The Role of Order in Distributed Programs*

Kenneth Birman
Keith Marzullo

TR 89-1001
May 1989

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

*This work was supported by the Defense Advanced Research Projects Agency (DoD) under ARPA order 6037. Contract N00140-87-C-8904, and also by a grant from the Siemens Corporation. The views, opinions, and findings in this report are those of the authors and should not be construed as an official Department of Defense position, policy, or decision.

The Role of Order in Distributed Programs

Kenneth Birman
Keith Marzullo

Abstract

We discuss the role of order in building distributed systems. It is our belief that a "principle of event ordering" underlies the wide range of operating systems mechanisms that have been put forward for building robust distributed software. Stated concisely, this principle is that one achieves correct distributed behavior by ordering classes of distributed events that conflict with one another. By focusing on order, one can obtain simplified descriptions and convincingly correct solutions to problems that might otherwise have looked extremely complex. Moreover, we observe that there are a limited number of ways to obtain order, and that the choice made impacts greatly on performance.

1 Introduction

Researchers have proposed a variety of mechanisms for building distributed software within which component programs cooperate to perform tasks concurrently, maintain replicated data, and respond to failures or recoveries by dynamically reconfiguring. These mechanisms typically provide guarantees of "consistent" (correct) behavior, but the precise meaning of *consistency* and the methods by which consistency is achieved differ widely. This makes it difficult to compare the different methods with one another.

This paper is based on the premise that most forms of distributed consistency can be achieved by order generating and preserving mechanisms. While it is not surprising that consistency should be closely related to ordering, we believe that the fundamental nature of this relationship has not been widely appreciated. Here, we show that the manner in which order is generated and preserved has significant performance implications, and observe that dissimilar high-level abstractions are often implemented using surprisingly similar ordering mechanisms. Abstracted from any particular system, such ordering mechanisms may provide a suitable base for building distributed operating systems and programming languages.

2 Relating consistency to order

2.1 Reasoning about consistency

To establish the "correctness" of a distributed system one specifies what we will call a *distributed consistency property* and shows that it is maintained during execution. Such a property takes the form of a predicate on the states of system components. Because the states of components (and perhaps the predicate itself) may evolve during computation, one also gives a rule telling *when* it should be satisfied.

This raises two issues. The first concerns the meaning of *time* in a non-realtime distributed system. Lamport has observed that for such systems, any mechanism capable of ordering events, giving a way to label them and providing a way to compare labels can play the role of time [Lam78]. In fact, although it is common to loosely treat time as synonymous with realtime, there is imprecision in the degree to which realtime clocks can be synchronized in a network. Lamport argues that temporal algorithms for asynchronous distributed systems should be based on mechanisms such as *logical clocks*, which enable a program to determine the order in which events occurred, without requiring clock synchronization. In the case of consistency predicates for asynchronous settings, it follows that a rule telling "when" to evaluate a predicate should be expressed in terms of which events occur before the predicate is examined and which occur afterwards.

A second issue relates to "inconsistency" that has no operational consequences. If a system is consistent in all *observable* states, we would argue that it behaves correctly – even if it passes through *unobservable* states that violate consistency. A system can have many sorts of "observers". The states on which operations are executed are *internally* observable. This leads us to require that the distributed consistency property hold on the process states a distributed operation reads or modifies. For a system that takes *external* actions, the external world is an observer of system state – and hence consistency should hold in situations where such an observer might be able to detect violations.

