# Understanding the Limitations of Causally and Totally Ordered Communication

*David R. Cheriton* *and* *Dale Skeen*

*Computer Science Dept.*
*Stanford University*
*cheriton@cs.stanford.edu*

*Teknekron Software Systems, Inc.*
*Palo Alto, California*
*skeen@tss.com*

## Abstract

Causally and totally ordered communication support (CATOCS) has been proposed as important to provide as part of the basic building blocks for constructing reliable distributed systems. In this paper, we identify four major limitations to CATOCS, investigate the applicability of CATOCS to several classes of distributed applications in light of these limitations, and the potential impact of these facilities on communication scalability and robustness. From this investigation, we find limited merit and several potential problems in using CATOCS. The fundamental difficulty with the CATOCS is that it attempts to solve state problems at the communication level in violation of the well-known “end-to-end” argument.

## 1 Introduction

Causally and totally ordered communication support (CATOCS) has been proposed as another important facility for constructing reliable distributed systems [2, 3, 13, 22, 24]. Causally-ordered communication ensures that messages are delivered in an order that is consistent with the potential causal dependencies between messages, following the logical clock model of imposing an overall partial ordering on events in a distributed system [16]. Totally ordered communication support goes a step further to ensure that messages are delivered in the same order to all participants. CATOCS implementations to date also provide atomicity of message delivery, ensuring that either all messages are received or none are, at least at processes that do not fail during the interval of interest. CATOCS implementations may also provide failure notification that is causally (or totally) ordered with respect to the message traffic. With these facilities, CATOCS has been advocated as well-suited for solving several important distributed computing problems, including, in particular, data replication.

However, CATOCS can only be justified by identifying a significant class of real applications for which its facilities are sufficient and efficient, ***and*** whose requirements can not be efficiently satisfied using alternative, general-purpose mechanisms. In this paper, we investigate the applicability of CATOCS to several classes of distributed applications with particular focus on examples used in the CATOCS literature. From this investigation, we find limited merit in using CATOCS. The basic difficulty with CATOCS can be best explained in terms of the end-to-end argument [25]. CATOCS is at the communication level, but consistency requirements are typically expressed in terms of the application’s state.

The next section describes in greater detail what we mean by causally and totally ordered communication support (CATOCS). Section 3 describes a number of significant limitations and cost of these techniques in a variety of communication situations. Section 4 considers the applicability of causal and total ordering protocols to a number of classes of distributed applications, including examples from literature illustrating the use of CATOCS protocols. We show that the limitations presented in Section 3 can and do significantly restrict the benefits of using CATOCS in these application areas. Section 5 presents arguments why CATOCS on the communications data transfer path has a significant scaling cost. We conclude that a state-level approach implemented using object-oriented technology provides a more efficient, robust, scalable and easier to specialize solution to the distributed systems problems we have considered.

## 2 Causally and Totally Ordered Communication

In a causally ordered message system, messages are delivered in the order messages are sent, as determined by the happens-before relationship [16] but restricted to message sending and receiving events[1]. For convenience, we extend the happens-before relationship to include messages.

This research is supported in part by DARPA under Contract DABT-63-91-K-001.

**This is a preprint of a paper to appear in the *Proceedings of the 14th Symposium on Operating Systems Principles*, ACM, December 1993, Asheville, North Carolina.**

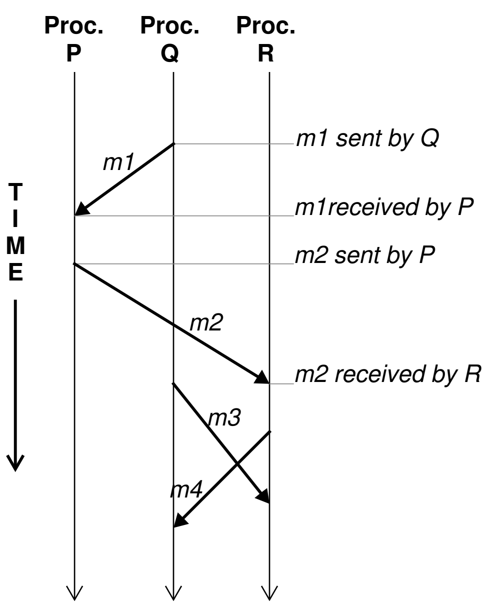
1. This is a natural restriction, given that message events are the only events that the message system knows about.

In the simplest case, message m1 is said to happen-before m2 if there exists a process P such that m1 is sent by or received at P before P sends m2. Taking the transitive closure of these simple cases yields the full happens-before relationship on messages. The term causally precedes is a synonym for happens-before, i.e., m1 causally-precedes m2 if m1 happens-before m2.

Figure 1 illustrates causal ordering using a common charting device known as an *event diagram*. Message sending and receiving events are arranged by column, one per process, with time advancing from top to bottom. The diagram shows that Process *Q* sends *m1*, which is later received by Process *P*. Later, *P* sends *m2*, which is subsequently received by Process *R*, etc. Message *m1* causally precedes both *m2* and *m4*. Messages *m3* and *m4* are said to be "concurrent" because neither causally precedes the other.

*Causal multicast*[2] delivers messages in accordance with the happens-before relationship within a specified group of processes, known as a *process group* [13]. Specifically, if two messages are multicast to the same process group and the sending of one message happens before the sending of another message, then the first message is delivered before the second message at all processes in the group. Hence, *causal multicast* really means "happens-before-preserving."

**Figure 1.** A 3-process event diagram. Messages *m1* "causally precedes" *m2* and *m4*. Messages *m3* and *m4* are "concurrent" and, hence, causally unrelated.




2. In the literature, this is sometimes referred to as a "broadcast" protocol [13], and sometimes as a "multicast" protocol [4]. We prefer the term multicast.

*Totally ordered multicast* delivers **all** messages in the same order to all processes within a process group, even concurrent messages. Message delivery is commonly in accordance with the happens-before relationship, and we make this assumption herein, although some systems do not provide any causal guarantees.

Many systems providing CATOCS also implement *atomic message delivery*, ensuring that a message is delivered to all non-failing processes or none. Without atomic message delivery, the loss of a message at a process could delay indefinitely the receipt of all casually dependent message, effectively dropping those messages. Clearly, loss of a message could transitively force the dropping of an arbitrary number of subsequent messages. Atomic message delivery can be implemented in a relatively straight-forward way by having each process in the group buffer each message it receives until it is sure the message is *stable*, i.e. received by all other members of the group. Therefore, we assume for this discussion that CATOCS systems also provide atomic message delivery. Note, however, that the atomicity property of message delivery does **not** include processes that fail during the execution of a CATOCS multicast. Hence message delivery is atomic, but not *durable*. (An action is *durable* if its changes survive failures and recoveries.) Specifically, if the sender fails during CATOCS protocol execution before the message is stable, there is no guarantee that the remaining operational processes will ever receive and deliver the message [4]. As a special case of this problem, a process can send a message to its process group, receive and act on the message locally (as a member) and then fail, without any other members of the process group receiving the message, potentially leaving its state inconsistent with the other members of the process group. Atomic, durable message delivery within CATOCS appears expensive to provide, is not supported in any implementations we are aware of, and yet is a significant deficiency for using CATOCS for reliable replicated data update.

CATOCS systems may also provide failure notification that is consistently ordered with respect to message receipt at each process in the group. When appropriate, we consider the utility of this ordering in conjunction with the merits of CATOCS in general. Note that ordered failure notification can be provided without CATOCS and is useful as a stand-alone capability [10]. However, it should be stressed that our focus is on understanding the merits of ordered message delivery and, hence, we do not consider the merits of ordered failure notification as a stand-alone capability.

There are systems supporting multicast and process groups but without causal communication support, UDP on IP Multicast [7] and the earlier V-System [5] being two examples. There are also systems that exploit causal relationships and other ordering relationships without incorporating this mechanism into the communication system, Munin [14] being one example. Our concern is with the merits of implementing ordering relationships on delivery of messages from multiple sources within the communication system.

