https://ntrs.nasa.gov/search.jsp?R=19920004492 2020-03-17T13:46:14+00:00Z

Maintaining Consistency in Distributed Systems*

Kenneth P. Birman

TR 91-1240
November 1991

IN-62-CR
53366
p-38

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*The author is in the Department of Computer Science, Cornell University, and was supported under DARPA/NASA grant NAG - 2-593, and by grants from IBM, HP, Siemens, GTE aand Hitachi.

Maintaining Consistency in Distributed Systems*

Kenneth P. Birman

November 12, 1991

Abstract

How should distributed systems preserve consistency in the presence of concurrency and failures? For systems designed as assemblies of independently developed components, concurrent access to data or data structures would normally arise within individual programs, and be controlled using mutual exclusion constructs, such as semaphores and monitors. Where data is persistent and/or sets of operations are related to one another, transactions or linearizability may be more appropriate. Systems that incorporate *cooperative* styles of distributed execution often replicate or distribute data within groups of components. In these cases, group-oriented consistency properties must be maintained, and tools based on the virtual synchrony execution model greatly simplify the task confronting an application developer. All three styles of distributed computing are likely to be seen in future systems - often, within the same application. This leads us to propose an integrated approach that permits applications that use virtual synchrony to with concurrent objects that respect a linearizability constraint, and vice versa. Transactional subsystems are treated as a special case of linearizability.

Keywords and phrases: Transaction, atomicity, monitors, serializability, linearizability, virtual synchrony, object-oriented programming, distributed computing, federated databases, fault-tolerance.

1 Introduction

The emerging generation of database systems and general purpose operating systems share many characteristics: object orientation, a stress on distribution, and the utilization of concurrency to increase performance. A consequence is that both types of systems are confronted with the problem of maintaining the consistency of multi-component distributed applications in the face

*The author is in the Department of Computer Science, Cornell University, and was supported under DARPA/NASA grant NAG-2-593, and by grants from IBM, HP, Siemens, GTE and Hitachi.

of concurrency and failures. Moreover, large applications can be expected to combine database and general purpose components. This paper discusses four basic approaches to the distributed consistency problem as it arises in such hybrid applications:

- *Transactional serializability,* a widely used database execution model [Gra78,BGH87], which has been adapted to distributed and object-oriented settings by several research efforts [LCJS87, LS83,Spe85].
- *Traditional operating systems synchronization constructs,* such as monitors [Hoa74], used within individual system components, and with no system-wide mechanism for inter-object synchronization.
- *Linearizability,* an order-based execution model for object-oriented systems with internal concurrency proposed by Herlihy and Wing [HW90] (similarly restricted to synchronization within individual objects).
- *Virtual synchrony,* a non-transactional execution model used to characterize consistency and correctness in ***groups*** of cooperating processes (or groups of objects, in object-oriented systems) [BJ87a,BJ87b,BSS91].

We suggest that no single method can cover the spectrum of issues that arise in general purpose distributed systems, and that a composite approach must therefore be adopted. The alternative proposed here uses virtual synchrony and linearizability at a high level, while including transactional mechanisms and monitors for synchronization in embedded subsystems. Such a hybrid solution requires some changes to both the virtual synchrony and transactional model, which we describe.

The organization of the paper is as follows. We begin by reviewing the database data and execution models and presenting the transactional approach to concurrency control and failure atomicity. We then turn to distributed systems, focusing on aspects related to synchronization and fault-tolerance and introducing virtually synchronous process groups. The last part of the paper focuses on an object oriented view of distributed systems, and suggests that the linearizability model of Herlihy and Wing might be used to link the virtual synchrony approach with transactions and "internal" synchronization mechanisms such as monitors, arriving at a flexible, general approach to concurrency control in systems built of typed objects. We identify some technical problems raised by this merging of models and propose solutions. Appendix A formalize the composite model, and Appendix B discussed some practical implications, concluding that the approach could be supported with at most minor changes to existing systems.

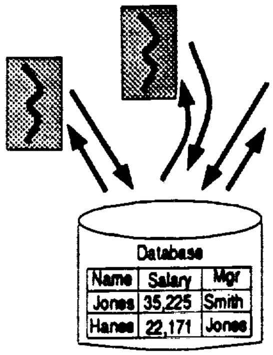

FIGURE 1. INDEPENDENT APPLICATIONS INTERACT WITH A DATABASE SYSTEM

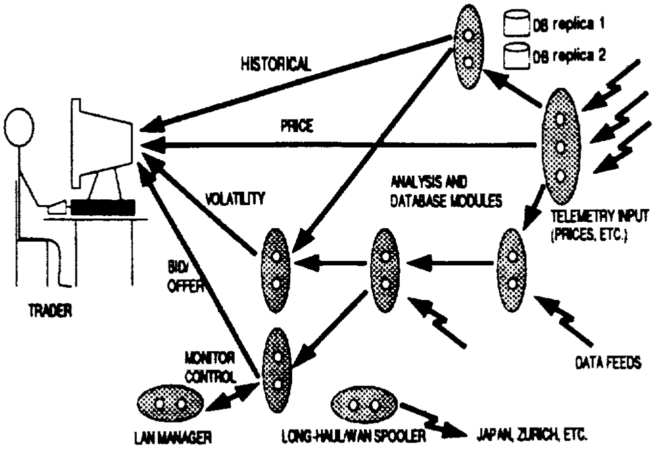

FIGURE 2. A GENERAL PURPOSE DISTRIBUTED SYSTEM IN SUPPORT OF A WIDE-AREA BROKERAGE APPLICATION

2 Database systems

It is difficult to find any single definition of a "database system": data management systems arise in such a wide variety of applications, and use such widely varied data models and access methods, that definitions easily become mired in operational details of particular implementations or specific system realizations. Furthermore, the emerging class of object-oriented database systems present the database through interfaces that can be customized both from the (external) perspective of an application that employs the database, and from the (internal) perspective of the data objects stored in the database, and this clouds the question of recognizing a database when one is encountered. Nonetheless, database systems share a number of distinguishing properties (see also Figure 1):

1. *They manage data objects.* Data objects may be of varied types, and may be multiply instantiated. For example, in a database organized as a set of relations, each tuple might be considered a composite data object, with each of its fields consisting of more primitive objects. In a transactional file system, the objects might be files, or records within files. It should be remarked that the database system "knows" the object structure of the database, and consequently can recognize (and potentially delay or reorder) operations upon them at runtime.

2. *The database satisfies consistency constraints that may span multiple objects.* Database applications are typically abstracted as functions that transform a database from one consistent state into another consistent state. Although an update may modify several data items, this implies that updates should be treated as a single, logically indivisible action.

3. *Application programs are generally designed to run independently.* They interact with other database applications through the shared database.[1] The *database concurrency control* problem arises when independent application programs access a shared database concurrently. The role of the concurrency control mechanism is to prevent these concurrent accesses from violating the consistency of the database.

4. *The major fault-tolerance problem is to restore persistent data into a consistent state after a crash.* That is, when a failure occurs, the system model assumes that the database managed at the failed site will become temporarily unavailable, but that the site will eventually resume execution and that the data it managed must be recoverable. This can be contrasted with "high-availability" schemes in which failures are tolerated by dynamically reconfiguring the system to restore function even if some components remain unavailable for long periods. In this case the recoverability of data at a failed site may be relatively unimportant, particularly when unavailable data can rapidly fall out of date.[2]

5. Application programs can be coarsely classified as "short running" or "long running". The latter typically access data objects in repetitious ways, for example by performing some operation on all objects of a given type.

A *transaction* is a sequence of database accesses performed by a single database application program, or a single computational thread, in a distributed setting that supports concurrency and remote procedure calls. Each transaction begins either when the application program is started or when it explicitly executes some sort of *begin* primitive, performs a series of read and update operations on the database, and is terminated by normal program completion or execution of a *commit* operation. An application that is unable to complete successfully, or crashes, is said to *abort*. A correctly functioning database system must preserve the effects of committed transactions, while completely erasing any effects of aborted transactions.

The *transactional execution model* is concerned with two problems:

Serializability: When multiple transactions concurrently access a shared database, the database system may interleave the execution of operations provided that the execution that results

[1] Throughout this paper, when we talk about a single program or a single application, this should be understood to encompass distributed programs that may be multi-threaded and that use remote procedure calls to invoke services from other programs.

[2] There is, of course, a substantial literature concerning software techniques for database replication. However, when the amount of data stored in the database is very large, the cost of these techniques can be prohibitive, and their use substantially increases overall system complexity. A consequence is that although there are several important high availability database machines and products, most general purpose database systems do not support high availability through replication.

is indistinguishable from one that could have resulted from the execution of the same set of transactions in some serial ordering.

Failure atomicity: The database state will reflect the effects of committed transactions and the effects of aborted transactions will never be seen. As noted above, a transaction may abort explicitly, or may abort implicitly when the application program or the machine on which it was running crashes.

