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

1. Introduction

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 this paper we shall present an overview of this journey and illustrate some of the lessons learnt.

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 atomic transactions) controlling operations on persistent (long-lived) 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 early nineties and was maintained and made available freely by us on the Web for research, development and teaching purposes during most of the nineties. 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 the Arjuna software: 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.

Designing and implementing a programming system capable of supporting 'objects and actions' based applications by utilising existing programming languages and operating systems was a challenging task for us in 1985, as research on distributed object systems and architectures was only just beginning. Furthermore, basic services for distributed computing (naming, binding, object invocation, etc.) now commonly available on ORBs, were non existent then, so had to be built from scratch. We began by assuming that every major entity in the application will be an object. This philosophy also applied to the internal structure of *Arjuna* itself. Thus, *Arjuna* not only supports an object-oriented model of computation, but its internal structure is also object-oriented. This approach has permitted the use of the type inheritance mechanism of object-oriented systems for incorporating the properties of fault-tolerance and distribution in a very flexible and integrated manner. This structure together with the modular way the system was put together are the main factors that has enabled its continued use.

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.

2. An Overview of Arjuna

2.1. Design and Implementation Goals

The design and implementation goals 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.

(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) *Flexibility:* 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.

In *Arjuna,* the first goal was met by dividing the overall system functionality into a number of modules which 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 a different one. 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.

2.2. Objects and actions

Arjuna supports a computation model in which application programs manipulate persistent (long-lived) objects under the control of atomic actions (atomic transactions). Distributed execution is achieved by invoking operations on objects which may be remote from the invoker using remote procedure calls (RPCs). All operation invocations may be controlled by the use of atomic actions which have the well known ACID properties (ACID: Atomicity, Consistency, Isolation, Durability). Atomic actions 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 atomic action aborts, no updates get recorded. Typical failures causing a computation to be aborted include node crashes and continued loss of messages. It is assumed that, in the absence of failures and concurrency, the invocation of an operation produces consistent (class specific) state changes to the object. Atomic actions then ensure that only consistent state changes to objects take place despite concurrent access and any failures.

The object and atomic action model provides a natural framework for designing fault-tolerant systems with persistent objects. A persistent object not in use is assumed to be held in a passive state with its state residing in an object store (a stable object repository) and activated on demand (i.e., when an invocation is made) by loading its state and methods from the object store to the volatile store, and associating an object server process for receiving RPC invocations.

2.3. System Architecture

We assume that for each persistent object there is one node (say α) which, if functioning, is capable of running an object server which can execute the operations of that object (in effect, this would require that α has access to the executable binary of the code for the object's methods as well as the persistent state of the object stored on some, possibly remote object store). Before a client can invoke an operation on an object, it must first be connected or bound to the object server managing that object. It will be the responsibility of a node, such as α, to provide such a connection service to clients. If the object in question is in a passive state, then α is also responsible for activating the object before connecting the requesting client to the server. In order to get a connection, an application program must be able to obtain location information about the object (such as the name of the node where the server for the object can be made available). We assume that each persistent object possesses a unique, system given identifier (UID). The typical structure of an application level program is shown below:

<create bindings>; <invoke operations from within atomic actions>; <break bindings>

In our model, bindings are not stable (do not survive the crash of the client or server). Bindings to servers are created as objects enter the scope in the application program. If some bound server subsequently crashes then the corresponding binding is broken and not repaired within the lifetime of the program (even if the server node is functioning again); all the surviving bindings are explicitly broken as objects go out of the scope of the application program. We now identify the main modules of *Arjuna* and the services they provide for supporting persistent objects, shown in figure 1.

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

(ii) *RPC module.* 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 group [5] and used novel (for the time) C++ stub-generation techniques to enhance distribution transparency [6].

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

(iv) *Naming and Binding module.* 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.

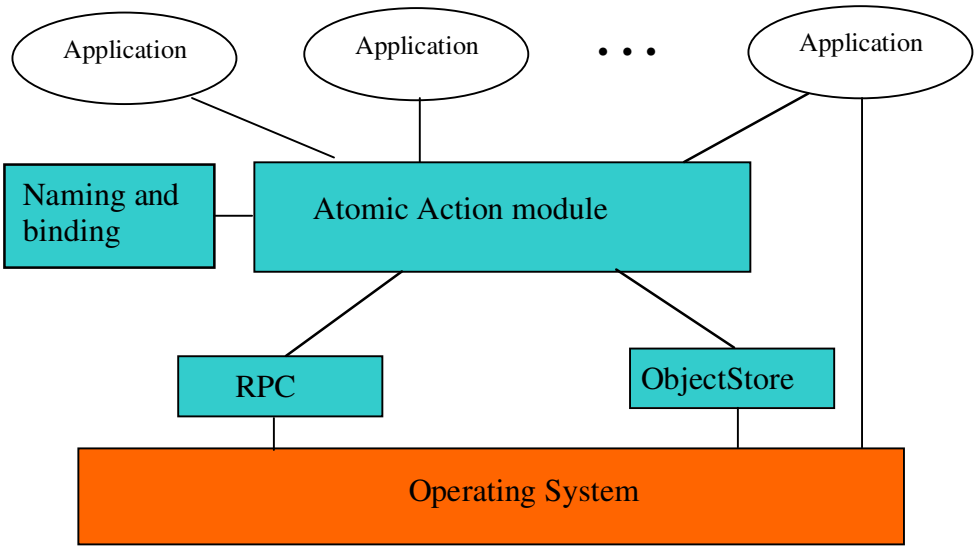

Fig. 1: Components of *Arjuna*

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. This system structure is highly modular: by encapsulating the properties of persistence, recoverability, shareability, serialisability and failure atomicity in an Atomic Action module and defining narrow, well-defined interfaces to the supporting environment, we achieved a significant degree of modularity as well as portability for *Arjuna*.

