School of Computing Science,
University of Newcastle upon Tyne

Middleware Support for Non-repudiable Transactional Information Sharing between Enterprises

Nick Cook, Santosh Shrivastava and Stuart Wheater

Technical Report Series

CS-TR-814

August 2003

Middleware Support for Non-repudiable Transactional Information Sharing between Enterprises

Nick Cook, Santosh Shrivastava
University of Newcastle, UK
{nick.cook, santosh.shrivastava}@ncl.ac.uk

Stuart Wheater
Arjuna Technologies, 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 between enterprises 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 member-specific 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 policies agreed by the enterprises. 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 platforms; inter-enterprise interactions; transactions; security; non-repudiation*

1. Introduction

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 between enterprises 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 the strict policing of inter-enterprise interactions. Thus there is a requirement for dependable mechanisms for information sharing between enterprises who do not necessarily trust each other. In this context, each party to a multi-party interaction requires:

1. 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

2. 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 B2BObjects [3] 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.

Regulated information sharing of the kind described above is essential for the successful formation of virtual enterprises and for continued interaction in the context of a virtual enterprise. However, shared information does not exist in isolation. There are dependencies between private information held by each member of a virtual enterprise and the shared information that is held in common. A given enterprise is also likely to be involved in more than one virtual enterprise. This results in dependencies between information that is shared in the context of different virtual enterprises. 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 both the regulatory and the consistency requirements identified above. 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 virtual enterprise who together own the resources.

Section 2 presents an application that illustrates the need for transactional access to B2B application state. Section 3 briefly describes how distributed object middleware (such as CORBA, J2EE) is used to support intra-enterprise distributed transactions. An overview of B2BObjects is provided in Section 4. The extension of B2BObjects to support distributed transactions over B2B application state is described in Section 5. Related work is presented in Section 6. The paper concludes with a discussion of future work.

2. Application Scenario

In this section we present the scenario of a specialist car manufacturer who combines components from various part suppliers to satisfy the particular requirements of specialist car dealers (acting on behalf of the ultimate customer). Figure 1(a) presents the overall

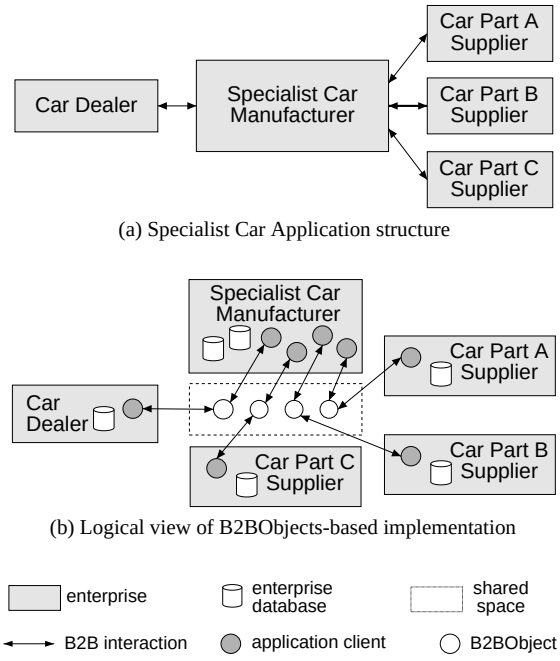


Figure 1. Specialist Car Application

structure of the interaction between specialist car dealer, car manufacturer and, in this case, three car part suppliers. In effect, these enterprises collaborate to form a virtual enterprise for delivery of a specialist car to the car dealer's customer.

The initial phase of the scenario involves negotiation between the car manufacturer and the dealer to agree the car's specification. Both the car manufacturer and the dealer require the maintenance of non-repudiable evidence of the state of their negotiations. During negotiation, the car manufacturer and car part suppliers share information such as: part specifications, prices, quantities and delivery schedules. The car manufacturer and the part suppliers require non-repudiable evidence of updates to this information. On successful

completion of the negotiation phase, the interaction enters an acceptance phase in which the dealer commits to the purchase. To achieve this and to fulfill the order, the car manufacturer requires non-repudiable acceptance of the agreement that has been reached from all parties involved. Furthermore, the car manufacturer requires that acceptance is contingent on successful updates to its own databases to reflect the order from the dealer. Finally, the dealer requires non-repudiable commitment to delivery of the agreed order by the car manufacturer.

