A Framework for Configurable Distributed Transactions

S.M. Wheater and S.K. Shrivastava
Department of Computing Science
University of Newcastle, Newcastle upon Tyne, NE1 7RU, England
email: stuart.wheater@ncl.ac.uk

1. Introduction

We consider a computation model in which application programs manipulate persistent (long-lived) objects under the control of atomic actions (atomic transactions). The Common Object Request Broker Architecture (CORBA) together with its services is a well known example of a distributed object model that will support the above model. At the basic level CORBA consists of the Object Request Broker (ORB) that enables distributed objects to interact with each other. At the next level a number of system level services have been specified. These services include persistence, concurrency control and Object Transaction Service (OTS). The OTS is a *protocol engine* intended to guarantee that transactional behaviour is obeyed, relying on other system level services to support the ACID properties [1]. Such a structure has the advantage that an application can be bound to any given set of compliant system services for obtaining transactional behaviour. There is therefore considerable scope for customising a transactional application (say for performance improvements) by choosing the most appropriate implementations of individual services. In this paper we explore this idea in detail and present the design of a transaction framework (a distributed object transaction support system) that permits a transactional application to be customised (configured) to a degree that has not been possible before. A very modular approach is presented that enables individual objects to be made atomic, with each object bound to concurrency control, persistence and recovery services in an application specific manner. Furthermore, these bindings can be changed at run time. So for example, it is possible for an object to switch from pessimistic to optimistic concurrency control at run time.

The framework to be presented here is based on our experience of designing and implementing a number of transaction services for distributed objects. These include the Arjuna system [2], the OTS version of the Arjuna system that we have recently completed and a Java transaction system [3]. This experience has enabled us to design a set of 'implementation neutral' service interfaces that permit considerable scope in the provision of a wide variety of implementations, all conforming to the same interface specification.

2. Diversity of requirements

The distributed object transaction support system is required to be configurable for several reasons; we enumerate some of them here:

(i) An important requirement which needs to be addressed by an implementation of the support system is object server management policy, indicating the relationship between passive persistent object states (on stable storage) and active objects (objects loaded in servers which are capable of having operations performed on them). Most applications could do with some control over object to server binding, including enabling and disabling of caching of objects at clients. It may be desirable to permit caching of read mostly objects and prevent caching of write mostly objects. Whats more, it may be desirable to dynamically switch on/off these as access patterns on an object vary.

(ii) Objects have different concurrency control requirements, so the system should be able to support these, e.g., pessimistic and optimistic.

(iii) The atomic action structure may need to be extended to provide more flexible structures, such as split transactions, glued and coloured actions.
(iv) The support for storage and retrieval of the persistence state of the transactional object can take many forms and will depend on the non-functional requirements of the objects such as performance and availability. To allow high availability it may be required to have the persistence support to perform replication. On the other hand it may be desirable to use some high performance log-based object store, or special server on a dedicated machine, may be required.

3. Transactional object support system

3.1. Overall structure

To implement ACID properties, the transaction framework (distributed object transaction support system) must monitor the operations performed on transactional objects and the beginning and ending of transactions. If an operation on an object will compromise one of the ACID properties, the system either prevents the operation from being performed or any effects of the operation from becoming visible.

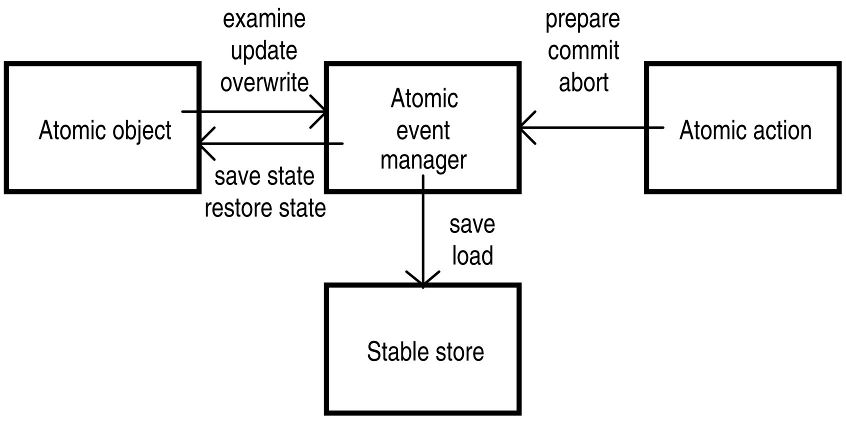

Figure 1: Transactional object support system events.

