Configuring Distributed Applications using Object Decomposition in an Atomic Action Environment*

Stuart M. Wheater
Daniel L. McCue

Computing Laboratory,
University of Newcastle upon Tyne.

Abstract

A common technique for constructing reliable distributed applications is to use atomic actions for controlling operations on persistent objects. Atomic actions are used to ensure that inconsistencies in application state do not arise when failures occur or when concurrent activities operate on shared objects. Within such an application, objects provide a convenient unit for distribution and concurrency-control. The properties of atomic actions and objects can be exploited together to configure distributed applications, without affecting the correct functioning of the application. This leads to the possibility of changing the configuration of concurrency and distribution of the distributed application to improve availability and performance. These changes in concurrency and distribution can be achieved by varying the object decomposition within the application. In this paper, we show how some kinds of re-configuration can be achieved without any modification to client applications. The observations are a result of constructing reliable distributed applications using the Arjuna system, which provides tools and libraries for programming with atomic actions and persistent objects in C++.

1. Introduction

One of the major concerns in the construction of reliable applications in a distributed environment to do with the maintenance of the consistency of an application's state, despite the failure of components (hardware or software). One technique, which has been used successfully for constructing particular classes of reliable distributed applications, is the use of *atomic actions* [1]. Applications which have been found suitable for the use of atomic action are those which must process complex operations on persistent data concurrently, such as: banking systems and airline seat reservation systems.

One way of structuring applications using atomic actions, which has been advocated by many research projects, for example, Argus [2], Arjuna [3], Avalon/C++ [4], Clouds [5] and ANSA [6], is for atomic actions to operate on persistent objects. Persistent objects being objects which still exist after the application which created them has terminated.

When constructing such an application, two of the most critical sets of design decisions that need to be made are:

---

* Appeared in the Proceedings of the IEE Workshop on Configurable Distributed Systems, London, March 1992

i) *Atomic action structure*: How should the operations of the application be decomposed into atomic actions? Should a number of atomic actions be grouped within an enclosing atomic action (nesting)? Which atomic actions can be run concurrently? Which atomic actions must finish before others can be started?

ii) *Object decomposition*: How should the application's state be decomposed into objects? Which objects should be decomposed into sub-objects? What operations do the objects have? Should a sub-object be independently concurrency-controlled, or should the concurrency-control applied to the container object also control access to (all of the) sub-objects? On which node should each object reside?

Some aspects of the behaviour of such applications, such as performance and availability, can be altered by changing the atomic action structure and/or the object decomposition. These alterations in the configuration of the application should not affect the correct functioning of the application. However, apart from minor changes in the scope of an atomic action, the atomic action structure cannot generally be changed without altering the semantics of the application. Hence, there is little scope for re-configuration by modification of the action structure. This paper is focused on describing the kinds of changes that can be made to the object decomposition of the application, which do not alter the correct functioning of the application. These changes to the object decomposition can be used to reconfigure the concurrency and distribution of the application thereby improving its overall availability, reliability, or performance.

2. Atomic actions

Distributing applications creates new opportunities which are not always available in centralised applications. Such applications can be designed to utilize the concurrency inherent in a distributed environment. Also, distributed applications can take advantage of the fact that the failure of a single hardware component is unlikely to cause the complete failure of the distributed environment.

These opportunities con also give rise to problems for the designers of an application. The concurrency within an application must be managed to ensure the application's state remains consistent. Component faults that cause part of the distributed environment to fail could also give rise to inconsistencies in an application's state unless steps were taken to prevent them.

Atomic actions provide a mechanism which addresses these two problems. Atomic actions aid maintenance of a consistent application state due to their properties of:

Serialisability: Concurrently executing atomic actions do not interfere with each other (i.e., their concurrent execution will produce the same effect as some serial order execution).

Failure atomicity: Atomic actions will either terminate normally (*commit*), producing the intended effects or producing no effects (*abort*).

Permanence of effect: Any effects produced by an atomic action which has terminated, are not lost due to later component failures.

