Constructing Reliable Web Applications using Atomic Actions

M. C. Little, S. K. Shrivastava, S. J. Caughey, and D. B. Ingham

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 actions* (*atomic transactions*) for controlling operations on services. Atomic actions guarantee the consistency of applications despite concurrent accesses and failures. Techniques for implementing transactions on distributed objects are well-known: in order to become "transaction aware", an object requires facilities for concurrency control, persistence, and the ability to participate in a commit protocol. While it is possible to make server-side applications transactional, browsers typically do not possess such facilities, a situation which is likely to persist for the foreseeable future. Therefore, the browser will not normally be able to take part in transactional applications. The paper presents a design and implementation of a scheme that does permit non-transactional browsers to participate in transactional applications, thereby providing much needed end-to-end transactional guarantees.

Keywords: World Wide Web, atomic actions, transactional integrity, consistency, fault-tolerance, distributed systems

1. Introduction

The Web is rapidly being populated by service providers who wish to sell their products to a large potential customer base. However, there are still important security and fault-tolerance considerations which must be addressed. One of these is the fact that the Web frequently suffers from failures which can affect both the performance and consistency of applications run over it. For example, if a user purchases a cookie granting access to a newspaper site, it is important that the cookie is delivered and stored if the user's account is debited; a failure could prevent either from occurring, and leave the system in an indeterminate state. For resources such as documents, failures may simply be annoying to users; for commercial services, they can result in loss of revenue and credibility. Therefore, making Web applications fault-tolerant is important.

Atomic actions (*atomic transactions*) are a well-known technique for guaranteeing application consistency in the presence of failures [DBL77][OMG95]. An atomic action guarantees that, despite failures, either all of the work conducted within its scope will be performed or it will all be undone. This is an extremely useful fault-tolerance technique, especially when multiple, possibly remote, transactional resources are involved. For this reason Web applications already exist which offer transactional guarantees to users. However, because significant resources are required to provide transactional properties, these guarantees only extend to resources used at Web servers, or between servers (see figure 1); browsers are not included, despite being a significant source of unreliability in the Web. Providing end-to-end transactional integrity between the browser and the application is important, however: in the previous example, the cookie *must* be delivered once the user's account has been debited. As we shall show later, using 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 successfully [OSF96].

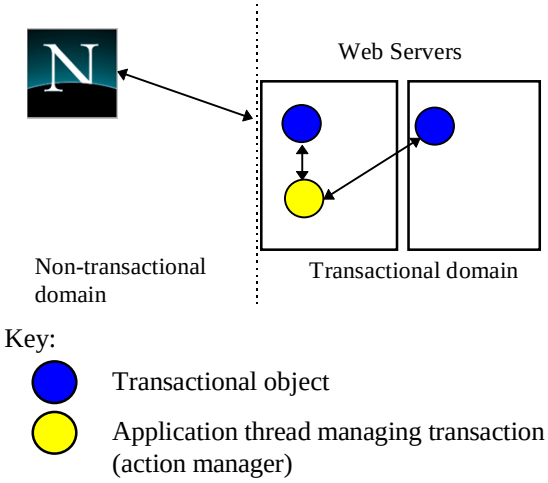

Figure 1: Current transactional Web applications.

Despite the advent of technologies such as Java, browsers are typically resource scarce; requiring a browser to incorporate support for transactions for all applications would be inefficient, and impose an overhead on all users. In this paper we shall present a lightweight solution for providing these end-to-end transactional guarantees which does not require the browser to be transactional. We shall also present an implementation of this model, the *W3OTrans* toolkit, which has been implemented on the *W3Objects* object-oriented framework [DBI95].

2. Atomic actions

Atomic actions are used in application programs to control the manipulation of persistent (long-lived) objects. Atomic actions have the following ACID properties [OMG95]:

- *Atomic*: if interrupted by failure, all effects are undone (rolled back).
- *Consistent*: the effects of a transaction preserve invariant properties.
- *Isolated*: a transaction's intermediate states are not visible to other transactions. Transactions appear to execute serially, even if they are performed concurrently.
- *Durable*: the effects of a completed transaction are persistent; they are never lost (except in a catastrophic failure).

