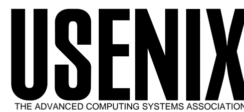
USENIX Association

Proceedings of the 2nd Workshop on Industrial Experiences with Systems Software

Boston, Massachusetts, USA
December 8, 2002



© 2002 by The USENIX Association All Rights Reserved For more information about the USENIX Association:
Phone: 1 510 528 8649 FAX: 1 510 548 5738 Email: office@usenix.org WWW: http://www.usenix.org
Rights to individual papers remain with the author or the author's employer.
Permission is granted for noncommercial reproduction of the work for educational or research purposes.
This copyright notice must be included in the reproduced paper. USENIX acknowledges all trademarks herein.

An Examination of the Transition of the Arjuna Distributed Transaction Processing Software from Research to Products

M. C. Little[1] and S. K. Shrivastava[2]
[1]HP-Arjuna Laboratories, Newcastle-Upon-Tyne,UK
[2]Department of Computing Science, Newcastle University, Newcastle-Upon-Tyne, UK

Abstract

The *Arjuna* transaction system began life in the mid 1980s as an academic project to examine the use of object-oriented techniques in the development of fault-tolerant systems; over 15 years later it is now a Hewlett-Packard product in its own right and is also embedded in several other offerings from HP. In addition, many of the original developers of Arjuna have accompanied the system on its journey and had first hand experience in taking this academic research vehicle into a commercial environment. At times the transition has been neither easy nor smooth but it has been interesting from many different perspectives. In this paper we shall attempt to give an overview of how this occurred and illustrate some of the lessons we have learned over the years.

1. Introduction

The *Arjuna* transaction system began life in the mid 1980s as an academic research project to examine the use of object-oriented techniques in the development of fault-tolerant distributed systems; 15 years later it is a product in its own right and is also embedded in several other product offerings. In this paper we shall examine how this transition was achieved and the lessons learnd along the way.

Arjuna is an object-oriented programming system that provides a set of tools for the construction of fault-tolerant distributed applications. *Arjuna* supports the computational model of *nested atomic actions* (nested transactions) controlling operations on persistent objects. *Arjuna* objects can be replicated on distinct nodes for obtaining high availability. The *Arjuna* research effort began in late 1985 at the University of Newcastle. A version of the system written in C++ to run on networked Unix systems was operational in the early nineties and maintained and made available freely on the Web for research, development and teaching purposes. The arrival of the Web and industrial acceptance of CORBA and Java technologies for distributed object computing during this period encouraged us to productise *Arjuna*. In late 1998 we set up a company, Arjuna Solutions Ltd., with two products derived from *Arjuna*: OTSArjuna, a C++ version of the CORBA Object Transaction Service (OTS) and JTSArjuna, the OTS counterpart in Java. Through a series of company acquisitions, these later became part of HP's middleware product lines. Within HP, the original Arjuna software continues to be of use in creating customised transactional services for new application areas, such as Web Services and mobile computing. In this paper we examine the reasons for the longevity of the Arjuna software.

The paper is structured as follows. In the next section we present an overview of the *Arjuna* system that was implemented in C++. Sufficient details of the system are presented here to enable the readers to follow the subsequent discussions concerning middleware. The material of this section is taken from our published papers on Arjuna [1,2,3,4]. Section three describes how the system was adapted for use as a transaction service for CORBA and Java middleware; here we also compare and contrast the functionality of the original Arjuna system with that of the modern component based middleware. Section four concludes the paper with some lessons we have learnt from our experiences.

2. An Overview of Arjuna

2.1. Design and Implementation Goals

The design and implementation goal of *Arjuna* was to provide a state of the art programming system for constructing fault-tolerant distributed applications. In meeting this goal, three system properties were considered highly important:

(i) *Modularity:* The system should be easy to install and run on a variety of hardware and software configurations. In particular, it should be possible to replace a component of *Arjuna* by an equivalent component already present in the underlying system. As we shall show in the rest of this paper, the modularisation aspect is an important factor in

Arjuna's longevity. At the time, this was accomplished through the use of relatively new object-oriented techniques of interface-implementation separation: each module, which defines a specific well-defined set of functionality, is interacted with through an interface without exposing the underlying implementation details.

(ii) *Integration of mechanisms:* A fault-tolerant distributed system requires a variety of system functions for naming, locating and invoking operations upon local and remote objects, and for concurrency control, error detection and recovery from failures, etc. These mechanisms must be provided in an integrated manner such that their use is easy and natural.

(iii) *Extensibility:* These mechanisms should also be *flexible,* permitting application specific enhancements, such as type-specific concurrency and recovery control, to be easily produced from the existing default ones.

The first goal was met by dividing the overall system functionality into a number of modules that interact with each other through well defined *narrow* interfaces. This facilitated the task of implementing the architecture on a variety of systems with differing support for distributed computing. For example, it was relatively easy to replace the default RPC module of *Arjuna* by Sun RPC and to provide persistence implementations ranging from flat file to non-volatile RAM based. The remaining two goals were met primarily through the provision of a C++ class library for incorporating the properties of fault-tolerance and distribution. Finally, and purely for pragmatic reasons, we decided that it was important to develop *Arjuna* using commonly available tools and hardware: being an academic institution, we had very few funds on which to call for projects.

2.2. Objects and actions

Arjuna supports a computation model in which applications manipulate persistent objects under the control of atomic transactions. Distributed execution is achieved by invoking operations on remote objects using remote procedure calls (RPCs). All operation invocations may be controlled by the use of transactions, which have the well known ACID properties (Atomicity, Consistency, Isolation, Durability). Transactions can be nested; nesting provides fault-isolation: a nested action can abort without causing the abortion of the enclosing action. A (two-phase) commit protocol is used during the termination of an outermost atomic action (*top-level action*) to ensure that either all the objects updated within the action have their new states recorded on stable storage (committed), or, if the transaction aborts, no updates get recorded. It is assumed that, in the absence of failures and concurrency, the invocation of an operation produces consistent state changes to the object. Transactions then ensure that only consistent state changes to objects take place despite concurrent access and any failures.