Atomic actions can be combined with other fault tolerance mechanisms to cope with a wide range of failures, such as hardware and software design errors, but in the rest of this paper, we consider the use of atomic actions only to tolerate fail-stop faults i.e., faulty components are silent, and operations performed in isolation in the absence of any faults will be regarded as always producing consistent state changes.

Given that an application is constructed using atomic actions which perform certain operations, it is sometimes necessary to combine these atomic operation sequences to form larger operation sequences, which are also required to be atomic [7]. The resulting larger atomic action's effects being some combination of the effects of the atomic actions from which it is composed. The atomic actions which are contained within the resulting atomic action are said to be *nested*, and the resulting action is referred to as the *enclosing* atomic action. The enclosing atomic action is sometimes referred to as the *parent* of the *nested* or *child* atomic action. The outermost atomic action in an atomic action structure is called a *top-level* atomic action.

Nested atomic actions differ from top-level atomic actions in that the effects of a nested atomic action will be recovered if the enclosing atomic action aborts, even if the nested atomic action has committed. This is required to ensure the failure atomicity of the enclosing atomic action. Also, to maintain the serialisability of the enclosing atomic action, the locks obtained by the nested atomic actions which commit, must be inherited by their enclosing atomic action.

Figure 1, shows a pictorial representation of an enclosing atomic action *A* which contains the nested atomic actions *B* and *C*, where atomic action *B* must be completed before atomic action *C* can be started.

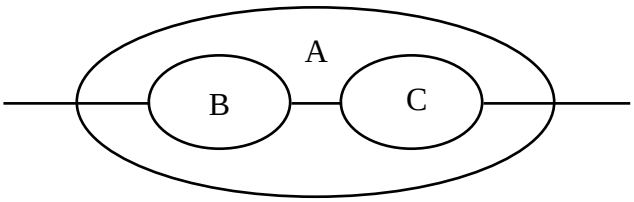

Figure 1.

The design of an application's atomic action structure is mostly governed by the application's requirements and how the application's state has been decomposed into objects [8]. The application's requirements generally leave little scope for reconfiguring the application by changing the atomic action structure.

3. Object-oriented programming

The basic concept in object-oriented programming is support of the abstraction of *objects*. An object is an entity with some internal state which can only be manipulated using a set of operations provided by the object. Objects provide a form of *data abstraction* in that the internal state of the object, and the implementation of the set of operations need not concern the users of the object; they need only understand the behaviour of its operations. In many object-oriented programming languages, the definition of the objects is given using the *class* construct. Thus objects are instances of the corresponding classes. The classes can be related

through inheritance to form higher level abstraction, but this inheritance capability of object-oriented programming languages is of secondary importance to the discussion that follows.

Because the object model completely encapsulates the internal implementation of an object, objects are well suited to constructing reconfigurable applications. The implementation of the object can be reconfigured without affecting its users, as long a the interface to the object remains unchanged both syntactically and semantically.

3.1. Object decomposition

In the initial stages of designing an application, a designer must decide how the application's state should be decomposed into objects, what operations these objects provide, and how these objects are decomposed into further objects [9]. These decisions will vary greatly from designer to designer. What is of more interest as far as configuration is concerned is how the resulting abstractions provided by the objects can be used for reconfigured. If objects are the units of concurrency-control, distribution, recovery and persistence, the decomposition of the application state into objects constitutes a configuration

Two important properties which can be varied by altering the object decomposition of an application to create a new configuration are its concurrency-control and distribution. Concurrency-control ensures the serialisability of concurrent atomic actions acting upon an object. Distribution specifies where the object (and it's sub-objects) are located. In the next sections these topics will be examined in more detail.

3.2. Object concurrency-control

A simple way to provide serialisability for concurrent atomic actions within an object-oriented environment, would be for the operations of an object to acquire an appropriate lock on the object. This approach maintains the encapsulation of many implementation details (e.g. the type of lock required read or write) of the object. The encapsulation of an object means that it's concurrency-control behaviour (and it's sub-objects) can be refined, without affecting other objects in the application.