Fig. 1, illustrates the main structure of the transactional object support system and the events that can occur within it. An atomic object is responsible for generating events `examine`, `update` and `overwrite` to indicate that the object is about to be examined, updated or overwritten, respectively. An atomic action generates `prepare`, `commit` and `abort` events. The *event manager* acts on these events by directing these events to its concurrency control, recovery and persistence services to which the atomic object and the atomic action object are bound. Reconfiguration along the lines hinted in the previous section is made possible essentially by changing these bindings to different implementations of these services. The event manager is also responsible for generating events corresponding to saving and restoring the states of objects for recovery purposes, and saving and loading the states of objects to and from stable store to ensure state changes are durable.

3.2. Design of the distributed object transaction support system

As can be appreciated from the previous sub-section, the idea behind the design of the object transaction support system is to provide interfaces that isolate applications from the actual services responsible for implementing the ACID properties. This allows us to choose specific implementations on a per object basis, without functionally affecting applications. The distributed object transaction support system has been implemented using the Gandiva application building framework [4]. Standard RPC technology hides the differences between local and remote invocations. The framework also supports a variety of object to server mapping policies. Software components are split into two separate entities: the *interface* *component* and the *implementation component*. The interactions between implementations can only occur through interfaces. A single interface can be used to access multiple implementations, and a single implementation can be accessed through multiple interfaces. Gandiva permits the bindings of interfaces to implementations to be dynamic and configurable. Applications are written only in terms of interfaces, and although an application can request a specific implementation, it occurs in a way that allows this request to be changed without modifying the application.

The event manager shown in fig. 1 has been split into four constituent classes, concurrency control (CC), persistence (P), recovery (R) and a coordinating class. Each instance of CC, P and R has two interfaces: atomic object manager interface (operations `examine`, `update` and `overwrite)` and atomic action manager interface (operations `prepare`, `commit` and `abort`); the coordinating class has only the atomic object manager interface. The resulting structure is illustrated in fig. 2.

The atomic action manager interface is used to isolate the atomic action from the details of transaction processing. An atomic action will maintain a list of atomic action manager objects, which are processed when the transaction ends. During the execution of an atomic action, instances of atomic action manager may be added to the list in response to specific events. The processing that is performed on the list when the action ends differs depending on whether the action commits or aborts. If it commits, the processing of this list takes the form of a two-phase commit protocol, using the `prepare()` and `commit()` operations of the atomic action manager objects. If it aborts the `abort()` operation is called. To make the preparing phase more flexible the prepare operation can be performed in multiple rounds. The `prepare()` operation takes an integer to indicate which round of prepare it is for. The return status of the `prepare()` operation will indicate if the transaction manager is fully prepared or requires to be involved in another round of preparation.

The atomic object manager interface is used to isolate atomic objects from the details of their support. The interface provides `examine()`, `update()` and `overwrite()` operations that are called to indicate to the atomic event manager that the object is about to be examined, updated or overwritten, respectively.

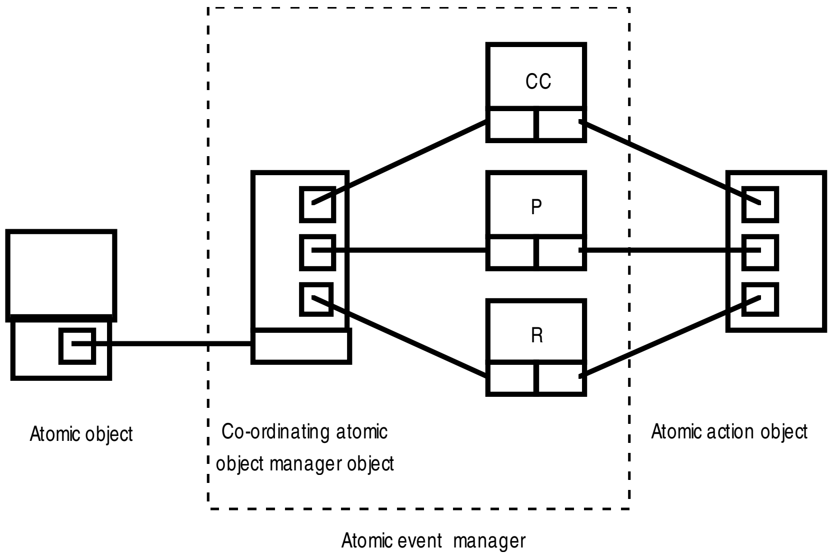

Figure 2: Object structure of the transactional object support system.

