

# Porting a Distributed System to PJama: Orthogonal Persistence for Java<sup>TM</sup>

Huw Evans and Susan Spence  
The RAPIDS Group  
{huw | susan}@dcs.gla.ac.uk  
<http://www.dcs.gla.ac.uk/~huw> | [~susan](http://www.dcs.gla.ac.uk/~susan) | rapids}

Department of Computing Science  
The University of Glasgow  
Glasgow, G12 8RZ, UK

## Abstract.

This paper describes the experience of porting a 25,000 line distributed Java [GJS97] program to version 0.3.5.1 of the orthogonally persistent programming environment PJama. The port was performed in two stages. The first stage, changing the source code, took two days. The second stage, ensuring the ported program exhibited a suitable run-time behaviour, took one week. This paper discusses the main changes that were required to move from a distributed system with support for persistence via serialization to one that supported an implementation of orthogonal persistence with support for distributed programming.

## 1 Introduction

This paper describes the author's experience of porting 25,000 lines of Java, spread over 188 classes, to PJama version 0.3.5.1 [ADJ<sup>+</sup>96] during March 1998.

The program is part of the DRASTIC [ED97] platform which provides support for the run-time evolution of a distributed software system. The platform was originally built using Java and ported to PJama to make use of orthogonal persistence. Porting the code to PJama turned out to involve more than just running the original code in a new environment with hooks into the underlying per-

sistence mechanism.

This paper is divided into seven main sections. Section 2 describes those features of DRASTIC's architecture and implementation which are pertinent to the port. Section 3 justifies why DRASTIC requires persistence and why it already had a limited form of persistence provided by object serialization. Section 4 describes how the programmer selects between object-serialization-based and PJama-based persistence and details the changes required to use PJama. Section 5 contains a detailed description of how porting to PJama affected the use of RMI within DRASTIC. Section 6 describes the use of class loaders within a PJama persistence context. Section 7 details how the expectations of the first author led to code that crashed the interpreter and section 8 gives some very brief performance figures for reading from an ad-hoc store based on object serialization for comparison with PJama. Section 9 offers some conclusions.

## 2 Features of DRASTIC

Section 2.1 describes the main features of DRASTIC that were pertinent to the port. Section 2.2 very briefly describes the components that make up the DRASTIC run-time system to show how these main features are used.

The DRASTIC platform is implemented using

version 1.1.5 of Sun's JDK.

## 2.1 Main Features

DRASTIC is a distributed, run-time platform that supports the evolution of code at program execution time. Programmers write their application code to run on top of this platform. The application is divided into identifiable collections of components called zones and, for the purposes of code evolution, the contents of one such zone can be replaced at run-time without having to replace others. Pair-wise contracts shared between two zones are used to enforce the evolution of types by describing whether these zones may legally exchange objects of given types<sup>1</sup> and whether objects of certain types may migrate between these two zones. A DRASTIC supported distributed application does not need to be shut-down entirely to change some part of it. Only a single zone needs to be suspended and the processes within it terminated if they need to be evolved. For a description of the architecture and platform see [ED97]. For a discussion of how evolution is performed using it see [ED98].

Before the port was performed, DRASTIC already contained file-based support for persistence via Java's object serialization mechanism. This form is not orthogonal to type as the programmer has to indicate that an object can be made persistent by having its class implement `java.io.Serializable`. At process run-time the programmer indicates whether they require their process to use this kind of persistence. If not, the state of the process is completely transient.

Support for distribution is provided by Java RMI. Each process that boots over a DRASTIC platform is provided with a remotely invokable object (the `inhandler`) which is used to contact the evolution technology in the platform and to communicate with application level objects running on top of it.

---

<sup>1</sup>represented as Java classes.

A Java-level object that the application programmer wishes to be either remotely invokable across a DRASTIC platform or migratable within it, must have the source code of the class passed through the DRASTIC pre-processor. The collection of classes that are generated contains one class that replaces the original so that application-level object references are redirected through the DRASTIC platform. This class is instantiated to create a proxy object that refers to another object which is instantiated from a renamed version of the original class. The proxy class implements a super-set of the original interface, to support calls for object migration. The DRASTIC preprocessor also generates other code so that such references can be used in a remote method invocation, supported by RMI. The preprocessor also adds the code to implement the `java.io.Serializable` interface to all necessary classes to allow objects instantiated from them to persist in files via the object-serialization mechanism.