The concurrency-control, can be implemented so that the object (and its sub-objects) are a single concurrency-controlled entity, or implemented as numerous individually concurrency-controlled objects.

For the purposes of explanation, consider a queue object, which contains banking requests, and provides operations to:

- Remove a request from the head of the queue.
- Add a request to the tail of the queue.
- Apply operations to requests within the queue.

For example, the queue could apply an operation which obtained the account numbers of the bank accounts involved in a request, or altered the amounts involved in a request. The internal sub-objects of the queue are the banking requests in the queue, and the objects which indicate the head and the tail of the queue.

The queue object could be implemented so that it and each individual request object were independently concurrency-controlled entities (depicted in figure 2). In such an implementation, the operations of the queue object which operated on the bank requests, would acquire a read lock on the queue object, then invoke the appropriate operation on the bank request object. The bank request operation would then acquire the appropriate lock on its object to perform the operation. Because such operations would not necessitate acquiring a lock on the head or tail objects within the queue, they could be performed concurrently with, for example, an operation which added a new request to the queue.

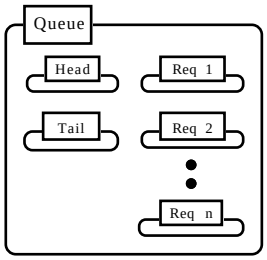

Figure 2.

Alternatively, the queue object could be implemented so that it and all the individual request objects were a single concurrency-controlled entity (depicted in figure 3). In such an implementation, the operations of the queue object which operated on the bank requests would acquire a read lock on the queue object, then invoke the appropriate operation on the bank request object, as before. But, when the bank request operation acquires the lock, it would be placed on the queue object, not on the request object. Because in this alternative implementation such operations would effectively acquire a lock on the head and tail objects within the queue, any update operations, whether they were updating individual bank requests or updating the queue structure, would be mutually exclusive.

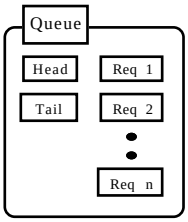

Figure 3.

3.3. Object distribution

Because objects provide data abstraction and encapsulation, object-oriented programming has advantages when constructing distributed applications. This is because the interface provided by objects are a convenient boundary over which to provide service, whether local or remote.

The interface to an object specifies the operations which can be performed upon the object. To allow these operations to be invoked remotely by a user of the object, a *stub* object can be provided which has the same interface as the original object. The implementation of the stub object causes a remote procedure call [10] to invoke the operation on the original object which may reside on a remote node. Figure 4 depicts a queue object which contains stub request objects (in grey).

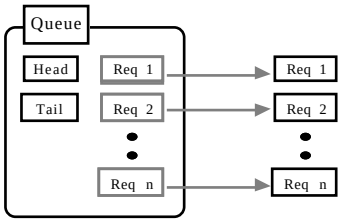

Figure 4.

The provision of stub objects can be automated by the use of a language specific *stub generator* (for example [11]). A stub generator processes the interface of the original object and produces an implementation of that interface (the stub) which *forwards* the operation invocations to the object and reports the results back to the local user.

The management of concurrency-control for a set of related objects and the distribution of objects (and their sub-objects) over a network of nodes in a distributed application are important aspects of the configuration of the application. Different configurations will have different properties even though the overall semantics of the application remain unchanged. The following sections discuss the impact of changes in distribution and concurrency-control and their implementation in a system of objects and actions.

4. Configuration

When designing any application to a given specification, trade-offs must be made between different attributes for the application, such as performance and availability. If an application is to be configurable, it will have to be designed so that these attributes can be changed at a later date, to take account of new priorities or circumstances, while still satisfying the original functional specification of the application.

