Middleware Support for Non-repudiable Transactional Information Sharing between Enterprises

Nick Cook[1], Santosh Shrivastava[1], and Stuart Wheater[2]

[1] School of Computing Science
University of Newcastle, UK
{nick.cook,santosh.shrivastava}@ncl.ac.uk
[2] Arjuna Technolgies, Newcastle, UK
stuart.wheater@arjuna.com

Abstract. Enterprises increasingly use the Internet to offer their own services and to utilise the services of others. An extension of this trend is Internet-based collaboration to form virtual enterprises for the delivery of goods or services. Effective formation of a virtual enterprise will require information sharing across organisational boundaries. Despite the requirement to share information, the autonomy and privacy requirements of enterprises must not be compromised. This demands strict policing of inter-enterprise interactions, including non-repudiable access to shared information. For a member of a virtual enterprise, a typical requirement is the ability to inspect/modify shared information together with private information within a single ACID transaction. At the same time, inspection/modification of the shared information should both generate non-repudiation evidence and be consistent with inter-enterprise agreements. The paper describes how information sharing middleware can be enhanced with distributed transaction support to perform regulated transactional information sharing. Design and implementation of a prototype Java middleware is presented.

Keywords: middleware; inter-enterprise interaction; transactions; security

1 Introduction

As noted above, the formation of and continued interaction within a virtual enterprise (VE) demands regulated information sharing. In this context, each party to a multi-party interaction requires: (i) that their own actions on shared information meet locally determined, evaluated and enforced policy, and that their legitimate actions are acknowledged and accepted by the other parties; and (ii) that the actions of the other parties comply with agreed rules and are irrefutably attributable to those parties. These requirements imply the collection, and verification, of non-repudiable evidence of the actions of parties who share and update information. We have implemented distributed object middleware called

J.-B. Stefani, I. Demeure, and D. Hagimont (Eds.): DAIS 2003, LNCS 2893, pp. 125–132, 2003.

B2BObjects [1] that both presents the abstraction of shared state and meets these requirements by regulating, and recording, access and update to shared state. It is assumed that each enterprise has a local set of policies for information sharing that is consistent with an overall information sharing agreement (business contract) between the enterprises. Multi-party coordination protocols ensure that the local policies of an enterprise are not compromised despite failures and/or misbehaviour by other parties; and that, if no party misbehaves, agreed interactions will take place despite a bounded number of temporary network and computer related failures. Each party validates any proposed update to shared information and the update is only accepted if all parties agree to it.

The shared information of a VE does not exist in isolation. There are dependencies between private information held by each member of a VE and the shared information that is held in common. A given enterprise is also likely to be involved in more than one VE, resulting in dependencies between information that is shared in the context of different VEs. To manage these dependencies, support is required to make updates to shared information contingent on successful completion of updates to related private information (and vice versa). From the viewpoint of each member, their Business-To-Business (B2B) application state can be seen as the combination of any private information that is related to the B2B interaction and the information that is shared with the other members. The requirement then is to maintain the integrity and consistency of B2B application state by ensuring that updates to shared information are consistent with updates to private information and that such updates can be completed transactionally (atomically).

The paper presents a novel distributed middleware for updating B2B application state while meeting the above regulatory and consistency requirements. The main contribution of this work is the development of middleware with the ability to manage transactions that span private and shared resources at the same time as observing inter-enterprise agreements that govern update to the shared resources. The middleware is designed to provide local autonomy for each enterprise, within the constraints imposed by the need to share information, and interoperability between and within enterprises. The shared resources participate in transactions using the same mechanism as for private (transactional) resources (such as enterprise databases). Update to shared resources is subject to independent validation by the members of the VE who together own the resources.

Section 2 provides an overview of B2BObjects. Section 3 presents the extension to support distributed transactions over B2B application state. A related technical report [2] provides a more detailed description of the middleware with an example application scenario.

