An Overview of Arjuna: A Programming System for Reliable Distributed Computing

S.K. Shrivastava, G.N. Dixon, G.D. Parrington

Abstract

*Arjuna* is an object-oriented programming system which provides a set of tools for constructing fault-tolerant applications. *Arjuna* supports *nested atomic actions* (nested atomic transactions) for structuring programs. Programs invoke operations on *objects* which are instances of abstract data types. In *Arjuna*, objects are *long lived entities* (persistent) and are the main repositories for holding system state. By ensuring that objects are only manipulated within an atomic action, it can be guaranteed that the integrity of objects (and hence the integrity of the system) is maintained in the presence of failures such as node crashes and the loss of network messages. A number of mechanisms are required to achieve fault tolerance and to provide distribution. This paper gives an overview of these mechanisms, which have been implemented in the language C++, and describes how they support the object model of computation employed by *Arjuna*.

Series Editor: M.J. Elphick

©1989 University of Newcastle upon Tyne.
Printed and published by the University of Newcastle upon Tyne, Computing Laboratory, Claremont Tower, Claremont Road, Newcastle upon Tyne, NE1 7RU, England.
SHRIVASTAVA, Santosh Kumar


Newcastle upon Tyne: University of Newcastle upon Tyne: Computing Laboratory, 1989.

(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 298)

Added entries
UNIVERSITY OF NEWCASTLE UPON TYNE.
Computing Laboratory. Technical Report Series. 298
DIXON, Graeme Neville
PARRINGTON, Graham Douglas

Abstract

Arjuna is an object-oriented programming system which provides a set of tools for constructing fault-tolerant applications. Arjuna supports nested atomic actions (nested atomic transactions) for structuring programs. Programs invoke operations on objects which are instances of abstract data types. In Arjuna, objects are long lived entities (persistent) and are the main repositories for holding system state. By ensuring that objects are only manipulated within an atomic action, it can be guaranteed that the integrity of objects (and hence the integrity of the system) is maintained in the presence of failures such as node crashes and the loss of network messages. A number of mechanisms are required to achieve fault tolerance and to provide distribution. This paper gives an overview of these mechanisms, which have been implemented in the language C++, and describes how they support the object model of computation employed by Arjuna.

About the author

Professor Shrivastava joined the Computing Laboratory in August, 1975, where he is a Professor.
Dr. G.N. Dixon was a Research Associate in the Computing Laboratory from December 1985 to June 1989.
Dr. G.D. Parrington is a Research Associate in the Computing Laboratory.

Suggested keywords

ATOMIC ACTIONS
DISTRIBUTED SYSTEMS
FAULT-TOLERANCE
OBJECT-ORIENTED SYSTEMS
PERSISTENCE

Suggested classmarks (primary classmark underlined)
Dewey (18th): 001.64404 001.6425
U.D.C. 519.687 681.322.06
An Overview of Arjuna: A Programming System for Reliable Distributed Computing

Santosh K Shrivastava, Graeme N Dixon and Graham D Parrington

Computing Laboratory
University of Newcastle upon Tyne
Newcastle upon Tyne, NE1 7RU, UK.

Abstract

Arjuna is an object-oriented programming system which provides a set of tools for constructing fault-tolerant distributed applications. Arjuna supports nested atomic actions (nested atomic transactions) for structuring programs. Programs invoke operations on objects which are instances of abstract data types. In Arjuna, objects are long lived entities (persistent) and are the main repositories for holding system state. By ensuring that objects are only manipulated within an atomic action, it can be guaranteed that the integrity of objects (and hence the integrity of the system) is maintained in the presence of failures such as node crashes and the loss of network messages. A number of mechanisms are required to achieve fault tolerance and to provide distribution. This paper gives an overview of these mechanisms, which have been implemented in the language C++, and describes how they support the object model of computation employed by Arjuna.