The DRASTIC platform uses its own classloader to load classes from other processes. This is so that the platform is not reliant on technology such as NFS to make classes available to a process from a remote machine. Zone contracts contain information on which processes have access to which application-level classes. This information is given to the classloader when a process is booted over a DRASTIC platform. A process can load a class from any other process in the same zone. This restriction is imposed because loading classes across a zone boundary would make them subject to evolution. Considering the problem of evolving class bytecodes was not the focus of the DRASTIC project. Section 6 describes how the port affected the use of DRASTIC's classloader.

The DRASTIC platform contains a simple form of fault-tolerant invocation. If a reference to a remote object is invalid, a daemon process is contacted for a refreshed version. If the reference to the object in the daemon process is broken, a different daemon process is contacted for a reference to the daemon process' object. This escalation of responsibility is carried out until a top-level daemon is reached.

If this escalation process is not successful, human intervention is required. If a queried daemon is available and it contains a valid reference, each contacted daemon is given a refreshed reference and eventually a working reference to the remote object is supplied. The DRASTIC platform ensures that new, valid object references are registered with the relevant daemon processes, so that eventually a refreshed reference will be registered. Section 5.2.2 discusses how the port affected the use of DRASTIC's fault-tolerant invocation mechanism.

## 2.2 DRASTIC System Components

Figure 1 shows all the components of a DRASTIC system. This section briefly describes their various roles.

The `PASManager`<sup>2</sup> contains a description of the current application in terms of the zones it is decomposed into and the contracts that define the inter-zone communication. The `EvolverMgr` is used when the types in one zone need to be updated. If a type is updated in one zone that is visible from another, the contracts between those zones may need amending. If this is the case, the `EvolverMgr` is used to install the new contracts in the zone that has initiated the evolution. The `Registry` is a replacement for the Java RMI registry that associates a simple `String` based name with a reference to a remotely invocable object. The Java RMI registry forces the programmer to have one registry on each host that makes available a remotely invocable object. The DRASTIC `Registry` relaxes this requirement so that there only needs to be one registry in the whole system. Having only one `Registry` is adequate for the purposes of DRASTIC and it simplifies run-time name management.

Each zone contains a single `ZSPMDaemon`<sup>3</sup>. This daemon is used by an application process when it is booted over a DRASTIC platform. The process

registers itself with the daemon under a programmer selected name. This name is used by other processes in the same zone so that they can contact it.

Each zone also contains a number of `ZBP`<sup>4</sup> processes. These processes embody and enforce the contract currently in use between any two zones. For example, in figure 1, zones A and C have a contract and so zone A has a ZBP to represent its half of the A and C contract and zone C has an equivalent process on its side. All inter-zone references are redirected via the relevant ZBP processes to ensure the contract is enforced.

When an evolution takes place, the zone that is to be evolved is informed by the `EvolverMgr` that it should suspend all incoming and outgoing activity through its ZBP processes. Processes that contain code that is to be evolved are terminated cleanly by running application-programmer defined methods. Objects are inserted at the ZBPs to ensure a change inside the zone is not visible outside it. Using the pair-wise contract, the programmer specifies the objects to be installed into a particular ZBP. The DRASTIC platform uses this contract to add the objects to the necessary inter-zone references to preserve pre-evolution object interfaces. Application-level processes are then restarted using any evolved code the programmer has provided.

DRASTIC was designed from the start to make use of a persistent programming language. If a process is running over a persistent store, it is assumed that only one process may do so at any one time. If a persistent process is terminated to perform an evolution to its code, it is also assumed that technology exists to evolve the contents of the store before rebooting the new process over it.

In the figure, user process 1, in zone A, contains an object of type N that holds a logical reference to an object of type M in user process 2, which is held inside zone C. This reference is redirected via the DRASTIC platform in both processes and through

---

<sup>2</sup>Persistent Application System Manager.

<sup>3</sup>Zone Specific Process Manager Daemon.

<sup>4</sup>Zone Boundary Process.

