots, data and code contained in them. This naming scheme uses Persistent Object Locators (POLs) as URLs to contain the physical location of a persistent object that can be globally named and used. The following example conforms to the standard format for a URL, identifies the naming scheme **pol** and specifies an object in a persistent store at a specific physical location.

pol://pjava.dcs.gla.ac.uk/storename.object

Ideally, these POLs should be type-complete[1] to make it possible to reach anything in the store. Although it is not addressed in this paper, it is acknowledged that these POLs must also have a well-defined failure model and future work on them will include specification of their failure semantics.

2.1. Location Transparency

Use of extended URLs support the creation and naming of objects in individual stores independent of any global control. The programmer works in the context of what has been described as a federated persistent store model where they are aware of store boundaries [10]. While the location transparency of a one-world model of distributed stores

1. In the context of PJava this implies type-completeness *with respect to Java*

Proceedings of The Thirtieth Annual Hawwaii International Conference on System Sciences ISBN 0-8186-7862-3/97 $17.00 © 1997 IEEE

1060-3425/97 $10.00 (c) 1997 IEEE

does support a simple programming model by hiding distribution, the resulting system is not very scalable and it is costly to maintain transparency. Exposure of distribution allows for a scalable, efficient system with a large degree of autonomy of sites and knowledgeable handling of remote references and distribution-related errors by applications. Support for this exposure of distribution aids large-scale application implementation.

While exposure of distribution is advocated for the reasons given, we still consider it desirable to support location transparency for cooperation between a limited number of stores in support of smaller, distributed applications where the benefits (such as easy of programming) outweigh the costs of maintaining such transparency. Standardisation of URIs (Uniform Resource Identifiers) has included work on both URLs (Uniform Resource Locators) for specifying the physical location of network resources [4] and on URNs (Uniform Resource Names) for specifying unique location-independent names that require a URN service to map them to one or more URLs or any other network resource or service [24],[1]. In order to support location-transparency, where it is appropriate, we also plan to support a complementary naming scheme intended to conform to the emerging (though seemingly stalled) standard for URNs. A Persistent Object Name (PON) contains a unique location-independent name to be mapped to one or more physical objects e.g.

pon:storename.object
maps to
pol://pjava.dcs.gla.ac.uk/storename.object

If PONs are used, a naming scheme is necessary to assign names to resources and a network-accessible service must be provided to resolve the PON to the location of the object.

2.2. Use of Extended URLs

Extended URLs, in the format of POLs and PONs, can be used for access to persistent roots, data and code in a remote store. They will have a failure model and, subject to access/protection mechanisms, they will allow the user to reach anything in a remote store. A break-down of POLs is described in detail below, followed by an explanation of the distinction between them and PONs.

The extended URL for a POL is composed of a URL type **pol** which indicates that this is a naming scheme for accessing a persistent store. The following scheme-specific-part of the string may contain an optional type name for the object being referenced, so that type-checking may be done against the result of analysis of the rest of the given string. It is followed by a standard domain name for specifying location (as supported by DNS [20]), an object store name, a persistent root name and the object to be reached, perhaps via an object path. All external references made using POLs *must* reference an object of the persistent store via an instance of an object named as a persistent root in that store.

pol--> domain name --> object store -->
persistent root --...--...--> object

The referencing of a range of objects can be done via a POL. The failure semantics assume support for exceptions; it is intended that an exception should be raised when an error is found during the processing of a POL. A POL can reference

- the name of a persistent root, either to return it or as part of the path to an object reachable from it e.g
  pol://dcs.gla.ac.uk/aPJavaStore.
  aPersistentRootName
- the name of a field of an object,
- the index operation into a vector or a dimension of an array,
- the application of a method to the currently reached object: this is equivalent to a remote method invocation e.g
  pol://dcs.gla.ac.uk/theJavaStore.
  aPersistentRootName.objectName.
  methodName
- the name of a class to which the currently reached object should be cast.