This application illustrates the need for:

- the generation of non-repudiable evidence both of changes to shared information and of the acceptance of those changes;
- transactional access to information to perform a set of changes; and
- extension of transactional access to span both shared and private information.

Figure 1(b) shows the use of the B2BObjects middleware to maintain the state of negotiations between the dealer and car manufacturer, and between the car manufacturer and the parts suppliers. Negotiation state is modelled as a set of shared objects (B2BObjects). The middleware ensures that actions on shared objects are non-repudiably bound to the actor. Further, the acceptance, or otherwise, of those actions is non-repudiably bound to the other parties who share the state. Support for the shared objects to participate as resources in distributed transactions will ensure that a set of updates can be completed as an atomic action and can be made contingent on the successful completion of local database updates. For simplicity, a set of two-party interactions coordinated by the car manufacturer is shown. However, B2BObjects supports multi-party, peer-to-peer interaction. Neither the B2BObjects middleware, nor its support for transactions, restricts the structure to a set of two-party interactions coordinated by a single party such as the car manufacturer.

In the next section we describe how existing middleware supports transactions for consistent update to a set of local distributed resources (such as the car manufacturer's databases). In Section 4, we provide an overview of the B2BObjects middleware. The remainder of the paper addresses the combination of B2BObjects and transactions.

3. Middleware Support for 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. This section describes how commonly used middleware, such as CORBA and J2EE, supports transactional update to a set of local resources.

Figure 2 shows the three basic roles in a distributed transaction: the transactional client (application) that is responsible for setting the transaction boundaries; the transactional resources or services (such as enterprise databases) that can be updated consistently in the context of a transaction; and the transaction coordinator to coordinate delivery of the ACID

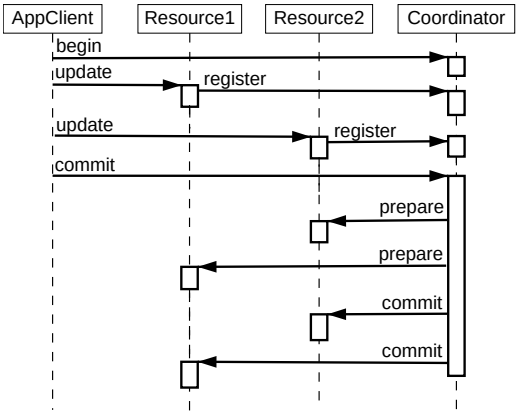


Figure 2. Middleware-supported Distributed Transaction

properties. As shown, the client first requests the`begin` of a new transaction from the coordinator. Then the client invokes update operations on the resources. The transaction-aware resources register their participation in the transaction with the coordinator. The application indicates the end of the transaction by requesting that the coordinator`commit`. As a result, the coordinator executes a two-phase commit protocol with the resources. In the prepare phase, each resource votes either to commit the transaction, if their updates can be made durable; or to abort, if not. In the commit phase, if, as shown, all resources vote in favour, the coordinator invokes `commit` on each resource. On completion of the commit phase, control is returned to the application. If any resource had voted to abort the transaction, the coordinator would have invoked `rollback` on each resource.

Different transactional resources may use different mechanisms to meet transactional requirements and may have different interfaces to those mechanisms. The XA standard [8] has been defined to manage the resulting heterogeneity. The standard defines the contract (interface) between transactional resources and transaction coordinator in a distributed transaction processing environment. Thus, a transactional resource that implements the XA interface to the two-phase commit protocol can participate in a distributed transaction that is controlled by an external transaction coordinator. The Java Transaction API (JTA) [2] is a standard interface to Java-based transaction management that includes a Java mapping of the XA interface (XAResource). Access to enterprise resources is mediated by Resource Managers that export the XAResource interface to a JTA Transaction Manager. In Section 5 we describe a JTA-compliant transaction adapter that presents B2BObjects as transactional resources to a Transaction Manager via an XAResource interface. In this way, distributed transactions can be combined with multi-party coordination of shared state.

4. 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 interested reader is

referred to [3] for a more detailed discussion of the middleware.

