Characterizing Multicast Orderings using Concurrency Control Theory

P.A. Jensen N.R. Soparkar A.G. Mathur

Electrical Engineering & Computer Science
The University of Michigan
Ann Arbor, MI 48109-2122 USA
email:{pjensen,soparkar,mathur}@eecs.umich.edu

Abstract

Coordinating distributed executions is achieved by two widely used approaches: process groups and transactions. Typically, the two represent a trade-off in terms of the degrees of consistency and performance. By applying transaction concurrency control techniques to characterize and design process group multicast orderings, we aim to provide aspects of both ends of the trade-off. In particular, we propose a framework in which each message multicast is regarded as a transaction. Appropriate message ordering protocols are devised and shown to be correct using a variant of concurrency control theory. Also, we are able to incorporate certain aspects of application semantics for which existing process group approaches are inadequate. Finally, our framework provides a means to characterize the performance of orderings to allow a comparison of different ordering protocols.

1 Introduction

Distributed applications typically require that the largely autonomous executions at the separate sites be coordinated with one another in some way. For this purpose, there are two widely used approaches: the process group model [6, 4] and the transaction model [1]. The process group model is characterized by modeling the distributed system as a collection of processes that communicate by sending (multicasting) messages. Techniques are provided for ordering concurrent messages as well as ensuring atomicity of message delivery in the presence of process and link failures, and changes in group membership. The transaction model is characterized by operations being grouped into units called transactions, and techniques are provided for ordering concurrent transactions and ensuring their atomicity. In both models there is a need to order concurrent events; in the case of process groups it is the message multicasts that need to be ordered, while in the case of transactions it is the concurrently executing transactions that need to be ordered. Furthermore, in both models there is a need to ensure atomicity of these concurrent events in the presence of failures.

Some applications are better coordinated by group multicast techniques whereas others are better suited to transactions (e.g., see [2, 5]). Generally, the multicast approach is more efficient and provides adequate consistency criteria for some applications, and transactional systems are better suited to cases where a high degree of consistency criteria are needed. However, there are numerous other distributed applications which could benefit from a good meld of the performance and consistency criteria (e.g., see [2, 5]). To this end, more efficient multicast techniques based on applications semantics and advanced transaction models have been proposed. In this paper, we examine the former approach with respect to message ordering, and to do so, we use techniques prevalent in the latter.

We develop a framework to unify techniques for ordering concurrent message delivery with the theory for ordering transactions using concurrency control. Our framework is based on a novel adaptation of database concurrency control theory to message multicasts. Each message multicast is viewed as a transaction. The message delivery module at each site delivers the message multicasts to the application after enforcing the appropriate message ordering in a manner similar to a database scheduler ordering operations for concurrent transactions.

Essentially, our framework provides a better understanding of the manner in which consistency requirements for distributed applications are maintained. Since it is difficult to state consistency requirements for different applications explicitly (as is the case in transaction systems), we begin with an acceptable, sequential message ordering which is regarded as being correct by definition. Thereafter, based on the manner in which certain events can commute in a distributed history, we exhibit the correctness of other concurrent, non-sequential histories. The commutativity of events is to be derived from application semantics, and is linked to non-conflicting actions in transaction concurrency control theory. Note that our approach primarily helps in ratifying larger classes of message orderings as being correct, thereby increasing the potential for improved performance.

We consider variations of standard message orderings to exhibit our approach in incorporating (hitherto difficult [5, 12]) application semantics. We describe protocols to effect these orderings, and using the theoretical development of our approach, we explain how they meet the required correctness criteria. Furthermore, we consider a qualitative approach based on a characterization of delivery delay to measure the performance for such multicast orderings. The development of our approach closely parallels the theory of database transaction concurrency control, and it becomes more apparent that the similarity to message multicast orderings is more than coincidental.

The remainder of our paper is organized as follows. In Section 2 we describe related work, and follow-up in Section 3 with the rationale for applying concurrency control to message multicasts. We present our general framework for capturing message ordering, and we explain how it is applied for several orderings in Section 4. We demonstrate how to handle the correctness issues for multicasts in section 5. In Section 6 we examine protocols for effecting various multicast, and discuss qualitative performance issues in Section 7. Finally, we present our conclusions in Section 8.

2 Related Work

Cheriton and Skeen [5] discuss the kinds of ordering constraints that are appropriate for a variety of distributed application classes. They argue for using semantic information in specifying these various ordering. Our framework is adequate for a wide range of ordering constraints, and also allows one to use certain application-specific semantics in specifying these constraints. Other aspects of orderings such as false causality and hidden channels [5, 2] can also be captured within our framework. Essentially both of these anomalies can be alleviated by defining appropriate conflict relations.

Schmuck [13] analyzed message ordering protocols, and also used the notion of conflicts between events. However their approach is different in that it involves finding an appropriate *linearization operator*, which maps partially ordered sets of events to totally ordered histories. Thereafter, it becomes simple to prove the correctness of the resulting histories. In our approach, message multicasts are viewed as transactions, and the correctness of the orderings are shown using an adaptation of serializability theory from database concurrency control. Linearization is a very specific approach of concurrency control theory.