The transactional model is backed by a rich literature concerned with formalizing the model, solutions to the concurrency control problem (such as two-phase locking), extensions (such as *nested transactions* and *nested top-level transactions* [Mos82]), and approaches to implementing failure atomicity (such as the use of write-ahead logs, intention lists, and recovery blocks). For our purposes in this paper we will not need to discuss these approaches in detail, nor will we consider the issues raised by applying the model within specific sorts of database systems. However, the handling of long-running transactions warrents further comment.

Efficient support for long-running transactions is more difficult than for short ones, because of the need to enforce serializability. An application serialized after a long-running transaction may be delayed for an extended period of time, and in a setting that mixes frequent short transactions with occasional long ones, database designers often go to considerable lengths to "manage" the long-running transactions. For example, the designer may undertake to fragment a long-running transaction into multiple short transactions, to develop a special purpose concurrency control mechanism, or to limit the execution of long-running transactions to periods of the day when the performance impact will be minimized. We will return to this point in Sec. 3.

Summary

Before we move on, it will be useful to reflect briefly on the close match between the transactional execution model and the assumptions we listed about database systems. The view of database applications as active computational agents that access the database through a well-defined interface is a key to the whole approach. It allows the concurrency control algorithm to intervene by intercepting and possibly delaying operations when necessary. Concurrent access by independently developed programs creates both a well-defined synchronization problem and also an intuitively appealing correctness constraint, namely that the system should behave as if access were not concurrent.

Additionally, we claim that the practicality of the overall approach is tied to the assumption that operations constituting a logical database access - a transaction - are all initiated by a single program (or by a set of threads sharing a common ancestral thread). That is, although transactions can be nested within one another, they ultimately have a single, identifiable parent. This assumption offers a straightforward way to assign a transaction identifier to each program that can be carried along with any communication it performs and used to distinguish database operations issued by independent transactions from operations issued within a single transaction. For example, the parent program's process identifier can be used, perhaps extended by counters in the nested case. To see why this is important, suppose that some transaction were composed of operations originating apparently unrelated programs. Perhaps, the programs actually are related, but by some external attribute not apparent to the system; for example, they might all be commands issued by the same user. Even if it were desirable that this set of operations be treated as a transaction, the database system lacks a transaction identifier with which to group them together. Such identifiers are used within the concurrency control algorithm and the database commit protocol. The ability to automatically associate a unique identifier with the sequences of operations that constitute a transaction is thus a key element of the database model. This issue is discussed in more detail in [CR91].

3 General purpose distributed systems

The preceding section discussed characteristics of database systems and applications. What can be said about "general purpose" applications running in a distributed environment?[3]

1. *General purpose systems often manage data, but lack any sort of data schema or well-defined data interface.* Although concurrency control problems arise, the operating system lacks a point at which it could intervene in a standard, system-wide manner.

2. ***Although general purpose distributed applications may manage collections of files containing complex data structures, these structures are more heterogeneous than the objects managed by a typical database system.*** The operating system interfaces used to manipulate objects external to the application tend to be byte-stream or virtual memory oriented, rather than object or record oriented. Moreover, studies have found comparatively little dynamic sharing of files between independent applications [Ous85,Bak89,KS89], particularly if "sequential sharing", such as occurs when files are used to communicate between phases of a single application, is

[3] It has often been suggested that almost all general purpose applications could be designed to run over database systems; indeed, several important commercial products have precisely this structure. However, as we show, such an approach requires special care by the programmer. Moreover, indirection through a database could be expensive. Most distributed operating systems impose few restrictions on application designers, and this freedom results in applications that are significantly less structured and "regular" in their style of interactions than typical database applications.

treated separately. Although file locking may be used for coarse grained mutual exclusion, the fine-grained concurrency control problems seen in databases don't arise (or only arise in a limited manner) at the level of data management in typical distributed applications.

3. *Consistency constraints are highly application specific.* Although such constraints may span multiple system components or data objects, the separation between program and data objects is less clear then in a database setting, and hence less amenable to a simple characterization. We will see a number of examples illustrative of this point below.

4. *Distributed applications are often designed as collections of cooperating programs.* Direct interactions between applications and direct sharing of data are common (Figure 2). Indeed, the exploitation of memory mapped shared data represents one of the hot topic in contemporary operating systems research.

5. *Failure recovery is often by some form of active replication that permits a backup to take over for a system component that fails.* Two considerations underly this. One is that distributed systems often contain huge numbers of components – both hardware and software. A system incapable of continued operation in the presense of failures of some components would provide very low availability. The second is that critical aspects of the state of the system will often be maintained in volatile memory by the servers, and because the size of such data may not be huge, the cost of replicating it may actually not be very high. Indeed, most "general purpose" application programs maintain comparatively small amounts of "state". Even where the state of the application resides in a file, studies of file access patterns have found that most files are fairly small and that most application programs access small numbers of files [Ous85,Bak89, KS89]. This is in contrast to the situation of a database server, which may be faced with the management of huge amounts of data. Moreover, even if only a small part of the database is "critical" to system availability, the whole thing would normally have to be replicated. Thus, a distributed application may be able to employ active replication selectively within the application itself, using "online" replication techniques (i.e. by replication of volatile data, or by maintaining replicas of selected data files). This possibility is not commonly seen in database applications.

These points suggest that general purpose distributed applications embody a different design philosophy than typical database systems. Indeed, distributed systems commonly violate the assumptions underlying the transactional model: they are simply not structured transactionally – and we will see this even more clearly in the next section. On the other hand, a substantial fraction of distributed applications manipulate files, or collections of files, in ways reminiscent of a database system. In some cases such files will be shared among independently executing application programs and updated concurrently. In these cases, transactional mechanisms can be extremely useful [LW86,Spe85].

Many non-database programmers have pointed to the ability to back out of partially completed actions by means of an abort operation as particularly convenient [SW89].

Thus, we are faced with a contradictory situation. Transactions are obviously of value in general distributed systems, but there seem to be major aspects of such systems that the transactional model fails to address. Indeed, as we will see in the remainder of this section, distributed systems raise a number of issues that require solutions unlike anything seen in traditional database settings, and this will bring us to the question of how the resulting collection of techniques can be integrated into a single system.

3.1 Client-server software

Earlier, we noted that programs in distributed settings employ direct interprocess communication, whereas database applications more often interact indirectly, through the database itself. In part, this is related to what is known as the *client-server* model of distributed systems design. According to the client-server approach, applications are organized into a set of services that are shared among a collection of client programs. The client programs typically use remote procedure calls to invoke server operations.

Like a database server, a server in a client-server environment is faced with concurrency control issues stemming from the presense of multiple, concurrent clients. However, the solution favored in most servers is not at all transactional. There are several reasons:

- As noted above, servers lack a simple way to identify related operations. One might think that the process identifier could be used for this purpose, but because many components of a large distributed system remain continuously operational, the identification of a single process with a single transaction simply yields extremely long-running transactions. In practice, there seems to be no obvious way to introduce transaction identifiers in long-running programs without explicit code modifications (for example, by inserting calls to a transactional begin/end mechanism, as is done in programs running under CAMELOT [Spe85]).
- A server generally manages its data directly in memory, hence the operations on data tend to be hand-crafted for each server. To employ a transactional mechanism in such a setting would often impose a substantial burden on the developer of the server program. This is in contrast to a database system, where data access is by queries or file operations, creating an obvious interface at which operations can be intercepted and "scheduled".
- The concurrency control problem most commonly encountered in a server arises when multithreading is used to obtain concurrency in a single address space. However, this type of

concurrency is readily controlled using synchronization tools such as monitors or semaphores, which are a standard part of most threads packages. Because the data is typically all in memory, the serializability problem reduces to a simple mutual exclusion problem.

A database researcher confronted with this situation would no doubt express concern that client-server systems might tend to behave in non-deterministic ways that could make interference between concurrent users visible and reduce reliability [Wei89]. In fact, although interference effects do arise and may even be visible to users of contemporary distributed services, this rarely poses any major difficulty for the designer. First, the consistency guarantees provided by typical servers are weaker than for a database system, hence the user might not be "surprised" by non-serializable behaviors. Additionally, the designer often has recourse to other mechanisms, such as file-locking primitives, that can be used to restrict concurrency when necessary.

Two examples will help illustrate these points. A *source code control system* is a software environment for managing the software components of large applications. Clearly, there is the potential for a concurrency control conflict when multiple software developers work on related modules of the system. However, at the level of the operating system, the source files are simply unrelated text files, and the application programs that make up the source control system are apparently unrelated and independently executed. To edit a file one would normally "check out" that file and its dependents, but the file dependency information would be stored in a second file. Thus, the operating system itself has no special information about the relationships between files. The editing operation (and any related compilations or printing operations) would appear as independent accesses to the files; any concurrency problems that could arise being addressed at the application level by preventing multiple users from checking out the same file simultaneously. Finally, the check-in operation that terminates the "transaction" could occur long after the check-out (days) and again will not be explicitly identifiable at the operating system or file system level as being related to the original check-out event. Thus, a transaction occurs, but it lacks many of the features that distinguish transactional database application [CR91].