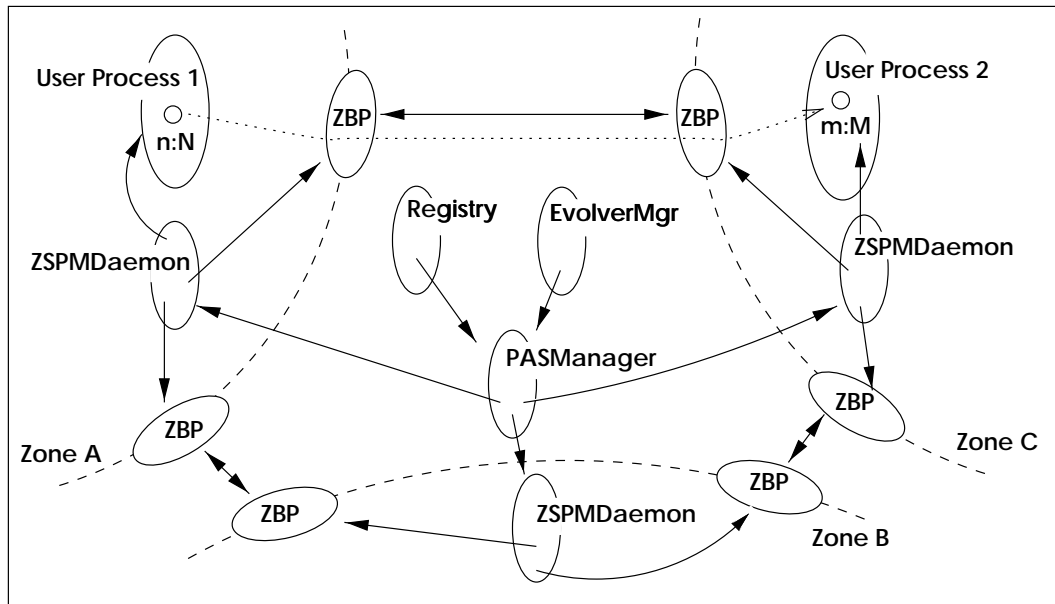


Fig. 1: A System Wide View of an Application Running over DRASTIC

the ZBP processes in each zone. Thus, the ZBPs can enforce the currently active contract whenever methods on the remote object are invoked or if an object of either type M or N is migrated. The types N and M are not related in the sense one has evolved from the other. Object n just has a reference to object m which is in the other zone.

### 3 Support for Persistence

Large, long-lived systems typically have a requirement that their data outlive the process that created them. DRASTIC is concerned with the run-time evolution of such systems and it therefore follows that DRASTIC should include support for persistence. DRASTIC does not provide the technology to perform the evolution of a persistent store, this is assumed to exist, e.g. [Dmi98]. However, it does support the programmer in being able to describe and manage the run-time evolution of a distributed system.

The DRASTIC project always intended to make use of an orthogonally persistent language. This is because it is felt to be the easiest abstraction to use. The persistent language implementation hides complexities such as tracking object movement between main memory and stable storage. Minimal complexity is exposed via a small number of APIs that allow the programmer to integrate their code with the persistence technology. Persistence of application-level objects is achieved by reachability from specified roots of persistence and programming by reachability is natural in an object-oriented language.

At the start of the DRASTIC project an orthogonally persistent Java was not available. Sun's Java Development Kit (JDK) was therefore used to build the basic platform with the intention of porting to a persistent environment at a later stage.

Before such an implementation became available, the platform was extended to make use of Java's object-serialization mechanism. This was provided so that parts of the platform's state could be

retained between executions, to allow for experimentation. It was not made available to the application programmer as it is not considered a viable long-term persistence solution. See section 4.1 for more detail on the object-serialization-based persistence mechanism.

Before the port was performed, it was decided that all processes should be able to make use of persistence, except for the `ZSPMDaemons` and the `Registry`. This is because these two processes contained named reference bindings. It was considered easier to have a process re-register with a new instance of the daemon and rely on the fault tolerant invocation protocol than manage a daemon with persistent name bindings. If the daemon is shut down because the system is being evolved, it is possible that the names bound to a reference will change. Allowing processes to simply re-register with new names was considered a straight-forward solution to the problem.

An application-level DRASTIC process has a root of persistence containing one object, the `inhandler`. The transitive closure of this object is the state associated with the technology necessary to perform run-time software evolution and remote method invocations across such a system.