Keywords:

1. Introduction

*Arjuna* is an object-oriented programming system that provides a set of tools for the construction of fault-tolerant distributed applications. A prototype version of the system has been designed and implemented in C++ [10] to run on a collection of Unix† workstations connected by a local area network. *Arjuna* provides nested atomic actions (also known as nested atomic transactions) for structuring application programs. Atomic actions control sequences of operations upon (local and remote) objects, which are instances of abstract data types (C++ classes). Operations upon remote objects are invoked through the use of remote procedure calls (RPCs). This paper presents an overview of *Arjuna*, describing how the abstraction of atomic actions controlling operations on objects within a distributed environment has been realised.

Any distributed object and action based system must provide a number of integrated mechanisms for supporting a variety of system functions. These mechanisms include those for naming, locating and invoking operations on objects, concurrency control, recovery control, managing object states for long term as well as short term storage etc. In addition they should be flexible, permitting application specific enhancements of the existing mechanisms, for example type-specific concurrency control. *Arjuna* has been designed to provide such a set of mechanisms through a number of C++ classes which are organised in a class/type hierarchy that will be familiar to the developers of ‘traditional’ (single node) centralised object–oriented systems. Thus *Arjuna* illustrates that such an object–oriented design approach can be extended to apply to fault–tolerant distributed systems.

2. Objects and Actions in Distributed Systems

A computational model that has been widely advocated for constructing robust distributed systems is based upon the concept of using atomic actions controlling operations on persistent objects. An object is an instance of some type or class. Each individual object consists of some variables (its instance variables) and a set of operations (its methods) that determine the externally visible behaviour of the object. The operations supported by an object have access to the instance variables and can thus modify the internal state. It is assumed that the invocation of any of these operations produces consistent (class specific) state changes to the objects in the system in the absence of failures.

In a distributed system, an operation upon a remote object is typically invoked via a remote procedure call (RPC), which passes value parameters to the operation and returns the results of the operation to the caller. Furthermore, all operation invocations may be controlled by the use of atomic actions which have the properties of (i) serialisability, (ii) failure atomicity, and (iii) permanence of effect. The first property ensures that the concurrent execution of programs which access common objects is free from interference (i.e. a concurrent execution can be shown to be equivalent to some serial order of execution). The second property

†Unix is a registered trademark of AT&T
ensures that a computation can either be terminated normally (committed), producing the intended results or it can be aborted producing no results. This abortion property may be obtained by the appropriate use of backward error recovery, which is invoked whenever a failure that cannot be masked occurs. Typical failures causing a computation to be aborted include node (workstation) crashes and communication failures such as the continued loss of messages. It is reasonable to assume that once a computation terminates normally, the results produced are not destroyed by subsequent node crashes. This is the third property – permanence of effect – which ensures that any state changes produced (i.e. new states of objects modified in the action) are recorded on stable storage, a type of storage which can survive node crashes with high probability. A commit protocol is required during the termination of an action to ensure that either all the objects updated within the action have their new states recorded on stable storage (committed), or no updates get recorded (aborted) [5]. Some form of concurrency control policy, such as that enforced by two-phase locking, is also required to ensure the serialisability property of actions.

The object and action model provides a natural framework for designing fault-tolerant systems with persistent objects. Persistent objects normally reside in object stores which are designed to be stable. Atomic actions are employed to control the state changes to these objects and the properties of atomic actions given above ensures that only consistent state transformations take place on objects, despite failures. An object normally resides on a single node in one object store, however, the availability of an object can be increased by replicating it on several nodes and thus storing it in more than one object store. Such replicated objects must be managed through appropriate replica-consistency protocols to ensure that the object copies remain mutually consistent. Such protocols can be integrated within action based systems as discussed in [1].

It is assumed that the hardware components of the system are workstations (nodes), connected by a communication subsystem (for example, a local area network). A node is assumed to work either as specified or simply to stop working (crash). After a crash a node is repaired within a finite amount of time and made active again. A node may have both stable and non-stable (volatile) storage or just non-stable storage (the former modelling a diskfull workstation and the latter a diskless one). All of the data stored on volatile storage is assumed to be lost when a crash occurs; any data stored on stable storage, as stated earlier, remains unaffected by a crash. It is also assumed that the faults in the communication subsystem are responsible for failures such as lost, duplicated or corrupted messages. Well known network protocol level techniques are available for coping with such failures, so their treatment will not be discussed further.