The ordering provided by CATOCS is called *incidental ordering* in this paper, because it is based on the ordering of "incidents" of communication within a process group. This incidental ordering is not necessarily consistent with the *semantic ordering* determined by the information contained in a message. For example, the notifications of changes to a database should be disseminated in the order committed by the database system, but CATOCS only ensures this semantic ordering if the updates are provided to the communication system as "incidents" occurring in the semantic order. Consequently, many systems use or provide what we call *prescriptive ordering* where message delivery order is effectively based on ordering constraints explicitly specified or prescribed by a process at the time it sends a message. As described in the following sections, prescriptive ordering using state-level clocks, temporal or logical, offers an attractive alternative to CATOCS in many real applications.

# 3 Limitations and Costs of CATOCS

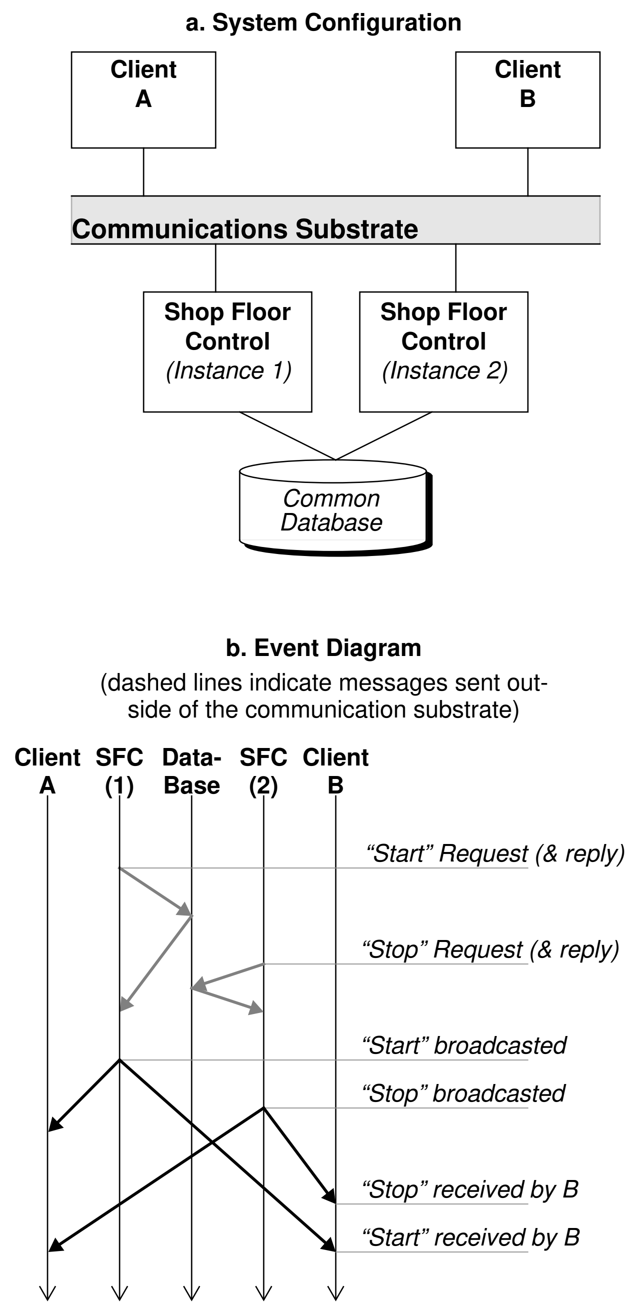
The general problem that arises when attempting to apply CATOCS to actual application problems can be characterized as follows: CATOCS is not adequate in itself to ensure application-level consistency, and providing additional mechanism at the state level to remedy this deficiency eliminates the need for CATOCS, or it is expensive except at very small scale, and there is a far more efficient state-level solution that is straight-forward to provide. The rest of this section identifies specific limitations and the costs of CATOCS that lead to this situation. We informally summarize these limitations of CATOCS as: *1) it can't say "for sure," 2) it can't say the "whole story," 2) it can't say "together," and 3) it can't say efficiently.*

1. *Unrecognized Causality (or it can't say "for sure").* Causal relationships can arise between messages at the semantic level that are not recognizable by the "happens-before" relationship on messages and, hence, not enforceable by CATOCS. This situation typically arises from an external or "hidden" communication channel [16], such as a shared database or external environment.

Figure 2 illustrates this limitation with a manufacturing system that is typical of current manufacturing environments. Multiple instances of a "shop floor control" system are executing and accessing a common database. A "start processing lot A" request arrives at instance (1) and shortly thereafter a "stop processing lot A" arrives. Each instance processes its respective request, updates the shared database, and multicasts the results. A semantically-meaningful causal relationship is created between these requests using the database as a (hidden) communication channel. However, because this interaction is outside the scope of the communication substrate, the multicast messages are not recognized as causally related and can arrive in an order at recipients that violates their semantic-level causal ordering.

The same anomaly can arise if the two "instances" referred to in the example of Figure 2 are two concurrent threads within the same multi-threaded process, with the shared state of the address space constituting the "hidden channel". It is possible that thread 1 updates the shared

**Figure 2.** An example illustrating unrecognized, hence unenforced, causal relationships. The shared database orders all requests made to the Shop Floor Control (SFC) system, but this ordering is unknown to the communications substrate.

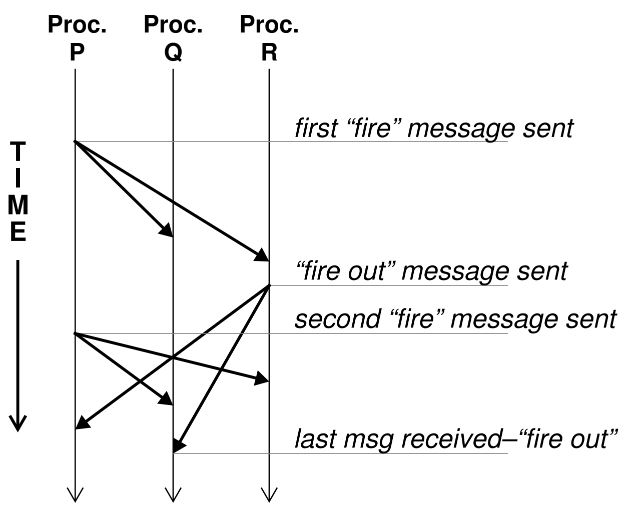


memory data structures first, but is delayed by scheduling in sending its multicast message so that the second update by thread 2 is actually multicast first and therefore is delivered by CATOCS out of order with respect to the actual shared state update and the true causal dependencies. Forcing all inter-thread communication to take place using the message system instead of shared data structures would impractically reduce the performance of multi-threaded servers, given the cost of messages compared to the cost of shared memory access. Even maintaining an optimized causal graph between threads interacting through shared data structures would be a significant code complexity and performance overhead.

Figure 3 illustrates this same limitation with an external channel, namely a fire, in a similar manufacturing setting. A process controlling a furnace detects a fire on two separate occasions and multicasts "fire" warnings accordingly. A separate monitor program detects the "fire out" in the first instance and multicasts its results. Unfortunately, the last message received by a third process (Q) is the "fire out" message, and it incorrectly concludes that the fire is out. All messages are sent by causal multicast. Note that the same behavior could be exhibited using a total-ordered multicast. The fire is effectively an external channel through which a semantic causal dependency is created between the "fire" and "fire out" messages. This dependency is not recognized by the internal communication facilities, and therefore not enforced by CATOCS.

This limitation significantly limits the usefulness of causal communication for many applications because many, if not the majority, of interactions take place through shared resources and external channels that are outside the scope of the communications substrate. For example, the increasingly popularity of multi-threaded shared servers makes the shared state of threads a major problem. Similarly, a significant amount of communication in real-time control systems occurs through the external channel provided by the external system being controlled. In any case, it is very difficult to guarantee the absence of hidden channels, prompting us to again characterize this limitation of CATOCS as: *can't say "for sure"*.

**Figure 3.** An Event Diagram illustrating another example of anomalous behavior, whereby dependent messages are delivered late. All communication is via causal multicast.




These causal dependencies can be easily handled by adding prescriptive ordering information to messages that reflect the true ordering or causal dependencies, and having recipients use this information to ensure the proper ordering. For example, considering the shared manufacturing database above, if "lot status" records contained version numbers, then any recipient can easily and correctly order the messages. However, the provision of these version numbers, which can be viewed as logical clocks on the database state, obviates the need for CATOCS. Moreover, these application state techniques also eliminate the uncertainty of an overlooked hidden channel that is present with a CATOCS-only solution[3].

