© Hewlett-Packard Strictly Company Confidential

How HPTS supports end-to-end transactional applications (and has done so since 1997)

Dr. Mark C. Little

Distinguished Engineer, Transactions Architect

Hewlett-Packard Arjuna Labs, Newcastle upon Tyne, England.

1. Introduction

This document represents a response to a recent enquiry concerning the perceived inability for modern transaction processing systems, and HPTS in particular, to support end-to-end transactional guarantees [MKP01]. Before addressing the question of whether or not any specific transaction system can be used to provide end-to-end transactional guarantees, it is important to realise the following: end-to-end transactionality is not some holy grail of transactioning that people have been searching for in myths and legends; it is a solution to one specific problem area, and is in no way a global panacea to all transaction issues. The ability, or lack thereof, to provide end-to-end transaction integrity guarantees does not in and of itself prevent a specific transaction system provider from tackling many other equally important issues in today's evolving world of e-commerce and mobile applications.

Let us first consider what such end-to-end guarantees are. Atomic transactions (*transactions*) are used in application programs to control the manipulation of persistent (long-lived) objects. Transactions have the following ACID properties [OMG95]:

- *Atomic*: if interrupted by failure, all effects are undone (rolled back).
- *Consistent*: the effects of a transaction preserve invariant properties.
- *Isolated*: a transaction's intermediate states are not visible to other transactions. Transactions appear to execute serially, even if they are performed concurrently.
- *Durable*: the effects of a completed transaction are persistent; they are never lost (except in a catastrophic failure).

A transaction can be terminated in two ways: committed or aborted (rolled back). When a transaction is committed, all changes made within it are made durable (forced on to stable storage, e.g., disk). When a transaction is aborted, all of the changes are undone. Atomic transactions can also be nested; the effects of a nested transaction are provisional upon the commit/abort of the outermost (*top-level*) atomic transaction.

1.1 Commit protocol

A two-phase commit protocol is required to guarantee that all of the transaction participants either commit or abort any changes made. Figure 1 illustrates the main aspects of the commit protocol: during phase 1, the transaction coordinator, C, attempts to communicate with all of the transaction participants, A and B, to determine whether they will commit or abort. An abort reply from any participant acts as a veto, causing the entire transaction to abort. Based upon these (lack of) responses, the coordinator arrives at the decision of whether to commit or abort the transaction. If the transaction will commit, the coordinator records this decision on stable storage, and the protocol enters phase 2, where the coordinator forces the participants to carry out the decision. The coordinator also informs the participants if the transaction aborts.

When each participant receives the coordinator's phase 1 message, they record sufficient information on stable storage to either commit or abort changes made during the transaction. After returning the phase 1 response, each participant which returned a commit response *must* remain blocked until it has received the coordinator's phase 2 message. Until they receive this message, these resources are unavailable for use by other transactions. If the coordinator fails before delivery of this message, these resources remain blocked. However, if crashed machines eventually recover, crash recovery mechanisms can be employed to unblock the protocol and terminate the transaction.

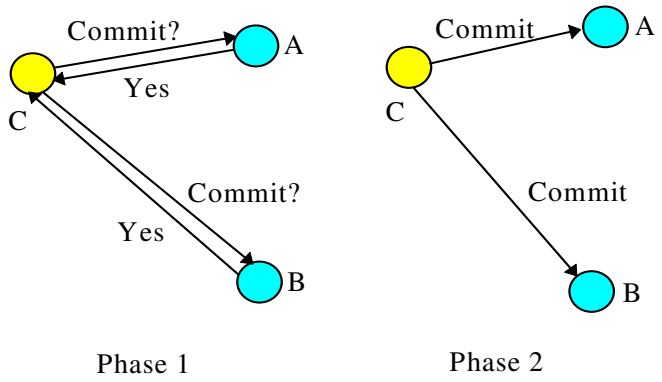

Figure 1: Two-phase commit protocol.

The role that a participant plays will depend upon the application in which it occurs. For example, a J2EE JTA XAResource is such a participant that typically controls the fate of work performed on a database (e.g., Oracle) within the scope of a specific transaction.

Note, the two-phase commit protocol that most transaction systems use is not client-server based. It simply talks about a coordinator and participants, and makes no assumption about where they are located. Different **implementations** of the protocol may impose certain restrictions about locality, but these are purely implementation choices.

1.2 End-to-end transactionality

In this section we shall use the Web as an example area where *end-to-end transactionality integrity* is required. However, there are other areas (e.g., mobile) that are equally valid. Transposing the issues mentioned here to these other areas should be relatively straightforward.

