Real-Time Causal Message Ordering In Multimedia Systems

Frank Adelstein

Mukesh Singhal

Department of Computer and Information Science

The Ohio State University

Columbus, Ohio, 43210

Abstract

In multimedia systems, not only do messages that are sent to and received by multiple sites need to have a consistent order imposed by all sites, but cause and effect relations must be maintained. Causal ordering allows the cause and effect relations of messages to be maintained. This paper presents an algorithm that insures that multimedia data with real-time deadlines are delivered to the application layer in causal order. The algorithm is designed to insure that any message that arrives at a destination site before its deadline will be delivered to the application before the message expires. In addition, by focusing on a form of causal ordering violations caused by "the triangle inequality," this algorithm has a low overhead with respect to the amount of information that must be appended to each message.

Keywords: Real-time, causal ordering, multimedia, systems, delta-causality, triangle inequality.

1. Introduction

Multimedia systems are becoming widespread [2][3][6][7]. They are characterized by the use of several different methods of presenting data, such as voice, still and motion video, music, and text. A conference call between several parties that uses both voice and video, and a playback of a recorded conversation are two examples of multimedia applications. Each type of multimedia data has different properties. For example, the bandwidth required for CD quality audio is very different from that required for full motion video, which is different from that of plain text. However, a property common to all of these data types is that they are all real-time in nature. The applications send streams of data that must be used within a time interval because the data is useless after that interval [4].

Real-time data differs from non-real time data in that it is transmitted under a time constraint. The data must be used before a deadline; after the deadline the data is of no use. In some systems data that has missed its deadline can be discarded, while in others missed deadlines cause the system to fail or even cause physical damage. Multimedia data falls into the first category, in which missed data can be discarded and only represents a degradation in the quality of the playback or presentation that should be minimized.

Problem Overview:

In multimedia applications, elements of a conversation can originate from several distinct sources. This could either be due to several physically separate participants in a multimedia conference, or several data streams being stored at different sites, such as text at one site, voice and video at another. The application will combine these different data streams into one presentation, such as the sounds and images in a conference call. Each participating site may be running its own application that combines the different streams. Ideally, each site should see all of the messages in the exact same order and have cause and effect relation maintained, so that all sites see the same consistent view of the presentation. However, the communication medium does not deliver messages in the same order at all sites, because messages from different sites travel though different routes. Therefore it is necessary for each site to impose some kind of consistent order on the messages so that the applications at the sites will have messages delivered in the same order, regardless of the order in which the messages actually arrive at the site.

The method used should insure that messages sent before other messages should be received in that order, even if they arrive out of order. This requires some information to be included in each message. Since there is no global clock in distributed systems, the information added to the messages must indicate the knowledge of other messages in the system that were sent before it. A message is said to *depend upon* other messages that were sent before it, and a message can not be delivered until all the messages that it depends upon have been delivered. This is an informal definition of causal ordering of messages. The transitive closure of this relation denotes the "transitive dependencies" or "dependency chain."

Birman-Schiper-Stephenson [1] and Schiper-Eggli-Sandoz [11] present algorithms to implement causal ordering of messages. These algorithms use the concept of logical clocks and vector timestamps [9] to provide information as to which messages have been sent before a message. Logical clocks are a good tool to capture casual ordering of messages at a logical level. However, logical clocks increment only when an event happens, such as the transmission or reception of a message. So, they have little correlation to physical time. Real-time systems require the notion of physical time because of the constraints imposed by the deadlines on the data. Therefore, while logical clocks are appropriate for causal ordering in non-real-time systems, they are not appropriate for real-time systems.

While there is no physical global clock in a distributed system, it is possible to keep all of the local clocks loosely synchronized [5][10], which means that a bound can be placed on the clock drift. In addition, the clock drift can be kept small enough so that local clocks at each site can be used for global physical time.

Multimedia data has a limited lifetime after which the data can not be used. A message that arrives out of order should be held until all messages that were sent before it have expired or are received. However, a message should not be held beyond its lifetime while waiting for messages that were sent before it, because it will be discarded anyway. We can assume that all data of the same type (e.g., video or audio) have the same life-span. So, if message *m2* is being held until message *m1* arrives, *m1* must expire before *m2* expires. After *m1* expires, there is no longer any reason to hold *m2*, because even if *m1* arrives, it will be discarded.

