A technical overview of Arjuna: a system for reliable distributed computing


Abstract

Arjuna is an object-oriented fault-tolerant distributed programming system which is being designed and implemented in the language C++ on a set of UNIX workstations connected by an Ethernet. Arjuna employs nested atomic actions (atomic transactions) for structuring programs. Programs operate 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 atomic actions, 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 will describe how these mechanisms have been integrated into the object model employed in Arjuna.

Series Editor: M.J. Elphick

© 1988 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.
Bibliographical details

A TECHNICAL OVERVIEW OF ARJUNA: A SYSTEM FOR RELIABLE DISTRIBUTED COMPUTING

Newcastle upon Tyne: University of Newcastle upon Tyne, Computing Laboratory, 1988.
(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 262.)

Added entries

SHRIVASTAVA, S.K.
UNIVERSITY OF NEWCASTLE UPON TYNE
Computing Laboratory. Technical Report Series. 262

Abstract

Arjuna is an object-oriented fault-tolerant distributed programming system which is being designed and implemented in the language C++ on a set of UNIX workstations connected by an Ethernet. Arjuna employs nested atomic actions (atomic transactions) for structuring programs. Programs operate 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 atomic actions, 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 will describe how these mechanisms have been integrated into the object model employed in Arjuna.

About the authors

Professor Shrivastava joined the Computing Laboratory in August, 1975, where he is a Professor.
Mr. G. Dixon is a Research Associate in the Computing Laboratory.
Miss F. Hedayati is a Demonstrator in the Computing Laboratory.
Mr. G.D. Parrington is a Research Associate in the Computing Laboratory.
Mr. S.M. Wheater is a Ph.D. student in the Computing Laboratory.

Suggested keywords

DISTRIBUTED SYSTEMS
FAULT-TOLERANCE
OBJECT-BASED PROGRAMMING

Suggested classmarks (primary classmark underlined)
Dewey (18th): 001.84404  001.6425
U.D.C. 519.587  681.322.06
A TECHNICAL OVERVIEW OF ARJUNA: A SYSTEM FOR RELIABLE DISTRIBUTED COMPUTING


Computing Laboratory
University of Newcastle upon Tyne, England

Abstract

Arjuna is an object-oriented fault-tolerant distributed programming system which is being designed and implemented in the language C++ on a set of UNIX† workstations connected by an Ethernet. Arjuna employs nested atomic actions (atomic transactions) for structuring programs. Programs operate 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 atomic actions, 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 will describe how these mechanisms have been integrated into the object model employed in Arjuna.

1. Introduction

A widely used technique of introducing fault tolerance - particularly in distributed systems - is based upon the use of atomic actions (atomic transactions) for structuring programs [1]. An atomic action possesses the properties of serialisability, failure atomicity and permanence of effect. Arjuna is a distributed programming system currently under development which employs such atomic actions for structuring application programs. Arjuna employs an object model of computation where programs operate on objects which are instances of abstract data types. By ensuring that objects are only manipulated within atomic actions, it can be guaranteed that the integrity of the system is maintained in the presence of failures such as node crashes and the loss of network messages. The paper is structured as follows: section two introduces the object model employed in Arjuna and discusses some design issues; section three presents the architecture of Arjuna, while some implementation details are given in section four; section five summarises the work done so far and presents future plans.

†UNIX is a trademark of AT&T
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 operating upon objects. For example, the architecture for distributed systems proposed by ANSA (a consortium of UK industries) is based upon such a model [2]. What follows is a brief introduction to the basic concepts of this model.

Objects are instances of abstract data types. An object encapsulates some data and provides a set of operations for manipulating this data; these operations are the only means by which an object may be manipulated. In most object-oriented fault-tolerant systems, an operation is invoked on an object using a remote procedure call (RPC), which passes value parameters to the object and returns the results of the operation to the caller (see [3,4,5] for some typical architectures). Programs which operate on objects are executed as atomic actions with the properties of (i) serialisability, (ii) failure atomicity and (iii) permanence of effect. The first property ensures that concurrent executions of programs are free from interference (i.e. a concurrent execution can be shown to be equivalent to some serial order of execution [6]). The second property ensures that a computation can either be terminated normally (committed), producing the intended results or can be aborted, producing no results. This 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 crashes and communication failures such as lost 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 state changes produced are recorded on stable storage, a type of storage which can survive node crashes with a high probability of success. 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 (normal termination), or no updates get recorded (aborted termination) [1]. Some form of concurrency control mechanism, such as two phase locking is also required to enforce the serialisability property of actions [6].