2 Overview of B2BObjects Middleware

This section provides an overview of the B2BObjects middleware, including a brief introduction to the Java API of the experimental implementation. The mid-

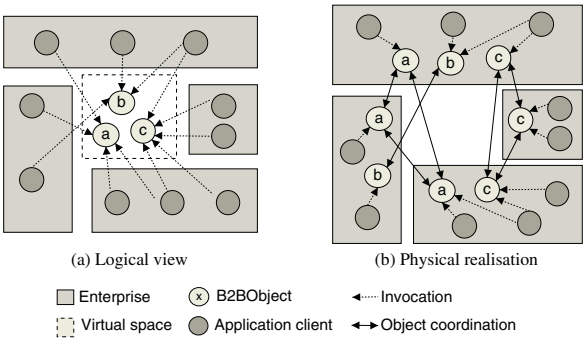


Fig. 1. B2BObjects-based interaction

dleware addresses the requirement for dependable information sharing between enterprise. The abstraction of shared objects is used to represent the information that enterprises wish to share (or "jointly own"). Coordination protocols provide multi-party agreement on access to and update of object state. As shown in Fig. 1, the logical view of shared objects in a virtual space is realised by the regulated coordination of actions on object replicas held at each enterprise. Application-level invocations on local copies of B2BObjects are intercepted by the middleware and state changes coordinated with remote enterprises. A non-repudiable two-phase commit protocol is used to coordinate object state as follows: the proposer of a new state dispatches a state change proposal, comprising the new state and the proposer's signature on that state, to all other parties for local (application-level) validation. Each recipient produces a response comprising a signed receipt and a signed decision on the (local) validity of the state change. All parties receive the collected responses and a new state is valid if the collective decision represents unanimous agreement to the change. The signing of evidence generated during state validation binds the evidence to the relevant key-holder. The actions of honest parties cannot be misrepresented by dishonest parties and invalid state cannot be imposed on local object replicas. The evidence generated is stored systematically in local non-repudiation logs. Systematic check-pointing of object state provides recovery, in the event of failure, and rollback, in the event of invalidation by one or more parties. Certificate management and non-repudiation services provide: authentication of access to objects; verification of signatures to actions on objects; and logging of evidence of each enterprise's actions.

2.1 B2BObjects API

This brief introduction to the B2BObjects API concentrates on the aspects that provide hooks for transactional update to B2BObjects. The relevant classes of the API are: B2BObject — the augmentation of an application object to ensure access is mediated by the middleware; and B2BObjectController — the local interface to configuration, initiation and control of information sharing. A B2BCoordinator executes the coordination protocols between objects. The

B2BObject interface is a wrapper for application objects that allows the controller to obtain object state, to initiate local validation of proposed state changes and to install newly validated object state following successful state coordination. The relevant part of the controller interface is:

```
public interface B2BObjectController {
  void enter();     // start of scope of access to state
  void examine();   // read in this scope
  void overwrite(); // completely overwrite in this scope
  void update();    // partial update in this scope
  void leave();     // end of scope of access to state
  ...
}
```

Given an application object (`appObject`) with a typical update operation: `setAttribute(SomeType attr)`, the corresponding B2BObject wrapper code is:

```
setAttribute(SomeType attr) {
  controller.enter();           // start of scope
  controller.overwrite();       // will overwrite object state
  appObject.setAttribute(attr); // set the appObject attribute
  controller.leave();           // end of scope, trigger coordination
}
```

This code can be auto-generated if the application object's read/write methods are identified. From the application viewpoint, the B2BObject `setAttribute` method is invoked in the same way as for `appObject`.

The controller `enter` and `leave` operations are used to demarcate the scope of access to object state. These calls may be nested to allow the "rolling-up" of a series of state changes into a single (atomic) coordination event. If `overwrite` has been called within the current state change scope, then invocation of the final `leave` triggers execution of the state coordination protocol. If a proposed change is invalidated, the proposer's local object state is rolled-back. A similar process applies to update of a part of object state (indicated by the `update` operation) as opposed to overwrite of the whole state. The `examine` operation indicates that object state will be read but not written in the current scope. The controller operations shown provide transactional access to all copies of a single B2BObject and, as described in Section 3.2, are the hooks for transactional update across multiple B2BObjects.

