A Review of Arjuna

Nigel Edwards
Information Management Laboratory
HP Laboratories Bristol
HPL-90-170
September, 1990

object oriented,
distributed system,
fault-tolerance,
C++, transactions,
persistence

This paper reviews the Arjuna toolkit for constructing reliable distributed object oriented systems. It describes the principal mechanisms provided by Arjuna and explains how these can be made available to the programmer in an extremely flexible way. The architecture of programs written using Arjuna and the architecture of the toolkit itself are described. I report on my experiences using Arjuna: the underlying concepts are easy to grasp and appear to be sound. Some suggestions are made for making Arjuna easier to use. The paper also discusses the possibility of porting Arjuna. This would be easier if there was a clean separation between the RPC mechanism and transactions. Finally it examines possible areas for future research using Arjuna.

Internal Accession Date Only

One of the benefits of the object oriented approach to engineering fault-tolerant distributed systems is that inheritance can be used to provide mechanisms and abstractions in a very flexible way. This offers the prospect of having a variety of different mechanisms and abstractions and allowing the programmer to select those most appropriate for a given application.

Arjuna is a toolkit for constructing reliable distributed object oriented systems being developed at Newcastle University. It is one of the first systems to demonstrate some of the benefits of the object oriented approach. This paper examines the mechanisms and abstractions provided by Arjuna, discusses how the flexibility mentioned above can be achieved and looks at the Arjuna infrastructure to see exactly what is required to support these mechanisms and abstractions.

Section 1 explains the principle mechanisms and abstractions provided by Arjuna, and explains how these are made available to the programmer. It goes on to look at the potential flexibility of systems like Arjuna and how new mechanisms and abstractions might be made available to the programmer. Section 2 gives a brief description of the architecture of the toolkit. Section 3 describes the architectures of programs written using Arjuna. Section 4 describes my experiences of using Arjuna and makes some suggestions for making it easier to program with the toolkit. Section 5 reports the current status of Arjuna (July 1990). Section 6 discusses the practicalities of porting Arjuna to different systems. Finally section 7 suggests possible areas for future research using Arjuna.

1 The Mechanisms Provided by Arjuna

Arjuna provides the programmer with mechanisms for persistence, recovery and concurrency control. Transactions are provided to coordinate these mechanisms and structure distributed computations. This section examines these mechanisms and their coordination by transactions in more details. Finally I shall discuss how Arjuna allows the programmer to customise its default mechanisms.

1.1 Concurrency Control

Arjuna provides read and write-locks for concurrency control. Any object derived from the Arjuna class LockManager inherits operations to set and manipulate read and write-locks. If an operation needs exclusive access to an object's state, because it changes the state of an object, it is advisable to set a write-lock before starting the operation. If an operation needs to prevents any state changes occurring whilst it is accessing the object, but does not make any state changes itself, then it is advisable to set a read-lock before starting the operation. LockManager will not grant any other locks whilst a write-lock is set. If a read-lock is set, LockManager will allow new read-locks, but no write-locks are allowed. This locking rule is sometimes known as multiple-readers-single-writer. As section 4 shows read and write-locks can lead to concurrency being severely restricted.

1.2 Recovery

Arjuna provides roll-back state-based recovery. This means that the state of an object is stored so that, should a failure occur during an operation or sequences of operations, the old state can be recovered. Possible alternatives which have been explored in the literature include roll-forward state-based recovery and operation-based recovery. In the former approach recovery takes the failure state of the object and makes it consistent and correct without losing all the state changes which have occurred. In the latter, failure results in an undo operation being invoked to undo the state changes. For example, a "pop" operation could be invoked on a stack object to undo a previous "push" operation. Roll-Back state-based recovery is the most common technique in general use, as it is the simplest to understand and implement. Details of how recovery is implemented in Arjuna are given in [Dixon88] and [Dixon 89].

1.3 Persistence

If a programmer declares an Arjuna object to be persistent, that object will still exist after any computation which has created or accessed it has terminated. The object is assigned a unique identifier (uid) and is stored in the object store. Any object which is derived from the Arjuna class StateManager can be declared to be persistent. This class provides operations to store and retrieve objects into and from the object store. Objects are distinguished by class and unique identifier, so in the current implementation all objects of a particular class are stored in the same UNIX directory and the uid is used to distinguish between objects of the same class.