A ZBP process is responsible for enforcing the contract currently defined between two zones. A ZBP process contains two `inhandlers`, one is used for communication with processes in its zone, the other is used for communication with the ZBP in the other zone. When a ZBP is booted, it requires a persistent root with references to its two `inhandlers`.

## 4 Selecting the Persistence Model

Section 4.1 describes the kind of persistence support DRASTIC provided before the port together with its implementation. Section 4.2 describes how the platform was changed to exploit PJama

based persistence.

### 4.1 Original Persistence Support

Before the DRASTIC platform was ported to PJama it already contained support for file-based persistence by reachability using Java's object serialization mechanism. When a DRASTIC process is started, a flag is used to indicate the type of persistence required. Before the port, this flag could either be `ad-hoc` or `no_persistence`. If `ad-hoc`<sup>5</sup> is specified, the platform uses the object serialization mechanism, while `no_persistence` indicates that no persistence should be provided. An optional flag can also be given to indicate the store's filename. If this flag is not provided, the platform creates a name based on a combination of the persistent application, zone and process names. Processing the flags and initialising the system with the required persistence mechanism is done by the Persistence Module.

The object-serialization mechanism works by reachability, starting at a root, each object in the transitive closure is converted into a form suitable for writing to a file or sending across a network. Enough information is embedded within the stream so that a graph of the same shape can be constructed when the stream is deserialized. When a process is started with `ad-hoc` persistence selected, the Persistence Module first checks whether a store already exists. If it does, it reads from the file, passing back the persistent root to the DRASTIC platform. If it does not exist, a new root is created and passed back. The root for the `ad-hoc` persistence is the DRASTIC platform's `inhandler` object.

The `ad-hoc` persistence implementation is contained in a thread that periodically serializes a given root into one of two files to give some simple

---

<sup>5</sup>It is referred to as `ad-hoc` because persistence via object-serialization is `ad-hoc` with respect to PJama's orthogonal approach, in particular, object identity is not preserved for overlapping sub-graphs.

protection against store corruption. If a store being read is found to be corrupted, the other is used. If this is also corrupt, a new store is created. When writing the store, the persistence thread sleeps for one second<sup>6</sup> and the root is once again serialized. The serialization mechanism will abort the reading or writing of a store if an error is detected, such as not being able to find a class when performing a deserialization. This default behaviour is imposed by the serialization mechanism and it can only be overridden if the `java.io.Externalizable` interface is used. This interface requires the programmer to take complete control of the serialization mechanism including defining the layout of the streams used to read and write the data. Therefore, DRASTIC uses the default mechanism and makes a best-effort to ensure an abort does not take place.

Using the object-serialization mechanism has a number of problems: reads and writes are very slow (see section 8); the entire store has to be read before it can be used and it has to be completely written before the serialization thread sleeps, allowing another thread to proceed<sup>7</sup>. If an incoming remote-method invocation is received while the store is being written, it will be delayed until the write is completed and the serialization thread is suspended; as a process now takes considerably longer to boot when using ad-hoc persistence, the sequence of events at system boot time can be different. This can cause previously true assumptions about the ordering of events to be false, such as a process registering itself with a daemon so that another may contact it. The waiting process may have to contact the daemon several times before the required process has registered itself. This problem was encountered when ad-hoc persistence was added to the DRASTIC platform. It made programming the boot phase more complicated and error prone and the change in the runtime behaviour is something that the application-level programmer would have to be aware of.

---

<sup>6</sup>This value can be changed at run-time.

<sup>7</sup>It is assumed all threads are running at the same priority.

See [Eva98] for more information on why object serialization in Java is not considered a viable long-term persistence solution.

## 4.2 PJama-based Persistence Support

After the port was performed, the platform was extended to allow the programmer to boot a process over a PJama supported store. This required changes in three areas: the Persistence Module was enhanced; a new class was added, called `drastic.kernel.persistence.pjama.PJamaPersistence`, and a script was provided to run the correct Java interpreter.

The new class, `PJamaPersistence`, was added to the Persistence Module. This class integrates DRASTIC with the underlying PJama-based persistence. It allows a DRASTIC platform to: create a PJama store; register and retrieve a named root of persistence; stabilise the current store and check whether a named store exists. This level of support is provided to abstract over the particular persistence technology being used. The same interface is also used to control the use of the ad-hoc persistence. `PJamaPersistence` is only 83 lines long.