Work on managing the consistency of replicas in a distributed environment is also related to our work. Many replication schemes are based on the state-machine approach of Schneider [14], which can be implemented using totally ordered multicasts. Replication schemes based on weaker message orderings have been proposed, (e.g., see Birman and Joseph [3, 4], Ladin et al [7], Mishra et al [9]). Such weaker orderings can be represented within our framework.

3 Rationale for Concurrency Theory

In order to reason about consistency or correctness requirements of distributed applications, it is generally the case that application semantics must be understood rigorously. However, these requirements are difficult to state explicitly, especially if they are to be used to order message multicasts. Below, we explain our approach to handle this problem, delineate the overall rationale for our approach, and provide some preliminary definitions.

3.1 Handling Consistency Issues

First, we assume that a set of multicast messages, if effected sequentially, will represent a consistent sequence of state changes in the distributed application environment. That is, we make the assumption that each completed multicast represents a transition of the distributed system from one consistent state to another. Thereby, we avoid the problem of dealing explicitly with application state consistency requirements — at least with regard to message multicasts.

Second, we identify certain "conflict" relations between the multicast primitives. These relations indicate which events are disallowed from being freely re-ordered with respect to one another (in order to generate interleaved, concurrent executions among the messages). Intuitively, two events conflict if the order of their occurrence is significant with respect to consistency considerations. For instance, for FIFO ordered message multicasts (see definition in Section 4.1), two receiving events between messages at a given site may be regarded as conflicting if the corresponding send events occur at the same site (whereas they would be non-conflicting otherwise). Note that these conflict relations are either drawn from particular message orderings (e.g., FIFO and Causal), or more generally, from the application semantics.

Third, we regard those histories of events (with respect to the multicast message primitives) in the distributed system that arise as a consequence of commuting (with every site) the events that are non-conflicting. That is, a given concurrent message multicast history may be regarded as being correct (or preserving consistency) whose events can be re-ordered (without affecting the ordering of conflicting events) to result in a serial history. In effect, we are using concurrency control theory.

Our development follows concurrency control theory closely (e.g., see [1]). In particular, each message is treated as a transaction, the global history is treated as a distributed schedule, and techniques similar to serializability theory are applied in our approach. Below we describe send and the delivery modules at the constituent processes that function as a distributed scheduler to order the message "transactions."

3.2 Basic Definitions

We start with a few preliminary descriptions. Our model for a distributed computing system assumes that there are no failures — although, the processes in the system are assumed to execute

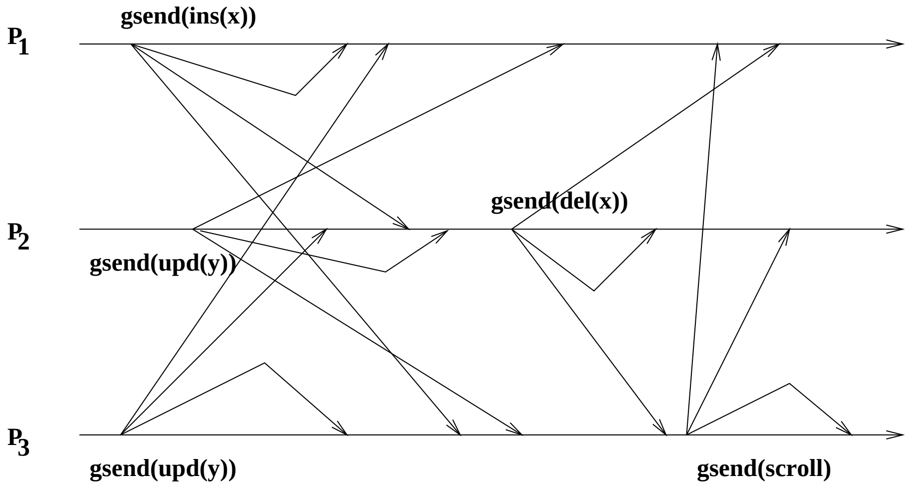

Figure 1: Example illustrating the gsend events in a global history

asynchronously, and the physical messages may be delayed or re-ordered.

The set of processes in the system is denoted by $P = \{P_1, P_2, ...\}$, and the execution of a process is modeled as a sequence of *events*. An event may be any one of a *global send*, *receive*, or a *local* event. A *global send* event, *gsend*, results in a physical message being sent (i.e., multicast) to all the other processes. A *receive* event, *rcv*, is executed by a process to receive an incoming physical message. A *local event* is executed by a process *locally*, and it does not involve the other processes. A *global history* H is the "happened-before relation" [8] on the process events which we denote with the symbol $\rightarrow$.