2. *Lack of serialization ability (or can't say "together").* CATOCS cannot ensure serializable ordering between operations that correspond to groups of messages.

Updates to data structures typically involve groups of memory operations, so some means of ensuring serializability of the group of operations constituting the update is required for application-level consistency. For example, a shared memory system among two or more processors can ensure a consistent total ordering of accesses to memory, but that alone does not ensure the consistency of the data structures in the shared memory. Locking is the standard solution, although optimistic concurrency control techniques can also be used. CATOCS only provides ordering between individual messages. An update that requires a group of messages to be handled as a serial unit requires additional mechanism. This extra mechanism typically obviates the need for CATOCS. For example, locks on shared data provide mutual exclusion between the memory updates of different processors, making the relative ordering of these memory access between processors otherwise irrelevant, so CATOCS is not required. This same argument

3. In theory, external communication channels can create problems for state-based approaches, such as having an external channel communicate uncommitted updates in a transaction to another part of the system. However in our experience, the computer channels are so much faster than the typical external channels that these problems do not arise except possibly under catastrophic failure conditions or with, for example, very long-lived transactions in design systems where the users are making a trade-off favoring availability over strict consistency. In these cases, user participation may be required to handle inconsistencies, just as arises after continued operations during a network partition.

applies in our examples in Section 4 which require transactional support. The attendant serialization facility of the transaction mechanism obviates the need for CATOCS. The lack of serialization ability for CATOCS is also a potentially serious incompatibility when interfacing between CATOCS-based applications and transactional systems, which are quite prevalent commercially.

The inability of CATOCS protocols to group operations is also a significant limitation in handling higher-level error conditions. For example, an update message of replicated state distributed across a group of server processes only results in a consistent replicated state if each process is able to accept and process the message, not just receive the message in a consistent order. However, in reality, one or more server processes may reject an operation because of lack of storage or protection problems or other state/application-level constraints. Standard atomic transaction protocols allow a participating server process to abort a transaction for these reasons. In many proposed uses of CATOCS, such as for maintaining distributed replicated data, a process must effectively fail and invoke the full failure notification mechanism if it rejects a message for this class of reasons, or else rely on a separate rollback recovery mechanism for undoing the effects of the message delivery at the other nodes. The former case is expensive and generally assumes that the message will eventually be accepted by the "failed" process while the latter approach entails an atomic transaction mechanism that again obviates the need for CATOCS. Note that dropping a message at the state/application level is equivalent to reordering the message delivery, so if dropping the message is acceptable in this case, the message ordering support of CATOCS is of limited or no value to the application.

3. *Unexpressed semantic ordering constraints (or it can't say the "whole story").* Many semantic ordering constraints are not expressible in the "happens-before" relationship and, hence, not enforceable by CATOCS.

Generalizing the previous limitation, the correct behavior of an application requires ordering constraints over operations on its state, and these constraints are typically stronger than or distinct from the ordering constraints imposed by the happens-before relationship. Such ordering constraints, referred to as "semantic" ordering constraints, run the gamut from weak to strong, and they may or may not require grouping as well. Example constraints include causal memory [1], linearizability [12], and, of course, serializability. Even the weakest of these semantic ordering constraints, causal memory, can not be enforced through the use of causal multicast [1]. Although this weak ordering constraint can be enforced using totally ordered multicast, such protocols are expensive and much cheaper protocols, which utilize state-level logical clocks, can be used instead. For the stronger ordering constraints, such as linearizability and serializability, neither causally nor totally ordered multicast is sufficient, as illustrated in the stock trading example developed in Section 4.1.

4. *Lack of Efficiency Gain over State-level Techniques (or can't say efficiently).* CATOCS protocols do not offer efficiency gain over state-level techniques, and appear far less scalable.

CATOCS imposes an ordering overhead on all messages yet does not eliminate the need for the prescriptive ordering on messages and operations required for end-to-end semantics. That is, CATOCS does not eliminate or reduce the need for timestamps or versions in the real-time manufacturing examples or the locks in the examples using transactional techniques. Moreover, CATOCS is prone to delaying messages based on *false causality*, namely messages that are incidentally causally dependent at the communication level but not semantically causally dependent. This situation arises because the happens-before relationship on messages indicates *potential* causality, *not actual* causality. Just because one message is received before a second message does not necessarily mean that the first message caused the second. For example, false causality would arise in Fig. 1 if message m2 just happened to be sent after receipt of message m1 but was not in any semantic sense "caused" by receipt of message m1 (It could have been caused by an internal timer or external input, for instance). False causality reduces performance by unnecessarily delaying messages until the earlier supposedly "causally related" messages are received and delivered.[4] It also increases the total memory requirements for buffering "unstable" messages, a potentially prohibitive cost with scale, as argued in Section 5. The amount of false causality appears to be dependent on application behavior and the causal domain or group size. However, there have not been any studies of the overheads incurred by false causality, so a major concern for the designer of systems using CATOCS is the uncertainty regarding the false causality overhead to expect, and of course the challenge to construct groups whose communication patterns minimize false causality.

Finally, CATOCS imposes overhead on every message transmission and reception—ordering information is added each transmission and checked on each reception. This overhead will be an increasingly significant cost as networks go to ever higher transfer rates and other aspects of protocol processing are further optimized. However, optimizing of CATOCS is of limited interest unless greater

4. Note that lightweight causal multicast protocols delay messages in order to preserve "causal" ordering [4]. As an alternative to delaying dependent messages, causal protocols can append earlier "causal" messages to later dependent messages, but this technique can significantly increase network traffic.

functional benefit can be identified, the question we return to in the next section.

Overall, CATOCS as a communication facility is limited to ensuring communication-level semantics and cannot recognize and enforce application state-level or "end-to-end" semantic requirements, whether they arise through "hidden channels," grouping requirements or state access control and resource limits. This overall limitation is basically a corollary of the well-known end-to-end argument, which states that a lower-level facility cannot ensure higher-level semantics, but can at best be an optimization for higher-level mechanisms. In their original paper describing this argument [25], this limitation is illustrated by considering a (careful) file transfer. A file transfer has completed successfully only when all the file data has been safely and correctly stored in the file system of the recipient machine, not just when the data has been delivered by the network to the recipient machine. (The recipient machine could crash or have a buffer overflow at that point, for instance.) An end-to-end check can determine and acknowledge when the entire file has been correctly and safely stored on the recipient's file system. However, in this example, the basic transport protocol is a useful lower-level optimization because it can identify individual packets of file data that have been dropped or reordered during transmission and correct the situation, optimizing for the common case of transmission problems and avoiding the overhead of retransmitting the entire file. As we have argued, CATOCS appears to provide no comparable optimization, and in fact appears to introduce significant extra overheads.

# 4 Classes of Distributed Applications

Several classes of distributed applications, including the major examples used in the CATOCS literature to justify CATOCS, are examined in order to evaluate the merits of using CATOCS for these applications, as compared to other established state-level techniques. The limitations identified in the previous section arise repeatedly in this discussion.

## 4.1 Data Dissemination Applications

**Netnews**. Communication support for Usenet newsgroups is often cited as an example of the need for causally ordered communication. In the current Usenet world, it is possible to receive a response to an inquiry to a news group before receiving the inquiry[5]. CATOCS would ensure that the multicast response was delayed until the causative inquiry had been delivered.

However, solving this problem using CATOCS would introduce even less desirable behavior. If the causal group was the entire news group, then all messages sent subsequent to the inquiry would have to be considered potentially causally related to the inquiry. In this case, a user would see all subsequent messages to a news group delayed if the inquiry was lost or delayed. To match actual causality to the incidental ordering of CATOCS, a new causal group would have to be created for each inquiry. The number of resulting causal groups would be enormous, given the number of independent inquires at any time across all Usenet news groups. The amount of state maintained by the communication system is proportional to the number of causal groups as well as the amount of traffic that is "outstanding". With the current and growing levels of traffic, scale and related individual failures and down-time of participating nodes, the overhead would be impractical. In addition, the logistics regarding generation and deletion of groups would be non-trivial. As many readers of news groups will realize, the actual grouping of inquires and responses in practice can be much more complicated than what we have considered here, rendering the use of CATOCS even more problematic.