1.4 Using Transactions to Coordinate Distributed Computations

Programmers can use transactions (also called atomic actions), which are provided by Arjuna, to coordinate the above mechanisms and construct reliable distributed systems. A client object can start a transaction and then invoke a number of operations on server objects. Any locks set in the server objects by these operations will be held until the transaction commits or aborts. The first time an object is accessed within the scope of a transaction the recovery mechanism is used to save the old state of the object. Should a failure occur and the client decides to abort the transaction, roll-back recovery will be invoked on all the objects involved in the transaction and all locks will be released. If no failures occur and the client decides to commit the transaction, then what happens depends on whether or not the transaction is top-level, or nested within another transaction. If the transaction is nested, then its locks are propagated to its parent. If the transaction is top-level all persistent objects are saved to the object store, all recovery information is lost and all locks are released. The commit uses a standard two phase protocol, so that the client, who invoked the operation, can be sure that everyone agrees to commit, or if someone refuses to commit, it retains the right to abort the transaction. The disadvantage of this is that, if a coordinating client fails

during the commit, server objects will be blocked on it. In an heterogeneous system, some coordinating clients may be very unreliable such as a PC which can be easily switched off. In such cases more expensive, but resilient protocols may be appropriate ([Rothermel90]).

Structuring distributed computations using transactions provides the programmer with a number of guarantees.

1. Serializable: Concurrent transactions can be serialized, so the computations packaged within those transactions will not interfere with one another.
2. Failure Atomic: All of the effects of the computation are visible or none of them are.
3. Permanence of effect: The effects of the action will persist after the action has terminated.

There are other possible solutions to the problem of coordinating distributed computations, These include checkpointing, which has recently been studied by the Clouds project ([Dasgupta88], [Lin90]). A completely different approach is taken by Isis ([Birman87]) which provides various communication primitives which can be used to construct distributed computations.

Recent work ([Martin90], [Wheater90b]) has shown that the guarantees provided by the traditional transaction, described above, are not appropriate for all distributed applications. An interesting research problem is how to make these guarantees available in a flexible way so that the programmer can "mix and match" them as appropriate. Another question is what other guarantees would be useful and how they can be packaged in abstractions like transactions to make them useful to programmers.

1.5 Flexibility

Inheritance can be exploited to make mechanisms such as concurrency control, recovery and persistence available in a very flexible way. Each mechanism can be implemented by a different class; an object can use a mechanism if it is derived from the class which implements that mechanism (i.e. it inherits the mechanism). This means that a programmer can select what mechanisms are appropriate for a class, but does not need to pay the price for the mechanisms which are not needed.

Multiple inheritance is needed to make different mechanisms available independently of others, so, for example, an object could inherit recovery by deriving it from one class and persistence by deriving it from another class. Unfortunately, the current implementation of Arjuna does not have multiple inheritance, but nevertheless it does allow the mechanisms for persistence and recovery to be inherited without inheriting the mechanism for concurrency control. This is because, LockManager, the class which provides concurrency control, is

derived from StateManager, the class which provides recovery and persistence. If an object only needs persistence and recovery it can be derived from StateManager, if it also needs concurrency control it can be derived from LockManager. I have not yet had time to experiment with objects which are persistent and recoverable, but do not have concurrency control.

Arjuna also allows programmers to define their own locks. This seems fairly easy to do as locks are implemented as objects in Arjuna. To define a new lock one merely has to derive it from the lock class, which is already provided, and define a lock-conflict operation which specifies how the new lock conflicts with locks already available in the system. This allows the implementation of type specific locks and is a further illustration of the potential flexibility of object oriented systems like Arjuna: if the existing mechanisms are designed with care, it is quite simple to customise them for a given application. (Further details of how to define new locks in Arjuna are given in [Parrington88b].)

2 The Architecture of the Arjuna System

The Arjuna system consists of three main components: core Arjuna, the stub generator and the remote procedure call (RPC) mechanism.

2.1 Core Arjuna

Core Arjuna is a C++ library for constructing reliable distributed object oriented systems. It uses inheritance to provide concurrency control, persistence and recovery. Transactions are provided to coordinate these mechanisms. The current implementation of Arjuna does not support replication, but this is being investigated ([Little90]). Also Arjuna does not yet have a location server. I shall not discuss the design of Core Arjuna further, since it is well documented ([Shrivastava89], [McCue90], [Dixon88], [Dixon 89], [Parrington88a], [Parrington88b]) and the Arjuna team have presented two seminars on the subject in the Labs at Bristol.

