Java Transactions for the Internet

M.C. Little and S.K. Shrivastava
(M.C.Little@ncl.ac.uk, Santosh.Shrivastava@ncl.ac.uk)
Department of Computing Science
University of Newcastle, Newcastle upon Tyne, NE1 7RU, England

Appeared in the Proceedings of the 4th Conference on Object-Oriented Technologies and Systems (COOTS), April 1998.

Abstract

The Web frequently suffers from failures which affect the performance and consistency of applications run over it. An important fault-tolerance technique is the use of *atomic transactions* for controlling operations on services. While it has been possible to make server-side Web applications transactional, browsers typically did not possess such facilities. However, with the advent of Java it is now possible to consider empowering browsers so that they can fully participate within transactional applications. In this paper we present the design and implementation of a standards compliant transactional toolkit for the Web. The toolkit allows transactional applications to span Web browsers and servers and supports application specific customisation, so that an application can be made transactional without compromising the security policies operational at browsers and servers.

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 inconsistent 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 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; clients (browsers) are not included, despite their role being significant in applications such as mentioned previously. 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. 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 *JTSArjuna* system, a transaction toolkit that meets the above requirements. Our toolkit allows transactional applications to span Web browsers and servers and supports application specific customisation, so that an application can be made transactional without compromising the security policies operational at browsers and servers. The toolkit complies with the OMG Object Transaction Service (OTS) and the Java Transaction Service (JTS) standards [OMG95][VM96]. Although the OMG has specified several object services, there is no specification for an overall object model with which to glue them together into a coherent application development framework. Therefore, we have provided a high-level API which allows programmers to be isolated from many of the issues involved in building transactional applications. This API is the result of extensive experience with the original C++ Arjuna distributed transaction system [GDP95][SKS95].

2. Transaction standards for distributed objects

For a transaction system to be widely applicable, it must conform to the standards. The most widely accepted standard for distributed objects is the Common Object Request Broker Architecture (CORBA) from the Object Management Group (OMG). It consists of the Object Request Broker (ORB) that enables distributed objects to interact with each other, and a number of services have also been specified, which include persistence, concurrency control and the Object Transaction Service.

2.1 The Object Transaction Service

The Object Transaction Service supports the well known concept of ACID transactions. 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. This is achieved be registering appropriate objects that support the `Resource` interface (or the derived `SubtransactionsAwareResource` interface) with the current transaction. In contrast, a simple transactional object need not necessarily be a recoverable object if its state is actually implemented using other recoverable objects. The major difference is that 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.

It is important to realise that the OTS is simply a *protocol engine* that guarantees that transactional behaviour is obeyed but does not directly support all of the transaction properties. As such it requires other co-operating services that implement the required functionality, including:

- *Persistence/Recovery Service*. Required to support the atomicity and durability properties. (There is no recovery service currently specified by the OMG.)
- *Concurrency Control Service*. Required to support the isolation properties.

2.2 Writing OTS applications

To participate within an OTS transaction, a programmer must be concerned with:

- creating `Resource` and `SubtransactionAwareResource` objects for each object which will participate within the transaction/subtransaction. These resources are responsible for the persistence, concurrency control, and recovery for the object. The OTS will invoke these objects during the prepare/commit/abort phase of the (sub)transaction, and the `Resources` must then perform all appropriate work.
- registering `Resource` and `SubtransactionAwareResource` objects at the correct time within the transaction, and ensuring that the object is only registered once within a given transaction. As part of registration a `Resource` will receive a reference to a `RecoveryCoordinator` which must be made persistent so that recovery can occur in the event of a failure.
- ensuring that, in the case of nested transactions, any propagation of resources such as locks to parent transactions are correctly performed. Propagation of `SubtransactionAwareResource` objects to parents must also be managed.
- in the event of failures, the programmer or system administrator is responsible for driving the crash recovery for each `Resource` which was participating within the transaction.

The OTS does not provide any `Resource` implementations. These must be provided by the application programmer or the OTS implementer. The interfaces defined within the OTS specification are too low-level for most application programmers. Therefore, we have designed JTSArjuna to make use of raw Common Object Services interfaces but provide a higher-level API for building transactional applications and frameworks. This API automates much of the above activities concerned with participating in an OTS transaction.

The architecture of the system is shown in figure 1. As we shall show, the API interacts with the concurrency control and persistence services, and automatically registers appropriate resources for transactional objects.

