Integrating the Object Transaction Service with the Web

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

Appeared in the Proceedings of the 2nd IEEE Workshop on Enterprise Distributed Object Computing (EDOC98), La Jolla California, November 1998, pp. 194-205.

Abstract

The Web frequently suffers from failures which can affect both the performance and consistency of applications running over it. 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. However, their use within Web applications is currently limited to Web servers: browsers are not included, despite their role becoming more significant in electronic commerce applications. With the advent of Java it is possible to empower browsers so that they can fully participate within transactional applications. However, requiring a browser to incorporate a full transaction processing system for all applications would impose an overhead on all users. Therefore, in this paper we shall show how the interfaces defined by the OMG's Object Transaction System can be used to provide a lightweight solution to obtaining end-to-end transactional requirements. We shall illustrate this technique with a worked example.

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 granting access to a newspaper site, it is important that the cookie is delivered 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, with their "all-or-nothing" property, are a well-known technique for guaranteeing application consistency in the presence of failures. Although Web applications exist which offer transactional guarantees to users, these guarantees only extend to resources used at Web servers, or between servers; clients (browsers) are not included, despite their role becoming more significant within the Web. Providing *end-to-end transactional integrity* between the browser and the application (server) is therefore important, as it will allow work involving *both* the browser and the server to be atomic.

However, current techniques based on cgi-scripts cannot provide end-to-end guarantees [1]. As illustrated in figure 1, the user selects a URL which references a cgi-script on a Web server (message 1), which then performs the transaction and returns a response to the browser (message 2) *after* the transaction has completed [2]. Returning the message during the transaction is incorrect since the server may not be able to commit the changes.

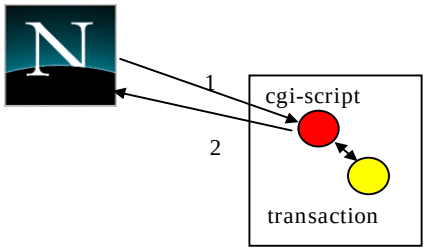

Figure 1: Transactions through cgi-scripts.

In a failure free environment, this mechanism works well, with transactions 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, resulting in work at the server not being atomic with respect to any browser related work. 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, its loss 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.

With the advent of Java it is possible to empower browsers so that they can fully participate within transactional applications [3]. To be widely applicable any such transaction system must comply with appropriate standards. The most widely accepted standard for distributed transactions is the Object Transaction Service (OTS) from the OMG [4]. However, requiring a browser to incorporate a full transaction processing system for all applications would be inefficient, and impose an overhead on all users. Therefore, in this paper we shall show how the interfaces defined by the OTS can be used to provide a lightweight solution to these end-to-end requirements which does not require the browser to be transactional.

We believe that this technique will be generally useful within a Web environment, and in particular for electronic commerce applications, where the back-office application may be debiting bank accounts or arranging the transfer of funds between organisations, before delivering purchased goods (such as the current issue of an on-line magazine, or a new browser plug-in) to the user's browser. Using the OTS framework allows the browser to be incorporated within existing OTS applications which may be operating within the back-office systems. Based upon this work we have developed a toolkit for constructing applications which employ Web transactions, and we shall illustrate this with an example.

2. The Object Transaction Service

The most widely accepted standard for distributed objects is the Common Object Request Broker Architecture (CORBA) from the Object Management Group (OMG) [5]. 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, which supports the well known concept of ACID transactions:

- *Atomicity*: The transaction completes successfully (commits) or if it fails (aborts) all of its effects are undone (rolled back).
- *Consistency*: Transactions produce consistent results and preserve application specific invariants.
- *Isolation*: Intermediate states produced while a transaction is executing are not visible to others. Furthermore transactions appear to execute serially, even if they are actually executed concurrently.
- *Durability*: The effects of a committed transaction are never lost (except by a catastrophic failure).

The OTS provides interfaces that allow multiple distributed objects to cooperate in a transaction such that all objects commit or abort their changes together. Transactions can optionally be nested to improve fault-isolation. 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. Objects supporting (partial) transactional behaviour must have interfaces derived from the `TransactionalObject` interface. The relevant parts of the OTS module are shown below:

```
module CosTransactions
{
  enum Vote { VoteCommit, VoteRollback, VoteReadOnly };

  interface Resource
  {
    Vote prepare () raises (HeuristicMixed, HeuristicHazard);
    void rollback () raises (HeuristicCommit, HeuristicMixed, HeuristicHazard);
    void commit () raises (NotPrepared, HeuristicRollback, HeuristicMixed,
                           HeuristicHazard);
    void commit_one_phase () raises (HeuristicHazard);
    void forget ();
  };

  interface SubtransactionAwareResource:Resource
  {
    void commit_subtransaction (in Coordinator parent);
    void rollback_subtransaction ();
  };

  interface Coordinator
  {
    RecoveryCoodinator register_resource (in Resource r) raises (Inactive);
    void register_subtran_aware (in SubtransactionAwareResource r) raises (Inactive,
                                                                            NotSubtransaction);
    . . . .
```

```
  };

  interface Control
  {
    . . . .
    Coordinator get_coordinator () raises (Unavailable);
  };

  interface Current : CORBA::ORB::Current
  {
    void begin () raises (SubtransactionsUnavailable);
    void commit (in boolean report_heuristics) raises (NoTransaction,
                                                       HeuristicMixed, HeuristicHazard);
    void rollback () raises (NoTransaction);
    void rollback_only ()raises (NoTransaction);
    Control get_control () ;
    . . . .
  };

  interface TransactionalObject
  {
  };

  . . . .
};
```