2.3. System Architecture

With reference to Fig 1, we shall now identify the main modules of *Arjuna* and the services they provide for supporting transactional persistent objects.

1) *Atomic Action.* Provides atomic action support to applications in the form of operations for starting, committing and aborting transactions. It provides a high level API called the *Arjuna Integrated Transactions* (AIT).

2) *RPC.* Provides facilities to clients for connecting/disconnecting to object servers and invoking operations on objects. The initial implementation of this was developed within the Arjuna research group [5] and used novel C++ stub-generation techniques to enhance distribution transparency [6].

3) *Object Store.* Provides a stable storage repository for persistent objects; these objects are assigned unique identifiers (Uids) for naming them.

4) *Naming and Binding.* Provides a mapping from user-given names of objects to Uids and a mapping from Uids to location information such as the identity of the host where the server for the object can be made available.

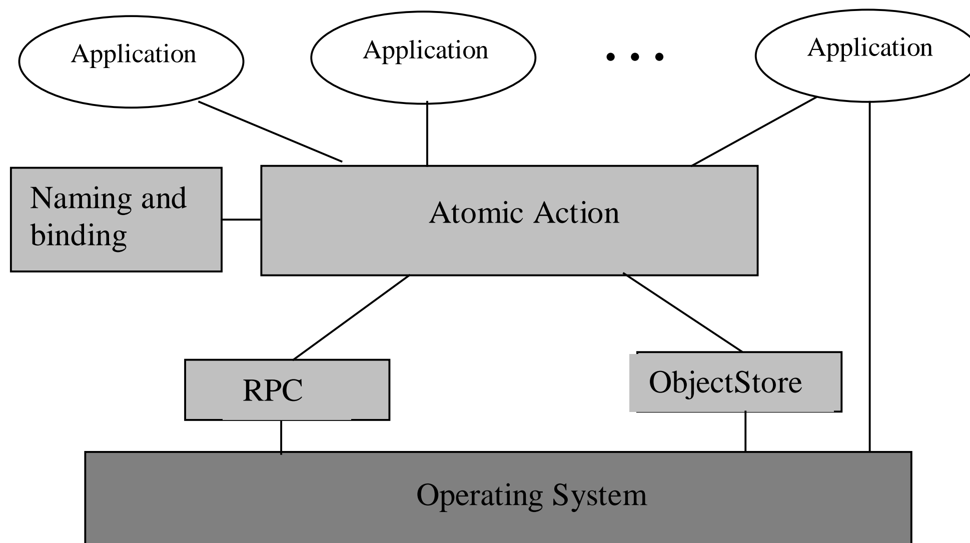


Fig. 1: Components of *Arjuna*.

Every node in the system provides the RPC and Atomic Action modules. Any node capable of providing stable object storage in addition contains an Object Store module. Nodes without stable storage may access these services via their local RPC module. The Naming and Binding module is not necessary on every node since its services can also be utilised through the services provided by the RPC module.

Although atomic transactions guarantee consistency in the presence of failures, they do not provide a means of guaranteeing forward progress: the failure of a machine can lead to the abortion of the transactions using that machine Therefore, as part of the Object Store module, *Arjuna* provides a high-availability option that allows persistent object states to be replicated on an arbitrary number of machines, thus improving the probability that machine failures will not cause a transaction to abort.

All of these modules are accessed through interfaces that allow their implementations to be replaced at runtime. For example, there are multiple interfaces between the Atomic Action and RPC modules offering different, pluggable functionality. When making invocations on remote objects which occur within transactions, it is necessary to propagate information about the transaction (the *context*) to the remote Atomic Action module. Therefore, there are interfaces for allowing the RPC component to obtain and serialise the context on outward calls and to de-serialise the context and recreate the transactions it refers to on incoming calls. The Object Store module has multiple different implementations, each tailored to a specific mode of use (e.g., non-volatile RAM, flat file space or database) and these can be selected on a per object basis.

2.4. Coordinating Recovery, Persistence and Concurrency Control

The atomic action module is the most important part of *Arjuna*. Its design is based on the principle that as objects are assumed to be encapsulated entities, then they must be responsible for implementing the properties required by atomic actions themselves (with appropriate system support). This enables differing objects to have differing recovery and concurrency control strategies. Given this proviso, any atomic action implementation need only control the invocation of the operations providing these properties at the appropriate time and need not know how the properties themselves are actually implemented.

The principal classes that make up the class hierarchy of Arjuna Atomic Action module are depicted in Fig. 2. This is an important hierarchy because, as we shall see, certain classes within it (e.g., `AbstractRecord`) have been critical in the longevity of Arjuna. In addition, the relative ease with which complex applications can be developed using this hierarchy has also helped in Arjuna's success.

To make use of atomic actions in an application, instances of the class `AtomicAction` must be declared by the programmer; the operations this class provides (`Begin`, `Abort`, `End`) can then be used to structure atomic actions (including nested actions).