A second example involves access to the network information service (NIS). Such a service maintains a set of maps from names to values, and is useful in performing such tasks as mapping from host names to internet addresses, from service names to communication port numbers, or from user names to encrypted passwords or mail files. An application that is searching an NIS system could see an inconsistency if an update were occurring at the same moment. For example, a single host might appear to have multiple addresses. However, this sort of brief inconsistency is generally considered tolerable. Most operating system software that actually uses an NIS system will run correctly even if such events occur, and the administrators of the system generally avoid making updates during working hours. The only guarantee is that an update will *eventually* become visible everywhere.

Now, one could certainly argue that even if this behavior is common in current distributed systems, it reflects a sort of careless unreliability that is at best unfortunate and in the long term, unacceptable. This author agrees that every component of a distributed system should have a well-specified interface, and that the behavior under concurrent access should be easily understandable. On the other hand, having built quite a number of distributed servers, using both conventional and transactional concurrency control, the author finds it hard to see transactional mechanisms as a preferable alternative to monitor-style synchronization. Monitors can yield highly reliable applications, if the design methodology is sufficiently cautious and rigorous. On the other hand, transactions prove extremely awkward in applications such as these – so much so, that one is forced to change the model before using it at all, resulting in an overall approach that lacks much of the conceptual elegance of the original one and a style of programming that obscures much of the program logic behind layers of reasoning and meta-reasoning. To justify these claims, it will be useful to consider the NIS example in more detail.

Example: NIS Service

Figure 3 illustrates a linked list that might be maintained by an NIS service. The list contains 4 elements mapping from names to telephone numbers, and is being maintained in sorted order. Typical operations that an NIS service might support include lookup of the telephone number associated with some name, inserting a new entry, deleting an entry, or updating an entry. Of course, such a service would typically include multiple lists; indeed, fancier subsystems for this purpose such as Xerox's CLEARINGHOUSE, are capable of managing arbitrary linked data structures.[4]

This interface raises concurrency control problems at two levels. As noted above, one could imagine a multi-threaded NIS service; such a service would be faced with possible access conflicts when two concurrently active threads happen to access the same list. One can solve this problem by locking the nodes being accessed, perhaps using a hierarchical scheme to avoid risk of deadlock. Such an approach would guarantee that no thread ever sees a node that has only been partially linked into place, or that is being deleted.

A second approach would use transactions for access control. This presumes that the application accessing the service is linked with a transactional subsystem or coded in a transactional programming language, so that each operation will be correctly tagged with a transaction identifier. The resulting transactional service would have several advantages over the non-transactional one. For

[4] We use a linked list in this example simply to illustrate a basic point. Although the problems cited can certainly be overcome using specialized solutions, the same *types* of problems are encountered for almost any data structure one might adopt. Our point is thus not that link lists are somehow especially troublesome, but rather that the transactional model itself becomes an obstacle when an application employs application-specific data structures.

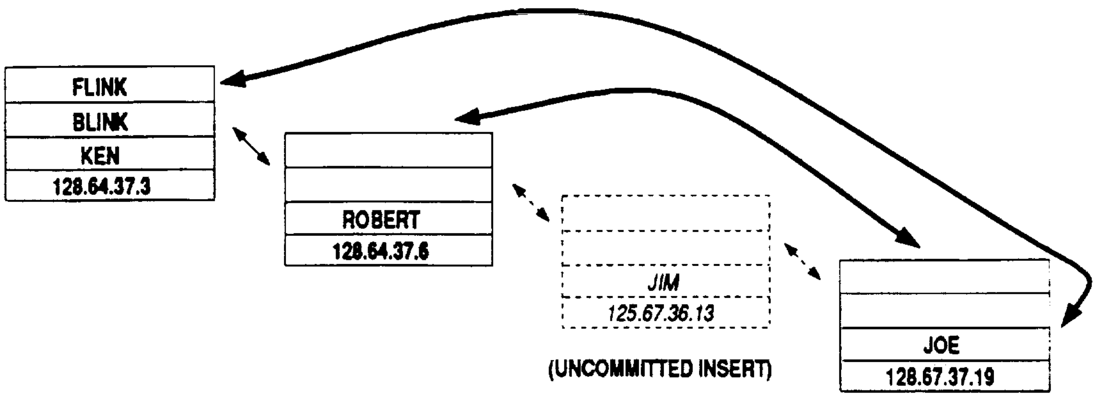

FIGURE 3: A LINKED LIST USED IN AN NIS SERVER

example, a transaction to insert a new user may create NIS entries listing that user's telephone number, home directory, workstation ID, etc. The transactional interface will ensure that such a composite operation occurs atomically, avoiding any risk that inconsistency will become apparent to users querying the service while such an operation is underway. However, users may now experience a different problem: poor performance due to the simplistic serialization model and the clumsy nature of concurrency control schemes not tailored to the problem.

These are strong statements. Why do we make them? The basic problem revolves around the issue of long-running transactions raised earlier. Let us assume that the transaction system uses two-phase locking for concurrency control – this is not the only option, but it is by far the most common approach. If a transaction on the NIS service runs for any significant amount of time, it will leave records locked in its wake. For example, if the data structure is represented directly as a sorted list a transaction that has read a record for user u would lock the record of every user u' with $name(u') < name(u)$. A transaction that has inserted or deleted a record will similarly leave the list in a locked state pending its commit or abort. Thus, essentially all forms of concurrent access to the list are excluded by a naive application of the serializability model. Essentially the same problems are seen with other common concurrency control approaches, such as timestamped concurrency control. Likewise, problems are encountered when working with other possible data structures, such as balanced trees or hashed lists.

The literature is full of solutions to problems such as these, but they tend make fairly sophisticated demands on the designer of the data structure. For example, one approach is to design a special purpose concurrency control scheme that exploits the semantics of this type of data structure; some algorithms and a good survey can be found in [SG88]. More realistically, designers might be urged to use pre-designed structures drawn from some sort of a toolkit developed by experts. Another approach might be to use what are called *top-level subtransactions*, which are a way to spin off independent transactions from within the body of an active transaction [LS83]. In such an approach, top-level transactions could be used to manipulate the links that interconnect nodes on the list, while the true transactional mechanisms are confined to operations on the data part of each node. (But this requires care; for example, it may not work if the insert operations are mixed with deletes). Yet a third approach would be to somehow relax the semantics of the list, presenting user's with an interface that explicitly anticipates the possibility that access will be concurrent and forces the user to anticipate some degree of non-determinism [Wei89].

Although any of these solutions might work in specific cases, we contend that none is satisfying in general settings where the class of applications to be supported is unknown and unconstrained. Development of a special purpose concurrency control based on an analysis of semantic properties of the list operations is likely to strain the abilities of a typical applications programmer. One would hope for a more straightforward approach. Restriction to some collection of pre-determined data structures would be severely limiting, although this is the approach used in many commercial products. Nested transactions with top-level subtransactions force the programmer to maintain a mental model concerning the scope within which each operation performed by the program will be visible: top-level actions are globally visible, while nested actions are visible only if the current transaction commits, and this can have a number of surprising effects. Moreover, a top-level action will not automatically be "rolled back" if the transaction that spawned it terminates, creating a possible need for some sort of explicit recovery mechanism to supplement the basic system support for transactions. This gets especially complicated when using the *nested transactional model*, in which *orphan* subtransactions might remain active long after the parent transaction aborts. As for the third approach, we see this as a reversion to the one used in non-transactional settings: it evades the issue by disabling the inconvenient aspects of the transactional model. In any case, the relevance of the approach is limited because existing transactional systems lack software support for such a scheme.

This reasoning is not purely empirical. The late Howard Sturgis, in describing experience with the Clearinghouse (which supported a transactional interface), pointed to every one of these problems [Stu86]. Clearinghouse eventually legislated against long-running transactions: atomicity was provided for groups of operations, but only if the group could be performed rapidly and without leaving the database "locked" for any extended period.

It would be hard to imagine a simpler problem than the development of an NIS service such as the one above. If solving this problem using the transactional model requires elaborate contortions, one can only question whether the model makes any sense for this sort of problem. Our point in this paper is that transactions are simply mismatched with the type of data structure, the type of access patterns, and the overall goals of a service like the NIS service. One certainly wants a computing environment in which the service will offer a fully specified interface and will correctly respect this interface, but it is also important that the developer have the freedom to use data structures that will perform well without engaging in a complex concurrency control analysis.

3.2 Linearizability

Herlihy and Wing examined concurrency control for object-oriented applications in a 1990 paper. Recognizing that transactional correctness is not always appropriate, but that non-deterministic object behavior is also undesirable, they suggest that an intermediate, order-based correctness approach be employed instead. The correctness condition that they favor is called *linearizability* [HW90].