Atomic transactions, with their "all-or-nothing" property, are a well-known technique for guaranteeing application consistency in the presence of failures. Although Web applications exist which offer transactional guarantees to users, these guarantees only extend to resources used at Web servers, or between servers; clients (browsers) are not included, despite their role being significant in applications such as mentioned previously. Providing end-to-end transactional integrity between the browser and the application (server) is therefore important, as it will allow work involving *both* the browser and the server to be atomic. However, current techniques based on cgi-scripts cannot provide end-to-end guarantees. As illustrated in Figure 2 the user selects a URL which references a cgi-script on a Web server (message 1), which then performs the transaction and returns a response to the browser (message 2) *after* the transaction has completed. Returning the message during the transaction is incorrect since it may not be able to commit the changes.

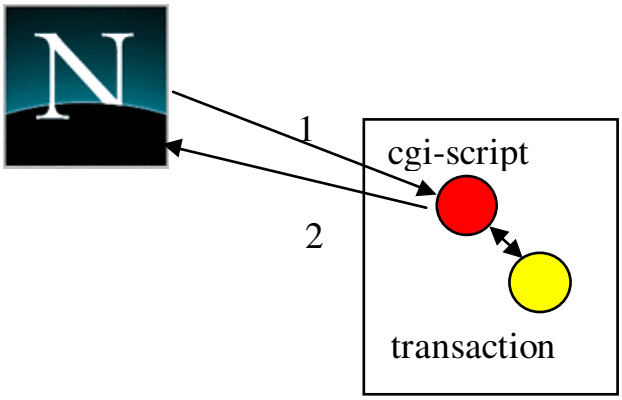

Figure 2: Transactions through cgi-scripts.

In a failure free environment, this mechanism works well. However, in the presence of failures it is possible for message 2 to be lost between the server and the browser, resulting in work at the server not being atomic with respect to any browser related work. Thus, there is no end-to-end transactional guarantee.

2. Is end-to-end transactionality possible?

Yes. There is nothing inherent in **any** transaction protocol (two-phase[1], three phase, presumed abort, presumed nothing, …) that prevents end-to-end transactionality. The transaction engine (essentially the coordinator) has very little effect on this, whether or not it is embedded in a proprietary service such as CICS,

[1] Most transaction systems use a two-phase protocol with presumed abort.

or within an industry standard application server such as HP-AS. What does make end-to-end transactionality difficult is that it requires a transactional participant to reside at each "end". Importantly it **does not** require a transaction coordinator and its associated baggage (e.g., CICS) to reside at each "end"[2].

There is a contract that exists between transaction coordinator and participants. In brief (and with many simplifications[3]), this contract states:

- Once the transaction coordinator has decided whether or not to commit or roll back the transaction, it guarantees to deliver this information to every participant, regardless of failures of itself or participants. Note, that there are various optimisations to this, such as a presumed abort protocol, but in essence the contract remains the same.
- Once told to prepare, a participant will make durable sufficient information for it to either commit or cancel the work that it controls. Until it determines the final outcome, it should neither commit nor cancel the work itself. If a failure occurs, or the final outcome of the transaction is slow in arriving, the resource can typically communicate with the coordinator to determine the current progress of the transaction.

In most transaction systems the majority of the effort goes into designing and developing the transaction coordinator engine, making it as performant and reliable as possible. However, this by itself is insufficient to provide a useable system: participants are obviously required. Although any contract-conformant participant implementation can be plugged into the two-phase protocol, typically the only ones that the majority of people use are those that control work performed on (distributed) databases, e.g., the aforementioned XAResource. This tends to result in the fact that many people equate transactions with databases only, and hence the significant amount of resources required for participant implementations. This is not the case: a participant can be as resource hungry or not as it needs to be in order to fulfill the contract. Thus, a participant could use a local file-system in order to make state information durable, for example, or it could use non-volatile RAM. It really depends upon what the programmer deems necessary.

With the advent of Java it is possible to empower thin (resource scarce) clients (e.g., browsers) so that they can fully participate within transactional applications. Transaction participants tailored to the application and environment in which they are to work can be downloaded (on demand) to the client to ensure that it can fully participate within the two-phase commit protocol. This is not a new concept and in fact we have been principle in demonstrating this technique for two different transaction engines over the years [MCL97a][MCL97b]. There is nothing special about the transaction system implementations that makes them more easily adapted to this kind of environment. In fact, we have participated in discussions with colleagues from IBM and Microsoft who agree that it would not have been any easier to implement such

---

[2] Some transaction systems do require a component footprint to reside at each end, but this is an implementation choice, and not required by the general concept of end-to-end transactionality.