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. This is achieved be registering appropriate objects that support the `Resource` interface (or the derived `SubtransactionAwareResource` interface) with the current transaction. Registration is supported by the `Coordinator` interface. A `Coordinator` will either be passed as a parameter (if explicit propagation is being used) or may be retrieved using operations on the `Current` interface (if implicit propagation is used), which is the preferred way of interacting with the transaction manager.

The OTS uses a two-phase commit protocol to complete a top-level transaction by invoking operations on `Resources` registered with it. These `Resources` are responsible for managing any transactional state changes, lock propagation etc. on behalf of a specific recoverable object in order to guarantee its ACID properties. For example, when the `prepare` method is called on a `Resource` it must make any state changes to the recoverable object provisionally durable, since a subsequent `Resource` may force the transaction to abort. Importantly, the OTS specification does not define any `Resource` implementations: these must be provided by the application programmer or the specific OTS implementer. As we shall show, these objects will typically be specific to the application for which they have been created.

A 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 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.

Finally, 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.
- *Concurrency Control Service*. Required to support the isolation properties.

The application programmer is responsible for using appropriate services to ensure that transactional objects have the necessary ACID properties.

3. End-to-end transactional guarantees

As illustrated in figure 2, one straightforward way to provide end-to-end transactional guarantees for a Web application and its users would be to empower the browser (e.g., through Java) and incorporate into it an OTS implementation, along with the necessary persistence and concurrency control services. Application specific transactional objects could then be constructed and downloaded into the browser on demand. These objects would be able to participate directly in the application's transactions, i.e., the browser would appear simply as another address space for the application and its objects. For instance, the newspaper example could place a transactional object within the browser which has operations for adding and removing the cookie, and enforcing appropriate concurrency control strategies.

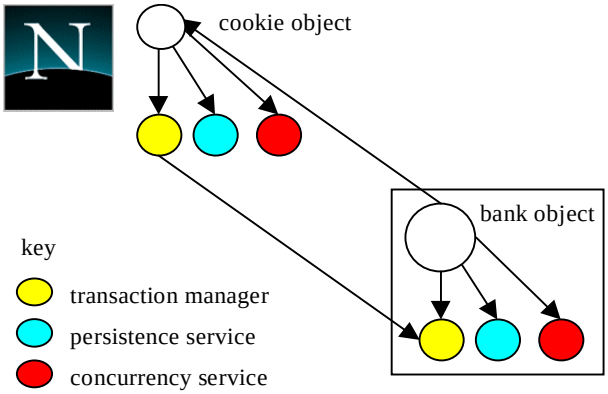

Figure 2: Transactional browser.

Although this may be acceptable for some applications, we are interested in the case where it would be considered either undesirable or impractical, e.g., because of the overhead and security implications involved in incorporating the required CORBA services within a browser. In addition, we expect that the majority of an application's resources, particularly in the area of electronic commerce, will continue to reside within protected domains, e.g., Web servers and back-office intranets.

Our aim, therefore, is to allow the construction of transactional applications with end-to-end guarantees without requiring (substantial) support from the browser; the level of browser support will depend upon the application, but should be minimal, e.g., simply writing a cookie to the user's disk. We want to allow the application objects and the services they require to remain at the Web server, but allow any necessary browser changes to be performed within the scope of, and therefore managed by, the server's transactions. Figure 3 shows such a lightweight transactional bank account: the application objects reside within the server, and the only addition to the browser is a proxy object for the cookie object (which can be downloaded on demand). This proxy will be responsible for making the cookie persistent should the transaction running at the server commit; the server is responsible for ensuring the browser changes are transactional in the presence of failures.