The only objects controlled by the resulting atomic actions are those objects which are either instances of Arjuna classes or are user-defined classes derived from `LockManager` and hence are members of the hierarchy shown. Most Arjuna classes are derived from `StateManager`, which provides primitive facilities necessary for managing persistent objects. These facilities include support for the activation and de-activation of objects, and state-based object recovery. Thus, instances of `StateManager` are the principal users of the Object Store module and isolate users from its underlying implementations (e.g., database or file based). `LockManager` uses the facilities of `StateManager` and provides the concurrency control required for implementing isolation. The implementation of atomic action facilities for recovery, persistence management and concurrency control is supported by a collection of object classes derived from `AbstractRecord` which is in turn derived from `StateManager`. For example, instances of `LockRecord` and `RecoveryRecord` record recovery information for Lock and user-defined objects respectively. `AtomicAction` manages instances of these classes (using an instance of the class `RecordList` which corresponds to the intentions list used in traditional transaction monitors) and is responsible for performing aborts and commits.

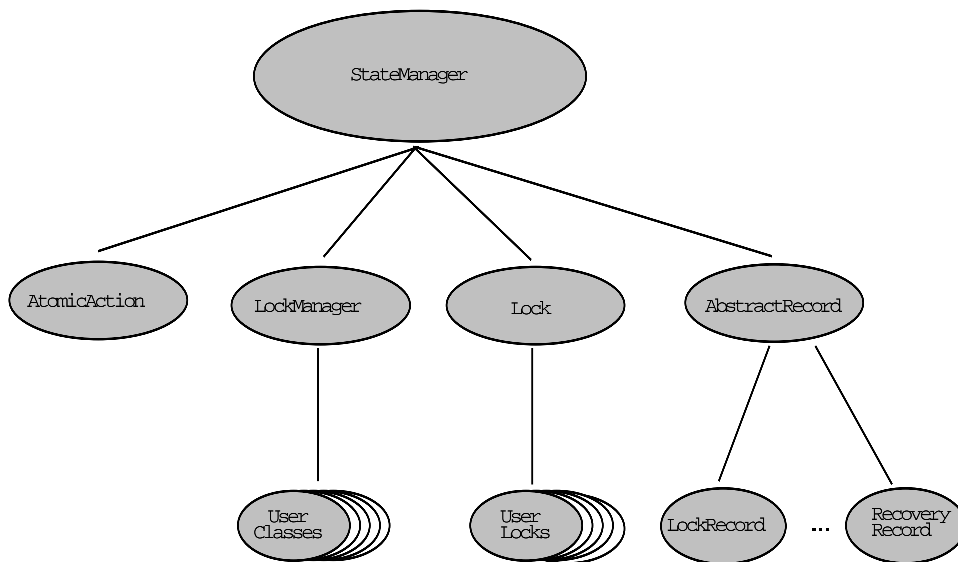


Fig. 2: The Arjuna Class Hierarchy.

Consider a simple example. Assume that `O` is a user-defined persistent object. An application containing a transaction `A` accesses this object by invoking 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 (see Fig. 3):

```
{
 // body of op1
     if setlock (new Lock(WRITE)
=== GRANTED)
     {
       // actual state change
operations follow
        ...
     }
}
```

Fig. 3: The use of Locks in implementing operations.

The operation `setlock`, provided by `LockManager`, performs the following functions in this case:

(i) check write lock compatibility with the currently held locks, and if allowed,

(ii) use StateManager operations for creating a RecoveryRecord instance for O (this is a WRITE lock so the state of the object must be retained before modification) and insert it into the RecordList of A;

(iii) create and insert a LockRecord instance in the RecordList of A.

Suppose that action A is aborted sometime after the lock has been acquired. 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 LockRecord 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. 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. 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 release_lock operation. Similarly, no special mechanism is required for aborting an action that has accessed remote objects. In this case, instances of RpcCallRecord are inserted into the RecordList as RPCs are made to the objects. Abortion of an action then involves invoking the abort operation of these instances which in turn send an "abort" RPC to the servers.

In keeping with the tradition of a university research group, the system as described above was developed and used by several graduate students (including the first author) as a part of their doctoral research work, which included building a distributed database engine for book citations. *Arjuna* was also used in a number of industrial research projects [2], for example use in the telecoms arena for managing bandwidth reservation and connection setup. A particularly demanding application has been the electronic student registration system in use since 1994 by Newcastle University [8]. The registration system has a very high availability and consistency requirement; admissions tutors and secretaries *must* be able to access and create student records (particularly at the start of a new academic year). In addition, the University required any solution to be software based and to run on existing hardware and operating systems, including Unix, Microsoft Windows and MacOS. At that time, no other software based solution existed that could fulfil all of those requirements. Arjuna provided the right set of mechanisms: transactions for consistency and replication for availability. In this particular application, the student record database was triplicated. During the 8 years that the system has been in use, there have been several network and machine failures; with one exception (see below), Arjuna has coped with them all, leaving users unaware that anything untoward has occurred.

The student registration system is composed of two sub-systems: the 'Arjuna sub-system' that runs on a cluster of Unix workstations and is responsible for storing and manipulating student data using transactions, and the 'front-end' sub-system, the collection of PCs and Macs each running a menu driven graphical user interface that users employ to access student data through the Arjuna sub-system The Arjuna-subsystem was engineered to run in a non-partitionable environment by ensuring that the entire cluster of machines was on a single, lightly loaded LAN segment; this decision was made to simplify the task of consistency management of replicated data (as can be appreciated, the problem of consistency management is quite hard in a partitionable environment). The current Arjuna sub-system configuration consists of ten Unix workstations, of which three act as a triplicated database (object store) Within the Arjuna sub-system, great care was taken to ensure that safely (pessimistically) chosen timeouts together with network level 'ping' mechanisms do act as an accurate indication of node failures. During the first year of deployment, we did experience initial consistency problems when one of the replica was inaccurately diagnosed as failed; this led to further testing and adjustment of failure detection timeouts.