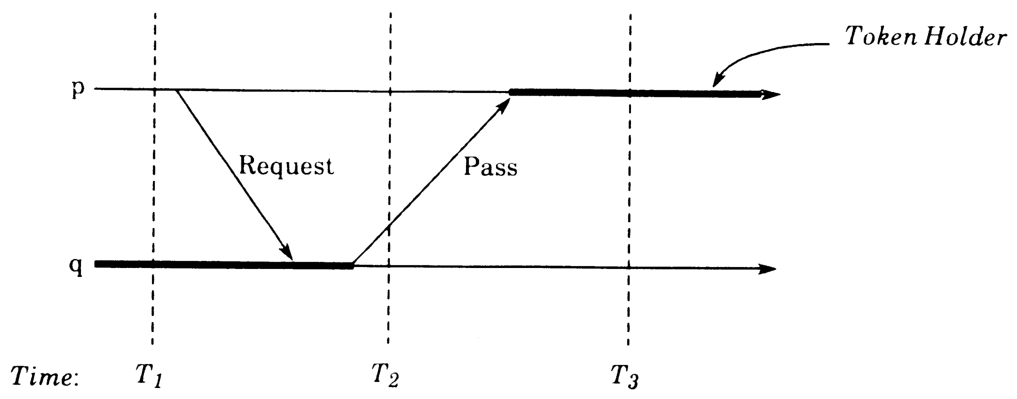

Figure 1: Execution of a token-passing algorithm

2.2 A token passing example

How do these issues enter practical problems? Consider the problems of *mutual exclusion* and *resource management.* A simple mutual exclusion scheme might support two operations: ACQUIRE and RELEASE. A distributed system can implement mutual exclusion with token passing, employing REQUEST and PASS operations for token transfer.[1] A correct execution will satisfy many consistency properties: that there is at most one holder of mutual exclusion at any time, this holder previously did an ACQUIRE, and so forth. Concerning the token, we would require that the process holding mutual exclusion also holds the token, that there is always exactly one token holding process, etc.

Now, consider the execution in Fig.1, where q initially holds the token and passes it to p in response to a request. There are many ways to examine the state of this system. If we look at p at time T_1 and q at time T_3, we would find two tokens, and if we looked at both at time T_2 we would find none. Obviously, the former "state" is observed in an unreasonable way: one should examine system states at the same time. On the other hand, in the absence of precisely synchronized realtime clocks, there is no way to take a simultaneous snapshot: times T_1 and T_2 may well be indistinguishable to processes p and q. Now consider the latter system state. This seems to expose a weakness in our consistency rule, which fails to address the case where the token is in transit when we look at the system. Yet, it is

[1]In systems that support lightweight tasks, a process holding the token could use any correct scheme for "local" mutual exclusion.

reasonable to claim that unless such a state is observable the consistency rule shouldn't need to cover it.

One solution to this class of problems is to examine only *consistent cuts* [CL85]. A consistent cut is any sampling of process states such that if q is examined after receipt of message m from p, then p must be examined subsequent to sending m. (A consistent *snapshot* is a consistent cut in which channel contents are also recorded). Consistent cuts eliminate the first of the above problems, but not the second: we would still need to extend the consistency rule to allow the case where the token is in transit (and hence there is no holder).

2.3 A resource management example

This specific problem of messages in transit is symptomatic of a larger class of problems where a complex operation violates consistency while it is in progress. Consider a set of *resource manager* processes that cooperate to control a set of resources accessed by *application* processes. Overloaded managers can transfer responsibility for a resource by sending a TRANSFER message that informs all processes in the system of the new manager. Until the transfer occurs, a manager continues to handle new service requests; after the transfer it rejects requests, which the client retransmits to the new manager. The consistency rule could be that each resource is managed by a unique process at any given time, and that during a period when a resource is transferred t times, no request ever needs to be retried more than $t + 1$ times.

This system can be easily built using multicast primitives to implement the transfers, such as the ones the ISIS system supports [BJ87b,BJ87a]. Requests can then be performed using conventional RPC. However, such an implementation will violate the consistency predicate along many possible consistent cuts. Consider the multicast message corresponding to a TRANSFER. This cannot be delivered instantaneously, hence there will be cuts in which some processes have received the message and some have not. Worse, some multicast protocols transfer messages to their destinations in a first phase but delay delivery until some other event takes place. For such a protocol, there will be consistent cuts in which all messages have arrived, but some have not been delivered. Short of devising a very complicated consistency predicate or modifying the consistent cut algorithm to be knowledgeable about the way the protocol works, the only thing one can say is that consistency will hold for those cuts reflecting all-or-nothing message delivery. The same issue would have arisen in the token passing problem if tokens were passed using a multi-phase commit protocol.

2.4 A conjecture about consistency and ordering

Is there a way to formulate an *arbitrary* problem so that it is clear when a consistency property should hold? A consistency property is a predicate that refer to a distributed state. Complex operations, such as the TRANSFER operation described above also refers to a distributed state. Clearly, some precondition over the distributed state should hold prior to doing these sorts of operations too. Let us refer to distributed operations like these as *meta–operations*.

We conjecture that:

> *Any system concerned with maintaining consistency must impose and respect ordering when meta-operations conflict.*

Two issues arise. One is how to tell when meta-events conflict and need to be ordered. This question is highly dependent on semantics, and lies outside the scope of this paper. The second concerns how to impose and respect ordering when necessary.