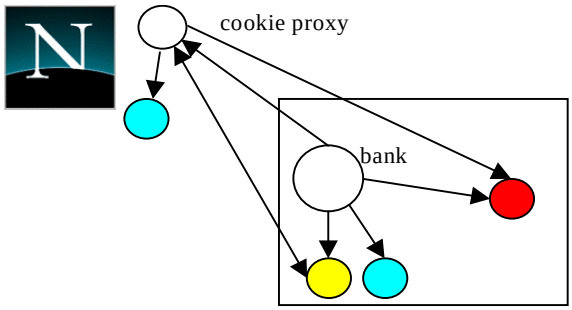

Figure 3: Lightweight transactional browser.

As we shall show, the `Resource` interface offers a convenient way to do this within the OTS: recall that the specification allows transactional objects to delegate their ACID responsibilities to recoverable objects (`Resources`); through a similar technique it is possible to encapsulate arbitrary non-transactional resources (e.g., a Web browser) within appropriate OTS `Resources`, such that they assume the responsibility of ensuring the object is transactional.

For example, consider the case of a non-transactional file system which is required to store the states of durable transactional objects. We shall assume that an appropriate concurrency control mechanism exists for ensuring the isolation properties; therefore, the persistence `Resource` can be written assuming that no conflicts can occur between competing transactions. If we assume the file system has the ability to atomically rename files, then the OTS `Resource` shown below (in pseudo-code) is sufficient to ensure the file system can be used in a transactional manner. (For simplicity we shall ignore crash recovery conditions and most error checking.)

```
Vote FileSystemResource::prepare ()
{
    // must ensure that newStateName is not
    // in use, so generate a unique name.

    if (writeObjectState(newStateName))
        return CosTransactions::VoteCommit;
    else
```

```
        return CosTransactions::VoteRollback;
}
```

The `prepare` method attempts to write the updated state to a temporary location within the file system (e.g., a file with a different name which is guaranteed to be unique). The `prepare` operation should perform sufficient error checking to ensure that a subsequent invocation of `commit` will not fail.

At this stage the transaction could still abort, so this must be a provisional update. The implementations of `commit` and `rollback` either rename the new file to have the same name as the old file (effectively overwriting the old state with the new state), or remove the new file from the file system, respectively. If the `commit` operation fails, e.g., because it cannot rename the file, then it can raise an exceptional condition to the transaction coordinator which can then rely upon failure recovery mechanisms to complete the transaction. The continued inability to terminate a committed transaction may require manual intervention.

```
void FileSystemResource::commit ()
{
    // must be an atomic "move"
    if !renameFromTo(newStateName,oldStateName))
        throw ExceptionalCondition;
}

void FileSystemResource::rollback ()
{
    removeObjectState(newStateName);
}
```

As can be seen, the file system `Resource` is responsible for interacting with the transaction manager and driving the underlying file system appropriately, depending upon where in the commit/abort protocol the transaction is. The `Resource` is also responsible for crash recovery, which is not shown. Therefore, in this situation there is no modification of the file system interface or functionality required to support transactions [6]. Other non-transactional objects can be encapsulated within `Resources` in a similar manner (e.g., legacy databases); the `Resource` for each object will be specific to that type of object.

3.1 Browser resources

The solution we propose to the problem of providing lightweight end-to-end transactional guarantees is through the use of `Resource` implementations, which can "wrap" the non-transactional browser, in a similar manner to the file system example described in the previous section. These `Resources` are registered with, and driven by, the transaction manager at the application server on behalf of the browser, and work with the browser-side application (assumed to be written in Java for the purposes of this discussion) to make it transactional. This allows the browser portion of the application to be extremely simple and lightweight.

The `Resource` at the server must communicate with the browser applet in order to drive it through the commit protocol; this requires the applet to support an object with which the `Resource` can communicate during `prepare`, `commit`, or `rollback`. It is this object, which we shall call the *resource-proxy*, which is responsible for making the browser transactional, e.g., storing the cookie received from the server-side `Resource`. In the rest of the paper we shall not differentiate between the resource-proxy and browser application, and shall use the terms interchangeably.

Although the `Resource` required to encapsulate the browser within a transaction will typically be specific to the application for which it was created, the functional requirements all such browser-`Resources` must fulfil will be the same. We shall now enumerate them in terms of the phases of the transaction commit protocol.

- *prepare*: the `Resource` should transmit to the browser application the identity of the transaction it has been registered with, which the user (or crash recovery mechanisms) can use to replay the transaction in the event of a failure. This identifier should be recorded either by the browser or the user. If the `Resource` passes any results back to the browser application (e.g., a cookie) it should not use them at this stage, since the transaction has not terminated. (Otherwise a situation similar to using cgi-scripts could arise.) If the browser does not respond to this message, the server-side `Resource` may assume it has failed, and cause the transaction to abort (by returning `VoteRollback` from `prepare`.) If the transaction aborts, *all* of the work performed within its scope will be undone. This enables the application to detect browser failures as late as possible, while still being able to undo any other work performed within the transaction at the server.