[3] For example, the exclusion of heuristics.

functionality in their products. The specialisation comes from the end-point resource, i.e., the *client-side participant*. However, even there the same techniques could have been used to write an equivalent participant for, say, CICS.

3. OLTP versus OO-TP

So, are "traditional" online transaction processing (OLTP) engines more suited to end-to-end transactionality guarantees than "newer" object-oriented transaction systems? The quick answer is: no. Why should they be? It does not matter whether a transaction system is supported by proprietary remote procedure call (RPC) and stub generation techniques or by an open-standard remote object invocation mechanism such as a CORBA ORB; once the distributed layers are removed, they all share the same core – a two-phase commit protocol engine that supports durable storage[4] and failure recovery. How that engine is invoked, and how it invokes its participants is immaterial to its overall working.

The real benefit of OO-TP over OLTP is openness. Over the past 8 years there has been a significant move away from proprietary transaction processing systems and their support infrastructure to open standards. This move has been driven by users who have traditionally found it extremely difficult to move applications from one vendor's product to another, or even between different versions of a product from the same vendor. The OMG pioneered this approach with the Object Transaction Service (OTS) in 1995, when IBM, HP, Digital and others got together to provide a means whereby their existing products could essentially be wrapped in a standard veneer; this approach would allow applications developed with this veneer to be ported from one implementation to another, and for different implementations to interact (something else which up until then was extremely difficult to do reliably).

It is therefore inaccurate to conclude that OLTP systems are superior in any way to OO-TP equivalents because of their architecture, support environment, or distribution paradigm. OLTP systems are monolithic closed systems, tying users to vendor specific choices for implementations, languages, etc. If the experiences gained by the developers of efficient and reliable implementations of OLTP are transposed to OO-TP, then there is nothing to prevent such an OO-TP system from competing well. The advantages should be obvious: open systems allow customers to pick and choose the components that they require to develop their applications, without having to worry about vendor lock-in. Such systems are also more readily ported to new hardware and operating systems, allowing customers even more choice for deployment[5].

---

[4] So that it can guarantee to complete any "in flight" transactions upon recovery.

[5] "Now let me see, which version of CICS can I run on my laptop with RedHat 7.2?"

3.1 The OTS

The OTS architecture provides standard interfaces to components that all transaction engines possess[6]. It does not modify the model underlying all of the existing different transaction monitor implementations: it mandates a two-phase commit protocol with presumed abort, and **all** implementations of the OTS must comply with this. It was intended as an adjunct to these systems, and not as a replacement. No company that has spent many years building up reliability in such a critical piece of software as transactions would be prepared to start from scratch and implement again. In addition, no users of such reliable software would be prepared to take the risk of transitioning to this new software, even if it were "open"[7]. In the area of transactions, which are a critical fault-tolerance component, it takes time to convince customers that new technology is stable and performant enough to replace what they have been using for many years.

Although the CORBA model is typically discussed in terms of client-server architecture, from the outset its designers did not want to impose any restrictions on the types of environment in which it could run. There is no assumption about how "thin" a client is, or how "fat" a server must be in order to execute a CORBA application. Many programmers these days simply use the client-server model as a convenient way in which to reason about distributed applications, but at their core these applications never have what would traditionally be considered a thin client: services that a user requires may well be co-located with that user **within the same process**. CORBA was the first open architecture to support configurable deployment of services in this way, correctly seeing this separation of client and service as just that: a deployment issue. **There is nothing in the CORBA model that requires a client to be thin, and functionally deficient.**

The OTS is a comparable model to CICS, Tuxedo, ArjunaClassic [GDB95], DEC ACMS. It differs only in that it is a standard, and allows interoperation. There is nothing fundamentally wrong with the OTS architecture that prevents it from being used in an end-to-end manner. The OTS supports end-to-end transactionality in exactly the same way CICS, or any other "traditional" OLTP would: through its resources. We have already shown how we can support this with HPTS [MCL97b], and it has been an OMG success story for many years.

3.2 To ORB or not to ORB, that is the question

There also appears to be some confusion as to whether CORBA implementations (ORBs) are sufficient for mission-critical application. It is true that back in the mid-90's when implementations first appeared on the

---

[6] The OTS specification essentially only specifies the two-phase commit protocol engine, and leaves the implementations of persistence and concurrency control (necessary to obtain ACID properties) open. There is no such thing as an OMG Recovery service.