A transaction can be terminated in two ways: committed or aborted (rolled back). When a transaction is committed, all changes made within it are made durable (forced on to stable storage, e.g., disk). When a transaction is aborted, all of the changes are undone. Atomic actions can also be nested; the effects of a nested action are provisional upon the commit/abort of the outermost (top-level) atomic action.

2.1 Commit protocol

A two-phase commit protocol is required to guarantee that all of the action participants either commit or abort any changes made. Figure 2 illustrates the main aspects of the commit protocol: during phase 1, the action coordinator, C, attempts to communicate with all of the action participants, A and B, to determine whether they will commit or abort. An abort reply from any participant acts as a veto, causing the entire action to abort. Based upon these (lack of) responses, the coordinator arrives at the decision of whether to commit or abort the action. If the action will commit, the coordinator records this decision on stable storage, and the protocol enters phase 2, where the coordinator forces the participants to carry out the decision. The coordinator also informs the participants if the action aborts.

When each participant receives the coordinator's phase 1 message, they record sufficient information on stable storage to either commit or abort changes made during the action. After returning the phase 1 response, each participant which returned a commit response *must* remain blocked until it has received the coordinator's phase 2 message. Until they receive this message, these resources are unavailable for use by other actions. If

the coordinator fails before delivery of this message, these resources remain blocked. However, if crashed machines eventually recover, crash recovery mechanisms can be employed to unblock the protocol and terminate the action.

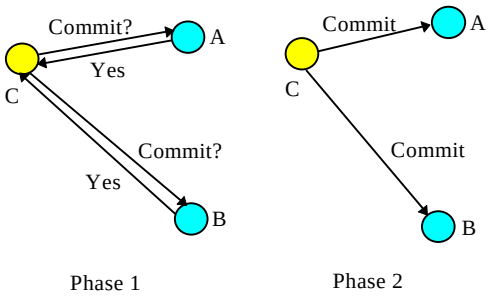

Figure 2: Two-phase commit protocol.

2.2 Transactional proxies

The action coordinator maintains a *transaction context* where resources taking part in the action are required to be registered. Such a resource must obey the transaction commit protocol guaranteeing ACID properties; typically this means that the resource will provide specific operations which the action can invoke during the commit/abort protocol. However, it may not be possible to make all resources transactional in this way, e.g., legacy code which cannot be modified. To use these resources within an action it is often possible to provide *transactional proxies*: the proxy is registered with, and manipulated by, the action as though it were a transactional resource, and the proxy performs implementation specific work to make the resource it represents transactional. This requires that the proxy participate within the commit/abort protocol. Because the work of the proxy is performed as part of the action, it is guaranteed to be completed or undone despite failures of the action coordinator or action participants.

How proxies are created and registered with an action will depend upon specifics of the transactional system, (some systems may not support proxies). For example, a system based on reflective techniques may use an appropriate meta-object protocol [RJS95]. Another technique would be through the use of *open commit protocols*, which give programmers the ability to control what an action participant does during *each* phase of the protocol; such a protocol can also be used to register resources which are not transactional but which can be driven by the commit/abort protocol [OMG95].

2.3 Transactional gateways and browser proxies

One straightforward way to provide end-to-end transactional integrity for a Web application and its users would be to empower the browser and incorporate into it transactional objects which are part of the application. These objects would then be able to participate directly in the application's transactions. For instance, the newspaper example could place a transactional object within the browser which has operations for adding and removing the cookie. However, this then requires the browser to support transactional objects and (distributed) atomic actions. 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 security considerations.