We will now use a simple program to illustrate how these modules interact. The program shown below (fig. 2) is accessing two existing persistent objects, `A`, an instance of class `O1` and `B`, an instance of class `O2`. In *Arjuna* a primitive operation `initiate(...)`, provided by the RPC module is used for binding to an object. A complementary operation, called `terminate(..)`, is available for breaking a binding. Clients and servers have communication identifiers for sending and receiving messages. The RPC module of each node has a connection manager process that is responsible for creating and terminating bindings to local servers. The *Arjuna* stub generation system for C++ generates the necessary client-server stub-codes for accessing remote objects via RPCs and also generates calls on `initiate` and `terminate` as an object comes and goes out of the scope of a computation [6].

```
{
 O1 objct1(Name-A);          /* bind to A */
 O2 objct2(Name-B);          /* bind to B */
 AtomicAction act;
 act.Begin();                      /* start of atomic action act */
 objct1.op(...);
 objct2.op(...);             /* invocations ....*/
 .......
 act.End();                  /* act commits */
 }                                 /* break bindings to A and B */
```

Fig. 2: Outline Action Example

To bind to A, a local instance of O1 called objct1 is created, passing to its constructor an instance of the *Arjuna* naming class called Name-A suitably initialised with information about A (e.g., its UID, the location of the server node, etc.); this enables the client side stub-constructor to initiate A, resulting in binding to the server for A. The Object Store module of *Arjuna* enables a server to load the latest (committed) state of the object from the object store of a node. The state is loaded, where necessary, as a side effect of locking the object.

Assume that the client program is executing at node N_1 and the server node for A is at N_2 (see Fig. 3). The client process at node N_1 executing the stub for object1 is responsible for invoking the initiate operation of the local RPC module in order to send a connection request to the connection manager at N_2. The connection manager locates the object sever for A who then returns its connection identifier to the client at N_1, thereby terminating the invocation of initiate at N_1. The storage and retrieval of object states from an object store is managed by a *state daemon*. The object server uses the state demon for retrieving the state of an object from the object store. For efficiency reasons, an object server can (and will) directly access the object store, bypassing the daemon if the server and the store are on the same node. However, if the object store is remote, then it must contact the state demon of the remote node managing the object store.

To manipulate objects under the control of an atomic action, the client creates a local instance of an action (act) and invokes its Begin operation. The End operation is responsible for committing the atomic action (using the two-phase commit protocol). When an object goes out of scope, it is destroyed by executing its destructor. As a part of this, the client-side destructor (e.g., the stub destructor for object1) breaks the binding with the object server at the remote node (using the operation terminate).

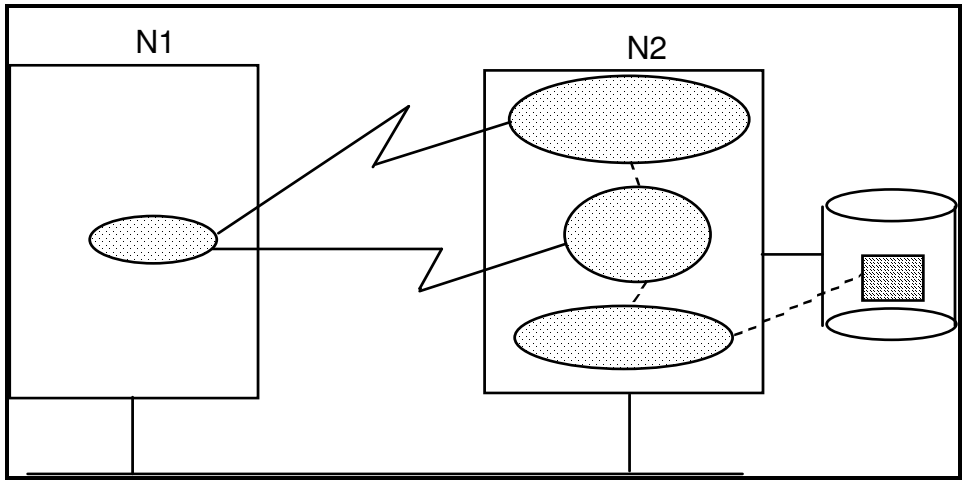

Fig. 3: Accessing an Object

Once the execution of an action begins, any failures preventing forward progress of the computation lead to the action being aborted, and any updates to objects undone. However, as establishing and breaking bindings can be performed outside of the control of any application level atomic actions, it is instructive to enquire how any clean-up is performed if client, server or network partition failures occur before (after) an application level action has started (finished). The simple case is the crash of a server node: this has the automatic effect of breaking the connection with all of its clients; if a client subsequently enters an atomic action and invokes an operation in the server, the invocation will return exceptionally and the action will be aborted; on the other hand, if the client is in the process of breaking the bindings then this has occurred already. More difficult is the case of a client crash.

Suppose the client crashes after binding to a server. Then explicit steps must be taken to remove any state information kept for the *orphaned* bindings; this requires that a server node must have a mechanism for breaking the binding if it suspects the crash of a client. This mechanism will also cope with a partition that prevents any communication between a client and a server. The *Arjuna* RPC level facilities for the detection and killing of orphans [5] is responsible for such a cleanup, ensuring at the same time that an orphaned server (a server with no bindings) is terminated.