Success in meeting the requirements of the registration system was one of the factors that led the Arjuna group to consider turning the system into a product in 1996. By then CORBA and Java middleware were attracting industry attention. The OMG architecture, CORBA [9] was well established, so it seemed natural to adapt *Arjuna* to meet the specification of the Object Transaction Service (OTS) and later to the Java Transaction Service (JTS) that is optional within the Java component middleware, J2EE [10]. In the next section we describe how this was achieved.

3. Arjuna and Middleware

3.1. Basic middleware concepts

We present a brief overview of middleware concepts, using CORBA as an example. Fig. 4 depicts the main elements of the CORBA middleware. It consists of an 'object bus', the object request broker (ORB) which allows client to interact with remote objects. A number of services are available for facilitating this task; these include naming, persistence, event notification and transactions. JAVA/RMI is a broadly similar Java language specific middleware.

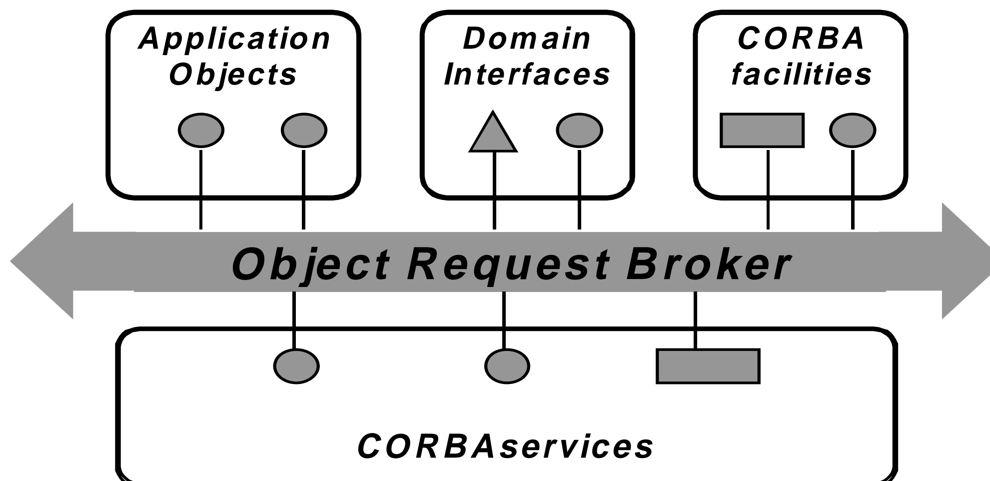


Fig. 4: CORBA middleware.

Although this middleware simplifies the construction of distributed applications by providing type checked remote invocations and standard ways of using commonly required services, there is still the problem that programmers have to worry about application logic as well as technically complex ways of using a *collection of services*. For example, transactions on distributed objects require concurrency control, persistence and the transaction services to be used in a particular manner. To address this difficulty, object-based middleware has been extended to component-based middleware. In simple terms, a component is an 'application object with the capability for using middleware services in a standard manner'. A component is hosted by a *container* (a server process), and normally the container uses the underlying middleware services on behalf of the application object. A component descriptor specifies, in a declarative manner, the services that are required by the component. Containers are provided by *application servers* that provide tools for deploying components onto containers using the information specified in the descriptors. A J2EE Enterprise Java Bean (EJB) is a good example of a component. In the rest of this section we will first examine how OTSArjuna and JTSArjuna were implemented. Secondly, bearing in mind that Arjuna is not just a transaction service, but a complete system for building transactional applications, we will compare and contrast its functionality with that of J2EE application servers.

3.2. Arjuna, the OTS and the JTS

In 1995 the industry standard for distributed transactions changed from being predominately based on X/Open XA [7], which is procedural-oriented, to the OTS (OMG Object Transaction Service) [9]. This was based on the experiences of all of the major transaction processing vendors, including IBM, HP and DEC. Around this time, the *Arjuna* group attracted R&D funding from industry led consortia and members of these consortia expressed interest in standards-related developments. In particular several of our industrial sponsors were interested in funding transactions research and development within a CORBA environment and the original Arjuna system, with its own RPC and Naming and Binding implementations, did not meet these requirements.

The OTS is a *protocol engine* that guarantees that transactional behaviour is obeyed but does not directly support all of the transactional properties. As such it requires other services that implement the required functionality, such as persistence and concurrency control. The application programmer is responsible for using these services to ensure that transactional objects have the necessary ACID properties.

In addition, the OTS does not provide any participant implementations (e.g., database resource managers). These must be provided by the application programmer or the OTS implementer. As such, a pure OTS implementation actually provides much less functionality than that available in *Arjuna*. The OTS does not define a complete toolkit for the construction of transactional applications: it has no equivalent of AIT.

Examining just the transaction engine component of *Arjun*a as provided by the Atomic Action module, it was clear that there was a good match with the OTS. Although the interfaces that are exposed by the OTS were different, it was relatively straightforward to provide these same abstractions on top of the equivalent *Arjuna* APIs. Most effort was directed at fully integrating *Arjuna* (now *OTSArjuna*) within the CORBA framework, e.g., how to do distributed invocations and ensure that the transaction context is passed as mandated by the specification. The interfaces we had defined between the Naming and Binding and RPC modules were sufficiently powerful that only minimal modifications had to be made. In addition, the interfaces allowed *OTSArjuna* to be ported to a variety of different ORBs rather than being tied to one specific implementation. This was an important aspect as other OTS implementations required users to own a specific ORB (usually the one from the same vendor), e.g., IBM WebSphere and BEA WebLogic. Being able to choose the vendor of different components in a distributed architecture was important to many prospective users.