Consider a groupware application that allows for sharing of documents. The document being edited is replicated at each user site. At each site the document is displayed in a window, and since the interface is to be kept identical across the sites (to allow synchronous collaboration), the window is also, in a sense, replicated. For ease of exposition, let the document be a *set* of objects x, y, z, etc., with three operations permitted: insert, update, and delete. Assume that there is one operation permitted on the window: the scroll. As shown in Figure 1, consider three processes P_1, P_2, and P_3 participating in a collaborative session. Process P_1 performs a $gsend(ins(x))$ event which results in a message containing the operation $ins(x)$ being multicast to the other processes. The message is then received, delivered, and the associated operation is executed at each of the processes. The other $gsend$ events result in a similar sequence of events. The local histories correspond to the horizontal lines together with the constituent events, and the global history corresponds to the entire diagram.

We use the transaction-oriented concept of conflicts among events to capture the semantic information about orderings of application operations. Intuitively, two events conflict if the order of their execution is important. Two events can commute in terms of their execution if they do not conflict. For example, in the groupware application considered above, the pairwise conflict relations are as follows. Two inserts, two updates, two deletes, an insert and an update, an insert and a delete, and an update and a delete, conflict with each other if they apply to the same object. Two scroll operations conflict with each other, but not with the update, insert or delete operations. In Figure 1, $ins(x)$ and $del(x)$, and the two $upd(y)$ operations conflict with each other, but the $ins(x)$ and $upd(y)$, and $del(x)$ and $upd(y))$ do not. Also, the *scroll* operation does not conflict with any of the other operations. In consequence, the corresponding message events are mandated as either

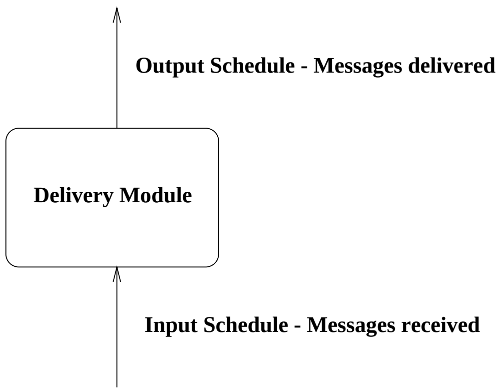

Figure 2: The Delivery Module

conflicting or non-conflicting.

As messages from various processes in the system are received at a process, the messages are delivered to the applications such that the execution of conflicting operations results in consistent states. In order to ensure this, it may be necessary to delay some messages until certain other messages have been delivered. Such decisions are taken by a local *delivery module* at each process as shown in Figure 2. The input to a delivery module is the sequence of messages as received, and the output of the delivery module is in an appropriate order.

4 Conflict Relations for Message Events

In a similar manner in which transactions use a conflict relation to capture commutativity of operations, we are using a conflict relation on messages events to define how these events are allowed to commute. These conflict relations, along with the concept of *conflict equivalence*, can be used to define a class of histories. For our purposes, this class consists of all the permutations of a given class (in this case, a *serial history*) achieved by commuting message events which do not conflict. These concepts are described further in Section 5.

The purpose for this framework is to provide a mechanism to reason about classes of message orderings. The properties for FIFO, Causal, Total are well known already, however, we wish to consider other orderings as well. We discuss the framework in the remainder of this section as follows. First, we present the conflict relations that capture the incidental orderings, FIFO, Causal, Total. Then, we introduce new classes of orderings and show the conflict relations to capture these orderings. Finally, we explain further concepts of the framework (i.e., serial histories, conflict equivalence, and serializability graphs) which we use to reason about classes of message orderings.

4.1 Incidental Orderings

We use the term *incidental* to refer to classes of message orderings whose conflict relations only rely upon incidents of communication among the processes (see [5]) , and their relationships with respect to the happened-before relation. These are distinguished from the classes of orderings we discuss following which use *semantic* information where the conflict relations are based on information which the messages themselves contain. We consider three well-known incidental orderings here, and for each we present a predicate (commonly found in the literature) which defines the ordering class, and the conflict relationships which will subsequently be shown to define the same class.

- *FIFO Order:* Messages are delivered in sender-based FIFO order,

$$gsend_i(m) \to gsend_i(m') \Rightarrow rcv_r(m,i) \to rcv_r(m',i)$$

  A conflict relation on messages that captures FIFO delivery is:

  1. $rcv_j(m,i)$ conflicts with $rcv_j(m',i)$.
  2. $gsend_i(m)$ conflicts with $gsend_i(m')$.

- *Causal Order:* Messages are delivered in causal order,

$$gsend_i(m) \to gsend_j(m') \Rightarrow rcv_r(m,i) \to rcv_r(m',j)$$

  Note that if $i = j$, then we have FIFO ordering; therefore, causal delivery implies FIFO delivery.

  The conflict relation on messages that captures causal delivery is:

  1. $rcv_r(m,i)$ conflicts with $rcv_r(m',j)$ iff $gsend_i(m) \to gsend_j(m')$ or $gsend_j(m') \to gsend_i(m)$.
  2. $gsend_i(m)$ conflicts with $gsend_j(m')$ iff $gsend_i(m) \to gsend_j(m')$ or $gsend_j(m') \to gsend_i(m)$.