2.3. Comparison with PONs

PONs can be distinguished from POLs by the scheme name prefix at the beginning of the URL:

pon--> object store -->
persistent root --...--...--> object

The Persistent Object Name starts with the naming scheme prefix **pon** and contains, the optional typename, a storename and the path to the object. The storename is a location-independent virtual name that can be mapped to a physical location.

3. Object Availability

The extended URL is intended to be used to reference persistent objects which have been made available by one persistent application for use by another persistent application that may reside on a remote node of a distributed system. The management of remote references is an important issue. The implications for a store making an object available for remote referencing and the complexities of remote reference usage are examined

Proceedings of The Thirtieth Annual Hawwaii International Conference
on System Sciences ISBN 0-8186-7862-3/97 $17.00 © 1997 IEEE

1060-3425/97 $10.00 (c) 1997 IEEE

below.

3.1. Referential Integrity

If even a single reference is allowed between one node of a distributed system and another then referential integrity becomes an issue. If a node dereferences a reference to a remote object, it expects that object to exist at the location referred to. The conventional principle of referential integrity requires that if the node containing the object wishes to maintain referential integrity, it is required to maintain the object at the location for at least as long as any other node may have a reference to it. If the node maintaining the object has no way of knowing what other nodes have references to it, it could then be obliged to maintain the object at its location for the lifetime of the system. We propose that the principal of referential integrity can be weakened from all or nothing to provision of referential integrity for a limited period of time. On this basis, the degree of referential integrity to be supported for an application depends on whether it is more important to maintain object availability or persistent store autonomy.

A related issue is that if a large number of remote references are allowed between nodes then the nodes become heavily inter-dependent. A large number of dependencies can build up between stores over time if a lot of cooperation between the stores is permitted. Experience with Distributed PS-Algol [26], for example, has shown that it becomes almost impossible to do any independent management of stores (e.g. for stabilisation, recovery or garbage collection) when they become so inter-related. Remote reference management issues of particular relevance to the support of persistent distributed application systems are support for performing distributed computations and maintenance of long-running systems. The former issue includes support for the execution of distributed transactions; the persistent stores involved in the transaction must provide sufficient guarantees of object availability for the distributed transaction to succeed. This may involve support for locking of objects used in the transaction, permission for copies and/or updates to be made and support for a potentially expensive, distributed commit protocol. The latter issue includes the support of stabilisation (the writing of and updates on persistent objects to stable storage) and support for garbage collection. If a distributed transaction, that modifies persistent objects on more than one node, is to be committed, stabilisation should be performed across the distributed nodes involved; this can only succeed if the remotely-referenced objects are guaranteed to exist at least until the termination of the stabilisation. If a store has made an object available for remote referencing, how can it determine when the object has no references to it from other nodes and so may be garbage-collected? If it cannot, and it wishes to support referential integrity, the store cannot garbage-collect the object and over the lifetime of a long-running system this could prove to be a big space-leak problem.

In order to address the issues described above, a persistent store must adopt a policy on the management of remote references.

1. It can guarantee referential integrity by committing to provide objects for remote referencing for the lifetime of the system, thus losing control over the referencing of the object from other stores.
2. It can refuse to maintain any referential integrity by not allowing any remote referencing of objects in its store; this is impractical since it allows no cooperation between stores and severely limits the scale of persistent applications.
3. It can refuse to maintain any referential integrity by not making any guarantees about object availability; it retains complete control, does no tracking of remote use and the obligation is put on persistent applications using remote references to handle dereferencing failures.
4. It can guarantee referential integrity within well-specified time limits; the store retains control over its objects while still providing some support for cooperation between stores in the distributed system. However, it does no tracking of remote use so the obligation is on remote applications to coordinate use of remote objects within the given timelimits and deal with dereferencing failures.
5. It can guarantee referential integrity within well-specified time limits and commit itself to tracking remote use so it can notify other stores on the loss of referential integrity. This requires the use of data structures tracking references to its objects which are likely to be difficult to scale and maintain consistently.