The Persistence Module was enhanced to include PJama-based persistence and several new flags were defined. These let the programmer tell the platform that PJama should be used and, optionally, the name of the store. The Persistence Module was also extended to handle the case of booting a ZBP which requires the persistent root to have references to the two `inhandlers`, one for communication within the zone, the other for communication with the corresponding ZBP in the other zone.

A Unix-shell script was written to run the PJama interpreter if PJama persistence was selected and the Sun Java interpreter otherwise. This was not strictly necessary as the PJama interpreter can run code that does not make use of any persistence fa-

cilities. It was done so that the development of DRASTIC could proceed independently of PJama.

As DRASTIC already contained support for ad-hoc persistence, adding the necessary code to integrate the Persistence Module with PJama took longer than it would have if only PJama based persistence was being considered because the effect on the ad-hoc persistence support had to be checked. Therefore, two working days to perform the first stage of the port is not representative of the amount of work required to port to PJama. The strictly PJama related work of integrating with its technology took half a working day to complete and test.

## 5 RMI and PJRMI

This section describes how the port changed the use of remote-method invocation (RMI) within the platform. This is done by first introducing RMI and its use in the platform in section 5.1 and then in section 5.2 considering the changes that had to be made to the platform after porting to PJama.

### 5.1 RMI and DRASTIC

RMI allows an object in one virtual machine to invoke the methods of another object in a different virtual machine. The virtual machines can be running on the same host, although, more commonly, they are running on different machines and communication across a network is required. Method parameters and results are passed between the two objects using pass-by-value, unless a reference to a remotely-invokable object is being passed, in which case, pass-by-reference is used. Primitive types are copied.

An RMI compiler is used to generate stub and skeleton classes that are used at the client and server side respectively, to copy parameters and results between the objects and to handle the presentation of remote exceptions to the client.

A Registry is provided so that references to remote objects may be associated with simple string-based names. This allows other objects to obtain a copy of a reference to a remote object by contacting the registry, which is at a well-known location, giving it the name used to register the reference. The reference that is passed to the requesting object is actually a reference to an object instantiated from the stub class. The Java Registry requires that the Registry and the remotely invocable object being made available are running on the same host. This is enforced by the Registry when a reference is bound to a name. If this is not the case, an exception is raised and passed to the client. This was felt to be too restrictive for DRASTIC so the Registry was reimplemented to remove this requirement.

When a remote method is invoked, the client side code must handle a **java.rmi.RemoteException**. This is because the remote method may fail in ways that are not possible when invoking a local method; for example, the remote host may crash. DRASTIC builds on this to provide the simple fault-tolerant invocation protocol described in section 2.1.

A process booted over DRASTIC is provided with one remotely invocable object. This object, the `inhandler`, is used to contact the evolution technology in the process and it is used when an application-level object is invoked. The `inhandler` is passed all the information necessary from the calling process to invoke a method at the application-level, such as the name of the method and the name of the class the method is being invoked on. Before the port was performed, a system running on top of DRASTIC could be booted over a combination of transient and ad-hoc persistent stores.

### 5.2 PJRMI and DRASTIC

PJRMI [Spe98] is the name given to the model and implementation of RMI within PJama. PJRMI differs from RMI in a number of ways because it pro-

vides facilities for distributed programming within a persistent context.

This section is divided into three subsections that reflect the three areas where the port affected the implementation and run-time behaviour of DRASTIC. With reference to before and after the port: section 5.2.1 explains the persistence of all exported objects; section 5.2.2 reconnecting to remote objects and section 5.2.3 describes the use of the Registry.

### 5.2.1 Exported Persistent Objects

In PJama version 0.3.5.1, that DRASTIC was ported to, all objects that are available for remote invocation are made persistent. Although it is desirable in an orthogonally persistent system for the persistence of an object to be independent from its other features, it was not possible to maintain the separation of the persistence and distribution qualities of a remotely invocable object in the *first* version of PJRMI.