3. System Architecture

Assume that the system is quiescent such that no applications are currently running. Then all of the objects in the system will be in a passive state, stored in various object stores. An object becomes active once an operation is invoked upon it from within an atomic action (say A). It remains active until such time as A either commits or aborts; if A is nested inside other atomic actions, then the object will remain active until
either the outermost action commits or else A (or an enclosing atomic action) aborts. The procedure for activating an object entails creating a server process on a node and loading the state of the object (the object's image) from the object store into the address space of the server (in volatile store); deactivating an object entails either discarding the current state of the object in the volatile store (if abortion has taken place or the object was not modified) or saving the current state from the volatile store in the object store as the latest version of the object's image (if normal termination has taken place and the object was modified).

The main architectural features of Arjuna will be explained further with the help of Figure 1, which shows

![Figure 1: Objects, Clients and Servers](image)

a system with six nodes, out of which one (N₄) is diskless, and others have disks organised as object stores.
An atomic action which is part of an application being executed by client process C₁ (at N₁) is controlling the access to objects O₁ (at N₁), O₂ (at N₂) and O₄ (at N₄). These objects have been activated; for example O'₂ is an active object, with server S₂ associated with it to receive operation invocations from C₁; O₂ represents the old passive copy of the object in the object store. An object may be activated at a node other than its host; for example, O₄ has been activated at N₄. This form of remote activation provides a simple object caching scheme for fast access. Objects can also be replicated, for example in the figure object O₁ is replicated with copies at N₁ and N₂. The activation scheme for replicated objects will depend upon the replication strategy being employed for that class. For example, in the available copy approach [1], all of the available copies (i.e. the ones on operating nodes) will be activated (as shown in the figure, where O'₁ is the active replica of O₁). The figure also shows another client (C₂) accessing some objects. Both C₁ and C₂ have access to O₄. Multiple activations of an object are permitted provided no state changes are involved (this corresponds to the 'shared read' access). Both C₁ and C₂ have cached O₄ at their local hosts. The execution of any operation may involve further operation invocations on other remote objects, which may result in a client–server hierarchy of arbitrary depth, with some servers also acting as clients; for the sake of simplicity such a deep hierarchy has not been depicted in the figure. At the time of the termination of the client program, the top level client process, such as C₁, will initiate the commit of the action in which all the servers (S₁, S₂, S₃ and S₇ for C₁) will take part.

In order to make this paper self-contained, a brief introduction to type inheritance in C++ is presented in the next section, so that subsequent sections containing C++ code fragments can be understood by the reader.

4. Type-Inheritance in C++

C++ is an object-oriented superset of the language C, and includes facilities for type-inheritance, data abstraction, and operator overloading. The data abstraction and type-inheritance facilities are based on the class concept. Instances of a class are objects, with specific operations provided for their manipulation. The type-inheritance mechanism of C++ works as follows: given a base class C₁, another class C₂—a derived class of C₁—can be defined so that it inherits some or all of the attributes of C₁.

Classes are defined in the manner shown in Figure 2(a) which is a skeleton declaration of a class called baseclass. The variable and function declarations which occur before the protected label (in this example) are private members of the class; the only operations which may access private variables or invoke private operations are the member operations of the class itself (in this example, protop, baseclass, ~baseclass, op1, op2 and op3). The variable and operation declarations following the public label constitute the public interface to objects of the class (here, op2 and op3 in Figure 2(a); the operations baseclass and ~baseclass are special, see below). An example of a class derived from the baseclass class is shown in Figure 2(b). This new class, called derived, inherits the protected and public operations protop, op2 and op3 from baseclass.
class baseclass
{
    int val1;
    int val2;
    void op1();
    protected:
    protop();
    public:
    baseclass();
    ~baseclass();
    void op2();
    void op3();
};

class derived : public baseclass
{
    int val3;
    public:
    derived();
    ~derived();
    void dop4();
    void dop5();
};

Figure 2: C++ syntax

In this example the inherited public operations are also made public operations of the derived class by the use of the keyword public in the class header. Protected attributes may be accessed by members of derived class but cannot be accessed from the outside. Thus, in this example, the operation protop may be used by the member operations of baseclass and derived but not by anyone else.

