Causal Broadcasting and Consistency of Distributed Shared Data

K. Ravindran & K. Shah

Department of Computing & Information Sciences
Kansas State University
Manhattan, KS 66506 (USA)

Abstract

The paper develops a generalized model to capture the interactions between the ordering of messages exchanged across various entities of a distributed application and the consistency requirements on a shared data across these entities. The model is based on causal broadcasting of data access messages that allows messages to be ordered at all entities as per the constraints specified by the application. This allows each entity to change its local data copy based on the messages processed and still be in agreement with other entities at selected points of message exchanges that are meaningful to the application, which we refer to as *stable points* in the underlying execution. Since the causal relationships among messages depict an invariant property of the application and stable points are reproducible across different execution instances, application-specific consistency of the data can be enforced in many cases without explicit protocols to reach agreement. The model of integrating causality with data consistency allows more flexibility in the implementation of data access protocols for distributed services and offers potential for increased performance of the protocols.

1 Introduction

A distributed computation is often structured as a set of one or more entities that communicate with one another to share a common data. The entities often reside on different machines interconnected by a network, with each entity exchanging messages with other entities over the network to synchronize their access to shared data (or state). For example, a distributed file service may be implemented by a group of servers, with each server maintaining a local copy of files and exchanging messages with other servers in the group to update the various file copies in response to client requests. As another example, a conferencing service may be provided by a set of workstation agents, with each agent managing a local window, say, on a design document and exchanging messages with other agents to support interactive sharing of the document by various conference participants. Such types of data access may often be supported in distributed and/or replicated services using a message broadcast facility that allows a data access message to be seen by all entities concerned with the data. See Figure 1.

The value of a shared data may change when an entity updates the data, and often the data may be read by some entities while the changes caused by other entities are occurring. In such a dynamic environment, correctness of the computation requires that the data be *consistent*, i.e., the data value changes at various points of message exchanges among entities during execution in a manner acceptable to the application. The consistency of data is a basic problem to be solved in constructing service access protocols [1].

The framework of transactional serializability to achieve data consistency is less amenable for integration into a generalized computation model than a framework that uses global ordering of data access messages across various entities [2]. In existing models based on message ordering, the ordering semantics is often inseparably buried into the problem-specific computations. This precludes a generalization of the interactions between message ordering and problem-specific consistency requirements, which often limits the insight into how message ordering constraints may be generated in a given computation. The goal of this paper is to develop a generalized model for programming distributed services that allow sharing of a common data.

The application level messages M exchanged across various entities occur in a partial order satisfying the *causality constraints* generated by the application. The causality constraints depict inter-dependencies among messages M, and manifest as precedence relations '$\succ$' on the processing of M by entities. These relations are denoted as $\mathcal{R}(M) \equiv \{\succ\ (m, m')\}_{\forall m,m' \in M' \subseteq M}$, where a relation $m \succ m'$ indicates the application level requirement that a message m' should occur after a message m in an underlying execution of the application [3]. The causality relations $\mathcal{R}(M)$ basically generate a set of allowed sequences in which messages in M can occur. In a given instance of execution of the application, an entity may process messages in one of the allowed sequences. $\mathcal{R}(M)$ depicts a *stable information* in the application, i.e., holds true in all instances of execution of the application.

To establish the mapping between causal ordering and

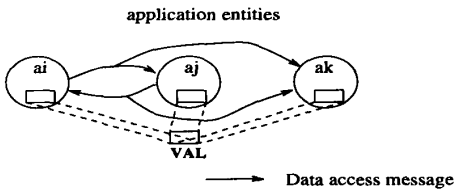

Figure 1: Message exchanges to access shared data

data consistency, consider a causal relation $m \succ m'$ in an application. Then a relation $m \rightarrow m'$, denoting the sequence of message occurrence 'm happens before m'', holds true at each entity observing a value $VAL(m)$, where $VAL(m)$ is the agreed value of data in various entities at the point of occurrence of m in an execution of the application. In the presence of dynamic changes to the data, the data agreement requires that $VAL(m') = \mathcal{F}(VAL(m), m')$, where $\mathcal{F}$is a function depicting changes to $VAL(m)$ during the interval between the processing of m and m'. This means that the occurrence of changes to the data due to m' are agreed upon by all entities by virtue of propagating the knowledge of causal relation $m \succ m'$ across these entities.

