The evolution of a transaction processing system

Submission for High Performance Transaction Systems Workshop, September 2005

Mark Little[†], Santosh Shrivastava[‡]
[†]Chief Architect, Arjuna Technologies Ltd
[‡]Professor, University of Newcastle upon Tyne

Introduction

The Arjuna Transaction Service (ATS) began life in the mid 1980s as an academic project (called Arjuna) to examine the use of object-oriented techniques in the development of fault-tolerant systems [1]. It provides a set of tools for the construction of fault-tolerant distributed applications using (nested) transactions controlling operations on persistent objects. A version of the system written in C++ to run on networked UNIX systems was operational in the early nineties. The arrival of the Web and industrial acceptance of CORBA and Java technologies for distributed object computing during this period encouraged the productization of Arjuna. Through a series of acquisitions Arjuna became part of HP's middleware product lines and was used in creating customised transactional services for new application areas, such as Web Services [2] and mobile computing. In 2002, we span out of HP to continue to evolve ATS, where it supports traditional ACID transactions and "relaxed" transaction models (e.g., compensation transactions). In this paper we shall give an overview its evolution, concentrating particularly on those changes concerned with improved performance and availability.

An overview of Arjuna

The original aim of Arjuna was to provide a system for constructing fault-tolerant distributed applications using the following properties:

1) *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.
2) *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.
3) *Extensibility:* These mechanisms should also be flexible, permitting application specific enhancements, such as type-specific concurrency control and recovery, 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. The remaining two goals were met primarily through

the provision of a class library for incorporating the properties of fault-tolerance and distribution.

System architecture

The original architecture of Arjuna was similar to traditional transaction processing monitors, containing the following main modules:

- *Transactional API*: provides applications with an API for starting, committing and rolling back transactions.
- *Transaction coordinator*: provides the reliable transaction coordinator, which supports the traditional two-phase commit protocol.
- *Communication services*: provides the support for transactional invocations on remove services. Because Arjuna predates CORBA, COM, J2EE etc. this module originally included our own RPC implementation. However, as Arjuna evolved to support different environments, this was replaced with CORBA, Web Services, or whatever interaction mechanism was required.
- *Durable storage services*: provides a stable storage repository for persistent objects.

Unlike some transaction systems it was decided that Arjuna would not be XA specific: any two-phase aware resource should be capable of participating in a transaction. The key to this was to make the transaction coordinator work in terms of an abstract participant encapsulated in an interface called AbstractRecord. Later transaction specifications such as the OMG's Object Transaction Service (OTS), took a similar approach. In this way, the coordinator does not need to understand the internal implementation semantics of the individual participants, only that they will obey the two-phase commit protocol. Examples of specific participant implementations that have evolved include those for concurrency control, recovery and XA.

Evolution

In this section we shall examine the changes to Arjuna that occurred once the basic transaction architecture had been put in place.

Coordinator high availability

Although transactions guarantee consistency in the presence of failures, they do not provide a means of guaranteeing forward progress: if a transaction fails because a machine has crashed, simply retrying that transaction again may find the machine is still unavailable and hence the new transaction will also fail. Therefore, the first major change to Arjuna occurred when a high-availability option was added, allowing objects to be replicated on an arbitrary number of machines [3]. Importantly, both participants and the transaction coordinator could be replicated independently.

There are basically two classes of replication protocols:

1) *Active replication*: more than one copy of an object is activated on distinct nodes and all copies perform processing. Active replication is often the preferred choice

for supporting high availability of services where masking of replica failures with minimum time penalty is considered highly desirable.

2) *Passive replication*: only a single copy (the primary) is activated; the primary regularly checkpoints its state to the backups. One of the advantages of this form of replication is that it does not require deterministic replicas; however, its performance in the presence of primary failures can be poorer than active, because the time taken to switch over to a secondary is non-negligible.

Rather than support only one of these categories, which would either restrict the types of objects which could be replicated, or affect the performance of all applications, we believed it was important to support both protocol types. Similar to [4], when the coordinator service is actively replicated, individual transactions can commit despite N failures; a weighted voting protocol was used to tolerate network partitions.

In 1994, this combination of transactions and replication was put to the test when it was used in the electronic student registration system used by Newcastle University [5]. The registration system has a very high availability and consistency requirement. At that time, no other software based solution existed that could fulfil those requirements. During the many years the system was used, there were several network and machine failures and Arjuna's replication and transaction combination coped with them all.

Standards (CORBA and Java)

In 1995 the industry standard for distributed transactions looked like it was going to change from being predominately based on X/Open XA to the OMG's OTS. Examining Arjuna it was clear that there was a good match with the OTS and we received industry encouragement to pursue a mapping and subsequently form a company to market the implementation. Although the interfaces exposed by the OTS were different, it was relatively straightforward to provide these same abstractions. Most effort was directed at fully integrating Arjuna within CORBA, e.g., how to do distributed invocations and ensure that the transaction context is passed as mandated by the specification.