2.3. 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 then 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 which make up the class hierarchy of Arjuna Atomic Action module are depicted in Fig. 4. To make use of atomic actions in an application, instances of the class, `AtomicAction` must be declared by the programmer in the application as illustrated in Fig. 2; 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 in Fig. 4. Most Arjuna classes are derived from the base class `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 the class `StateManager` are the principal users of the object store service. The class `LockManager` uses the facilities of `StateManager` and provides the concurrency control (two-phase locking in the current implementation) required for implementing the serialisability property of atomic actions. The implementation of atomic action facilities for recovery, persistence management and concurrency control is supported by a collection of object classes derived from the class `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. The `AtomicAction` class 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.

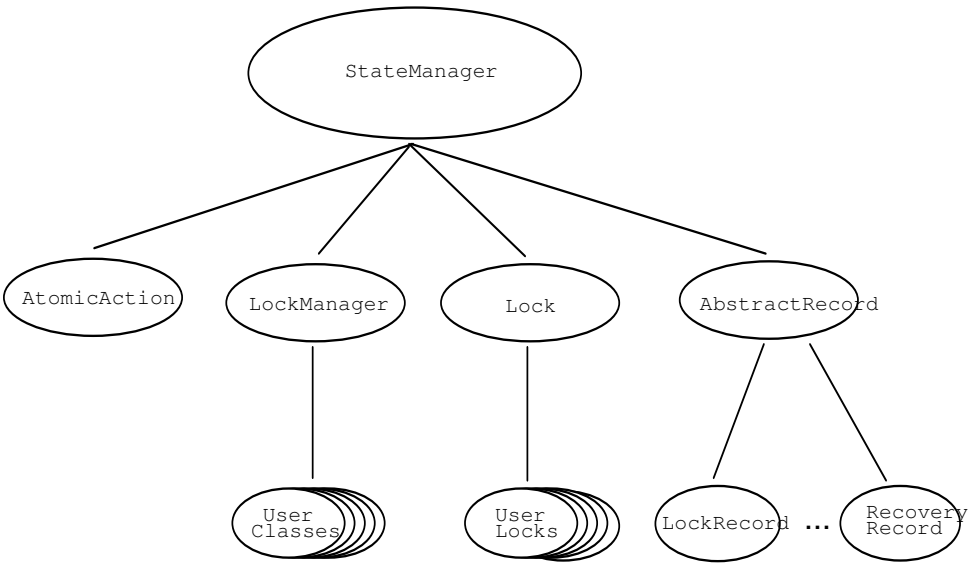

Fig. 4: The Arjuna Class Hierarchy

Consider a simple example. Assume that `O` is a user-defined persistent object. An application containing an atomic action `A` accesses this object by invoking an operation `op1` of `O` which involves state changes to `O`. The serialisability property requires that a write lock must be acquired on `O` before it is modified; thus the body of `op1` should contain a call to the appropriate operation of the concurrency controller (See Fig. 5):

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

Fig. 5: The use of Locks in Implementing Operations

The operation `setlock`, provided by the `LockManager` class, 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` (the Lock 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. Then the `abort` operation of `AtomicAction` will process the `RecordList` instance associated with `A` by invoking the `abort` operation on the various records. The implementation of this operation by the `LockRecord` class will release the `WRITE` lock while that of `RecoveryRecord` will restore the prior state of `O`.

The `AbstractRecord` based approach of managing object properties has proved to be extremely useful in *Arjuna*. Several uses are summarised here. `RecoveryRecord` supports state-based

recovery, since its abort operation is responsible for restoring the prior state of the object. However, its recovery capability can be altered by refining the abort operation to take some alternative course of action, such as executing a compensating function. This is the principal means of implementing type-specific recovery for user-defined objects in *Arjuna*. The class `LockRecord` is a good example of how recoverable locking is supported for a Lock object: the `abort` operation of `LockRecord` does not perform state restoration, but executes a `release_lock` operation. Note that locks are, not surprisingly, also treated as objects (instances of the class `Lock`), therefore they employ the same techniques for making themselves recoverable as any other object. Similarly, no special mechanism is required for aborting an action that has accessed remote objects. In this case, instances of `RpcCallRecord` are inserted into the `RecordList` instance of the atomic action as RPCs are made to the objects. Abortion of an action then involves invoking the `abort` operation of these `RpcCallRecord` instances which in turn send an "abort" RPC to the servers. In summary, what work a participant in the two-phase protocol does when instructed by the transaction coordinator, is typically not of interest to the coordinator. It may update a database, modify a file on disk, etc: it depends upon the type of transactional resource it is responsible for manipulating. Some transaction implementations place restrictions on the types of resources that can be used within the two-phase protocol; for example, in the X/Open Distributed Transaction Processing standard adopted by industry [7], they must support the XA protocol, which imposes restrictions on the underlying participant implementations, typically resulting in only databases being used.