[7] This is evidenced by the fact that IBM did try both approaches with CICS and Encina, whose development by Transarc it closely funded during the late 1990s. CICS with an OTS veneer sold well, but Encina did not, despite having what many consider to be a better architecture.

market, their performance was not as good as hand-crafted solutions. However, that has certainly changed over the latter few years. The footprint of some ORBs is certainly large, but likewise there are other ORBs that have been tailored specifically for real-time, or embedded use. Companies such as IBM, IONA and BEA have seen ORBs develop over the years to become a critical part of the infrastructure that they have to control, and therefore they have their own implementations. Other companies have licensed ORB implementations from elsewhere.

The crash failure of an ORB does not typically mean that it can recover automatically and continue on from where it left off. This is because the ORB does not have necessary semantic and syntactic information to automatically check-point state for recovery purposes. However, by making used of suitably defined services such as the OTS and the persistence service, it is possible for applications to do this themselves or for vendors to use these services to do this for applications.

It may be said that OLTP systems provide this kind of feature out-of-the-box, and it may well be true. However, it is an unfair comparison: an OLTP system does just one thing, and does it well – it manages transactions. A CORBA ORB is meant to provide support for arbitrary distributed applications, the majority of which will probably not even need fault-tolerance let alone transactions. However, for those applications which do need these capabilities, it is entirely possible to provide **exactly** the same recovery functionality using OMG open standards.

Finally, the OMG standard specifies a large number of different services, many for specific deployment environments (e.g., space, or medical). The CORBA specification does not mandate that all of these services should be supported by every vendor. They are free to pick and choose those services that they wish to support. It is only the ORB that is mandated, since without this no form of distributed invocation is possible.

4. J2EE

Although the J2EE model is client-server based, as with CORBA there is nothing to prevent a client being rich in functionality. It is a deployment choice that is made at build time and runtime (obviously the capability for being so rich is required to be built into the client, and even if it were present, it would be down to the user to determine whether or not such functionality was required or was possible.) The “failure” of it being used in e-commerce and mobile applications is a subjective point, given that companies such as IBM, BEA and IONA are tying more and more of their infrastructural applications to Java and J2EE. These companies see J2EE as the key to their future, and they have had significant financial success as a result.

Note, although J2EE did not start out as an infrastructure that used CORBA, it quickly became evident that the experiences that the OMG companies had obtained over the years in developing CORBA were extremely important to **any** distributed system. As a result, over the past few years J2EE has got closer and closer to the CORBA world, and now requires some critical ORB components in order to run.

Even with the advent of Web Services, and protocols such as the Business Transactions Protocol (of which HP is a key player [BTP01]) which is intended for transactions in the Web, the back-end transactional resources will typically be driven by traditional transaction systems such as HPTS. It has taken Java several

years to be accepted into the group of languages that are considered "strong" enough to implement transaction processing engines, and this is the real reason why more take-up has not happened before now. There is certainly an argument to be made as to whether Java is the best language for all transaction systems, but that is a different issue.

Note, it is incorrect to say that the J2EE specification favours a web server as the repository for server-side objects. Neither does it specify a web browser as the preferred means of client interaction.

The typical way in which J2EE programmers use transactions is through the Java Transaction API (JTA), which is a *mandated* part of the specification. The JTA is intended as a higher-level API for programmers, to try to isolate them from some of the more complex (and sometimes esoteric) aspects of constructing transactional applications. The JTA does not imply a specific underlying implementation for a transaction system, so it could be layered on CICS, Tuxedo, or even ArjunaClassic. However, because the OTS is now the transaction standard for most companies[8] and it allows interoperation between different implementations, it was decided that the preferred implementation choice would be based on this (called the JTS, just to place it firmly in the Java domain!) The JTS is currently *optional*, but indications are that it will eventually become a mandated part of the J2EE specification.

5. Application Server means thin client?

No. As shown above, this is essentially a deployment issue. It is certainly correct to say that most J2EE programmers currently use a thin(-ish) client, with most of the business logic residing within the server; however, this is simply because this is the solution which matches 90% of the problems. Closer examination of all application server applications would certainly reveal that although thin clients are the norm, they are by no means the complete picture.

The application server has nothing fundamentally wrong with its model either. There is **nothing** to say that the client-side of an application server application has to be wafer-thin. If the client wants to embed functionality such as a two-phase aware transactional resource within itself, then that is entirely possible. In fact, a client could just as easily be embedded within an application server itself, if the footprint allowed. The reasons for not doing this are more to do with the footprint size than any architectural issue.