The linearizability model assumes an object-oriented programming style, in which objects are accessed through *invocations* that return *results*.[5] Operation B is said to depend on operation A if the invocation of B was issued by a thread that had witnessed the result of A.[6] Such a dependency is denoted $A < B$. An object is said to *execute sequentially* if it accepts one invocation at a time, computes a result, and replies to the caller. An object that does not execute sequentially is *concurrent*. Given an execution of a concurrent object, the execution is said to be *linearizable* if there exists some sequential execution that respects the dependency relationship of the concurrent execution, and such that each operation yields the same results as in the concurrent execution.

Readers familiar with the work done by Lamport will recognize the *dependency* relation as a form of causality relation specialized to object-oriented interactions. In this terminology, an execution is linearizable if there exists an equivalent sequential execution that respects causality.

Although linearizability is similar in spirit to serializability, the approach focuses on concurrency control, and the model does not treat failures. The dependency relation is used instead of any notion of a transaction, and a concurrent object would need an operational way to reason about operation dependencies (such as the Psync primitives [PBS89]), much as a concurrency control algorithm needs a way to obtain and manipulate transaction identifiers. However, a database system that implements transactions would normally satisfy the linearizability property - provided that the serialization ordering respects invocation dependencies. Given a database system with this

[5] The notion of a result is quite flexible in this model; in addition to the value actually sent by the object to the caller, the result is understood to encompass the entire changed state of the object.

[6] In some models, B would be considered dependent on A even if A has merely been invoked and has not yet returned a result. However, the Herlihy/Wing model can represent this situation by considering operation A to have replied immediately with some sort of identifier that could later be used to *rendezvous* with the result.

property (any database complying with the XOPEN/XA architecture would satisfy this property), linearizability can be viewed as a weakening of the transactional model.

The linearizability model addresses the objections to transactions raised in the example above. On the one hand, it is less constraining than serializability; on the other, it avoids non-deterministic executions. This leads us to concur with Herlihy and Wing, who suggest that the correctness constraint be considered as a possible "minimal" one for distributed systems. Moreover, linearizability is relatively easy to implement; methods for doing so and proving the correctness of the resulting synchronization algorithms are discussed in [HW90].

3.3 Process groups in the Isis system

Above, we suggested that transactions are sometimes a poor match with the programming style used in distributed systems, although weaker order-based correctness properties remain useful. However, this is only part of the story, for there is a substantial class of distributed systems in which *stronger* synchronization properties are needed.

Synchronization and fault-tolerance issues also arise in settings where a distributed system is designed to achieve high availability through redundancy or a group execution mechanism. One solution to problems of this sort involves using *groups* of cooperating processes to implement key system services and functionality. The basic idea is that the members of such a group will back one another up, so that if one member fails but the others remain operational, services can be provided without interruption. A failed group member will need to "rejoin" the group when it recovers, but this is often done by simply copying the current state of the group to the joining process. Such an approach shifts the issues from failure atomicity at the level of update transactions to persistent storage to synchronization among programs cooperating to replicate data and coordinate actions.

The Isis system, developed by the author's research group at Cornell University, provides a collection of tools for building software using process groups. Isis applications raise a number of concurrency control and fault-tolerance problems that differ from those seen in transactional settings. This section briefly summarizes the issues that lead to the Isis execution model, called *virtual synchrony*, which resembles the transactional model in its spirit but differs in its details.

It will be useful to start with a review of how process groups are employed in Isis applications [BC90]. Groups are inexpensive in Isis: one application can employ large numbers of groups, and indeed a single program can join and leave groups dynamically. The members of a group need not be identical – they need only agree on the interpretation of group operations and the nature of the group "state," which is typically maintained online (in volatile memory).

Uses of groups seen in existing systems include the following:

- *Groups as an addressing construct.* Process groups are often used to accurately track a set of process that share some characteristic. For example, a group might represent a set of descendents of a common parent processes, a set of processes that have mapped a certain file into virtual memory, or a set of processes that need to be informed each time a certain event occurs.
- *Groups used to manage replicated data.* Replication is important in distributed systems, both for fault-tolerance and because having a local copy of a data item may enable an application to avoid communication delays. A process group might can be used to represent a set of processes each of which has a copy of the data item. Replication in the active sense intended here differs from data management in a database system, because the group members all take actions directly on the basis of their local data, and because update operations are not permitted to abort – a group member would typically issue an update (often, asynchronously) simply to inform other members of its actions, and using some sort of predetermined partitioning of the data or mutual exclusion scheme to avoid update conflicts. Thus, an online form of safety and liveness is the primary objective here, whereas a database system is more oriented towards safety of external data and recoverability.
- *Groups used to subdivide a workload.* Process groups are useful for load balancing, for example when a collection of server processes are available for some task and it is necessary to share the incoming requests among them.
- *Groups used for high availability (fault-tolerance).* Beyond replicating data, a process group can be used to ensure that a service will remain continuously available despite failures.
- *Groups used for control.* An important class of distributed application involves using a group of processes to monitor for some condition and trigger an appropriate reaction if the condition occurs. Such a process group basically implements a distributed state machine [Sch86] and can be designed to mask failures so as to carry out the desired action exactly once and exactly when desired. The approach is preferable to a centralized one because an action can generally be taken local to where an event occurs.

These group-based styles of computing raise synchronization problems that resemble but differ from the ones seen in transactional systems.

- When groups are used as an addressing construct, there arises the question of synchronizing changes in the group membership with read-accesses to the group membership, i.e. for use in applications parameterized by the membership.

- When groups are used to manage replicated data, there is the further need to synchronize updates so as to avoid conflicts and ensure consistency.
- When groups are used to subdivide a workload, the issue arises of how the workload subdivision scheme should be structured. Ideally, the division of labor for each request should be evident to all the group members; in this manner, a server can process requests promptly on receiving them without first running a potentially complex distributed agreement protocol. This, in turn, requires that requests be seen in consistent orders by the members of a group.
- When groups are used for fault-tolerance, synchronization issues arise out of the need to coordinate the actions of a primary server with its backup, and with updates to the group state.
- When groups are used for control, synchronization arises out of the need to coordinate the actions of the processes implementing the state machine, and to ensure that each action is taken exactly when necessary and taken exactly once.

Informally, the virtual synchrony model concerns itself with these concurrency control issues by introducing synchronization at the level of events seen by group members (see Appendix A for a formal treatment of the model):

- **Membership changes.** Associated with each group is a list of its members (called a *view* of the membership in ISIS). Process group views change in a well-defined, one-by-one manner as processes are added or dropped from the list (voluntarily or because of failure). Members can monitor a process group, in which case they are each sent a "view change" message for each new view.
- **Communication.** In group settings, group communication is clearly important.[7] This is the problem known as *atomic group multicast*, and involves two elements: the "expansion" of the group address to obtain a list of processes that will receive copies of the message, and the fault-tolerance and ordering properties of the communication primitive itself.
- **Asynchronous computation.** ISIS supports two styles of communication: *asynchronous*, meaning that the sender desires no replies, and *synchronous*, meaning that each request will be delayed until replies arrive from one, several, or all members of the destination group. Most ISIS applications are oriented towards asynchronous communication, which has important

[7] The option of performing group communication augments other communication mechanisms, such as point-to-point message passing, remote procedure call, data streams, or shared memory. In Isis applications, the designer normally picks the least costly primitive that matches the needs of the application. Thus, a single application may combine remote procedure calls, directly mapped shared memory, and group multicast communication.

performance advantages. In the most common case, asynchronous communication allows the thread that sends a message to continue computing without interruption, improving performance on the sending side, while also permitting a pipelined data flow that encourages efficient use of communication hardware by allowing multiple messages to be packed into a single data packet, improving performance in the data transport system and on the destination side. Achieving this sort of pipelining in a transactional setting would generally be possible only with very detailed knowledge of how the system has been implemented.

Like transactional serializability, virtual synchrony is a scheduling constraint – the model specifies permissible orderings in which group members may observe events such as group view changes, and deliveries of message sent using group multicast. The abstract idea is to assign each event a logical time using a system-wide logical clock. Distinct events are given distinct times. Group events (multicasts and view changes) trigger multiple local events in the group members, and these local events will all be given the same logical time. By delivering messages in the temporal order so established, all processes see the same events in the same order, and hence it is straightforward to keep group members synchronized. There are many ways to implement this abstraction [BSS91, PBS89,LLS90]; the ISIS solution is such that the application programmer does not have any direct access to logical timestamps and does not involve actually implementing any sort of system-wide clock. Nonetheless, knowing that the execution was virtually synchronous greatly simplifies the development of distributed software.

Virtual synchrony has several additional aspects. One is the guarantee that a multicast delivered to a group g at time t will be delivered to all the members of view$_t(g)$; the group view that applies at time t. A second guarantee is that *causality* will be preserved throughout the communication system. Causality is an extension of the idea of a FIFO ordering. Introduced by Leslie Lamport, we say that $m_1 \rightarrow m_2$ if information may have flowed from the point at which m_1 was sent to the point at which m_2 was sent. In a virtually synchronous setting, if $m_1 \rightarrow m_2$, then any processes that observe both m_1 and m_2 will observe m_1 before m_2. This guarantee substantially simplifies the development of asynchronous communication, and when combined with failure atomicity, also simplifies reconfiguration if a component fails during execution.