Suppose all operations provided by a class of objects are performed as atomic actions. The internal state of such objects would remain consistent, despite concurrency and component failures. If the entire state of the application were composed of such objects, then the entire state of the application would be guaranteed to be consistent (given that it has been assumed that the individual operations have been correctly programmed and only produce consistent state changes). By making all operations of all objects atomic actions, the location of objects within the distributed environment becomes irrelevant to the users of those objects. Even if the node on which an object resides crashes during one of its operations, the object will remain consistent. The operations of objects can be implemented without worrying about concurrent invocation of other operations on the object, as long as the appropriate locks are acquired to ensure serialisability. Hence, applications constructed using atomic actions for every object operation invocation will be transparently configurable in terms of their decomposition and distribution.

5. Configuration trade-offs

Performance and availability are key attributes of an application which are affected by changes in configuration. Two aspects of the application's configuration which can be manipulated to alter these attributes are:

Concurrency-control: Whether an object and its sub-objects are a single concurrency-controlled entity or maintain separate concurrency-control information

Distribution: Where the objects of the application are located within the distributed environment.

Grouping an object and its sub-objects into a single concurrency-controlled entity will reduce the level of concurrent processing possible on the object, because more operations on the object will be considered to be potentially interfering. The effect of decreasing the amount of concurrency within an application could be to a reduce the performance of the application. However, this potential performance loss may be compensated for by reducing the considerable overhead involved with providing each individual sub-object with it own concurrency-control. So, if the operations of an object commonly use most of the sub-objects and the object is not commonly used concurrently by atomic actions which could interfere, then there might be performance benefits gained by grouping an object and its sub-objects as a single concurrency-controlled entity.

Apart from the overhead due to providing more advanced concurrency-control, there is another reason why increasing the potential concurrency within an application may not increase the performance of the application: without appropriate load-balancing and distribution, increases in "possible" concurrency may not be realised since a set of potentially concurrent accesses may all be scheduled for a single (uni-) processor. This problem can be reduced by locating objects which place large demands on the processor on different processors. By identifying and relocating such objects, the potential concurrency might be exploited and overall performance of applications might be improved. However, this solution introduces another source of overhead: the relatively large cost involved in accessing remote objects. These trade-offs affecting the performance of applications are difficult to evaluate analytically. This is why run-time configuration options are so important - they allow performance tuning after the application has been developed, based on actual data.

An example of an object which could improve the performance of an application by being located away from other objects is a directory service object. Suppose a directory service object is used to locate other objects within an application. It is possible that such an object will be constantly performing concurrent look-up operations (which could be performed as non-interfering atomic actions). Such an object would be used throughout an application, so it may not be possible to locate it "close" to its users.

The location of the objects, within the distributed environment, will also affect the availability of the application. An increase in "distributedness" of the application can cause the availability to decrease because the application becomes dependent on other parts of the distributed environment operating without failure. This effect can be reduced by replicating the objects (for example [12]), but replication will generally introduce overheads which may decrease overall performance.

Applications which are distributed, in an environment which is not fully operational, may still be able to provide a part of their original functionality. This means that distributing objects within the application can improve the availability of parts of the application.

6. Example application: Theatre seat reservation

One of the distributed applications which has been implemented using the ideas described in this paper is a simple theatre seat reservation system. We have build this application using the Arjuna system [3]. The application provides facilities which allow groups of seats to be reserved, unreserved or have their status checked. This application is designed to allow multiple users (booking offices), to perform arbitrary sets of operations, concurrently, and from different locations in the distributed environment. These groups of invocations of operations can be performed within user created atomic actions, so allowing users to reserve rows of seats or redisplay all the theatre seats "atomically".

The application is implemented using two classes of objects, theatre objects and seat objects. The operations of both theatre objects and seat objects are all performed as atomic actions. Seat objects are sub-objects of theatre object, and as such are only accessible via the operations of theatre objects.

Such an object decomposition leads to the possibility of different configurations, for the theatre seat reservation system. For example:

i) The seat objects obtain their concurrency-control from the theatre object, and are co-located with the theatre object.

ii) The seat objects are individually concurrency-controlled, and are located with the theatre object.

iii) The seat objects are individually concurrency-controlled, and are distributed.