The propagation of knowledge of causal relations is achievable by a communication construct — referred to as *causal broadcasting* — that allows delivery of messages M at all entities for processing in the causal order specified in $\mathcal{R}(M)$. From a distributed programming point of view, serialization of data access is achieved simply by projecting the problem of data consistency in terms of causally ordered message exchanges, which in turn allows reaching agreement on a data value *without explicit protocols* (similar to 'virtually synchronous executions' in ISIS [2]). In comparison with existing models of implementing distributed data access where application level message causality information is used only indirectly [1, 4], the integration of message causality and data consistency in our model:

- Allows more flexibility in the implementation of data access protocols across a variety of applications;
- Offers potential for increased protocol performance.

The generality and extensibility of our model allows it to be incorporated in a distributed programming system in the form of data access primitives (such as causally related 'read' and 'write' operations [5]) that can be used to analyze and implement complex applications.

The paper is organized as follows: Section 2 discusses the relation between message causality and application semantics. Section 3 discusses models of causal broadcasting. Section 4 gives a computational framework for distributed data access. Sections 5-6 explore how the relationship between causal dependencies and consistency of shared data can be used in data access protocols. Section 7 concludes the paper.

2 Message causality in computations

Let $\{a_i, a_j, a_k\}$ be the set of entities executing an application. When an event occurs at an entity, say a_i, the event processing manifests as an action taken by a_i that may include accessing the data maintained by a_i and/or generating a message to access the data maintained by a_j and a_k. The globally distributed data may be realized by a message broadcast facility that allows each access message to be seen by a_i, a_j, a_k.

2.1 Causal relations

The order of event occurrence with respect to other events is based on the causal relationship among various events. The causal relationship is specifiable from an application perspective in the form of a constraint on the ordering of events acceptable to the computation. Consider the set of messages $M = \{m_1^i, m_2^i, m^k\}$, with $\{m_1^i, m_2^i\}$ and m^k generated by a_i and a_k respectively. Each dependency relation in $\mathcal{R}(M)$ indicates a *partial order* in which m_1^i, m_2^i and m^k are to be seen by a_i, a_j and/or a_k. Suppose the application requires that m^k should occur before m_1^i can be generated; then we say 'm_1^i occurs after m^k', as denoted by the relation '$m^k \succ m_1^i$'. The '$\succ$' is a transitive relation on M in that $m^k \succ m_1^i$ and a relation, say, $m_1^i \succ m_2^i$ imply that m_2^i causally depends on m^k. When neither $m_1^i \succ m_2^i$ nor $m_2^i \succ m_1^i$ are specifiable, m_1^i and m_2^i are *concurrent* messages (denoted as $\|\{m_1^i, m_2^i\}$). In this case, both $m_1^i \rightarrow m_2^i$ and $m_2^i \rightarrow m_1^i$ are acceptable message processing sequences ('$\rightarrow$' is basically Lamport's 'happens before' relation on externally observed events [6]).

Causal broadcasting is a communication construct that allows propagation of the knowledge of causal relations across all application entities, which enables each entity to order messages in the computation. Suppose, for example, an application requires a read operation to follow the last increment operation on an integer data. When an entity observes an increment performed by message m during execution, it will order the subsequent read message m' using the causal relation $m \succ m'$. Figure 2 provides a sample causal broadcast scenario.

2.2 Causal ordering and state consistency

An application is often implemented in the form of client entities accessing server entities for specific services. Each service provides a set of operations for use by clients. The service semantics captures the inter-relationships between service operations, and is specifiable in the form of interleavability of various operations during execution. Clients may invoke one or more of the server operations by specifying appropriate causal ordering on the invocation messages that meets the consistency requirements in the application. Consider, for example, a service which provides increment/decrement (inc/dec) and read (rd) operations on an integer data, with the requirement that "a

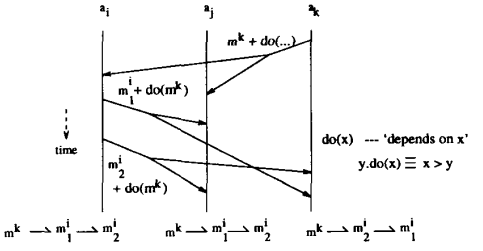

Figure 2: Scenario of causal broadcasting in a sample computation: $\mathcal{R}(M) \equiv m^k \succ \|\{m_1^i, m_2^i\}$

rd operation cannot be concurrent with a *inc/dec* operation, while the *inc* and *dec* operations can be concurrent". Accordingly, clients may, say, strictly order the *rd* and *inc* operations but relax the ordering of *inc/dec* operations. As can be seen, application characteristics can be cast on the orderings allowed on a given set of client messages.

When a service is replicated and/or distributed across a set of server entities, intra-service state consistency requires the server entities to coordinate with one another and provide the same view of a shared state at every tick in logical time (i.e., at every message exchange to access the state). In the above example, a client requirement may be that each server entity performs an *inc* before a *rd*, i.e., $inc \succ rd$, on their local copies of the integer data, (this also guarantees 1-copy serializability).