By 1996 Java started attracting serious attention from industry and many existing OMG standards made their way into the J2EE architecture. Critical amongst them was the OTS. Fortunately there were enough similarities between C++ and Java to make the task of converting the entire Arjuna system to Java relatively straightforward: Arjuna became the first 100% pure Java transaction system [6].

Coordinator performance

One of the limiting factors in the performance of any transaction system is the time it takes to write (and sync) the coordinator's log. This is necessary to allow recovery in the presence of failures of the coordinator and/or participants. As a result, a number of optimisations to reduce the need to write the log are typically implemented in TPMs, including *presumed abort*, *one-phase commit* and *read-only participants*. Although these were supported by Arjuna from the start, we made several other modifications to the original architecture based on implementation and usage experience.

Because of the original modular approach to the architecture, the ATS coordinator component is lightweight and has a small footprint. As such, it can be embedded in hand-

held devices as well as mainframe systems. Although the coordinator can run as a separate transaction manager process, the default configuration is to co-locate coordinator instances in the same process as the clients/services that use them, thus reducing the network latency and reducing independent failure modes. Capabilities such as durability are incorporated into the transaction coordinator implementation on demand. Therefore Arjuna assigns log structures only when strictly required, which contrasts with some other transaction systems, where logs are created on disk as soon as a transaction is created.

Furthermore, some applications have states with lifetimes that do not exceed the application program that creates them and yet these applications want transaction-like semantics for those states, i.e., the ability to commit or rollback changes held purely in memory. Therefore although these *recoverable* states require a coordinator to achieve atomicity, the coordinator does not have a durable log to write; it has a purely in-memory log for the purposes of consensus only. Being able to differentiate recoverable states from *durable-recoverable* (traditional transaction participants) states enables the coordinator to further optimise the disk accesses. This functionality is similar to that in some recent transaction service implementations [7]. It is also possible that recoverable and durable-recoverable states may be manipulated within the same transaction. Fortunately using the same logic as the classic one-phase commit optimisation, there is no need for the coordinator to produce a log entry unless manipulating at least two distinct durable-recoverable entities.

Supporting multiple protocols

As part of Hewlett-Packard and subsequently as our spin-off, there was a requirement for both J2EE transactional messaging and Web Services transactions implementations (including WS-AtomicTransaction/WS-BusinessActivity [8][9] and OASIS WS-CAF [10]). Although messaging specifications require traditional XA, all of the Web Services transactions specifications require support for compensation based transactions. It was possible to categorise the various product requirements as follows:

- Irrespective of traditional XA or compensation based models, it is possible to map the transaction protocol messages on to an implementation of a two-phase completion protocol (in some cases the implementation of one of the phases may do nothing).
- Transaction context information is carried in a manner suitable to the environment, e.g., XML and SOAP, or IIOP for CORBA.
- Transaction participant implementations are opaque to the transaction engine.

Careful examination showed that it was possible to use the same core protocol engine that was within ATS to support all of the transaction products. The various ATS interfaces (e.g., AbstractRecord) isolate the coordinator from the specifics of participant implementations and how transactional distributed invocations occur.

Therefore, the core transaction engine was separated from the original ATS product, which at that time was tied to CORBA and J2EE for some failure recovery aspects. This component became ArjunaCore, a fully-functional transaction engine that has *no*

dependencies on any distribution infrastructure. It is concerned solely with the use of local transactions, i.e., transactions that run on a single machine. If distributed transactions are required, the system provides hooks to enable information 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.

Conclusions

In achieving the transition of the Arjuna distributed transaction processing software from research to a range of different products, we have learned a number of lessons, some of which should be relevant to others involved in or embarking on a similar process. It has not been as straightforward as this extended abstract might imply and there have been several twists and turns in the road.

References

[1] 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.

[2] "A Framework for Implementing Business Transactions on the Web", Hewlett-Packard initial submission to BTP, March 2001, http://www.oasis-open.org/committees/business-transactions/

[3] M. C. Little and S.K. Shrivastava, "Replicated K-Resilient Objects in Arjuna", Proceedings of IEEE Workshop on the Management of Replicated Data, pp. 53-58, Houston, Texas, November 1990.

[4] J. Gray and L. Lamport, "Consensus on Transaction Commit", Microsoft Research Technical Report, MSR-TR-2003-96, April 2004.

[5] M. C. Little, S. M. Wheater, et al, "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.

[6] M. C. Little and S.K. Shrivastava, *"Distributed Transaction in Java," Contribution to* High Performance Transaction Systems (HPTS) workshop, Monterey, Sept. 1997.

[7] "Indigo/XML Enterprise Services – Transactions Overview", February 2005, http://lab.msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/N_System_Transactions.asp?frame=true

[8] The Web Services Atomic Transaction specification, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnglobspec/html/wsat.asp

[9] The Web Services Business Activity specification, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnglobspec/html/wsba.asp

[10] The Web Services Composite Application Framework Technical Committee, http://www.oasis-open.org/committees/documents.php?wg_abbrev=ws-caf