- *Total Order:* The *total ordering* of messages ensures that messages are delivered in the same order at all processes,

$$rcv_r(m,i) \to rcv_r(m',j) \Rightarrow rcv_k(m,i) \to rcv_k(m',j)$$

  The conflict relation on messages that captures total delivery is:

  1. $rcv_r(m,i)$ conflicts with $rcv_r(m',j)$

4.2 Semantic Incidental Orderings

We describe three classes of multicast orderings, *semantic FIFO*, *semantic causal*, and *semantic total*, that we obtain by augmenting the orderings FIFO, causal, and total respectively with semantic information. We augment these incidental orderings with application-level semantic information by taking into account the constituents of each message by utilizing a conflict relation derived from the contents of the messages. This results in the classes of orderings that we term *semantic incidental orderings*.

The conflict relations are the same in each case as the associated incidental ordering except for an addition of a semantic condition. This addition is a conflict relation on messages, as opposed to the conflict relation on the incidents of messages, which could represent, for example, the commutativity of the operations contained in the messages. In the following we use $(type(m), type(m')) \in Con$ to represent this conflict relation, where *type* is simply a function which is some indicator of the contents of the message, and Con is the conflict relation on the types of messages.

The relation Con and function *type* are unspecified and in general they derive from application semantics. As examples of this, we describe two situations where the semantics lead to different relations. In the first example we consider application data values carried by particular message, and in the second we consider particular application operations associated with messages. Note also that it may be desirable to use different protocols for different Con relations and $type$ functions. As is evident below, there are protocols which work for the first example, yet do not work for the second example.

For data, consider a situation where each message carries an update operation for only one particular data item X, Y, Z (each data item is replicated across all sites). Furthermore, assume that we are not concerned with the order of operations among the different data items (e.g., updates of X are allowed to commute with updates for Y). Also, in order to preserve consistency, we require that updates for a particular data item are executed in the same order at each site. In this situation we could use semantic total ordering where the type function simply returns a value indicating what data item the message is carrying, and the relation Con is defined such that $(type1, type2) \in Con$ iff $type1 = type2$.

For operations, consider an application having one data item X (replicated at all sites) and three operations *increment*, *decrement*, and *update*. In this case, it may be acceptable to allow *increment* and *decrement* to commute, but *update* should not commute with *increment* or *decrement*. Assume that each message carries only one operation. Then, as for the previous example, we could use semantic total ordering in this situation to preserve the consistency of X among the sites. In this case $type$ simply returns i, d, or u to indicate that a message contains *increment*, *decrement*, *update* respectively, and $Con = (u, u), (u, i), (u, d)$.

- *Semantic FIFO Order:*

  The *semantic FIFO ordering* ensures that messages that are sent in FIFO order *and* which conflict with each other, are delivered in FIFO order:

$$(gsend_p(m) \rightarrow gsend_p(m')) \wedge ((type(m), type(m')) \in Con) \Rightarrow$$

$$rcv_q(m, p) \rightarrow rcv_q(m', p)$$

  The conflict relation for semantic FIFO is:

  1. $rcv_i(m, j)$ conflicts with $rcv_i(m', j)$ iff $(type(m), type(m')) \in Con$.
  2. $gsend_i(m)$ conflicts with $gsend_i(m')$ iff $(type(m), type(m')) \in Con$.

  For example, in the scenario illustrated in Figure 1, the $ins(y)$ and $del(x)$ operations issued by process P_2 are sent FIFO, but they need not be delivered in FIFO order since the two operations do not conflict with each other. The assumption here is that the messages invoke the corresponding operations.

- *Semantic Causal Order:*

  The *semantic causal ordering* ensures that operations that are sent in causal order *and* which conflict with each other, are delivered in in causal order:

$$(gsend_i(m) \rightarrow gsend_j(m')) \wedge ((type(m), type(m')) \in Con) \Rightarrow$$

$$rcv_k(m, i) \rightarrow rcv_k(m', j)$$

  The conflict relation on messages that captures causal delivery is:

1. $rcv_i(m,j)$ conflicts with $rcv_i(m',k)$ iff $(type(m), type(m')) \in Con$ and $(gsend_j(m) \rightarrow gsend_k(m')$ or $gsend_k(m') \rightarrow gsend_j(m))$.
2. $gsend_i(m)$ conflicts with $gsend_j(m')$ iff $(type(m), type(m')) \in Con$ and $(gsend_i(m) \rightarrow gsend_j(m')$ or $gsend_j(m') \rightarrow gsend_i(m))$.

For example, in Figure 1, the operation $ins(x)$ causally precedes $del(x)$, and since the two operations conflict with each other, they need to be delivered in the same causal order (i.e., $ins(x)$ before $del(x)$) at each process. On the other hand, the operation $del(x)$ causally precedes $scroll$, but since the two operations do not conflict with each other, they can be delivered in either order at the processes.