The architecture of *OTSArjuna* is shown in Fig 5, where the dark grey boxes comprise *OTSArjuna* and everything else is either provided by the application programmer/user or other software components such as database drivers (Resource/SubtranAware, for example). The OTS protocol engine, State Management and Concurrency Control components are essentially exactly as they appeared in the original *Arjuna* Atomic Action module. AIT was provided to programmers via the *OTSArjuna* API. The RPC and Naming and Binding modules from the original architecture were replaced by whatever the underlying ORB implementation provided. The existing Object Store module was made available via suitable Resource/SubtranAware implementations.

Importantly, to maintain the original pluggable abstraction, any OTS-specific modifications that were made (such as API updates) occurred in self-contained modules accessed through well defined interfaces.

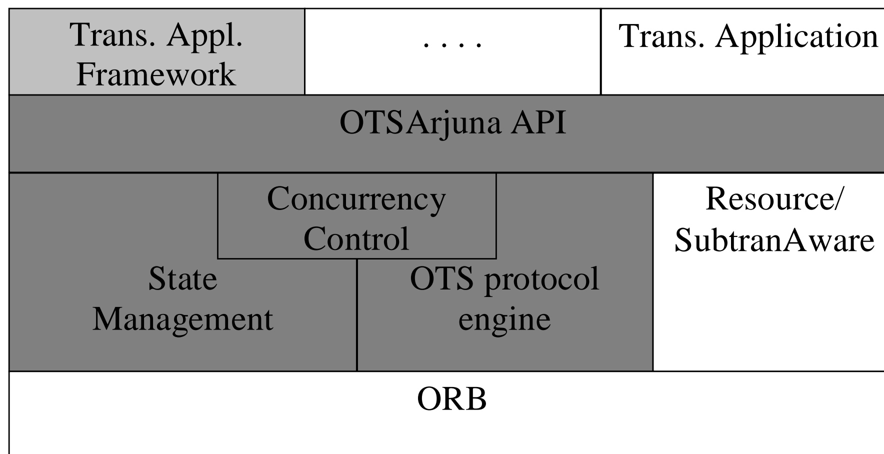


Fig. 5: OTSArjuna architecture.

Remote participants, XA compliant databases etc. were all transparently controlled by the core via `AbstractRecord` implementations: the transaction protocol engine could not tell that it was now running within a CORBA environment. However, there was one notable exception where the original *Arjuna* abstraction we had simply did not work: *failure recovery*.

To guarantee ACID properties in the event of failures, a recovery subsystem is required. This ensures that any transactions that were in progress are completed, either by being committed or rolled back. In order to do this, it may be necessary to recreate any resources that were participants within the transaction: the recovery system will recreate the distribution tree that was present prior to the failure. In order to achieve this, recovery must have intimate knowledge about the resources (e.g., do

they use a file system for persistence?) and the RPC mechanism (e.g., what is the machine the object resides on?) The *Arjuna* failure recovery implementation was closely tied to the original RPC mechanism. We were therefore unable to take this component or to reuse the interfaces it provided. As such, we were forced to re-implement recovery and in doing so we tied it to the OTS model for expediency.

By 1996 Java started attracting serious attention from industry and many existing OMG standards made their way into J2EE. Critical amongst them was the OTS. *OTSArjuna* was developed using C++, which made the task of converting to Java relatively straightforward. Arjuna had been developed with the very earliest versions of C++, which had no multiple inheritance and hence this helped facilitate the language transition. We had made extensive use of C++ templates and reference parameters, neither of which had their equivalents in Java at that time and this did require us to expend effort in re-coding. However, none of the language differences presented issues that were substantial enough to require redesigning large portions of the code.

JTSArjuna, became the worlds first 100% pure Java transaction system. This system and the group which helped develop it became part of Hewlett-Packard's middleware division through a series of acquisitions. After two years of intensive work on turning an academic system into a product (mainly increasing the team from 5 to 24 people, of which nearly 40% were dedicated to quality assurance, and putting the product through much more rigorous testing than it had ever been through before), the *Hewlett-Packard Transaction Service* was created and became an integral part of their middleware offerings. Interestingly, because of its long history of development and use, the number of issues (bugs) thrown up by quality assurance was relatively small for such a complex system.

3.3. Arjuna and J2EE

In many ways, the programming model and environment presented by Arjuna has many things in common with modern day application servers. In this section, we shall compare and contrast Arjuna with the J2EE application server architecture and try to have an objective examination of why differences exist.

A simplified J2EE application server architecture is shown in fig. 6. Although this depiction hides many details, it should be sufficient for a high-level comparative analysis. It has very similar components to *Arjuna*, but utilises industry standard technologies for the object invocation mechanism and naming and binding. Although the interfaces to the various components are different to those in *Arjuna*, the fact that these components exist in an identifiable (and typically replaceable) manner is testimony that the original *Arjuna* architecture and design goals were sound.