These resources may also use the persistence and concurrency services.

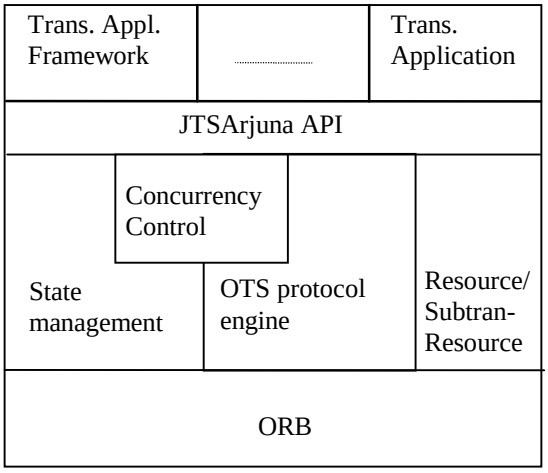

Figure 1: JTSArjuna structure

3. Requirements for configuration

The use of Java to implement transactional applications raises some important security issues. Java security is imposed by a SecurityManager object, which defines what a program can, and cannot do [DF97][JSF95]. However, there is no standard for the SecurityManager implementation, with the result that an application written for one interpreter may not be able to execute as intended on another. The recent addition of digital signatures, allowing users to specify security capabilities on a per signature basis, increases the difficulties of building truly portable applications.

The constraints imposed by SecurityManagers can directly affect transactional applications which may require, for example, to make state updates persistent by accessing the local disk. There are two obvious solutions to this problem: (i) all objects must reside within domains which have well-behaved security constraints (Web servers), or (ii) modify the Java language and the interpreter and provide an implementation of the SecurityManager which relaxes these security restrictions [MA96]. The first solution is unnecessarily restrictive in environments where SecurityManagers do allow programs increased flexibility. The second solution lacks portability as it requires users to have access to specialised implementations.

Our solution was to design and implement the *JavaGandiva* configuration support framework based on the model described in [SMW96], which isolates applications and programmers from the differences between Java SecurityManagers. Applications can be dynamically configured to take advantage of the environment in which they execute. As we shall show, several JTSArjuna classes must use this framework to provide portability across SecurityManagers.

3.1 Configuration model

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 by ensuring that interfaces use only those implementations which can operate within a particular environment.

3.2 JavaGandiva implementation

In an object-oriented language like Java, it is possible to map interface components and implementation components onto *interface and implementation classes* respectively. Object-orientation allows us to specify the binding between interface class and implementation class either through *inheritance* or *delegation*. We require the binding between interface classes and implementation classes to be evaluated when the interface class is instantiated. Therefore, delegation best matches our requirements to control this binding at run-time [SMW96].

In order to leave this binding until run-time we must specify it as data and not within the code of the interface class. The instance of the interface class (*interface object*) uses this data to create and bind to the correct instance of the implementation class (*implementation object*). To provide this separation of interface component and implementation component requires changing what would have been a single Java class into three classes, and a Java `interface`:

(i) *the interface class*: users interact with instances of this class, which defines the public operations that can be invoked on the implementation. The only implementation specific information present in the class definition is a reference to an instance of an

implementation interface, to which the interface delegates all operations.

(ii) *the implementation interface*: this is a Java `interface` and all implementations accessible to an interface class implement it. This guarantees that all implementations conform to a known type.

(iii) *the implementation class*: instances of this class represent the implementation of an object. Implementation classes can be derived from multiple implementation interfaces.

(iv) *the control class*: this class provides access to operations that manipulate the non-functional characteristics of an implementation class. Implementation classes provide an operation that returns a specific instance of this control class. Interface classes provide an operation that can be used to request an instance of the implementation's control class.

Figure 2 shows an object structure formed by the above classes, where the implementation specific objects are shown in grey.