- *Semantic Total Order:*

  The *semantic total ordering* ensures that operations that conflict with each other are delivered in the same order at all processes:

$$(rcv_i(m,j) \rightarrow rcv_i(m',k)) \wedge ((type(m), type(m')) \in Con) \Rightarrow$$

$$rcv_r(m,j) \rightarrow rcv_r(m',k)$$

  The conflict relation on messages that captures total delivery is:

  1. $rcv_i(m,j)$ conflicts with $rcv_i(m',k)$ iff $(type(m), type(m')) \in Con$.

  For example, in the scenario illustrated in Figure 1, the two update operations performed concurrently by process P_2 and P_3 need to be ordered identically at all processes since they conflict with each other, but these need not be ordered identically with respect to the other operations (such as $ins(x)$, $del(x)$, $scroll()$).

Note that the incidental orderings are special cases of their corresponding semantic incidental orderings. In general, even the semantic incidental orderings can be regarded as being special cases of the entire class of *explicit consistency* orderings which, without being defined, may be regarded as all those orderings which maintain the consistency of the applications being considered.

5 Correctness of the Orderings

Now, we develop our framework further in order to prove properties related to the multicast ordering classes. This will be useful first, to show equivalence between conflict relations for the incidental orderings and the well-known definitions, and subsequently for discussions on correctness of protocols (Section 6) and performance (Section 7). For simplicity of our discussion, we regard each global history to be *complete* in that for every message in the history, all the events of each message are included in the history.

A global history is said to be a *serial history*, with respect to the *gsend* and *rcv* events, if each individual message is contained within an interval of time as measured by a physical global clock, and distinct messages are contained in disjoint time intervals. This is illustrated for two messages m_i and m_j in Figure 3; a global clock tick is assumed to occur between the dotted ovals.

Given a set of conflict relations M, two histories H_1 and H_2 are said to be *conflict equivalent*, with respect to M, if they contain the same set of events and any two conflicting events o and p

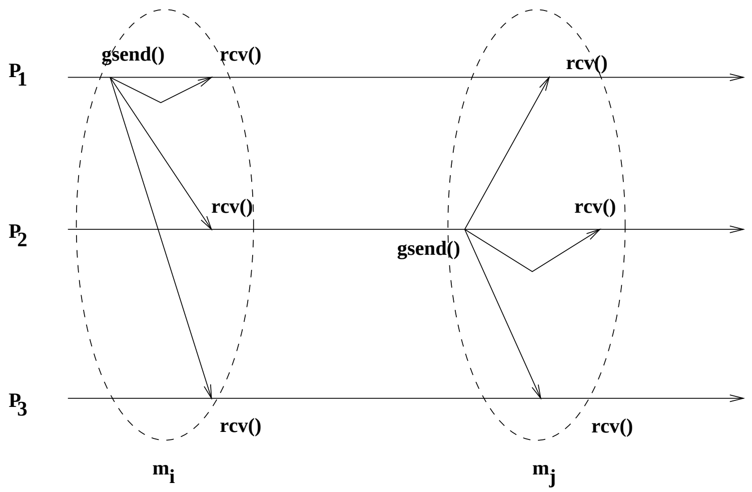

Figure 3: Viewing a message multicast as a transaction

are ordered the same way in H_1 and H_2 (i.e., $o \rightarrow p$ in H_1 iff $o \rightarrow p$ in H_2). We use the concept of equivalence to define more precisely the class of message orderings corresponding to a given set of conflict relations, and to prove a theorem which is useful for establishing correctness properties for orderings. An *allowable* global history H is one where, for a set of conflict relations M, H is is conflict equivalent with respect to M, to a serial history. Given that a serial history is "correct", any allowable global history must be conflict equivalent to a serial global history. For a given history H, we will find it useful to define a *precedence graph*, $PG(H)$, which has the messages as its nodes, and an edge $m \stackrel{pg}{\rightarrow} m'$, iff m has an event that happened before, and conflicts with, an event of m'.

Equivalence Theorem *For a given set of conflict relations M, a global history H is conflict equivalent wrt M to a serial history iff $PG(H)$ is acyclic.*

Proof Sketch: Similar to concurrency control theory (e.g., see [11, 1]).

($\Rightarrow$) Given H is conflict equivalent to a serial history H_s. If $PG(H)$ is cyclic, then there exists two messages $m_i, m_j \in PG(H)$ such that $m_i \stackrel{pg^*}{\rightarrow} m_j \stackrel{pg}{\rightarrow} m_i$. Having $m_i \stackrel{pg^*}{\rightarrow} m_j$ implies m_j happened before m_i in H_s. Having $m_j \stackrel{pg}{\rightarrow} m_i$ implies m_i happened before m_j in H_s. This is contradictory, therefore $PG(H)$ must be acyclic.