Each class may have a constructor which is a public operation with the same name as the class (baseclass() and derived()), and which will be invoked each time an instance of the class is created. There is also a complementary operation (~baseclass() and ~derived()), called a destructor, which is invoked automatically when the object is deleted. The constructor allows type-specific initialisation of an object, and the destructor enables an object to tidy up before it is deleted. Both operations are special in that they will be automatically invoked when objects are created or deleted and even though a part of the public interface to the object, they cannot be directly invoked by a user of the object.

5. An Implementation Overview

This section describes the overall implementation of the Arjuna system. Of necessity, several details have either been omitted or are mentioned only briefly; the reader wishing to delve further should consult [4] (for object store and the naming of objects), [7] (for concurrency control), and [8] (for further implementation details, group management and performance evaluation). As stated previously, operations of remote objects are invoked via RPCs. At the application level, objects are the only visible entities; the client and server processes that do the actual work are hidden. In Arjuna, server processes are created dynamically as RPCs are made to objects; these servers are created using the facilities provided by the underlying RPC system Rajdoot [6]. This RPC system has been considerably enhanced to provide support for object group management, such as that required when activating and invoking operations on replicated objects and performing commits and aborts [8]. Rajdoot also detects and exterminates orphans. For example, referring to the scenario depicted in Figure 1, if node N₄ crashes in the middle of the action being executed by C₅, then
the servers on remote nodes \((S_1, S_2, S_3)\) will become orphans, and will be killed automatically (and the relevant objects will be deactivated). A C++ stub generator has been implemented to provide a high level interface to the RPC system, thereby automating the task of object name to location binding, server management and parameter marshalling.

The principal classes which make up the class hierarchy of \textit{Arjuna} are depicted in Figure 3. To make use of atomic actions in an application, instances of the class \texttt{AtomicAction} must be declared by the programmer in the application; the operations this class provides (\texttt{Begin}, \texttt{Abort}, \texttt{End}) can then be used in a suitable manner (see section 6 for an example). The only objects controlled by the resulting atomic actions are those objects which are either instances of \textit{Arjuna} classes or are user-defined classes derived from \texttt{LockManager} and hence are members of the hierarchy shown in Figure 3.

All \textit{Arjuna} classes are derived from the base class \texttt{StateManager}, which provides the primitive facilities necessary for constructing persistent objects and atomic actions. These facilities include support for the activation and deactivation of objects, and object recovery. The class \texttt{LockManager} uses these facilities and provides the concurrency control (two-phase locking in the current implementation) required for implementing the serialisability property of atomic actions. The classes derived from \texttt{AbstractRecord} provide facilities for recording and manipulating action management information regarding the recovery, concurrency control and persistence of specific objects; for example, instances of \texttt{LockRecord} and \texttt{RecoveryRecord} record recovery information for \texttt{Lock} and user-defined objects respectively. The \texttt{AtomicAction} class manages instances of these classes (using an instance of the class \texttt{RecordList}) and is responsible for performing aborts and commits.
For example, assume that O is a user-defined persistent object, which is currently passive. An application containing an atomic action A accesses this object by invoking an operation op₁ of O which involves state changes to O. The serialisability property requires that a write lock must be acquired on O before it is modified; thus the body of op₁ should contain a call to the appropriate operation of the concurrency controller (Figure 4):

```java
{
    // body of op₁
    setlock (new Lock(WRITE));
    // actual state change operations follow
    ...
}
```

Figure 4: Use of locks

The operation setlock is provided by the LockManager class, and it performs the following functions:--
(i) check if O can be locked in WRITE mode; (ii) if so, activate O by using the StateManager operation activate;
(iii) call the modified operation of StateManager (since the Lock is a WRITE lock) causing the creation of a RecoveryRecord and its insertion into the RecordList of A; (iv) create and insert a LockRecord instance in the RecordList of A.