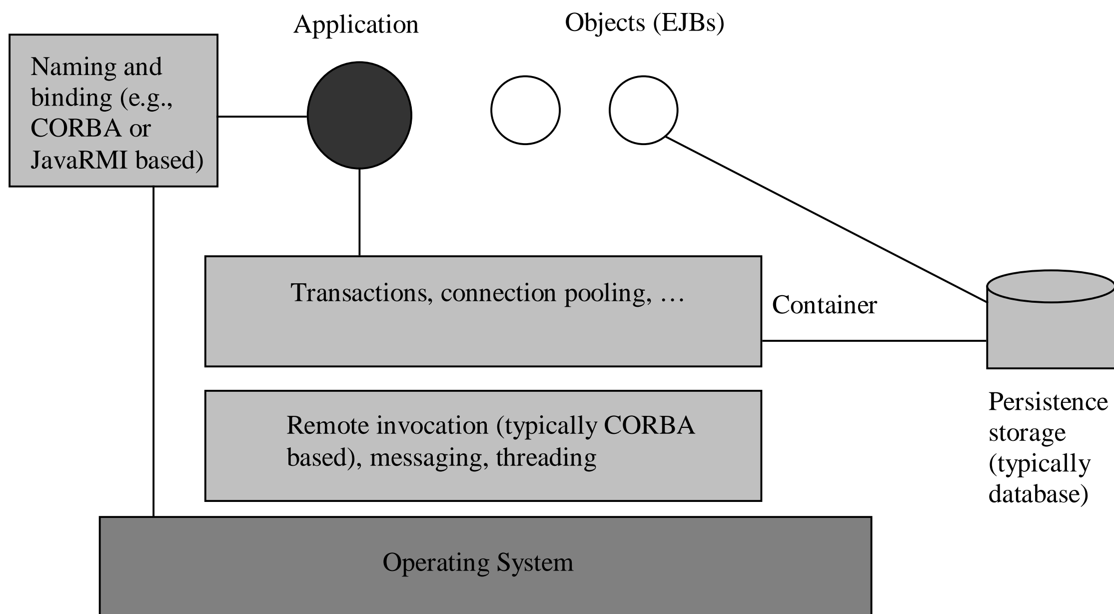


Fig. 6: Simplified application-server architecture.

A feature that is missing in *Arjuna* is the declarative way of managing transactions that is provided by EJBs. EJBs also provide explicit transaction management using a high level (compared to JTS) API, called Java Transaction API (JTA); however, it is not as convenient to use as AIT, and its use is not generally recommended [11]. Compared to *Arjuna*, EJB transactions come with several restrictions:

(i) *Participant restriction*: both the original *Arjuna* system and the OTS, allow arbitrary participant implementations. The AbstractRecord interface and the OTS equivalent do not mandate a specific implementation. This allows recovery, concurrency control etc. to be transparently enlisted with a transaction. However, the JTA interface restricts participants to being XA-aware. This effectively mandates that applications must use databases for persistence: other types of persistence are possible, but a lot of effort is necessary from the developers in order for the implementations to be driven by the JTA, since whatever persistence implementation is used, it must be XA compliant. Several users of *Arjuna* and its descendant products have found the fact that persistence implementations are configurable and not bound to a specific model (e.g., XA) extremely useful, especially in the area of ease of deployment.

(ii) *No nested transactions*: if an object's methods are required to use transactions then, in an environment which supports nested transactions, the implementer can use transactions without concern about how those methods will be used: if the invoker uses transactions, then the methods will be nested within them, otherwise they will be top-level. In addition, nesting provides a level of failure containment, since the failure of a nested transaction does not require the enclosing transaction to roll back. The JTA does not support nested transactions because XA does not. Once again, user feedback on nested transactions has shown that they are an extremely useful structuring tool, especially when developing large-scale systems which may have many engineers working on them.

(iii) *Poor concurrency control*: Unfortunately, no satisfactory way of using read and write locks is available in EJBs. Because the JTA mandates that all participants must be XA-aware, this ties the persistence model to using a database. Most databases implicitly couple persistency and concurrency together, such that, for example, when an object loads its state it obtains a lock on the entire database table, which is maintained for the duration of the transaction. All other object states held within the same table are also implicitly locked. In order to provide object-level concurrency control, the programmer is supposed to make use of the Java language `synchronized` construct, which obtains an exclusive lock on a method. However, this construct is not transaction-aware and as such cycles (where object A calls object B which calls object A) can result in deadlock and are

illegal. Because AIT locks are transaction-aware, not only can an object use multiple-reader/single-writer policies, but cycles within a transaction are supported. This makes the construction of complex, distributed applications more straightforward as programmers need not worry about whether cycles may occur, which could require in-depth knowledge of objects implemented by others.

(iv) *No support for orphan detection and elimination*: Client crashes or network partitions can occasionally create orphan servers.. Orphans are undesirable as they consume resources, and need to be destroyed. The RPC mechanism used in *Arjuna* detected and eliminated orphans [5]. Experience with the use of application servers has indicated that orphans do occur in practice; unfortunately no automatic support for orphan detection and elimination is provided in application servers (or any other middleware system for that matter). Given our experience with the Student Registration system and phantom machine failures, this lack of orphan detection is worrying.

4. Further evolution

The increasing use of the Web for commercial activities has led to a paradigm shift from closely-coupled, synchronous systems to loosely coupled asynchronous systems. Several researchers (e.g. [12,13]) have argued that in the Internet/Web environment, a practical way of gluing applications is through loose coupling as provided by asynchronous messaging. The main reason behind this is that asynchronous communication de-couples producers of information from consumers; they do not need to be both ready for execution at the same time.

The one constant amongst traditional and new distributed environments is their requirement for transactions. Irrespective of whether applications are closely-coupled or loosely-coupled, failures happen that affect both the performance and consistency of applications run over them. Transactions can be used in all of these environments to ensure consistency and specifically within Hewlett-Packard, Arjuna transaction technology has been used.

4.1. ArjunaCore

As part of the Hewlett-Packard NetAction product suite, there was a requirement for both transactional messaging (based on the JMS specification) and Web Services transactions implementations (based on the OASIS Business Transactions Protocol [14]). The existing transaction products covered the J2EE and CORBA arenas using a two-phase commit protocol. When comparing the functionality provided by *JTSArjuna* with that required by the JMS, for example, it was clear that there was much overlap. In fact, it was possible to categorise all of the transaction products with the following requirements:

- The use of a two-phase commit protocol.
- They carry transaction context information in a manner suitable to their environment, e.g., XML and SOAP for BTP, or IIOP for CORBA.
- Their transaction participant implementations are opaque to the two-phase transaction engine.