Our conjecture sounds like a serializability constraint on meta–operations. However, notice that that except for its use of ordering, the problem has little in common with transactions that read and write a shared database using concurrency control and multi-phase commit protocols, which is the traditional context for serializability arguments. Nor are we trying arguing for totally ordered multicast protocols or the ISIS *virtual synchrony* property, although this paper is certainly motivated by the latter. Our goal is to understand when one needs to pay for the ordering that protocols such as these provide; to use them without regard for need would be a costly proposition. Also, multicasting arises in only a subset of distributed systems. By abstracting away from message passing *per–se*, one arrives at more general results. Moreover, multicast based systems typically treat each multicast as a separate event. It is unclear that results derived for such a setting apply when such tactics as piggybacking one message on top of another are permissible.

Let us summarize our observations:

1. One achieves consistency in a distributed system by ordering events that conflict with one another.

2. These events involve sets of operations that occur in multiple processes but are nonetheless related.

3. The type of conflict-based ordering needed to ensure consistency is more primitive than transactional serializability or virtual synchrony.

The remainder of this paper starts by making the ideas of meta-events and meta-event orderings more precise, asking how meta-event orderings come about, and exploring the cost implications that ordering may have at the application level. Then, we look at how order is used in a variety of distributed computing systems. The paper concludes by asking what the implications of a principle of distributed ordering might be for future distributed systems design efforts.

It is not our goal here to provide a formally rigorous treatment of meta-events and order. We will also only touch on issues of fault-tolerance and realtime.

3 Meta-event orderings

In this section, we introduce an event-ordering model that includes meta-events and discuss what it means for two meta-events to be ordered.

3.1 Events and Event Orderings

We will start with the standard partial-order representation of a distributed system. At the most primitive level of a distributed system, only certain types of events can be said to be "ordered". Consider a system of processes that communicate using messages.[2] Processes execute *operations*, which are indivisible units of work: computation, sending a message, or receiving a message. We use the term *event* to denote any of these activities.

Event orderings come about when an operation is influenced by (i.e. reads from) some prior operation, or when the logic of the program delays the start of one operation until another has finished, or when a message sent by one process is received in another. It follows that if one stops the execution of the system at some instant in time, the execution up to that point can be described by a tuple $(P,E,\rightarrow)$, where P is a set of processes, E is a set of events, and $\rightarrow$ gives the order in which the events occurred.

[2]The ability to share memory between processes doesn't change things in a fundamental way, but it would introduce complexity.

The partial order $\rightarrow$ is actually defined as the transitive closure of two more primitive partial orders:

1. The *internal ordering on operations*, defined on a per-process basis. For events a and b occurring in some process p, $a \rightarrow b$ denotes that a read from b, or was constrained by the logic of the program to execute after b.

2. The *communication ordering*, defined on a per-message basis. For message m sent from process p to q, $snd_p(m) \rightarrow rcv_q(m)$.

3.2 Meta events and meta orderings

Earlier, we discussed the idea of a meta-operation, the execution of which gives rise to a *meta-event.* A meta-event is a set of logically related events at multiple processes. Examples of meta-events include the delivery of a multicast message to some group of destination processes, the creation of a snapshot of the distributed state of a system, the detection of a process failure by the processes that survived, or a transaction on a database.

We model a meta-event by a tuple (i, M) where i is an initiating event and M is a set of events satisfying $\forall m \in M : i \rightarrow m$. We will say that two meta-events are ordered if their event sets are ordered:

$$(i, M) \rightarrow (j, N) \text{ iff } \forall m \in M, n \in N : m \rightarrow n.$$

No statement is made about the ordering of the initiation events; we want to allow the case where two meta-events are started concurrently but ultimately give ordered outcomes.

Notice that a meta-event need not correspond to an invocation of a multicast or some other communication protocol. That is, we do not require here that the meta-event be a discrete activity separable from the rest of the system's execution. The communication that links an initiating event to the outcome events could be hidden in any of the mechanisms by which information is transmissible within a distributed system.

It may appear that failure detection by timeout and the detection of "external" events through sensors give rise to meta-events that lack a single initiating event. However, if multiple processes try to do these things in parallel, they may not all observe the same outcome. For example, one process might see an overloaded process timeout, while other processes believe that it remained operational. If internal consistency is needed, a software agreement protocol would have to be executed. This converts the physical timeout events to logical ones, which fit our rule.