In some cases, applications have an inherent ability to tolerate, to some extent, inconsistent data values at various server entities without affecting the correctness. In the above example, two *inc* operations may be performed concurrently, i.e., $\|\{inc_1, inc_2\}$, which may result in different intermediate values due to different sequences in the processing of inc_1 and inc_2 messages but result in the same final value. Referring to Figure 2, the view of global state available at entities a_j and a_k may not be the same when the concurrent messages m_1^i and m_2^i are processed since they can be delivered in different sequences. However, when another message m_3^i is generated with a causal relation $\|\{m_1^i, m_2^i\} \succ m_3^i$, a_j and a_k will have the same view when m_3^i is delivered (i.e. consistent). Such a message constitutes a *synchronization point* in the computation signifying agreement on the views among entities.

Thus consistency requirements can be uniformly specified in terms of the causal order in which messages need to be exchanged between clients and servers in the access operations. The ordering constraints on messages can be weaker than a strict ordering (as per application requirements), which provides a potential for a higher degree of concurrency in accessing the shared data.

In the next section, we describe the models of causal broadcasting.

3 Models of causal broadcasting

The causal broadcasting may be realized by organizing various entities as members of a *group*, and sending every message and the causal relations associated with this message to all the members. So members can deterministically process messages in a sequence allowed by the causal order and in the context of their local knowledge of causality, and have the same view of application level state at every distinct point in logical time.

Causal broadcasting exists at the conceptual level in a distributed computation, and is typically supported by the kernel of a distributed operating system in the form of a communication interface model. The interface model deals with expressing the causal relationships in an application as a partial order specification on messages, exchanging this information across members of a group and mapping the relation to the delivery sequence of the messages to various members.

3.1 Message dependencies and state changes

An application protocol $\mathcal{T}$ is executed by a group of entities. Occurrence of events in $\mathcal{T}$ manifest as processing of messages M by the member entities subject to the causal order constraints $\mathcal{R}(M)$ (refer to Figure 2). Each member receives messages in the required order along with causality information, and changes state by processing them, as governed by the state transition function F given by:

$$\mathcal{F} : M \times \underline{S} \rightarrow \underline{S}, \qquad (1)$$

where $\underline{S}$ is the state space as viewed by the member. Accordingly, a message sequence $m_a \rightharpoonup m_b$ occurring in state s_0 of a component x causes the function invocation sequence $s_1 := \mathcal{F}(m_a, s_0)$ and $s_2 := \mathcal{F}(m_b, s_1)$. Suppose a message sequence $m_b \rightharpoonup m_a$ occurs instead, causing the invocation $s_1' := \mathcal{F}(m_b, s_0)$ and $s_2' := \mathcal{F}(m_a, s_1')$. If $s_2' = s_2$, then m_a and m_b are concurrent (e.g., m_a and m_b depict *inc* operations on an integer data). In this case, $\mathcal{F}(m_b, \mathcal{F}(m_a, s_0)) = \mathcal{F}(m_a, \mathcal{F}(m_b, s_0))$. These notions capture the application level state transitions in terms of the messages exchanged.

The causal dependency $\mathcal{R}(\{Msg, m, \cdots\})$ may be represented by a graph in which, say, the dependency of Msg on m is represented with a directed edge connecting an ancestor node Msg to a descendant node m (see Figure 3). A *many-to-one* dependency refers to the dependency of one or more messages $\{m'\}$ on a single message Msg. In this case, the various m' are concurrent messages (i.e., $\mathcal{R}(\{Msg, \{m'\}\}) \equiv Msg \succ \|\{m'\}$) because no dependency relationships are specified among them. This is illustrated below:

$$Occurs\_After\ (m'_1, Msg);$$
$$Occurs\_After\ (m'_2, Msg);$$
$$\vdots$$

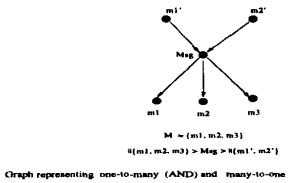

Figure 3: Message dependencies as a graph

A *One-to-many* dependency refers to the dependency of a single message Msg on one or more messages $\{m\}$. In one form, Msg can be processed after *all* messages in $\{m\}$. The $Occurs\_After$ relation to capture this AND dependency is

$$Occurs\_After(Msg, (m_1 \wedge m_2 \wedge \cdots)), \quad (2)$$

The dependency may be represented with Msg as the ancestor node and each m as its direct descendent node.