It is relatively straightforward to solve the Netnews ordering problem at the application state level. Assign each News article a globally unique id and, for each response, have a designated field containing the id of inquiry article[6]. The local news database could maintain order relationship, and specifically note which articles were missing. When the news database is browsed, the user would have the option of displaying out-of-order responses or not. The complexity of maintaining ordering information in the local news database is proportional to the number of inquiries that are of interest to the user, rather than to the number that have been sent. Furthermore, the database maintains only the actual causal dependencies since it has access to the required semantic information. This example further points out that the problems of CATOCS lie not with notion of causal relationships, but with attempting to implement the causal ordering at the communication level.

**Trading Application Example.** Security trading is another application that has been widely cited in support of the utility of CATOCS. Dissemination of trading information to trader workstations in a consistent order seems like a natural use for causal multicast—certainly one would not want to observe price changes out of their actual order of occurrence. However, causal multicast cannot enforce all necessary semantic ordering constraints and appears to be too expensive[7].

5. The actual problem is not particularly severe because most responses, in fact, contain sufficient information about the inquiry, if not actually including the original inquiry, that the news reader can infer the inquiry from the response. In an informal poll of our colleagues, we found no strong sentiment that this misordering of messages in Usenet newsgroups was a significant problem. However, we have considered this example in detail because it has been widely discussed in support of CATOCS.

6. A version of this scheme already exists. The "References" line of a posting can contain the "Message-ID" tags on which the posting depends.

**Figure 4.** An Event Diagram for the trading example illustrating anomalous ordering behavior (all communication is via causal multicast).

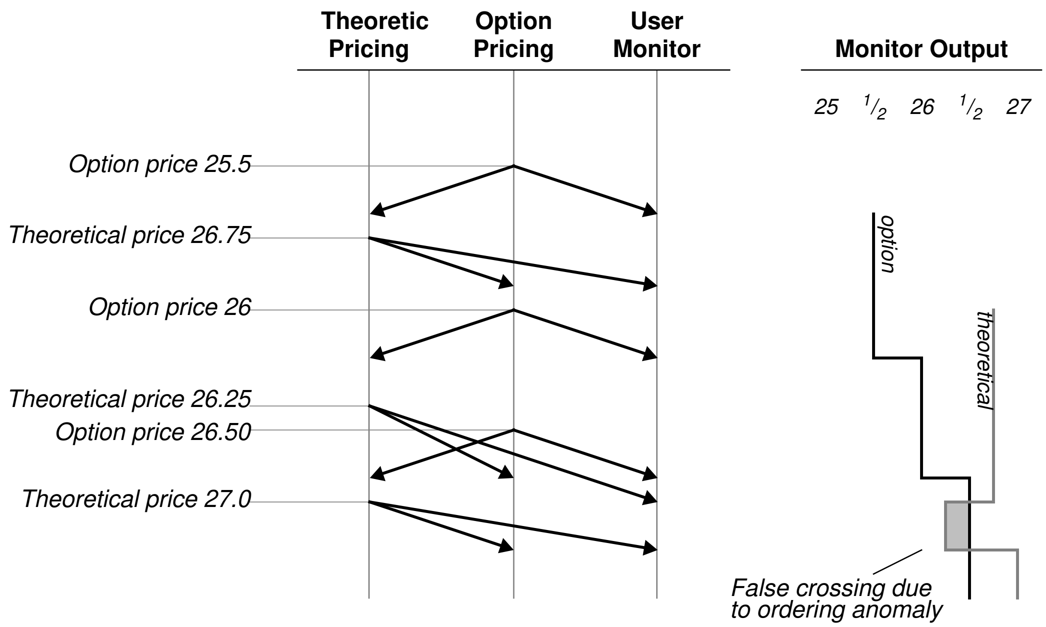



Figure 4 illustrates a scenario in which the semantic ordering constraints are not enforced. The price of an option is multicast by one server. Another server calculates the theoretical price of the option, which is the basis for many trading decisions. The correct behavior for this application obeys a semantic ordering constraint defined informally as follows: a theoretical price is ordered after the underlying option price from which it is derived and before all subsequent changes to that underlying price. An anomaly occurs in the depicted example when the theoretical price data is delivered after the underlying option price has changed. A monitoring program could then observe an inconsistent view of the world: the new option price and a theoretical price based on the old option price. Because the new option price and the old theoretic price are "concurrent messages" according to the happens-before relationship on messages, neither causal or total multicast can avoid this anomaly.

This example illustrates limitation 3—"can't say the whole story." The semantic ordering constraint between the new option price and the old theoretic price is stronger than the "happens-before" relationship and, hence, unenforceable by CATOCS. Note that a large percentage of the data on a trading floor is computed data, e.g. the theoretical pricing data, and therefore semantic ordering constraints are frequent. (Note that although the anomaly in this example superficially resembles the "hidden channels" anomalies depicted in Figures 2 and 3, the limitation illustrated is different.)

A CATOCS-based solution also appears too expensive because each unique stock and instrument should be assigned its own process group so as to not over-constrain message ordering when using causal multicast. However, a large trading floor must monitor price changes on several hundred thousand stocks and derivative[8] instruments, requiring more process groups than we understand current CATOCS implementation can support.

In production systems we have designed, every pricing service[9] maintains version numbers on security prices, either as a real-time timestamp or as a sequence number. Each computed data object records the id and version number of its base data object in a designated "dependency" field. General-purpose utilities maintain the dependencies among data objects, and applications exploit this information in ordering and presenting data. Thus, consistent with our overall theme, a simple state-level solution based on dependency-preserving utilities can be implemented with low cost and a high-degree of generality. These utilities do not require causal multicast, nor would they be simplified by using it.

7. We have helped design over 150 commercial trading floors that are in production use and so speak from a significant basis of experience.

8. An example of an derivative instrument is an "buy option" on an underlying stock.

9. Pricing information comes via communication lines from external sources, such as the NYSE, with a local pricing server for each such line.

Both the Netnews and the trading solutions outlined above can be generalized to the notion of an order-preserving data cache, which can be useful for a number of applications. This cache approach is another example of how problems can be simplified when tackled at the right level of abstraction within the system. Semantic information, such as the inquiry-response relationship in Netnews and the object version dependencies in trading applications, is readily available in the application state level, but is not exposed to the communication level.

## 4.2 Global Predicate Evaluation

Global predicate evaluation problems are normally expressed in terms of a stable predicate (i.e., once true, always true—at least until detected and corrective action is taken). Examples include distributed deadlock detection, distributed garbage collection and orphan detection. The most general solution to this problem involves taking a snapshot of local process states that represent a *consistent cut* of local process states, which can be done in a straightforward way with CATOCS [29].

CATOCS-based solutions for the stable predicate detection problem, though elegant, have a couple of major disadvantages. Firstly, they require the use of CATOCS on every communication interaction that could possibly affect the state of the stable predicate. Given that detection protocols typically run periodically, rather than continuously, and at a frequency that is often 3 orders of magnitude less than the frequency of message sending, it would hard to justify the cost of using CATOCS on every communication just to detect stable properties. Secondly, with limitation 1 of CATOCS, a solution that relies on a CATOCS may also fail when there are "hidden channels" or unrecognized causal relationships.

Given the important role of stable predicate detection in real systems, much research has focused on identifying efficient protocols not requiring the use of CATOCS. Elnozahy et al have proposed a periodic consistent snapshot protocol that takes a full "consistent cut" at the state level without CATOCS [9]. Such a protocol is useful both for checking global predicates and for failure recovery.

Marzullo and Sabel have identified a subclass of detection problems based on the notion of "locally stable" predicates [21]. This subclass contains most problems of practical importance, including the detection of transaction deadlock, RPC deadlock, k-of-n deadlock, loss of a token, orphans, termination, and some forms of garbage collection. The authors present an efficient, general-purpose detection protocol for this class of problems and then proceed to derive message-efficient, special-purpose detection protocols from the general protocol. Schiper and Sandoz have independently investigated a similar subclass of detection problems characterized by "strong stable" properties, and they have presented an efficient detection protocol for that class [26].