($\Leftarrow$) Given $PG(H)$ is acyclic. Let $m_1, m_2, ...m_n$ represent the messages present in H. There is a topological sort for $PG(H)$, $m_{i_1}, m_{i_2}, ...m_{i_n}$ such that for any two messages where $m_j \stackrel{pg}{\rightarrow} m_k$ is in $PG(H)$, m_j appears before m_k in the sort. Let H_s be a serial history where $H_s = m_{i_1} m_{i_2} ... m_{i_n}$. Take any two conflicting events $o_j \in m_j, o_k \in m_k$ where $m_j \stackrel{pg}{\rightarrow} m_k$ in $PG(H)$. Clearly, we must have $o_j \rightarrow o_k$ in both H and H_s. Therefore H is conflict equivalent to serial history H_s. □

Corollary 1. *For a global history H and a set of conflict relations M, H is in the class of orderings allowed by M iff $PG(H)$ is acyclic.*

Proof Sketch: Follows directly from Equivalence Theorem and definitions. □

As an illustration for exhibiting correctness, we demonstrate for the case of FIFO, that the class of message orderings defined by the conflict relations is equivalent to the class of orderings allowed by the known definition. The basic idea is to show, for any history H, the acyclicity of $PG(H)$ implies, and is implied by, the generally accepted multicast order definitions. First, let $PG(H)$ be acyclic, and consider two send events such that $gsend_i(m) \rightarrow gsend_i(m')$. Any two events $rcv_j(m,i)$ and $rcv_j(m',i)$ conflict, and $rcv_j(m,i) \rightarrow rcv_j(m',i)$ must hold for an acyclic $PG(H)$. Therefore, the FIFO definition holds: $gsend_i(m) \rightarrow gsend_i(m') \Rightarrow rcv_j(m,i) \rightarrow rcv_j(m',i)$. For the converse, assume the FIFO ordering definition holds for H. Now, if $PG(H)$ is cyclic, let $m_1 \stackrel{pg^*}{\rightarrow} m_2 \stackrel{pg}{\rightarrow} m_1$ be a cycle in $PG(H)$. Using the definition of FIFO ordering, it is seen that an edge $m \stackrel{pg}{\rightarrow} m'$ in $PG(H)$ implies $gsend(m) \rightarrow gsend(m')$ in H. That would require $gsend(m_1) \rightarrow gsend(m_2) \rightarrow gsend(m_1)$, which is impossible since $\rightarrow$ is a partial order. Therefore, $PG(H)$ must be acyclic.

6 Multicast Ordering Protocols

We now discuss protocols for the message ordering classes described in the previous section. To show that a particular protocol ensures the corresponding ordering of messages, we use our framework as follows. We prove that a global history that evolves under the protocol will result in an acyclic PG with respect to the corresponding message order class, thereby implying the correctness of the protocol (by Corollary 1). Note that we assume that each site knows the conflict relations.

Protocols which ensure FIFO, Causal, and Total order message delivery are well known, and as an illustration, we use our framework to prove the correctness of a FIFO protocol. A simple protocol which ensures FIFO delivery is as follows. The sender timestamps each message using a sequence number, and then increments the sequence number. The recipients of messages deliver only in order of the sequence numbers. In order to show that this protocol results in an acyclic PG_{FIFO}, assume to the contrary, i.e., assume that there is a cycle in PG_{FIFO}. An edge from message node m to m' in PG_{FIFO} implies that $gsend(m) \rightarrow gsend(m')$, by the conflict definitions and the fact that the protocol timestamps ensure that for any two messages, $rcv_i(m) \rightarrow rcv_i(m') \Leftrightarrow gsend(m) \rightarrow gsend(m')$. The existence of a cycle in PG_{FIFO}, $m_1 \stackrel{pg^*}{\rightarrow} m_2 \stackrel{pg}{\rightarrow} m_1$ implies that $gsend(m_1) \rightarrow gsend(m_2) \rightarrow gsend(m_1)$, which is impossible considering that the happened-before relation is a partial order. Therefore, the protocol ensures that PG_{FIFO} is acyclic.

Now, we turn our attention to semantic incidental orderings. Consider first the orderings permitted with consideration given to affected data items. For each of the timestamp based protocols for FIFO, Causal, and Total orderings, the timestamps can be augmented with the identity for the data item in question. Moreover, for every data item, a different timestamp counter is maintained. With this, we can ensure correct orderings for the case where each message pertains to a particular data item. Events for messages which pertain to the same data item are not allowed to commute, whereas events pertaining to different data items may commute. Our framework facilitates proving that the protocols would work correctly in the manner shown above for the case of FIFO ordering. In this paper, we do not provide protocols for the cases where a message may pertain to more than a single data item. Note that while the protocols for the simple case of a single data item per message is similar to maintaining separate message groups, it is not easy to characterize the more complex cases using other traditional techniques.

Now consider the cases for operation based semantic incidental orderings. We provide protocols for all three semantic incidental orderings for a subset of the conflict relations. In particular, we consider the case where the conflict relation Con among the messages is an equivalence relation, and forms equivalence classes. That is, receive events pertaining to messages within an equivalence class are not allowed to commute, whereas other receive events may do so. The protocols are essentially the same as for the case of separate data items described above. In fact, the protocols for the data item based conflict relations may be seen to be special cases of the corresponding operation based ones. Again, except as an illustrative protocol below, in this paper we do not provide protocols for more complex cases of conflict relations (e.g., those involving mixed data and operation based conflicts etc.).