3.2 Agreement based on causal ordering

An agreement protocol that is based on the guarantee of an identical message sequence at every member (say, total order on messages) operates at the granularity of individual messages. On the other hand, an agreement protocol that uses the stable form of a message dependency graph can be made to operate on the same set of concurrent messages between two synchronization points as seen at various members. The sequence of messages between synchronization points may not be same at various entities. Since every member observes the same set of messages between synchronization points, agreement among members on the value of shared data is feasible at a higher granularity of sets of messages (rather than individual messages), namely, the synchronization points. This implies that there is more asynchronism in the execution of the agreement protocols .

A stable form of the graph representing message dependencies in an application is extractable by observing its execution behaviour in terms of messages exchanged and generating therefrom a specification of the intended communication requirements of the application. The observation may be available by:

- Providing a communication interface layer on top of the ISIS **CBCAST** or x-Kernel **Psync** primitives [7, 8] to isolate the application level communication requirements and the potential linearization of partial orders on messages by the physical communication system during transport and delivery;
- Using a communication interface that provides an explicit specification of the causal relationships among messages in a non-procedural form, such as our **O_Send** primitive [3].

In this paper, we use the **O_Send** primitive to generate the message dependency graphs[1], as discussed below.

[1] A more recent analysis of causal ordering in distributed computations [9] stresses the need for a causal order on messages generated by the application to reflect the 'semantic ordering' determined by the message contents, rather than inferring the causal order from the observed 'incidental ordering' of messages on the physical communication system.

3.3 'O_Send' primitive to extract graphs

A member may encapsulate a causal relation in a **O_Send** primitive that takes the following form:

$$\textbf{O\_Send}\ (Msg,\ \mathcal{T},\ Occurs\_After(m)), \quad (3)$$

where $Occurs\_After$ is the ordering predicate, namely, a message Msg should be processed by various members of the group associated with $\mathcal{T}$ after a message m, i.e., $m \succ Msg$. When $m = NULL$, Msg can be processed without any constraint. A member of $\mathcal{T}$ changes from its current state to a new state by processing Msg in the context of causal relation $m \succ Msg$. So the order on messages received and their contents specify the state of the member. The receipt of m guarantees that any dependency on m, such as $m \succ Msg$, is eventually satisfiable at all members because the dependency is a stable information in the application.

4 A framework for shared data access

In this section, we identify a framework for shared data access using the stable message graphs extractable during execution of the application. The framework is based on identifying suitable points in a message graph as synchronization points.

4.1 Stable points and causal activities

Let $\widehat{F}$ be a relation characterizing the state transition in $\mathcal{T}$ at a member due to the causal order $\mathcal{R}(M)$:

$$\widehat{F}:\ \mathcal{R}(M) \times \underline{S} \rightarrow \underline{S}. \quad (4)$$

Let $\{Ev\_Seq_1, Ev\_Seq_2, \ldots, Ev\_Seq_L\}$ be the list of message sequences generatable from $\mathcal{R}(K)$ with $K = \{m_0, m_1, \ldots, m_r, m_{r+1}\} \subseteq M$, with m_0 placing the member in an initial state s_{old}, where $1 \leq L \leq (r+1)!$. A message sequence Ev_Seq_i depicts the event occurrences $e_{i1} \rightarrow e_{i2} \rightarrow \ldots \rightarrow e_{i\overline{r+1}}$, where an event e_{ij} maps one-to-one to a message in $\{m_1, m_2, \ldots, m_{r+1}\}$. From relations (1) and (4), Ev_Seq_i may be characterized as

$$\begin{aligned} s_{new,i} &:= \widehat{F}([e_{i1} \rightarrow e_{i2} \rightarrow \ldots \rightarrow e_{i\overline{r+1}}], s_{old}) \\ &:= \widehat{F}([e_{i2} \rightarrow \ldots \rightarrow e_{i\overline{r+1}}], \mathcal{F}(e_{i1}, s_{old})), \end{aligned}$$

where $s_{new,i}$ is the state after the occurrence of Ev_Seq_i. A state s_{new} generated by $\mathcal{R}(K)$ is a *stable point* in the execution of $\mathcal{T}$ if s_{new} can be reached by any of the event sequences, i.e., $s_{new,i} = s_{new,j} = s_{new}$ for any

pair of event sequences Ev_Seq_i and Ev_Seq_j ($i \neq j$). In this case, $\mathcal{R}(K)$ represents a causal activity (e.g., $m_0 \succ \|\{m_i\}_{i=1,2,\ldots,r} \succ m_{r+1}$) and the underlying event sequences are *transition-preserving*.