Focusing on deadlock detection problems, an important subset can be re-formulated in terms of local predicates whose evaluation is insensitive to message ordering—effectively transforming the detection problem from one of taking a consistent cut to one of taking just a cut, which is a much simpler problem. These deadlock detection protocols are a proper subclass of both locally stable detection problems and strong-stable detection problems, and can be solved by a simple protocol. For example, consider distributed deadlock detection in transaction system using 2-phase locking. Its local property is given below and is insensitive both to event ordering and the message state of the system.

- *Consider a set of transactions t1, t2, t3, .... tn that uses 2-phase locking. The set is deadlocked if and only if each of the following is independently true at some time during their execution—t1 waits-for t2, t2 waits-for t3,... tn-1 waits-for tn and tn waits-for t1*[10].

This property implies that "wait-for" relationships can be detected incrementally, that it need not be detected in the order in which events actually occurred, and that the order of receipt of "wait-for" information does not affect the detection of valid deadlocks. Hence, to construct the global "wait-for" graph it is sufficient to have each node multicast its local wait-for graph to all nodes running the detection algorithm. No stronger ordering properties are required. Also it follows from the "only-if" clause of the first property that only actual deadlocks are detected— no "false" deadlocks are detected. Simple variants of the above deadlock detection algorithm can also be used for nested transactions [18], for RPC deadlock detection [6], and for orphan detection.

As the above discussion indicates, most of the important stable predicate detection problems occurring in real systems fall into subclasses that can be solved with general purpose detection protocols that do not use CATOCS. These protocols are cheaper than CATOCS-based protocols because the latter require the use of CATOCS on every communication interaction, not just those communications involved in taking a consistent cut. Even detection problems requiring a full "consistent cut" can be solved using a periodic consistent snapshot protocol, which can also be implemented efficiently at the state level without CATOCS.

## 4.3 Transactional Applications

Transactional applications have also been cited as an example use of CATOCS [4]. However, a distributed transaction management protocol already orders the transactions

10. Note that this property does not necessarily hold for non-2-phase protocols.

(i.e. ensures serializability). In particular, with pessimistic transaction management, the ordering of transactions is dictated by 2-phase locking on the data that is accessed as part of the transaction. The relative message ordering from concurrent, but separate, transactions is irrelevant with regards to correctness because each transaction is committed independently, The prepare-to-commit phase of the protocol necessarily requires end-to-end acknowledgments because each participating node must be allowed to abort the transaction. Thus, by limitation 2, CATOCS cannot be used to execute this phase. Because the commit protocol is executed by a single site, namely the commit coordinator, the delivery of commit phase messages is easily ordered by conventional transport mechanisms without CATOCS. Hence, CATOCS is not needed, and as noted as limitation 2, not sufficient in place of locking because mutual exclusion across a group of operations is required, not just a consistent ordering of the individual operations.

Considering intra-transaction ordering of operations, conventional transactions are executed by a single sequential process so the operations are ordered by the process's execution and conventional transport protocol ordering applied to the messages generated by the process. Within a multi-threaded transaction, the threads need to be synchronized at the state level in general. Otherwise, groups of operations by separate threads could be interleaved resulting in inconsistencies, even with a consistent causal or total ordering on these actions. Thus, CATOCS is again neither adequate by itself nor necessary when a state-based solution is applied.

With a so-called optimistic transaction system, transactions are globally ordered at commit time, with a transaction being aborted if it conflicts with an earlier transaction. Thus, no inter-transaction ordering is required during the execution of the transaction operations. Moreover, a simple ordering mechanism, such as local timestamp of the coordinator at the initiation of the commit protocol, plus node id to break ties, provides a globally consistent ordering on transactions without using or needing CATOCS.

## 4.4 Replicated Data

Replicated data management is a frequently cited reason for using CATOCS [4] because it supposedly simplifies the higher-level mechanisms, avoids some of the update aborts due to failures that can arise with a transactional solution, and improves performance through asynchrony. However, on close inspection, an optimized atomic transaction approach appears uniformly superior.

First of all, the simple "begin-transaction/end-transaction" facilities of a transaction mechanism provide a simple high-level interface for application code. It also provides more functionality in its ability to atomically group updates and abort groups of updates (say "together').

Secondly, a replicated data management system such as a replicated transactional file system using a "read-any, write-all-available" protocol can be optimized to match the behavior of CATOCS in the presence of failure. In particular, a transaction updating replicated files can drop failed servers from the availability list at transaction commit and then commit the transaction with the remaining servers provided the transaction was not holding read locks on any of the failed servers. (The availability list mechanism is required for bringing servers back up into a consistent state before they begin serving clients with both CATOCS and transactions.) With this transaction protocol optimization, a simple replicated file update aborts for the same failure cases as does CATOCS. However, the CATOCS provides no comparable support for consistent update of groups of files or objects (i.e. can't say "together") so it appears again as an inferior solution compared to optimized atomic transactions.

Finally, the actual asynchrony one achieves with CATOCS systems is limited, as illustrated by considering the Deceit file system [27] which was built using the ISIS system, a toolkit implementing CATOCS protocols [2]. In Deceit, the write updates are sent to the replicas using causal multicast but there is essentially no asynchrony because each "cbcast", the ISIS causal multicast operation, waits for k acknowledgments with a so-called "write safety level" or fault-tolerance level of k. A write-safety level of 0 is asynchronous but it has no acknowledgments so the write data could be lost after a *single failure* (because of the lack of durability), compromising the semantics of, and presumably the purpose of, replication. With a typical replication level of 2, a non-zero write safety level implies synchronous update with all servers, just as with conventional RPC, because the write is multicast by the primary site server handling the write operation. With higher degrees of replication, one would only expect some benefit if there were significant differences in round trip times to the different servers, and the write-safety level was smaller than the number of replicas.

Generally speaking, CATOCS requires trading *concurrency* for *asynchrony*. Transactional systems execute individual updates synchronously, but permit concurrent updaters because these systems necessarily support concurrency control (for serializability). Therefore, although parallelism within a single update is restricted, parallelism among concurrent updates is not. Even individual computations can use threads to increase concurrency, hence parallelism, while updating. In contrast, CATOCS-based implementations typically enforce a primary updater approach because CATOCS provides no explicit mechanism for concurrency control [3, 27]. Similarly, the primary can not be multi-threaded. However, asynchrony of updates is limited because of non-durability, as discussed above, so the

CATOCS solution ends up being less parallel and therefore less efficient.

Considering Deceit in more detail, a number of considerations in this server further limit the benefits of CATOCS. Firstly, considerable server-specific mechanisms are required to handle reconciliation of files after significant failures or network partitions, and CATOCS did not aid the implementation of these mechanisms. In fact, according to the implementor of Deceit, the failure detection was the primary benefit provided by the ISIS facilities for this server, and CATOCS (specifically, causal multicast) was of relatively limited benefit [28]. Secondly, every failure in Deceit results in a flurry of messages between members of the process group for each "active" file to create a new view and "flush" [4] messages sent under the previous view. This view change is managed as a synchronous operation and imposes a processing and messaging overhead to the "availability set" update similar to the overhead required with a transactional system.

As a specific point of comparison, the HARP file server [4] is a transaction-based replicated NFS file server providing a service similar to the CATOCS-based Deceit file server. HARP uses highly optimized atomic transaction techniques, is claimed to provide better performance than Deceit and tolerates a wide range of failures. This is achieved in spite of the fact that NFS dictates treating each file write as a separate transaction.

## 4.5 Replication in the Large

Replication in the large, such as with large-scale naming services [17], can exploit application state-specific techniques to ensure consistency of updates and also exploit application-specific tolerance of inconsistencies and anomalies to favor availability and performance over strict ordering. For instance, Lampson's design suggests that duplicate name binding can be resolved by undoing one of the name bindings. In the scale of multi-national directory service that this design addresses, tolerating the occasional "undo" of this nature seems far preferable in practice than having directory operations significantly delayed by message losses or reorderings. Moreover, there is no experience with operating causal or total ordering support on this scale, and the size of communication state that would be required in each node seems impractical, given expected levels of traffic and reasonable probabilities of node and communication failures. Finally, although CATOCS might be viewed as a more general solution, the saving in code and the expected extra costs of operation do not appear to justify using it over a more specialized solution, especially extrapolating from the Deceit experience where the overall system complexity was dominated by management and recovery aspects outside of, and independent of, the base CATOCS facility.