For such cases, a practical means of obtaining transactional guarantees between the application and browser will be through the use of *transactional gateways* and *transactional browser proxies*. A transactional gateway provides browsers with access to the transactional application by exporting application specific operations which can be invoked from the browser, e.g., an operation to purchase the newspaper. Importantly, an application may be accessed via many gateways: each gateway is configured to be used by browsers with specific functionality. For example, a gateway for transactional aware browsers may download transactional application objects to it; however, if the browser is not transactional, the gateway may be responsible for creating a transactional browser proxy, which is then responsible for cooperating with the browser to make it transactional, as described below. The gateway can also be made responsible for enforcing security, such as authenticating the user before allowing operations to be performed.

A transactional browser proxy consists of code which resides within the browser and code which resides in the gateway. The proxy is used to allow a non-transactional browser to participate within an atomic action. The gateway receives a request from a browser, starts an atomic action and registers a browser proxy with it. The gateway then performs the operation, gives the result to the proxy, and depending upon the outcome of the operation, either commits or aborts the action. During the commit/abort protocol the proxy delivers the response to the browser, which must acknowledge receipt. For example, in the case of the newspaper example described earlier, the proxy can transmit the cookie to the browser during phase 2 of the protocol, when the action has guaranteed that all other changes have been committed, and the application can be sure that it will be delivered despite failures. If the action aborts, the proxy can transmit an error message, giving the reason for the failure.

The server-side of the proxy performs most of the transactional work, whereas the browser code is relatively simple, needing only to be able to make any "changes" permanent in the case of a commit, or to undo them if the action aborts. This will typically be application specific, e.g., for a cookie, the browser-side proxy will need to write it to disk, whereas if the user is purchasing a specific edition of an electronic newspaper the proxy may receive the newspaper during phase 1, and either display it during phase 2 or discard it, depending upon whether the action commits or aborts. Therefore, transactional gateways and browser proxies provide a lightweight means for applications to gain end-to-end transactional integrity.

A browser proxy can also support application specific processing; we give two examples below:

- if the browser is invoked during phase 1 of the commit protocol and does not respond (e.g., to a simple "are-you-alive?" message), the proxy can cause the action to abort. If the action aborts, all of the other work performed within the action will be automatically 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 action, such as aborting a debit of the user's account. If this did not occur, resources may remain unavailable until crash recovery mechanisms complete the transaction.
- Rather than the simple "are-you-alive?" protocol, a more complex protocol could be used. For example, if the browser is executing on a mobile machine which is about to shut down due to low battery power, it could return the address of another agent during phase 1, to which the action coordinator should redirect the phase 2 outcome.

Figure 3 shows two transactional gateways, A and B, for the same transactional newspaper application: users purchase cookies, and their payments are debited from a bank account which is held elsewhere, possibly by a different organisation. A purchase operation is required to atomically debit an account *and* deliver the purchased cookie to the user. Therefore, the transaction spans multiple domains and transactional objects. Transaction aware browsers use gateway B, whereas A is used by non-transactional browsers. As illustrated, prior to invoking the operation on the newspaper object, A creates a browser proxy which is registered with the atomic action.

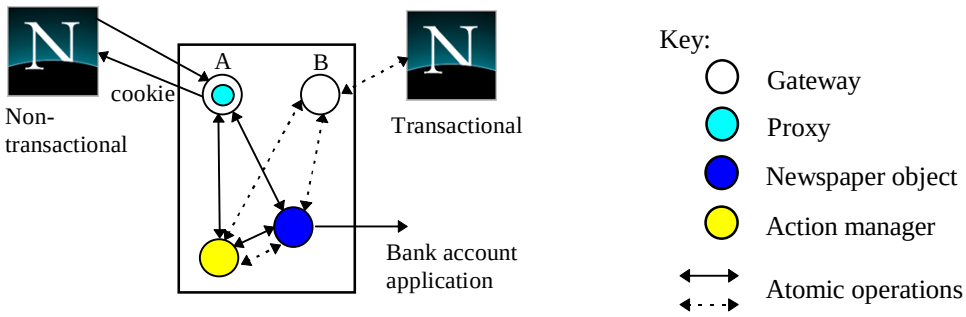

Figure 3: Transactional gateways.