Careful examination showed that it appeared possible to use the same *core* protocol engine that had been within the original *Arjuna* system and was now within *JTSArjuna*, within HP's BTP (HP Web Services Transactions) and JMS (HP Messaging Service) implementations. The interfaces (e.g., `AbstractRecord`) isolate the coordinator from the specifics of participant implementations and how transactional distributed invocations occur. By providing different implementations of`AbstractRecords`, for example, it was possible to drive BTP Web Service participants or JMS participants through a two-phase commit protocol using the *exact* same transaction engine.

Therefore, the work that was performed to transform *Arjuna* into *JTSArjuna* was generalised. The first step was to create a fully-functional transaction engine that had *no* dependencies on CORBA (including failure recovery) or any distribution infrastructure. This created *ArjunaCore*, which is concerned solely with the use of local transactions, i.e., transactions that run on a single machine. If distributed transactions are required, *ArjunaCore* provides the necessary hooks to enable information about its transactions to be transmitted in a manner suitable for the environment in which it is running, e.g., CORBA IIOP or SOAP/XML. Systems that use *ArjunaCore* for their transaction requirements are then required to utilise these hooks in order to obtain the context information and then transmit it in a suitable manner.

The main obstacle to the design of *ArjunaCore* was the failure recovery sub-system. When designing *OTSArjuna* it had been closely tied to the CORBA OTS model. In order to determine transaction statuses, it was a requirement that *all* transactions were implemented by CORBA objects, whether or not they were used in a distributed manner. By re-examining

the failure recovery architecture, a more modular approach was taken, that was essentially based on the original Arjuna goals: recovery is directed through an abstract interface (the RecoveryModule) that does not imply specific implementations. Each RecoveryModule is responsible for recovering specific types of resources (transactions, application objects etc.) without exposing implementation choices such as whether or not CORBA was used, to the recovery framework. Since *ArjunaCore* is responsible for only local transactions, its default RecoveryModule implementations are relatively simple. Other products in which *ArjunaCore* is embedded, e.g., BTP, provide suitable implementations to do distributed recovery where necessary.

With the benefit of hindsight, this abstraction should have been used from the start when creating *OTSArjuna*. The reason it was not was basically that, in an effort to productise the system we believed that the OTS was to be the final destination for *Arjuna*: CORBA was being widely adopted as the de facto middleware standard and J2EE helped to propagate that myth. As such, there did not seem to be any reason to believe that Arjuna would be used anywhere except within an OTS environment and hence no requirement for such flexibility in the recovery sub-system. Obviously this did not turn out to be the case and we paid the price in time and effort spent in re-implementation and backward compatibility support.

Despite the necessary redesign of the failure recovery subsystem, the remainder of *ArjunaCore* is the same as existed in the original *Arjuna* system. The interfaces have proven sufficient to allow the system to be the core transaction engine within a diverse range of products. The fact that it is no longer reliant on a specific distribution infrastructure makes it extremely small (less than 250 kilobytes compared to almost 1 megabytes for *JTSArjuna*) and embeddable: experiments have been performed to port ArjunaCore onto mobile devices (PDAs) something which other industrial transaction products would find extremely difficult to accomplish.

5. Concluding remarks

Thus, after nearly 15 years of design and evolution, Arjuna has evolved through its various OTS/JTS incarnations, to a stand-alone, non-distributed transaction engine. Of the original architecture shown in fig 1., only the Atomic Action and ObjectStore modules remain. However, of these, the majority of the code that was written while in academia continues to exist within the *ArjunaCore* product.

Figure 7 attempts to illustrate the history of Arjuna as a timeline, showing the relevant events we have discussed previously from its start in 1986 to the present day.

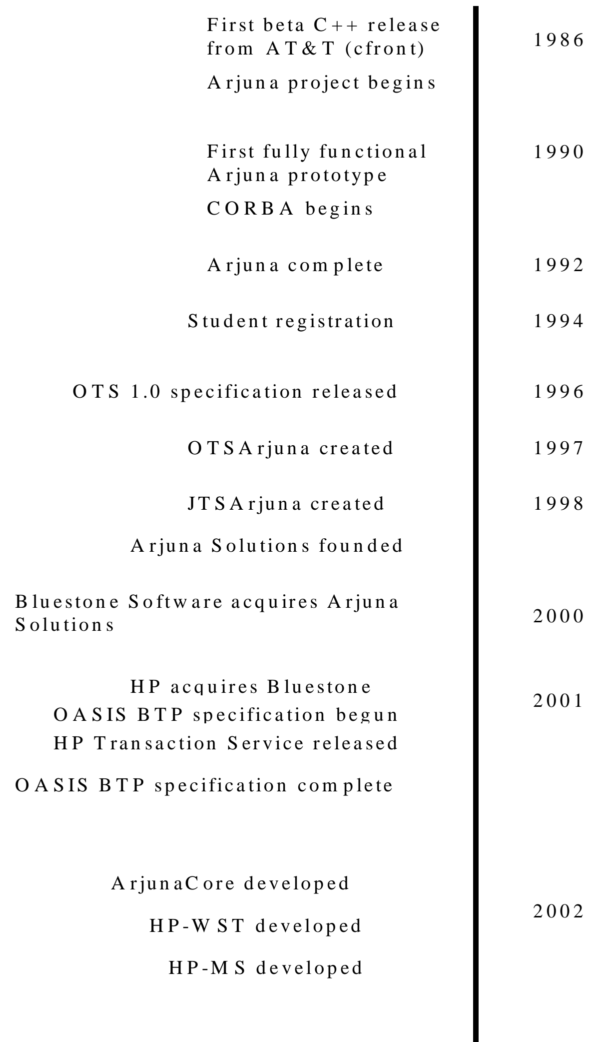


Fig 7. The evolution of Arjuna.