B2BObjects addresses the requirement for dependable information sharing between enterprises (identified in Section 1). The middleware uses the abstraction of shared objects to represent the information that enterprises wish to share (or "jointly own"). Changes to object state are subject to a locally determined and evaluated validation process. Validation is application-specific and may be arbitrarily complex. A simple example of application-level validation is that the car manufacturer is allowed to alter the specification of a car part and its delivery date, within agreed bounds, but not the price of the part. While a car part supplier is allowed to alter the price of the part and its delivery date, within agreed bounds, but is not allowed to alter the specification of the part. Coordination protocols provide multi-party agreement on access to and update of object state.

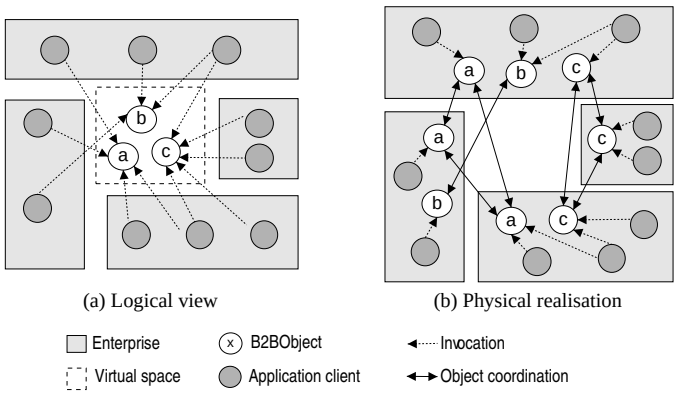


Figure 3. B2BObjects-based Interaction

As shown in Figure 3, 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. This triggers application-level validation of the proposed changes at the remote enterprises. The regulated coordination proceeds 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 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. Evidence 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.

The middleware supports autonomy and interoperability. The constraints imposed by information sharing are that: a common representation of the state that is coordinated must

be agreed and parties must execute agreed coordination protocols. However, the local realisation of the shared state is under the control of each enterprise as is the implementation of the middleware to coordinate it. For example, the middleware can be configured to use an adapter for application-specific transformation of the common state representation to some local realisation. Similarly, the local application-level interface to a B2BObject can be different for each enterprise as can the process to determine the validity of proposed state changes (each enterprise is autonomous with respect to the validation decisions it makes). Compliance with the XA standard (as described in Section 5.2) ensures interoperability with local transaction management systems.

4.1. 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 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 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, coordinate state
}
```

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 described in Section 4.2. 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 5.2, are the hooks for transactional update across multiple B2BObjects.

4.2. State Coordination

A non-repudiable two-phase commit protocol is used to coordinate the state of object replicas. The protocol ensures that a given state transition is unanimously agreed or does not occur. Evidence is generated to ensure that the actions of honest parties cannot be misrepresented by dishonest parties and that invalid state cannot be imposed on local object replicas. If all parties behave correctly, liveness is guaranteed despite a bounded number of temporary failures (crashes and message loss). Evidence generated by the protocol can be used to detect misbehaviour and to resolve disputes.

For a set of n parties, $\{P_i \mid i \in 1:n\}$, coordinating the state of an object, the basic form of the protocol is:

$$\begin{array}{lll} propose & P_k \rightarrow Rset_k & : \; mp,\, sig_k\left(h\left(mp\right)\right) \\ respond & Rset_k \rightarrow P_k & : \; mr_j = \{d_j,\, sig_j\left(h\left(mp\right),\, h\left(d_j\right)\right)\} \\ resolve & P_k \rightarrow Rset_k & : \; rn_k,\, \sum mr_j \end{array}$$

Where: P_k is the proposer of a state transition. $Rset_k = \{P_j \mid j \in 1:n \, and \, j \neq k\}$ is the recipient set for P_k's proposal. mp is P_k's state transition proposal message. mr_j is a response message from P_j. d represents P_j's decision on the validity or otherwise of the proposed change. $sig_i\left(x\right)$ is P_i's signature on data x and $h\left(y\right)$ is a secure hash of data y. rn_k is a secure random number generated by P_k, a hash of which forms part of the unique identifier of a state change proposal (see below). $\sum m_i$ is the concatenation of a set of messages.

The protocol messages, mp and $\sum mr_j$, contain the information necessary to verify the internal integrity of each message and the consistency of the set of messages taken together. For example: each proposed state transition is uniquely identified by a tuple of the form:$\langle seqno,\, h\left(rn\right),\, h\left(S\right)\rangle$; where $seqno$ is a proposal sequence number, $h\left(rn\right)$ is a hash of a secure random number, and $h\left(S\right)$ is a hash of the state to which the tuple refers. The tuple is generated locally by the proposer of a new state and, if the state is validated, the tuple uniquely identifies the new agreed state of the object being coordinated. The protocol messages include the tuple that identifies the currently agreed state as viewed by each party. Thus, at resolution of the protocol, all parties can determine that the proposed transition is from the same known current state to the mutually agreed new state. Other

information exchanged ensures a consistent view of group membership. Any decision,d_j, that represents rejection of the state change results in invalidation of the proposal, as does any inconsistency between protocol messages. Messages are signed, and time-stamped, to provide non-repudiation both of state changes and of decisions with respect to their validity. Update to, as opposed to overwrite of, object state and changes to group membership are coordinated using similar non-repudiation protocols.

5. Support for Distributed Transactions

In this section we first outline the principles of the state transitions that underly B2B-Objects support for distributed transaction. We then provide details of the Java-based infrastructure to enable B2BObjects to participate as transactional resources in distributed transaction middleware of the type described in Section 3.

5.1. 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 [3] — 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: $\langle s_{i+m}, s_{i+m} \rangle$ and $\langle t_{j+n}, t_{j+n} \rangle$.