During the second phase of the commit protocol, the proxy transmits the cookie to the browser, which must acknowledge after storing it on disk; if a failure of either the browser or server occurs, the cookie will be retransmitted by the crash recovery mechanisms. Apart from starting and ending transactions, the application does not have to perform additional work in order to guarantee that the user's request will be performed despite

failures. (In the case of the transactional browser, the cookie is simply the state of one of the transactional objects within the browser which were downloaded by B.)

Another advantage of transactional gateways and browser proxies is that they can be used to support the construction of modular applications. The same transactional application can be accessed from the Web and from other (distributed) environments, only the gateway need change. For example, in figure 4 the same application has been made available to Web users and ORB clients [OMG91]. Each gateway is responsible for the protocol aspects of interfacing the application with the specific distributed environment.

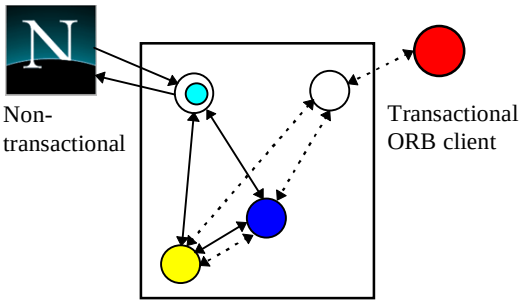

Figure 4: Inter-working.

Having presented a model for providing lightweight end-to-end transactional guarantees for Web applications, we shall now describe a system which implements it.

3. W3OTrans

We have developed the W3OTrans toolkit which provides programmers with the ability to implement Web applications using atomic actions and objects. Although we shall concentrate on the support for building transactional gateways and browser proxies, W3OTrans also supports the following types of transactional application:

(i) the application executes entirely at the Web server, as the result of the browsing user selecting a specific URL.

(ii) the application executes entirely within the browser.

(iii) the application executes within the browser and the Web server; the server may contain the more security-sensitive resources, for example. In this case, the browser is simply another address space for the application.

Using W3OTrans, programmers can build transactional applications which span multiple Web servers, multiple browsers, or execute locally within a single browser. In addition, transactions can span browser invocations, and can be arbitrarily nested. In order to do this, W3OTrans is implemented in C++ and Java, enabling server-side applications to take full advantage of more efficient compilation techniques. However, since all of the W3OTrans facilities are available in both languages, applications can be written entirely in Java.

3.1 Impact of the Java security model

The use of Java to implement the browser-side version of W3OTrans raises some important security issues. Java security is imposed by a SecurityManager object, which defines what an applet can, and cannot do [SUN95][ORA96]. Although it is believed that users will eventually be able to provide their own SecurityManager, currently the Java interpreter provides the only implementation. As described in [ORA96], generally an applet cannot remotely communicate with a node other than the one from which it was loaded. In addition, an applet cannot write to the disk of the machine on which it is being run. If an applet is loaded directly from the user's disk then many of these restrictions are relaxed.

However, the problem for Java programmers is that each implementer of a Java interpreter can provide a different security model and SecurityManager object, which may impose different constraints. Therefore, to enable applications written with W3OTrans to execute in any Java-aware Web browser, our two major premises were not to modify the language or the interpreter. Services such as persistence and concurrency control are accessed through interfaces which do not specify an implementation, allowing objects to determine the required implementation at run-time. For example, an object written using W3OTrans can choose implementations for persistence based upon the security restrictions of the Java interpreter, the user's requirements, and the application, without modifications to the application. For example, where the SecurityManager prevents applets from accessing the local disk for persistence, a remote persistence service can be used.

3.2 W3OTrans classes

W3OTrans is built on the W3Objects system [DBI95][DBI96]. W3Objects is a framework for the construction of Web applications using object-oriented techniques. W3Objects are responsible for their own persistence, concurrency control, fault-tolerance, etc., which are provided through appropriate classes and libraries. In addition, every W3Object has the ability to guarantee referential integrity: the object can be prevented from being deleted as long as a single reference to it exists. W3Objects are available to standard Web browsers using the HTTP protocol, and can also possess interfaces which enable them to be invoked using other protocols.