In achieving the transition of the Arjuna distributed transaction processing software from research to products, we have learned a number of lessons, some of which will be relevant to others involved in or embarking on a similar process. We shall attempt to enumerate them below:

- Modularity within the architecture helped us to restructure the uses to which we put Arjuna without requiring re-implementation of the entire system. As we have discussed, the core transaction

engine available today remains relatively unchanged from its original C++ version.

- The use of object-oriented techniques helped to make the structuring of the architecture flexible and extensible. It also helped to make its use relatively intuitive for new developers. A crucial factor has been the structure of the atomic action module for coordinating concurrency, persistence and recovery for atomic actions using `AbstractRecords` (section 2.4), which meant that transaction coordinator need only control the invocation of the operations providing these properties at the appropriate time and need not know how the properties themselves are actually implemented.

- A commercial product requires a lot more emphasis on quality assurance (QA) and testing processes than a research system. At the time of writing, the number of QA tests for Arjuna number in the thousands, cover every aspect of the system and can take days to run to completion. Only with the evidence of these tests is it possible to convince people to invest time and money in purchasing the product.

- Within each component there are typically many places where configuration choices are made (e.g., the location of the object store, the maximum size of the transaction log before the system begins to prune it, etc.) When Arjuna started, many of these choices were hardwired in at compilation time. Over the years (and particularly when it became a product) the requirement for these choices to be exposed to developers was intensified. With hindsight, as designers of the system we tended to cater for the optimum configuration for ourselves and this was often inappropriate for others.

- At the time of writing, JTSArjuna is used within 5 separate products and has been sold to 3 other companies to embed within their own products. It is impossible to say with certainty how many users it has, but it has brought millions of dollars to the various companies that have sold it.

- Once we were acquired by Bluestone, the use of JTSArjuna increased significantly and hence so did the support and training load put on the developers. We quickly realised that a commercial product is much more than the software that actually executes: there is a significant amount of collateral material required too, e.g., training material, white papers etc.

- The biggest mistake we made was in the development of crash recovery for OTSArjuna and tying it to the CORBA model. With hindsight it is possible to see that Arjuna could be used in other, non-CORBA environments and we should have designed accordingly.

- Commercial requirements on robustness of software systems are far more rigorous than you would expect from research prototypes and this was the probably the hardest aspect for us to tackle.

- Working with various standards bodies (OMG, Java Community Process, OASIS to name but a few) has been fruitful but also extremely frustrating at times (committees rarely agree on anything, especially if there is existing product to protect).

The discussion in section three indicates that the original structure of the Arjuna system was sound for its adaptation in various types of middleware. The 15 year journey from academic project to commercial product has been an interesting one and enlightening in many respects. And finally, the most surprising thing has been the amount of use to which Arjuna has been put over the years.

Acknowledgements

The *Arjuna* system is the product of a diverse and varying team over many years. We would particularly like to mention the efforts of Steve Caughey who managed the transition process from academic project to industrial product, Dave Ingham for his work on the JMS, Stuart Wheater for the QA and testing work and Jim Webber for his work on the BTP implementation. The research work at the University was funded by many grants from the UK Engineering and Physical Sciences Research Council, the European Commission and industry that included Marconi, Nortel, IBM and HP.

We would also like to thank the anonymous reviewers and Doug Terry of Microsoft, our shepherd for this paper, all of whose comments helped to improve it.

References

[1] S.K. Shrivastava, G. N. Dixon, and G. D. Parrington, “An Overview of Arjuna: A Programming System for Reliable Distributed

Computing," IEEE Software, Vol. 8, No. 1, pp. 63-73, January 1991.

[2] S.K. Shrivastava, "Lessons learned from building and using the Arjuna distributed programming system", Theory and Practice in Distributed Systems, K P Birman, F Mattern, A Schiper (Eds), LNCS 938, Springer-Verlag, July 1995, pp. 17-32

[3] S.K. Shrivastava and D. McCue, "Structuring Fault-Tolerant Object Systems for Modularity in a Distributed Environment", IEEE Trans. on Parallel and Distributed Systems, Vol. 5, No. 4, April 1994, pp. 421-432.

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

[5] F. Panzieri, and S.K. Shrivastava, "Rajdoot: a remote procedure call mechanism supporting orphan detection and killing", IEEE Trans. on Software Eng. 14, 1, pp. 30-37, January 1988.

[6] G.D. Parrington, "Reliable Distributed Programming in C++: The Arjuna Approach," Second Usenix C++ Conference, pp. 37-50, San Fransisco, April 1990.

[7] X/Open Reference Model, Version 3, X/Open Ltd. 1996.

[8] M. C. Little, S. M. Wheater, D. B. Ingham, C. R. Snow, H. Whitfield and S. K. Shrivastava, "The University Student Registration System: a Case Study in Building a High-Availability Distributed Application Using General-Purpose Components", Chapter 19, Advances in Distributed Systems, Springer-Verlag, LNCS No. 1752.

[9] OMG, CORBAservices: Common Object Services Specification, Updated July 1997, OMG document formal/97-07-04. www.omg.org

[10] Java 2 Enterprise Edition (J2EE) specification, www.javasoft.com

[11] R. Monson-Haefel, "Enterprise Java Beans", O'Reilly & Associates, CA, 2001.

[12] K. Mani Chandy and Adam Rifkin Systematic Composition of Objects in Distributed Internet Applications: Processes and Sessions. Computer Journal in October 1997

[13] G. Banavar, T. Chandra, R. Strom and D. Sturman, "A case for message oriented middleware", Proceedings of the Symposium on Distributed Systems, DISC99, Bratislava, September 1999, LNCS. Vol. 1693.

[14] Business Transaction Protocol specification, http://www.oasis-open.org/