Co-existence of Transaction and Non Transaction-Managed Activity in a Persistent Object Store

Frans A. Henskens[1] and Maurice G. Ashton[2]

[1] *School of Electrical Engineering & Computer Science, The University of Newcastle, N.S.W. 2308, Australia*
Frans.Henskens@newcastle.edu.au

[2] *Avondale College, Cooranbong, N.S.W. 2265, Australia*
Maurice.Ashton@avondale.edu.au

Abstract

Persistent object stores provide an execution environment in which data and its inter-relationships are, by default, retained in their original form beyond the lifetimes of the program or programs that created them. Stability mechanisms ensure that such stores always start up in a self-consistent state, even after non-orderly shutdowns that result from events such as power outages or hardware failures. An efficient means of implementing stability uses Directed Dependency Graphs (DDGs) to facilitate execution of user processes in parallel with updates to the durable store image. The authors have previously shown how these DDGs can be extended and used to provide optimistic, transaction-based concurrency control for processes executing in persistent object stores [2].

The management of persistent objects differs from that afforded by conventional DBMS because the entire dataset exists in the same repository. As in conventional systems, it is appropriate that some data is accessed (queried and mutated) independently of the transaction system. In this paper, we examine the issue of interaction between processes that execute under transaction control with those executing independently of the transaction system. Interestingly, this co-existence is achieved without enforcing transaction semantics on the independent activity.

1. Introduction

The descriptor, *persistent system*, is applied to a wide range of systems offering a variety of features as described and classified in [3, 15]. The work described in this paper is based on persistent systems that offer the following features:

1. Data structures and relationships are, by default, retained in their original form beyond the lifetime of the program or programs that created them.
2. Address spaces and processes are orthogonal. Processes execute in a logically single-level object store. This addressing environment is typically shared by a number of processes associated with multiple users and consequently the normal protection mechanisms offered by conventional virtual memory management techniques are not applicable. Rather, protection is provided through the use of mechanisms such as capabilities [7].

Systems that provide such an environment include Monads [10-12, 16], Grasshopper [5, 6, 14] and Speedos [13].

In a system where multiple concurrent processes execute in a single-level object store and in the absence of protection or concurrency control mechanisms, any process may interact with the objects by either querying or mutating them. Objects in the store become dependent on each other through the activities of the processes [9, 17]. A store

Presented at International Conference on Systems Engineering (ICSeng'05), IEEE Computer Society, August 2005

that is able to re-start (either from orderly or unexpected shut-down) in a state that respects such dependencies is said to be *stable* and exhibit stability.

The graph-based stability mechanism developed by Jalili [8] uses directed graphs to record dependencies formed between processes and objects. This information is recorded in directed dependency graphs (DDGs) and used by the stability mechanism to allow parts of the store to be stabilized/rolled-back independently (thus avoiding stop-the-world management) and to determine the extent of checkpoint and rollback operations. There are similarities in the information recorded in stability DDGs and the information required for transaction management and concurrency control. However, there are also differences, namely: the DDG stability technique maintains dependency information on a per-process rather than a per-transaction basis; the DDG stability technique records information about dirty-read and write accesses, whereas transaction isolation also requires knowledge of clean-read accesses; stability checkpoints and transaction commits have different semantics.

The techniques described in [1, 2] extend the directed-graph-based stability scheme that implement transaction-based concurrency control for single-level persistent object stores. They are optimistic and support separate management for concurrent transaction-managed and stability-managed activities that co-exist in the object store. This support ensures that transactions are aborted if they are compromised either by transaction-managed or stability-managed activity. Importantly, assessment of each transaction's potential for success is made at the completion of each involved process time slice, so that if the ACID properties of the transaction have been compromised this is discovered earlier than with other optimistic schemes.

The graph-based transaction management scheme has been extensively evaluated and measured by simulation experiments and the authors intend to separately report the obtained results. They may be summarized as showing that the graph-based transaction management scheme has performance consistent with that of conventional pessimistic and optimistic bulk data management systems. Significantly, the simulation results show that the graph-based technique provides excellent general-purpose performance across a wide range of transaction sizes, levels of concurrency and object store sizes.

In this paper we investigate the interaction of transaction-based and non-transaction-based process activity as these processes access the common set of store objects.

2. Concurrent transaction and non-transaction-based access

Conventional DBMSs protect shared data by ensuring every access to the data complies either explicitly or implicitly with transactional requirements. By contrast, in a persistent system any process may access any data object for which the process has appropriate access rights. In the absence of any further protection mechanism, it is possible for a data object involved in a transaction to be accessed by processes not involved in that transaction. This could lead to the transaction's loss of integrity.