Although we do not discuss more complex protocols in this paper, we provide an example of a generic semantic causal ordering multicast protocol, and argue for its correctness. The key idea of the semantic causal ordering protocol is to make the causal ordering protocol cognizant of the conflict relations. Now, causal delivery can be effected, in principle, by piggy-backing on each message the causal history of that message [15]. To ensure semantic causal ordering, the delivery of a message is delayed only if a message in its causal history has not been delivered (i.e., causal ordering) *and* the receive events at that site for the two messages conflict. The argument to show that this protocol results in an acyclic PG is similar to the case for FIFO: Assume to the contrary there is a cycle $m_1 \overset{pg^*}{\rightarrow} m_2 \overset{pg}{\rightarrow} m_1$ where $gsend_i(m_1) \rightarrow gsend_j(m_2)$ and $rcv_k(m_1, i) \rightarrow rcv_k(m_2, j)$. The protocol would not allow receives processed in this order, and $rcv_k(m_1, i)$ would be delayed until after $rcv_k(m_2, j)$. Therefore, the protocol would ensure an acyclic PG.

7 Performance Characterization

It is difficult to gauge the performance benefits of different ordering protocols quantitatively as they are affected by the particular platform, physical communication media, network topology etc.. Instead, we propose here a qualitative means to characterizing the performance of the ordering protocols. This allows comparison of the protocols in a manner that is application and operating environment independent. The characterization is based on two metrics: the number of *message phases* required to achieve the necessary ordering, and the *degree of delay* encountered by the message at the delivery modules. Furthermore, we extend our reasoning to cases where reliability issues are considered within our performance characterization.

7.1 Message Phases

The protocols described above involve message phases to procure the necessary control information in order to send a message, and also, to send the message itself. Both FIFO and causal message delivery can be accomplished using a single phase. A message, along with a "timestamp" is sent to one or more destinations. The delivery module at the destination can look at the timestamp and decide whether to deliver the message or whether it should be delayed.

Total ordering, on the other hand, requires two phases of communication. The first phase involves requesting a "timestamp" which fixes the position of that message in the total order. The second phase involves distributing the message along with the timestamp obtained to each process. The delivery modules at each process can then look at the timestamp and deliver the message in the right order. It is possible to show, using knowledge-theoretic arguments, in a manner similar to [10], that these number of message phases is minimal and it is not possible to do any better.

7.2 Delivery Delay

Consider the delivery module as a scheduler which takes a message sequence as input and produces an output message sequence such that the specified tests indicated in our protocols hold for each message that is output. We refer to the projection of the global history at each local site as

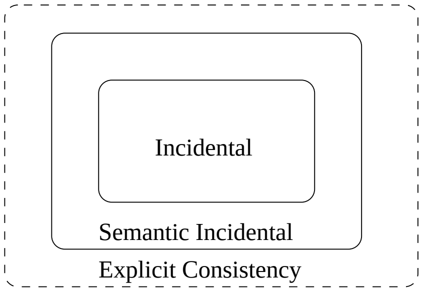

Figure 4: A general hierarchy of event ordering classes

the *schedule* at the site. The scheduler executes in an online manner in that it examines the next arriving message and delivers it if permissible. Also, a delivery module should impose the minimum delay on an arriving sequence, which is characterized by requiring that the arriving sequence be altered minimally: As in [11], for message sequences x_n and y_n of length n each, let the metric $d(x_n, y_n)$ be defined as $n - c(x_n, y_n)$, where $c(x_n, y_n)$ is the length of the longest common prefix of x_n and y_n. Let S_A be a scheduler for an acceptable class A of sequences delivered, and let I be the domain of input sequences. A scheduler for A may be regarded as a function, $S_A : I \rightarrow A$, such that $d(x, S_A(x)) = min\{d(x, y) : y \in A\}$. That is, S_A leaves sequences $A \in I$ unchanged, and the set A may be regarded as those schedules that are minimally delayed by S_A. Our characterization could use a more refined metric for schedules not in A, but our approach suffices for the coarse qualitative characterization we are providing here.

In principle, our characterization provides an indication of the availability of delivery schedulers as well (e.g., as in [11]). For a class A of message sequences, A has an efficient scheduler S_A iff the prefixes of the schedules in A can be recognized in polynomial time in the number of input messages. For, if a new message m arrives after a sequence s of messages has been output, an efficient check whether the sequence s concatenated with m is among the prefixes of A indicates whether m may be output without delay. Conversely, submitting a candidate for a prefix of the sequences in A to a scheduler S_A would produce the same sequence as an output, and that may be checked efficiently, if the candidate sequence is indeed a prefix for the sequences in A.