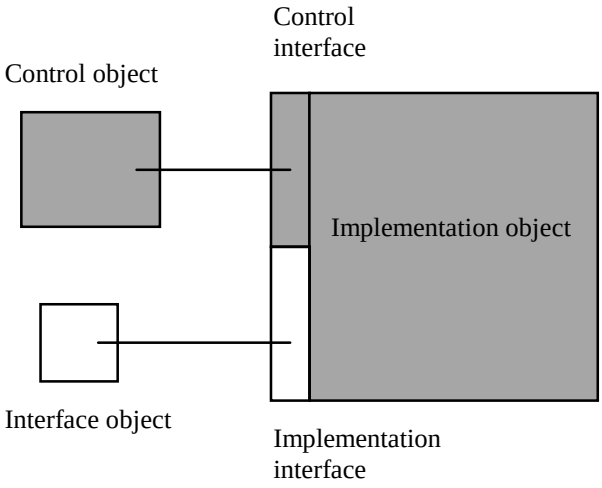

Figure 2: Interface, Implementation and Control Objects.

3.3 JavaGandiva built-time support

The JavaGandiva 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, and any data required by implementations for initialisation. 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.

3.4 JavaGandiva run-time support

The run-time consists primarily of an *Implementation Repository* which is used for creating new instances of (arbitrary) implementation classes given their class names. Implementation classes can be registered with the repository so that instances of them can be created later. The repository isolates interfaces from direct implementation creation; as we shall see, all aspects of implementation creation are hidden within the repository, so that modification of the types of implementations available to an application and interface does not require changes to either.

Figure 3 illustrates how an interface uses these objects when binding to an appropriate implementation. 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 repository. If the requested implementation type 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.

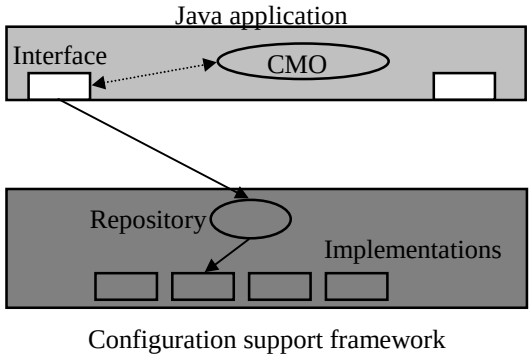

Figure 3: Application execution environment.

3.5 Specifying an application's configuration

The configuration management object is implemented by the `ObjectName` class. This configuration information is maintained as a set of *attributes*; each attribute is a name (string), value pair. An interface object uses the attributes of `ObjectName` to determine the type of its implementation; this implementation can also use the `ObjectName` to configure itself, e.g., to obtain its initial state. If multiple bindings are possible for the interface because of possible security restrictions, the `ObjectName` can specify alternate implementations.

The (simplified) signature of `ObjectName`, without the exceptions it can throw, is shown below:

```
public class ObjectName implements
                                    Serializable
{
// the supported attribute types

public static final int SIGNED_NUMBER = 0;

// for C++ compatibility
public static final int UNSIGNED_NUMBER = 1;

public static final int STRING = 2;
public static final int OBJECTNAME = 3;
public static final int CLASSNAME = 4;
public static final int UID = 5;

public int attributeType (String attrName);
public String firstAttributeName ();
public String nextAttributeName (String curr);

/*
 * Now a series of set/get methods for each
 * type of attribute. We show only two for
 * simplicity.
 */

public long getLongAttribute (String atr);
public String getStringAttribute (String atr);

public void setLongAttribute (String atr,
                              long value);
public void setStringAttribute (String atr,
                                String value);

public boolean removeAttribute (String atr);
public boolean equals (ObjectName objectName);
public boolean notEquals (ObjectName objName);

// how to store/retrieve data
private NameService _nameService;
}
```

An attribute value can be one of six basic types. `ObjectName` is responsible for run-time type checking: an exception is raised if an interface requests the wrong type for an attribute. There are methods for creating new attribute name, value pair mappings, and for retrieving an attribute given its name. Additionally, it is possible to query the type of an attribute using `attributeType`, and to iterate through all of the attributes using `firstAttributeName` and `nextAttributeName`.

To enable the configuration information to be stored in a flexible manner, `ObjectName` stores and retrieves the information using a separate NameService interface and implementation. Therefore, the means of storing this configuration data can be changed simply be changing the NameService implementation. For example, the JDBC (Java Database Connectivity) API is a standard SQL database access interface, providing uniform access to a wide range of relational databases. By providing a suitable NameService implementation, the `ObjectName` data could be maintained within such a database. However, to minimise external dependencies, our current implementation for Web applications embeds the `ObjectName` data within the HTML document which is downloaded with the Java application. The HTML document is created automatically from a separate description language.