Two options may be considered for preserving transactional consistency in persistent systems:

1. Enforce transaction behavior on access to all data objects in the persistent store. Such an approach is regarded as inflexible as it restricts computation to the transaction model only [4].
2. Provide support for coexistent transactional and non-transactional activity.

In the remainder of this paper the relationships between transactional and non-transactional activities in persistent systems are discussed on the basis that option 2 above, applies.

3. Computation spaces

A stable store may be viewed as a collection of objects acted on by a set of processes. As processes interact with objects, dependencies form between them as described in [8]. Where stability is implemented using incremental check-point, these dependencies are recorded and later used to determine which subset of data objects is written to the durable store, or reverted to a previous state, on a checkpoint or roll-back operation. In the absence of transaction-managed activity, the store may be seen as a single, stable, object repository. When transactions are introduced, this single store is dynamically partitioning into two spaces:

- The stable store, where stability operations are managed, as usual, by the operating system, and
- The transaction-managed store, where stability operations are managed by the transaction manager.

This approach is illustrated in Figure 1. Using the view expressed in [4], the stable store can be seen to be under the control of an all-encompassing “transaction”, beginning with the completion of one checkpoint and extending to the next checkpoint. By default all data entities exist and execute within the stable store, which manages all non-transactional activity. On the execution of an event indicating transaction activity, responsibility for the stability of entities involved in that transaction passes to the transaction manager. The transaction manager then assumes responsibility for durability of entities modified by the transaction and also for transaction atomicity and concurrency control thus ensuring the transaction conforms to the required ACID properties. In other words, entities involved in transactions are moved to the transaction-managed space for the duration of the transaction as illustrated in Figure 1.

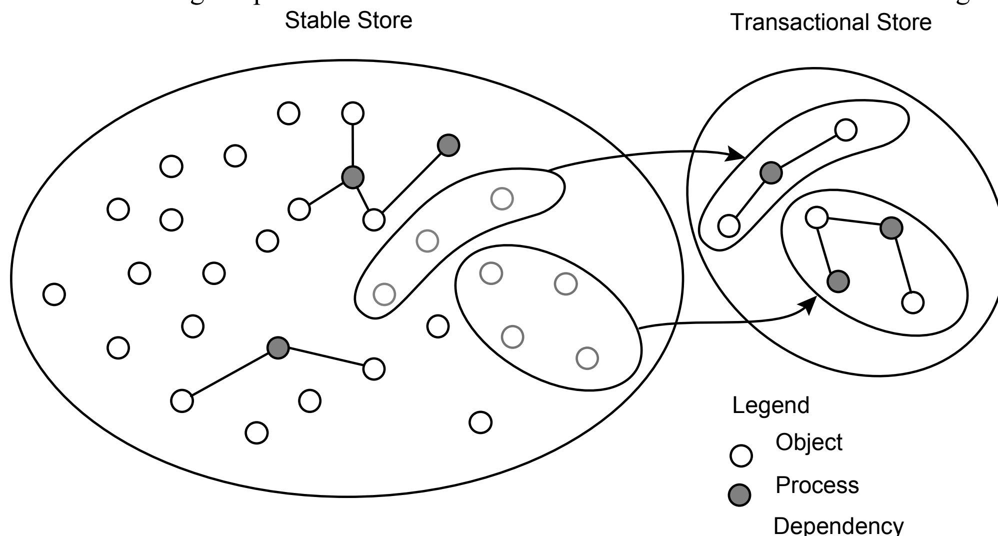

Figure 1 Entity management.

Presented at International Conference on Systems Engineering (ICSeng'05), IEEE Computer Society, August 2005

A transaction continues until one of three possibilities occurs:

- The event associated with a COMMIT-TRX instruction is executed, signaling the successful completion of the transaction,
- The event associated with an ABORT-TRX instruction is executed, signaling the unsuccessful completion of the transaction, or
- An unexpected system shutdown occurs, which also results in the unsuccessful termination of the transaction.

After any of these actions or events, object management reverts to the stable-store. In a sense, the above technique avoids rather than solves the conflict between store stability and transactional data processing by partitioning the set of addressable objects into subsets comprising, respectively, those that can be managed using transactions and those that cannot. This partitioning of the store is dynamic and transparent to the user and/or programmer, who remain unaware of the movement of data between the management schemes in the same way that programmers are unaware of data movement between the durable and computational store. In this way orthogonality of persistence is maintained and durability of transactions is decoupled from durability provided by the stability scheme.