2.2 The Stub Generator

The stub generator is a program which takes an Arjuna class and generates its client and server stub classes. Client and server stub objects are used when running distributed Arjuna programs. More information on the stub generation system is given in [Parrington90] and section 3 shows how the client and server stubs are used.

2.3 The RPC Mechanism

The RPC mechanism used in Arjuna is called Rajdoot. Rajdoot is a multi-casting RPC system - it can send RPC requests to more than one server at a time. No optimisations are

made for local communication (an RPC to an object on the same node); all communication uses the same RPC mechanism regardless of location. Rajdoot requires several servers to be installed on each node in the system.

- **mcrdaemon**: This is the multi-cast receive daemon. It should be started just once (preferably at system boot time).
- **manager**: This is the RPC system manager and is responsible for the creation of server processes which execute server objects. It can be invoked with a debugging option in which case logging files will be written so that the progress of computation by the servers it creates can be observed.
- **debugdupserver, debuggmserver, duplicateserver groupmanserver**: These are servers which are started by the manager when they are needed.

More information on Rajdoot is given in [Panzieri88]. The orphan detection and killing mechanisms described in that paper are not implemented in the version used in Arjuna.

3 The Architecture of Arjuna Programs

The architecture of an Arjuna program depends on whether or not it is distributed.

3.1 NonDistributed Arjuna Programs

Nondistributed Arjuna Programs are ordinary sequential C++ programs. They can be compiled and run in the normal way; the RPC mechanism and stub generator are not required, but Core Arjuna is required so that user objects can inherit mechanisms for concurrency control, recovery and persistence. Client objects execute in the same address space as server objects and use the normal C++ mechanism for method invocation. There will only be a single thread of control in such a program; concurrency would be possible if a threads package was available with Arjuna. The only way of running concurrent Arjuna programs (and thereby exercising the concurrency control mechanisms) is to run them as distributed Arjuna programs.

3.2 Distributed Arjuna Programs

A distributed Arjuna program consists of one or more driver programs which act as clients invoking operations on a number of remote server objects. Some server objects may themselves be clients of other, possibly remote, server objects. Every time a remote (server) object S comes into scope in a client C, C invokes the C++ constructor for S's client stub object. This requests the RPC manager process running on the remote node to create a server process. The server process is responsible for executing S's methods and those of its

server stub. A server object, such as S, will have one server process for each client which has declared it; each server process executes in its own separate address space. To invoke an operation on a remote object S a client C invokes an operation on its local client stub object for S. The client stub object for S invokes the RPC mechanism which sends a message to the appropriate server process. The server process receives the remote invocation by executing code in S's server stub object which invokes the required method on S. To invoke an operation on a local object, in the same address space, the standard C++ mechanism is used.

Concurrency control for multiple server processes executing the methods of a server object is provided by deriving the object from the Arjuna object LockManager. To set a lock an object's server process must:

1. Load the object's concurrency control state from the Arjuna Object Store;
2. Check that the desired lock does not conflict with any locks which are already set;
3. Update the concurrency control state so that the new lock is recorded;
4. Store the new concurrency control state back in the object store.

Once the lock is set the Server can load the remainder of the object state from the object store (activate the object). If the server is unable to set the lock it blocks until it can. If the lock is a write lock no other server will be able to set a lock and load the object state from the object store until the write lock is released. The use of the object store to pass concurrency control information for an object between its servers means that the object must be persistent; it is not possible for non-persistent objects to support multiple servers and hence concurrent execution of methods. This implies that non-persistent objects do not require concurrency control in the current implementation of Arjuna.

The manipulation of the concurrency control state is transparent to the programmer of the server object who merely has to invoke the setlock operation provided by the LockManager class. The current implementation prevents an object's server processes from concurrently accessing its concurrency control state by using semaphores which use SUN specific shared memory calls. As Parrington points out in [Parrington88b] the use of the object store to pass concurrency control information does have quite a severe effect on Arjuna's performance. The solution which he suggests is to use a multi-threaded server able to create a new thread to service each incoming call. All threads would share the same address space and would see the same object state, so there would be no need to pass concurrency control information around using the object store. Standard mutual exclusion techniques could be used to ensure that threads had exclusive access to the concurrency control state.