3.6 Implementation repository

The implementation repository is provided by the Inventory, which is an interface class and a set of implementation classes. To be able to create implementations for interfaces, the inventory must be populated with these implementations. Populating the inventory can occur:

1) *statically at build time*: each implementation can be registered with the inventory when the application is built, i.e., a specific inventory is constructed for each application. If implementations are required to be added or removed from the inventory then the inventory implementation must be modified.

2) *dynamically at run time*: implementations may be loaded across the network or from the local disk. Given the name of a class, an inventory can attempt to load it dynamically. This has the advantage of flexibility, but requires the sources of these implementations (e.g., Web servers) to remain available while the application is being configured.

Because the inventory is accessed through a well-defined interface, changing the implementation from, say 1) to 2), does not require any changes in an application.

The Inventory interface class has methods for obtaining an instance of an implementation from its class name. For simplicity we show only a representative set of these methods, without the exceptions they throw:

```
public class Inventory
{
public synchronized Object createVoid
                            (String typeName);
public synchronized Object createObjectName
                            (String typeName,
                  ObjectName paramObjectName);
public synchronized Object createResources
                            (String typeName,
                      Object[] paramResources);

/*
 * A handle on the application's inventory
 * for bootstrapping (already bound interface
 * and implementation.
 */

public static Inventory inventory ();
}
```

Each create method takes the name of the implementation class to instantiate and, depending on the method, may pass additional parameter(s) to the created implementation. For example, `createObjectName` will pass the `ObjectName` parameter to the implementation when it is created. In order that the inventory can deal with any Java implementation class, it returns all created objects to the interface as instances of the Java `Object` class, which is the base class from which all Java classes are derived. The interface can then safely convert this back to the actual type.

3.7 Determining security restrictions

In order to configure itself to operate within a specific security environment, an application must be able to determine the restrictions imposed by that environment. At bind time an interface must be able to determine whether the implementation it receives from the inventory can work within the current security restrictions. Therefore, each implementation object must provide a `canExecute` method which returns either true if it can execute within the current environment, or false if it cannot. When the inventory returns an implementation object, the interface calls this method to determine whether the object can function. If it cannot, the interface can ask the `ObjectName` for the name of another implementation, and pass this to the inventory.

To determine whether or not it can function within the security environment, the implementation object may extract information from the `ObjectName` it is given when it is created, e.g., the location of the object store database to use. Shown below is the `canExecute` method for a simple object store service which writes to the local file system:

```
public SimpleObjectStore implements
                                ObjectStoreImple
{
public boolean canExecute ()
{
  /*
   * First get handle on current
   * SecurityManager.
   */

   SecurityManager manager =
                  System.getSecurityManager();

   if (manager == null)
     return true; // no restrictions!
   else
   {
     /*
      * There is a SecurityManager, so
      * interrogate it.
      */

     try
     {
       /*
        * Assume these file names were read
        * from the ObjectName when we were
        * created.
        */

        manager.checkRead("/ObjStore/data");
        manager.checkWrite("/ObjStore/data);
        manager.checkDelete("/ObjStore/data");

        return true;
     }
     catch (Exception e)
     {
       /*
        * SecurityManager raised an
        * exception, could try alternate
        * location.
        */

       return false;
     }
   }
}
}
```

4. JTSArjuna implementation

JTSArjuna exploits object-oriented techniques to present programmers with a toolkit of Java classes from which application classes can inherit to obtain desired properties, such as persistence and concurrency control [MCL97]. These classes form a hierarchy, part of which is shown below.

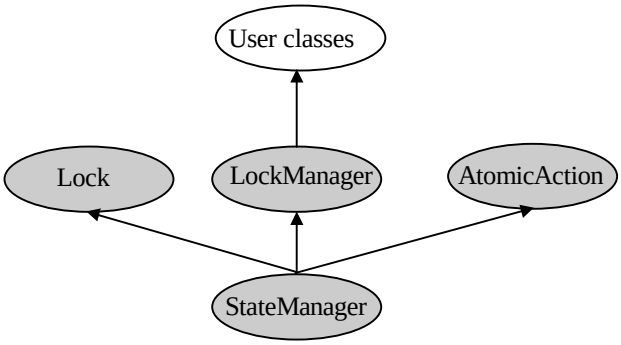

Figure 4: JTSArjuna class hierarchy

As we shall show, apart from specifying the scopes of transactions, and setting appropriate locks within objects, the application programmer does not have any other responsibilities: JTSArjuna guarantees that transactional objects will be registered with, and be driven by, the appropriate transactions, and crash recovery mechanisms are invoked automatically in the event of failures.