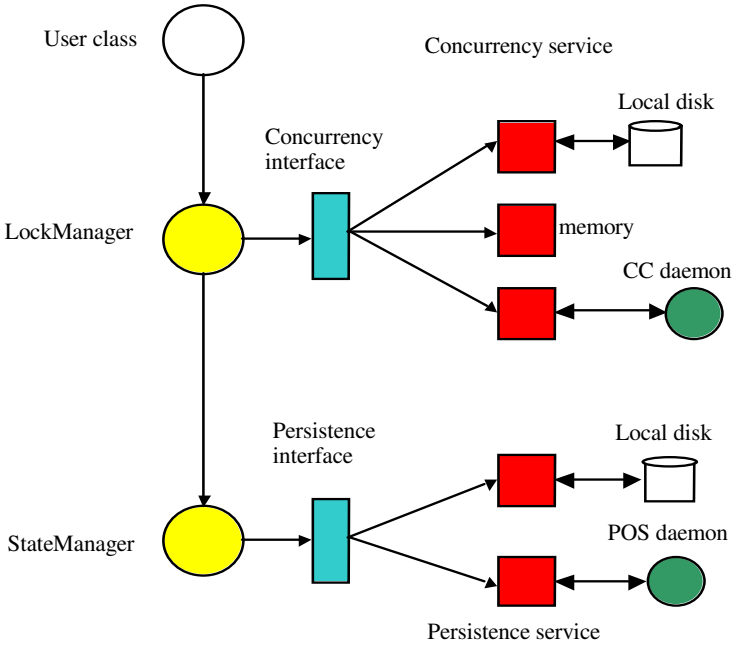

Fig. 6: Implementation binding

Fig. 6 shows how multiple implementations of concurrency and persistence can be at the disposal of a transactional user class. Internally, `LockManager` accesses the concurrency service through an interface, and `StateManager` does likewise with the persistence service. For each application object, the implementations are not chosen until run-time. Additional implementations can be provided without changing *Arjuna* or applications which use it. The *Arjuna* API (AIT) isolates programmers from the different implementations, allowing them to concentrate on the application development.

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. In addition, *Arjuna* was used in a number of industrial research projects [2]. A particularly demanding application of *Arjuna* has been the electronic student registration system in use since 1994 by the Newcastle University [8]. The electronic 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 when new students arrive).

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. During the 8 years that the system has been in use, there have been several network and machine failures and, with one exception, Arjuna has coped with them all, leaving users unaware that anything untoward has occurred.

The one notable exception occurred in the first year of deployment: in any distributed environment, it is not possible to determine in a finite period of time the difference between a machine/network failure and network/machine congestion, i.e., a machine that is extremely slow to respond may appear to its users to have failed. Without waiting forever (or until a failed machine recovers) it is necessary to employ heuristics to decide that after a certain period of time, a machine that has not responded has failed and to act accordingly. However, if the time period is not chose correctly and an incorrect decision is taken, the result could be the loss of data integrity. Up until the deployment of the student registration system, Arjuna has been used in small-scale, closely coupled environments where round-trip times for RPCs were measured in several milliseconds and machines were rarely congested. In the student registration environment, 20000 students were registered over 5 days using 10 server machines and 120 clients and often requiring access to a student record many times. We quickly found that our original assumptions about when to assume a silent machine had failed were wrong and adapted the system accordingly.

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. By then CORBA and Java middleware were attracting industry attention. The OMG object 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 required 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 very brief overview of middleware concepts, using CORBA as an example. Fig. 7 depicts the main elements of the CORBA middleware. It consists of an ‘object bus’, the object request broker (ORB) using which clients can interact with remote objects. A number of services (CORBAservices) are available for facilitating this task; these include services for naming, persistence, event notification, transactions (OTS) and so forth. JAVA/RMI is a broadly similar Java language specific middleware.

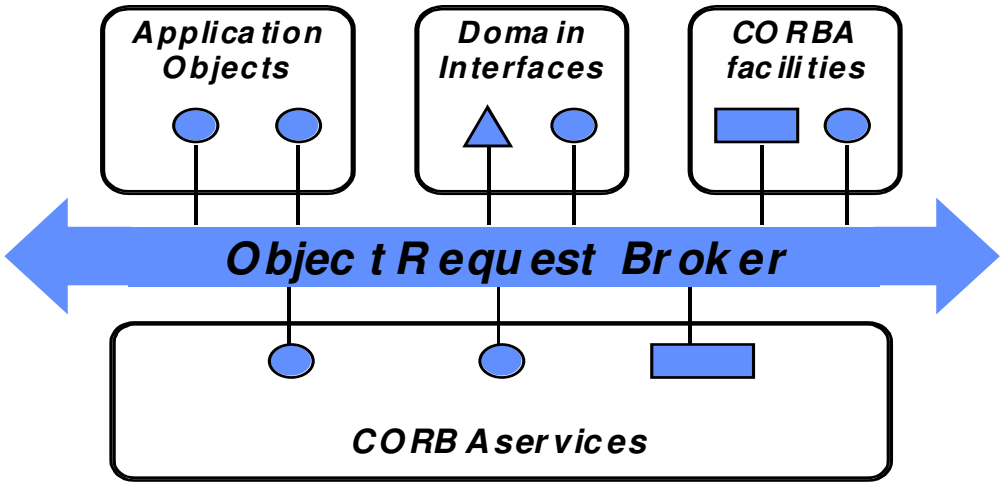

Fig. 7: CORBA middleware

Although this middleware provides type checked remote invocations and standard ways of using commonly required services - a major advance over the prior practice of writing ad hoc networking code - 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 + capability for using middleware services in a standard manner'. A component is hosted by a *container* (a server process), and normally, it is the container that uses the underlying middleware services on behalf of the application object. A component descriptor specifies, in a declarative manner the middleware 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. An Enterprise Java bean (EJB), part of the J2EE suite of specifications, is a good example of a (Java) component; there are a number of publicly available J2EE application servers capable of hosting EJBs. In the rest of this section we will first examine how OTS and JTS 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 the functionality provided by Arjuna with that of J2EE application servers.