So, because data is only useful during a certain interval, the dependency on previous messages applies to it only for a limited interval. This time-constrained dependency is referred to as "delta-causality" [13] and is defined formally in the next section.

In addition to limiting the interval of the dependency, the number of messages in a dependency chain is limited. For every message in a dependency chain, the dependency information must pass through another site other than the source and destination. Every message induces a delay due to processing at the source and destination sites as well as propagation delay. After passing through a few sites, the delays can be comparable to the lifetime of a message. By the time the message at the end of a chain arrives, the messages at the beginning would have been processed or discarded. The "triangle inequality" is a special case where the dependency chain is of length two. Enforcement of causal dependency in this case requires much less information to be passed with each message, because of the small size of the transitive dependencies.

This paper presents an algorithm that solves the problem of delivering messages containing multimedia data with real-time constraints in causal order. It attempts to deliver messages in the same order at all sites while trying to minimize the number of messages past their deadlines that must be discarded. The paper focuses on handling "triangle inequality" causal order violations and argues that handling this case should be sufficient for most multimedia applications.

The paper is organized as follows. Section 2 presents formal definitions of terms and the problem statement. Section 3 presents the algorithm to insure that messages are delivered in order and an analysis of the overhead it imposes. Section 4 presents a proof of correctness of the algorithm. Finally, Section 5 presents conclusions.

2. Preliminaries

2.1 Definitions

The terms "delivered" and "received" are used throughout this paper. It is important to note the difference between them. A message is said to be *received* when it reaches the destination site from the communication network. It is said to be *delivered* when the communication level software passes the message on to the application software. While a message has been received at a site, it may not necessarily be deliverable to the application due to dependencies. A received message is held in a queue until all missing messages that it depends upon have arrived or expired.

The next two definitions provide a framework for relations between messages.

Definition 1: "$\longrightarrow$" ("causal dependency") is defined in [8] such that *m1* $\longrightarrow$ *m2* if and only if:

1. *m1* and *m2* are messages that were sent from the same site and *m1* was sent before *m2*., or
2. *m2* was sent by site *i* after *m1* was received by site *i*.

Note that in Figure 1, *m0* $\longrightarrow$ *m1* and *m2* $\longrightarrow$ *m3*.

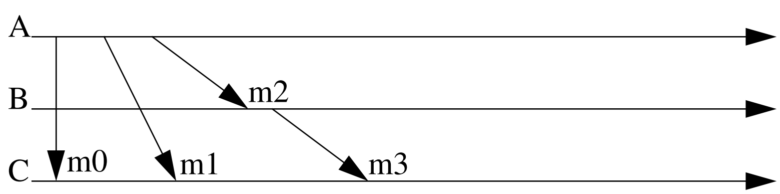

Figure 1 Causal Ordering

.

Definition 2: Causal ordering, denoted by "$\xrightarrow{*}$", is defined as the transitive closure of the "$\longrightarrow$" relation for the reception of messages at a site [11].

Causal ordering implies that there is the same causal relation between message send events and the corresponding message receive events. In Figure 1, site C receives messages in causal order, because it receives *m0* before *m1*, and *m1* before *m3*. Message *m2* "bridges the gap" of the dependency information from *m1* to *m3*. By the definition of causal ordering, there can be an arbitrary number of messages involved in a chain of dependencies, since there is no notion of a deadline or expiration time to the data. This issue is addressed subsequently.

Disparity in the speed of communication links as well as network congestion can contribute to causal order violations. The simplest form of a violation of causal ordering is due to the "triangle inequality." This occurs when a message that passes through a site takes less time to travel than one that is sent directly to the same destination. For example, in Figure 2, let X, Y, and, Z represent the time required to send a message from A to B, B to C, and A to C, respectively. If Z is greater than the sum of X and Y, then messages arrive at site C out of causal order. Figure 3 shows how the triangle inequality could cause a violation of causal ordering. A simple example of the effects of causal order violations is an audio conference in which a question is asked by one participant and is answered by another, but they are heard in the opposite order by a third participant.