Each of these configurations has its merits; configurations (ii) and (iii) being suitable for situations where a theatre object is often used concurrently. Configuration (i) is more appropriate if concurrent use of the theatre object is rare; in such situations the individual concurrency-control of each seat, which is provided by configurations (ii) and (iii) would impose unnecessary overheads. As far as availability is concerned, configurations (i) and (ii) have advantages over (iii) because they depend less on the distributed environment operating correctly. Due to the overheads involved in distributing the seat object, configuration (iii) is likely to have a lower performance than (ii); because seat reservation is not a processor intensive operation, remote access overheads are likely to dominate access times.

The outline of the interface definition (in C++) for the Theatre class is given below:

```
enum SeatStatus { UnReserved, Reserved };
enum Result     { OK, Failed, AlreadyReserved, NotReserved };

class Theatre : public . . .
{
private:
      Seat* seats[MAX_SEAT_NUM];
      . . .
public:
      void GetSeatStatus(int seat, SeatStatus& status, Result& res);
      void ReserveSeat(int seat, Result& res);
      void UnreserveSeat(int seat, Result& res);
      . . .
};
```

The Theatre class interface provides the operations *GetSeatStatus*, *ReserveSeat* and *UnreserveSeat*. Each of these operations takes as an input, the number (*seat*) of the seat which is to be operated upon. The outcome of the operation is indicated using the parameter *res*, if the operation is completed successfully it is set to *OK*, if the operation is completely unsuccessful it is set to *Failed*. If the operations was unsuccessful for a particular reason (for example, attempting to reserve an already reserved seat), a special value is returned. The *GetSeatStatus* operation also has a parameter which is the status of the seat indicated.

The implementation (slightly simplified) of the theatre object's *ReserveSeat* operation is given below:

```
void Theatre::ReserveSeat(int seat, Result& res)
{
      AtomicAction A;
      A.Begin();

      res = Failed;
      if ((seat >= 0) && (seat < MAX_SEAT_NUM))
            if (setlock(READ) == GRANTED)
                  seats[seat]->Reserve(res);    // Operation on seat object

      if (res != Failed)
            A.End();
      else
            A.Abort();
}
```

The theatre object's *ReserveSeat* operation uses the atomic action sub-system provided by Arjuna system. To start an atomic action, the operation creates an *AtomicAction* object, then invokes that AtomicAction object's *Begin* operation. To acquire a lock on the theatre, the operation invokes the setlock operation (provided by the class from which the Theatre class is derived). For this operation, a *read* lock is required on the theatre object. If the lock is granted the reservation operation of the seat object can be invoked. If the reservation operation has been successfully completed, the atomic action can be committed by invoking the AtomicAction object's *End* operation, otherwise the atomic action can aborted using the *Abort* operation.

The Seat class provides operations which manipulate a single seat, allowing it to be reserved, unreserved, and its status checked. In this implementation, the only information held about

individual seats is their status, i.e. whether the seat has been reserved or not. The outline of the interface definition for the Seat class is given below:

```
class Seat :public . . .
{
private:
      SeatStatus seat_status;
public:
      void GetStatus(SeatStatus& status, Result& res);
      void Reserve(Result& res);
      void Unreserve(Result& res);
      . . .
};
```

Like the operations of the theatre object, the operations of seat objects are performed as atomic actions.

```
void Seat::Reserve(Result& res)
{
      AtomicAction A;
      A.Begin();

      res = Failed;
      if (setlock(READ) == GRANTED)
      {
            if (seat_status == Reserved)
                  res = AlreadyReserved;
            else
                  if (setlock(WRITE) == GRANTED)
                  {
                        res = OK;
                        seat_status = Reserved;
                  }
      }

      if (res != Failed)
            A.End();
      else
            A.Abort();
}
```

The distribution of components of the theatre application described here can be reconfigured in several ways without changing any of the code. Three plausible arrangements with different performance and availability characteristics are:

Centralised: The theatre object can be co-located with its users (e.g., booking applications) and with its constituent parts - the seats.