Since the model of distribution described in this paper is intended to have support for scalable systems with potentially long lifetimes and the flexibility for support of both largely-autonomous stores and cooperation between stores, only options three and four are realistic and option four is seen as the most flexible and feasible for support of a range of distributed persistent application models.

3.2. Specification of Object Availability

Persistent stores can retain a large amount of autonomy if they are allowed to maintain control over object

availability and usage (such as restrictions on object copying). The principle upon which this autonomy is based is that a persistent store should be able to provide services, such as object availability for use by remote applications running over remote stores, with a *lack* of obligation on the service provider to provide that service as long as there are users or even indefinitely. This principle ensures that the availability of data is usually under the control of the provider and allows the provider to withdraw it at its discretion.

Thus, in conformance with the fourth option on referential integrity, as described in section 3.1. above, a timeout can be associated with an object for remote use; after this timeout, the object provider no longer guarantees referential integrity. The use of such a timeout, associated with a referenced object, allows a store to completely control the availability of the object. It can specify, at the granularity of any object in its store, what its provision of referential integrity will be. The store can strictly control what objects are available for remote use by manipulating the value of timeouts. Only a subset of objects contained in the store are likely to be made available for remote use at all and when an object that has been available is found to have a timeout less than or equal to the current time it is no longer available for remote use. Such timeouts can be used by a store to limit the objects involved in distributed transactions and distributed stabilisation of persistent updates; thus controlling the cost of such operations. Timeouts also allow dependencies between stores to be minimised over the time span of system operation, allowing more effective garbage collection of objects; an object that has timed-out and is thus no longer available for remote referencing can be garbage-collected if it also has no local references left.

Support for the specification of timeouts on the usage of object references requires a form of global time to be agreed and maintained across the distributed system. A lot of work has been done on this non-trivial issue [7]. The conclusions to be drawn from it are that, for large-scale distributed systems, we cannot afford fine-grained timeouts. There seems to be no algorithm that maintains clock synchronisation within known bounds over a wide area network in the face of communication failures. Also, it is difficult to estimate the execution time of code when the same code takes different amounts of time on different platforms; especially when that execution time may include failure and restart time, since the persistence model supports recovery of program execution state after some categories of failure. By making timeouts coarse-grained, it is sufficient for us to depend on the degree of accuracy of the Network Time Protocol [19], which is the standard for clock synchronisation throughout the Internet. Further work is required on determining the granularity of timeout (minutes/hours/days) that will be workable.

Given that it is desirable for a store to maintain control over not only its objects' availability but also its usage, in order for it to maintain a large degree of autonomy within a distributed system, further information can be associated with each object that a store makes available that dictates exactly how that object can be used by remote sites. This support for a store's control of objects can be considered as conditions of use. These conditions specify:

1. whether the object is available for use by remote stores,
2. how long the store guarantees that the object will continue to be available,
3. whether the object can be copied by a remote store,
4. whether the object can be changed by a remote store,
5. whether an object method can be invoked,
6. whether the object can be queried.

Mechanisms, probably language-based, will be needed to enforce these conditions of use. For example, if the default is to protect objects in a store from remote use, it will be necessary to make an object reachable from a persistent root which has been specified as visible to other stores in order to make it accessible. Future work will consider what protection mechanisms [23] are appropriate for our distribution model. Protection may be provided by encapsulating conditions of use in capabilities and/or associating access lists with objects made available for remote use.

During operation of a distributed persistent application, it should be possible for the application processes running on one node to dynamically reconfigure the conditions of the objects in the store over which it runs. It should also be possible for an application process running on one node to query another node to find out what objects are available and their conditions of use.