## 4.6 Distributed Real-time Applications

Real-time applications, another cited application area for CATOCS [3], are generally characterized as systems that monitor and/or control some physical system in real-time or "clock-on-the-wall" time. Examples include factory monitoring and control, airplane autopilots, distributed interactive simulation and many others. One can distinguish a monitoring aspect and a control aspect to most of these systems and applications. The limitations of CATOCS in real-time systems are significant.

Firstly, as pointed out in Section 3, the causal relationships implemented by CATOCS in a real-time system may be incomplete because many true or semantic causal relationships are "implemented" in the monitored system, totally outside of the CATOCS mechanism (i.e., the "unrecognized causality" limitation). For example, a message sent to an actuator to move a temperature sensor within a factory oven may semantically cause an alarm message from a separate oven temperature sensor but this relationship is not recognized in the CATOCS mechanism. In particular, the control message may be received at a message logging device after the alarm message.

Secondly, CATOCS does not support the need in real-time systems to execute groups of operations at the same real time to achieve a desired effect, which is otherwise easily implemented by timestamping messages and possibly a transaction mechanism to abort the group. For example, in starting up a factory oven, the lighting of the pilot should be grouped with the opening of a gas value to the pilot flame.

Finally, the CATOCS inefficiency of delaying message delivery because of false causality and its general communication overheads detracts, not just from the performance, but from the correctness of a real-time system. In a monitored system, the correctness of the system, that is the semantic notion of consistency, is maximized by minimizing the difference between the computer-stored state and the actual state of the monitored system. For example, the value for the oven temperature stored by a computer-based oven control in a factory should be close to the actual temperature of the oven, what we call "sufficient consistency". Sufficient consistency is normally provided by the sensors transmitting periodic updates, the communication system giving priority to the most recent updates (dropping older updates if necessary), and the monitoring system interpolating, smoothing and averaging updates to accommodate lost updates, replicated sensors and erroneous readings [20]. Update messages delayed by CATOCS reduce consistency with the monitored system and therefore detract from the correctness of operation. The delay occurs whenever a message is received that is potentially causally dependent on another message that has not yet been received. Moreover, the consistent ordered view of failures provided in some CATOCS systems means that additional message delays are often incurred when a process fails while the new view is determined and propagated to all process group members. Additional group-wide delay in real-time systems is often a worse form of failure than a failure of an individual group member, given that the process functions are often replicated in systems requiring reliability.

Implementing only part of the message traffic in a system with CATOCS to avoid these disadvantages only aggravates the problem of the message delivery using CATOCS accurately reflecting neither the "happens-before" nor the true causal relationships among messages. For example, an update message to an oven controller may cause it to send out a message to a cell controller, which may in turn cause it to initiate some reconfiguration action. If the first message is sent outside of CATOCS, but the rest sent within causal or totally ordered process groups, only part of the causal relationship is recognized by the CATOCS. The real-time programmer must therefore carefully reason that this partial correctness of the CATOCS delivery is sufficient for the application.

Similar arguments can be put forward for the controller aspects of real-time distributed systems (see [6]).

In contrast to CATOCS-based approaches, a state-based approach using real-time clock values, the key shared piece of state in a real-time system, is simple to implement and provides far better semantics, including true temporal precedence, the most important precedence relationship in real-time systems. For example, if an oven sensor places a real-time timestamp in each update message, this information can be used for ordering of events by real-time, recording these events in a log and correlating these events with other factory events that may have occurred outside of the monitoring system. For example, the local power utility may report a power surge at a particular (real) time that the factory manager wishes to correlate with the log of oven behavior. A real-time timestamp provides this ability, while the purely logical and incidental ordering provided by CATOCS does not. The implementation of distributed (real-time) clock synchronization is well understood, takes little communication or processing, and is available in a variety of distributed systems. The amount of mechanism required is substantially smaller than that required for CATOCS and is also "off the critical path" in the sense that the time synchronization is not invoked on every data communication action. Synchronized time service is required in most real-time systems in any case and can be made highly reliable. Finally, a computer-maintained timestamp can be far more accurate than the timing of events within most real-time systems. For example, a timestamp can have a granularity in the microsecond range and an accuracy to less than one millisecond, and yet the events in most real-time systems occur at the granularity of tens of milliseconds or more. For example, even computer-controlled flight surfaces on the advanced tactical fighters (ATF) operate in only the 25 Hz range. Moreover, for very high-performance control, it does not make sense to separate the sensor from the controller by a network, given that packets can be lost, introducing delays exceeding those acceptable to the application. (Note that the potential delay due to a lost packet is generally well above the round-trip time, especially for multicast communication where the problem of acknowledgment implosion precludes tight time-outs.)

Overall, "temporal precedence" provided by timestamping specifies an ordering of events that is semantically meaningful in real-time systems and that provides all the ordering of events that is required without imposing load or complexity on the communication system. In contrast, CATOCS cannot with its incidental ordering capture the actual causal relationships and semantic ordering constraints between messages, unnecessarily delays the delivery some messages detracting from correctness and performance, and imposes a complexity and processing burden on the communication support.

# 5 CATOCS Scalability

CATOCS appears to introduce significant problems with scaling because of the roughly quadratic growth in expected message buffering that arises with growing numbers of participating processes[11]. This section provides an informal argument of this expected behavior.

Consider scaling a system of N processes using CATOCS. The causal order of messages in a system can be represented as a directed acyclic graph with nodes as messages and an arc between two nodes represents messages that are potentially causally related. The *active causal graph* is the subgraph that results from deleting nodes corresponding to "stable" messages and their incidental arcs, where *stable messages* are those known to have delivered everywhere.

The number of nodes (messages) in the active causal graph is proportional to the number of processes in the system by the following reasoning. The time between the sending of a message and its deletion from the active causal graph is roughly proportional to the "diameter" of the system, that is the time T to propagate a message across the system. The number of messages transmitted during time T, and thus the number constituting the nodes of the active causal graph, is equal to the message rate per process times the number of processes times T. The time T grows with the number of processes, roughly proportional to the square root of the number of processes.[12] Actually, we just need T

11. We use the term *process* here to designate a participant in a CATOCS group to be consistent with the previous discussion and the terminology of most CATOCS literature. However, a process in this discussion is typically representative of a service on a network node.

to be non-decreasing with the number of processes for the desired result.

The number of arcs can grow quadratically in the number of messages. For instance, a process that multicasts a new message to the group after receiving a message introduces N new arcs into the graph. Although one might argue that properly structured applications exhibit linear growth in the active message graph, the number of arcs grows quadratically in almost all possible causal graphs and “false” causality tends to increase the number of arcs beyond the actual causal relationships in the application.

The amount of buffering in a CATOCS system tends to grow proportional to the number of arcs in the active causal graph for several reasons. If the number of causal graph arcs is quadratic in the number of processes, the number of causal dependencies communicated in each message must be, on average, linear in the number of processes. The number of messages referenced in these causal dependencies that have not been received by the process also grows linearly in the number of such references, assuming the error/delay rate is roughly a fixed percentage of the message rate, a standard and realistic assumption. Thus, the amount of buffering used by each process for holding these delayed messages can be expected to grow linearly in the number of processes in the system.

Therefore, the buffering requirements in the system tend to grow quadratically with the number of processes. More practically stated, the buffering requirements at an individual node tend to grow linearly with the increase in scale of the system in which the node participates, an unfortunate property that has caused problems in other systems (like the Xerox Grapevine mail system). The buffering requirements also grow quadratically because of the memory required to store the active causal graph although these descriptors tend to be much smaller than the messages themselves.

Buffering requirements are further increased in implementations that provide atomic message delivery because each node must retain a copy of each message it references in any message it sends until the referenced message is known to be stable, i.e. all the recipients have received these referenced message. For example, if a cell controller sends out a message that is potentially causally dependent on a message from a sensor according to the CATOCS, the cell controller buffers the message from the sensor until the sensor message is stable. As argued in Section 2, atomic message delivery seems necessary for any useful implementation of CATOCS, plus it is assumed in many example uses of CATOCS for fault-tolerant systems. That is, the receiver of a new message assumes it can get copies of the causally referenced messages from the sender of the new message even if the original sender of one of the referenced messages has crashed.