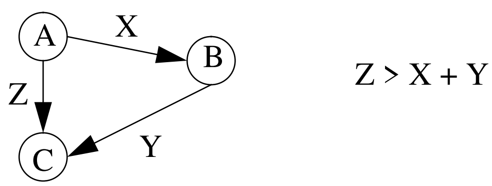

Figure 2 Triangle Inequality

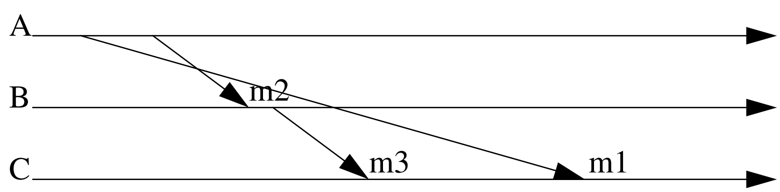

Figure 3 Causal Ordering Violated

Definition 3: "$\xrightarrow{\Delta}$" ("delta causal order") is defined in [13] such that *m1* $\xrightarrow{\Delta}$ *m2* if:

1. *m1* $\xrightarrow{*}$ *m2* AND
2. *m1* is sent at most Δ time units before *m2*.

Figure 4 shows how delta-causality can be observed even if causal ordering is not observed. The deadline of a message is defined as being delta time units after the message is sent. Because *m1* is sent more than Δ time units before *m3* is sent, there is no delta causality relation between *m1* and *m3*, so *m3* is not queued when it arrives. If *m3* is held until *m1* arrives, *m3* as well as *m1* are discarded since the data in both messages has expired. Thus, in this case the data in message *m3* is discarded along with *m1*, whereas the information in *m3* could have been used.

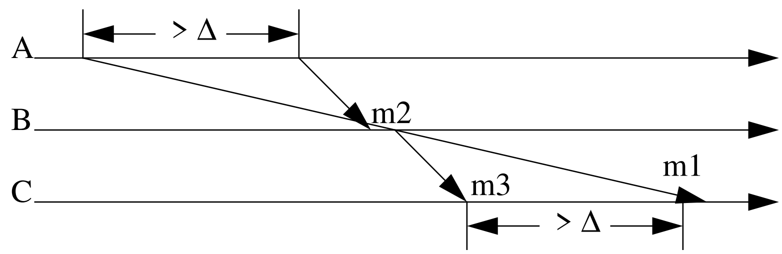

Figure 4 Delta Causal Ordering Observed (*m1* Discarded)

The causal ordering relations due to messages that have passed through many sites are not as significant, because violations in that ordering require the late message to take longer to go from its source to its destination than it takes for several messages to travel through several sites. This is why the algorithm this paper presents is designed to handle causal order violations due to the triangle inequality. In general, it should be sufficient for multimedia applications to handle only this case.

Figure 5 shows how higher order causality violations, such as a transitive relation spanning four sites, do not violate delta causality. Causal ordering forces *m4* to be held until *m1* is received, even if it is known that *m1* is past its deadline and is useless. Delta-causality does not impose that constraint however, so *m4* is delivered before its deadline, even though *m1* has not yet arrived. When *m1* eventually arrives, it is discarded.