In keeping with the W3Objects methodology, objects obtain ACID properties through inheritance. W3OTrans presents programmers with an AtomicObject base class and the AtomicAction class. The AtomicObject class is responsible for supporting the properties of serialisability, failure atomicity and permanence of effect. Concurrency control can be applied on a per object basis, and uses a multiple reader/single writer policy. Each derived class must provide methods to convert an object's state to/from a format which can be saved to stable storage. Therefore, the programmer has explicit control over which objects, and which parts of an object, are transactional.

The AtomicAction class is used to create atomic actions. An action is automatically nested if it is created within the scope of another action. Whenever an AtomicObject is used within an action it is automatically registered with that action, i.e., it becomes part of the transactional context and will be manipulated by the action to guarantee ACID properties. If the object is in another address space (e.g., on a different machine), then the action will automatically span these address spaces. This occurs transparently to the application programmer: there is no special syntax required to ensure that transactional contexts cross address spaces.

3.3 W3OTrans gateways and proxies

When building applications which require end-to-end transactional integrity for the browser but which do not require, or cannot allow, the browser to contain transactional objects, the programmer can create application specific transactional gateways and browser proxies. After developing the transactional application using the W3OTrans classes, the programmer uses a high-level definition language to specify the required gateway interface, i.e., the operations which the application supports and wants to export to the Web. Using this interface definition, W3OTrans tools automatically generate the server-side transactional gateway and Java code which allows it to be invoked from a browser applet, along with C++ and Java implementations of the browser proxy. The gateway is represented in the applet as a Java class with one method for each operation; each method is responsible for communicating with the actual server-side gateway. Currently the proxy protocol is based upon a simple "are-you-alive?" message as described in section 2.3. The programmer then modifies the server-side gateway code to begin and end atomic actions, and register browser proxies. There is no necessity to consider possible failure scenarios and attempt to perform compensating work: the transactions will automatically guarantee consistency and integrity. At this stage, the programmer has a complete server-side transactional application, and can now concentrate on the browser applet, which will allow users to interact with the application through the gateway.

4. Bank ATM example

Having given an overview of the W3OTrans system, we shall now describe a bank account application which we have used it to build. The application presents Web users with the ability to insert, withdraw and inspect their accounts, which are accessed via a PIN. To preserve consistency in the presence of failures and concurrent users, the operations are performed as separate atomic actions. In order that the bank can service

concurrent write operations, each account is a separate transactional object. As shown in Figure 5, a Java applet displays a graphical representation of a bank ATM machine. When the user selects an operation, the applet displays a unique transaction identifier for this operation and then invokes the appropriate method on the bank gateway.

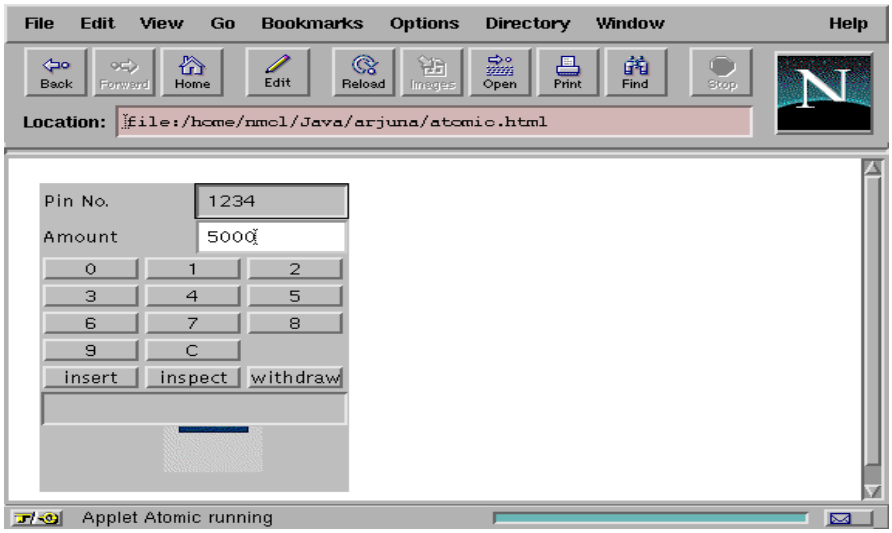

Figure 5: Bank applet.