```
// 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)      // coordinate objS to prepared state: ⟨s_{i+1}, s_i⟩
    enter(objT)
    overwrite(objT)  // locally change objT to prospState: t_{j+1}
    leave(objT)      // coordinate objT to prepared state: ⟨t_{j+1}, t_j⟩
    ...
    /* Perform further state changes. For each enter/leave block,
     * object 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)
    // coordinate objS to committed state: ⟨s_{i+m}, s_{i+m}⟩
leave(objT, txId, TX_SUCCESS);
    // coordinate objT to committed state: ⟨t_{j+n}, t_{j+n}⟩
```

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

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

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:

1. application-level validation of prospState s_{i+1} and, therefore, of committed state $\langle s_{i+1}, s_{i+1} \rangle$; and
2. 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

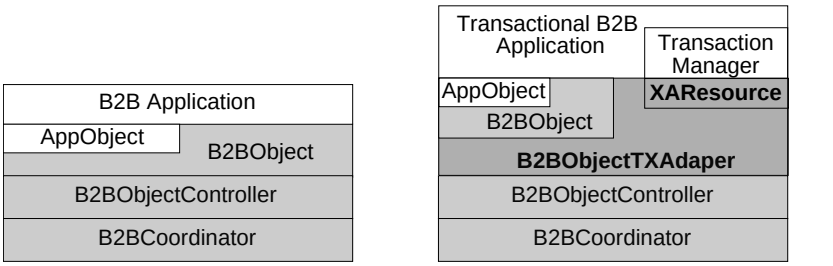
relevant state). The only state that is physically transfered to remote parties is the new prospState for transitions 1, 2 or 3. The unique state transition identifiers described in Section 4.2 are used to reference the retroState for each transition and the prospState for transitions 4 and 5. Coordination 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.

5.2. 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:

1. that a JTA transaction manager can control the participation of B2BObjects in transactions through a transaction adapter that exports the XAResource interface; and
2. 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 application and the underlying layers of the information sharing middleware; and to parameterize the B2BObjectController operations described in Section 4.1 to effect the state transitions described in Section 5.1.



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

Figure 4. Transaction Layer in B2BObjects-enable Application

Figure 4(a) shows the B2BObject interface as a wrapper for an application object. The B2B Application uses the same AppObject interface for operations on the application object as it would in a non-B2B application. Together, the B2BObject, B2BObjectController and B2BCoordinator provide the regulated state coordination described in Section 4. Figure 4(b) shows the extension of B2BObjects to support a transactional application. The application still uses the same AppObject interface to the underlying application object but now a B2BObjectTXAdapter layer exports the XAResource interface to a Transaction Manager and instruments the B2BObjectController to ensure the B2BCoordinator executes appropriate state transitions. The B2BObjectTXAdapter and the transaction-aware B2BObjectController together fulfill the role of Resource Manager for a B2BObject. They are both provided as part of the extended B2BObjects middleware.

A B2BObjectTXAdapterFactory instantiates a single B2BObjectTXAdapter for a given local instance of a B2BObject. The adapter provides operations to obtain an auto-generated proxy for the object being coordinated and the XAResource interface of the adapter (to enlist with a Transaction Manager):