4. Remote Usage

Having considered issues on object availability and the use of timeouts, the issues raised by provision of the other listed conditions are now examined. The conditions on usage of objects that can be remotely referenced must have clear semantics. An application programmer must have a clear understanding of the implications of each of the conditions that they place on an object for remote use and must know how they can use an object made available remotely, given specified conditions of use. Before examining the semantics of specific object conditions, the issues of copying remote objects between persistent stores and querying stores are considered in general.

Proceedings of The Thirtieth Annual Hawwaii International Conference on System Sciences ISBN 0-8186-7862-3/97 $17.00 © 1997 IEEE

Deep copying of an object from one store to another, including all the objects it references, the objects its references reference and so on, will result in the copying of the transitive closure of the object from one store to another. When an object's transitive closure is large, this copying can be expensive in terms of bandwidth when communicating it between stores, it may delay computation while the copying is taking place and there are problems of placing it into the destination store [10] which can result in multiple copies of the same object ending up in the destination store unintentionally. In the most extreme case, if the copied object is the main persistent root of the store, or references the persistent root(s) of the store, the whole store could end up being copied in one go.

Where the copying or transfer of large amounts of data between stores is required, as would be necessary for deep copying of a large transitive closure of objects, it is desirable to have specific support for communicating the data in bulk in order to optimise cooperation. Good support for such bulk data transfer requires efficient algorithms for the marshalling/serialisation of data, use of compression where appropriate to minimise bandwidth and consideration of batching techniques for cutting down the network traffic between stores.

The alternative to deep copying of objects is to limit the depth of copying of an object between stores [12],[8]. The conditions associated with objects in our model of distribution are intended to support the limitation of copying of an object between stores. Only objects which are available for remote use and for which copying is permitted may be copied, either directly or because they are referenced from another copied object. Thus it is within the application programmer's control to limit the depth of copying of objects. For example, in a store containing a set of standard objects plus user-defined objects, it would be appropriate to only allow user-defined objects to be copied.

Support of copying objects between stores raises the issue of potential loss of referential transparency [10],[8]. If two objects reference a common sub-graph, independent copying of these two objects can result in the unintended replication of the subgraph. Further work on the semantics of conditions on copying objects will need to address this issue.

Since persistent application systems are likely to be long-lived and large-scale, their stores are likely to contain complex objects, large objects and large collections of objects. It seems desirable to be able to support querying of data as well as communication of this scale of data between stores, as results of queries, as parameters to remote method invocation or on the dereferencing of large and/or complex objects. It may be possible to extend the use of eURLs to add querying over specified stores or objects. While persistent programming languages have not so far proved to be able to support querying mechanisms in a way which is comparable to more traditional database systems, there may be some benefit in doing further investigation into the integration and support of a querying language like CPL (Collection Programming Language) which has been successfully used to extend the query language Kliesli to support the querying and transformation of complex scientific data in the Human Genome Project, as described in [5]. CPL can be used to specify the same queries as SQL, *plus* queries on complex objects, including objects that themselves contain sets of objects, and collection types including lists, bags, arrays and indexed structures.

4.1. Remote Copying

If the store containing the object requires this object to be the only copy of this object in the whole system, while allowing for invocation of methods to provide access to its attributes, then the object can be made available but cannot be copied. (If the programmer then chooses to allow copies of the whole object to be returned by one of its methods then this is obviously a complete contradiction of the copying restriction on the object and the association of this condition with an object providing such a method should be considered a programmer error!) If the store permits copying of an object, this implies that any application with a remote reference to the object can obtain its own copy of the object which it may put in its own store.

4.2. Remote Update

If the store containing the object makes it available for use and allows the object to be copied but not updated then it is essentially allowing read-only copies of the object to be made by remote stores. A store may wish to do this because it contains the original and wants to retain the rights to change the object itself or because it contains only a copy of the object itself and does not have the right to grant copying and updating of the object to other stores. If the store permits updating of an object then it is either not concerned about maintaining one original copy of the object and evolution of the object on different stores is permitted or there are consistency protocols for maintaining consistency between updatable copies of an object in a number of stores (a potentially costly protocol).