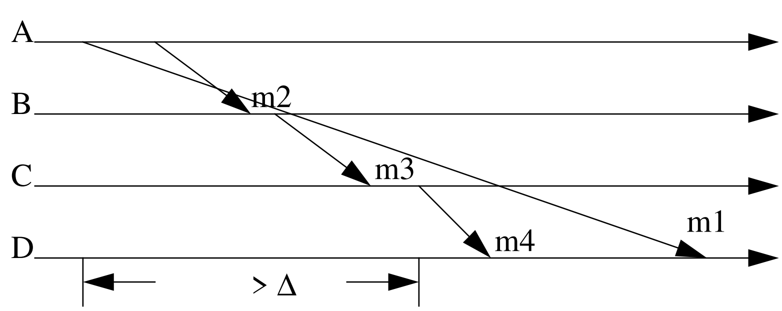

Figure 5 No Delta Causal Relation Between *m1* and *m4*

2.2 Problem Statement

Given a multimedia system with real-time data, its messages must be ordered such that all messages are delivered to the sites in the same causal order. This paper presents an algorithm that insures that multimedia messages with real-time data are delivered in delta-causal order. It handles triangle inequality violations and ignores higher order violations. It also discusses why higher order dependencies are not likely to have delta causality dependencies. In addition, it is designed to minimize the number of messages that are discarded. Once a message is received by a site, it will be delivered rather than discarded unless it arrives late. Messages with dependencies on other messages that have not arrived are held until either the missing messages arrive or the missing messages have passed their deadlines. In either case, the held message is delivered.

The algorithm initially assumes that all sites have perfectly synchronized clocks. This is used to derive the basic ideas. This assumption is relaxed later, such that all sites have loosely synchronized clocks with a clock drift bound by some known value.

3. The Algorithm

3.1 Data Structures and Notations

• Each site i has an NxN dependency matrix denoted Dep_i, where element (x, y) of the matrix represents the timestamp of the last message site x sent to site y, as far as site i knows.

• Each site has a physical clock that is constantly updated by an underlying clock synchronization algorithm and is denoted by the variable “current_time.”

• Each site maintains a priority queue of messages that have arrived but have not yet been delivered, sorted by expiration time.

• The variable “min_wait” denotes the amount of time necessary to hold a message to compensate for the network transit time. A typical value is 30-35 milliseconds for LANs.

• Δ represents the time the data in a message is useful and is defined to be < 3*min_wait. An upperbound for Δ for audio is 100 milliseconds, after which the sound quality degrades[13].

3.2 Overview

The algorithm adds two vectors, denoted vect1 and vect2, to messages. When site i sends a message to site j, the first vector, vect1, is the i^{th} row of i’s dependency matrix and denotes the timestamp of the last message site i sent to other sites (i’s vector timestamp). The second vector, vect2, is the j^{th} column of i’s dependency matrix and denotes the timestamp of the last message sent to j by other sites. Inclusion of the two vectors allows causal violations due to the triangle inequality to be detected because out of order messages will have a timestamp greater than the corresponding timestamp in the destination site's matrix. Messages that have no causal relation between them are defined as being concurrent. Concurrent messages are handled by holding all messages longer than the network latency. This insures that a message sent earlier than another but delayed due to network delays is still delivered in the proper order. After two concurrent messages whose arrivals are separated by at most min_wait have been received, their timestamp is used to determine the delivery order. Since every site that receives a message gets the same timestamp value and use the same ordering algorithm, all sites order concurrent messages in the same way.

3.3 Sending A Message From Site i To j[1]

send_message (msg, j) {

    Dep_i[i,j] = current_time
    $\forall x$ (msg.vect1[x] = Dep_i[i,x])   /* row i of dependency matrix */
    $\forall x$ (msg.vect2[x] = Dep_i[x,j])   /* column j of dependency matrix */
    Send message to site j.
}   /* end function */

3.4 Receiving A Message From Site i At Site j

receive_message (msg, j) {

    IF (msg.vect1[j] + Δ > current_time) THEN
            Discard message.   /* discard if it is past its deadline */
            RETURN
    ENDIF
    /* insure the message is not delivered until at least min_wait time units after being sent */
    wait (MAX(0, (msg.vect1[j] + min_wait - current_time)))   /* Rule 1 */
    check_and_deliver(msg, j)
}   /* end function */

check_and_deliver (msg , j) {

/* compare the message with the Dep matrix of the destination site */

1. Multicasting can be supported by multiple calls to send_message() with different destinations using the same timestamp, i.e., the model assumes point to point sends are used for multicasting.

```
    IF (dependency_exists(msg, Dep_j)) THEN                    /* Rule 2 */
            Put message msg in queue.
            RETURN
    ENDIF
            deliver_msg(msg)          /* deliver message with no dependencies */
    /* check other messages in queue and deliver any that do not have dependencies */
    /* since other messages may have been waiting for this message. */
```
$\forall x, x \in$ (message-queue) $\wedge \neg$ (*dependency_exists* $(x, Vect_i)$) :: *deliver_msg*(x)

```
}   /* end function */
```

deliver_msg (msg) {

```
    /* deliver message and merge sending sites dependency vector into local matrix */
    Remove message msg from queue.
    FOR x = 1 to N DO
            IF (msg.vect1[x] > Dep_j[i, x]) THEN               /* Rule 3 */
                    Dep_j[i, x] = msg.vect1[x]
            ENDIF
    ENDFOR
}   /* end function */
```