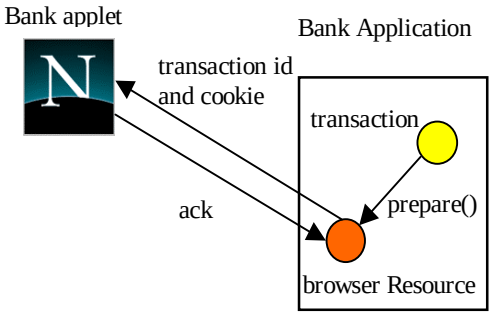

Figure 4: Prepare phase.

- *commit*: the application is guaranteed that all of the work performed within the transaction will be made durable, despite failures. Crash recovery mechanisms can be relied upon to complete partial transactions arising due to failures. Therefore, the `Resource` can make any "state changes" permanent, e.g., by instructing the browser to store the cookie, or display/decrypt the purchased document. Failure of the browser to respond to this invocation will require intervention by crash recovery.

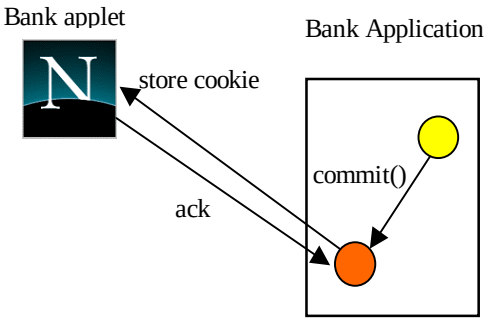

Figure 5: Commit phase.

- *rollback*: the `Resource` transmits the reason for the rollback to the browser, e.g., insufficient funds within the user's account.

The OTS uses a "presumed abort" protocol. If the browser fails prior to preparing, the transaction will rollback, undoing any state changes which occurred. No record of the transaction will be kept. If the transaction manager does not have any record of the transaction when asked to recover, it means that the transaction aborted.

If the browser fails after responding successfully during the prepare phase, but before it is told the transaction outcome, the user will have to rely upon crash recovery mechanisms to replay the transaction. If the browser obtained the transaction's identity as part of the prepare protocol, this can be resubmitted to the application upon recovery.

If the server fails and the transaction decision was to commit, then upon recovery it may attempt to contact the browser or simply wait until the browser reconnects, to finish the transaction. If the transaction aborted, then no crash recovery mechanisms are required to be used.

This technique provides a lightweight means for Web applications to gain end-to-end transactional integrity. The transaction coordinator, and the majority of the transaction infrastructure, reside within the server, while the browser application remains relatively simple, needing only to be able to make any "changes" permanent in the case of a commit, or to undo them if the transaction aborts.

3.2 Impact of using Java

In order to provide end-to-end transactional guarantees we require application specific control within the browser. Because of its portability we have chosen Java for this role, although ActiveX, or a suitable browser plug-in, could have been used.

Java security is imposed by a SecurityManager object, which defines what a program can, and cannot do [7]. In a stand alone environment, security can be relaxed. However, for a browser the constraints imposed by SecurityManagers can directly affect the types of applications we have been describing. Generally a browser program

downloaded over the network 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.

Some browsers allow the user to specify their own security restrictions (e.g., to allow programs to write to specific locations of a file system), or simply relax the security restrictions applied to applets, but others do not. Modifications to the language have been proposed which will make application specific customisation of security policies more generally available [11]. However, currently we may require the Java application within the browser to be customised for each type of browser, as described in [8].

In the following sections we shall illustrate this lightweight Web transaction technique with an example. However, in order to do this we must first describe the implementation of the OTS which we have used.

4. Overview of an OTS implementation

We have designed and implemented *OTSArjuna*, a C++ transaction toolkit which complies with the OTS specification [9]. Java classes which provide access to the OTS manager and its resources exist, which enable the construction of lightweight transactional Java applications.

Rather than require programmers to make use of the low-level OTS, concurrency control and persistence interfaces, we have designed a high-level API for building transactional applications and frameworks [9][10]. This API automates the activities concerned with participating within an OTS transaction, such as creating and registering appropriate `Resource` implementations, propagating locks etc. The architecture of the system is shown in figure 6.

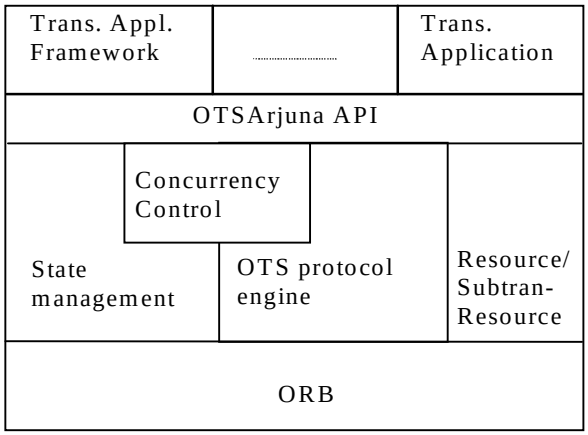

Figure 6: OTSArjuna structure.