```
public interface B2BObjectTXAdapter {
    Object getB2BAppObjectProxy();  // get application object proxy
    XAResource getXAResource();     // get XAResource adapter for
                                    // distributed transactions
    ...
}
```

The proxy ensures that all operations on the application object are intercepted by the adapter. The adapter associates the current transaction with the object and propagates this association to the B2BObjectController. The adapter maps operations at the XAResource interface to controller operations.

The controller is responsible for mapping adapter requests to state coordination requests. It guarantees the persistence of B2BObject state to facilitate recovery and rollback; and the persistence of transaction state information. For example, it maintains a persistent link between a transaction identifier (provided by the adapter) and the state coordination events associated with the transaction. The B2BObjectController interface shown in Section 4.1 is extended to include parameterised versions of `enter` and `leave` that associate a transaction identifier with these operations. As shown below, 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.

```
public interface B2BObjectController {
    public void acquireLock(int flag);   // acquire indicated lock
    public void enter();                 // enter state change scope
    public void enter(Object txId);      // enter txId transaction
                                        // scope
    public void leave();                 // leave state change scope
    public void leave(Object txId,       // leave txId transaction
                      int flag);         // scope
    public void examine();               // read-only indication
    public void overwrite();             // overwrite indication
    public void update();                // update indication
    public void forgetTxId(Object txId); // forget a heuristically
                                        // completed transaction id
    public Object[] recoverTxIds();      // recover id(s) for any
                                        // prepared transaction(s)
    public void releaseLock();           // release lock

    ...
}
```

The XAResource interface provided by the B2BObjectTXAdapter 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.

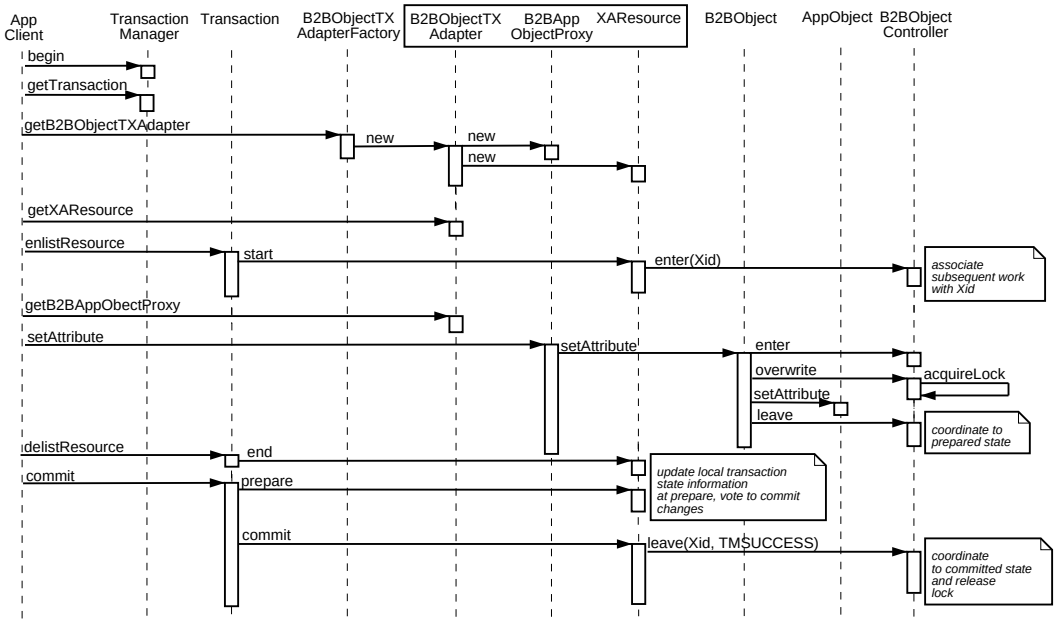


Figure 5. B2BObject Transaction Sequence Diagram

Figure 5 is a sequence diagram for update of a B2Bobject in the context of a JTA-based distributed transaction — showing the process from application demarcation of the transaction to successful commit. In a typical distributed transaction other resources such as enterprise databases would be enlisted with a transaction in the same way and participate in the two-phase commit protocol via their own XAResource interface. For simplicity, these other resources are not shown. The application-level `setAttribute` is invoked on the B2BAppObjectProxy provided by the B2BObjectTXAdapter. The proxy associates operations on the application object with the current transaction (identified by `Xid`) and ensures that the transaction context is propagated