Boolean dependency_exists (msg, matrix) {

```
    /* a dependency is represented by a timestamp in the message that is later than */
    /* the corresponding one in the dependency matrix, provided that that timestamp */
    /* is no older than by delta (i.e., missing message has not expired) */
    FOR x = 1 to N DO
            IF ((msg.vect2[x] > matrix[x, j]) AND ((current_time - msg.vect2[x]) < Δ)
            THEN
                    RETURN true
            ENDIF
    ENDFOR
    RETURN false
}   /* end function */
```

When a message is put in the queue, an asynchronous timer is set at MAX (msg.vect2[x] + Δ) for all x. When the timer expires, the system checks if the message has any dependencies by calling the function dependency_exists(). The message is discarded if so and delivered otherwise. All other messages in the queue are checked for dependencies because undelivered messages may be delivered at that point, if the queued message depends upon the discarded or delivered message. Once the message is delivered (or discarded), the other messages on its dependency chain can be delivered.

3.5 Loosely Synchronized Clocks

For simplicity, we have assumed that the physical clocks local to each site were all perfectly synchronized. If that constraint is relaxed and the clocks are no more than ε time units out of synchronization with each other, then the only change to the algorithm is the definition of delta, which only increases the "safety margin." Algorithms exist to keep clocks synchronized within 5-10 milliseconds [5][10][13]. If ε is non-zero, then delta is defined to be the time data in a message is useful *plus* the maximum clock drift (i.e., $\Delta_{new} = \Delta_{old} + \varepsilon$). The functionality and validity of the algorithm remains, because Δ only defines the lifetime of a message. By increasing Δ, messages are not discarded as soon, but the relation between messages does not change. Even with loosely synchronized clocks, the triangle inequality problem does not change. The transitive information is still sent and the ordering is still preserved. Lemmas 1 and 2, in Section 4, are still valid with a non-zero ε.

The use of physical time clocks requires that there be an external program that periodically synchronizes clocks at the sites. The overhead of such a program is small, however, as the level of synchronization increases, the overhead increases [5] [10]. A higher level of synchronization is required to decrease the clock drift.

3.6 Discussion

Transitive dependencies with a length greater than two (i.e., the triangle inequality) are likely to have no delta-causal relation. Delta < 3 * min_wait, by the definition of delta. So a message with a transitive dependency of length three or more that arrives before a message sent directly from the source to the destination implies that the out of order message has spent at least 3 times min_wait in transit (a delay of min_wait at each site). Since this exceeds delta, the life-span of the message, the data in that message is no longer of use and the message is discarded when it finally arrives. There will be no delta causal order between it and the message at the end of the transitive chain. In Figure 5, there is no delta causal relation between *m1* and *m4*, since they are separated by more than delta time units.

Rule 1 in receive_msg() forces messages to wait at least min_wait time units after they were sent before they are delivered. But messages will not be delayed more than min_wait, even if they must wait in a queue to be received at the destination site. Once the message has spent min_wait time units in the queue, all messages that are ahead of it in the queue must have waited min_wait as well, so they will be processed before this message. Rule 1 will cause the message to wait only if min_wait time units have not elapsed since the message was sent. The delay due to processing the message by the rest of the algorithm should be small. So at most, a message can spend min_-wait time waiting to be delivered, plus whatever time it takes to process the messages ahead of it, which should be small in comparison to min_wait.

3.7 Message Size and Complexity

The message size overhead imposed by this algorithm is O(n), since every message contains two vectors of n elements, where n is the number of sites. This handles transitive dependencies of size 2. To handle larger transitive dependencies, a worst-case message overhead of $O(n^2)$ is required. Refer to Lemma 4 for a proof.