Although the inspect operation is atomic, it only reads the state of the account and therefore there is no 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 cause the applet to animate either the withdrawing or inserting of money. This is accomplished by the bank returning a token during the commit protocol which indicates whether the operation committed or aborted; the applet uses this to either animate, or print an error message. A failure to perform the operation may be the rest of insufficient funds, for example, or that the bank application crashed before debiting the account. As shown in figure 6, during phase 1 of the commit protocol the proxy determines whether the browser is still available. If the browser responds *and* the attempt to debit the user's account can be made permanent, the protocol proceeds to phase 2 and commits the action, making permanent any changes. During this phase the browser proxy transmits the token to the applet, which dispenses the money and acknowledges receipt of the token.

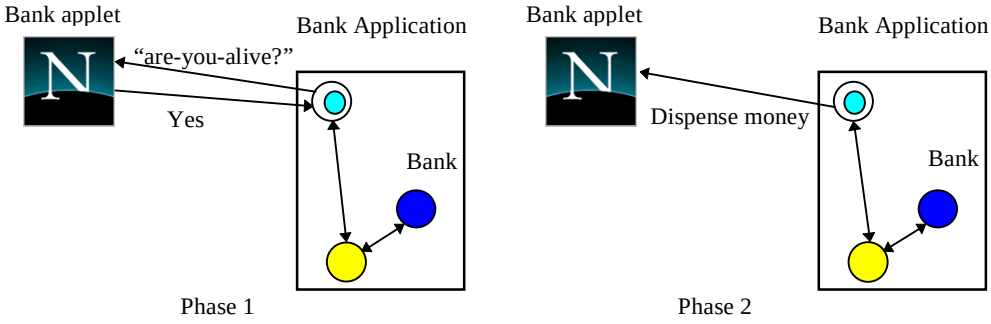

Figure 6: Bank Withdraw operation.

If the Web server crashes during the action, the user can re-issue the request when the machine recovers by typing in the transaction identifier. The crash recovery mechanism at the Web server will determine the transaction outcome and complete the action. If the browser crashes before the end of phase 1 it will cause the action to abort. If it crashes after the coordinator has decided to commit, crash recovery will be responsible for resolving the action when the browser recovers. Therefore, when withdrawing money, if the user's bank account is debited the application can guarantee that, regardless of failures, the money will be dispensed. The

programmer can develop the application without having to consider the possible failure scenarios and how to provide recovery mechanisms for them.

5. Comparisons with other systems

In this section we shall briefly consider other systems intended to integrate atomic actions and the Web.

5.1 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.

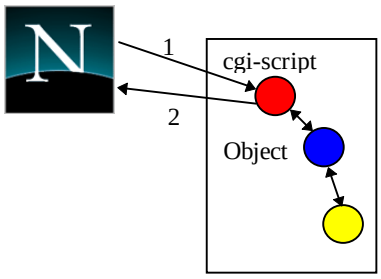

Figure 7: transactions through cgi-scripts

5.2 Transactions in Java

There are several groups working on incorporating transactions into Java [MA96][AG96]. 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 applet simply starts and ends transactions, and every object which is manipulated within a transaction will automatically be made atomic. We consider these approaches unsuitable for Web applications as they suffer from the following limitations:

(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.

6. Conclusions and future work

In this paper we have presented an model for integrating atomic actions and the Web. Using transactional gateways and browser proxies we have shown how it is possible for non-transactional browsers to participate within a transaction. Applications can therefore obtain end-to-end transactional integrity, guaranteeing consistency in the presence of failures and concurrent access. The model is lightweight, and does not require the browser to incorporate support for transactions. We have presented the W3OTrans system which implements this model, and described an example which we have built using it.

