NOTE: This is a preliminary release of an article accepted by the ACM Transactions on Modeling and Computer Simulation. The definitive version is currently in production at ACM and, when released, will supersede this version.

Copyright (C) 2003 by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works, requires prior specific permission and/or a fee. Permissions may be requested from:

Publications Dept., ACM Inc.,
1515 Broadway, New York, NY 10036
USA
fax +1 (212) 869-0481, or
permissions@acm.org.
An Alternative Time Management Mechanism for Distributed Simulations

WENTONG CAI, STEPHEN J. TURNER and BU-SUNG LEE
Nanyang Technological University
and
JUNLAN ZHOU
University of California Los Angeles

Over the past few years, there has been keen interest in the management of time in distributed simulation environments. Previous emphasis in Time Management (TM) services has been based on time stamp ordering, which is both computation and bandwidth intensive. This paper discusses an alternative approach to time management based on causal ordering. Traditional causal ordering protocols incur a large amount of communication overhead, which is generally of the order of $N^2$ for a distributed system of $N$ processes. A new causal ordering protocol proposed by the authors, the Modified Schiper-Eggers-Sandoe (MSES) protocol, is presented in this paper. This new protocol minimizes the control information overhead of causal ordering by using the direct dependency tracking technique. The MSES protocol works well in both unicast and multicast environments, without relying on information about the underlying network topology and communication pattern among the processes of the distributed system. The MSES protocol has been successfully implemented as a middleware on top of DMSO RTI. Experiments have been conducted to benchmark the performance of the new time management mechanism with respect to the existing TM mechanisms available in DMSO RTI. The simulation scenarios of the experiments vary with different degrees of inter-federate dependency and federate event granularities. The ordering limitations of the causality based TM mechanism are addressed in this paper and the trade-off of the degree of event ordering and execution speed of simulations is discussed.

Categories and Subject Descriptors: J.6.8 [Simulation and Modeling]: Types of Simulation—Distributed

General Terms: Performance, Algorithm

Additional Key Words and Phrases: distributed simulation, time management, causal order, high level architecture

Authors' address: W. Cai, S. J. Turner, B. S. Lee, School of Computer Engineering, Nanyang Technological University, Singapore, 639798, email: {wercai, wsjturner, bslee}@ntu.edu.sg; J. Zhou, Computer Science Department, University of California Los Angeles, Los Angeles, CA 90095, USA, email: zjl@cs.ucla.edu.

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires specific permission and/or a fee.

© 2001 ACM /1529-3785/2001/07004-0002 $5.00

1. INTRODUCTION

1.1 High Level Architecture

There has been a continuing need for interoperability between new and existing simulations. It is observed that no individual simulation system can satisfy the needs of all users and application areas. A large simulation is usually built up by combining a group of simulation components which are developed at different times and by different teams. To promote the interoperability and reusability of the simulations, the Defense Modeling and Simulation Office (DMSO) has been working on the formation of a common technical framework for simulations since 1995. The efforts of DMSO led to the advent of the High Level Architecture (HLA) [IEEE Standard 1516. 2000]. The HLA generalizes and builds upon the results of Distributed Interactive Simulation (DIS) [IEEE Standard 1278. 1993] and related efforts such as Aggregate Level Simulation Protocol (ALSP) [Wilson and Weatherly 1994].

The primary mission of HLA is to create a synthetic, virtual simulation environment by systematically connecting individually developed simulations executing at geographically distributed locations. It defines an architecture where simulation components (federates in the HLA) complying with a set of HLA rules can interact via the services defined in the HLA Runtime Infrastructure (RTI) [IEEE Standard 1516. 2000] to create a combined large scale simulation (a federation in the HLA). Services defined in the RTI are divided into six categories: Federation Management, Declaration Management, Object Management, Ownership Management, Time Management, and Data Distribution Management.