4. Examples

4.1. Mixing optimistic and pessimistic concurrency control

This simple example illustrates how two types of concurrency control can simultaneously be used within the same application. The application consists of a single atomic action *t* which performs operations on two objects *a* and *b*. Object *a* is using pessimistic concurrency control whereas object b is using optimistic concurrency control. The operations oper1 and oper2 performed by *t* are assumed to be update operations. The bodies of these operations therefore contain calls on the `update` operations of atomic object manager interface.

```
Action t;

t.begin();

a.oper1();
b.oper2();

if (. . .)
    t.commit();
else
    t.abort();
```

Within `oper1` the `update` operation will be invoked on *a*'s atomic object manager, which in turn will invoke `update` operations on the concurrency control, persistence and recovery (in that order); only if all these objects return true will the operation be allowed. The concurrency control object will perform conflict detection with the locks held on *a*, if none of the locks conflict the concurrency control will return true. The persistence support object will ensure that the object contains the current state of the object. The recovery support object will ensure that appropriate recover information is recorded. The sequence of events for operation `oper2` is the same. However, the optimistic concurrency controller for b will always return true.

If *t* commits, the objects which support *a* and *b* will each will be asked to prepare; for object *a* the concurrency control and recovery support objects will simply reply PREPARE_OK, whereas *a*'s persistence support object will first save a copy of *a*'s state and then return PREPARE_OK. Process is simpler for object *b*, except that *b*'s persistence support object will verify that the state within the object store is that which `oper2` was performed upon.

4.2. Mixed-mode persistence implementation

Our framework can support multiple implementations of persistence. A simple implementation will consist of each object forcing its state on stable store during `prepare.` We describe here how an optimised log-based persistence can be supported (without affecting application objects of course). We assume a two stage log management policy: in the first stage, a log of various object updates is constructed in volatile store and in the second stage this log is forced on to the stable store. This way, multiple force operations on the stable store from individual objects is avoided. A mix mode of operation is possible: some objects are using log-based persistence, whereas others are using the simple strategy. This is made possible because the framework supports multiple rounds of `prepare` during commit processing. Objects requiring multiple rounds return 'prepare continue' response, eventually returning 'prepare ok'.

4.3. Object server management policies

The need for exercising control over object server management policy, indicating the relationship between passive persistent object states (on stable storage) and active objects (objects loaded in servers which are capable of having operations performed on them) was briefly mentioned in section two. For the sake of illustration, assume that the following two policies:

- For each persistent object state there exists at most a single active object: this means that no co-ordination is required to maintain the properties of serialisability, failure atomicity and permanence of effect. This model can provide high performance, but the service will become unavailable if the server managing the object fails.

- For each persistent object state there can exist many active objects, co-located on the same node: the co-ordination required can be performed via fast single node inter-process communication mechanisms such as shared memory. This model can tolerate the failure of a server containing an active object, but not the failure of the entire node.

Our framework will permit an object to be managed by any such server. We have implemented and tested several object management policies.

5. Concluding Remarks

The framework presented here permits an application designer to choose the atomic object support system implementations that suits the requirements of the application. Considerable degree of freedom exists in selecting from a variety of concurrency control, persistence and server management implementations to control non-functional behaviour of application, while leaving functional characteristics unchanged. An application can be bound to the desired implementations at build time and the bindings can be changed at run time. The framework and a set of implementations, some of which have been discussed above, have been designed and implemented.

Selected References

[1] CORBA services: Common Object Services Specification, OMG Document No. 95-3-31, March 1995.

[2] G.D. Parrington, S.K. Shrivastava, S.M. Wheater and M. Little, “The design and implementation of Arjuna”, USENIX Computing Systems Journal, vol. 8 (3), pp. 255-308, Summer 1995.

[3] M.C. Little and S.K. Shrivastava, “Distributed transactions in Java”, submitted to this HPTS workshop.

[4] S.M. Wheater and M.C. Little, “The design and implementation of a framework for configurable software”, Third IEEE Intl. Conf. on Configurable Distributed Systems, CDS96, May 1996, Annapolis, Maryland, pp. 136-146.