Certain information about a reference to a remote object is inherently transient, such as the state describing the socket used for inter-object communication. During the execution of a PJama program, remotely-invokable objects may become part of the persistent state, if they become reachable from a root of persistence. In order for PJRMI to maintain the illusion of persistence for an application programmer, when a store containing remotely-invokable objects is re-opened, these objects need to be re-exported to reinitialise their transient state.

PJRMI must be able to keep track of those exported objects which do become reachable from persistent roots, so that they may be re-exported automatically the next time the store is re-opened. To do this, PJRMI makes use of a PJama feature called the `PJActionHandler`. `PJActionHandlers` provide a callback service, enabling code to be associated with an object, to support special treatment of that object at certain points in the execution of a persistent

program: on the re-opening of a store, before or after the promotion of persistent objects to stable storage or on store shutdown. Thus, a predefined `PJActionHandler` is associated with every remotely-invokable object. A method of this object is executed to re-export the object when the store is re-opened, before the execution of any application code that might use it.

As the table of `PJActionHandler` bindings is part of the persistent state, all remotely-invokable objects are reachable from it so they are also made persistent. The first version of PJRMI lacks the necessary support to ensure that the remotely invocable objects are only made persistent if they are also reachable via a normal reference from a persistent application object.

This current limitation of PJRMI is made clear in the documentation but it can be confusing to the programmer. Since the use of `PJActionHandlers` is implicit, programmers using PJRMI may not be aware that their remotely-invokable objects are reachable from these persistent objects. If these programmers have not made their remotely-invokable objects reachable from another persistent root they will assume that they will not be made persistent at all, but they will become persistent because of the implementation described above.

After porting DRASTIC to PJama, this known limitation did not prove to be a problem for the remotely-invokable object (the `inhandler`) inside the DRASTIC platform. This is because the state associated with the `inhandler` is already part of the persistent context and the fact that it was also made persistent because of the implementation above was acceptable. A process booted over a DRASTIC platform only contains one remotely invocable object. There should not be any application-level remotely invocable objects as this would allow them to bypass the evolution technology within the platform. To make an application-level object remotely invocable, its source-level class is preprocessed and all references are then indirected via the platform and the

process' `inHandler`. Remote invocations are then made using the platform, passing the necessary information, such as method name and type-name, to the `inHandler` of the target process.

### 5.2.2 Reconnecting Remote References

Referential integrity is an important concept in the field of persistent programming languages as implementing it prevents, for example, dangling pointers from occurring in the store, thus ensuring store consistency and correctness. Once a reference to a persistent object is created, the use of that reference is guaranteed to work and will always refer to the same object. In a distributed persistent context, references between stores can be created. Guaranteeing referential integrity in such an environment is not possible because of the potential for partial failure within the distributed system. Consider two stores that are available at two different machines. If an inter-store reference to a persistent object is created, its availability cannot be guaranteed as one of the two machines may crash, for example, or the network between the two machines may not be available. For this reason, referential integrity, in the form described above, cannot be guaranteed in a distributed context. However, some aspects of PJRMI's design have been affected by the perceived need to attempt to retain this concept.

To retain the illusion of referential integrity, PJRMI automatically refreshes client-side persistent references to remotely invokable persistent objects when the reference is brought out of persistent storage. This is required by the definition of referential integrity used above. As the reference is part of the persistent state at the client side, when the store is re-started, it should be available for use. However, the state associated with the socket is inherently transient. Therefore, PJRMI automatically reconnects the client to the server through a new socket, so that the reference can be used without the need for the programmer to explicitly retrieve a new reference. If the programmer had to retrieve a new reference to the remote

object, referential integrity of the original would be lost. Client references to a remote object indirect through a local stub object. If the programmer retrieved a new stub object, referential integrity is only maintained if all client-side references can be updated to refer to the new stub object instead of the old one. PJRMI maintains the referential integrity of all client-side references to a remote object by preserving the original, persistent stub object, which is a client-side representation of the remote object.

This design is an example of the consequences of combining distribution and orthogonal persistence by reachability. Reconnection is required to retain the illusion of referential integrity, so that the use of the remote object is possible, assuming an error free distributed communication. In addition, with the current reconnection interface, there is no opportunity for the programmer to control *how* this reference is reconnected; there is no mechanism for executing user-defined code before or after the reconnection is attempted. For a more detailed discussion of referential integrity and distribution within the context of PJama, see [Spe97].