6. How does HPTS (Arjuna) compare?

The HPTS product is based on the Arjuna transaction system, which has been around since 1986 [GDP95]. It pre-dates CORBA, J2EE and Web Services, and has been used extensively by academic and industrial partners. It is held in wide regard by our colleagues in IBM, Microsoft and BEA. It has an extremely flexible and configurable architecture which allowed the key core components of HPTS to be embedded successfully

[8] With one noted exception: Microsoft.

within hand-held devices, laptops, and even mainframe solutions. This is something which our competitors do not currently have, making it extremely difficult for them to offer end-to-end solutions.

Prior to its being given an OTS veneer, Arjuna used its own optimised distributed invocation model. However, since being made OTS (and JTS) compliant, it is true to say that it has found its place amongst the traditional OLTP heavyweights such as CICS and Tuxedo. It is embedded within HP-AS, but is also available as a stand-alone product. As such, Arjuna does not rely on any application server functionality in order to run efficiently. It handles its own threading issues, uses a fully distributed two-phase commit protocol, and importantly does not impose any restrictions on locality of client, server, or transaction coordinator. It also does not make any assumptions about the relative "bulkiness" of clients or servers. In addition to being the only 100% pure Java implementation, HPTS is the only transaction system that truly supports nested transactions.

As our clients have found, the time and effort that has gone into the architectural development and implementation of HPTS, have paid off. Its performance compares extremely favourably with other transaction systems, it has a much easier deployment and management configuration, and scales well with increasing client load. In addition, it is highly reliable, and fault-tolerant.

In conclusion, it is entirely possible to provide end-to-end transactional guarantees using HPTS with or without an application server.

7. Q&A

In this section we shall look at some of the more common questions that have arisen around transaction systems, CORBA and J2EE:

- Modern transaction systems cannot provide end-to-end guarantees? This is incorrect. As we have shown, there is nothing intrinsic in the transaction protocol that is common to most implementations (CICS, Tuxedo, HPTS) that prevents this from occurring. There is a necessary collaboration between the transaction system and its end-points to accomplish this, but that is a relatively straightforward problem to solve.
- CORBA and J2EE imply a thin-client? This is incorrect. It is more accurate to say that most applications will tend to use a thin-client approach, but there is nothing in either of these models that mandates this.
- End-to-end transactional guarantees are the solution to the world's transaction problems? No. They are a solution to a specific problem.
- J2EE is closely tied to web servers and browsers? This is incorrect. Again, it may be a common implementation strategy for users because of the general availability of this type of software, but it is not a requirement.

- Application servers do not scale? Once again, this is incorrect. Current application servers (including HPAS) scale extremely well with increasing load, and the transaction component in particular scales well.
- J2EE mandates that transactions only run at the server? Incorrect. This is a deployment choice.
- CORBA implementations are too large, slow and not backed by industry? Incorrect. As we have shown, over the last few years CORBA has crept into all manner of devices, with all kinds of different configuration and runtime requirements. CORBA is the standard for interoperation, and J2EE is tying itself closely to this.
- HPTS (Arjuna) cannot support end-to-end transactional guarantees, and does not support two-phase commit. Incorrect. Arjuna has **always** supported two-phase commit, and we have been supporting end-to-end transactionality for many years.

References

[BTP01] http://www.oasis-open.org/committees/business-transactions/, June 2001.

[GDP95] "The Design and Implementation of Arjuna", G.D. Parrington et al, USENIX Computing Systems Journal, Vol. 8., No. 3, Summer 1995, pp. 253-306.

[MCL97a] "Constructing Reliable Web Applications Using Atomic Actions", M. C. Little, S. K. Shrivastava, S. J. Caughey, and D. B. Ingham, Proceedings of the 6th Web Conference, April 1997.

[MCL97b] "Providing end-to-end transactional Web applications using the Object Transaction Service", M. C. Little, OMG Success Story, OMG Web Site, 1997.

[MCL97c] "Distributed Transactions in Java", M. C. Little and S. K. Shrivastava, Proceedings of the 7th International Workshop on High Performance Transaction Systems, September 1997, pp. 151-155.

[MCL98] "Java Transactions for the Internet", M. C. Little and S. K. Shrivastava, Proceedings of the 4th Usenix Conference on Object-Oriented Technologies and Systems, Santa Fe, New Mexico, April 1998.

[OMG95] "CORBAservices: Common Object Services Specification", OMG Document Number 95-3-31, March 1995.