3.3 Origins of meta-order

Imagine a distributed system in which process p wishes to initiate meta-operation (i, M). Say that this operation should be ordered with respect to certain types of meta-events. The question is how p can achieve this ordering and how it is conveyed to the processes executing the events $m \in M$. There are two static cases and one dynamic case:

Computational ordering: The first static case is when the meta-events that should be ordered are initiated within a single computation. That is, p "knows" about previously initiated meta-events which should terminate first at any destinations where they overlap with M. Say that (i', M') is such a prior meta-event. Then it must be the case that $i' \rightarrow i$, because p could not otherwise know about (i', M'). Since $\rightarrow$ is transitive, $\forall m \in M, i' \rightarrow m$. In other words, there is a way to pass information about events prior to M to the places where events in M will occur. As we will see below, this can be exploited in different ways, but the essential point is that the system has the ordering information it needs and has a way to get it to where it will be needed. No additional messages are needed, although some messages will to be larger since they need to carry some representation of this information to the places where it will be used.

Locking schemes also fit into the computational paradigm. In these, the "lock manager" delays granting a lock until after it has been released by prior holders, establishing a causal relationship in which the new holder's actions occur after the previous holder's release.

A priori ordering: The second static case occurs when p knows about some event that will take place elsewhere, but was not initiated prior to i. That is, the semantics of the operations include the requirement that $\forall m \in M, m' \in M', m \rightarrow m'$. For example, p's operation may be in response to an earlier operation that concurrently started an operation on q, and we want the result of q's operation to be ordered before that of p's. Again, p can pass whatever information it has to the places where the events $m \in M$ will take place. Presumably, the processes that receive this information can wait if necessary. (Otherwise, p will need to wait until q's event has occurred before initiating (i, M), but this takes us back to the computational case).

Dynamic ordering: The third case is the hardest. Here, p does not know if other meta-events might be initiated elsewhere, concurrent with (i, M). If some other process has initiated a meta-event that conflicts with (i, M), the two events should be ordered, although the order will not be known *a priori*. On the other hand, if no "conflicting" event is present, (i, M) should simply be allowed to occur. In the dynamic case, additional information is needed before the events in M can take place. As we will see below, the cost of this information (the communication needed to obtain it) is significantly higher than in the two static cases.

4 The cost of dynamic ordering

How much does dynamic order cost? This problem is well known in distributed systems.

An extreme case of dynamic ordering arises in algorithms for computing a consistent snapshot. An example is the method of Chandy and Lamport [CL85], which flushes messages from the communication channels using marker messages; the channels are assumed FIFO, hence messages sent prior to the snapshot are received prior to the markers. The cost is high: a marker message flows in each direction between each pair of processes. Moreover, the only meta-events in the basic algorithm are those related to forming snapshots. That is, the snapshots are ordered relative to base events, but might cut through other types of meta-events used in the application. The cost of forming a snapshot ordered relative to other sorts of meta-events could be even higher.

Dynamic ordering also arises in multicast protocols that provide ordered message delivery and in protocols for distributed mutual exclusion. Here, we will focus on the multicast case. Ignoring failures, a multicast can be modeled as a meta-event (i, M) in which i denotes the initiation of the multicast and M the set of message delivery events. Existing atomic multicast protocols achieve delivery ordering in one of two ways, and mutual exclusion protocols can generally be classified into the same two categories.

One approach is to implement a rule by which for each pair of concurrent multicasts, some uniquely identified process must pick their ordering. For example, this can been done using token passing (the token holder picks) [CM87], or with a tree-structured scheme (the least common ancestor picks) [GMS88]. In sufficiently static settings, this can also be done with a hashing scheme [CG86].

The other approach is decentralized. In this scheme, the processes vote on the ordering they prefer, and any process with the full set of votes can deduce the ordering rule to use. A two-phase protocol based on this approach is described in [BJ87b]. The "state machine" approach is similar; it employs a fully decentralized, fault-tolerant protocol to achieve ordering [Sch86].

There are ordered broadcast protocols that utilize external sources of order. We would classify these as instances of the first case, where the order generating "process" is implemented in hardware, or hidden within a clock synchronization algorithm. For example, the Linda S/Net implementation exploited broadcast communication hardware for this purpose [CG86]. Likewise, the Delta-T protocols operate using synchronized clocks and realtime timestamps; they delay message delivery long enough to compensate for uncertainties in clock synchronization and transmission time [CAS86].

There are also ordered broadcast protocols that combine some of the work done for protocol invocation t with that for invocation $t+1$; for example, the ISIS failure detection protocol [BJ87b]. Such a strategy may reduce the overall cost of the protocol, but does not constitute a fundamentally new way of obtaining ordering.