Since one of the main features of the Arjuna system is that it provides the programmer with the ability to program using atomic actions, it is instructive to consider what happens

when a client starts a distributed atomic action and how Arjuna coordinates this distributed atomic action. In Arjuna atomic actions are managed by objects of the class AtomicAction. This class provides begin, end and abort operations for a transaction. A global variable in each address space called CurrentAtomicAction indicates which atomic action, if any, is executing in that address space. When a new atomic action is started by invoking the begin operation of an AtomicAction object, CurrentAtomicAction is set to point to that AtomicAction object. In addition a pointer in the AtomicAction object is set to point to the AtomicAction object which is managing the parent atomic action, if there is a parent. Whenever an RPC is invoked by an object the value of CurrentAtomicAction is sent with the call; if the call is not made within an atomic action then the value sent is null. Before sending an RPC the client stub invokes an operation to check whether or not it is a new atomic action; if it is, an object called ServerGroupRecord is created which stores the address of the remote server. Subsequent invocations to different remote servers will result in their addresses being added to ServerGroupRecord.

When a server process receives an RPC the server stub checks the value of CurrentAtomic-Action sent with the RPC. If it detects a new atomic action it creates a local object of the class ServerAtomicAction. This object is responsible for managing the atomic action at the server and corresponds to the instance of AtomicAction at the client. ServerAtomicAction behaves similarly to AtomicAction except that its commit and abort is controlled by the remote client. When a client wishes to commit or abort a distributed transaction it invokes the commit or abort operation of AtomicAction. This results in the commit or abort operation of ServerGroupRecord being invoked which, in turn, will send commit or abort invocations to all the remote servers whose address it has recorded. Rajdoot's multi-cast facility is used for this.

The original design of this implementation of distributed atomic actions was by Dixon and further details can be found in [Dixon88]. The current implementation differs slightly from Dixon's design: Dixon suggests the use of one ServerActionRecord to record each remote server involved in the atomic action rather than the use of a single ServerGroupRecord to record all remote servers. The advantage of using a single ServerGroupRecord is that multi-casting can be used to do a so-called "flat commit".

4 Programming with Arjuna

This section is based on my experiences of trying to program applications in Arjuna during a recent stay at the University of Newcastle. It points out some of problems that make Arjuna more difficult to use than it otherwise would be, and it makes some suggestions for improvements. It should be remembered that Arjuna is a research prototype and is not intended to be of production quality. The Arjuna team have been very successful in achieving most of their research goals; many of the problems I have encountered are in areas which they have not yet had time to address fully.

The idea of programming with objects and actions was easy to grasp and it seemed quite easy to program. In general, the mechanisms provided for recovery and persistence seemed to work well. One reservation I have is that nesting transactions will lead to very deep nesting of `if ... else` statements which are hard to keep track of and make programs very difficult to read. Figure 1 gives some idea of the level of nesting involved; it shows an assignment operation for a recoverable integer and only involves a single transaction - no nesting of transactions occurs within the operation. It may be possible to use macros to avoid this nesting of `if ... else` statements being made explicit to the reader, additionally a folding editor would also make the programs more readable. Figure 1 also shows that many of the `if ... else` statements are needed to deal with error conditions, such as the transaction failing to abort. If we could find a neater way of dealing with such errors figure 1 would be a lot simpler.

As the following example shows, whenever one wishes to access encapsulated objects it is advisable to set locks on the encapsulating object. Consider a recoverable and persistent queue of recoverable and persistent objects. Both the queue and the objects which it contains inherit the Arjuna mechanisms for concurrency control. Three operations are provided by the queue.

1. *Add* adds an object to the tail of the queue. It sets a write-lock on the queue object since it needs to change the value of a pointer in the queue to point to the new object.
2. *Remove* removes the object at the head of the queue. It sets a write lock on the queue object as it needs to change the state of this object.
3. *Read* reads the value stored in the object at the head of the queue. It sets a read-lock on the object at the head of the queue, but does not set any locks on the queue object itself.