Proceedings of The Thirtieth Annual Hawwaii International Conference
on System Sciences ISBN 0-8186-7862-3/97 $17.00 © 1997 IEEE

4.3. Remote Method Invocation

If the store containing an object makes that object available but does not allow remote invocation of methods on that object, this may be because remote invocation does not make sense in the context; it may still be possible for public attributes of the object to be examined remotely or even copied. Permitting remote invocation of object methods implies that there is support for such functionality at the store containing the object. The invocation of the remote method may require that objects are copied from the store of the invoking application as parameters; in this case the parameter objects must have appropriate conditions associated with them that make them available for copying to remote stores. Remote method invocation supports cooperation between stores where this is desirable, while limits on remote method invocation aid store autonomy.

The exact granularity of permissions for remote method invocation is an issue for debate. The permission given for remote method invocation on a specific object could apply to all public methods of that object. This could be considered a reasonable interpretation on the grounds that support for specifying exactly which public methods of an object can be invoked remotely is too heavyweight. This issue is also under consideration by those working on access control for a Java Object system called JaDE [16].

4.4. Remote Queries

If the store containing an object makes it available but does not have to allow queries to be run against it, this implies that any queries run on a remote store are only applicable to those objects for which querying has been approved. Where querying has been approved, a query from a remote application can be run against the available objects of a store and a collection of objects matching the query can be copied and returned.

5. Using Object References and Conditions

An implementation of the model of distribution described in this paper will support the API presented below. The API should be considered an extension of the PJavaStore API [3]. It provides the methods necessary for a persistent application to set up and use references to objects in remote stores.

In order to support this API effectively, it will be necessary to have clear semantics for failure. One of the high costs of a distributed system is the network traffic between nodes when one node detects a failure and informs the affected clients only to have the clients send it a number of messages across the network in order to find out more about the failure so they can handle it correctly. It should be possible for a node, on detection of a failure to perform some local analysis of the problem, since it is close to the source of the error, generate a number of informative error messages based on that analysis and send a batch of these error reports to affected clients in one bulk error report. The client is then less likely to have to generate further network traffic on error analysis. Thus, future work will include the design of a series of exception-handlers for the catching and analysis of predictable errors and the development of bulk error reporting techniques, where possible, to complement the API below.

5.1. Making Objects Available

An application wishing to make objects in its store available for remote use must first set the appropriate conditions for use of those objects

public final boolean setConditions
(boolean available, UniversalTime timeLimit,
boolean copy, boolean update, boolean invoke,
boolean query);

This method sets the conditions to the parameters given:

available True if the object can be remotely referenced, false otherwise. If this is set to false, the other parameters of the method are ignored, since they become irrelevant in this case.

timeLimit Availability of the object is guaranteed only until the specified time limit.

copy True if the object can be copied to a remote store, false otherwise.

update True if the object can be updated by a remote store, false otherwise.

invoke True if public methods of the object can be invoked by a remote application, false otherwise.

query True if a query can be run over the object, false otherwise.

The method returns True if the conditions on the object are valid conditions and have been associated with the object. False is returned if given conditions are invalid or conflicting.

5.2. Reference Generation

An application needs be able to generate appropriate extended URLs for the remote objects it wishes to reference. This involves querying a remote store for its available objects.

public final ObjectInfo[]
getStoreAvailability(URL store);

Proceedings of The Thirtieth Annual Hawwaii International Conference
on System Sciences ISBN 0-8186-7862-3/97 $17.00 © 1997 IEEE

public final ObjectInfo
getObjectAvailability(URL obj);

The method getStoreAvailability, given the global name (extended URL) for a persistent store, returns an array of ObjectInfo elements, which represents an index of objects through which all the objects available for remote use from that store are reachable. An appropriate model of use should ensure that this index is limited to a reasonable size so that the array passed to the remote site is not too large to be communicated efficiently. Future work on this issue will focus on how to provide sufficient information to support remote access to very large stores.