3.2. Arjuna, the OTS and the JTS

In 1995, the industry standard for transaction processing systems changed from X/Open XA [7], which was very procedural-oriented, to the *OMG* Object Transaction Service [9]. This change was based on the experiences of all of the major transaction processing vendors, including IBM, HP and DEC. The OTS provides interfaces that allow multiple distributed objects to co-operate in a transaction such that all objects commit or abort their changes together. Transactions can optionally be nested to improve fault-isolation. However, the OTS does not require all objects to have transactional behaviour. Instead objects can choose not to support transactional operations at all, or to support it for some requests but not others. Importantly, the OTS is simply 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 co-operating services that implement the required functionality, including:

- *Persistence/Recovery Service*. Required to support the atomicity and durability properties.

- *Concurrency Control Service*. Required to support the serialisability/isolation properties.

The application programmer is responsible for using these services to ensure that transactional objects have the necessary ACID properties. To participate within an OTS transaction, a programmer must be concerned with:

- creating `Resource and SubtransactionAwareResource` objects for each object which will participate within the transaction/subtransaction; these are the two-phase aware entities. The OTS will invoke these objects during the prepare/commit/abort phase of the (sub)transaction, and the `Resources` must then perform all appropriate work.

- registering `Resource` and `SubtransactionAwareResource` objects at the correct time within the transaction, and ensuring that the object is only registered once within a given transaction. As part of registration a `Resource` will receive a reference to a `RecoveryCoordinator` which must be made persistent so that recovery can occur in the event of a failure.

- ensuring that, in the case of nested transactions, any propagation of resources such as locks to parent transactions are correctly performed. Propagation of `SubtransactionAwareResource` objects to parents must also be managed.

- in the event of failures, the programmer or system administrator is responsible for driving the crash recovery for each `Resource` which was participating within the transaction.

The OTS does not provide any `Resource` implementations. These must be provided by the application programmer or the OTS implementer. As such, a pure OTS implementation would actually provide much less functionality than that available in *Arjuna*. The OTS does not define a complete toolkit for the construction of transactional applications as *Arjuna* did: 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 already a good match with the required OTS functionality. Obviously the APIs and implementation interfaces that are exposed by the OTS did not match those available in *Arjuna*. However, it was relatively straightforward to provide these same abstractions on top of the equivalent *Arjuna* APIs.

At this time, the sources of funding for the *Arjuna* group began to change from purely research driven bodies to industrial driven consortia and members of these consortia expressed interest in standards-related development. 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 their requirements.

Most effort was directed at fully integrating *Arjuna* (*OTSArjuna*) within the CORBA framework, e.g., how to do distributed invocations and ensure that the transaction context is passed as mandated by both the OTS and CORBA specifications. The interfaces we had defined between the Naming and Binding and RPC modules were sufficiently powerful that only minimal modifications had to be made (typically because of deficiencies in the immature CORBA implementations we used).

The architecture of *OTSArjuna* is shown in fig. 8. The OTS protocol engine, State Management and Concurrent Control are essentially exactly as they appeared in the original Arjuna implementation. The external interfaces defined by the OTS specification are layered over them to give a CORBA look-and-feel, but the implementations are essentially the same. Importantly, to maintain the original *Arjuna*

pluggable abstraction, any OTS-specific modifications that were made to the system (such as API updates) occurred in self-contained modules. AIT was provided to programmers via the OTSArjuna API. This API automates much of the activities concerned with participating in an OTS transaction: it interacts with the concurrency control and persistence services, and automatically registers appropriate resources for transactional objects.

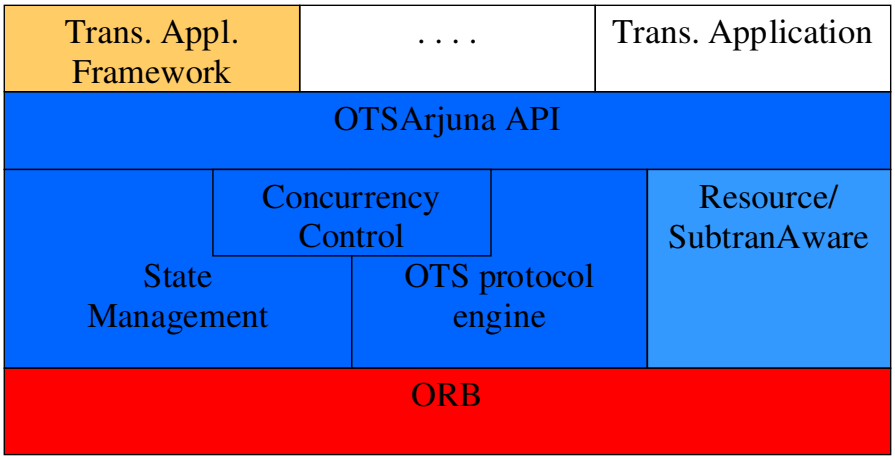

Fig. 8: OTSArjuna architecture

Remote (CORBA-based) participants, XA compliant participants etc. were all transparently controlled by the core of *OTSArjuna* via appropriate implementations of the AbstractRecord class. As such, the protocol engine within *OTSArjuna* could not tell that it was now running within a CORBA environment. Using the interfaces defined in the original *Arjuna* system, we were able to convert most of Arjuna to be CORBA compliant. However, there was one notable exception where the abstractions we had originally in place simply did not work within the OTS: *failure recovery*.

To guarantee ACID properties in the event of failures (e.g., machine crash) and eventual recovery, a failure recovery subsystem is required. This ensures that any transactions that were in progress when the failure occurred are completed, either by being committed or rolled back. In order to do this, it may be necessary to recreate any resources (e.g., remote objects) that were participants within the transaction and may have also failed. In essence, the failure recovery system will recreate the distribution tree (resources and their network connections) that were present prior to the failure.