3 Support for Distributed Transactions

Transactions have long been used to ensure the consistency of shared information despite concurrent accesses and system failures — delivering the well-known ACID properties of Atomicity, Consistency, Isolation and Durability. The Java Transaction API (JTA) [3] is a standard interface to Java-based transaction management that includes the XAResource mapping of the XA standard [4] for participation in distributed transactions. In this section we describe a JTA-compliant transaction adapter that presents B2BObjects as transactional resources to a Transaction Manager via an XAResource interface. In this way, dis-

tributed transactions can be combined with multi-party coordination of shared state. First we outline the principles of the state transitions that underly B2BObject support for distributed transactions and then we provide an overview of the Java-based transactional infrastructure.

3.1 Outline of Transactional Support

To support transactions, the notion of B2BObject state, S, is extended to include both the prospective new state of the object (*prospState*) and the retrospective agreed state of the object (*retroState*). That is, for state coordination purposes, B2BObject state is described by the tuple: $S = \langle s_j, s_i \rangle$, where s_j is the prospState and s_i is the retroState. Given this description of object state, we can say that: an object is in a **committed state**, if $j = i$ (the prospState is the retroState); and an object is in a **prepared state**, if prospState has been coordinated (and validated) **and** $j \neq i$ (the prospState and retroState are different). The following state transitions are then permitted:

1 *committed* to *committed* :	$\langle s_i, s_i \rangle \rightarrow \langle s_{i+1}, s_{i+1} \rangle$	
2 *committed* to *prepared* :	$\langle s_i, s_i \rangle \rightarrow \langle s_{i+1}, s_i \rangle$	
3 *prepared* to *prepared* :	$\langle s_{i+1}, s_i \rangle \rightarrow \langle s_{i+2}, s_i \rangle$	
4 *prepared* to *committed* :	$\langle s_{i+1}, s_i \rangle \rightarrow \langle s_i, s_i \rangle$	(abort)
5 *prepared* to *committed* :	$\langle s_{i+1}, s_i \rangle \rightarrow \langle s_{i+1}, s_{i+1} \rangle$	(commit)

Transition 1 describes the behaviour of B2BObjects in [1] — transition from one committed state to the next with no intermediate prepared state. Transitions 2 and 3 to prepared states can be mapped to the prepare phase of a distributed transaction. In both cases, the retroState is unchanged and represents the state to which the object will ultimately return if the prospState is subsequently revoked. A prospState may be revoked because a transaction coordinator requests rollback of resources participating in a transaction or because a subsequent new state proposal is invalidated. Transitions 4 and 5 can be mapped to completion of a transaction: abort (or rollback) to the previously committed state $\langle s_i, s_i \rangle$; and commit of a new committed state $\langle s_{i+1}, s_{i+1} \rangle$, respectively. The difference between a prepared state and a committed state is that the former is revocable. If a prepared state is revoked, the object returns to the most recently committed state (identified by the retroState). If a prepared state is committed, the new retroState is the current prospState.

The following pseudo-code illustrates how the above transitions, demarcated by `enter/leave` blocks, can be combined to perform a distributed transaction across two B2BObjects: `objS` and `objT`. At the start of the transaction the objects are in states $\langle s_i, s_i \rangle$ and $\langle t_j, t_j \rangle$, respectively. The code is annotated with intermediate (prepared) states and the successful commit of final states.

```
// start transaction txId
enter(objS, txId)
enter(objT, txId)
  // perform state changes
```

```
    enter(objS)
    overwrite(objS) // locally change objS to prospState: s_{i+1}
    leave(objS)     // trigger coordination to prepared state: ⟨s_{i+1}, s_i⟩
    enter(objT)
    overwrite(objT) // locally change objT to prospState: t_{j+1}
    leave(objT)     // trigger coordination to prepared state: ⟨t_{j+1}, t_j⟩
   ...
    // Perform further state changes. For each enter/leave block,
    // state is coordinated so that, if all changes succeed,
    // objS is in state: ⟨s_{i+m}, s_i⟩ and objT is in state: ⟨t_{j+n}, t_j⟩
    ...
  // commit transaction txId
  leave(objS, txId, TX_SUCCESS)
                // trigger coordination to committed state: ⟨s_{i+m}, s_{i+m}⟩
  leave(objT, txId, TX_SUCCESS);
                // trigger coordination to committed state: ⟨t_{j+n}, t_{j+n}⟩
```