The method getObjectAvailability, given the extended URL for an object, returns the ObjectInfo object containing the conditions of use associated with it. This method can be used to find out the conditions currently set on a specified object.

The onus is on the application to obtain information on what objects are available, what their conditions of use are and to handle errors when objects are not available. The inter-store reference in the application program may be represented as an indirection object/proxy containing the location of the object as an extended URL for use when it does need to be dereferenced.

5.3. Requests for Availability

Once a persistent store has made objects available for remote use, remote stores can make requests on this availability. Examples of such methods and their semantics are included below to illustrate what sort of requests can be made.

public final UniversalTime
retainPlease(URL ref, UniversalTime ut);

The method retainPlease is available for use by a remote persistent application to allow it to request that the object referenced by ref should not be made available for garbage collection until after the specified UniversalTime ut. The URL of the remote reference or application may also be supplied for analysis purposes.

It is up to the local persistent application servicing the retainPlease request to decide whether it wishes to comply. If the specified time has already past or is now then the current time is returned. If the specified time is in the future but the local application does not currently wish to guarantee that it will retain the object until the requested time, then the time until which the local program is willing to retain it is returned.

public final UniversalTime
expectedDeath(URL ref);

The method expectedDeath is provided for the remote application to make a request to find out when the object referenced by ref is currently expected to be made available for garbage collection.

6. Comparison with Related Work

The intended use of extended URLs and object conditions described in this paper is similar to features of a number of existing systems but it can most easily be compared to the use of Handles for digital objects in ongoing work on digital information systems such as digital libraries [18]. Just as our objects have global names and conditions of use, a digital object is composed of data and an associated properties record, which contains a unique identifier called a Handle and, optionally, conditions of use for the data. Digital objects are stored in repositories and access to them is controlled by imposing the terms and conditions associated with either the object itself and/or the repository. A properties record can contain information including whether an object is currently accessible, whether the object can be replicated, changed or removed and the date and time the object was deposited in the repository. Where one object contains references to other constituent objects, a union is done on the conditions of the object and each of its constituent objects in order to determine the conditions for the constituent objects in that context. Our implementation would also have to consider the combination of an object's conditions with those of the objects that it contains.

A site wishing to use digital objects downloads a handle directory, containing the mappings of handles to network resources, from a handle directory server at a well-known location. Handles can then be used as references to digital objects. However, no guarantee is made about the availability of the object; the handle directory entry can be considered as the last known location of the object only. The data and conditions of a digital object can evolve with the exception of key properties of an object, such as the handle, and objects it refers to, unless the originals are also changed.

The use of timeouts on remote references can be compared to the use of leases [13], which have been used for the maintenance of file cache coherency in the V system. It has been found that short leases on cached files are usable and fault-tolerant. However, synchronised physical clocks with bounded drift were assumed. We consider this assumption to be unrealistic in large-scale systems which is why we have chosen to support timeouts at a coarse grain that should be rarely affected by the clock drift of NTP.

Proceedings of The Thirtieth Annual Hawwaii International Conference
on System Sciences ISBN 0-8186-7862-3/97 $17.00 © 1997 IEEE

1060-3425/97 $10.00 (c) 1997 IEEE

However, a provision of fault-tolerance that is workable with our model of timeouts does need further consideration.

7. A Selection of Supporting Technologies

An implementation of the model of distribution described in this paper should support remote references, copying of objects between stores and remote method invocation, preferably in a sufficiently flexible way to allow conformance to any sensible combination of object conditions as described in section 3. above. In the context of an implementation for support of PJava applications such as Forest [17], existing and developing technologies will need to be evaluated to consider whether they can provide the required support.

While it is not our intention to present a thorough evaluation of a range of supporting technologies for our distribution model in this paper, several existing and developing technologies are considered here to give some indication of which ones it might be appropriate to incorporate in PJava.