Group membership change protocols, required by CATOCS to enforce atomic delivery semantics, are another scalability concern because the rate of member failures increases linearly with group size as well as the cost of each protocol execution. Membership change protocols also suppress the sending of new messages during a significant portion of the protocol.

Partitioning a large process group into smaller process groups does not necessarily reduce this problem unless the smaller groups are not causally related. For instance, the “causal domain” [4], proposed as a causally related set of groups, can have the same quadratic growth. The division into groups only reduces the application-generated message traffic to each receiver, not the message delivery delays.

The buffering requirements can be reduced by delaying the sending of messages (in addition to delaying their delivery). This delaying has obvious detrimental effects on application performance, particularly in real-time or near real-time applications such as trading example of Section 4.1. The delaying of messages also produces a highly synchronous execution, because messages must be delayed for end-to-end acknowledgments indicating stability. Finally, it increases the communication overhead for “stabilizing” messages because there are fewer application messages on which to piggyback acknowledgment information (such as the “vector clock” [4]).

In summary, the worst-case and statistically expected behavior for CATOCS is for the buffering requirements to grow quadratically in the number of processes in the system. Delaying transmission of messages and restricting the communication topology can reduce this problem at the cost of application performance. Various optimizations such as partitioning process groups, sending additional message traffic, and such like may reduce the buffering required as well, but these techniques provide no guarantees and increase the complexity of the implementation—a troublesome situation for applications that are striving for fault-tolerance. Finally, the state-level solutions that we have put forward as alternatives throughout the paper either do not require extensive buffering, as in the real-time applications, or are able to buffer updates on secondary storage, as in the replicated data implementations. Moreover, state-based applications can easily distinguish updates from queries to reduce the buffering or logging, whereas a CATOCS approach cannot without using a separate process group for reads from writes with the attendant loss of causal ordering between these types of operations.

12. The square root growth assumes a uniform world of nodes packed into a circle. In reality, there is a significantly higher delay for wide-area communication compared to local-area communication. Relatedly, growing scale introduces growing heterogeneity of communication technology and capacities, further increasing delay and packet loss.

# 6 Conclusions

Causally and totally ordered communication support (CATOCS), while an appealing idea, appears to be significantly limited by four major limitations, that we have identified informally as: 1) can't say "for sure," 2) can't say "together," 3) can't say the "whole story," and 4) can't say efficiently. In the classes of distributed applications we have considered, which include those used in the CATOCS literature to justify this support, these limitations imply that CATOCS is not adequate or else is only very narrowly applicable. In each case, we have presented a state-based solution using established techniques that appears relatively straight-forward to implement and obviates the need for CATOCS. In the cases we have examined, CATOCS also fails to provide much simplification to the application writer. Finally, scalable performance appears to be another problem with using CATOCS. It adds processing overhead and buffering to the performance-critical message transmission and reception paths, it unnecessarily delays messages because of false causality, and it fails to allow much asynchrony because of the need for end-to-end acknowledgments (except in real-time uses where the CATOCS mechanism appears to have limited benefit). In fact, these considerations and its expected quadratic cost in message buffering appear to restrict its practical deployment to applications that are significantly restricted in size and communication topology.

The well-known end-to-end argument has played a key role in our investigation and these conclusions. By considering real distributed applications in some detail, the "end-to-end" semantics of these applications were available for critically evaluating both their full requirements and alternative techniques to CATOCS. In contrast, much of the CATOCS literature contains abstract process/message examples that lack sufficient detail to identify these alternatives. We repeatedly observe that the applications place requirements on the distributed state, whereas CATOCS can only provide limited consistency guarantees on the communication. By the end-to-end argument, state consistency can only be ensured by state-level operations and CATOCS can at best be regarded as an optimization. However, the mechanism required to ensure state-level consistency subsumes the need for CATOCS, and CATOCS provides no performance improvement in the cases we considered.

It is attractive to have one generic mechanism or abstraction such as CATOCS that can be used across a variety of distributed applications and systems even if there is an application-specific solution that is better for each problem. However, the ideal framework should be a state-level framework, not a communication-level one like CATOCS, because, again, consistency requirements are at the state-level. Distributed systems supporting objects and abstract types, under development in a number of research and development organizations, and in fact in commercial use in some cases, provide such a framework [8, 11, 15, 23, 30]. Objects are state-level entities so object systems are focused on the state level techniques, with communication being incidental to their implementation, as appropriate for consistency and reliability considerations. Moreover, object systems provide the powerful object-oriented mechanisms of inheritance and type-safe dynamic binding to allow applications to use generic base class functionality (such as atomic transaction support) yet specialize from these base classes as needed to meet application-specific requirements. The resulting frameworks are easier to use than programming at the basic message level, solve state problems at the state level, and avoid complicating the base communication facilities (thereby avoiding the attendant performance and robustness penalties).

Broadening our focus slightly, we conjecture that the view expressed by Lamport of logical clocks based on communication activity is better applied at the state level. In the object-oriented view of distributed systems, it is the "clock ticks" on the state, the object versions, that are relevant, not the "communication clock ticks." Moreover, the state clocks generally advance at a rate that is an order of magnitude slower than communication clocks (because reads outnumber writes.) Finally, state clocks are easily made as durable as the state they relate to because one can write out the clock value as part of updating the state, whereas the high rate of communication clock ticks generally makes their stable storage infeasible. Thus, reasoning about distributed systems using communication clocks fails to focus on the relevant aspect, namely the state, separates one from the important dynamics, namely the state updates, and deals with rather ephemeral and easily recreated activity, the communication, rather than the key issue for fault-tolerance and recovery, namely state durability. State-level clocks not only more accurately drive our reasoning but correspond more directly with practical implementations.

Although this work has focused primarily on understanding the limitations of CATOCS, it also addresses the general issue of placement of function in distributed systems, using the powerful end-to-end argument to show the merits of state-level reasoning and implementation techniques over communication-level approaches. With applications dictating requirements and with a choice of implementation techniques available at each level, we regard the correct level of function placement as a key challenge for systems designers. Our proposed principle, solve state problems at the state level, represents a promising starting point. We hope and expect that further research and experience with distributed systems and applications will refine and extend our understanding in this area.

# 7 Acknowledgments

The paper benefited enormously from the comments of the anonymous reviewers as well as those from Stuart

Cheshire, Ed Lazowska, Keith Marzullo, Hector Garcia-Molina, Alex Siegel, Raj Vaswani and Willy Zwaenepoel.

# 8 References


[1] M. Ahmad, P. Hutto and R. John, Implementing and Programming Distributed Shared Memory, *Proc. of the 11th International Conference on Distributed Computing Systems*, May, 1991.

[2] K.P. Birman and T.A. Joseph, Reliable Communication in an Unreliable Environment, *ACM Transactions on Computer Systems 5*, 1 (Feb. 1987), 47-76.

[3] K.P. Birman and T.A. Joseph, Exploiting replication in distributed systems, in *Distributed System*s, edited by S. Mullender, ACM Press, Addison-Wesley, New York, 1989.

[4] K.P. Birman, A. Schiper and P. Stephenson, Lightweight causal and atomic group multicast, *ACM Transactions on Computer Systems 9*, 3 (August 1991), 272-314.

[5] D.R. Cheriton and W. Zwaenepoel, Distributed Process Groups in the V Kernel, *ACM Transactions on Computer Systems 3*, 2 (May 1985), 77-107.

[6] D.R. Cheriton and D. Skeen, Understanding the Limitations of Causally and Totally Ordered Communication, Comp. Sci. Research Report STAN-CS-93-1485, Stanford Univ., Sept. 1993.

[7] S.E. Deering, Host Extensions for IP Multicast, Internet RFC 1112, Aug. 1989.

[8] D.L. Detlefs, M.T. Herlihy, and J.M. Wing, Inheritance of Synchronization & Recovery Properties in Avalon/C++, *IEEE Computer*, Dec. 1988, 57-69.

[9] E.N. Elnozahy, D.B. Johnson and W. Zwaenepoel, The Performance of Consistent Checkpointing, *Proc. of the 11th Symp. on Reliable Distributed Systems*, Oct. 1992, 39-47.

[10] N. Goodman, D. Skeen, A. Chan, U. Dayal, S. Fox and D. Ries, A Recovery Algorithm for a Distributed Database System, *Proc. of the 2nd Symp. on the Principles of Database Systems*, Atlanta Georgia, March 1983, 8-15.