The OTSArjuna model for building transactional applications exploits object-oriented techniques to present programmers with a toolkit of C++ classes from which application classes can inherit to obtain desired properties, such as persistence and concurrency control. Each class is concerned with a single functionality, and these classes form a hierarchy, part of which is shown in figure 7.

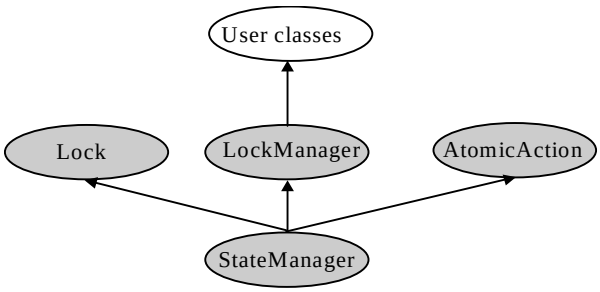

Figure 7: OTSArjuna class hierarchy.

By inheriting from `LockManager`, user classes are automatically transactional, with `LockManager` and `StateManager` being responsible for guaranteeing the ACID properties. Apart from specifying the scopes of transactions, and setting appropriate locks within objects, the application programmer does not have any other responsibilities: the system guarantees that appropriate `Resource` objects are registered with transactions, and that in the event of failures crash recovery mechanisms are invoked automatically.

4.1 Saving object states

OTSArjuna 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 class `ObjectState`. The class maintains an internal array into which instances of the standard types can be contiguously packed (unpacked) using appropriate overloaded `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.

4.2 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:

```
class StateManager
{
public:
  Boolean activate ();
  Boolean deactivate ();

  Uid get_uid () const; // the object's unique
                        // identifier.

  // methods to be provided by a derived class

  virtual Boolean restore_state(ObjectState&)=0;
  virtual Boolean save_state(ObjectState&)=0;

protected:
  StateManager ();
  StateManager (const Uid& id);
};
```

Objects are assumed to be of two 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. Alternatively, 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. Persistent objects are assigned unique identifiers (instances of the Uid class), when they are created, and this is used to identify them within the object store [9].

`StateManager` invokes the operations `save_state` (while performing `deactivate`), and `restore_state` (while performing `activate`) at various points during the execution of the application. As their name implies, these methods are responsible for converting an object's state to/from a format which can be stored in a persistent object store. These operations *must* be implemented by the programmer since `StateManager` cannot detect user level state changes. 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:

```
Boolean Example::save_state (ObjectState& os)
{
  return (os.pack(A)&&os.pack(B)&&os.pack(C));
}
```

4.3 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. The primary programmer interface to the concurrency controller is via the `setlock` operation.

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

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.

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. If the transaction were nested, `LockManager` would automatically propagate locks to the transaction's parent when it commits.

The code below shows how we might try to obtain a write lock on an object derived from `LockManager` within a transaction:

```
Boolean Example::foobar ()
{
  Boolean result = FALSE;

  // start a new transaction

  OTS::get_current().begin();

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

    // automatically aborts if commit fails

    OTS::get_current().commit(TRUE);
    result = TRUE;
  }
  else
    OTS::get_current().rollback();

  return result;
}
```

4.4 Web transactions

Although we have implemented the necessary `Resources` for OTSArjuna classes which encapsulate the persistence and concurrency control services, this does not currently extend to Web transactions and the types of browser resources described in Section 3. Therefore, at present application programmers must construct their own `Resources` and are also responsible for registering them with the transaction at the correct time. However, we are currently building additional tools and classes which will automate much of this work.

5. Bank cashpoint example

To illustrate the technique of incorporating browsers into server-side transactions, we have implemented a variation of the bank account example used throughout this paper. The example is available on-line (http://arjuna.ncl.ac.uk/).