Store partitioning raises issues about managing interaction between transactional and non-transactional activities. These issues are discussed in the following section.

4. Interaction between stable and transaction-controlled processes

Transactional and non-transactional processes interact in two ways:

- A transaction accesses a stability-managed entity.
- A non-transactional process accesses a transaction-managed data entity.

These interactions have consequences for the management of both transaction and non-transaction-managed processes. At the end of a timeslice the activities executed by the process during that timeslice are recorded in the DDG and decisions are made about the subsequent actions and viability of any involved processes. In the case of a transaction process the consequences of those accesses are used to determine whether the transaction should continue or be aborted.

In the discussion that follows, *T* represents a transaction process, *N* represents a non-transaction process, and *E* represents an entity accessed by both *T* and *N*. Described edge insertions apply to the DDG describing inter-entity dependencies.

4.1. Transaction access to stability-managed entities

In this case a transaction process *T* reads or modifies an entity *E* that is represented as a node of a DDG in stability-managed space. A stability-managed entity *E* is either:

1. An unmodified entity represented by a single node stability DDG, or
2. A modified entity represented by a node in a multimode stability DDG. A data entity only exists in such a DDG because that data entity has been modified since it was last check-pointed. Consequently any data node in a multi-node DDG is connected by at least a single write edge to another node in that DDG.

The possibilities for the access performed by the transaction on non-transaction-managed data are:

1. A transaction process reads an unmodified data entity: a clean-read edge is inserted *T* to *E*; the accessed entity is moved to the transaction-managed space.
2. A transaction modifies an unmodified data entity: a write edge is inserted *T* to *E*; the accessed entity is moved to the transaction-managed space.

3. A transaction process reads a modified data entity: a non-final dirty-read edge is inserted *T* to *E*; the integrity of transactions is maintained by all of the following possible subsequent actions.
    a. A COMMIT-TRX operation on *T* propagates to a commit on *E* hence *N*.
    b. An ABORT-TRX operation on *T* does not propagate to *E* or affect *N*.
    c. A checkpoint operation on *N* propagates to a checkpoint of *E*; the existing dirty-read edge *T* to *E* becomes a clean-read edge; *E* is moved into the transaction-managed space.
    d. A rollback operation on *N* propagates to *E* hence forcing *T* to abort.
4. A transaction process writes to a previously modified data entity. A write edge is inserted *T* to *E*. Subsequently integrity is maintained as follows.
    a. A COMMIT-TRX operation on *T* propagates to *E* causing it to be committed and hence to a checkpoint of *N*.
    b. An ABORT-TRX operation on *T* also propagates to rollback of *E* and *N*.
    c. If a checkpoint operation on *N* was allowed to occur it would propagate to *E* and *T*. This would effectively represent a premature commit on transaction *T*. To avoid a compromise of the transaction's integrity, either:
        i. Require both *T* and *N* to immediately roll back, or
        ii. Adopt an optimistic approach by leaving the edge in place in the hope that *T* commits before the *N* checkpoints.
    d. A further write access by *N* violates the isolation of *T* (inconsistent view) requiring *T* to abort and *E* to roll back, propagating to rollback of *N*. Further write operations by *T* are acceptable.

4.2. Non-transaction access to Transaction-managed entities

The possibilities for an access performed by a non-transaction process to transaction-managed data are:

1. The non-transaction process reads an unmodified data entity. No edge is inserted *E* to *N* because no stability-relevant dependency is created.
2. The non-transaction process mutates an unmodified data entity. A write edge is inserted *N* to *E*. The clean read edge is given "final" status preventing an inconsistent view by *T*. Integrity is maintained in the future by:
    a. A COMMIT-TRX operation on *T* does not propagate to *E* or *N*.
    b. An ABORT-TRX operation on *T* does not propagate to *E* or *N*.
    c. A checkpoint operation on *N* propagates to checkpoint of *E* but not *T*. Blocking edges are added to *T* and any entity modified by *N* to ensure that *T* is not compromised by an inconsistent view of data modified by *N*.
    d. A rollback operation on *N* propagates to rollback of *E* but not *T*.
    e. Further writes by *N* do not affect the transaction *T's* view of the data.
3. The non-transaction process reads a modified entity. A dirty-read edge is inserted *E* to *N*. Future integrity is maintained by:
    a. A COMMIT-TRX operation on *T* propagates to commit of *E* but not *N*. The dirty-read edge between *N* and *E* is removed.
    b. An ABORT-TRX operation on *T* propagates to rollback of *E* and *N*.
    c. If a checkpoint operation on *N* was allowed to occur it would propagate to *E* and *T,* effectively representing a premature commit on *T*. To avoid a compromise of the transaction's integrity, either:
        i. Before checkpoint of *N*, check for dependencies that would propagate the checkpoint to *T* in which case both *N* and *T* are rolled back.

   ii. Acknowledge that the situation represents an unacceptable relationship requiring both *T* and *N* to roll back.

  d. A rollback operation on *N* does not propagate to *E* or affect *T*.
  e. *T* may execute further writes on *E* without violating the isolation of *T*. There is no requirement to limit repeated reads by *N*.