A causal activity may be serializable with respect to other activities, so $\mathcal{T}$ may use a stable point as the initial state for the next activity. The causal activities establish distinct points in logical time (or ticks) as $\mathcal{T}$ progresses. Since causal broadcasting allows a message dependency graph to be seen at various members identically, each member has the same view of when stable points occur in $\mathcal{T}$.

4.2 Synchronization at stable points

The explicit representation of the causal relationships among messages, as underscored in the predicate-style specification of ordering constraints, allows flexibility in the form of computations being able to: i) specify complex ordering relationships among messages, typically various extents of *weak orderings* on messages, and ii) construct higher level *causal activities*, where a causal activity is described by a set of messages $K \subseteq M$ and an ordering relationship $\mathcal{R}(K) \subseteq \mathcal{R}(M)$ [3]. To access a shared replicated data for instance, $\mathcal{R}(K)$ may indicate the unit of access at which consistency of the data needs to be guaranteed. In other words, consistency needs to be guaranteed only at stable points.

This flexibility can be beneficial in building computation models for service replication in which consistency requirements on a shared data among entities may be expressed at application-specific granularity of causal activities (rather than at message granularity). The benefits accrue in the form of better structural tools for building applications and/or increased execution performance of applications. In terms of the 'state-machine' approach to design distributed services [10] for instance, each entity is simply a 'state-machine' replica, and consistency is achieved by producing the same set of transitions at every 'state-machine' replica as allowed by the causal order.

Structuring of an application in terms of causal activities allows flexible interworking of various protocol components in distributed data access. A consistency enforcement protocol that allows access to a local copy of the data maintained by an entity at each causal activity may simply be based on recognizing the occurrence of stable points.

5 Models to enforce data consistency

Consider a set of data items $\underline{X} = \{x\}$ shared among members of a group implementing an application $\mathcal{T}$. Each member maintains an instance of $\underline{X}$ and operates on one or more items in the local instance upon receiving messages. Our goal is to specify consistency among various instances of $\underline{X}$ at the granularity of causal activities and design appropriate execution support mechanisms.

5.1 Exploiting Causality

A set of messages $\{m_1, m_2, \ldots, m_r\}$ depicts a stable point if the event sequences for $\mathcal{R}(\{m_1, m_2, \ldots, m_r\})$ are transition-preserving for each $x \in \underline{X}$. This condition relates to decomposition of the data $\underline{X}$ into distinct items and scoping out the effects of messages on these items . It also subsumes the case where messages affect disjoint subsets of $\underline{X}$. As an example, a member may apply increment/decrement operations on one or more integer data in any sequence, and hence process the corresponding messages concurrently. Such sequence of operations are called *commutative* operations. For example, if three operations $inc(x)$, $dec(x)$, and $rd(x)$ are to be ordered, we may relax ordering between $inc(x)$ and $dec(x)$ as the increment and decrement operations on same integer data are commutative, while the read operation is not commutative with respect to increment/decrement operations. Hence, we may order operations as $\|\{inc(x), dec(x)\} \succ rd(x)$, resulting in more concurrency.

Consider, as another example, a multiplayer card game where r players share common data space in a window system and where players take turns in a certain presequence [11]. Suppose an action of the l^{th} player does not depend on the action of the preceding $(l-1)^{th}$ player ($l = 1, 2, \ldots, r$) but on that of some other player k, where $k < (l-1)\ mod\ r$. In this case, the l^{th} player generates his action after seeing the action of the k^{th} player in his workstation window, as given by the causal relations: $card_k \succ card_l$ and $\|\{card_l, card_i\}_{i=(k+1,\ldots,l-1)\ mod\ r}$. This results in a relaxed ordering of the messages and is thus reflected in higher concurrency.

In general, the application semantics often does not require messages to be processed in a strict order. Instead, it suffices that agreement among members on the value of various $x \in \underline{X}$ be guaranteed only at stable points. Because of the relaxed order however, the sequence of state transitions between stable points may not be the same at various members. So a read operation on $\underline{X}$ requested at a member may be deferred to occur at the next stable point so that the value of $\underline{X}$ returned by the member is the same as that by every other member. Such a level of data consistency can be achieved by an access protocol because causal broadcasting delivers M along with the stable causal relations $\mathcal{R}(M)$ at all members in the group.