[11] M.P. Herlihy., A Quorum-Consensus Replication Method for Abstract Types, *ACM Transactions on Computer Systems 4*, 1 (Feb. 1986), 32-53.

[12] M.P. Herlihy and J.M. Wing, Linearizability: A Correctness Condition for Concurrent Objects, *ACM Trans. on Programming Languages and Systems 12*, 3 (July 1990), 463-492.

[13] T.A. Joseph and K.P.Birman, Reliable Broadcast Protocols, in *Distributed Systems*, edited by S. Mullender, ACM Press, Addison-Wesley, 1989.

[14] P. Keleher, A.L. Cox, and W. Zwaenepoel, Lazy Release Consistency for Software Distributed Shared Memory, *Proc. of the 19th Int. Symp on Computer Architecture*, 13-21, May 1992.

[15] R. Ladin, B. Liskov, L. Shrira and S. Ghemawat, Providing High Availability Using Lazy Replication, *ACM Transactions on Computer Systems 10*, 4 (Nov. 1992), 360-391.

[16] L. Lamport, Time, Clocks and the Ordering of Events in a Distributed System, *Comm. of the ACM 21*, 7 (July 1978), 558-565.

[17] B. Lampson, Designing a Global Name Service, *Proc. 5th ACM Symp. on Principles of Distributed Computing*, Calgary, Aug. 1986, ACM, 1-10.

[18] B. Liskov, D. Curtis, P. Johnson and R. Scheifler, Implementation of Argus, *Proc. of the 11th Symposium on Operating System Principles*, Austin, TX, November 1987.

[19] B. Liskov et al., Replication in the Harp File System, *Proc. of the 13th Symposium on Operating System Principles*, Oct. 1991, Pacific Grove, CA, 226-238.

[20] K. Marzullo, Tolerating Failures of Continuous-Valued Sensors, *ACM Transactions on Computer Systems 8*, 4 (Nov. 1990), 284-304.

[21] K. Marzullo and L. Sabel, Using Consistency Subcuts for Detecting Stable Properties, *Proc. of the International Workshop on Distributed Algorithms*, Delphi, Greece, Oct. 1991, 273-288.

[22] S. Mishra, L. Peterson, R. Schlicting, Implementing Fault-tolerant Replicated Objects using Psync, *Proc. of the 8th Symposium on Reliable Distributed Systems*, Seattle, Washington, Oct. 1989.

[23] B. Oki, M. Pfleugl, A. Siegel, D. Skeen, The Information Bus®– An Architecture for Extensible Distributed Systems, *Proc. of the 14th Symposium on Operating System Principles*, Dec. 1993, Asheville, North Carolina.

[24] L. Rodrigues and P. Verissimo, xAMp: a Multi-primitive Group Communications Service, *Proc. of the 11th Symposium on Reliable Distributed Systems*, IEEE, Houston, TX, October 1992.

[25] J.H. Saltzer, D.P. Reed and D.D. Clark, End-to-End Arguments in System Design, *ACM Trans. on Computer Systems 2*, 4 (Nov. 1984) 277-288.

[26] A. Schiper and A. Sandoz, Strong Stable Properties in Distributed Systems, Technical Report LSE-TR93-02, Dept. of Computer Science, EPF Lausanne, March 1993.

[27] A. Siegel, K.Birman and K. Marzullo, Deceit; A Flexible Distributed File System, *Proc. of the Usenix Summer Conf.*, Anaheim, CA June 1990, 51-61.

[28] A. Siegel, Private communication, April 1993.

[29] R. van Renesse, Causal Controversy at Le Mont St.-Michel, *Operating Systems Review 27*, 2 (April 1993), 44-53.

[30] W.E. Weihl and B. Liskov, Implementation of Resilient, Atomic Data Types, *ACM Transactions on Programming Languages and Systems 7*, 2 (April 1985), 244-269.

# 9 Appendix

This appendix presents additional examples contrasting CATOCS-based approaches with non-CATOCS-based ones.

## 9.1 Drilling Example

Some control actions entail scheduling sequences of actions that must be performed by independent nodes according to a particular schedule. An example of using causal ordering support to control cell controllers sequencing drilling operations is given by Birman [3]. It is *possible* in some cases, such as in this scenario, to map application-level scheduling requirements onto the incidental ordering provided by causal communication. The solution is appealing in providing a highly distributed execution in which the driller controllers all schedule independently from one broadcast drilling request. However, the question we address is whether one should actually implement this application using CATOCS. We think not.

The input to the process is a set of holes to be drilled. The end product is a checklist, describing holes that need to be checked, because of drills that may have failed part way into the drilling step. The constraint is that holes should not be drilled multiple times, even if only partially drilled the first time. The solution is supposed to survive failures of driller controllers. The state is thus the set of holes and their states, namely undrilled, being drilled, completed or to be checked, plus the state of the driller controllers. In a realistic solution, a central cell controller would monitor the state of the driller controllers, assigning drilling operations to each driller and transmit the assigned holes to each driller, with the driller reporting back as it completed each hole. For reliability, the central controller state might be replicated at one or more other sites, although in practice a second backup is sufficient and storing the state on stable storage as protection against power glitches common to factories would be more important than higher degrees of replication. With this approach, the communication traffic is linear in the number of driller controllers, not quadratic as claimed for Birman's solution [3], and no CATOCS is required.

The distributed scheduling approach used in Birman's solution in which each driller controller independently assigns holes to each driller relies on causal ordering to guarantee that two drills do not collide on the same hole. The same thing can be accomplished in a state approach, making the state a distributed replicated object, and having each driller monitor update this state as it completes each hole. The update can be performed as a simple multicast-based replicated update to the schedule without locking or commit because each hole is assigned to a separate controller, and the update cannot be refused. In fact, each driller only has to wait for as many acknowledgments as the Birman solution waits for synchronously, which is tied to the degree of fault-tolerance required. When a drill fails, the failure monitor can synchronize all the driller controllers to the new schedule and the remaining drillers can proceed with a new non-conflicting schedule. The latter requires a synchronous atomic transaction across all driller controllers. However, failures are not frequent and the overall performance of the resulting system should be faster than CATOCS techniques in the normal case because of less communication, i.e. non-ordered multicast RPCs for the normal drilling updates.

## 9.2 RPC Deadlock Detection

A RPC deadlock detection algorithm was recently presented by van Renesse [29], and its apparent simplicity is attributed to the use of causal communication protocols. However, this algorithm appears to be too expensive in most realistic settings. The following alternative RPC deadlock detection algorithm, based on a more general formulation of the problem, is as simple as van Renesse's algorithm, performs better, and does not require causal communication. The solution is based on a re-formation of the detection problem expressed as a stable local predicate.

In van Renesse's algorithm, each process causally multicasts each RPC invocation and each RPC return. A monitor process receives all RPC-related events and constructs a *wait-for* graph. The appearance of a cycle in the wait-for graph indicates a deadlock. (More than one monitor process can be used to provide fault-tolerance.) The use of causal communication yields a simple, but expensive algorithm. Each RPC invocation in this algorithm results in 2 causal multicasts, and these multicasts are to a process group that consist of *all monitor processes plus all processes that could potentially invoke one another using RPC*, in order to propagate potential causality information. Thus, the performance penalty of this algorithm appears prohibitive, especially for detection of a relatively infrequent event like deadlock.

Consider the following alternative solution based on a simple re-formation of the problem. Assign each RPC invocation a (locally) unique instance identifier. Augment the "wait-for" information with instance identifiers. For example, $A_{15} \rightarrow B_{37}$ denotes that RPC instance 15 executing within process A is waiting for RPC instance 37 within process B. Each process periodically multicasts the augmented wait-for information to a set of monitor processes, with a conventional sequence number or timestamp ensuring that multicasts sent by the each process are received in the order sent. The monitors construct the wait-for graph where nodes represent instances. As before, a cycle implies a deadlock.

This alternative algorithm is as simple as van Renesse's, but does not require causal communication. Moreover, it is more general: it can handle multi-threaded processes where several invocations may be active at the same time, and it can detect deadlocks that involve resource sharing among RPC instances within a multi-threaded server.