### 5.2.3 The Registry

PJRMI follows the design of RMI in terms of how the Registry is used. Therefore, PJRMI also requires that a Registry is made available on each machine that hosts a remotely-invokable object. This is a sensible requirement as the behaviour of an RMI-based system will be the same after a port to PJRMI, making the port easier. However, DRASTIC requires behaviour that is not provided by the original RMI implementation. DRASTIC normally assumes that only one Registry is required in the entire system. To get the desired behaviour before the port, the `Registry` had to be reimplemented.

To support this model within PJama, a PJRMI class, `org.opj.distribution.PJamaPJExported`, also has to be reimplemented to relax the same-host requirement.

`PJamaPJExported` manages the exporting of remotely-invokable, persistent objects. Its implementation assumes the existence of a local Registry, where it registers its service to refresh references to re-exported, persistent, remotely invokable objects. Without access to the source code, relaxing this requirement is not possible. Neither the `Registry` nor `PJExported` classes provide the programmer with the ability to switch off this check, therefore, only the default behaviour, that of performing the check, is possible. It should be possible to augment the behaviour of both `PJRM` and `RMI` so that the programmer could indicate the `Registry` is on a particular host.

## 6 Classloaders

The DRASTIC platform uses its own zone-aware classloader to load in application code from remote processes, so that it is not reliant on any underlying technology to make classes remotely available, e.g. NFS.

Before porting DRASTIC to PJama it was not clear how the classloader would perform in a persistent context. However, porting to PJama has not affected the use of the classloader. The classloader is part of the persistent state of the platform and the classes it contains will need to be checked when a store is evolved. However, DRASTIC does not provide support to evolve a store, see [Dmi98] for a possible solution; its focus is on providing the framework to support and help manage the run-time evolution of a distributed system.

## 7 Store Creation

The PJama documentation recommends that a store is created with one program and opened and used by another. This methodology was not followed when DRASTIC was ported to PJama. Creation of a new store is performed with the

`PJStore` object. The author is used to the object-oriented programming approach of using an object whenever it is needed, without having to write a separate program to use it. In order to encapsulate all store manipulation code in one place, the Persistence Module contains code to create a new store and to open an already existing one. When a DRASTIC platform is started, specifying the creation of a new PJama store, the platform first of all boots and then attempts a stabilisation to make the newly acquired state persistent. This stabilisation occurs after a significant amount of computation has been performed and as a result garbage objects exist.

Due to a bug in the PJama interpreter, the garbage collector ran during stabilisation. As stabilisation was only partly complete, the garbage collector did not mark as reachable some objects which were in the process of being promoted to stable storage. The garbage collector removed objects from the system that should be made persistent. When stabilisation continued, null references were encountered and the interpreter failed.

In the case where calls to stabilisation are made over an existing store, the garbage collector no longer interferes. However, the problem still exists when calling stabilisation to create the store. This is a known problem, which is why the documentation now recommends that application programmers use the PJama tool `opjcs` to create an empty store before running their application over it. By using the recommended methodology, it was easy to avoid the bug described above.

The approach of placing all the store manipulation code into the Persistence Module initially seemed a sensible one. It was not until this problem came to light that the need for a separate store-creation program became necessary. However, this is a new approach for the first author who is used to programming in an environment which does not provide orthogonal persistence. Moving from an environment consisting of an operating system and file-store to one where the file-store is abstracted away by the persistent language technology re-

quires a change of approach that, at first, feels artificial. Following the recommended methodology requires that a separate program be executed to perform the task of creating a store. The reason for this becomes clear when the paradigm of developing and running many programs against the same store is understood. In this approach there is a unique event of creating the store which needs to be distinguished by a separate act by the developer.

## 8 Performance

One of the main differences between using PJama-based persistence and file based persistence using object serialization is the time taken to read from the store.

When using one of the DRASTIC demo programs with ad-hoc persistence, a 600Kb file is created. This takes between 30s and 60s to read when booting the process over the store. The object serialization system has to read the entire file before the program can proceed. Therefore, a process is delayed for at least 30s at boot time, leading to the problems described in section 4.1.