Let us briefly examine some performance and reliability related issues. Fast and reliable RPCs have been a subject of much ongoing research [7,8]. Communication and node failures can result in orphan executions during RPC invocations which, in addition to consuming resources, can interfere with normal computations. Measures employed for treating orphans can be quite expensive, since such measures themselves must be made robust against failures. Our recent effort in reliable RPC design represents one approach where great care has been taken in minimising orphan treatment overheads [8]. Concurrency control protocols necessary for enforcing the serialisability property of actions can severely restrict the degree of permissible concurrency in a system, particularly if long running actions are considered. It is typical for an application to be structured as nested actions. In such a case, locks on objects acquired by inner
actions are retained by the outermost action until the outermost action is itself committed. This means that objects can remain unavailable to other actions for a long time, thus introducing a potential performance problem. What is required is a controlled means of introducing concurrency within objects - for example, by incorporating type-specific locking. Availability of objects can be increased by replication, but it is necessary to ensure that all of the copies of an object remain consistent. Thus, modifications made to an object must be propagated to its replicas. In general, a facility for managing groups of objects (each group containing replicas of an object) is required. This has obvious consequences on RPCs, which should be extended with multicasting facilities for invoking operations on object groups.

Bearing the above discussion in mind, Arjuna has been designed to support group communication and type specific concurrency control and recovery. Group communication is supported in the form of multicast RPCs permitting simultaneous invocations on a group of objects. Type specific concurrency control, recovery and permanence of effect properties of atomic actions are supported by exploiting the type inheritance facilities \[9, 10\] of the implementation language, C++ \[11\]. The use of inheritance in this fashion has a number of advantages. As will be seen, it is not necessary to design and implement either a new language and run time system, nor an operating system kernel.

3. Arjuna Architecture

3.1 Actions

This section presents a brief overview of the architecture of Arjuna (see \[12\] for a more detailed treatment). Atomic actions are instances of the class Action which provides operations for aborting and terminating actions (operations Abort_Action and Commit_Action respectively); actions can be nested and concurrent (as shown in Figure 1).

![Diagram](image)

**Figure 1: Nested Actions**

Assume that there are no atomic actions executing in the system. Then all the objects in the system will be in a passive state. Every node has a stable object
store where passive objects local to that node are stored. An object becomes active once an operation is invoked upon it within an action (say A). It stays active until such time as A either terminates or aborts; if action A is nested inside other actions, then the object will stay active until either the outermost action terminates or else A aborts. Activating an object entails creating a server process and loading the state of the object (the object's image) from the object store into the address space of the server (volatile store); passivating an object entails either discarding the object state in the volatile store (if abortion has taken place) or unloading the current image from volatile store into the object store as the latest version of the image (if normal termination has taken place and the object has been modified). In either case the server process is destroyed.

The above discussion refers to persistent objects. In Arjuna, as indicated before, atomic actions are also treated as objects; however atomic actions are not themselves persistent. In the rest of the paper, unless stated specifically, objects will be assumed to be persistent.

3.2. Object Invocation, Clients and Servers

An operation on an object is invoked by an RPC. 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 RPC system Rajdoot [8]. If an object is replicated, then replicated servers are created (see the next section for some details). Consider the example shown in Figure 2 (here and elsewhere, simple notation will be used rather than depicting complete C++ programs). Assume that the program text for the atomic action shown in Figure 2(a) is being executed at node N1. A client process (P1) will be created at N1 for the execution of this program. The body of this program invokes operations on two remote objects, A (at N2) and B (at N3). When the first operation on A is invoked (A.op1), the RPC mechanism will be used for creating a server process (P2) at N2. Subject to access permission being granted by object A's concurrency controller (see later), P2 will activate object A by loading its image from the object store and will then execute the operation. Any subsequent operations on A from this atomic action (or any other nested actions) are executed by P2 which will passivate the object when the atomic action terminates. A server process (P3) will be created in a similar manner when the operation on B is invoked. The execution of the operation on B gives rise to a nested action containing an operation on A (see Figure 2(b)). This operation will be handled by P2, the server managing A. Thus the execution of the program depicted in Figure 2(a) will be carried out by three processes (Figure 2(c)) with P1 as the main client, P2 as a server of P1, and P3 as both a server of P1 and a client of P2. The commitment of the action invoked by this program (execution of Commit_Action) will involve P1, P2 and P3 taking part in the two-phase commit protocol to be discussed briefly later.
3.3. Concurrency Control, Recoverability and Commitment