When we discussed linearizability, we noted that the dependency relation is a form of causal ordering. As was the case in that model, there is nothing in the virtual synchrony model that corresponds to a transaction. Instead, the focus is on preserving the causal relationship between operations, so as maintain the simplicity of asynchronous programs. On the other hand, the virtual synchrony model is quite evocative of the transactional one: one could view a multicast as a form of read operation that accesses a process group view, and a view change as an update operation. The causal ordering property relates sets of operations and hence is reminiscent of the idea of relating a set of operations by calling them a transaction. Certainly, the idea of having the system guarantee that operations will be scheduled according to an ordering rule is very "transactional" in flavor. However, these similarities do not suggest a way to implement virtual synchrony using transactions (the converse is possible, but virtual synchrony does not substantially simplify the problem and almost as much mechanism is needed as in a non-virtually synchronous setting).

Summary

In this section we saw that general purpose systems may have uses for transactional mechanisms, but also that such systems raise a number of non-transactional synchronization issues. We suggested that these issues are often best solved using traditional synchronization constructs, such as monitors and semaphores, and that linearizability represents an appropriate "minimal" correctness constraint for systems built in this manner: anything weaker could violate the programmer's intuitive understanding of how an interface should be expected to behave. Where problems are solved using cooperative algorithms based on the process group model, and we have argued that this may include a substantial and varied collection of applications, the distributed synchronization abstraction called virtual synchrony proves well matched to the issues that arise.

4 Object-oriented systems and Federated Databases

The observations made in the preceding sections are particularly relevant to two areas of current interest: object-oriented computing systems and federated database systems.

4.1 Object-oriented systems

The emergence of a new generation of object-oriented systems has revived the concurrency control issue in a new light, and leads us to the main contribution of this paper. We will say that a system is *object oriented* if it supports the definition and management of collections of objects, consisting of private data and an associated set of interfaces. In this view, programs are considered to be *active objects* while data is managed in *passive objects*. One can talk about object-oriented database systems – these support database-like query and storage functions, but permit the basic data types to be augmented with arbitrary user-defined data types. Object-oriented programming languages focus on linguistic support for interface definition, interactions between objects, and modular styles of software development. Object-oriented distributed systems provide, primarily, mechanisms to specify system components in terms of abstract interfaces and behaviors, and to compose larger systems out of these lower level components. Clearly, object-oriented database systems and object-oriented distributed systems bear superficial similarity. Nonetheless, it is important to realize that in many large systems, object orientation is primarily a method for standardizing the description of system components. One would not expect it to change the fundamental patterns of interactions between components residing in different address spaces, and thus the basic distinctions between transactional and non-transactional systems seen above would be expected to carry over into object-oriented settings.

Despite this, at the time of this writing the majority of object-oriented distributed systems proposals appear to revolve around transactional mechanisms. There are several reasons: first, because object-orientation makes interfaces explicit, it may seem more reasonable to think about scheduling object invocations in accordance with a concurrency control mechanism. Certainly, it will be important that object interfaces retain their meaning even in the presense of concurrency, because concurrency and true parallelism are expected to become increasingly important in future hardware designs. A second reason is that transactions have been so successful in database settings. A third reason is, perhaps, that there has been little effort on the part of the operating systems community to understand and explain why transactions have never "caught on" in non-database, distributed settings.

Thus, although one would expect object-oriented systems to share characteristics of both general and database systems, we are seeing object-oriented environments that focus on transactions to the exclusion of other consistency and concurrency control models. The conjecture of this paper is that such systems will prove powerful for database applications, but awkward where the goal is to support high availability client-server applications, cooperative computing applications, and other group-based applications.

4.2 Federated databases

A *federated database* is a database system constructed by building an interface to a collection of existing databases. The interface operates by mapping operations on an abstract database – the "federated" data model – into operations on the constituent databases. Database federation raises a number of issues because of the possibility that some data will be missing or inconsistent, that data models may not be completely matched, etc. Although we will not discuss solutions to these problems, we do wish to observe that the discussion of the previous sections is relevant to the design of software having this structure. The layer of the system concerned with solving the database federation problem is likely to have the characteristics of a conventional distributed program, while the underlying databases will clearly be transactional and more traditional in their adherence to a conventional database model. Of course, this is not always the case – many databases can be federated using simple techniques, such as nested transactions (i.e. a top-level query is broken into sub-queries on the constituent databases). On the other hand, there are types of federated databases that are not at all amenable to such an approach.

To give just one example, at the time of this writing, it had recently been announced that a group of banks and brokerages have cwformedg a partnership (the Electronic Joint Venture) to design and implement a wide-area distributed database for use in trading systems. The goal is to develop a highly reliable, automatically managed distributed database spanning more than 1000 "sites", each containing large numbers of workstations (some small, others containing up to 500 machines) and running varied application software. Network partition failures will be common in the system, hence the communication between sites must be viewed as asynchronous, with occasionally long delays.

It is obvious that systems of this scale will require a variety of sorts of software components, using varied fault-tolerance and communication technologies. Despite this, the reliability and security requirements seen in the system imply a need for a strong consistency model. To build the system it will be necessary to combine mechanisms such as reliable long-haul communication spoolers, transactional databases, wide-area concurrency control, etc.

This banking system is one of the many large-scale distributed applications contemplated by industry in the near future. Indeed, it would be hard to identify a major commercial sector that is *not* confronted by problems of a similar nature and scale. If we are to provide the software tools needed to solve such problems, while also achieving fault-tolerance and high reliability, an integrated consistency model that combines elements of the mechanisms described earlier will be crucial.

5 Proposed concurrency control model

This leads us to a concrete proposal. We suggest that object-oriented distributed systems adopt the following execution model.

Object groups. First, we suggest that the system be composed of objects and object groups composed of simple objects. For simplicity of the model, it may be best to understand simple objects as groups of cardinality one. We do not see any significant need for groups of groups at the present time, although the model should not preclude this possibility in some] future extension.

Virtual synchrony. When an object group contains multiple objects, we will use the virtual synchrony model described earlier. The events in the the virtual synchrony model are message send and receive events, group membership changes, and local actions by processes. We will interpret a "send" as an asynchronous invocation of an operation on an object or a multicast to an object-group (here, one presumes that the object group could specify the algorithm for mapping a group operation into operations on its components, including the use of group communication).[8] We will interpret a message delivery as the invocation of a method in an instance of an object. And, a group view change will be reported to an object-group member through an invocation of a *new view* method.

Linearizability. We will require that invocations of objects be linearizable. Intuitively, this corresponds to a requirement that "concurrent objects behave in a sane, predictable manner." Notice that because the virtual synchrony model respects causality, operations on object groups would normally be linearizable, provided that the application does not explicitly reorder operations in a manner that would violate causality. Thus, we can now develop a model spanning both individual objects and groups of objects.

Failure atomicity. Many systems will benefit from failure atomicity where persistent data is to be accessed. We suggest that such a property be viewed as a strengthening of the linearizability model to encompass permanence of committed actions. The mechanism should, however, be optional. Thus, we must now anticipate invocations from applications that lack any atomicity guarantee into failure-atomic subsystems, as well as the converse.

Serializability. We noted earlier that serializability is a more restrictive model than linearizability. Moreover, we saw that many general purpose distributed systems require failure atomicity but not fine-grained concurrency control. Thus, it makes sense to view transactional serializability as an optional strengthening of the linearizability model, used selectively where an application matches closely with the database model. Where transactions extend across object boundaries, the nested transactional model of Moss [Mos82] would be applicable.

Appendicies A and B formalize the model and discuss the practical implications of the approach. However, this paper does not argue for any particular implementation of the concurrency control mechanisms needed to support the model. Moreover, our stress is really not on the details of the model itself. Rather, our purpose is to suggest that merging the most popular consistency models may be the best way to deal with consistency in complex distributed systems.

[8] In particular, an object group might support some sort of fault-tolerance mechanism, such as the causal process-pair technique described in [BCG91], so that a singleton system component could be replaced with a fault-tolerant group without changing applications that employ the component. Of course, groups could also be used in explicit ways that would be visible to their users, but this sort of transparent fault-tolerance would have strong appeal to developers working within an existing software base, and is easily supported in our approach.

It should also be noted that this type of model may admit many possible implementations. The best concurrency control mechanism in a parallel processor with hardware support for shared memory may be very different from the approach that would be used in a wide-area distributed network, and we do not which to preclude the developer from engineering a system in the most efficient manner possible. On the contrary, we argue merely that a distributed system should behave in a predictable manner, and that a multi-level consistency model may be the most realistic way of addressing the complex needs of large, complex distributed systems.