Suppose now that action A is to be aborted. Then the Abort operation of AtomicAction will process the RecordList instance associated with A by invoking the abort operation on the various records. The implementation of this operation by the LockRecord class will release the WRITE lock while that of RecoveryRecord will restore the prior state of O.

The AbstractRecord based approach of managing object properties has proved to be extremely useful in Arjuna. Several uses are summarised here. RecoveryRecord supports state-based recovery, since its abort operation is responsible for restoring the prior state of the object. However, its recovery capability can be altered by refining the abort operation to take some alternative course of action, such as executing a compensating function. This is the principle means of implementing type-specific recovery for user-defined objects in Arjuna. The class LockRecord is a good example of how recoverable locking is supported for a Lock object: the abort operation of LockRecord does not perform state restoration, but executes a releaseLock operation (a compensating operation). Note that locks are, not surprisingly, also treated as objects (instances of the class Lock), therefore they employ the same techniques for making themselves recoverable as any other object. Similarly, no special mechanism is required for aborting an action that has accessed remote objects. In this case, instances of ServerGroupRecord are inserted into the RecordList instance of the atomic action as RPCs are made to the objects. Abortion of an action then involves invoking the abort operation of these ServerGroupRecord instances which in turn send an 'abort' RPC to the servers.
The class declaration in Figure 5 shows the important operations provided by AtomicAction. The three

```cpp
class AtomicAction : publicStateManager
{
    RecordList List; // private instance variables
    ...
    protected: // protected operations
    PrepareOutcome Prepare();
    void Commit();
    public:
    static AtomicAction *Current; // global class variable
    AtomicAction();
    ~AtomicAction();
    virtual Action_Status Begin();
    virtual Action_Status End();
    virtual Action_Status Abort();
    ...
    bool add(AbstractRecord *);
    AtomicAction *Parent();
    ...
}
```

Figure 5: The class AtomicAction

operations Begin, End, and Abort, correspond to the three primitives mentioned earlier. The protected operations (Prepare and Commit) make up the first and second phases of the two-phase commit protocol employed by the End operation. To create an atomic action, an instance of the class AtomicAction must be declared in the program and the Begin operation invoked. To create nested atomic actions, multiple instances of the class are declared so that nesting occurs when a Begin operation is invoked within the scope of another atomic action. Figure 6 illustrates two atomic actions, one of which is committed (A) and the other aborted

```
AtomicAction A, B;
A.Begin(); // start of atomic action A
B.Begin(); // start of atomic action B
B.Abort(); // abortion of atomic action B
A.End(); // commitment of atomic action A
```

Figure 6: An atomic action example

(B). To ensure that nesting is correctly managed and the parent of an action can be traced, AtomicAction maintains a class variable called Current which points to the current (active) atomic action; the operation Parent returns the parent of the current action. Arjuna also supports nested concurrent atomic actions, for which a class ConcurrentAtomicAction derived from AtomicAction has been implemented.

The current Arjuna implementation makes use of the Unix file system for long term storage of objects, with a class ObjectStore providing an object-oriented interface to the file system. The design and implementation of the Arjuna object store (called Kabera) is discussed elsewhere [4], along with the object naming scheme. Here the existence of a stable storage medium is assumed. Such a medium is used for storing
and retrieving uniquely named object instances using the operations write_state and read_state provided by ObjectStore.

To support recoverability, the default mechanism provided by Arjuna is to take a snapshot of the state of an object before it is modified for the first time within the scope of an atomic action. If the atomic action aborts then the old state can simply replace the new, thereby achieving recovery. Persistence has complementary requirements, in that the new state of the object is used to replace the old state held in the object store at commit time. Hence, if mechanisms are provided that enable the state of an object to be collected in such a way that it can be transferred to or from stable storage, then support for both persistence and recovery is possible. Such mechanisms are provided by the class StateManager shown in Figure 7. This

class StateManager
{
    Uid object_uid;
    ...

    protected:
    void modified();
    ...

    public:
    StateManager();
    ~StateManager();

    virtual bool save_state(ObjectState *);
    virtual bool restore_state(ObjectState *);

    bool activate();
    bool deactivate();

    Uid get_Uid();
    ...
};

Figure 7: The class StateManager

class provides a recoverable interface (via the two operations – save_state and restore_state) that operates in conjunction with a class called ObjectState, and includes a unique identifier object which is an instance of the class Uid. Hence, all classes derived from StateManager will inherit the recoverable interface and an operation (called get_Uid) that returns a copy of the object’s unique identifier. As a result, all persistent objects which are instances of classes constructed using StateManager may be named in a uniform manner using unique identifiers. The class ObjectState is responsible for maintaining a buffer into which the instance variables that constitute the state of an object may be contiguously saved.