PJama-based persistence does not suffer from this problem. Objects are incrementally read from the store when the program uses them. Therefore, a PJama based program does not suffer from an initial “big-inhale” of the entire graph or from the other problems described in section 4.1, allowing the programmer to focus on the details of their program rather than the distracting side effects of using an ad-hoc store.

## 9 Conclusions

The experience of porting DRASTIC to PJama was largely a positive one. Porting a 25,000 line Java program took seven working days, longer than would be expected as DRASTIC already contained support for ad-hoc persistence and more

code needed to be changed and checked. Now that the move to PJama has been completed, it will replace the use of object serialization as PJama has become the preferred persistence solution.

The combination of orthogonal persistence and distribution caused a number of problems (section 5). It would be useful to be able to switch off the automatic reconnection to a persistent server from a persistent client so that the programmer is given maximal control over their application. The current requirement that all remotely-invokable objects be persistent did not affect the DRASTIC port, although this could be limiting for other programs. However, this limitation is a known one. If a PJRMI programmer does not want the restriction of having one `Registry` per machine hosting a remotely invokable object, they have to reimplement the `Registry`.

The methodology described in the PJama documentation (described in section 7 in this paper) is not intuitive to a programmer who is not used to working with an orthogonally persistent language. Creating a store with one program and using it with another initially seems less convenient than performing both operations in a single program. However, why the recommended methodology should be followed becomes clear when the paradigm of running different programs against the same store is understood. As the first author did not have much experience with such an approach, the expectations of what the environment should provide were different from those who developed the system. Once this is understood, following the approach is worthwhile as it separates creating a store from using it, resulting in a more flexible system. Following the methodology also had the added benefit of solving the bug encountered during stabilisation which caused the interpreter to fail.

Using PJama is relatively easy. Access to the underlying persistence technology is well encapsulated within the `PJStore` class and, in the general case, porting mainly requires classes to be added; very little existing code needs to be

changed. In the case of porting DRASTIC, more code required changing as there was already support for ad-hoc persistence. Only 83 lines of code had to be added to integrate DRASTIC with PJama and the change to the platform took two working days to perform. However, this is longer than usual as the code that abstracts over the persistence mechanism had to be changed. Therefore, half a day is more representative of the amount of work required when only considering the cost of moving to PJama.

Orthogonal persistence by reachability is an easy abstraction for the programmer to use and is the main reason why changing the code for use with PJama is relatively easy. No code changes are necessary to indicate that a class or object may be made persistent. The programmer only has to focus on the state they require to be persistent. This is easy to indicate through `PJStore` and programming by reachability is natural in an object-oriented language.

## 10 Acknowledgements

The authors thank the reviewers for their constructive comments on an earlier draft of this paper and Malcolm Atkinson, Peter Dickman and John Hagemeister for useful feedback.

## References

- [ADJ<sup>+</sup>96] M. P. Atkinson, L. Daynes, M. J. Jordan, T. Printezis, and S. Spence. An orthogonally persistent Java. *ACM SIGMOD Record*, 25(4):68–75, December 1996.
- [Dmi98] Misha Dmitriev. The First Experience of Class Evolution Support in PJama. In Malcolm Atkinson and Mick Jordan, editors, *To appear in the Third Persistence and Java Workshop*, Tiburon, California, September 1st to 3rd 1998.
- [ED97] Huw Evans and Peter Dickman. DRASTIC: A run-time architecture for evolving, distributed, persistent systems. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '97)*, volume 1241 of *LNCS*, pages 243–275, Jyväskylä, Finland”, June 1997. Springer.
- [ED98] Huw Evans and Peter Dickman. Supporting Distributed Software Evolution with Zones and Contracts. Technical report, Department of Computing Science, Glasgow University, Scotland, UK, 1998.
- [Eva98] Huw Evans. Why Object Serialization is Bad for Providing Persistence in Java. Technical report, Department of Computing Science, Glasgow University. *In preparation*, 1998.
- [GJS97] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, 1997.
- [Spe97] Susan Spence. Distribution Support for PJama. In *Workshop on Persistence and Distribution in Java*. PerDiS Esprit Project, INESC, Lisbon (Portugal), October 1997.
- [Spe98] Susan Spence. Persistent RMI Documentation, version 0.4.6.10 onwards. Department of Computing Science, Glasgow University, Glasgow, Scotland, March 1998.