This algorithm uses physical time instead of logical time. Expired data is easily detectable since its timestamp will not be within Δ time units of the current time. Such data can then be ignored, whereas with logical clocks, this information can not be obtained. In addition, the use of physical time allows concurrent events to be consistently ordered.

Simulation can be used to analyze the effectiveness of the algorithm. The percentage of discarded packets (with respect to the total number of packets) that occur both with and without the algorithm can be compared. In addition, the number of causal ordering violations and the overhead imposed by the algorithm, both in terms of processing time and messages can be analyzed.

4. Correctness

Definition 4: "$\xrightarrow{2Hop}$" ("two-hop delta causal dependency") is defined such that *m1* $\xrightarrow{\text{2Hop}}$ *m2* if and only if:

1. $m1 \rightarrow m2$ or
2. $\exists m3 \ni m1 \rightarrow m3 \wedge m3 \rightarrow m2$ and *m1* and *m2* are sent to the same destination

   and

3. $|t_{m1} - t_{m2}| \leq \Delta$, where t_{m1} and t_{m2} are the times *m1* and *m2* were sent, respectively

In Figure 1, *m0* $\xrightarrow{\text{2Hop}}$ *m1* and *m1* $\xrightarrow{\text{2Hop}}$ *m3*. Either the messages originate at the same site or there exists a single message between those two sites. In addition, they must be sent within delta time units.

Lemma 1: The algorithm guarantees that messages are delivered in 2Hop order.

Proof: Assume *m1* $\xrightarrow{\text{2Hop}}$ *m2*. This implies that either *m1* and *m2* are sent from the same site and *m1* is sent before *m2*, or that there is one message that is one "hop" between *m1* and *m2*, such that it has *m1*'s source as its source and *m2*'s source as its destination. Refer to messages *m1* and *m3* in Figure 1. If *m1* and *m2* are sent greater than Δ time units apart, then there is no delta-causal relation between them. If *m1* arrives after *m2*, which would be more than Δ time units after *m1* was sent, then the information in *m1* is useless and *m1* would be discarded.

<u>Same Site Case</u>:

Assume *m1* and *m2* are sent from i to j. If they were sent consecutively, then the timestamp of *m1* will be stored at position (i,j) of site i's dependency matrix and this value will be sent to site j in vect1 and vect2 of *m2*. If *m2* arrives before *m1*, by Rule 2 in check_and_deliver(), *m2* will be held in a queue until either *m1* arrives or *m1* is past its deadline, implying that there is no longer a dependence on *m1*. If site i sends messages to site j between *m1* and *m2*, then it can be shown by induction that each pair of consecutive messages will be properly ordered and therefore the entire sequence will be properly ordered.

<u>One Hop Case</u>:

Assume *m1* and *m2* are sent to site k from two different sites, i and j respectively. Then by the definition of 2Hop, there must exist a message, *m3*, such that *m3* was sent from site *i* after *m1* was sent, and received by site *j* before *m2* was sent. The j^{th} entry of vect1 (i.e., vect1[j]) of *m3* will contain the time stamp for *m1*. This will be incorporated into the dependency matrix at site *j* and will be passed to site *k* as vect2[j] of *m2*, by Rule 3 of deliver_msg(). If *m2* arrives before *m1*, site *k* will detect the dependency on *m1* from Vect2 of *m2* and *m2* will be held until either *m1* arrives or *m1* is past its deadline, in which case there will no longer be a dependence between *m2* and *m1*. Therefore, *m1* and *m2* will be delivered in the proper order and the algorithm delivers messages in 2Hop order. □

Lemma 2: Messages are delivered in delta-causal order.

Proof: If the transitive dependencies among messages are of length two, then from Lemma 1 the messages are delivered in delta-causal order since 2Hop is equivalent to delta causality restricted to transitive dependencies of length two.