Figure 8 shows the lifetime and state transitions of a persistent object along with the operations that produce the transitions (operations read_state and write_state are provided by the class ObjectStore, the rest by StateManager). In addition to maintaining the state of an object contiguously, the class ObjectState ensures that the primitive types that make up the instance variables of an object are stored in a form that
may be transmitted between nodes with different architectures. As a result, ObjectState instances may be sent in messages to other nodes, to support object caching.

6. A Simple Example

This section illustrates how Arjuna applications can be constructed using as an example a class that provides the abstraction of a spreadsheet. For simplicity it is assumed that the spreadsheet is represented by a two-dimensional integer matrix with a fixed set of dependencies that state that the last entry in each row (and column) contains the sum of the preceding elements of the row (column). The C++ class definition for this simple style of spreadsheet is given in Figure 9.

This definition shows the additional measures that need to be taken to integrate the Spreadsheet type into the Arjuna system. Firstly, the class is derived from the Arjuna provided class LockManager. This makes the concurrency control, recovery and persistence capabilities of Arjuna available to the implementor of the operations of the Spreadsheet class. Secondly, several Arjuna specific operations are added to the basic set required by the normal implementation of the class. This latter set of operations include the type-specific operations (save_state, restore_state) which are used by Arjuna when saving and restoring instances of this class to and from the object store. The class ObjectState provides a set of pack/unpack operations for most primitive types supported by C++, thereby easing the task of programming the save_state and restore_state operations. In addition, the constructor for the Spreadsheet class takes an instance of the class ArjunaName (which is responsible for mapping a user supplied object name in the form of a string into the corresponding Uid, as discussed in [4]) as a parameter so that the appropriate instance will be automatically activated from the object store when necessary (in the manner described in the previous section).
class Spreadsheet : public LockManager
{
  int Elements [SPRDSHT_SIZE][SPRDSHT_SIZE];

public:
  Spreadsheet (int *, ArjunaName * = 0);
  ~SpreadSheet();

  Outcome Set (int, int, int);
  Outcome Get (int, int, int *);

  // Arjuna specific operations
  virtual bool save_state (ObjectState *);
  virtual bool restore_state (ObjectState *);

};

Figure 9: The Spreadsheet class

Given this simple class definition an implementation of the Set operation is illustrated as Figure 10. This operation sets one element of the spreadsheet to a new value and updates the totals as appropriate. The operation is performed as an atomic action so that it either succeeds or fails in its entirety. Once the parameters to the operation have been verified an atomic action is started, and an attempt is made to lock the object in WRITE (exclusive) mode. If this succeeds then the object will have been activated and loaded from the object store automatically (if necessary) by the Arjuna system, before the active state held of the object is updated.

After the update, an attempt is made to commit the atomic action and if this is successful then the operation returns with a value indicating that it has succeeded, otherwise the atomic action is aborted and the value returned indicates that the operation has failed. In either case no further interaction with the Arjuna system is required since the system will ensure that the modified object state is returned to the object store (providing the atomic action committed) and any locks that were set are automatically released.

This example, albeit very simple, has shown the fundamentals of programming in the Arjuna system. At its simplest, this amounts to little more than deriving the user-defined class from LockManager; providing appropriate definitions for the Arjuna operations save_state, restore_state, etc; setting appropriate locks to govern the level of concurrency the object supports; and declaring and using atomic actions to control the recovery requirements of the class and any application programs.