4.1 Saving object states

JTSArjuna needs to be able to remember the state of an object for several purposes, including recovery (the state represents some past state of the object) and persistence (the state represents the final state of an object at application termination). Since these requirements have common functionality they are all implemented using the same mechanism: the classes `InputObjectState` and `OutputObjectState`. The classes maintain an internal array into which instances of the standard types can be contiguously packed (unpacked) using appropriate `pack (unpack)` operations. This buffer is automatically resized as required should it have insufficient space. The instances are all stored in the buffer in a standard form (so-called network byte order) to make them machine independent. Any other architecture independent format (such as XDR or ASN.1) could be implemented simply by replacing the operations with ones appropriate to the encoding required. (We are currently examining using the new object serialization mechanisms within the Java language.)

4.2 The object store

Implementations of persistence can be affected by restrictions imposed by the Java SecurityManager. Therefore, the object store provided with *JTSArjuna* is implemented using the techniques of interface/implementation separation described earlier. The current distribution has implementations which write object states to the local file system or database, and remote implementations, where the interface uses a client stub (proxy) to remote services.

Persistent objects are assigned unique identifiers (instances of the Uid class), when the are created, and this is used to identify them within the object store. States are read using the `read_committed` operation and written by the `write_(un)committed` operations.

```
public interface ObjectStoreImple
{
public boolean commit_state (Uid id);
public InputObjectState read_committed (Uid
id);
public InputObjectState read_uncommitted (Uid
id);
public boolean remove_committed (Uid id);
public boolean remove_uncommitted (Uid id);
public boolean write_committed (Uid id,
                    OutputObjectState state);
public boolean write_uncommitted (Uid id,
                    OutputObjectState state);
};
```

4.3 Recovery and persistence

At the root of the class hierarchy is the class `StateManager`. This class is responsible for object activation and deactivation and object recovery. The simplified signature of the class is:

```
public abstract class StateManager
{
public boolean activate ();
public boolean deactivate (boolean commit);

public Uid get_uid (); // object's identifier.

// methods to be provided by a derived class

public abstract boolean restore_state
                      (InputObjectState os);
public abstract boolean save_state
                     (OutputObjectState os);

protected StateManager ();
protected StateManager (Uid id);
};
```

Objects are assumed to be of three possible flavours. They may simply be *recoverable*, in which case `StateManager` will attempt to generate and maintain appropriate recovery information for the object. Such objects have lifetimes that do not exceed the application program that creates them. Objects may be *recoverable and persistent*, in which case the lifetime of the object is assumed to be greater than that of the creating or accessing application, so that in addition to maintaining recovery information `StateManager` will attempt to automatically load (unload) any existing persistent state for the object by calling the `activate` (`deactivate`) operation at appropriate times. Finally, objects may possess none of these capabilities, in which case no recovery information is ever kept nor is object activation/deactivation ever automatically attempted.

If an object is recoverable (or persistent) then `StateManager` will invoke the operations `save_state` (while performing `deactivate`), and `restore_state` (while performing `activate`) at various points during the execution of the application. These operations *must* be implemented by the programmer since `StateManager` cannot detect user level state changes. (We are examining the automatic generation of default save_state and restore_state operations, allowing the programmer to override this when application specific knowledge can be used to improve efficiency.) This gives the programmer the ability to decide which parts of an object's state should be made persistent. For example, for a spreadsheet it may not be necessary to save all entries if some values can simply be recomputed. The `save_state` implementation for a class `Example` that has integer member variables called `A`, `B` and `C` could simply be:

```
public boolean save_state(OutputObjectState o)
{
    return (o.packInt(A) && o.packInt(B)
            && o.packInt(C));
}
```

4.4 The concurrency controller

The concurrency controller is implemented by the class `LockManager` which provides sensible default behaviour while allowing the programmer to override it if deemed necessary by the particular semantics of the class being programmed. As with `StateManager` and persistence, concurrency control implementations are accessed through interfaces. As well as providing access to remote services, the current implementations of concurrency control available to interfaces include:

- local disk/database implementation, where locks are made persistent by being written to the local file system or database.
- a purely local implementation, where locks are maintained within the memory of the virtual machine which created them; this implementation has better performance than when writing locks to the local disk, but objects cannot be shared between virtual machines. Importantly, it is a basic Java object with no requirements which can be affected by the SecurityManager.