Notice that dynamic order comes at a high price. In addition to the extra communication required relative to the two static cases, dynamic ordering requires that events be delayed, and this latency can directly impact the application program. That is, there is always some process which prior to some point in its execution will not be able to"safely" perform certain actions. It must wait to obtain additional information, and this latency limits the rate at which the application can make progress.

Thus, the choice of ordering method can have important practical performance implications. We will see an example of this in section 5.

5 Two token passing implementations

A brief example will illustrate how ordering issues can enter into a higher-level algorithm. Our goal is to implement the token passing part of the mutual exclusion scheme described above using a static set of ISIS processes (the solution can easily be generalized to a dynamic set, but we will not do so here). The problem was first solved by Schmuck [Sch88]; the treatment given here follows one in [BJ89].

We need a some detail about two multicast primitives supported by ISIS: CBCAST and ABCAST.

- CBCAST ensures that if there are two CBCASTs satisfying $i \rightarrow i'$, then delivery order matches the invocation order: $(i, M) \rightarrow (i', M')$.
- ABCAST extends CBCAST by also ordering concurrent invocations, picking an order to use in the concurrent case.

5.1 ABCAST solution

Using ABCAST we can implement a very simple token passing algorithm (Fig. 2). All operations (PASS and REQUEST with no parameters) are multicast to the entire set of processes. Each process maintains a list of pending operations. All REQUEST operations are granted in a deterministic order, and a REQUEST is granted when a PASS is received. Since all see the same operations in the same order, behavior is identical.

5.2 CBCAST solution

An alternative token passing scheme multicasts operations using CBCAST (Fig. 3. A consequence is that processes may receive requests in different orders, and that the request lists at two different processes may not contain the same requests when a given PASS operation is received. In this implementation, the process doing a PASS operation first picks the request that it will grant (e.g. the first one on its list of pending requests), and includes this information as part of the multicast (delaying the PASS in the case where there are no pending requests). Processes receiving the pass operation must look up the granted request on their list of pending requests (it is easy to show that they will find it there) and delete it. This algorithm is proved correct in [BJ89], using the observation that REQUEST and PASS operations are totally ordered by $\rightarrow$ along the path that the token follows.

5.3 Use of order in the two algorithms

How would one pick between these two solutions to the problem? One is easier to understand than the other, but the the difference in performance far outweighs any difference in complexity. The ABCAST algorithm is "tightly synchronous": all processes move in lock-step, and computation advances slowly because of the costs intrinsic to ABCAST. The CBCAST version is much faster: all the multicasts can be done asynchronously, in which case computation will be limited only by processor speed and the capacity of the operating system to buffer multicast requests and perform them in the background. Using ISIS, these algorithms can be compared experimentally: for 5 processes running on SUN 3/60 hardware, the performance difference exceeds a factor of 10, and this grows with the number of processes.

From the perspective of ordering, the algorithms differ in the way that they obtain the ordering needed to maintain consistency. The former ignores the invocation order of the operations. It acts as if all operations potentially conflict with one another, and resolves this by generating a strong, globally observed ordering that it uses to control execution. The CBCAST version is more cautious in its use of available ordering. By having the process that is about to do a PASS decide what request to grant, consistent distributed behavior is achieved without ever resorting to a costly ABCAST, and the ordering problem is reduced to the computational case.

We believe this example is demonstrative of the general problem. When we build distributed systems without attention to the amount of ordering needed to achieve consistency, and the ways that ordering can be preserved, we can find ourselves using distributed programs as inefficient and "inelegant" as the ABCAST token passing algorithm.

6 Completeness of the model.

Could there not exist many levels of ordering, like the ones that CBCAST and ABCAST provide, but differing in the precise rule that they implement? In the case of multicasts, one can prove that these two types of ordering are *complete* (in the sense that these can implement any other ordered primitive) within the problem classes that they solve. Schmuck does this for CBCAST [Sch88], and Schneider discusses work on the *state machine* approach which includes type of ordering achieved by ABCAST [Sch86]. The same results can be expected to hold in the case of meta-event orderings.

This is not to say that more complex forms of ordering are not meaningful. In fact, if one moves to systems that require ordering on *sets* of meta-events, more complex algorithms are definitely needed. For example, the ISIS system implements a third multicasting primitive, GBCAST, which it uses for process group membership changes. GBCAST provides ordering with respect to more than one class of meta-events, and requires a more costly 3-phase protocol.

An interesting direction to pursue would be a theory about the composition of meta–operations and meta–orders. This is discussed more in the conclusions of this paper.