In order to achieve this, failure recovery must have intimate knowledge about the resources that are being recovered (e.g., do they use a file system for persistence or a database?) and the RPC mechanism (e.g., what is the host and port the object resides on?) The *Arjuna* failure recovery implementation was therefore closely tied to the its RPC mechanism. We were therefore unable to take this component from the original *Arjuna* system or to reuse the interfaces it provided. As such, we were forced to re-implement failure recovery and in doing so we tied it to the OTS defined model. This resulted in tying *OTSArjuna* to CORBA: although most of the core of *Arjuna* could still be used outside of CORBA, without failure recovery it would have limited use.

By 1996 Java started attracting serious attention from industry and many existing OMG standards made their way into the evolving J2EE specification. Critical amongst them was the OTS, which J2EE renamed as the Java Transaction Service (JTS). Interestingly, because *Arjuna* (and hence *OTSArjuna*) had been developed using C++, it was a relatively straightforward task to convert the system to Java.

This was done and the resulting system, *JTSArjuna*, became the worlds first pure Java transaction system.

3.3 Arjuna and J2EE

A simplified J2EE application server architecture is shown in fig. 9. As can be seen, it has very similar components (modules) to *Arjuna*, but utilises industry standard technologies such as CORBA for the remote object invocation mechanism and naming and binding. Although the interfaces to the various components within an application server are different to those chosen by the designers of *Arjuna*, the fact that these components exist in an identifiable (and typically replaceable) manner is testimony to the fact that the original architecture and design goal of *Arjuna* were probably right.

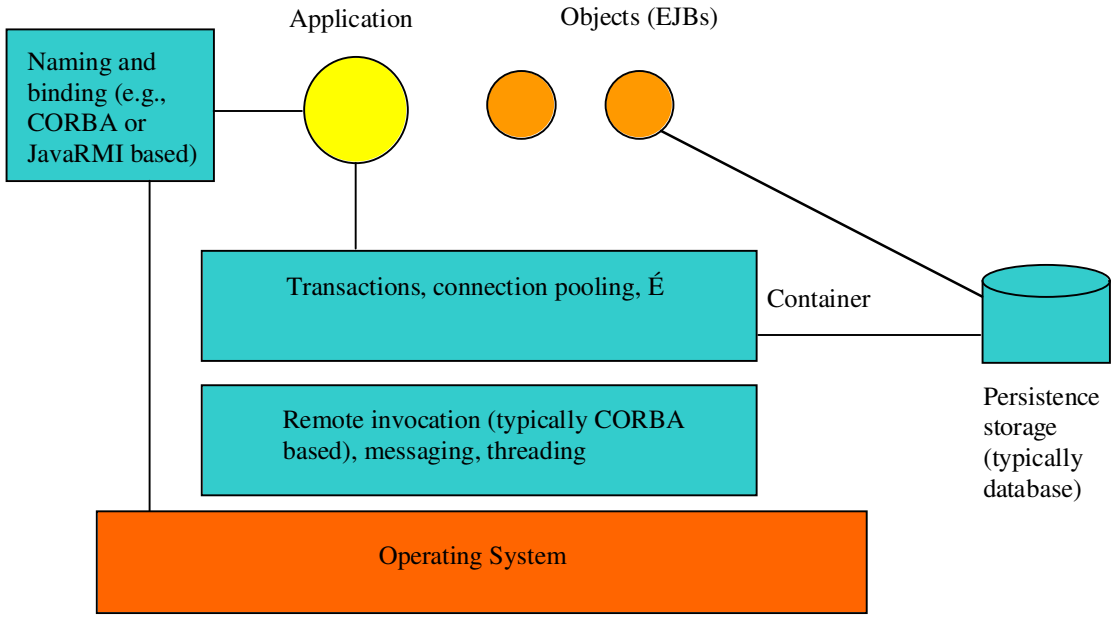

Fig. 9: Simplified application-server architecture.

We now compare several aspects of EJB/J2EE with *Arjuna* (JTSArjuna more precisely). 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 the *Arjuna* API, AIT, and its use is not generally recommended [11]. In any case, compared to *Arjuna*, transactions in EJBs come with several restrictions that we describe below (although the discussion is with respect to JTA, the observations apply equally well to the declarative use).

(i) *Participant restriction*: both the original *Arjuna* system and the OTS standard on which *JTSArjuna* is based, allow arbitrary participant implementations to be enlisted in a transaction. The *Arjuna* AbstractRecord interface and the OTS equivalent (CosTransactions::Resource) do not imply or mandate a specific implementation. As we described in the previous section, this allows recovery, concurrency control etc. to be transparently enlisted with a transaction manager. However, the JTA interface, restricts participants to being X/Open XA-aware. In essence, this effectively mandates that applications must use two-phase aware databases for persistence.

(ii) *No nested transactions*: if an object's methods are required to use transactions then, in an environment which supported nested transactions, the object implementer can use transactions without concern about how those methods will be invoked: if the invoker uses transactions, then the object's methods will be nested within them, otherwise they will simply be top-level. In addition, nesting provides a level of failure containment, since the failure of a nested transaction does not require the enclosing (parent) transaction to roll back. The JTA does not support nested transactions because the X/Open XA standard does not. Based on our experiences, in our opinion, for a component-based, distributed architecture this is a significant shortcoming.