Note that using our characterization, the larger the class of schedules A not delayed, the "better" is the delivery scheduler. For example, a delivery scheduler for causal orderings would be better than that for FIFO orderings. Similarly, the semantic incidental delivery schedulers would be superior to the corresponding incidental ones. Obviously, the explicit consistency approaches would be better than the others — assuming that it is possible to obtain the requisite delivery schedulers. Figures 4 and 5 represent the containment relationships between the classes (where the explicit consistency class represents the most liberal schedules).

7.3 Reliability Issues

An interesting aspect of these schedulers ties-in the reliability of multicasts, the delivery delays, and the feasibility of a protocol for a particular class of schedules. Consider a distributed application designed to withstand situations where a requested multicast is not effected, and the multicast attempt is abandoned or "aborted". If it were permissible to abort any message for which all the receive events could not be processed, a protocol can be derived (in principle) for any given class of orderings by using precedence graph testing (much the same as for transactions — see [1]). The

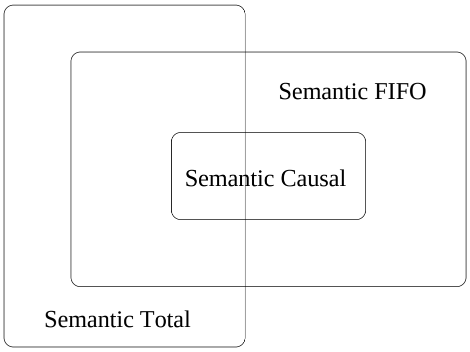

Figure 5: Relation of the three operation ordering classes to each other

idea is for delivery modules to maintain a global PG, and for each receive event, check to see whether a cycle is generated in the graph. If the graph remains acyclic, the event is executed (i.e., the message is delivered). If a cycle does result, to maintain acyclicity, aborting a message may be necessary. Of course, this is only an illustrative example — a delivery module of this nature may be highly inefficient.

Note that the concept of "atomicity" in transaction systems relates to reliability in message multicasts: a multicast system ensures reliability only if all the receive events can be executed, and thereby, all the recipients have the message delivered to them. The availability of aborts in transaction systems allows the scheduler to operate "optimistically" in that all schedules that could possibly be extended without loss of consistency are permitted; and if something goes wrong, appropriate transactions are aborted. This facilitates a larger class of schedules — much in the same way that our approach in the context of multicasts would allow.

8 Conclusions

There are two widely used models for distributed computing: the process group model and the transaction model. In both models, there is a need to order concurrent events. In the process group model concurrent message multicasts need to be ordered, while in the transaction model, concurrently executing transactions need to be ordered. We presented a framework that applies the techniques from transaction concurrency control to ordering concurrent messages in the process group model Our framework regards each message multicast as a transaction. The message delivery module at each site delivers the message multicasts to the application after enforcing the appropriate message ordering constraints in a manner similar to a concurrency control scheduler. Our framework provides easier incorporation of application semantics within the process group model, and a qualitative characterization of the performance of different orderings.

References

[1] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA, 1987.

[2] K. Birman. A Response to Cheriton and Skeen's Criticism of Causal and Totally Ordered Communication. *ACM Operating System Review*, 28(1):11–21, Jan. 1994.

[3] K. P. Birman and T. A. Joseph. Low-Cost Management of Replicated Data in Fault-Tolerant Distributed Systems. *ACM Trans. on Computer Systems*, 4(1):54–70, Feb. 1986.

[4] K. P. Birman and T. A. Joseph. Reliable Communication in the Presence of Failures. *ACM Trans. on Computer Systems*, 5(1):47–76, Feb. 1987.

[5] D. R. Cheriton and D. Skeen. Understanding the Limitations of Causally and Totally Ordered Communication. In *Proc. of the 14th ACM Symp. on Operating Systems Principles*, pages 44–57, Asheville, NC, Dec. 1993.

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

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

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

[9] S. Mishra, L. L. Peterson, and R. D. Schlichting. Implementing Fault-Tolerant Replicated Objects Using Psync. In *Proc. of IEEE 8th. Symp. on Reliable Distributed Systems*, pages 42–52, Seattle, WA, Oct. 1989.

[10] P. Panangaden and K. Taylor. Concurrent common knowledge: defining agreement for asynchronous systems. *Distributed Computing*, 6(2):73–94, 1992.

[11] C. Papadimitriou. Serializability of Concurrent Database Updates. *Journal of the ACM*, 26(4):631–653, October 1979.

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

[13] F. Schmuck. The Use of Efficient Broadcast Protocols in Asynchronous Distributed Systems. Technical Report TR CS88-928 (Ph.D. Thesis), Computer Science Dept., Cornell University, Aug. 1988.

[14] F. B. Schneider. Implementing Fault-Tolerant Services using the State-Machine Approach. *ACM Computing Surveys*, 22, Dec. 1990.

[15] R. Schwarz and F. Mattern. Detecting causal relationships in distributed computations: in search of the holy grail. *Distributed Computing*, 7:149–174, 1994.