7 Higher level consistency and ordering.

So far, our examples have been low-level ones. Most distributed computing mechanisms, in contrast, exist at a very high level: reliable multicasts, transactions and 2-phase commit, the ISIS virtually synchronous toolkit, virtual time, the Linda tuple-space, the Psync primitives [Pet87], and so forth. Our task in this section will be to bridge the gap. In so doing, the relationships between these systems will become more clear.

7.1 Who is currently operational?

Agreement on failure is perhaps the most fundamental problem in a distributed system. Even the simplest real distributed systems must address this issue. For example, in a system with two processors connected by a single communication link, usually two kinds of failures are assumed: processors may crash, and the link may suffer from a performance failure (*i.e.* it may delay or drop a message). These two kinds of failures are indistinguishable by a sending process, yet the results of an inconsistent decision can be disastrous [Gra79]. As a result, several existing systems structure their state such that the recovery from both kinds of failures are identical. For example, in the Sun NFS protocol [SUN86] a connection holds very little state, so a crash can be treated as a particular kind of performance failure. Many other protocols "time out", treating performance failures as crash failures.

When more than two processes are involved, substantial additional complexity arises. For example, transactional concurrency control mechanisms impose order on read and write operations so that the execution of a set of concurrent transactions is equivalent to a serial execution. The best known such mechanism for managing replicating data is the *available copies* algorithm. It has been shown that if all processes agree on when a process fails, serializability is maintained. If they do not agree, then a transaction may not see a the effects of a conflicting operation from a virtually earlier transaction [BHG87]. Thus, transactional concurrency control mechanisms must order failure events relative to the execution of other operations – specifically, commit operations.

The ambiguity of crash and performance failures makes agreement on who is operational difficult. If the effect of executing an operation can be undone, as an abort of a transaction, then a good way to deal with apparent failures is simply to cause an abort. This is simple and conservative: if the failure was not real, the only cost is that the transaction must be redone. On the other hand, abort is not always meaningful; for example, in systems that take external actions. Lacking this alternative, the operational processes must instead agree on which processes have crashed, forcing those processes to crash or rejoin the system if it later turns out that the problem was a performance failure.

The protocols for agreeing on who is operational are clearly an important part of any distributed system. Moreover, such protocols bear a strong relationship to the ordering and consistency preserving mechanisms discussed above.

To see this, consider an application that makes use of a list of the operational processes. At some point in an execution, process p goes from being operational to having failed. An application that depends on this information may dynamically adapt itself to the failure of p, and it then becomes important that events initiated after the failure only encounter processes at which the failure is already known. For example, if q sends s a message that relates to the failure of p, inconsistency could easily arise if s receives the message prior to observing the failure. Any messages ordered prior to the observation of the failure of p must be flushed from the channels, effectively forming a snapshot. A protocol capable of achieving this handling of process failures necessarily creates meta-order.

It is not surprising to find that the ISIS system solves this problem using a multiphase consensus protocol that terminates in two phases after the last failure [BJ87b]. During the last phase, this protocol does a flush, much like the transmission of channel markers that occurs in the Chandy-Lamport consistent snapshot algorithm [CL85]. The ISIS protocol can thus be understood as a mechanism for drawing a line (cut) across the system execution: events prior to the cut have not observed the failure, all processes observe the failure "simultaneously" along the cut, and events after the cut all reflect the failure event. In other words, the ISIS solution works by establishing meta-ordering. Similar mechanisms appear in other systems ([CM87], for example). We would argue that while these protocols are necessary, they have for the most part not been well presented and understood. This seems to be a problem for which an order-based treatment could lead to significant simplification. For example, Cristian's solution to the membership problem, in [Cri88], is notable for a specification that uses ordering properties and for the simplicity of the algorithms proposed.

7.2 Creating and Preserving Order

According to the thesis of this paper, consistency-preserving distributed mechanisms should fall into two categories: those where the meta–operations are ordered non-dynamically, and those where the meta–operations are ordered dynamically. In fact, it was this observation that led us to investigate ways meta–operations can be ordered.

An example of the use of *a priori* ordering arises in timestamp-based concurrency control algorithms [BHG87]. When created, a transaction is assigned a *timestamp* from a total order, and the concurrency control algorithm can delay or abort a transaction if it attempts to access a variable in an inconsistent order. Liskov and Radkin use a similar method in their work on highly available servers [LL86]. A 2-phase locking algorithm, in contrast, works by dynamically ordering meta–operations. Any transaction is ordered either before the lock point of another transaction, or after its commit point. The meta–operations in this case are the read, write and commit operations, and perhaps the write-lock operation if locks are replicated.