Currently W3OTrans provides support for the automatic construction of simple browser proxies and gateways. We intend to extend this support to arbitrary, application specific implementations. W3OTrans transactions are

based upon those provided by the Arjuna system, which is a toolkit for building fault-tolerant distributed applications in C++ using objects and actions [GDP95]. Arjuna provides a number of fault-tolerance mechanisms, including object replication [MCL91], which we are investigating providing within the W3OTrans framework. It is also our intention to make W3OTrans compatible with the commercial standard for atomic actions, the Object Transaction Service from the OMG [OMG95]. This will enable Web applications to interoperate with commercial transactional applications and objects in other domains. Finally, we are investigating the Visa and Mastercard work in SET [SET96], to determine how to incorporate the security issues it raises.

Acknowledgments

The work reported here has been partially funded by grants from the Engineering and Physical Sciences Research Council (EPSRC) and the UK Ministry of Defense (Grant Numbers GR/H81078 and GR/K34863), Hewlett-Packard Laboratories and GEC-Plessey Telecommunications.

References

[DBI95] D. B. Ingham, M. C. Little, S. J. Caughey, S. K. Shrivastava, "W3Objects: Bringing Object-Oriented Technology To The Web," The Web Journal, 1(1), pp. 89-105, Proceedings of the 4th International World Wide Web Conference, Boston, USA, December 1995. <http://www.w3.org/pub/Conferences/WWW4/Papers2/141/>

[DBI96] D. B. Ingham, S. J. Caughey, and M. C. Little, "Fixing the Broken-Link Problem: The W3Objects Approach," Computer Networks and ISDN Systems, 28(7-11), pp. 1255-1268, Proceedings of the 5th International World Wide Web Conference, Paris, France, May 1996. <http://www5conf.inria.fr/fich_html/papers/P32/Overview.html>

[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. <http://arjuna.ncl.ac.uk/arjuna/papers/designimplearjuna.ps>

[DBL77] "Process Structure, Synchronisation and Recovery using Atomic Actions", D. B. Lomet, Proceedings of the ACM Conference on Language Design for Reliable Software, SIGPLAN Notices, Vol. 12, No. 3, March 1977.

[SUN95] "Java Security", J. S. Fritzinger and M. Mueller, Sun Microsystems, 1995. <http://www.javasoft.com/security/whitepaper.ps>

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

[MA96] "Draft Pjava Design 1.2", M. Atkinson et al, Department of Computing Science, University of Glasgow, January 1996. <http://www.dcs.gla.ac.uk/pjava/pjava-design.ps.gz>

[AG96] "Transactions for Java", A. Garthwaite and S. Nettles, MS-CIS-96-17, University of Pennsylvania, 1996.

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

[TRA96] "Transarc DE-Light Web Client Technical Description", Transarc Corporation, February 1996. <http://www.transarc.com/afs/transarc.com/public/www/Public/ProdServ/WWW/delov.html>

[RJS95] "Using Meta-Object Protocols to Implement Atomic Data Types", R.J. Stroud and Z. Wu., in Proceedings of ECOOP 95, vol. 952, pp. 168-189, Springer-Verlag, 1995.

[MCL91] “Object Replication in a Distributed System”, M. C. Little, PhD Thesis (Newcastle University Computing Science Laboratory Technical Report 376), September 1991. <ftp://arjuna.ncl.ac.uk/pub/Arjuna/Docs/Theses/TR-376-9-91_USLetter.tar.Z>

[OMG91] “Common Object Request Broker Architecture and Specification”, OMG Document Number 91.12.1.

[OSF96] “Applications of the Secure Web Technology in Transaction Processing Systems”, T. Sanfilippo and D. Weisman, The Open Group Research Institute, November 1996. <http://www.osf.org/RI/PubProjPgs/tp.htm>

[SET96] “Secure Electronic Transaction (SET) Specification, Book 1: Business Specification”, Mastercard and Visa, June 1996. <http://www.visa.com/sf/set/SETBUS.PDF>