Suppose a client C1 has invoked *Read*. There is nothing to stop another client C2 simultaneously invoking *Remove*, even though C1 has read-locked the object at the head of the queue. If C1 had set a read-lock on the queue object C2 would not be able to obtain a write-lock and would be blocked until the read-lock was released. This severely restricts concurrency: requiring *Read* to set a read-lock on the top level queue object prevents an object being added to the tail until that lock is released.

One solution is to exploit Arjuna's flexibility and define type specific locks which I shall call access and move-locks. Both these kinds of locks would be associated with a particular object in the queue, but would be a lock on the queue object rather than an object in the queue. An access-lock indicates that a particular object in the queue is being accessed. For example *Read* would set an access-lock on the head of the queue. A move-lock indicates that a particular object is being moved into or out of the queue. For example, both *Remove* and *Add* would set move-locks. A move-lock associated with an object in the queue would

```
int RecoverableInteger::assign(int x)
{

        AtomicAction    A;
        int res = UnknownError;

        //start the transaction
        if (A.Begin() == RUNNING)
          {
                //first try to set the lock on the integer
                if (setlock(new Lock(WRITE), 0) == GRANTED)
                {
                  //make the assignment
                  Element = x;
                  res = OperDoneCorrectly;

                  //try to commit the transaction
                  if (A.End() != COMMITTED)
                  res = UnableToCommitAction;
                }
                else
                {
                        res = UnableToSetLockOnRInt;

                        //we haven't got the lock so abort the transaction
                        if (A.Abort() != ABORTED)
                                res = UnableToAbortAction;
                }
          }
        else
                res = UnableToStartAction;

        return res;
}
```

Figure 1: An Assignment operation for a recoverable integer

conflict with all other move and access-locks associated with the same object together with read and write-locks set on the top-level queue object. An access-lock only conflicts with a move-lock associated with the same object in the queue or a write-lock on the top-level queue object.

This problem with encapsulated objects will occur in any distributed object oriented system in which locks are used to provide concurrency control. The general principle appears to be that whenever an encapsulated object is being accessed, that access needs to be flagged in the encapsulating object. If the only way of doing this is by setting read and write-locks on the encapsulating object, concurrency will be severely limited - read and write-locks are too coarse. A convenient way of solving this problem is to use type specific locks as described above. Further study is necessary to see if we can define sufficiently general locks to deal with all cases of encapsulated objects. We can use Arjuna for this study, as it allows type specific locks to be defined.

In the current implementation of Arjuna failing to set locks on encapsulating objects when accessing encapsulated objects is even more dangerous. This is because Arjuna does not activate persistent objects when they come into scope. Instead persistent objects are activated when the first lock is set on them. If *Read* did not set a lock on the top-level queue object we could not be sure that the queue object was active when *Read* was invoked. If it were not active then the pointers to the objects in the queue would not be valid and *Read* would fail.

When trying to write an Arjuna program it is a good idea to write a nondistributed version of it first. It is much easier to debug a nondistributed program, as the execution is confined to a single address space which makes the use of debuggers and other tools a lot easier. Also even the smallest distributed program in Arjuna will have sources in several different files, whereas its nondistributed version may well be confined to a single file. Unfortunately the current version of Arjuna does not have a threads package, so the nondistributed version of a distributed Arjuna program will also be a sequential version. This means that the concurrency control mechanisms will not be exercised until the program is distributed and it is only then that we will see errors such deadlock.

Arjuna does have some facilities for distributed debugging. Debugging versions of the various servers can be used which write logging information as the computation proceeds. There are a few minor problems with the information that is logged: the parameters of an RPC are not printed in a readable form, and the output buffer is not always flushed so occasionally some of the logging information is missing. Apparently it is possible to step through a distributed computation and set up breakpoints by doing some clever tricks with the debugger GDB, but I have not used this. If such a facility was available and easy to use, it could be very useful. However, distributed debugging is intrinsically hard and it is good idea to start with a debugged nondistributed version of the distributed program. Unfortunately, for very large complex distributed applications it may not be practical to have a non-distributed version of the whole application, although it will probably still be possible to test parts of the system

as nondistributed programs. However, the engineering of such systems is an open research topic and further discussion is outside the scope of this paper.