If the transitive dependency is of length greater than two, then since delta is defined as < 3 * min_wait, any dependency that spans more than 2 sites will take more than delta time units because a delay of min_wait is incurred at each destination, by Rule 1 of receive_msg(). Since the delay is greater than or equal to 3 * min_wait, it is also greater than delta. By the definition of delta causality, there is no delta-causal relation between that message and others, so the algorithm delivers messages in delta-causal order. □

Lemma 3: All messages that arrive before their deadlines are delivered in the same order at all sites.

Proof: If messages arrive out of order, the algorithm holds the out of order messages until they either arrive or pass their deadlines. Since all sites are running the same algorithm, all messages arriving at a site before their deadlines are delivered and the algorithm guarantees that delta-causal ordering holds by Lemma 1 and Lemma 2. The only way the sequence can be different is if a message arrives at one site before its deadline and after its deadline at another site. But this will only effect the sequence, not the ordering. The only change can be dropped messages that were not received in time. Therefore all messages that arrive before their deadlines are delivered in the same order. □

In Figure 5, there is a transitive dependency involving three messages. To preserve the dependency information, all sites must send their dependency information to any site they communicate with, since that receiving site may send a message to a site with a dependency in the dependency vector. For example, in the figure, site A sends its dependency vector to B and site B sends that vector to site C, so site C will have that information when it sends a message to site D. There is no way to know what the potential destinations are, so all dependency vectors must be included, i.e., the entire dependency matrix. The next proof shows that a much greater overhead is required to maintain transitive dependencies of lengths greater than length two.

Lemma 4: Transitive dependencies greater than length two require up to N vectors of information.

Proof:

Necessary Condition:

The first part shows that it is necessary to have N vectors of information for transitive dependencies of length three or more.

Proof by contradiction:

Assume sites send X dependency vectors, where $X < N$. Then current dependency information can be maintained for at most N -1 out of the N sites, because the dependency vectors for at least one site can not be sent. Let that site be denoted as site i. If site i contains required transitive dependency information, the exclusion of site i's dependency vector causes that transitive dependency to be undetected. Without that information, subsequent messages can be delivered out of causal order. We now show no matter how the vectors are selected, it is possible for site i to contain such transitive dependency information.

Assume each site randomly selects a vector to not send. Let that site be site 1 in Figure 6. Then sites 2 and 3 have no record of message *m1* that was sent from site 1 to 4. Consequently, site

4 does not hold message *m4* until after *m1* arrives and the messages are not delivered in causal order.

Consider a scheme where only the most recent dependency vectors are sent. If site 2 receives a message from the other N - 1 sites after the message from site1 arrives, then site 1's dependency vector will be the oldest and not included in the message from 2 to 3 and the same violation of causal ordering occurs.

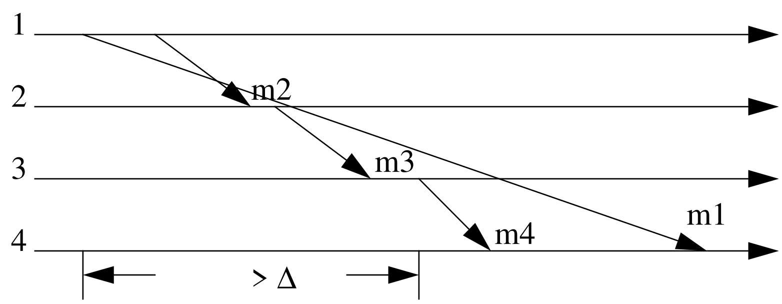

Figure 6 Transitive Dependency Information Lost If Site 1's Dependency Vector Is Not Sent.

No matter what method is used to select which vector is to be excluded, there are always situations in which the dependency vector of the selected site contains information necessary to ensure causal ordering.

Therefore, even by sending N dependency vectors, all transitive dependencies of length three can not be captured. This can be generalized to show that N-1 dependency vectors are insufficient for transitive dependencies of lengths greater than three. The dependency information for a chain of length four would be lost, since by the previous argument, the transitive information can not be preserved across four sites (for chains of length three), and would not be regained by adding another site at the end of the chain.

Sufficient Condition:

We now show that N dependency vectors are sufficient to capture transitive dependencies of length three or more.