(iii) *Poor concurrency control*: In the section 2.3 we described how in Arjuna, transactional locking has been introduced which interacts correctly with the associated persistence and recovery mechanisms. Unfortunately, no satisfactory way of using read and write locks are available in EJBs. Because the JTA mandates that all participants must be XA-aware, this essentially 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 table within the database 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 within EJBs, the programmer is supposed to make use of the Java language `synchronized` construct, which obtains an exclusive lock on a method or data structure. 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 within EJBs. Because locks within AIT 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, some scenarios were discussed towards the end of section 2.2. 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).

Web Services, messaging, ubiquitous computing and transactions

Over the past decade, distributed computing has become more prevalent. The advent of Java and ubiquitous computing devices such as palm-top devices, has helped to move distributed computing from research and high-end commercial systems into (almost) everyday use. There has also been a paradigm shift from closely-coupled, synchronous environments to large-scale, loosely coupled and asynchronous systems.

The one constant amongst traditional and new distributed environments is their requirement for transactions. Whether applications run on closely-coupled local area networks and interact through RPC or they run on the loosely-coupled Internet and use messaging, 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. In order to understand how and where Arjuna has been utilised, we shall first briefly describe these environments.

The Java Message Service

The Java Message Service (JMS) defines a set of interfaces and the associated semantics that facilitate communication between Java applications and messaging implementations. By leveraging the JMS API applications can create, send, receive, and read messages. Thus, JMS enables communication that is loosely coupled, asynchronous, and reliable.

The combination of transactions and JMS offers developers a platform for the creation of robust enterprise messaging applications. Developers can interact with the messaging provider transactionally to ensure that multiple messages are delivered or received as a single group; if a single operation fails, the entire set of message receipts or deliveries will be rolled back.

A JMS client can use *local* transactions to group message sends and receives using the `Session` interface. By invoking the `commit` method, the client indicates that all produced messages are to be send and all consumed messages are to be acknowledged. Similarly, an invocation of `rollback` indicates that all produced messages are to be destroyed and all consumed messages are to be recovered and redelivered by the JMS provider. Multiple message sends and receives can be grouped into a single local transaction, but the user cannot *combine* sends and received within the same transaction: a message send cannot take place until the associated transaction has committed because the message is not actually sent until the transaction is committed. Therefore, the transaction cannot contain any receives that depend upon a message sent during the same transaction.

An optional part of the JMS specification allows a transactional session to be enrolled with a distributed transaction managed by the JTS. Such a session must support the XAResource interface. XA sessions work similarly to local transacted sessions since the JMS operations performed in the scope of an XA transaction are conditional on the committal of the transaction. The main difference is the party acting as the transactional coordinator: in the local case, the JMS provider manages the execution of the transaction; using XA transactions, the outcome of the transaction is dependant on the decision of the external JTA-compliant transaction manager into which the JMS XAResource is enlisted.

The Business Transactions Protocol

As Web services have evolved as a means to integrate processes and applications at an inter-enterprise level, traditional transaction semantics and protocols have proved to be inappropriate. Web Service transactions differ from traditional transactions in that they will execute over long periods, will require commitments to the transaction to be "negotiated" at runtime and isolation levels will have to be relaxed. Furthermore, business transactions are not only long lived but will in may instances incorporate multiple parties, requiring flexibility in determining transaction outcome, meaning relaxed atomicity while negotiating commitment to the transaction. These types of transactions, *Business Transactions*, require an extended transaction model that can support complex Web service interactions that need transactional semantics and guarantees, at the inter-enterprise level.

The *Business Transaction Protocol* (*BTP*) has been designed to allow the coordination of business transactions that span multiple participants ensuring the consistent result of a transaction without concern for whether the transaction spans disparate applications, developed with disparate technologies and potentially deployed by different organisations. In such circumstances, participants may not represent resources controlled by a single party and therefore must maintain some autonomy: able to manage their own resources while maintaining adherence to any commitments they have made, with respect to the transaction. In this way, the participants in a business transaction may use recorded

before- or after-images, or compensation operations to provide the "roll-forward, roll-back" capacity, which enables their coordination in respect to the overall outcome of the business transaction.

Open-top coordination

In a traditional transaction system, the application has very few verbs with which to control transactions. Typically these are "begin", "commit" and "rollback". When an application asks for a transaction to commit, the coordinator will execute the entire two-phase protocol before returning an outcome (what BTP terms a *closed-top commit protocol*). The elapse time between the execution of the first phase and the second phase is typically milliseconds to seconds.

However, the two-phase algorithm does not impose any restrictions on the time between executing the first and second phases. Clearly the longer the period between first and second phases, the greater the chance for failures to occur and the longer resources remain locked. BTP allows the time between the two phases to be set by the application by expanding the range of verbs available to include explicit control over both phases, i.e., "prepare", "confirm" and "cancel"; what BTP terms an *open-top commit protocol*. The application has complete control over when transactions prepare, and using use whatever business logic is required later determine which transactions to confirm or cancel.

Atoms and cohesions

In order to address the requirements imposed by running business- transactions, BTP introduced two types of *extended transactions*, both using the open-top completion protocol:

- *Atom*: an atom is the typical way in which "transactional" work performed on Web services is scoped. The outcome of an atom is guaranteed to be atomic, such that, all enlisted participants (acting on behalf of their associated Web services) will see the same outcome, which will either be to accept (confirm) the work or reject (cancel) it.

- *Cohesion*: this type of transaction was introduced in order to relax atomicity and allow for the selection of work to be confirmed or cancelled based on higher level business rules. Atoms are the typical participants within a cohesion but, unlike an atom, a cohesion may give different outcomes to its participants such that some of them may confirm whilst the remainder cancel. In essence, the two-phase protocol for a cohesion is parameterised to allow a user to specify precisely which participants to prepare and which to cancel. The strategy underpinning cohesions is that they better model long-running business activities, where services enrol in atoms that represent specific units of work and as the business activity progresses, it may encounter conditions that allow it to cancel or prepare these units, with the caveat that it may be many hours or days before the cohesion arrives at its *confirm-set*: the set of participants that it requires to confirm in order for it to successfully terminate the business activity. Once the confirm-set has been determined, the cohesion collapses down to being an atom: all members of the confirm-set will see the same outcome.

