Distributed Transactions in Java

M.C. Little and S.K. Shrivastava
Department of Computing Science
University of Newcastle, Newcastle upon Tyne, NE1 7RU, England
email: m.c.little@ncl.ac.uk

1.   Introduction

The Web frequently suffers from failures which can affect both the performance and consistency of applications running over it. For example, if a user purchases a cookie (a token) granting access to a newspaper site, it is important that the cookie is delivered and stored if the user's account is debited; a failure could prevent either from occurring, and leave the system in an indeterminate state. For resources such as documents, failures may simply be annoying to users; for commercial services, they can result in loss of revenue and credibility.

Atomic actions (atomic transactions) are a well-known technique for guaranteeing application consistency in the presence of failures. Web applications already exist which offer transactional guarantees to users. However, currently these guarantees only extend to resources used at Web servers, or between servers; browsers are not included, despite being a significant source of unreliability. Providing end-to-end transactional integrity between the browser and the application is important: in the previous example, the cookie *must* be delivered once the user's account has been debited. Cgi-scripts cannot provide this level of transactional integrity since replies sent *after* the transactions have completed may be lost, and replies sent *during* the transaction may need to be revoked if the transaction cannot complete [OSF96]. This is an inherent problem with the original "thin" client model of the Web, where browsers were functionally barren. With the advent of Java it is now possible to consider empowering browsers so that they can fully participate within transactional applications. However, to be widely applicable, we claim that any such transaction system must meet the following three requirements:

(i) it must support distributed, nested transactions;
(ii) it must not compromise the security policy imposed at the browser's site; and,
(iii) it must comply with appropriate standards.

We have designed and implemented the *JavaArjuna* system, a transaction toolkit that meets the above requirements. Our toolkit allows transactional applications to span Web browsers and servers. The toolkit supports application specific customisation - at build time as well as at run time - so that an application can be made transactional without compromising the security policies operational at browsers and servers. Finally, the toolkit complies with the OMG Object Transaction Service (OTS) and the Java Transaction Service (JTS) standards, enabling it to interoperate with other compliant objects and applications. JavaArjuna is unique in that at the time of writing (March 1997), we do not know of any other working system that simultaneously meets all of the above three requirements. A productised version of this system is expected to be commercially available towards the latter half of this year.

2.   Transaction standards for distributed objects

The Object Management Group (OMG) has specified a Common Object Request Broker Architecture (CORBA) that at the basic level consists of the Object Request Broker (ORB) that enables distributed objects to interact with each other [OMG95]. At the next level a number of system level services have been specified. These services include persistence, concurrency control, events and transactions etc. We concentrate here on the Object Transaction Service (OTS) standard.

The OTS is a *protocol engine* intended to guarantee that transactional behaviour is obeyed, but it does not directly support all of the transaction properties. As such it depends on other system level services mentioned before for the required functionality. Although the OTS specification allows transactions to be nested, an implementation need not provide this functionality. The OTS provides interfaces that allow multiple distributed objects to cooperate in a transaction such that all objects commit or abort their changes together. However, the OTS does not require all objects to have transactional behaviour. Instead objects can choose not to support transactional operations at all, or to support it for some requests but not others.

The Transaction Service specification distinguishes between recoverable objects and transactional objects. Recoverable objects are those that contain the actual state that may be changed by a transaction and must therefore be informed when the transaction commits or aborts to ensure the consistency of the state changes. In contrast, a simple transactional object need not be a recoverable object if its state is actually implemented using other recoverable objects. A simple transactional object need not take part in the commit protocol used to determine the outcome of the transaction since it does not maintain any state itself, having delegated that responsibility to other recoverable objects which will take part in the commit process.

The Java Transaction Service (JTS) is a direct mapping of the OTS to the Java language using the standard Java IDL mapping [JTS96]. The JTS is *not* a new standard, but an OTS for Java. Therefore, it benefits from the interoperability with other OTS implementations. Applications and objects written using the JTS can make use of objects written in OTS, and vice versa. This presents advantages to application developers who may have legacy backoffice applications, and when building Web applications which encompass browsers and servers they may take advantage of more efficient compilation techniques at the servers.

For example, the figure below shows a transactional Java application executing at a web browser. The application encompasses JTS transactional objects residing within other browsers and OTS transactional objects residing within an ORB. The JTS objects are written in Java, whereas the OTS objects may be written in another language, e.g., C++ or Smalltalk.
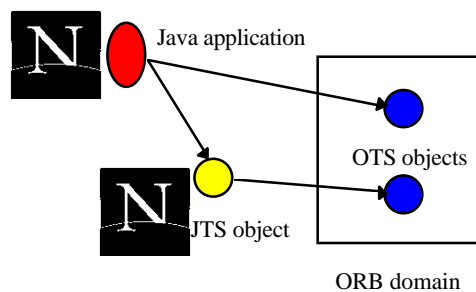

Figure 1: JTS and OTS interaction.