The primary programmer interface to the concurrency controller is via the `setlock` operation. By default, the runtime system enforces strict two-phase locking following a multiple reader, single writer policy on a per object basis. Lock acquisition is (of necessity) under programmer control, since just as `StateManager` cannot determine if an operation modifies an object, `LockManager` cannot determine if an operation requires a read or write lock. Lock release, however, is under control of the system and requires no further intervention by the programmer. This ensures that the two-phase property can be correctly maintained.

```
public abstract class LockManager
                            extends StateManager
{
public LockResult setlock (Lock toSet,
                           int retry,
                           int timeout);
};
```

The `LockManager` class is primarily responsible for managing requests to set a lock on an object or to release a lock as appropriate. However, since it is derived from `StateManager`, it can also control when some of the inherited facilities are invoked. For example, `LockManager` assumes that the setting of a write lock implies that the invoking operation must be about to modify the object. This may in turn cause recovery information to be saved if the object is recoverable. In a similar fashion, successful lock acquisition causes `activate` to be invoked.

The code below shows how we may try to obtain a write lock on an object:

```
public class Example extends LockManager
{
public boolean foobar ()
{
  AtomicAction A = new AtomicAction;
  boolean result = false;

  A.begin();

  if (setlock(new Lock(LockMode.WRITE) ==
                                  Lock.GRANTED)
  {
    /*
     * Do some work, and JTSArjuna will
     * guarantee ACID properties.
     */

    // automatically aborts if fails

    if (A.commit() == AtomicAction.COMMITTED)
    {
      result = true;
    }
  }
  else
    A.rollback();

  return result;
}
}
```

4.5 Configuration hierarchy

Figure 5 shows a transactional user class inheriting from LockManager. Internally, LockManager accesses the concurrency service (CC) through an interface, and StateManager does likewise with the persistence service (POS). For each application object, the implementations of CC and POS are not chosen until run-time.

Additional implementations can be provided without changing the JTSArjuna system or applications which use it. The JTSArjuna API isolates programmers from the different POS and CC implementations, allowing them to concentrate on the application.

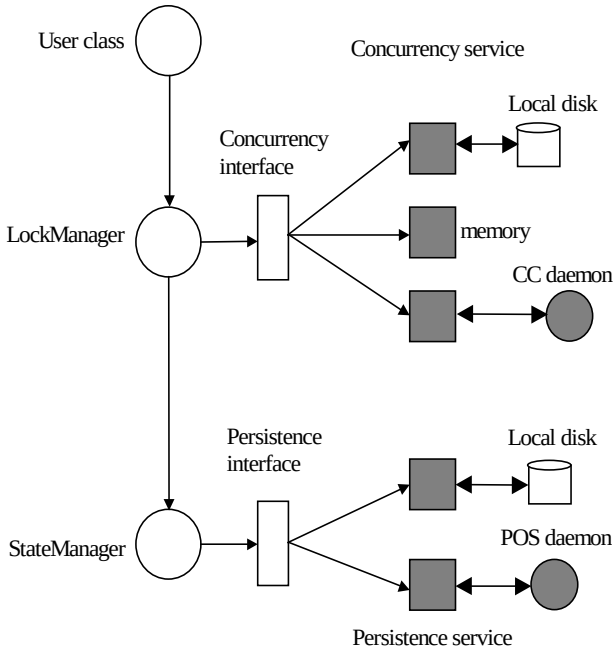

Figure 5: Configuration hierarchy

5. Performance results

Table 1 shows some basic performance results for JTSArjuna, obtained using JDK1.2 running on a Sun Ultra Enterprise 1/170 with 128Meg of RAM. In these tests, the transactional object operated upon had a single integer as its state. (As shown in the table, this transactional object was sometimes only recoverable, i.e., its state was not obtained from/saved to disk.) All timings have been averaged over 1000 runs.

Type of operation	Time taken
Update a persistent object	21.6 milliseconds
Update a recoverable object	11.2 milliseconds
Create and commit a null transaction	1.1 milliseconds
Create and commit a null nested transaction and its parent	1.9 milliseconds

Table 1:JTSArjuna performance figures

These figures represent the initial implementation of our Java transactions. Based upon our experiences with JTSArjuna and its C++ counterpart, we believe that further optimisations to the system are possible which will improve performance.