7.1. Support for Object Copying and Remote Method Invocation

The currently alpha release of Remote Method Invocation (RMI) from JavaSoft is an obvious contender for support of remote method invocation in a Java-based system. As the documentation for JavaSoft's RMI implementation [15],[28] points out, other software does exist and could be used for the support of remote method invocation on Java objects. For example, inter-operability between the WWW and distributed object systems using CORBA has been implemented as described in [22]. However, systems such as CORBA are not well-integrated with the Java language in comparison with Java RMI; CORBA presumes a heterogeneous, multi-language environment. This means that it is more heavy-weight to use since it has a language-neutral object model that would require translation of objects between the CORBA model and Java. The benefit of the Java Virtual Machine is that it allows Java RMI to operate in a homogeneous environment since it supports the one Java object model on an increasingly wide range of platforms.

Support for the preparation of objects for copying, as is necessary during remote method invocation for example, is provided for Java through use of Object Serialisation [14]. Implementation of Object Serialisation supports the marshalling and unmarshalling of any Java Object. This involves a serialisation of the specified object, including following all references in that object and serialisation of the referenced objects until the whole transitive closure/ graph of the object has been serialised. Serialisation of an object includes the encoding of the object's class and all its super-classes. Since by default, static and transient fields are not serialised, the setting of chosen references to transient prevents the serialisation and copying of the transitive closure of an object when this is not necessary or desirable. However, this is an ad-hoc solution to the problem of limiting the depth of copying of an object between stores. A more thorough treatment of the issue can be found in work on Octopus [12] and object migration by substitution [8]. The Octopus mechanism supports the cutting of bindings within the closure of an object to be copied, and the rewiring of the partially-copied object in another context. The use of such a mechanism would enable the support of our more sophisticated requirements on object copying.

Java RMI supports the passing of local Java objects of any type by copying where they are local arguments to and local results from a remote method invocation; remote Java objects are passed by reference. An implementation of RMI for our distribution model would need to respect conditions on object availability for object passing. Remote object references have URL-based names similar to our model for global naming so, for example, a remote object representing a bank account can have the URL ``rmi://java.Sun.COM/ account''. This name is mapped to the physical location of a particular host and port by an RMI Registry. Just as with our intended use of eURLs, the relaxed location transparency and support of a failure model requires programmers to be aware of when remote objects are being invoked in order for clients to deal with the exceptions, additional to normal method invocation, that can occur during a remote method invocation.

With the above features, it can be seen that Object Serialisation and RMI could provide a reasonable basis for the required distribution support in PJava.

8. Conclusion

A model of distribution has been proposed that supports both the autonomy of persistent stores and cooperation between them. Extended URLs can be used to globally name any object in any persistent store of a distributed, persistent system. Conditions associated with objects allow the store to control access to and usage of persistent objects. Support for dynamic modification of object conditions allows a store to adapt these conditions to provide the required degree of store autonomy at any time during the lifetime of the system; thus, object availability can be strictly controlled to minimise the cost of distributed

operations for stabilisation, recovery and garbage collection.

The aim of this paper has been to introduce the main concepts of a model of distribution for a range of distributed, persistent applications. Some justification has been provided for our design decisions and a few indications for implementation of these ideas have been given. It is our intention to explore this proposed model further in the future and to consider practical implementation strategies in detail.

9. References

[1] W. Arms, L. Daigle, R. Daniel, D. LaLiberte, M. Mealling, K. Moore, and S. Weibel. Uniform resource names: A progress report. D-Lib Magazine, February 1996.

[2]M. Atkinson. Personal communication on experiences with dpsalgol, 1996.

[3]M. Atkinson, M. Jordan, L. Dayn{\`e}s, 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.

[4]T. Berners-Lee, L. Masinter, and M. McCahill. Uniform resource locators (url). RFC 1738, December 1994.

[5]P. Buneman, S. B. Davidson, K. Hart, C. Overton, and L. Wong. A data transformation system for biological data sources. In Proceedings of the Twenty-first International Conference on Very Large Databases, Zurich, Switzerland, 1995.