Distribution is transparent to Arjuna objects, but not to the programmer. The client and server objects must be placed in separate files and certain "`#define`" preprocessor directives must be added to declare which client stubs an object needs and to declare whether or not it requires its own server stub. An object will need the client stubs of every object it is to send an RPC to and, if it is to receive an RPC, it will require its own server stub. The programmer must also include the ".h" file generated from an object S by the stub generator whenever S's client or server stub is required. The "`#define`" preprocessor directive is used to make the C++ preprocessor do textual substitution in the client object before it is compiled: all invocations on a remote server object S in client object C are substituted with invocations on S's client stub object. The programmer must remember to link in all the client and server stubs that an object needs when linking the object code. It is advisable to use the "make" utility to automate this and also to place the separate components of a distributed program in separate directories. This inevitably leads to quite complex "Makefiles" which initiate "makes" in various subdirectories even for the simplest of distributed programs. It would be very useful indeed to have some tools to make distributed programming in Arjuna easier. For example it would be helpful to have a tool which generated the Makefile for a component automatically by using the `#define` statements inserted when distributing a program.

Another problem for the programmer is the rather cryptic error messages that are produced by the compiler and the stub generator. (Note that the compiler is a standard C++ compiler). Also although distribution is transparent to objects and the syntax for local and remote operation invocation is the same, some care should be exercised with the parameters of a remote invocation. For example, standard C++ allows variable sized arrays as parameters, whereas the stub generator does not.

There is definitely an Arjuna paradigm for programming. If one does not follow this paradigm it is very easy to run into difficulties. For example, objects encapsulated within objects should be declared as pointers. This paradigm is not yet fully documented although an example is given in [Wheater90a]; it is advisable to follow the style of this example. Further examples and case studies will probably be needed before the Arjuna paradigm is fully understood.

There seems to be some kind of time-out problem in the current implementation of Arjuna which occasionally causes programs to hang for no apparent reason. However, the team are re-implementing parts of the system, so the problem may soon be solved.

5 Current Status of Arjuna and Future Plans

The current implementation of Arjuna was implemented using a mixture of C and C++ compilers because of bugs that existed in each. Arjuna applications can be compiled using g++ (version 1.34.2). This implementation is available on the Suns in HPLB. The entire

system is being rewritten to be compliant with C++ 2.0. This release is due in August 1990.

At present Arjuna does not support multiple inheritance (neither do the early versions of C++), so it is not possible to inherit the mechanisms for persistence, recoverability and concurrency control separately. However, some of the team are investigating a possible redesign which would take advantage of the multiple inheritance supported by C++ version 2.0.

6 Porting Arjuna

Arjuna uses the client and server stubs to coordinate a distributed atomic action. Using the RPC mechanism to do this adversely affects the portability of Arjuna. Porting Arjuna would be much easier if there was a clean separation between core Arjuna and its RPC mechanism, so that Arjuna could easily be run using the RPC facilities provided by other systems, e.g. NCS. The Arjuna team are considering a redesign along these lines, but any implementation is a long way off.

Rather than using a threads package for concurrency Arjuna uses multiple servers to execute an object methods. (No suitable threads package was available when the team started their work.) Object state and concurrency control information is passed between these servers using the object store and system specific semaphore primitives. This increases the complexity of the Arjuna system and makes porting it more difficult.

7 Future Research Using Arjuna

It is the flexibility of object oriented systems like Arjuna which makes them very attractive to study. We can experiment with different mechanisms and abstractions because the system is flexible. By learning how the flexibility is achieved we can build new mechanisms and abstractions which are also flexible. Any new mechanisms and abstractions must be coherent and simple for programmers to use. Different applications will have different reliability requirements, for example work reported in [Martin90] and [Wheater90b] suggests that the traditional transaction model is not appropriate for all applications. Any mechanism and abstractions we provide should take into account the applications they are designed to support.

Arjuna appears to be suitable for small networks (e.g. 10 - 50 nodes) with a few hundred objects. It is not clear whether it could scale to provide fault-tolerance in a world-wide network consisting of thousands of nodes and millions of objects. The question of how to build fault-tolerant applications which are distributed in such networks requires further study.

Another reason for studying Arjuna is to learn about its design, so that we know something about the trade-offs in the design of fault-tolerant systems and where best to place different

parts of the functionality. For example, is it acceptable to use a lightweight unreliable protocol for communication across the LAN and deal with errors in the network by aborting transactions, or should we use a more expensive, reliable protocol?