The prepare phase of the transaction corresponds to the following transitions:

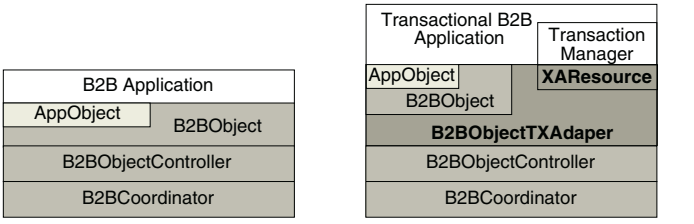
$$objS : \langle s_i,\, s_i\rangle \rightarrow \langle s_{i+1},\, s_i\rangle \rightarrow \cdots \rightarrow \langle s_{i+m},\, s_i\rangle$$
$$objT : \langle t_j,\, t_j\rangle \rightarrow \langle t_{j+1},\, t_j\rangle \rightarrow \cdots \rightarrow \langle t_{j+n},\, t_j\rangle$$

The final transitions to states $\langle s_{i+m},\, s_{i+m}\rangle$ and $\langle t_{j+n},\, t_{j+n}\rangle$ correspond to the successful commit phase. In contrast, any failure or invalidation of a transition to a prepared state for an individual object would result in transaction abort and the return of each object to the committed states: $\langle s_i,\, s_i\rangle$ and $\langle t_j,\, t_j\rangle$.

Any party's agreement to a transition to a prepared state, for example $\langle s_i,\, s_i\rangle \rightarrow \langle s_{i+1},\, s_i\rangle$, implies: (i) application-level validation of prospState s_{i+1} and, therefore, of committed state $\langle s_{i+1},\, s_{i+1}\rangle$; and (ii) their commitment to be able to subsequently install either of the related committed states: $\langle s_i,\, s_i\rangle$ or $\langle s_{i+1},\, s_{i+1}\rangle$. That is, to have made persistent the new prospState, s_{i+1}, and to be able to rollback the prospState to s_i. Thus transitions 4 and 5, from prepared to committed states, do not require application-level validation. Nor is it necessary to transfer the physical state of the object being coordinated for these transitions (since each party has already committed to local persistence of the relevant state). The only state that is physically transfered to remote parties is the new prospState for transitions 1, 2 or 3. Unique state transition identifiers are used to reference the retroState for each transition and the prospState for transitions 4 and 5. Coordination from a prepared to a committed state is required to ensure that all parties maintain a consistent view of object state and to generate evidence that the committed state is the currently agreed object state.

3.2 B2BObjects as Transactional Resources

This section describes the infrastructure to facilitate the participation of B2BObjects as JTA-compliant, transactional resources in distributed transactions. The essential requirements are: (i) that a JTA transaction manager can control



(a) B2BObject-enabled application (b) Transactional B2BObject-enabled application

Fig. 2. B2BObjects transaction layer

the participation of B2BObjects in transactions through a transaction adapter that exports the XAResource interface; and (ii) that the underlying B2BObject state management and coordination mechanisms can be instrumented to support this participation. The approach is to provide a transactional layer between the underlying layers of the middleware and the transactional application; and to parameterize the B2BObjectController operations described in Section 2.1 to effect the state transitions described above.