Intermediate: The theatre object is remote from its users, but co-located with its seats. Applications which use a theatre can be pre-processed, replacing the implementation of the theatre object with a stub object for the theatre object.

Decentralised: The applications, theatre and seats are all on different nodes. As above, but seat objects in the theatre object are replaced with stub code which forwards seat requests to the remote nodes.

The concurrency-control of the application can be reconfigured by changing the behaviour of the *setlock* operation. The *setlock* operation performed by the seat objects could be modified to place a lock on the containing theatre object. This new setlock implementation can be provided

by deriving the seat objects from a class which implements the setlock operation in this way. A more attractive approach would be the initialise the base class, at creation time, with information about how concurrency-control is to be performed.

7. Conclusions

Constructing an application which is required to be reconfigurable requires that the application be designed so that aspects of its configuration can be altered, while still satisfying it's original specification. This paper has examined the ways in which changes to object decomposition can create new configurations which alter the performance or availability of the application without changing its function. The appropriate use of atomic actions and objects allows these aspects of the application's design to be altered without significant alterations to the application structure or code.

To quantify the relationships between the different configurations, so that stronger guide lines can be obtained on how distributed applications can be reconfigured to improve the attributes of the application, will require further work, which is under way. Also, design of distributed applications which require very high levels of availability or reliability is another aspect of reconfiguring distributed applications which will require further work.

Acknowledgements

This work has been supported in part by grants from the UK Science and Engineering Research Council and ESPRIT project 2267 (Integrated Systems Architecture). Discussions with Santosh Shrivastava, Graham Parrington, Mark Little and João Geada have been helpful in clarifying the ideas put forward here.

References

1. Gray, J.N., *Notes on Data Base Operating Systems*, in Operating Systems: An Advanced Course, Lecture Notes in Computing Science, pp. 393-481, Springer-Varlag, 1978.

2. Liskov, B., *Distributed Programming in Argus*, Communications of the ACM, Vol. 31, No.3, pp. 300-312, March 1988.

3. Shrivastava, S.K., G.N. Dixon, and G.D. Parrington, *An Overview of the Arjuna Distributed Programming System*, IEEE Software, pp. 66-73, January 1991.

4. Herlihy, M.P. and J.M. Wing, *Avalon: Language Support for Reliable Distributed Systems*, in Seventeenth Annual Symposium of Fault-Tolerant Computing, Pittsburgh, July 1987.

5. Dasgupta, P., R.J. LeBlanc Jr, and E. Spafford, *The Clouds Project: Designing and Implementing a Fault Tolerant, Distributed Operating System*, Technical Report GIT-ICS-85/29, Georgia Institute of Technology, 1985.

6. ANSA, *ANSA Reference Manual*, Release 01.00. March 1989.

7. Moss, J.E.B., *Nested Transactions: An Approach to Reliable Distributed Computing*, Technical Report MIT/LCS/TR-260, Massachusetts Institute of Technology, Laboratory for Computing Science, April 1981.

8. Wheater, S.M., *Constructing Reliable Distributed Applications Using Actions and Objects*, Ph.D. Thesis, Techical Report TR/316, Computing Laboratory, University of Newcastle upon Tyne, June 1990.

9. Kramer, J., J. Magee, and M. Sloman, *Constructing distributed systems in Conic*, IEEE Transactions on Software Engineering, Vol. 15, No. 6, pp. 663-675, June 1989.

10. Nelson, B.J., *Remote Procedure Call*, Ph.D. Thesis, Technical Report CMU-CS-81-119, Department of Computing Science, Carnegie-Mellon University, 1981.

11. Parrington, G.D., *Reliable Distributed Programming in C++: The Arjuna Approach*, in Second Usenix C++ Conference, San Fransisco, April 1990.

12. Little, M.C. and S.K. Shrivastava, *Replicated K-Resilient Objects in Arjuna*. in Proceedings of IEEE Workshop on the Management of Replicated Data, Houston, Texas, IEEE Computer Society Press, November 1990.