The merging of the virtual synchrony and linearizability/serializability models requires that they be made "aware" of one another. When issuing a procedure call or object invocation from a virtually synchronous execution into a transactional one, the question arises of whether the transactional concurrency control scheme will correctly respect causality or other ordering constraints that arise from the group invocation and membership management layer. A concurrency control scheme that was "unaware" of such constraints might reorder operations in a way that would violate the causal dependency constraints shared by the virtual synchrony and linearizability models. This would prevent the designer from issuing concurrent invocations to the database, severely limiting performance. Thus, our approach would require that the lower-level concurrency control system either subordinate itself to constraints passed in by the software that invokes it, or be presented with sequential invocations.

The implication also goes in the other direction. A transactional subsystem within this model will require that its transaction identifiers be carried along in any object invocations it performs, and that any ordering mechanisms used at levels of the system "lower" than it be compatible with the concurrency control scheme employed by the subsystem as a whole. This requires some minor changes to the operating system software used for communication and thread management (specifically, each message should carry the transaction identifier, *if any*, of the thread that sent it, and each thread should acquire the transaction-id (again, if any) carried by the message that resulted in the most recent invocation. Recent work by the XOPEN/XA standards body [X/O91] has argued for precisely this approach, and a standard solution will be common in future operating systems.

When shared memory is used, our model may appear to prefer causal shared memory over alternatives [ABHN91]. However, causality can be implemented in a conservative way by limiting the degree to which the system runs asynchronously: if it is known that all operations in the causal past have terminated, no "work" need be done to enforce causality. For this reason, any of the popular shared memory approaches could be used in our model, provided that synchronization is adequate. However, causal shared memory would permit the highest level of concurrency.

Finally, some questions arise concerning the way that failures are detected and the situations in which a transaction can be permitted to commit: one would not wish to report a successfully committed transaction as having "failed". Again, we refer the reader to the appendix for details.

To summarize, we propose a layered model in which distributed software organized into cooperative groups can interact in a coherent manner with distributed objects and with distributed software organized using the transactional paradigm. Despite the increased operating system complexity introduced by such a layered model, we see it as advantageous because of the increased flexibility offered to the application designer.

5.1 Example: Network resource manager

To illustrate the points made above, consider a load-balancing remote execution service (a batch queuing system) that exploits the composite model proposed above. The function of such a service is to maintain a queue of jobs to be run. As machine resources become available, jobs are dequeued, executed, and then the user is notified of the termination status. We desire that the server be able to tolerate up to $k-1$ simultaneous failures of its members, and that it automatically restart any job placed on a machine that fails prior to job termination. Such a system will be concurrent in several ways: multiple jobs may be simultaneously pending, machines may come and go concurrently, and resource manager servers may fail or recover while the system is active.

First, notice that this type of subsystem will have a two-tiered structure. We can view the manager itself as a high level system, with the jobs that it executes remotely as a form of nested object invocation. This is illustrated in Figure 4.

The manager is a natural choice for implementation as a virtually synchronous process or object group. The group would contain k members to ensure tolerance of up to $k-1$ failures. Such an approach permits the full replication of the request queue and other control data structures; a primary/backup scheme can then be used to initiate actions on machines as they become available, i.e. with one of the manager-group members assigned primary responsibility for each machine. We used this approach in developing the Isis *network resource manager* [BC91] and had little difficulty in arriving at a highly robust, inexpensive tool that is now used within the community of Isis installations.

At the level of the jobs executed by the manager, however, one would like the cleanup of partial results of a failed job step to be as automatic as possible. Moreover, if the manager is really running arbitrary jobs in a heterogeneous environment, "automatic" checkpointing will generally not be practical (obviously, the application might do checkpointing on its own, but this would not

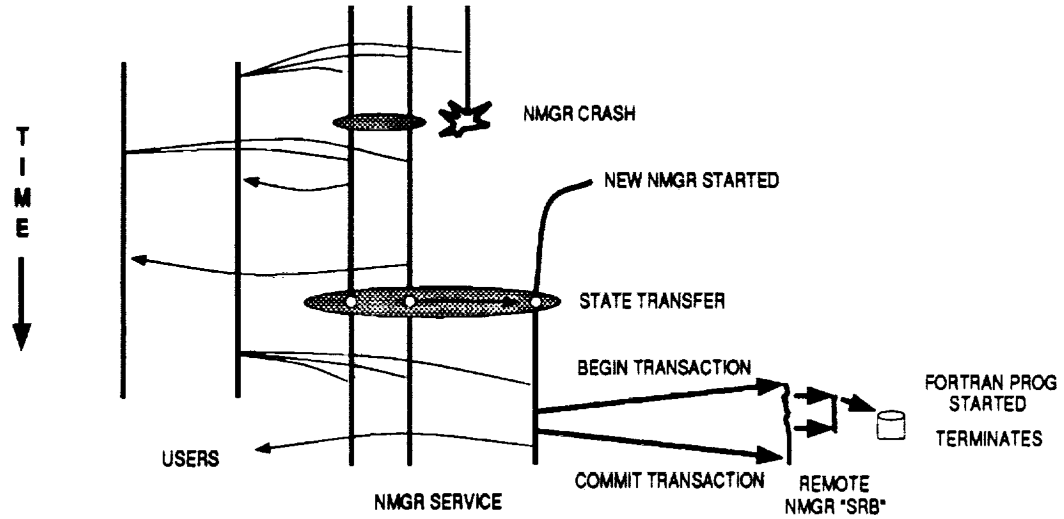

FIGURE 4. AN EXECUTION OF A NETWORK RESOURCE MANAGER

constitute a part of the manager subsystem). Despite this restriction, suppose that the manager is asked to run an n-step job, each step of which reads input generated by the prior step, computes some set of output files, and writes them out. If the i'*th* step were interrupted by a failure, it would be nice to know that any partial output generated during the run would automatically be deleted before the step is restarted elsewhere. Thus, it would be extremely convenient to run each job step as a transaction, making use of the failure atomicity mechanisms (concurrency control issues do not arise in this example).

This raises several points. One is that the jobs executed by a network batching system will generally not have been coded to use a transactional computing model. More often, they will be completely conventional programs written in a language like FORTRAN and designed to read a set of files, compute, and write out a set of files. Thus, the transactional begin and commit operations will probably have to be issued by the network resource manager and "carried along" within the job step, transparent to the actual user code that performs the embedded computation. In this manner, transaction identification information will reach the file system - in a conventional system this would not be the case.

But now suppose that step i is interrupted just as it is completing. If the manager assumes that step i has failed but the transactional file system considers the step to have completed and commits the actions it took, step i could be restarted without a prior cleanup. It follows that the manager - a level of the system using the virtual synchrony model - needs to participate in the commit protocol used by the subtransaction corresponding to step i, so that the step will be considered to have failed if (and only if) an abort is observed. This is the point alluded to at the end of the previous subsection: lacking some form of linkage between the failure mechanisms within the two models, it would be nearly impossible to obtain the desired behavior!

5.2 Example: Cooperation in a transactional database

For a second example, suppose that one wishes to implement a database system that exploits parallelism to improve performance. For example, a read-only query might be split into multiple subqueries, executed on different servers, and the results combined for delivery to the user. An update might be split into multiple updates, again depending on how data is partitioned among the participating database servers.

Query decomposition algorithms for distributed database systems are well understood [Ull80], and one could imagine using very sophisticated methods that adaptively adjust to changing workloads on the database managers, or to changing data distributions if the managers do not maintain identical data.

As in our previous example, one arrives at a two-level system design. The higher of these levels consists of a set of database servers which cooperate to manage the database. The servers would maintain the information needed to track the locations of data items, load on other servers, and other attributes of the database. This information changes dynamically: in the usual approach, servers inform one another of changes using some form of atomic multicast. Queries directed to this higher level would be decomposed, using this distributed data, into operations on individual servers. Moreover, execution of the decomposed query will now need to be monitored, so that a failed "job step" can be restarted elsewhere. The lower level software would be a conventional database system, accessible only through the higher level query decomposition mechanism.

As in the previous example, we have developed systems having this structure using Isis (however, we have not yet developed a serious database system using this purpose). We have been successful in using virtually synchronous tools for replicated data management, fault-tolerant execution of job steps, and distributed synchronization for the higher levels of such a system; the lower levels would clearly fit within a transactional or nested transactional model.

6 Conclusions

This paper has argued for a concurrency control mechanism that combines elements of a model called virtual synchrony with elements of the transactional serializability model, using the notion of linearizability as a sort of intermediate glue. The focus of the paper was on the match between synchronization models and classes of applications, hence the formal elaboration of the model was not a primary focus. Nonetheless, sufficient detail is presented to establish that the integration of these models should not be a overwhelming obstacle.

Ideally, our approach would require hooks in the transactional concurrency control mechanism (which may need to piggyback transaction identifiers on messages), the virtual synchrony mechanisms (which will need to coordinate failure reporting with the database commit protocol), and possibly even the mechanisms used to implement shared virtual memory. However, lacking these features, a mixture of conservative synchronization, runtime restrictions (such as limitations on calls from transactional into non-transactional subsystems), and interface compatibility checking could be used to implement the model over a convention operating system or database system.