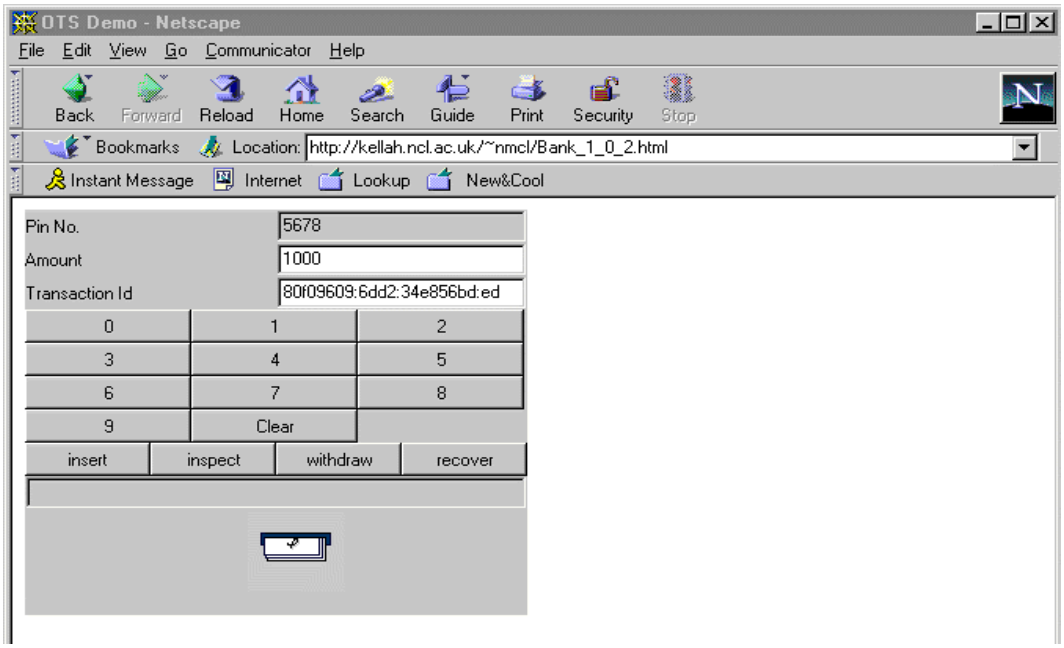

Figure 8: Bank applet.

Consider the case of an on-line bank which allows users to inspect, remove, and insert money from their accounts; the bank runs on an ORB which resides on a Web server, and the transaction manager runs at the same location. When money is removed it is converted into *digital cash tokens* which are stored and manipulated by the browser, and contain the token's current "cash" balance; we shall assume that these tokens can be presented to other Web-commerce applications as payment for services. Insertion of money to an account is simply the reverse, whereby a token is consumed, and the user's account is credited by the amount left within the token. The bank (and user) wish to make the delivery of the cash token to the user's browser *and* the debiting of the user's account atomic (and similarly for crediting the account). Failures must not result in inconsistencies at the browser or the bank.

The browser portion of the application, shown in figure 8, consists of a Java applet which displays a graphical representation of the bank; use accounts are accessed via a PIN. In addition, the applet contains the resource-proxy which the server-side of the application will use during transaction termination, and a client stub for the bank server (the *bank proxy*) which has a method for each of the bank's operations. The relationship between the applet, the bank proxy and the server-side application is illustrated in figure 9.

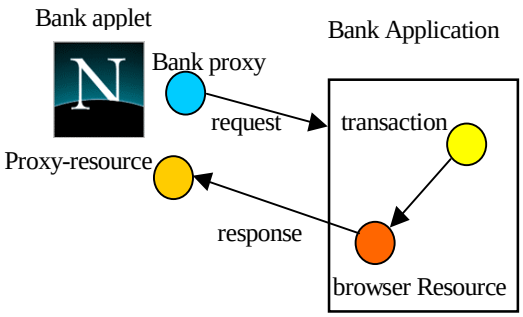

Figure 9: Applet/server interaction.

To guarantee consistency in the presence of failures, each operation is performed within the scope of a transaction. Although the operation to inspect an account is transactional, it only reads the state of the account and therefore there is no actual requirement for end-to-end transactional integrity: if a failure occurs, the user can simply re-issue the request. However, both withdrawing and inserting money require stronger transactional guarantees: each operation must atomically modify the account *and* either deposit or "consume" a digital cash token to/from the browser.

During the prepare phase of each transaction the applet will display the unique transaction identifier for the operation. During the commit phase, the bank server will return the digital cash token to the browser and the applet can then use this to animate (e.g., money appears to be dispensed). In a real system the token would need to be

stored by the browser. If the transaction aborts, an error message will be displayed giving the reason, e.g., insufficient funds. In the case of insertion, the cookie will be consumed and removed from the user's disk.

For simplicity, in the rest of this discussion we shall only consider the implementation of the `withdraw` operation. Similar modifications to those to be mentioned for `withdraw` will apply to the `insert` operation.

5.1 Browser applet

The browser applet is mainly concerned with the graphical display and co-operating with the server-side browser `Resource` to ensure end-to-end guarantees, such as making persistent any state changes (e.g., the cash token). Whenever a user selects one of the bank options available through the applet interface, the applet initiates a connection to the bank server (via the bank proxy) for the duration of each request, and receives replies as part of the server-side transaction termination protocol (as shown in figure 9). Successful replies signify either the arrival or consumption of a cash token, and the applet animates accordingly.