ArjunaCore

As part of the Hewlett-Packard NetAction product suite, there was a requirement for JMS and BTP implementations. As mentioned previously, HP had *JTSArjuna* and *OTSArjuna* products to cover the J2EE and CORBA markets respectively and as we have seen, these are both transaction systems which use a two-phase commit protocol. When comparing the functionality provided by *JTSArjuna* with that required by the JMS, for example, it was clear that with the exception of the J2EE specific components, there was much overlap. In fact, it was possible to categorise all of the product requirements as follows:

- The use a two-phase completion 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 participant implementations are opaque to the two-phase transaction engine.

At the heart of every transaction processing system is a transaction manager. It is the transaction manager that is responsible for ensuring the atomicity and durability properties of the transactions under its control. The isolation and consistency are provided by transactional resources that participate in the transaction on behalf of applications and services. The coordinator must maintain a transaction log in case of failures and a recovery system to use this log to complete transactions that were in flight and caught by any failures (e.g., a machine or process crash). It is important to realize that this functionality if required by *all* transaction systems, whether or not they support distributed transactions.

The same core protocol engine that had been within the original *Arjuna* system and was now within *JTSArjuna*, could be used within HP's BTP (HP Web Services Transactions) and JMS (HP Messaging Service) implementations. The interfaces described in Section **Error! Reference source not found.** abstract away from the participant implementations and how transactional distributed invocations occur. By providing different implementations for, say, AbstractRecords, it is possible to drive BTP participants or JMS participants through a two-phase commit protocol using the *exact* same transaction engine.

In effect, the work that was performed to transform the original *Arjuna* system into *JTSArjuna* was generalised for other environments. The first step was to create a fully-functional transaction engine that had no dependencies on CORBA (including failure recovery): *ArjunaCore*. *ArjunaCore* 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 local transactions (the transaction context) to be transmitted in a manner suitable for the environment in which it is running, e.g., CORBA IIOP or SOAP/XML.

The main obstacle to the design of *ArjunaCore* was the failure recovery sub-system. As described earlier, 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 local environment. Therefore, failure recovery was re-architected in such a way that recovery occurred through specific implementations of the RecoveryModule interface. Each RecoveryModule was 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 RecoveryModule implementations are relatively simple. Other products in which *ArjunaCore* is embedded, e.g., BTP, provide suitable implementations to do distributed recovery where necessary.

Despite the necessary resign or the failure recovery subsystem, the remainder of *ArjunaCore* is the same as existed in the original *Arjuna* system. Through the use of the original modularisation and flexibility goals, the interfaces have proven sufficient to allow the system to be embedded as the core transaction engine within a diverse range of products.

Timeline

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.

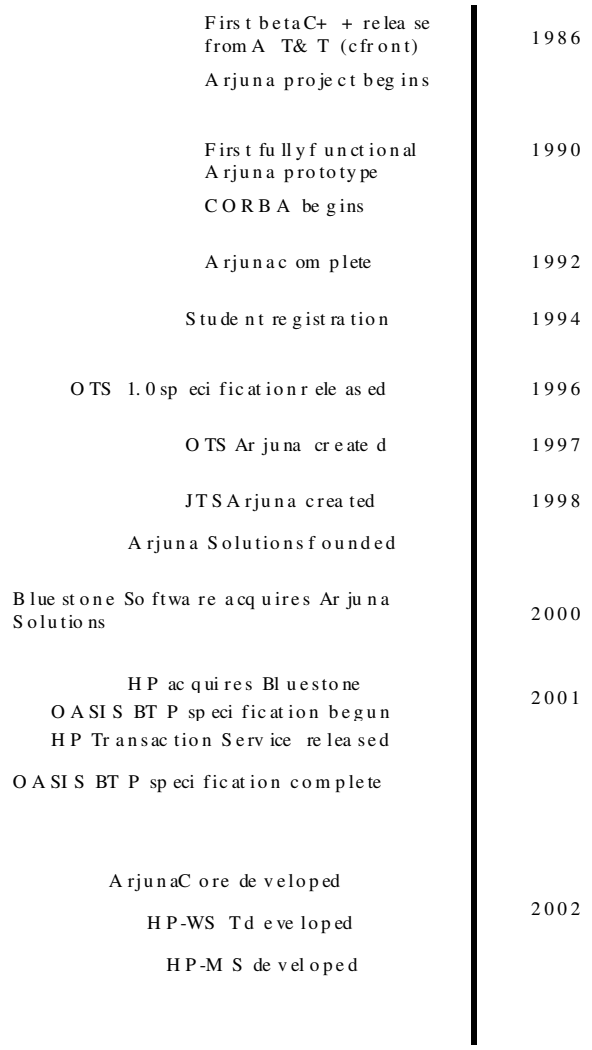

Fig 7. The evolution of Arjuna.

4. Concluding Remarks

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 reliability and robustness are typically more rigorous than you would expect from academia and this was the probably the hardest aspect for us to tackle.

- Working with various standards bodies (OTS, JTS and BTP 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 about right 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.

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] Panzieri, F. 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] Parrington, G. D., "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", Chapt. 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] Business Transaction Protocol specification, www.oasis.org