4. The non-transaction process mutates a previously modified entity. A write edge is inserted *E* to *N*. This action compromises the isolation of transaction *T*, hence *T* is aborted propagating to rollback of *E* and *N*.

5. Conclusion

This paper shows that the use of DDGs to provide transaction-based concurrency control in persistent object stores permits the co-existence of transaction and non-transaction processes executing against a common data set. This is an interesting consequence, allowing a level of flexibility not available in conventional DBMS-managed systems, which require all activity to be controlled by transactions (be they programmer defined or automatically created by the DBMS).

10. References

[1] Ashton, M.G., Management of Data, Access and Concurrency in Persistent Systems, Ph.D, *School of Electrical Engineering & Computer Science*, University of Newcastle, 2005.

[2] Ashton, M.G. and Henskens, F.A., Directed Dependency Graph-Based Concurrency Control for Persistent Systems, in *First International Workshop on Object Systems and Software Architectures*, (Victor Harbor, Australia, 2004), 10-25.

[3] Bem, E.A., Issues in Persistent Systems, in *Proc., The 6th IDEA Workshop*, (Rutherglen, Victoria, Australia, 1999).

[4] Blackburn, S.M., Zigman, J.N.,. Concurrency — The fly in the ointment?, in *Proc., The Third International Workshop on Persistence and Java*, (Tiburon, CA, USA, 1999), Morgan Kaufmann, 250 - 258.

[5] Dearle, A., di Bona, R., Lindstrom, A., Rosenberg, J. and Vaughan, F., User-level Management of Peristent Data in the Grasshopper Operating System, Universities of Adelaide and Sydney, 1994.

[6] Dearle, A., Di Bona, R., Farow, J., Henskens, F., Lindström, A., Rosenberg, J., Vaughan, F., Grasshopper: An Orthogonally Persistent Operating System. *Computing Systems*, *7* (3). 289-312.

[7] Fabry, R.S., Capability Based Addressing. *Communications of the ACM*, *17(7)*. 403-412.

[8] Jalili, R., A Failure Transparent Distributed Persistent Store, PhD Thesis, *Basser Department of Computer Science*, University of Sydney, Sydney, 1995.

[9] Jalili, R., Henskens, F.A., Entity Dependency in Stable Persistent Stores, in *Proc., The 28th Hawaii International Conference on System Sciences*, (Hawaii, U.S.A, 1995), IEEE, 665 - 674.

[10] Keedy, J.L., The MONADS-PC: A Programmer's Overview, University of Bremmen, Germany, 1989.

[11] Keedy, J.L., Projects Associated with the Department of Computer Structures: The Monads Project, University of Ulm, 1997.

[12] Keedy, J.L. and Brossler, P., Implementing Databases in the Monads Virtual Memory, in *Proc., The 5th International Workshop on Persistent Object Systems*, (San Miniato, 1992), Springer-Verlag, 318-338.

[13] Keedy, J.L., Espenlaub, K., Hellmann, R. and Pose, R., SPEEDOS: How to Achieve High Security and Understand It, in *CERT Conference*, (Omaha, Nebraska, 2000).

[14] Lindström, A.G., User-level Memory Management and Kernel Persistence in the Grasshopper Operating System, PhD Thesis, *Basser Department of Computer Science*, University of Sydney, Sydney, 1996.

[15] Morrison, R. and Atkinson, M.P., Persistent Languages and Architectures, in *Proc., The International Workshop on Computer Architectures to Support Security and Persistence of Information*, (1990), Springer-Verlag and the British Computer Society, 9-28.

[16] Rosenberg, J., The MONADS Architecture A Layered View, in *Proc., The Fourth International Workshop on Persistent Object Systems*, (Martha's Vineyard, Mass, USA, 1990), Morgan Kaufmann Publishers Inc, 215-225.

[17] Rosenberg, J., Dearle, A., Hulse, D., Lindstrom, A. and Norris, S., Operating System Support for Persistent and Recoverable Computations. *Communications of the ACM*, *39* (9). 62-69.