6. Newspaper example using JTSArjuna

In this section we shall illustrate the different aspects of constructing a transactional application using JTSArjuna. Consider the example of subscribing to an on-line newspaper described in the introduction.

The entities involved in the newspaper application are:

- the user's on-line bank, from where funds will be debited. We shall assume that the newspaper's account is also located here.
- the newspaper site, where the user's details will be added upon successfully completing the transaction.
- the user's browser site, where a cookie authenticating the user must be delivered and stored.

Each of the entities is represented as a separate transactional object (see figure 6). A transaction will begin when the user downloads the Java application and types in the bank account details. The application will then attempt to debit the account and, if successful, place the cookie within the cookie object at the browser. It will then commit the transaction. If a failure occurs, the transaction and all of its work will be aborted.

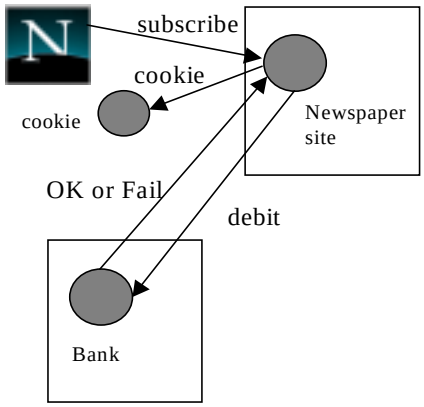

Figure 6: Transactional newspaper.

We first use the transactional toolkit to construct the application classes and partition the application as shown. An example of the cookie class which resides within the browser is:

```
public class Cookie extends LockManager
{
public Cookie ();

public boolean depositCookie (UserDetails obj)
```

```
{
  AtomicAction B = new AtomicAction();
  boolean result = false;

  B.begin (); // start transaction
              // automatically nested if one
              // is already running

  if (setlock(new Lock(LockMode.WRITE) ==
                                    Lock.GRANTED)
  {
    userDetails = obj;

    if (B.commit()) // aborts if cannot commit
      result = true;
  }
  else
    B.abort();

  return result;
};

public boolean save_state
                      (OutputObjectState os);
public boolean restore_state
                      (InputObjectState os);

private UserDetails userDetails;
};
```

An example of the server code is shown below. Apart from declaring instances of the required objects and invoking the methods for transferring funds between accounts and depositing the cookie, the programmer need only start and terminate the transaction. The transaction system will guarantee the outcome even in the presence of failures.

```
{
  AtomicAction A = new AtomicAction();
  BankAccount B1 = new BankAccount(UserNumb);
  BankAccount B2 = new BankAccount(PaperNumb);
  Cookie C = new Cookie();

  A.begin();

  if (B1.debit(amount) && B2.credit(amount))
  {
    if (C.depositCookie(UserDetails))
      A.commit();
    else
      A.abort();
  }
  else
    A.abort();
}
```

Once the application has been constructed, we can decide on the configuration. The transactional object within the browser represents the cookie, which is initially empty. Upon successful completion of the transaction, the cookie will have been stored for future use. The requirements on concurrency control for the cookie are minimal since there will be no concurrent access by multiple users; therefore, the local non-persistent concurrency control implementation can be used. Since this implementation can be guaranteed to work under all SecurityManagers, we require no alternate.

Obviously we would like to store the cookie on the user's local disk. However, security restrictions imposed by the browser's SecurityManager (or by the user if digital signatures are being used) may prevent this. Thus, we require an alternate form of persistence in these situations. In this example we shall assume that the newspaper site will provide a persistence service implementation which is available remotely should the local implementation fail.