For every message sent from site i to site j, add N vectors to the message. Site i's entire dependency matrix can be sent. When a message is received, the dependency matrix from the message can be merged into site j's dependency matrix, provided that there are no missing messages. Missing messages can be detected by values in column j of the message's matrix that are more recent than the corresponding values in j's matrix. This detects missing messages from transitive dependencies of up to length N - 1, which is the longest possible chain, since there are no other sites that can be involved in the chain. Therefore, by sending N vectors with every message, transitive dependencies of any length can be detected.

Therefore, N vectors is both necessary and sufficient to deal with transitive dependencies of lengths greater than two. □

As a space saving technique sparse matrices can be used wherein only cells in the dependency matrix that have non-zero entries can be sent. Initially, this will require a much lower message overhead. However, the number of these pairs that are sent will grow until they reach the size of the full dependency matrix (with added overhead to identify which cell they represent)[1][11]. Singhal and Kshemkalyani in [12] present a technique for compressing vector timestamps that could be useful for reducing the size of the vectors that are sent with the messages.

5. Conclusions

We have presented an algorithm that insures that real-time messages arrive in delta-causal order while limiting the message overhead to 2 vectors of length N per message. A proof was presented to show that the algorithm insures that messages will be delivered in the proper order unless they are past their deadlines in which case they will not be delivered. Transitive dependencies of lengths greater than two are ignored since two messages separated by more than two dependencies are unlikely to have a delta causal relation between them. The algorithm minimizes discarding messages that arrive at a site by delivering them when the messages they depend upon expire. This way any message that arrives at a site before its deadline is used.

This algorithm can be used to insure that multimedia data is delivered in delta-causal order. Multimedia data needs to be delivered in causal order in order to maintain the meaning of the data presentation. Because the algorithm supports data with real-time deadlines, data that has exceeded its deadline is discarded. However, the algorithm attempts to minimize the amount of data that must be discarded, and will utilize unexpired received messages, even if there are missing messages they depend upon. This algorithm is well suited for multimedia data, which has real-time deadlines.

Future work includes adding more support for multicasting and experimental simulation.

References

1 K. Birman, A. Schiper, and P. Stephenson, *Lightweight Causal and Atomic Group Multicast,* ACM Transactions on Computer Systems, Vol. 9, No. 3, pp. 282-314, 1991.

2 Computer, Special issue on Multimedia Information Systems, Vol 24, No. 10, October 1991.

3 The Computer Journal, Special issue on multimedia, Vol. 36, No. 1, 1993.

4 D. Ferrari, *Client Requirement for Real-Time Communication Services*, IEEE Communications Magazine, pp. 65-72, Nov. 1990.

5 R. Gusella, S. Zatti, *The Accuracy of the Clock Synchronization Achieved by TEMPO in Berkeley UNIX 4.3BSD*, IEEE Transactions on Software Engineering, Vol. 15, No. 7, pp. 847-853, July 1989.

6 IEEE Journal On Selected Areas In Communication, Special issue on multimedia, Vol. 8, No. 3, April 1990.

7 IEEE Transactions On Knowledge And Data Engineering, Special issue on multimedia, Vol. 5, No. 4, August 1993.

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

9 F. Mattern, *Virtual Time and Global States of Distributed Systems*, Proceedings of the International Workshop on Parallel and Distributed Algorithms, pp. 215-226, Amsterdam, 1989.

10 D. Mills, *Internet Time Synchronization: The Network Time Protocol*, IEEE Transactions on Communications, Vol. 39, No. 10, pp. 1482-1493, October 1991.

11 A. Schiper, J. Eggli A. Sandoz, *A New Algorithm to Implement Causal Ordering*, In Proceedings of the International Workshop on Distributed Algorithms, 1989, in Lecture Notes in Computer Science 392, Springer-Verlag, New York, pp. 219-232.

12 M. Singhal, A. Kshemkalyani, *An Efficient Implementation of Vector Clocks*, Information Processing Letters, 43, pp. 47-52, August 1992.

13 R. Yavatkar, *MCP: A Protocol For Coordination and Temporal Synchronization in multimedia Collaborative Applications*, The 12th International Conference on Distributed Computing Systems, 1992.