Associated with every object is a concurrency control object (an instance of the class LockCC, see later). All of the operations exported by an object must acquire the appropriate access permission by invoking the operations of the associated concurrency control object. Assume for instance that the operation A.op1() in Figure 2(a) modifies the state of A. The implementation of A.op1() must first acquire a write lock on A by calling the concurrency control object which manages A. Once a lock has been obtained by the calling action, subsequent operations on the object do not necessarily require further access control. For example, in the case just considered, once a write lock on A is obtained by the top level action, no further concurrency control is needed. However, assume that A.op1() is of type read, and A.op2() is of type write. Then having secured a read lock, the action must convert it to a write lock before accessing A. The granting of locks and conversion of locks is managed by the concurrency control object.
During the activation of an object, it is first necessary to obtain an appropriate lock on the object before loading the image. Since at any time, at most one action can possess a write lock on an object, the above discipline of activating an object ensures that only a single image of object is available for modification. (If several images were available for modification then clearly inconsistencies could arise.)

In object-oriented systems it is possible to define type-specific lock conflict rules permitting more concurrency than might otherwise be possible. The implementation of this and other types of concurrency control is discussed elsewhere [10,13].

Finally we will see how the abstraction of backward error recovery and the property of permanence of effect has been achieved in Arjuna. The method is based on the type inheritance based scheme published earlier [9] but since then the method has been considerably refined [14]. When an action is to be aborted, the following steps must be executed: the states of all recoverable objects modified within the action must be restored to those at the beginning of the action; any servers created within the action must be deleted; and any newly acquired locks must be released. To perform these operations, relevant bookkeeping information about objects, locks and servers, referred to as action management information, is kept with an action. Each recoverable object is responsible for ensuring that relevant management information about itself gets recorded in the current action (as will be seen, considerable support is provided for the recording of this information). Consider for example that the flow of control is at the point shown by the arrow in Figure 2. At this time, P₁ and P₂ will each have an instance of Action, with the action at P₁ maintaining 'client side' information and the action at P₂ maintaining the 'server side' information. In this particular case, the management information maintained at the server will be the address of the server, while that at the server will have information about object A and its lock. If the action is to be aborted at this juncture, (the Abort_Action operation is invoked by P₁), then the following sequence of steps occurs. The invocation of Abort_Action by P₁ results in 'aborted' RPC to be made to P₂. P₂ restores the prior state of A (this is possible because the management information regarding A contains the prior state of A), releases the lock on A; it then dies.

At this point we can reveal the class hierarchy of Arjuna (see Figure 3). The root class Object provides the basic capabilities which allows a type to be recoverable and persistent. Concurrency control is provided by the class LockCC. Thus a user-defined object, derived from LockCC will inherit properties of recoverability, persistence and concurrency control. These properties are utilised by instances of objects of class Action as discussed below. Instances of the classes Lock_Record, Object_Record and Server_Record contain management information for locks, user-defined objects and servers respectively; these instances are maintained by Action objects. These classes have been derived from a base class Abstract_Record which provides the operations Top_Level_Prep, Top_Level_Commit and Abort (to be used by Action, see below).

The base class Object provides two important types of facilities: (i) it provides operations for storing a user-defined object's state (the object's image)
onto the local stable object store and retrieving a given object's image from the object store; these operations are utilised for passivating and activating an object respectively; (ii) it provides an operation for adding action management information, which is instances of the lock, object or server records, to the current instance of Action. The Abort_Action operation of Action functions as follows: it systematically invokes the Abort operations of all the lock, object and server records maintained within the instance of the Action object. Thus, returning to the previous example, when P₁ invokes Abort_Action, this will result in the Abort operation of the Server_Record being executed. The abort operation of Server_Record makes an "abort" RPC to the relevant server (P₂ in this case). P₂ in turn will invoke Abort_Action, which has the effect of invoking the abort operations of the lock and object records (the abort operation of Lock_Record releases the corresponding lock while that of Object_Record performs state restoration).