5.2 Server application classes

To enable the bank to service multiple clients concurrently, each account is a separate transactional object, responsible for its own persistence and concurrency control. Therefore, concurrent modifications to different bank accounts will not interfere. Using OTSArjuna, each bank account is an instance of the `Account` class:

```
enum Outcome { DONE, NOTDONE, INSUFFICIENT_FUNDS, ACCOUNT_ERROR, LOCKED };

class Account : public LockManager
{
public:
  Account ();
  virtual ~Account ();

  void insert (int amount);
  void withdraw (int amount);
  void inspect (int& amount);

  virtual Boolean save_state (ObjectState& os);
  virtual Boolean restore_state (ObjectState& os);

private:
  int amount;
};
```

We shall first show the implementation of the `withdraw` method without browser participation. We will then describe the necessary browser `Resource` for end-to-end transactional guarantees, and return to the `withdraw` operation to show how it requires modification in order to incorporate this `Resource`.

The `withdraw` method first starts a new transaction and then attempts to obtain a lock on the account object. Because the `withdraw` operation will modify the state of the account, it tries to obtain an exclusive (write) lock. If the operation is performed successfully (e.g., there is sufficient funds in the account), the transaction is committed, otherwise it is rolled back.

```
void Account::withdraw (int money)
{
    Outcome result = NOTDONE;

    OTS::get_current().begin();

    if (setlock(new Lock(WRITE)) == GRANTED)
    {
      /*
       * Check whether the user has sufficient
       * money to withdraw.
       */

      if (amount >= money)
      {
          amount = amount - money;
          result = DONE;
      }
      else
        result = INSUFFICIENT_FUNDS;
    }
    else
        result = LOCKED;

    if (result == DONE)
    {
        OTS::get_current().commit(TRUE);
    }
    else
        OTS::get_current().rollback();
}
```

In order to involve the browser such that it can receive the cookie when the user's account is finally debited, we need to create an instance of a specific `Resource` which will encapsulate the browser for the duration of the transaction. The signature of the `BrowserResource` is shown below.

```
class BrowserResource : public CosTransactions::Resource
{
public:
  BrowserResource (const Uid& tran, int outcome, const CashToken& amount);
  virtual ~BrowserResource ();

  virtual Vote prepare  ();
  virtual void rollback ();
  virtual void commit ();
  virtual void forget ();
  virtual void commit_one_phase ();

private:
    Boolean saveState ();
    Boolean restoreState ();
    void removeState ();

    int opOutcome;
    CashToken cashAmount;
    Uid tid;
    Browser applet;  // our handle on the browser
                     // side of the application.
};
```

When created, each `BrowserResource` is given the identity of the transaction it has been registered with (its Uid), so that it can transmit it to the browser during the prepare phase; this can be used in the event of a failure. The `Resource` also receives the outcome of the operation and the cash token to send back to the browser. (In the case of the insert operation this will be a blank token.)

During transaction termination the bank sends replies to the browser applet's operation requests (i.e., to the bank proxy described earlier) via an instance of the `Browser` class, which has `prepare`, `commit`, and `rollback` operations, allowing any reply (including the cookie) to be returned to the browser. For simplicity the signatures of the token class and browser class are not shown.

To make the browser transactional, the `BrowserResource` must keep a durable record of its progress which can be used by crash recovery in the event of a failure. For example, consider the implementation of the `prepare` method:

```
Vote BrowserResource::prepare  ()
{
  if (!saveState())
    return CosTransactions::VoteRollback;

  // invoke operation on browser

  if (applet->prepare(tid))
    return CosTransactions::VoteCommit;
  else
  {
    // Failure - discard Resource's state

    removeState();
    return CosTransactions::VoteRollback;
  }
}
```

Before the `BrowserResource` sends the transaction's identity, it calls `saveState` which will make permanent sufficient information to allow its role in the transaction to be replayed in the event of a failure. If `saveState` fails, the transaction must abort.

If the subsequent `prepare` invocation on the browser fails, the resource assumes that it has failed, removes any state it previously saved, and forces the transaction to rollback. If no errors occur during `prepare` the resource informs the transaction coordinator that it is ready to commit.

During the commit phase, the `Resource` sends the token to the browser and awaits an acknowledgement. If no acknowledgement is received, or an exceptional response is returned (e.g., the connection to the browser fails), the `Resource` relies upon crash recovery mechanisms to complete its work.

```
void BrowserResource::commit ()
{
  // browser acknowledged receipt

  if (applet->commit(cashAmount))
  {
    removeState();
  }
  else
  {
    // Some failure has occurred (e.g., browser,
    // network). Crash recovery will have to
    // deal with this.
  }
}
```

In the event of a rollback the resource will send an error message to the browser indicating the reason for the failure.

```
void BrowserResource::rollback ()
{
    removeState();

    /*
     * Send reason for failure. If this fails,
     * the browser will try again later, and
     * be told the transaction rolled back.
     */

    applet->rollback(reasonForFailure);
}
```

If a failure occurs during the transaction, the user can re-issue the request when the machine recovers by typing in the transaction identifier. The crash recovery mechanisms at the server will determine the transaction outcome and complete the work. If no record of the transaction can be found it can be assumed that the transaction aborted.