3.   Impact of the Java security model

The use of Java to implement transactional applications raises some important security issues. Java security is imposed by a SecurityManager object, which defines what an applet can, and cannot do [ORA96]. Although eventually users should be able to provide their own SecurityManager, currently the Java interpreter provides the only implementation. Generally an applet cannot remotely communicate with a node other than the one from which it was loaded, neither can it write to the disk of the machine on which it is being run. If an applet is loaded directly from the user's disk then many of these restrictions are relaxed.

However, each implementer of a Java interpreter can provide a different security model and SecurityManager, which may impose different constraints. Therefore, an application written on one interpreter may not be able to execute on another. Moreover, the constraints imposed by a SecurityManager directly affect transactional objects which may require for example, to make state updates persistent by accessing the local disk. There are two obvious solutions to this problem:

·   only allow local transactional objects to reside within an applet, with other objects and services (e.g., persistence) being accessed remotely.

·   modify the Java language and the interpreter and provide a specific implementation of the SecurityManager which, for example, allows all objects to access the local disk [MA96].

Unfortunately, neither of these solutions is general enough. The first solution is unnecessarily restrictive in environments where SecurityManagers do allow applets to access local disks. The second solution lacks portability -the very reason for using Java- as it requires users to have access to specialised implementations.

Our approach, to be described in the next section, does not rely on modifying the language or the interpreter, yet it is flexible enough to enable an application to configure itself to make use of the resources a given SecurityManager permits.

3.   JavaArjuna Implementation

3.1.   Architecture

The approach we have taken to the problem of writing truly portable Java applications is to build an application support framework, *Gandiva*, which isolates applications and programmers from the differences between Java environments [SMW96]. The application can be dynamically reconfigured to take advantage of the environment in which it executes.

Software components are split into two separate entities: the *interface component* and the *implementation component*. The interactions between implementations can only occur through interfaces. A single interface can be used to access multiple implementations, and a single implementation can be accessed through multiple interfaces. The necessity of providing multiple interfaces to implementations has long been recognised. However, we take this further by allowing the bindings of interfaces to implementations, and the interfaces an implementation can be accessed through, to be dynamic and configurable. Applications are written only in terms of interfaces, and although an application can request a specific implementation, it occurs in a way that allows this request to be changed without modifying the application. Therefore, this allows the application to be adapted for each SecurityManager.

The framework has two components:

· *build-time*: programmers write applications without requiring knowledge about the environments on which they will execute. The build-time isolates the programmer and application from details such as how persistence and concurrency control are implemented.

· *run-time*: when an applet runs, it can be dynamically configured to adapt to the environment in which it executes, for example taking advantage of being able to write to a local disk.

The overall architecture of a JavaArjuna transaction system is shown below. The applet executes both on the Orb and Gandiva framework. The implementations within the applet may also involve the Orb or may bypass it, e.g., a particular persistence or concurrency control mechanism which does not involve remote communication.
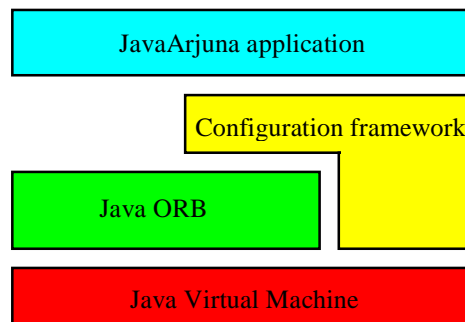

Figure 2: JavaArjuna architecture

3.2   Gandiva build-time support

The Gandiva build-time system offers support to programmers to construct applications from existing interfaces and to build new interfaces and implementations. Interfaces can be automatically generated from a high-level definition language, and contain the necessary code to interact with the run-time system to bind to an appropriate implementation (as described in the next section). To incorporate configurability into an application, the programmer creates a Configuration Management Object (CMO). The CMO contains *data* which specifies the interface to implementation bindings for the application. The data may also specify alternate implementations, e.g., because of possible security restrictions. At bind time an interface interrogates the CMO to determine which implementation it requires, and then passes this information to the run-time system. Importantly for our purposes, the CMO data associated with an application can be specified at run time, therefore providing a way to configure the application for each user and environment.

3.3 Gandiva run-time support

The run-time framework consists primarily of an Inventory Object (IO), which keeps track of the types of implementations available. When an interface requires to be bound to an implementation, it interrogates the application CMO for the implementation type. It then requests an instance of this type from the IO. If the requested implementation does not exist, or cannot be used within the current environment, then the binding will fail. The interface can then attempt an alternate binding if one is specified by the CMO. Importantly, none of this is visible to the application, which simply attempts to create and use an object.

Figure 3 shows the resultant environment within which a Java application executes. Initially the application contains interfaces which are not bound to particular implementations. When an interface is first used, it consults the CMO residing within the application to determine which implementation class to request from the IO. If such an implementation exists the IO returns an instance of this class which the interface binds to. If the SecurityManager will not allow that implementation and an alternate is specified in the CMO, then one will be chosen until either an implementation if obtained, or no further alternates are available.
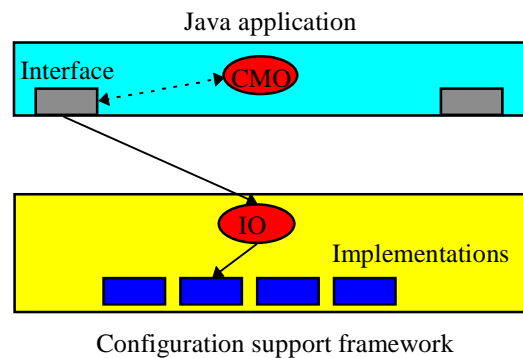

Figure 3: Application executing environment.