5.2 Additional support mechanisms

In loosely coupled applications, messages may be generated *spontaneously*, i.e., without any causal relationships to one another, to operate on $\underline{X}$. An example is a distributed conferencing in which the participants collaboratively annotate and/or modify a design document from their workstations [11]. Another example is a distributed name service in which resolutions from clients and registrations from servers may occur independently on a name repository. Such operations appear as concurrent messages

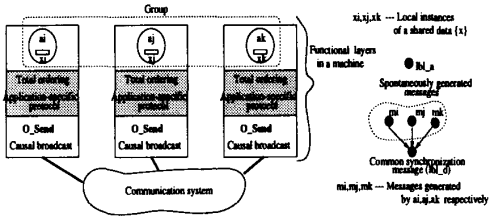

Figure 4: Functional layer for total ordering of messages and application-specific protocols

with many-to-one dependency on a common synchronization message in our model. There are two ways to handle these messages (see Figure 4):

Total ordering of messages

A function may be interposed between the causal broadcast and application layers to: i) impose an arbitrary delivery order on a set of messages spontaneously generated by members, and ii) enforce the order identically at all members. A total order on such messages $\{m_1', m_2'\}$ that are causally dependent on a message Msg is specified as

$$\textbf{A\_Send}\ (\{m_1', m_2'\},\ Occurs\_After(Msg)), \qquad (5)$$

where the **A_Send** primitive enforces either $Msg \rightarrow m_1' \rightarrow m_2'$ at all members or $Msg \rightarrow m_2' \rightarrow m_1'$ at all members. Thus the sequence of state transitions is the same at every member in an execution instance of the application.

In terms of the **O_Send** based causal broadcast interface, a total order can be defined over a set of messages $\{m\}$ specified by (lbl_a, lbl_d), where lbl_a and lbl_d refer to the ascendant node of $\{m\}$ and the descendant node(s) of $\{m\}$ respectively in the dependency graph. The case where lbl_d is $NULL$ and lbl_a is a termination message represents a total order on all messages generated by the application. Our **A_Send** is different from the 'causal **ABCAST**' primitive of ISIS [7] in terms of how the total ordering is determined in relation to causal ordering of messages.

The total order allows members to agree on the value of $\underline{X}$ at every message occurrence without explicit protocols to reach agreement. Typically, each member executes an application-specific deterministic algorithm on the messages to construct higher level functionalities. Total ordering may be feasible when the group size is not large [12].

Use of application-specific protocols

In some cases, it may be expensive and/or inflexible to track dependencies among spontaneously generated messages; so the knowledge of state transitions may not be completely available at members (e.g., when the group size is large, as in distributed name servers). These cases may lead to data inconsistency in some situations, whereby members may return different values of $\underline{X}$ for a read operation. This has to be tackled at the application level [4]. Consider, for example, an application which supports query (qry) and update (upd) operations on a data (e.g., name registry in a name service). Query operations (which return the data modified by update operations) may be allowed to be concurrent since they are commutative. So one member, for instance, may see a sequence $upd_1 \rightarrow qry_1 \rightarrow qry_2 \rightarrow upd_2$, while another member may see the sequence $upd_1 \rightarrow qry_2 \rightarrow qry_1 \rightarrow upd_2$. Here, both qry_1 and qry_2 return the same data value to members. Now, if one member sees the sequence $upd_1 \rightarrow qry_1 \rightarrow upd_2 \rightarrow qry_2$, the data value returned by qry_1 and qry_2 can be different. The application should discard qry_2 since it leads to incorrect result. To enable such a check (for inconsistency), the query operation carries sufficient context information in terms of the ordering of upd_1 and upd_2.

The application level support to deal with potential inconsistencies induces more complexity in the access protocol than the use of algorithms based on total ordering of messages, but provides more asynchronism in execution of the protocol when inconsistencies occur infrequently.

Our approach to maintaining consistency of distributed shared data is somewhat different from the 'distributed shared memory' model used in [5] in the way the shared data is realized and the application semantics is exploited in the access protocols.

6 Methods for enforcing consistency

The application level operations are categorized as commutative and non-commutative. Based on this categorization, a generic replicated data access protocol can be designed that can be used across a variety of applications. The knowledge of how the various operations affect the data may be embedded into the data access protocol in the form of a causal order on messages.

6.1 Use of stable points

The processing of messages M by a replica may be viewed as consisting of a repetitive cycle of processing a non-commutative message, denoted as $rqst_{nc}$, temporally followed by a set of $\overline{K}$ (≥ 0) commutative messages (on an average), denoted as $\{rqst_c\}$. This is shown by the causality relation:

$$rqst_{nc}(r-1) \succ \|\{rqst_c(r,k)\}_{k=1,2,\ldots,\overline{K}} \succ rqst_{nc}(r+1),$$

where $r = 1, 2, \ldots$ indicates the processing cycle. The messages $rqst_{nc}(r)$ and $rqst_{nc}(r+1)$ constitute the stable points for the r^{th} processing cycle. $\overline{K}$ reflects the mix of commutative versus non-commutative operations generated in the application. Typically, 90% of the operations are commutative (e.g., as in many database applications). Thus, for example, $\overline{K} = 20$.