[6]S. Ceri and G. Pelagatti. Distributed Databases: Principles and Systems}. McGraw-Hill, 1984.

[7]G. Coulouris, J. Dollimore, and T. Kindberg. Distributed Systems: Concepts and Design. Addison-Wesley, 1994.

[8]M. Mira da Silva and M. Atkinson. Higher-order distributed computation over autonomous persistent stores. In Proceedings of The Seventh International Workshop on Persistent Object Systems, Cape May, New Jersey, USA, May 1996.

[9]M. Mira da Silva, M. Atkinson, and A. Black. Semantics for parameter passing in a type-complete persistent rpc. In Proceedings of The 16th International Conference on Distributed Computing System, Hong Kong, May 1996.

[10]A. Dearle, J. Rosenberg, and F. Vaughan. A remote execution mechanism for distributed homogeneous stable stores. In Proceedings of the Third International Workshop on Database Programming Languages: Object Models and Languages, pages 125--138, Nafplion, Greece, 1991. Morgan Kaufmann.

[11]R. di Bona. Data Movement in the Grasshopper Operating System}. PhD thesis, University of Sydney, March 1995.

[12]A. Farkas and A. Dearle. Octopus: A reflective mechanism for object manipulation. In C. Beeri, A. Ohori, and D.E. Shasha, editors, Proceedings of the Fourth International Workshop on Database Programming Languages: Object Models and Languages, pages 50--64, Manhattan, New York City, USA, 1993. Springer-Verlag(Berlin).

[13]C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. SIGOPS Proceedings of the Twelfth ACM Symposium on Operating Systems Principles, 23(5):202--210, 1989.

[14]Sun Microsystems Inc. Java object serialisation specification, draft revision 0.9. http://chatsubo.javasoft.com/current/doc/rmi-spec/rmiTOC.doc.html, 1996.

[15]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.

[16]V. Jones and M. Winslett. Jade: Access control in a java-based object database. In Proceedings of OOSLA '95 Workshop on Object Database Behavior, Benchmarks, and Performance, Austin, Texas, October 1995.

[17]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.

[18]R. Kahn and R. Wilensky. A framework for distributed digital object services. Technical Report {cnri.dlib/tn95-01}, CNRI, May 1995.

[19]D.L. Mills. Internet time synchronisation: the network time protocol. IEEE Transactions on Communication, 39(10):1482--93, 1991.

[20]P. Mockapetris. Domain names - concepts and facilities. Technical Report {rfc1034}, Internet Network Information Center, 1987.

[21]R. Needham. Names, pages 315--327. Addison-Wesley, second edition, 1993.

[22]O. Rees, N. Edwards, M. Madsen, M. Beasley, and A. McClenaghan. Ansaweb: A web of distributed objects. In Proceedings of The Fourth World Wide Web Conference, Boston, US, December 1995.

[23]J. Rosenberg and J.L. Keedy, editors. Security and Persistence. Proceedings of the International Workshop on Computer Architectures to Support Security and Persistence of Information (Bremen, West Germany, 8--11 May 1990), Workshops in Computing. Springer-Verlag, 1990.

[24]K. Sollins and L. Masinter. Functional requirements for uniform resource names. RFC 1737, December 1994.

[25]R. van Renesse, K. Birman, and S. Maffeis. Horus, a flexible group communication system. Communications of the ACM}, April 1996.

[26]F. Wai. Distributed Concurrent Persistent Programming Languages: An Experimental Design and Implementation. PhD thesis, University of Glasgow, April 1988.

[27]J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A note on distributed computing. Technical Report {SMLI TR-94-29}, Sun Microsystems Laboratories Inc., November 1994.

[28]A. Wollrath, R. Riggs, and J. Waldo. A distributed object model for the java(tm) system. In USENIX Second Conference on Object-Oriented Technologies and Systems (COOTS)}, Ontario, Canada, June 1996.