After identifying the application configuration, we can construct the HTML document containing the configuration information which will be downloaded with the Java application. (The '~' and '!' characters preceding each attribute value are used for runtime type checking by `ObjectName`.) Importantly, there are no requirements from the application user: all implementations will be loaded across the network when required.

```
<HTML>
<HEAD><TITLE>Example Applet</TITLE></HEAD>
<BODY>
<APPLET CODE=TranApplet.class WIDTH=400
HEIGHT=200>
<PARAM NAME=OSClassName1
VALUE="~LocalObjectStoreImple">
<PARAM NAME=OSLocation1
VALUE="!/tmp/ObjectStore">
<PARAM NAME=OSClassName2
VALUE="~RemoteObjectStoreImple">
<PARAM NAME=OSLocation2
VALUE="!glororan.ncl.ac.uk">
<PARAM NAME=CCClassName1
VALUE="~LocalCCImple">
</APPLET>
</BODY>
</HTML>
```

The preferred type of the persistence service is LocalObjectStoreImple, with the attribute name OSClassName, and the location of the object store is the directory /tmp/ObjectStore. If this fails, the interface can use the alternate implementation RemoteObjectStoreImple which is on the specified machine. The concurrency service is local. If the programmer wishes to change the configuration of the application, only modifications to the HTML document are required.

7. Comparisons with other systems

We are not aware of any other working OTS/JTS compliant, configurable transaction system; therefore, in this section we briefly describe some systems which offer limited functionality.

7.1 Transactions through cgi-scripts

Figure 7 shows how it is possible to use cgi-scripts to allow users to make use of applications which

manipulate atomic resources [TRA96]: the user selects a URL which references a cgi-script on a Web server (message 1), which then performs the action and returns a response to the browser (message 2) *after* the action has completed. (Returning the message during the action is incorrect since the action may not be able to commit the changes.)

In a failure free environment, this mechanism works well, with atomic actions guaranteeing the consistency of the server application. However, in the presence of failures it is possible for message 2 to be lost between the server and the browser. If the transaction commits, the reply will be sent after the transaction has ended; therefore, other work performed within the transaction will have been made permanent. For some applications this may not be a problem, e.g., where the result is simply confirmation that the operation has been performed. If the result is a cookie, however, the loss of the cookie will leave the user without his purchase and money, and may require the service provider to perform complex procedures to verify the cookie was lost, invalidate it and issue another.

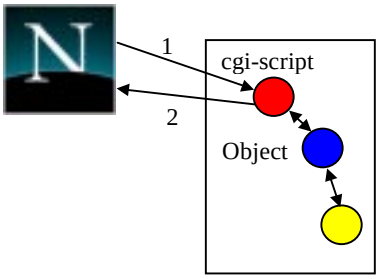

Figure 7: transactions through cgi-scripts

7.2 Transactions in persistent Java

There are several groups working on incorporating transactions into persistent Java [MA96]. These schemes are based on providing atomic actions with *orthogonal-persistence*: objects are written without requiring knowledge that they may be persistent or atomic: the Java runtime environment is modified to provide this functionality. The program simply starts and ends transactions, and every object which is manipulated within a transaction will automatically be made atomic. Although these approaches provide a convenient programming model, we believe that they are unsuitable for Web applications for the following reasons:

(i) They require changes to the Java interpreter and language. Applications written using these systems will only execute on specialised interpreters.

(ii) Both schemes assume that the entire application will be written in Java, and will not be distributed, i.e., it will either execute at the browser or at the Web server.

8. Concluding remarks

This paper has described the design and implementation of JTSArjuna, 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.

Acknowledgements

The work reported here has been supported in part by a grant from UK Engineering and Physical Sciences Research Council (grant no. GR/L 73708).

References

[DF97] D. Flanagan, "Java in a Nutshell 2nd Edition", O'Reilly and Associates, Inc., 1996.

[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.

[JSF95] J. S. Fritzinger and M. Mueller, "Java Security", Sun Microsystems, 1995.

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

[MCL97] M. C. Little and S. K. Shrivastava, “Distributed Transactions in Java”, Proceedings of the 7th International Workshop on High Performance Transaction Systems, September 1997, pp. 151-155.

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

[SKS95] S. K. Shrivastava, “Lessons learned from building and using the Arjuna distributed programming system,” International Workshop on Distributed Computing Systems: Theory meets Practice, Dagsthul, September 1994, LNCS 938, Springer-Verlag, July 1995.

[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.

[TRA96] “Transarc DE-Light Web Client Technical Description”, Transarc Corporation, February 1996.

[VM96] “JTS: A Java Transaction Service API”, V. Matena and R. Cattell, Sun Microsystems, December 1996.

Availability

Further information about JTSArjuna, including how to obtain and license the software, can be obtained from our Web site (http://arjuna.ncl.ac.uk) or by emailing M.C.Little@ncl.ac.uk