7. Concluding Remarks

This paper has presented an overview of Arjuna and described how its architecture has been realised using an object-oriented approach. This approach permits the properties of persistence, recoverability, serialisability and failure atomicity to be added to objects in a very flexible manner. A prototype version of Arjuna has been implemented and is currently being extended to include support for available copies based
// Set(x, y, v) changes elements x, y of spreadsheet to have value v.
// If successful returns SUCCESS else type of failure which occurred.
// Performed as an atomic action.

Outcome Spreadsheet::Set (int x, int y, int v) {  
    // Check parameters are ok
    if ((x < 0) || (y < 0) || (x >= SPRDSTS_SIZE) || (y >= SPRDSTS_SIZE))
        return ILLEGAL_ARGS;

    // Parameters are ok - now do the real work

    AtomicAction SetAction;
    Outcome result = SUCCESS;

    SetAction.Begin(); // Begin action
    if (setlock(new Lock(WRITE)) == GRANTED) // and try for lock
        {  // locked - update
            int OldValue = Elements[x][y];
            Elements[x][y] = v; // save old value
            Elements[SPRDSTS_SIZE][y] += v - OldValue;
            Elements[x][SPRDSTS_SIZE] += v - OldValue;
            Elements[SPRDSTS_SIZE][SPRDSTS_SIZE] += v - OldValue;

            if (SetAction.End() != COMMITTED) // try to commit action
                result = COMMIT_FAILED;
        }
    else
        {  // so abort action
            result = LOCK_REFUSED;
            SetAction.Abort();
        }

    return result;
}

Figure 10: The Set operation of Spreadsheet

object replication. Caching has not yet been implemented, but the fact that passive objects are stored in a contiguous representation (instances of the class ObjectState) means that objects can be transmitted within messages. The current implementation of LockManager will need modification to support object caching, it is mentioned here to illustrate that such implementations are possible and provide scope for experimenting with a variety of caching techniques.

The Avalon/C++ system [3] comes closest to Arjuna in its use of the type inheritance approach for implementing atomic actions. Thus the Avalon/C++ classes recoverable and atomic broadly correspond to the Arjuna classes StateManager and LockManager respectively. However, Avalon/C++ is different because it provides an object-oriented interface to an existing transaction processing system (Camelot [9]). Whereas, Arjuna has to implement most of the functionality provided by Camelot as well. For this reason, a closer inspection of the Avalon/C++ and Arjuna classes reveals them to be quite different. Both Avalon/C++ and
Arjuna are good examples of working systems illustrating that type inheritance provides an effective way to construct flexible fault-tolerant and distributed systems.

Both Avalon and Arjuna are systems that support atomic actions. A system which is different from these is ISIS₂ [2] in that rather than supporting serialisable atomic actions, it provides a general set of mechanisms (implemented using a variety of multicast protocols) for ordering of events in a distributed system. ISIS₂ can support applications with dynamic load sharing and online reconfiguration functionality in a clean fashion. This is one of its advantages. On the other hand, for applications which can be composed as nested atomic actions, the functionality provided by ISIS₂ appears to be heavyweight (due to the relative complexity of its multicast protocols). Nevertheless, ISIS₂ provides a very interesting example of a ‘non-transactional’ system supporting fault-tolerance.

The use of C++ in Arjuna has proved to have both advantages and disadvantages. Since Arjuna makes extensive use of inheritance and encapsulation, these features of the language have been of great use. In addition, the ability to overload the meaning of standard operators has also been useful. On the disadvantage side the lack of automatic garbage collection of unreferenced objects has been a problem as has the fact that the primitive types (eg. int) are not classes. The use of a stub generator also places certain restrictions upon the definition of classes. In particular, functions as arguments are not allowed, nor are variable length argument lists. Similarly public members of a class are restricted to being only functions, while the semantics of parameter passing is also restricted to be value-result. Notwithstanding these shortcomings, C++ on the whole has proved to be an effective tool for system building.

Acknowledgements

The Arjuna project has been and continues to be a team effort. Members who have played key roles include Fariba Hedayat, Mark Little and Stuart Wheeler. The work reported here has been supported in part by grants from the UK Science and Engineering Research Council and ESPRIT project No. 2267 (Integrated Systems Architecture).

References