Having considered the implementation of the `BrowserResource`, we can return to the `withdraw` operation. We wish to incorporate the browser within the transaction *before* it terminates, such that the browser will be informed of the outcome regardless of whether the transaction subsequently commits or rolls back. Therefore, as shown below in italics, after performing the withdrawal we obtain a reference to the current transaction's `Coordinator`, and register a new instance of the `BrowserResource` with it. The transaction then either commits or aborts, depending upon whether the withdrawal was successful, and drives the `BrowserResource` to either deliver a token or an error

message, respectively. If a failure occurs, the user's account will not be debited *and* neither will the browser obtain a cash token.

```
void Account::withdraw (int money)
{
  Outcome result = NOTDONE;

  OTS::get_current().begin();

  if (setlock(new Lock(WRITE)) == GRANTED)
  {
    if (amount >= money)
    {
      amount = amount - money;
      result = DONE;
    }
    else
      result = INSUFFICIENT_FUNDS;
  }
  else
    result = LOCKED;

  /*
   * Now involve the browser within the
   * transaction.
   */

  Control_ptr c = OTS::get_current().get_control();
  Coordinator_ptr coord = c->get_coordinator();
  BrowserResource res = new BrowserResource(OTS::get_current(), result, money);

  coord->register_resource(res);

  if (result == DONE)
  {
    // automatically aborts if it cannot commit

    OTS::get_current().commit(TRUE);
  }
  else
  {
    // undo all work performed

    OTS::get_current().rollback();
  }
}
```

As can be seen by comparing the new `withdraw` with the old, apart from implementing the `Resource`, the changes to an application necessary to incorporate a non-transactional browser into its operations are minimal. Therefore, this technique also offers possibilities for integrating Web browsers within existing legacy applications.

5.3 Results

Although there is some run-time overhead involved in the extra work necessary to make the browser transactional, this does not appear to be noticeable to users. The benefits gained, however, noticeable in the presence of failures, where the system automatically completes operations upon recovery. However, we intend to investigate the overhead in order to improve the efficiency of our transaction implementation and reduce the run-time footprint of the browser applets.

6. Conclusions and future work

In this paper we have presented a model for using the OMG's OTS specification within the Web. Using OTS `Resources` we have shown how it is possible for browsers to participate within a transaction to obtain end-to-end transactional guarantees in a lightweight manner. The browser does not need to incorporate support for transaction processing. We have presented an implementation of the OTS, OTSArjuna, and shown how we have used it to implement an on-line bank account example with these end-to-end guarantees.

Currently application programmers are responsible for implementing and registering Web `Resources`; we intend to extend OTSArjuna to provide application building tools and classes which will ease the construction of such applications. In addition, we have implemented a Java version of OTSArjuna [9] which is compliant with the Enterprise JavaBeans specification [12]. This will allow fully transactional Web browsers to work with server-side applications.

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

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

[2] "Transarc DE-Light Web Client Technical Description", Transarc Corporation, February 1996.

[3] M. C. Little et al, "Constructing Reliable Web Applications Using Atomic Actions", Proceedings of the 6th Web Conference, April 1997, pp. 561-571.

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

[5] "Common Object Request Broker Architecture and Specification", OMG Document Number 91.12.1.

[6] G. D. Parrington, S. K. Shrivastava, S. M. Wheater and M. C. Little, "The Design and Implementation of Arjuna", USENIX Computing Systems Journal, Vol. 8, No. 3, pp. 253-306, Summer 1995.

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

[8] M. C. Little and S. M. Wheater, "Building Configurable Applications in Java", Proceedings of the 4th International Conference on Configurable Distributed Systems, Annapolis, Maryland, May 1998, pp. 172-179.

[9] M. C. Little and S. K. Shrivastava, "Java Transactions for the Internet", Proceedings of the 4th Usenix Conference on Object-Oriented Technologies and Systems, Santa Fe, New Mexico, April 1998, pp. 89-100.

[10] 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.

[11] Li Gong et al, "Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2", Proceedings of the Usenix Symposium on Internet Technologies and Systems, Monterey, California, December 1997, pp. 103-112.

[12] Vlada Matena and Mark Harper, "Enterprise JavaBeans", Sun Microsystems draft specification version 0.9, December 1997.