Quorum-based replicated data algorithms are an example where one can see the relationship between what we have called dynamic order and computational order. Viewed externally, a quorum write is clearly a case of dynamic order, generated using a decentralized scheme. Now, consider the same operation at a microscopic level. Typically, the write will require two phases: a first phase during which the value to be written is distributed and the version number to be used is computed, followed by a second phase during which the update is committed provided that a quorum of responses was received and aborted otherwise. At this level of abstraction, the first phase determines an order using a decentralized rule, and the second phase respects that order. That is, the second phase is computationally ordered in the sense discussed above.

Given an order, there are many distributed mechanisms for preserving it. The pessimistic schemes preserve order at all times. In the case of transactional systems, a pessimistic way to preserve order is to force an operation to wait until the operations prior to it complete, as for locking. Above, we mentioned the ISIS CBCAST primitive. CBCAST maintains ordering information by augmenting the data sent with a message. When a message m is sent to a site, copies of any messages m' that m is causally dependent upon are included. (Of course, if the site has already received m', it need not be sent a second time).

Another example of such a mechanism is Psync [Pet87]. With this mechanism, the dependencies among messages are available to the programmer. The order information can be carried as unique message identifiers rather than by forwarding the whole message as done in CBCAST. It is interesting to note that the architects of Psync chose their model in order to unify other communication mechanisms. They reasoned that it would be relatively cheap to implement Psync, and then hopefully build other ordering mechanisms cheaply on top of Psync.

The optimistic schemes for preserving order depend on a mechanism for detecting order violations and rolling back. We would classify timestamped concurrency control and Jefferson's work on virtual time into this category.

7.3 Order in Realtime Systems

A realtime system is one in which a set of computer processes interact with a set of physical processes. In general, the meta–operations of the computer processes must be ordered with respect to external actions in the physical processes. Since a computer process can not in general delay a physical process, the order must in some part be generated by the physical process. This order is most easily represented as a total order of events with respect to some monotonically increasing physical variable. The most obvious candidate is the real time, but any such physical variable can be used.

Little of what we discussed above can be applied directly to realtime systems. For example, the CBCAST solution to the token passing problem gains a substantial performance improvement by substituting a form of logical ordering for the total ordering provided by the ABCAST protocol. This logical ordering bears no relationship to realtime, and hence the mechanism as presented above is inappropriate for use in a realtime system. The ABCAST solution, on the order hand, could be adapted fairly easily to a realtime setting (in fact, one could substitute Cristian's Delta-T atomic broadcast and use the algorithm without additional changes). Yet, some realtime systems place demanding performance requirements on the protocols they use, and the performance advantages of the CBCAST solution in the asynchronous case suggest that there might also be benefits to using it in the realtime case. The question that this raises, but which we will leave open here, is whether there might exist some modified version of CBCAST that could be used to similar advantage in realtime environments.

8 Conclusions

Our field has always searched for principles to guide the development of operating systems and distributed systems. We believe that the principle of distributed ordering meets this criteria, namely that distributed systems achieve consistency through consistent distributed orderings of conflicting events. Insights into the fundamental properties of order-based algorithms would impact a wide range of distributed and parallel systems.

Distributed computing has long been characterized by intense interest in performance and robustness. With the increasing focus on closely coupled distributed services, the sorts of consistency issues we raise here are becoming widely relevant. One implication of a principle of distributed ordering is that such services will achieve the maximum performance and robustness only through a careful understanding of their ordering requirements, and through the development of highly refined operating system primitives for satisfying these requirements.

Several directions suggest themselves for future study.

Order-based operating system primitives. An important question relates to how ordering mechanisms should be presented to applications programmers. Current systems offer a range of high level order-based abstractions, such as transactions, quorum replicated data, atomic multicasts, and virtually synchronous process groups. One cannot help but wonder if there is a more primitive abstraction from which these higher level mechanisms could be constructed. Such an abstraction would be particularly useful because it could support a variety of these mechanisms at once, while also addressing the needs of applications that have reason to order other sorts of operations, such as the execution of pieces of code or actions taken in response to external events. For example, it would be possible to support a notion of *ordered distributed operation* that might come close to directly implementing our meta-operations, but in which the operation to perform would be specified totally abstractly. A different approach might focus on order manipulation primitives, like those in Psync [Pet87], but augmented to have a stronger notion of distributed event and to reduce any dependence on message-passing.

Order-based language primitives? The development of convenient language support for transactions has played a major role in making transactional systems easier to use and popular as a distributed computing methodology. It seems natural to ask if we can devise effective language support for representing and manipulating order. For example, it would be useful to explore the possibility of supporting classes of ordered distributed operations in a multiple type inheritance framework.