Acknowledgements

I would like to thank the following members of the Arjuna team for extremely useful discussions: Mark Little, Dan McCue, Graham Parrington, Santosh Shrivastava and Stuart Wheater. I am particularly grateful to Stuart Wheater for tirelessly answering all my questions. Finally I am grateful to the University of Newcastle upon Tyne for allowing me to spend some time in the Computing Laboratory to study Arjuna. Any errors and omissions in this report are solely my responsibility.

8 References

[Dixon88] G.N. Dixon,
Object Management for Persistence and Reliability,
(PhD Thesis) Tech. Report No. 276, December 1988, Computing Laboratory, University of Newcastle upon Tyne, U.K.

[Dixon 89] G.N. Dixon, G.D. Parrington, S.K. Shrivastava, S.M. Wheater,
The Treatment of Persistent Objects in Arjuna,
in proc. ECOOP '89, European Conf. on Object-Oriented Programming, July 1989, Nottingham, U.K.

[Rothermel90] K. Rothermel, S. Pappe,
Open Commit Protocols for the Tree of Processes Model,
in Proc of the 10th Int. Conf. on Distributed Computing, Paris, France, May/June, 1990, p236-244.

[Dasgupta88] P. Dasgupta, R.J. LeBlanc, W.F. Appelbe,
The Clouds Distributed Operating System: a functional description, related work and implementation details,
In Proc. of the Eighth Int. Conf. on Distributed Computing Systems, June 1988.

[Lin90] L. Lin, M. Ahamad,
Checkpointing and Rollback-Recovery in Distributed Object Based Systems,
in Proceedings of the 20th International Symposium on Fault-Tolerant Computing, Newcastle, June 1990, p97-104.

[Birman87] K. Birman, T. Joseph,
Exploiting Virtual Synchrony in Distributed Systems,

in Proc. of the Eleventh ACM Symp. on Operating System Principles, p123-138, November 1987.

[Martin90] B. Martin, C. Pedersen,
Examples of Long-term Concurrent Activities,
July 1990, Hewlett-Packard Laboratories, unpublished working paper.

[Wheater90b] S.M. Wheater, S.K. Shrivastava,
Implementing Fault-Tolerant Distributed Applications Using Objects and Multi-Coloured Actions,
in Proc of the 10th Int. Conf. on Distributed Computing, Paris, France, May/June, 1990, p203-210.

[Parrington88b] G.D. Parrington,
Management of Concurrency in a Reliable Object-Oriented Computing System,
(PhD Thesis) Tech. Report No. 277, December 1988, Computing Laboratory, University of Newcastle upon Tyne, U.K.

[Little90] M.C. Little, S.K. Shrivastava,
Replicated Objects in Arjuna,
Tech. Report, Computing Laboratory, University of Newcastle upon Tyne, U.K. (in preparation).

[Shrivastava89] S.K. Shrivastava, G.N. Dixon, G.D. Parrington, F. Hedayati, S.M. Wheater, M.C. Little,

The Design and Implementation of Arjuna,
Technical Report TR280, Computing Laboratory, University of Newcastle upon Tyne, U.K.

[McCue90] D.L. McCue, S.K.Shrivastava,
Structuring Persistent Object Systems for Portability in a Distributed Environment,
To be presented at the ACM SIGOPS Workshop on Fault Tolerance Support in Distributed Systems, September 3-5, 1990, Bologna, Italy.

[Parrington88a] G.D. Parrington, S.K. Shrivastava,
Implementing Concurrency Control in Reliable Distributed Object-Oriented Systems,
in Proc. ECOOP88, the 2nd European Conference on Object-Oriented Programming, Oslo, Norway, August 1988.

[Parrington90] G.D. Parrington,
Distributed Programming in C++ Via Stub Generation,
in Proc. USENIX/C++ Conf., San Francisco, April 1990.

[Panzieri88] F. Panzieri, S.K. Shrivastava,
Rajdoot: A Remote Procedure Call Mechanism Supporting Orphan Detection and Killing,
IEEE Tran. on Software Engineering, Vol. SE-14, No. 1, pp. 30-37, January 1988.

[Wheater90a] S.M. Wheater,
The Arjuna Programmer's Guide,
Computing Laboratory, University of Newcastle upon Tyne, U.K., (Draft).