Understanding Transactions in the Operating System Context

Andrew P. Black
Digital Equipment Corporation

1. Introduction

Transaction processing has been described as "distributed computing that works".† And it is beginning to work better, and more efficiently, for more and more people. Nevertheless, it remains true that the model of computing embraced by TP systems is quite inappropriate for many kinds of applications, and that the various costs of transactions are too high for others, particularly for those applications that could benefit from only some part of the transaction model.

In short, I believe that transactions are an important tool for operating system support of fault-tolerance. Why, then, are they not part of the common systems that we use on a day-to-day basis? There are at least three reasons. First, the concepts are foreign to the people who design and build such systems: familiarization is needed. Second, transactions have been billed as a monolith that solves all problems, and system designers have tended to equate "transaction support" with the provision of a fixed set of transactional resources. For example, many people equate transactional operating system with transactional file system, since the file system is the main "database" that an operating system provides. (See, for example, reference 6.)

The third reason is that (with a few exceptions) the purveyors of transaction technology have failed to adapt their wares to the changing rôle of the operating system. Compared to its precursors, a modern operating system is less concerned with providing a collection of services and more concerned with making available a framework in which *clients* can provide the services that will be used by application programs. This trend is visible in the current popularity of "kernelized" operating systems, but is exemplified more clearly by the growth of special purpose servers – time servers, name servers, display servers – without which a modern distributed environment cannot function.

In keeping with this trend, operating system designers would rather include small, single-purpose abstractions that can be *composed* by servers (and by application programs) to provide the services that they need. This implies that the interactions between the abstractions be straightforward and well-characterized.

Currently, I do not believe that such a set of abstractions have been identified for transactions, although valuable contributions have been made by the implementors of Argus [5], Camelot [7] and Quicksilver [2]. The concepts that underlie the transaction mechanism are intertwined both with each other and with the various implementation techniques that have been developed in the context of databases. I believe that by attempting to separate the various aspects of the transaction concept from each other, and from their implementations, we will gain a better understanding of these concepts. This will enable us to move towards having a small set of independent abstractions that can be incorporated into an operating system as service interfaces, be implemented in ways that are outside of the scope of existing TP systems, and be used by programs that want some, but not all, of the properties of ordinary transactions.

This position paper represents some initial steps along this path.

2. What is a Transaction?

A transaction is frequently described as "an ACID operation"; the acronym "ACID" appears to have been coined by Haerder and Reuter [1]. Here is my interpretation:

A *All-or-nothing:* either the transaction completes, or it has no effect, even though there may be failures of some of the components involved in the transaction.

C *Consistency:* a transaction takes the system from one consistent state to another consistent state.

Author's address: Andrew P. Black, Digital Equipment Corporation, Cambridge Research Laboratory, One Kendall Square, Bldg 700, Cambridge, MA 02139. Electronic mail: black@crl.dec.com.

† Butler Lampson, personal communication.

I *Isolation:* the intermediate states of the data manipulated by a transaction are not visible outside of that transaction.

D *Durability:* the effects of a transaction that has completed will not be undone by a failure.

In order to understand these properties more clearly, I will axiomatize them; the treatment is meant to be suggestive rather than rigorous.

The "all-or-nothing" property is (perhaps, more usefully) known as *failure atomicity*, that is, the operation remains atomic with respect to certain kinds of failures. If a failure (and the subsequent recovery action) is modeled as an event $\lightning$ (like CSP's lightning [3]), failure atomicity says that these failures can appear to occur either before or after the transaction, but not in the middle. Thus the transaction may appear either to occur or not to occur, but never to "partially" occur. In symbols, for some meaning function $\mathcal{M}$ (and the corresponding disjunction $\vee$) and for some transaction a:

$$\mathcal{M}\,[\![\, a \;||\; \lightning \,]\!] \;\equiv\; \mathcal{M}\,[\![\, a \,]\!] \vee \mathcal{M}\,[\![\, \mathbf{skip} \,]\!] \;. \tag{A}$$

This can be rewritten to avoid dependence on a meaning function:

$$a \;||\; \lightning \;\equiv\; a \;\sqcup\; \mathbf{skip} \tag{A$'$}$$

where $\sqcup$ represents non-deterministic choice.

For what kinds of failure does this equation hold? If we partition the state of the system into two pieces, V (for volatile) and P (for persistent), then the $\lightning$ action can destroy V but not P, and must also reconstruct V. In other words, $\lightning$ includes the recovery action of the transaction system.