How much order is needed? The use of order is closely tied to the cost of an application. Systematic tools are needed for for determining how much order an application needs, and perhaps for using ordering as a complexity measure under which different solutions to a problem can be compared.

Theory of order composition. We noted in Section 6 that a theory is needed for describing the manner in which meta-orderings can be composed to obtain higher level orderings. Such a theory could be a valuable tool for reducing complexity and improving correctness in higher level systems. It might also help us to identify limitations on what can be achieved in distributed systems. For example, we noted that the external world may be an observer of consistency. We also observed that there are limits on the degree to which actions can be simultaneous within a distributed system. This limit does not apply to an external observer, hence one could profitably ask what constraints an external consistency requirement places on the implementation of a distributed algorithm. The answer would undoubtably be both deep and of practical value.

9 Acknowledgements

We are grateful to Robert Cooper for his many suggestions about this paper, and to Tommy Joseph, Ajei Gopal, Frank Schmuck, Fred Schneider, Sam Toueg and John Warne, for their insight on this and related topics.

References

[BHG87] Philip Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[BJ87a] Ken Birman and Thomas Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the Proceedings of the*

Eleventh Symposium on Operating System Principles, pages 123–138. ACM SIGOPS, 1987.

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

[BJ89] Ken Birman and Thomas Joseph. Exploiting replication. In *Lecture Notes: Arctic 88 advanced course on distributed systems.* S. Mullendar, *ed.* Addison-Wesley (forthcoming), 1989.

[CAS86] Flaviu Cristian, Houtan Aghili, and Ray Strong. Atomic broadcast: From simple message diffusion to byzantine agreement. Technical Report RJ 5244 (54244), IBM Almaden Research Laboratory, July 1986.

[CG86] Nicholas Carriero and David Gelertner. The S/Net's linda kernel. *ACM Transactions on Computer Systems*, 4(2):110–129, May 1986.

[CL85] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[CM87] J. Chang and M. Maxemchuck. Atomic broadcast. *ACM Transactions on Computer Systems*, 2(3):251–273, August 1987.

[Cri88] Flaviu Cristian. Reaching agreement on processor group membership in synchronous distributed systems. Technical Report RJ 5964 (59426), IBM Almaden Research Center, March 1988.

[GMS88] Hector Garcia-Molina and Anne-Marie Spautser. An ordered reliable broadcast protocol. In *Princeton-University Technical Report*, 1988.

[Gra79] J. N. Gray. *Notes on Database Operating Systems.* Springer-Verlag, Munich, 1979.

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

[LL86] Barbara Liskov and Rivka Ladin. Highly available servers and fault-tolerant garbage collection. In *Proceedings of the Fifth Symposium on the Principles of Distributed Computing*, 1986.

[Pet87] Larry L. Peterson. Preserving context information in an ipc abstraction. In *Proceedings of the 6th symposium on Reliability in Distributed Software and Database Systems*, pages 22–31, March 1987.

[Sch86] Fred B. Schneider. The state machine approach: A tutorial. Technical Report TR 86-600, Cornell University, Dept. of Computer Science, Upson Hall, Ithaca, NY 14853, December 1986.

[Sch88] Frank B. Schmuck. *The use of efficient broadcast protocols in asynchronous distributed systems.* PhD thesis, Cornell University, Department of Computer Science, August 1988.

[SUN86] Sun Microsystems, Inc., 2550 Garcia Ave., Mountain View, CA 94043. *Networking on the Sun Workstation*, revision B edition, February 1986.

```
To initiate a pass() or request():

          ABCAST('optype');

On receiving a pass() or request():

          case(optype) of
             'pass':
                   if(is_empty(request_queue))
                        wants_request = TRUE;
                   else
                        grant(head(request_queue))
             'request':
                   if(wants_request)
                        wants_request = FALSE;  grant(this_request);
                   else
                        append(request_queue, this_request);
          end;
```

Figure 2: ABCAST token-passing algorithm

```
To initiate a pass() or request():

          case(optype) of
            'pass':
                   while(is_empty(request_queue)) wait;
                   CBCAST('pass', qu_head(request_queue));
            'request':
                   CBCAST('request');
          end;


On receiving a pass() or request():

          case(optype) of
             'pass(granted)':
                   grant(dequeue_request(request_queue, granted))
             'request':
                   append(request_queue, this_request);
          end;
```

Figure 3: CBCAST token-passing algorithm