A non-commutative message may itself be a commutative set (i.e., when $\overline{K} = 0$). The messages in a commutative set can be processed in any order. The availability of such a knowledge of application level commutativity to the data access protocol may be exploited by relaxing the order of processing client requests at various replicas. To allow a relaxed ordering, the request messages may be causally ordered by clients. A replica basically processes messages in the sequence established by the causal order. So consistency is enforceable at the occurrence of stable points in the computation (c.f. section 5.1).

In the communication structure, the clients and the server replicas are organized into a group RPC_GRP. The clients have front-end managers which generate an ordering of the requests based on the knowledge available and broadcast the message using **O_Send**. The manager keeps track of the occurrence of commutative and non-commutative operations, and generates message labels along with the ordering. The message graph generated by the manager is same as that observed by all the replicas in the group. The generality of protocol rests in our capturing of transition-preserving operations also as commutative operations.

When the data access protocol is realized with **O_Send**, a replica may process the request messages, in the sequence in which they are received. The code skeleton of the base protocol may be realized as follows.

client()

$Ncid := 0$; /* sequence numbers of non-commutative */
$\{Cid\} := \phi$; /* and commutative messages */
forever
  wait for control event to generate request for operation;
  designate a replica as primary in *rqst* message;
  if (operation is non-commutative)
    assign next higher $Ncid$ to message;
    if ($\{Cid\} = \phi$)
      O_Send(*rqst*,RPC_GRP,*Occurs_After*($Ncid - 1$));
    else
      O_Send(*rqst*,RPC_GRP,*Occurs_After*($\wedge\{Cid\}$));
      $\{Cid\} := \phi$;
  if (operation is commutative)
    assign next higher Cid to message;
    O_Send(*rqst*,RPC_GRP,*Occurs_After*($Ncid - 1$));
    insert id from Msg in $\{Cid\}$.

If the requested operation is commutative, it is ordered after the last non-commutative message $Ncid_{r-1}$. $\{Cid\}_r$ is a set of commutative messages that occurred after $Ncid_{r-1}$. The next non-commutative message in sequence is ordered to occur after all the messages in $\{Cid\}_r$. This is realized as: $Ncid_{r-1} \succ \|\{Cid\}_r \succ Ncid_{r+1}$. The above replicated data access protocol is flexible, as it uses the explicit message orders on the client messages using **O_Send** primitive. This produces the reproducible graph in the computation, and lets us define the consistency at a higher level. The protocol can be embedded into distributed service implementations.

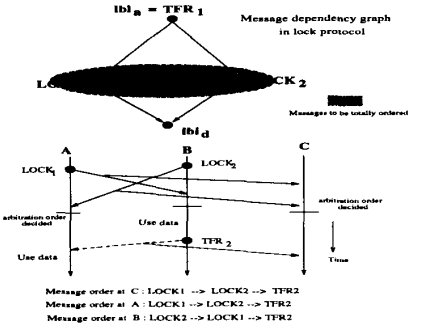

Figure 5: An arbitration protocol using total order

6.2 Use of total ordering of messages

As described in section 5.2, total order is useful for applications in which messages are generated spontaneously. Consider a distributed shared data access protocol [3], which uses LOCK messages for decentralized arbitration of data access. The access permission on a data item is obtained by acquiring a lock associated with that item. Such LOCK messages may be generated spontaneously in the computation, so we need a total order imposed on the LOCK messages and a deterministic arbitration algorithm which selects the next lock holder.

Each member a_i maintains a list of LOCK messages (*lock_rqst*) that it receives. On receiving specific predetermined number of LOCK messages, each member executes an arbitration algorithm. Since the algorithm is deterministic, all the members choose the same next lock holder, thereby ensuring consensus among members. A member a_i generates a LOCK request as follows :

A_Send($[LOCK, i, S]$, ...,
  Occurs_After($[TFR, 1, S-1] \wedge \ldots \wedge [TFR, M, S-1]$)),

where TFR advises transfer of lock and S indicates the arbitration cycle. When a current holder, say a_j, has completed page access, it broadcasts a TFR message to transfer lock to the next member in the arbitration sequence :

A_Send($[TFR, l, S]$, ...,
  Occurs_After($[LOCK, 1, S] \wedge \ldots \wedge [LOCK, j, S]$)).

After the last member in the arbitration sequence has transferred the lock, the next lock acquisition cycle (S+1) begins. A scenario is illustrated in Figure 5.

Thus, spontaneous actions in the application appear as concurrent messages and are totally ordered in our model.