to the controller. The controller is therefore aware that the `enter/leave` block enclosing the `setAttribute` call is within a transaction and initiates coordination to a prepared state on execution of the `leave`. Since the call is made through the proxy, the adapter is aware of the success or failure of the call (and of the resultant state coordination). In the example, the call succeeds and the adapter ensures that the `prepare` call to the XAResource results in a vote to commit the changes. As shown, XAResource calls are mapped to operations on the B2BObjectController. Here we provide brief details of three of the more significant operations:

start results in registration of a transaction identifier with the controller (`enter (Xid)`). After the parameterised `enter` has been called, the controller associates subsequent access with the given transaction and, for example, is aware that a lock must be acquired on the first call to `overwrite`.

prepare results in update to local transaction state and, assuming coordination to prepared

state succeeds, a vote to commit the changes.

commit results in a parameterised call to `leave` and in the controller initiating coordination of the object to a committed state (transition 5 in Section 5.1). The previously acquired lock is released during coordination to committed state. On completion, the controller can discard information relating to management of the identified transaction.

Object locking

The controller is responsible for guaranteeing single-writer, multiple reader lock semantics. Since any party can veto a proposed update from any other party, a read lock is granted locally by the controller. Acquisition and release of write locks is coordinated in the same way as object state by associating a lock-holder identifier with the object state. Thus, as noted above, a single coordination event can be used both for transition to committed state and release of the associated lock. Object's are either locked explicitly or implicitly when `examine`, `overwrite` or `update` are called in transaction context. Similarly, locks that are not released explicitly are implicitly released when a transaction completes.

Deferred state coordination

It is possible to configure the middleware to execute either immediate or deferred coordination. Immediate coordination is as described above — each update to an object in the context of a transaction results in coordination with remote replicas. In this case, invalidity with respect to remote parties is detected early with consequent rollback of the transaction. Deferred coordination is an optimisation where updates to an object are performed locally and coordination with remote replicas is deferred to a single coordination event during the prepare phase of the transaction. The adapter implements deferred coordination using the controller's support for nested `enter`/`leave` blocks. An additional `enter` call is made on the controller at the start of a transaction and the XAResource `prepare` call results in the corresponding `leave` to trigger coordination to prepared state. For example, the pseudo-code given in Section 5.1 is modified as follows:

```
// start transaction txId
enter(objS, txId)
enter(objT, txId)
     // defer coordination
    enter(objS)
    enter(objT)
        // perform state changes
        enter(objS)
        overwrite(objS) // locally change to prospState: s_{i+1}
        leave(objS)     // do not coordinate
        enter(objT)
        overwrite(objT) // locally change to prospState: t_{j+1}
        leave(objT)     // do not coordinate
        ...
```

```
        /* Perform further state changes without coordination
         * so that, locally,  objS is in state: ⟨s_{i+m}, s_i⟩
         * and objT is in state: ⟨t_{j+n}, t_j⟩
         */
    // prepare: end deferred coordination
    leave(objS)        // coordinate to prepared state: ⟨s_{i+m}, s_i⟩
    leave(objT)        // coordinate to prepared state: ⟨t_{j+n}, t_j⟩
// commit transaction txId
leave(objS, txId, TX_SUCCESS)
    // coordinate to committed state: ⟨s_{i+m}, s_{i+m}⟩
leave(objT, txId, TX_SUCCESS);
    // coordinate to  committed state: ⟨t_{j+n}, t_{j+n}⟩
```

For $objS$, the single transition:

$$\langle s_i,\, s_i\rangle \rightarrow \langle s_{i+m},\, s_i\rangle$$

replaces coordination through the series of prepared states described in Section 5.1.

Deferred coordination also optimises object locking since, until the prepare phase, it is sufficient to veto remotely-initiated update of object state by acquiring a read lock with respect to remote replicas. Then a write lock is acquired as part of the coordination to prepared state. Deferred coordination results in less interaction with remote parties at the expense of delayed validation of state changes. An advantage is that local failure during a series of updates to a B2BObject, or related resources, can be confined. The failure precludes coordination with remote parties.

Majority voting during commit phase

The requirement at the transaction commit phase is to ensure that the proposer does not attempt to issue a commit to some parties and abort to others. As noted in Section 5.1, transition from a prepared to a committed state does not require further application-level validation because the relevant B2BObject state has already been subject to validation by all parties. During transaction commit, or abort, it is therefore possible for the proposer to short-circuit the response phase of the state coordination protocol after receipt of replies from a majority of respondents (that is after receipt of $\frac{n-1}{2} + 1$ replies for an n-party interaction). Thus, non-cooperation of a minority of respondents at this stage can be tolerated.

6. Related Work

We are not aware of other work that integrates distributed transactions with regulated information sharing between enterprises.

The work of Wichert et al [9] 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.

Policy-controlled interaction is relevant to application-level validation of updates to shared information and is, therefore, complementary to B2BObjects. Ponder [4] provides a unified approach to the specification of both security and management policy for distributed

object systems. 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 [5]. Their model of long-running conversations, the state of which is maintained at each party, is similar to the notion of shared B2B application state.

The Business Transaction Protocol (BTP) [1] supports an alternate model to the transactional update of information that is shared by multiple enterprises. BTP allows enterprises to participant in transactions that are coordinated by another organisation in a loosely-coupled relationship where ACID transactions may be inappropriate. The state that is managed is not normally visible outside the enterprise that owns it. Each party effectively commits to delivery of some service in the context of the externally coordinated transaction. BTP does not address the consistency of internal resources with the state of the business interaction nor does it address security requirements such as non-repudiation. B2BObjects offers a tighter binding of parties to the outcome of a transaction and, therefore, may be more suited to collaboration in virtual enterprises.

7. Conclusions and Future Work

We have presented middleware that addresses the requirement for dependable information sharing between enterprises. The middleware presents the abstraction of shared state and regulates updates to that state. We have shown how this middleware can be extended to allow updates to shared information in the context of standards-compliant distributed transactions. The middleware presents a familiar programming abstraction to the application programmer and frees them to concentrate on the business logic of applications.

The paper describes update to B2BObjects in the context of locally-controlled ACID transactions. The application-level view is of transition from committed object state to committed object state. Remote parties are unaware, at the application level, that updates occur in the context of a transaction. We will investigate extension of this to support propagation of transaction context between enterprises to allow update to a B2BObject by multiple parties in the context of a global transaction. We will also investigate the integration of transactions with messaging middleware [7] to provide transactional and asynchronous update to B2BObjects. Another area for future work is support for extended (loosely-coupled) transaction models with compensation for partial failure driven by application-level semantics. We envisage the exposure of intermediate (prepared) object state and the instrumentation of coordination to those states as one mechanism for delivering different application semantics.

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] A. Ceponkus, S. Dala, T. Fletcher, P. Furniss, A. Green, and B. Pope. Business Transaction Protocol. OASIS Committee Specification, 2002.

[2] S. Cheung and V. Matena. Java Transaction API (JTA version 1.0.1B). Java Specification, 2002.

[3] N. Cook, S. Shrivastava, and S. Wheater. Distributed Object Middleware to Support Dependable Information Sharing between Organisations. In *Proc. IEEE Int. Conf. on Dependable Syst. and Networks (DSN)*, Washington DC, 2002.

[4] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder Policy Specification Language. In *Proc. Int. Workshop on Policies for Distributed Syst. and Networks (POLICY), Springer-Verlag LNCS 1995*, Bristol, UK, 2001.

[5] A. Dan, D. Dias, R. Kearney, T. Lau, T. Nguyen, M. Sachs, and H. Shaikh. Business-to-business integration with tpaML and a business-to-business protocol framework. *IBM Syst. J.*, 30(1):68–90, 2001.

[6] N. Minsky and V. Ungureanu. Law-Governed Interaction: A Coordination and Control Mechanism for Heterogeneous Distributed Systems. *ACM Trans. Softw. Eng. and Methodology*, 9(3):273–305, 2000.

[7] S. Tai and I. Rouvellou. Strategies for Integrating Messaging and Distributed Object Transactions. In *Proc. ACM/IFIP Middleware 2000*, New York, USA, 2000. LNCS 1795, Springer-Verlag.

[8] The Open Group. Distributed Transaction Processing: The XA Specification. X/Open CAE Specification XO/CAE/91/300, X/Open Company Ltd., 1991.

[9] M. Wichert, D. Ingham, and S. Caughey. Non-repudiation Evidence Generation for CORBA using XML. In *Proc. IEEE Annual Comput. Security Applications Conf*, Phoenix, Arizona, 1999.