The durability property is actually a statement about when information crosses the boundary from V to P: *after* the transaction has completed, all of its effects are recorded in the persistent part of the system, and therefore these effects are immune from failure. As an equation, we write

$$\mathcal{M}\,[\![\, a \;;\; \lightning \,]\!] \;\equiv\; \mathcal{M}\,[\![\, a \,]\!] \tag{D}$$

or

$$a \;;\; \lightning \;\equiv\; a \;. \tag{D$'$}$$

Isolation is also known as *concurrency atomicity*, that is, the actions remain atomic with respect to certain kinds of concurrent activity. There is no guarantee of atomicity in the face of *arbitrary* concurrent activity, only with respect to specific kinds – typically, other transactions that are managed by the same system. For two transactions a and b, isolation says that

$$\mathcal{M}\,[\![\, a \;||\; b \,]\!] \;\equiv\; \mathcal{M}\,[\![\, a \;;\; b \,]\!] \;\vee\; \mathcal{M}\,[\![\, b \;;\; a \,]\!] \tag{I}$$

or

$$a \;||\; b \;\equiv\; a \;;\; b \;\;\sqcup\;\; b \;;\; a \;. \tag{I$'$}$$

Isolation is the property that allows the system to perform operations concurrently while still maintaining the fiction that they are performed serially; for this reason it is also called *serializability*.

Consistency is usually though about in terms of *states* of the system. If $\mathcal{K}$ is a predicate that determines whether a state is consistent, and $\mathcal{M}$ a meaning function that maps states to states, then for any state σ

$$\mathcal{K}\,\sigma \;\Rightarrow\; \mathcal{K}(\mathcal{M}\,[\![\, a \,]\!]\,\sigma) \tag{C}$$

Once again this can be written without recourse to $\mathcal{M}$. Let K be a consistency predicate on *sequences of operations*; then the consistency property is, for any transaction a

$$K\,\mathbf{skip} \;\Rightarrow\; K\,a \;. \tag{C$'$}$$

Clearly, $K\,\mathbf{skip}$ should hold for any reasonable K: if the null sequence of actions can make the database inconsistent, then we have little basis for reasoning.

It is often argued that "consistency" is not really a property of transactions *per se*; rather, it is a reflection of the way in which they are expected to be used. In most transaction systems, it *is* possible to commit a transaction that puts the system into an inconsistent state: it is the responsibility of the client, not of the system, to ensure that this does not happen. This is reflected in the equations: while A′, I′ and D′ have no free variables, C′ depends on *K*. It is possible to imagine a system in which consistency checks are performed as part of the commit activity; indeed, certain kinds of consistency checking form part of the SQL standard, which is implemented by commercial database systems such as RDB and DB2. However, it is clearly necessary that the *user* of the database system provide either *K*, a predicate that can be applied to sequences of actions, or $\mathcal{K}$, which can be applied to system states, before consistency can be enforced.

Although each of the A, C, I and D properties is characterized by a single equation, it does not follow that the the properties are independent. To see this, let us look at the interaction between all-or-nothing and durability. In a system that does not enjoy the all-or-nothing property, running *a* in parallel with $\lesssim$ may lead to *part* of the effect of *a* to be visible: rule A′ must be modified to $a \parallel \lesssim \equiv \mathsf{prefix}\, a$, where $\mathsf{prefix}\, a$ is the program that executes some non-deterministic prefix of the actions that make up *a*. In a system without durability, the effect of an action may be undone by a subsequent failure: rule D′ is weakened to $a \mathbin{;} \lesssim \equiv a \sqcup \mathbf{skip}$. However, even this equation does not hold in a system that lacks both all-or-nothing *and* durability; in such a system, we have only the very weak property $a \mathbin{;} \lesssim \equiv \mathsf{prefix}\, a$.

3. Abstraction and Mechanism

I have argued that a modern operating system should not just provide useful abstractions, but must also provide the means whereby clients can provide additional abstractions. One useful tool in this effort is compositionality: in the context of transactions, this means that an OS client can easily compose several existing transactional resources into a *new* transactional resource that it in turn exports. In addition, it must be possible for an OS client to construct a transactional resource *out of whole cloth*, and yet have that resource coexist on an equal footing with system resources.

To make these things possible it seems to be necessary to expose some of the *mechanism* behind the abstraction. More precisely, in addition to exporting an interface for *users* of the abstraction, it is necessary to expose a second interface to other *implementors* of the abstraction.