Let us, very briefly, see how commitment of an action is mechanised. The operation Commit_Action of Action implements the commit protocol. First of all, the Top_level_Prepare operation of all lock, object and server records maintained in the current instance of Action is invoked; if these succeed, then the corresponding Top_Level_Commit operations are invoked, otherwise Abort operations are invoked. The function of the Top_Level_Prepare operation is to carry out phase one commit activities which requires making a given record instance stable (by storing its image on the object store - this is made possible because all objects are derived from Object).

4. Implementation

Arjuna is currently being implemented on a set of UNIX workstations connected by an Ethernet. Major software components of Arjuna are: a multicast communications layer, a multicast RPC system, a C++ stub generation system, an object store for persistent objects, concurrency, recovery and action
management classes (depicted in Figure 3). Prototype versions of these components have been implemented. Currently they are being integrated to form a complete system. A brief description of some of these components follows.

The multicast communications layer exploits the broadcast facility of the Ethernet for sending a message to multiple destinations [15]. This layer has been used for constructing an enhanced version of the Rajdoot RPC system supporting group communication. Four group management primitives are provided: \textit{initiate} (for creating a specified number of servers), \textit{join} (permitting servers to join a group), \textit{leave} (permitting servers to leave a group) and \textit{terminate} (to terminate a group of servers). Considerable care has been taken to make these primitives failure atomic. A \textit{multicall} primitive is available for sending a call message to a specified group of servers. The C++ stub generator hides the details of server management, and provides the conventional binding and parameter marshalling facilities [16]. The RPC system also supports orphan detection and killing [8]. This makes treatment of any crashes before the commit processing of an action quite simple: any orphans created are detected by the RPC system and aborted. Group communication for commit processing is exploited by ensuring that all processes of an action belong to the same group, thus permitting commit/abort messages to be sent to a group as an entity.

Here is a simplified description of how an operation such as \texttt{A.op1()} (Figure 2) may be coded. Assume that this operation modifies the state of the object and therefore needs to acquire a write lock.

\begin{verbatim}
{
    // body of A.op1
    SetLock(write);
    Modified();
    // the rest of the code
    ...
}
\end{verbatim}

The operation \texttt{SetLock()} (provided by \texttt{LockCC}) performs the following functions: it obtains a write lock on A; it activates the object; and further, it adds an instance of \textit{Lock Record} for this lock to the current action (the last two functions are performed using the operations provided by \texttt{Object} as discussed previously). Before the state of A is modified, it is necessary to ensure that the current state of A gets recorded - this function is performed by the operation \texttt{Modified()} (again provided by \texttt{Object}), which adds an \textit{Object_Record} for A (which includes the current image of A) to the action.

We conclude this section by briefly mentioning \textit{Kubera}, the stable object store of Arjuna [12]. Kubera stores objects of a single class \textit{Image}. An instance of \textit{Image} contains a contiguous block of storage. Any given object must first be converted to an object of type \textit{Image} before being stored in Kubera. The base class \textit{Object} provides a \textit{pack} operation for this purpose, and its converse, an \textit{unpack} operation, for converting an instance of Image to a given object. The current implementation of Kubera uses the UNIX file system.
5. Concluding Remarks

Arjuna is novel with respect to other such systems in taking the approach that every major entity in the system is an object. This approach permits the exploitation of type inheritance for incorporating fault tolerance without the necessity of requiring language and operating system design. Of the other comparable robust object-oriented systems described in the literature, only Avalon [17] is exploring using type inheritance. Although the approach adopted in Avalon is different from that presented here, its aims are broadly similar.

Arjuna is being developed on a present generation of operating system where rudimentary facilities are available for exploiting potential concurrency in distributed systems. In particular, the lack of 'lightweight processes' sharing a common address space has proved a severe limitation. Plans are currently being formulated with the ANSA group for the development of the next version of Arjuna on a more advanced high performance distributed operating system.

Acknowledgements

This work has been supported in part by an Alvey/SERC grant in Software Engineering. M.C. Little was responsible for testing the C++ stub generator.

References