7 Conclusions

The paper developed a generalized model that captures the interactions between the consistency requirements of a shared data accessed by a set of entities implementing a distributed application, and the causal ordering of messages exchanged between the entities to access the data. The model is more flexible and extensible for integration into a computational model than the traditional framework of transactional serializability that achieves consistency of the data by generating a serial schedule of transactions accessing the data.

In many existing models of consistency based on message ordering, the ordering semantics is inseparably buried into the problem-specific computations. This often limits the flexibility with which message ordering constraints can be generated to meet problem-specific consistency requirements. The model developed in this paper overcomes this limitation by providing a causal broadcasting construct that allows causal ordering constraints among messages to be explicitly specified by application entities and be propagated across various entities. Since causal relationships depict a stable information in the application and are reproducible across different execution instances, application entities can reach agreement on data values at meaningful points of message exchanges in the computation based on observing the causal relationships. These points, referred to as stable points in the computation, can be detected locally by each entity so that it can access a consistent copy of the data. The agreement protocols that use this model basically need to detect the occurrence of stable points and take local actions on the data. Such protocols reach agreement without requiring separate message exchanges across entities (a 'virtually synchronous execution' [2] at a higher message granularity). Where causality information is not completely available, other execution support mechanisms such as total ordering of messages and use of application-specific protocols are resorted to.

The paper also discussed our **O_Send** primitive for causal broadcasting. To gain deeper insight into our model, we studied the realization of data access protocols using the knowledge of application semantics (such as commutativity of operations on data).

Overall, the contribution of the paper is in relating the consistency of distributed shared data to causal ordering of data access messages. The establishment of this relationship allows solving the data consistency related problems within a generalized framework of executing agreement protocols embedded in the message communication system. We believe that such a 'down-shifting' of data access protocols from the application to a canonical communication system can result in improved asynchronism in the execution and more flexibility in the construction of data access protocols. The model we proposed can be incorporated in distributed programming primitives for use by various applications.

References

[1] R. Ladin, B. Liskov and B. Shrira. **Lazy replication: Exploiting the Semantics of Distributed Services.** *8-th Symposium on Principles of Distributed Computing*, ACM SIGOPS-SIGACT, Aug. 1990.

[2] K. P. Birman and T. Joseph. **Exploiting Virtual Synchrony in Distributed Systems.** In Proc. of *11-th Symposium on Operating Systems Principles*, ACM SIGOPS, pp.123-138, Nov. 1987.

[3] K. Ravindran and S. Samdarshi. **A Flexible Causal Broadcast Communication Interface for Distributed Applications.** *Journal of Parallel and Distributed Computing Systems*, vol. 16, no. 2, pp. 134-157, Academic Press Publ. Co., Oct. 1992.

[4] K. Ravindran and S. T. Chanson. **Relaxed Consistency of Shared State in Distributed Servers.** *Advances in Distributed Systems: Concepts and Design*, IEEE Computer Society Press (eds. T.L Casavant and M. Singhal), Feb. 1992.

[5] M. Ahamad, P. W. Hutto and R. John. **Implementing and Programming Causal Distributed Shared memory.** In Proc. of *11th International Conf. on Distributed Computing Systems*, IEEE-CS, Arlington (TX), pp.274-281, May 1991.

[6] L. Lamport. **Time, Clocks and the Ordering of Events in a Distributed System.** *Communications of the ACM*, Vol.21, No.7, pp. 558-565, July 1978.

[7] K. Birman, A. Schiper and P. Stephenson. **Lightweight Causal and Atomic Group Multicast.** *ACM Transactions on Computer Systems*, 9(3):272–314, Aug. 1991.

[8] L. L. Peterson, N. C. Buchholz and R. D. Schlichting. **Preserving and Using Context Information in Interprocess Communication.** *ACM Transactions on Computer Systems*, 7(3):217–246, Aug. 1989.

[9] D. R. Cheriton and D. Skeen. **Understanding the Limitations of Causally and Totally Ordered Communication.** In Proc. *Symp. on Operating Systems Principles*, ACM SIGOPS, Dec. 1993.

[10] F. B. Schneider. **Implementing Fault-tolerant Services Using the State Machine Approach: a Tutorial.** *ACM Computing Surveys*, Vol.22, no. 4, pp.299-319, Dec. 1990.

[11] K. Ravindran and B. Prasad. **Communication Structures and Paradigms for Distributed Conferencing Applications.** In Proc. of *12-th International Conference on Distributed Computing Systems*, IEEE-CS, Yokohama (Japan), June 1992.

[12] R. V. Renesse. **Causal Controversy at Le Mont St.-Michel.** *Operating Systems Review*, vol.27, no.3, pp.44-53, April 1993.