Fig. 2(a) shows the B2BObject interface as a wrapper for an application object. The B2B application uses the AppObject interface for operations on the object. The B2BObjects middleware provides the regulated state coordination described in Section 2. Fig. 2(b) shows the insertion of a transaction layer to support transactional applications. The application uses the same AppObject interface to the underlying object. The B2BObjectTXAdapter exports an XAResource interface to a Transaction Manager and instruments the controller to ensure the coordinator executes appropriate state transitions.

The B2BObjectTXAdapter generates a proxy for the application object being coordinated to ensure that, in transactional context, all operations on the object are mediated by the adapter. It maintains the association of the current transaction with the object and propagates this association to the controller. To meet transactional requirements, the adapter maps operations at the XAResource interface to controller operations. The transaction-aware controller guarantees the persistence of B2BObject state to facilitate recovery and rollback; and the persistence of transaction state information.

To ensure that application-level operations on an instance of a B2BObject are mediated by a transaction adapter, a B2BObjectTXAdapterFactory instantiates a single B2BObjectTXAdapter for a given B2BObject. The B2BObjectTXAdapter interface provides operations for the application to obtain an instance of the object proxy and for the Transaction Manager to obtain the adapter's XAResource instance.

To provide transaction-awareness, the B2BObjectController interface shown in Section 2.1 is extended to include parameterised versions of `enter` and `leave` to associate a transaction identifier with these operations. The extension also includes methods for explicit object locking and, for example, to support XAResource operations to manage heuristically completed transactions and recovery of prepared transactions. The XAResource interface provided by the B2BObject-

TXAdapter includes `start` and `end` operations to demarcate work on behalf of a given transaction; and `prepare`, `commit` and `rollback` operations for participation in the transaction two-phase commit protocol.

4 Concluding Remarks

We are not aware of other work that integrates distributed transactions with regulated information sharing between enterprises. The work of Wichert et al [5] is close to our approach to systematic generation of non-repudiation evidence. They provide non-repudiable RPC but do not address validation of state changes for information sharing. The work of Minsky et al on Law Governed Interaction (LGI) [6] supports interaction between organisations governed by global policy. It represents one of the earliest attempts to provide coordination between autonomous organisations. However, support for transactions is not available. Another approach to the automated control of interactions through agreements between enterprises is IBM's tpaML language for B2B integration [7]. Their model of long-running conversations, the state of which is maintained at each party, is similar to our notion of shared interaction state.

Acknowledgements

This work is part-funded by the UK EPSRC under grant GR/N35953/01 on "Information Co-ordination and Sharing in Virtual Environments"; by the EU under project IST-2001-34069: "TAPAS (Trusted and QoS-Aware Provision of Application Services)"; and by the UK e-Science project "GridMist".

References

1. Cook, N., Shrivastava, S., Wheater, S.: Distributed Object Middleware to Support Dependable Information Sharing between Organisations. In: Proc. IEEE Int. Conf. on Dependable Syst. and Networks (DSN), Washington DC (2002)
2. Cook, N., Shrivastava, S., Wheater, S.: Middleware Support for Non-repudiable Transactional Information Sharing between Enterprises. Technical Report 814, School of Computing Science, Univ. Newcastle (2003)
3. Cheung, S., Matena, V.: Java Transaction API (JTA version 1.0.1B). Java Specification (2002)
4. The Open Group: Distributed Transaction Processing: The XA Specification. X/Open CAE Specification XO/CAE/91/300, X/Open Company Ltd. (1991)
5. Wichert, M., Ingham, D., Caughey, S.: Non-repudiation Evidence Generation for CORBA using XML. In: Proc. IEEE Annual Comput. Security Applications Conf., Phoenix, Arizona (1999)
6. Minsky, N., Ungureanu, V.: Law-Governed Interaction: A Coordination and Control Mechanism for Heterogeneous Distributed Systems. ACM Trans. Softw. Eng. and Methodology **9** (2000) 273–305
7. Dan, A., Dias, D., Kearney, R., Lau, T., Nguyen, T., Sachs, M., Shaikh, H.: Business-to-business integration with tpaML and a business-to-business protocol framework. IBM Syst. J. **30** (2001) 68–90