3.4 Implementations for transactional applications

For JavaArjuna, we have implemented the following interfaces and associated implementations, which enable a transactional application to execute in any Java-aware browser:

· persistence: in addition to remote implementations of persistence services, there are several implementations which require the ability to access the local disk. These implementations are tailored for specific object types and access patterns, e.g., storing large objects, or concurrently accessing related objects.

· concurrency: as with persistence, remote implementations exist. However, there are several implementations optimised for local use. For example, where sharing of objects between Java applications is not required then only concurrency between individual threads is supported.

3.5 The transactional toolkit

The JTS standard states that the classes defined within it are too low-level for most application programmers, requiring them to manage persistence, concurrency control etc. on behalf of every transactional object. Therefore, a higher-level API should be provided which attempts to hide much of these details from programmers. The API we have provided has been based on the experiences gained from extensive use of the Arjuna system [GDP95]. Distribution support is provided by the Orb on which the application resides. In JavaArjuna objects obtain desired properties through inheritance. The classes form a hierarchy, as depicted below:

```
StateManager                // Basic naming, persistence and recovery control
    LockManager             // Basic two-phase locking concurrency control
        User-Defined Classes
    Lock                    // Standard lock type for multiple readers/single writer
        User-Defined Lock Classes
    AtomicAction            // Implements atomic action control abstraction
    AbstractRecord          // Important utility class
        RecoveryRecord      // handles object recovery
        LockRecord          // handles object locking
```

```
    RecordList        // Intentions list
    other management record types
```

3.6   Building transactional applications

The API relieves programmers from having to explicitly register resources with a transaction. Neither do they have to manage persistence or concurrency control, which are managed on their behalf by the JavaArjuna classes `StateManager` and `LockManager`. To make use of atomic actions in an application, instances of the class `AtomicAction` must be declared by the programmer. The operations this class provides (`begin`, `abort`, `commit`) can then be used to start and manipulate atomic actions (including nested actions). The only objects controlled by the resulting atomic actions are those objects which are either instances of JavaArjuna classes or are user-defined classes derived from `LockManager` and hence are members of the hierarchy shown previously. Most JavaArjuna system classes are derived from the base class `StateManager`, which provides primitive facilities necessary for managing persistent and recoverable objects. These facilities include support for the activation and de-activation of objects, and state-based object recovery. Thus, instances of the class `StateManager` are the principal users of the object store service, which is provided through a suitable interface/implementation separation. The class `LockManager` uses the facilities of `StateManager` and provides the concurrency control (two-phase locking in the current implementation) required for implementing the serialisability property of atomic actions.

4   Concluding remarks

This paper has described the design and implementation of JavaArjuna, a standards compliant toolkit for the construction of fault-tolerant Web and Internet applications using atomic actions. The toolkit addresses the requirement for end-to-end transactional guarantees by allowing applications to be built which encompass Web browsers, rather than just Web servers. Transactional objects can reside within Web servers, and interact with objects and applications within other browsers or backoffice environments. As well as being standards compliant, the system does not compromise the security policy imposed at the browser's site. This means that applications can be built without requiring specific security policies, such as being able to write to the local disk. An application can be configured at build-time or run-time to adapt to the environment/user in which it runs, enabling the same application to execute anywhere.

All of the work presented here has been implemented, and simple applications have been constructed to test the system [MCL97]. In the full paper we shall describe the transactional toolkit and its configurable environment in more detail, and provide performance figures.

Selected References

[GDP95]   "The Design and Implementation of Arjuna", G.D. Parrington et al, USENIX Computing Systems Journal, Vol. 8., No. 3, Summer 1995, pp. 253-306.

[JTS96]   "JTS: A Java Transaction Service API", V. Matena and R. Cattell, Sun Microsystems, December 1996.

[SMW96]   "The Design and Implementation of a Framework for Configurable Software", S. M. Wheater and M. C. Little, Proceedings of the 3rd International Workshop on Configurable Distributed Systems, May 1996, pp. 136-143.

[MCL97]   M. C. Little, S. K. Shrivastava, S. J. Caughey, and D. B. Ingham, "Constructing Reliable Web Applications Using Atomic Actions", Proceedings of the 6th Web Conference, April 1997.

[ORA96]   "Java in a Nutshell", O'Reilly and Associates, Inc., 1996.

[MA96]   "Draft Pjava Design 1.2", M. Atkinson et al, Department of Computing Science, University of Glasgow, January 1996.

[OMG95]   "CORBAservices: Common Object Services Specification", OMG Document Number 95-3-31, March 1995.

[OSF96]   "Applications of the Secure Web Technology in Transaction Processing Systems", T. Sanfilippo and D. Weisman, The Open Group Research Institute, November 1996.