We see substantial advantages to an integrated consistency model. Generations of distributed systems have left concurrency control and synchronization to the whim of the programmer, leading to software that embodies idiosyncratic and heuristic solutions, and is much less reliable than desired. The usual alternative has been to impose a transactional model throughout the system, but this proves too constraining for many distributed systems applications and hence is not always practical. Our approach would offer the distributed systems programmer a flexible collection of tools, based on a successful group-programming methodology at the highest levels, and permitting the use of transactional mechanisms at lower levels. The result is a substantial increase in systems reliability and the ability to treat systems behavior formally from the higher-level "weak" consistency models afforded by the virtual synchrony approach to the lower level "strong" serializability model.

Acknowledgements

The author would like to express his gratitude to Maureen Robinson, who did an outstanding job of preparing the figures for this paper. A number of people read an early draft of this paper; their comments were of great value in strengthening the presentation. The author is especially grateful to: Robert Cooper (Cornell), Barry Gleeson (Unisys), Holger Herzog (Siemens), Jacob Levy (Sun), Jishnu Mukerji (USL), Franklin Reynolds (OSF), Mike Reiter (Cornell), Dennis Shasha (NYU), Pat Stephenson (Cornell), Joe Sventek (HP), Chung Wang (TI), and Raffi Yaholom (Hebrew U.).

Appendix A: Execution model

This appendix introduces a formal model for the composite scheme described above. Our reasons for doing so are dual. First, such a model facilitates the development of correct application programs by providing a set of axioms that can be used in reasoning about the application program and establishing its behavioral properties. Secondly, a formal model makes explicit the obligations on the developer of an implementation of our scheme – although leaving open the engineering aspects of the problem. A correct implementation would then be one that provably provides the behavior required within the model.

Basic elements of the model

The basic entities of the model will be *objects*, denoted O_i, which interact through exchange of messages. We will say that the execution of each object consists of a series of *events*. An event e may be a send of a message, denoted $send(m)$, the delivery of a message m, or a local computation (below, we extend this to include group events). An object that fails executes the distinguished event *stop* and takes no additional actions. To relate our model to the one of Herlihy and Wing, delivery events would be understood to correspond to operation invocations and the delivery of replies.

Events are related by the flow of information through the system. After Lamport [Lam78], we will define the causality relation $\rightarrow$ on events as the transitive closure of the following base relation:

1. If e and e' are events local to an object O_i and e occurs before e', then $e \rightarrow e'$.
2. If $e = send(m)$ and $e' = deliv(m)$ for the same message m, then $e \rightarrow e'$.

Notice that the dependency relation of Herlihy and Wing can be expressed as a causal dependency by defining "e occurs before e'" with semantic knowledge concerning the nature of the events.

We will model the system as having an integer valued global clock that is not visible to the objects in the system. Because readers sometimes misunderstand a model to dictate implementation structure, it must be stressed that time is used to *reason* about the overall execution of events that occurred. Our work does not require that any sort of global clock actually be implemented. In fact, time can be eliminated from the model, but this leads to a more complex description. Let t denote a timestamp. We will write $O_i[t]$ to denote the event that occurred at object O_i at time t, or ϕ if there was none.

The system provides support for ***object groups***. Such a group is a set $g = \{O_i\}$. A group is created with some initial membership, and subsequently its membership changes as objects join (are added) and leave (are deleted or fail). All members of a group observe the same sequence of membership changes; we will write $view_i(g)$ to denote the *i'th* value taken on by the membership of group g, and $view(g)[t]$ to denote the view that applied at system time t ($\{\}$ if group g did not exist at time t).

We can now add ***group events*** to our basic event model. One type of group event is the definition of a new group view, which causes an event $view_i(g)$ to occur in the execution history for each object $O_i \in view_i(g)$. Additionally, an object can invoke a group multicast using the events $cbcast(g, m)$ and $abcast(g, m)$. Later, under conditions described below, these will result in a set of delivery events, one for each member of the group g in some view $view_j$ of the group ($i >= j$).

Basic execution axioms

Executions of the system are constrained by a set of rules. First, we will say that a history H is *complete* if for each event ***send(m)*** there is a corresponding event $rcv(m)$, and if for each multicast $cbcast(g, m)$ or $abcast(g, m)$, there is a corresponding set of delivery events $rcv_1(m), ..., rcv_n(m)$. Given a complete history H, we will say that ***H is legal*** if it satisfies the following constraints.

1. Each event e_i in H can labeled with a logical time $time(e_i)$.
2. Distinct events e_i and e_j are given distinct times, $time(e_i) \neq time(e_j)$ except for events corresponding to a single group view change, which all occur at the same logical time in the objects belonging to that view.
3. The rcv events corresponding to a single multicast are all delivered in the same view of the group g, and all members of the group register such a delivery. More formally, suppose that $O_i[t] = cbcast(g, m)$ or $O_i[t] = abcast(g, m)$. Then or there exists some $view_i(g)$ such that:
   (a) $t_0 \equiv time(view_i(g)) \leq t$, and
   (b) $t_1 \equiv time(view_{i+1}(g)) > t$, and
   (c) $\forall O_j \in view_i(g) : \exists t'' \in [t_0..t_1] : O_j[t''] = deliv(m)$.
4. Causality is respected.[9] If $e \rightarrow e'$ and e and e' are message send events or multicast invocations, then $\forall O_i \in dests(e) \cap dests(e') : time(deliv(e)) < time(deliv(e'))$.

[9] Our work on the Isis system convinces us that by default, a system should preserve causality throughout the communication system. Any weaker guarantees preclude the use of asynchronous communication, limiting the degree of communication-level pipelining (buffering) that can be achieved and correspondingly limiting performance [BCG91].

5. Both *cbcast* and *abcast* respect causality; *abcast* is also totally ordered within any single process group: If $e = abcast(g, m)$ and $e' = abcast(g, m')$ then either
   - $time(deliv(m)) < time(deliv(m'))$ for all members of g that receive both messages, or
   - $time(deliv(m')) < time(deliv(m))$ for all such members.

The above rules tell us how to decide if a complete history is legal. But, were one to create a history by taking a snapshot of the state of a system while it was active, the history will generally not be complete. Thus, we need a way to relate the histories generated during execution to the ones to which the model can be applied. Suppose that we are given a history H formed by writing down the events that occured in a distributed system while it executed. We will say that the system execution was *acceptable* if for any such H, there exists some other history $H' \in EXTEND(H)$ that is correct and legal.

We define *EXTEND(H)* to be the set of histories that can be obtained by taking H and extending the local histories for objects in H by appending any missing *rcv* events to correspond to unpaired *send* events in H. The members of the set differ in the ordering of these additional events. If an object fails (as seen below, this will be modeled through the execution of a distinguished, final, event *stop*, we extend the history of that object by prepending the *rcv* events before the *stop* event, but after any other events executed by the failed object prior to the failure.

We can now state that the goal of a system builder should be to demonstrate that the algorithms and protocols implementing the system yield only acceptable executions.

Our approach differs in one significant way from that of Herlihy and Wing. We have defined a system history to be acceptable if it can be *extended* into a legal history. Herlihy and Wing solve the same issue by eliminating unpaired *send* events (invocation events, in their model): they form a *complete* history by deletion of some events. The problem with this approach is that it would make it difficult to define multicast atomicity: none of the multicast axioms would apply to a multicast for which some destination has failed, since such a multicast would always be deleted from the history under consideration. Herlihy and Wing did not consider multicast, and hence were able to use a simpler model.

Failure model

Within our model, a failure appears as a *stop* event.

1. There is a system membership service that detects object failures and reports them in accordance with a "single system view" model of failures. That is, the membership service

behaves like a single, continuously operational, process. We note that this abstraction can be implemented in a distributed manner; an algorithm is given by Ricciardi [RB91].[10]

2. A failed object will be dropped from any groups to which it belongs: If $O_i[t] = stop$ then $\exists t' > t : \forall g : O_i \in view_g(t) => O_i \notin view_g[t']$.

3. Failure detection is, at best, approximate in any asynchronous distributed system [FLP85, RB91]. Therefore, we restrict the behavior of an object that has been perceived as faulty by requiring that failed objects not commit any operations: if a system history includes a *stop* event for object O_i, then O_i cannot append additional commit events to its local history.

The insight into this last requirement is that the history as "viewed" by the operational processes in the system may diverge from the history actually seen by an object while it executes. One form of divergence arises when an object crashes, but the operational processes continue execution as if it had executed certain events (such as multicast deliveries) to preserve the atomicity property. A second form of divergence, and the one to which we refer here, occurs if the system considers an object faulty (perhaps because of a communication failure), but the object is actually still operational. In this case, the system history will contain a *stop* event, but the object may not be immediately aware that it has been dropped from the membership list. From the perspective of the object, it will be possible to execute additional events for some period of time. If these are message *send* events, we can simply discard the messages in question – the failure detection mechanism of Ricciardi does just this (an operational object discards messages from failed objects). Where the object is transactional, however, we go further and require that it be impossible to commit an action in this "zombie" state (practical details are developed below).