To take a concrete example, in order to provide the abstraction of failure atomicity, the system must include an implementation of some sort of multi-phase commit. In order to permit new servers to take part in this commit protocol, its interface – for example, the particular messages that are sent in two-phase commit – must be exposed. Similar arguments apply to the implementors' interface to concurrency control (to obtain isolation) and to the data manager (to obtain durability). There are also other, less obvious interfaces that must be exposed; for example, various components of a nested transaction system or a system using timestamp concurrency control expect certain properties of transaction identifiers.

4. Failures

As discussed above, both durability and failure atomicity depend critically *as concepts* on the division of the system into two parts, volatile (*V*) and persistent (*P*). In the absence of failure, neither durability nor failure atomicity imposes any restriction on a transaction system. The same is true if *both* parts fail together: the guarantees of durability and failure atomicity are void by assumption. The case where *P* fails but *V* does not is usually ruled out by the mechanics of the implementation, which ensure that *P* cannot fail without also causing *V* to fail. The case in which the durability and failure atomicity axioms are valuable is when *V* fails while *P* continues to operate – and this is why we conventionally call *V* the volatile or transient state, and *P* the stable or persistent state.

However, we are apt to read too much into these names. It is not required that *P* be a disk or a tape for the failure atomicity property to hold; the guarantee merely states that *if V* fails but *P* does not, then either all of the transaction's effects will be visible in *P*, or none of them will be.

In the database world, the stable state is either raw disk storage, or some abstraction thereof with a read-block/write-block interface. In the operating system world, this can be generalized: we need to understand what the interface

---

[TM] RDB is a trademark of Digital Equipment Corporation; DB2 is a trademark of International Business Machines Corp.

should be between *P* and a client that wants the durability property, and between *P* and a client that wants in addition the all-or-nothing property. If the client also wants to use system-provided concurrency control primitives, how can these interfaces be composed?

Two alternative implementations of *P* are particularly interesting to me. One is the case where *P* is implemented as one or more remote processes accessed across the network. Such a *P* does not deserve the epithet "stable storage", but does have failure characteristics that are independent of *V*'s. The other case arises in implementing nested transactions; a subtransaction makes its changes "durable" by writing to a data area associated with its parent. The difference between a top-level transaction and a nested transaction is that the top-level transaction makes its changes durable by writing to the disk. Given the appropriate interface to *P*, the code that implements a transaction can be parameterized so that it can be used for either a top-level or a nested transaction.

5. Summary

I propose that the transaction model can be a useful guide to incorporating fault-tolerance support into operating systems. However, I believe that rather than adopting transactions *as is*, they should be subdivided into independent components, and that each component be isolated from its implementation by a well defined interface. It should be possible to freely intermix and compose different components that implement both the persistent and the volatile aspects of an operation in different ways. This cannot be done with traditional TP and database systems, because they have taken the view that increasing the number of transactions per second is more important than using abstract interfaces. This is probably true for traditional TP applications, but an important motivation for adding transactional concepts to operating systems is to broaden the base of application. A major rôle of the system should be to define the interfaces by which different implementations of different parts of the transaction model can cooperate.

References

[1] Haerder, T. and Reuter, A. "Principles of Transaction-Oriented Database Recovery". *Computing Surveys* **15**, *4* (December 1983), pp.287-317.

[2] Haskin, R., Malachi, Y., Sawdon, W. and Chan, G. "Recovery Management in QuickSilver". *Trans. Computer Systems* **6**, *1* (February 1988), pp.82-108.

[3] Hoare, C. A. R. *Communicating Sequential Processes*. Prentice Hall, Englewood Cliffs, NJ, 1985.

[4] Kronenberg, N. P., Levy, H. and Strecker, W. D. "VAXclusters: A Closely-Coupled Distributed System". *Trans. Computer Systems* **4**, *2* (May 1986), pp.130-146.

[5] Liskov, B. "Overview of the Argus Language and System". Programming Methodology Group Memo 40, M.I.T., Laboratory for Computer Science, February 1984.

[6] Mueller, E. T., Moore, J. D. and Popek, G. J. "A Nested Transaction Mechanism for LOCUS". *Proc. 9th ACM Symp. on Operating Systems Prin.*, October 1983, pp.71-89.

[7] Spector, A. Z., Bloch, J. J., Daniels, D. S., Draves, R. P., Duchamp, D., Eppinger, J. L., Menees, S. G. and Thompson, D. S. "The Camelot Project". *Database Engineering* **9**, *4* (December 1986). (Also Tech. Rep. CMU-CS-86-166).