Failure atomicity, serializability

A standard axiomatic definition of transactional atomicity and serializability can be employed within our model, such as the one in [BGH87]. This would be done by introducing events to model the execution of ***begin***, ***read***, ***update***, ***commit*** and ***abort*** operations. It would also be necessary to designate certain processes as active objects (TM's) and others as data objects (DM's); the later would maintain persistent states. The detailed formalization of such model would be completely standard (except for the constraint linking failures and commit events, and the requirement that the serialization order respect causality). For brevity, we omit a more elaborate treatment of this topic.

[10] We conjecture that any distributed system with non-trivial distributed consistency properties, for a broad definition of "consistency", requires a single system view of membership. However, this is an important theoretical question that will require further study.

Elaboration of the model

If our objective in this paper were primarily formal, it would be desirable to flesh out the model with theorems proving properties of the model, presenting algorithms that successfully implement the model, and giving examples of histories, extended histories, and legal histories. However, the goal of this paper was really to motivate the need for the proposed model, and the model itself is based closely on well known models that have appeared elsewhere in the literature. Accordingly, we defer these issues to a future paper.

Appendix B: Practical issues

Transaction identifier representation and piggybacking

If we wish to allow calls from transactional software into non-transactional software, the system must adopt a standard representation of transaction identifiers and must piggyback these on messages. Fortunately, industry has begun to develop standards in this area, as part of the XOPEN/XA architecture. We therefore see this as a problem likely to be solved in a standard manner by future operating systems.

Integration of Failure and Commit mechanisms

The obvious way to implement restriction (3) of the failure model is to treat the delivery of the reply as part of an atomic action controlled by the final commit/abort decision of the multiphase commit protocol. With this done, it suffices to include k representatives of the system membership service in any commit protocol, if we wish to be safe and able to commit even if up to $k-1$ failures occur in the failure detection service. The result is that reception of a reply implies that the commit also occurred, and that it is impossible to commit a transaction that is not also "live" in the sense that someone is waiting for the reply (here, we assume that a process invoking a job step waits for a reply until the reply arrives or a failure notification occurs).

Linearizability and Serializability

This issue arises when an object has internal concurrency and an internal concurrency control algorithm. Basically, one either arranges to convey causal dependency data into the concurrency control algorithm (i.e. something like the Psync primitives), or to delay invocations so that the only concurrently active invocations are in fact concurrent in the virtual synchrony layer (in the limit, this might imply serializing invocations, if the concurrency control mechanism is oblivious to the higher level and all invocations are causally related to one another).

Compatibility checking

This issue arises if it is not possible to piggyback transaction identifiers in some situations or if the other steps outlined above are not fully implemented. The consequence of such a partial system implementation is that certain combinations of system components might not work.

In this case, we recommend that object invocations be identified as transactional or non-transactional. Invocations from non-transactional subsystems into transactional ones can be permitted provided that the transactional concurrency control scheme employs a serialization ordering that extends the partial event ordering arising from the causality relation and the *abcast* delivery ordering, or that concurrency is limited as outlined above.

Invocations from transactional subsystems into non-transactional ones will be flagged at compile time and must be explicitly validated by the application designer.

Pragmatic considerations

Having had the experience of building Isis and porting it to a wide variety of UNIX platforms, the author has become realistic about the difficulty of changing software provided by other sources. It is therefore useful to note that none of the remarks above require any significant changes to pre-existing software, although in some cases changes could lead to better performance (for example, a pre-existing database concurrency control mechanism might not respect causality; were this a possibility, one could simply present the database with sequential invocations, but at a loss of concurrency). A positive feature of our proposed model is that it could be implemented and used in experiments without requiring some sort of industry-wide consensus, as seems to be necessary for even the most insignificant innovations in other areas of operating system research.

References

[ABHN91] Mustaque Ahamad, James Burns, Phillip Hutto, and Gil Neiher. Causal memory. Technical report, College of Computing, Georgia Institute of Technology, Atlanta, GA, July 1991.

[Bak89] M.G. et. al Baker. Measurements of a distributed file system. In *Proceedings of the Twelveth ACM Symposium on Operating Systems Principles*, pages 198–212, Litchfield Park, Arizona, November 1989. ACM SIGOPS.

[BC90] Ken Birman and Robert Cooper. The ISIS project: Real experience with a fault tolerant programming system. European SIGOPS Workshop, September 1990. To appear in Operating Systems Review, April 1991; also available as Cornell University Computer Science Department Technical Report TR90-1138.

[BC91] K. Birman and T. Clark. A fault-tolerant resource manager for computer networks. Technical report, Cornell University Computer Science Department, Ithaca, NY, December 1991. forthcoming.

[BCG91] Kenneth A. Birman, Robert Cooper, and Barry Gleeson. Programming with process groups: Group and multicast semantics. Technical report, Cornell University Computer Science Department, February 1991. Technical Report TR91-1185.

[BGH87] Philip A. Bernstein, Nathan Goodman, and Vassos Hadzilacos. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[BJ87a] Kenneth P. Birman and Thomas A. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 123–138, Austin, Texas, November 1987. ACM SIGOPS.

[BJ87b] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[BSS91] Kenneth Birman, Andre Schiper, and Patrick Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3), August 1991.

[CR91] P.K. Chrysanthis and K. Ramamritham. Acta: The saga continues, in *transaction models for advanced applications*. Morgan Kaufmann, 1991.

[Spe85] A. Spector et. al. Distributed transactions for reliable systems. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 127–146, Orcas Island, Washington, December 1985. ACM SIGOPS.

[Ous85] J. K. Ousterhout et. al. A trace-driven analysis fo the 4.2 BSD UNIX file system. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 15–24, Orcas Island, Washington, December 1985. ACM SIGOPS.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Patterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[Gra78] James Gray. Notes on database operating systems. In R. Bayer, R. M. Graham, and G. Seegmuller, editors, *Operating Systems: An Advanced Course*, volume 66 of *Lecture Notes on Computer Science*. Springer-Verlag, 1978. Also appears as IBM Technical report RJ2188.

[Hoa74] C. A. R. Hoare. Monitors: An operating systems structuring construct. *Communications of the ACM*, 17(10):549–557, October 1974.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness conditrion for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[KS89] J. Kistler and M. Satyanarayanann. Disconnected operation in the coda file system (preliminary version). In *Proceedings of the Twelveth ACM Symposium on Operating Systems Principles*, pages 213–225, Litchfield Park, Arizona, November 1989. ACM SIGOPS.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[LCJS87] B. Liskov, D. Curtis, P. Johnson, and R. Scheifler. Implementation of Argus. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 111–122, Austin, Texas, November 1987. ACM SIGOPS.

[LLS90] Rivka Ladin, Barbara Liskov, and Liuba Shrira. Lazy replication: Exploting the semantics of distributed services. In *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing*, pages 43–58, Qeubec City, Quebec, August 1990. ACM SIGOPS-SIGACT.

[LS83] Barbara Liskov and R. Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

[LW86] Barbara Liskov and William Weihl. Specifications of distributed programs. *Distributed Computing*, 1:102–118, 1986.

[Mos82] J. E. Moss. Nested transactions and reliable distributed computing. In *Proceedings of Second Symposium on Reliability in Distributed Software and Database Sysytems*, pages 33–39, 1982.

[PBS89] Larry L. Peterson, Nick C. Bucholz, and Richard Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3):217–246, August 1989.

[RB91] Aleta Ricciardi and Kenneth Birman. Using process groups to implement failure detection in asynchronous environments. In *Proceedings of the Eleventh ACM Symposium on Principles of Distributed Computing*, Montreal, Quebec, August 1991. ACM SIGOPS-SIGACT.

[Sch86] Fred B. Schneider. The state machine approach: a tutorial. Technical Report TR 86-800, Department of Computer Science, Cornell University, December 1986. Revised June 1987.

[SG88] D. Shasha and N. Goodman. Concurrent search structure algorithms. *ACM Transactions on Database Systems*, 13(1):53–90, March 1988.

[Stu86] H. Sturgis. Remarks during an invited colloquium. Technical report, Dept. of Computer Science, Cornell University, 1986.

[SW89] F. Schmuck and J. Wyllie. Experience with transactions in quicksilver. In *Proceedings of the Twelveth ACM Symposium on Operating Systems Principles*, pages 239–253, Litchfield Park, Arizona, November 1989. ACM SIGOPS.

[Ull80] J. Ullman. *Principles of Database Systems.* Computer Science Press, Rockville, MD, 1980.

[Wei89] W. Weihl. ?? transactions ?? In Sape Mullender, editor, *Distributed Systems*, pages 319–368, New York, 1989. ACM Press, Addison-Wesley.

[X/O91] X/Open. X/open distributed transaction processing: the xa specification. Technical Report Technical report, X/Open, 1991.