1.2 Time Management (TM)

Ideally, simulations reproduce exactly the temporal relationships among the events that occur in the real world being modeled. However, the heterogeneous delays associated with the modeling computations and message transmissions over the network may lead to a violation of such relationships. For example, an observer may see a tank destroyed before it was hit by an aircraft, where the tank and aircraft are simulated at two computers [Fujimoto 1998]. These temporal anomalies or distortions of event ordering cause simulations to deviate from the real world. The objective of Time Management (TM) services in the HLA is to reduce the occurrence and effect of these temporal anomalies to meet with the objectives of the simulation execution. To facilitate the reuse and interoperability of the simulations, a number of design rationales have been proposed and incorporated in the definitions of the current TM services. Some important design principles are time management transparency, flexibility to accommodate various internal TM mechanisms commonly used by the simulations in the literature, minimal communication latency, and minimal computational and communication overhead imposed by the TM services [Fujimoto 1998].

A fundamental issue to be addressed by the TM services in the HLA is the ordering of the messages processed by a federate. Each federate in the federation generates a stream of events modeling its behavior and exchanges messages representing those events with other federates. Since each of these events takes place at an exact moment in time, the messages representing the respective events must be handled in the correct order at other federates. The following five message ordering
mechanisms were specified in the first version of the HLA Time Management design document [Fujimoto and Weatherly 1996]:

1. **Receive order (RO):** The messages from the other federates are delivered to the local federate in the order that they arrive.

2. **Priority order:** The messages from the other federates are passed to the local federate in an order consistent with their priorities where the message with the highest priority is delivered to the local federate first.

3. **Causal order (CO):** Causal order captures the cause and effect relationship among the events and ensures that the messages are delivered to the local federate according to this relationship.

4. **Causal and totally ordered:** CO is a partial message order and messages without cause and effect relationship are passed to the local federate in receive order. Causal and totally ordered extends causal order and ensures that for any two events without cause and effect relationship, the messages representing them are delivered to all federates receiving both messages in the same order.

5. **Time stamp order (TSO):** Each message is assigned with a time stamp and the messages are delivered to the local federate in ascending time stamp order.

The complexity of a TM mechanism depends on its message ordering scheme and progresses from receive order to time stamp order. RO is the most straightforward ordering scheme and incurs the lowest latency. TSO requires synchronization activities to be executed among the federates in the federation at regular intervals to ensure that they always progress their time at the same pace. It typically generates a lot of communication traffic, incurs high latency and yields constraints on scheduling events.

Two classes of simulation applications have received the most attention: analytic simulations and distributed virtual environments (DVEs) [Fujimoto 2000]:

---

**Analytic simulations** are used to collect quantitative data concerning the system being modeled and typically execute as fast as possible. A typical application of analytic simulations is a network simulator modeling the network operations in different traffic situations. Analytic simulations are often required to reproduce the behaviors of the system being modeled, including the temporal relationships among the events, as exactly as possible to ensure the validity of the statistical results obtained. In addition, it is often desired that the repeated execution of these simulations yield the same results. As such, time stamp order, with a total message ordering scheme, is usually applied on analytic simulations.

**Distributed Virtual Environments (DVEs)** typically embed human participants or actual physical devices as entities within the simulation. A familiar use is a war game, where players acting as characters in the game are placed in a computer generated world representing a modern battlefield. Some other DVE applications include computer generated training and educational environments, collaborative editing systems and teleconferencing [Yavatkar 1992]. A principal objective of the DVE is to present a sufficiently realistic view of the actual system being modeled to its participants. The DVE often requires real-time performance of simulation executions and places stringent time constraints on message deliveries. However,
it is not essential to emulate the actual systems exactly and temporal anomalies are acceptable if they are not perceptible to human participants. As such, receive order, which incurs the lowest delay on message transmissions and maximizes the execution speed of simulations, is normally suitable for real-time applications.

Based on the requirements and constraints of the above application classes, the current RTI software incorporates two TM mechanisms, a TSO based mechanism and an RO based mechanism. Although time stamp ordering has been the norm in analytic simulations, the before and after relationships among the events are sometimes of more importance than the exact time interval between events. For example, in the analytic simulations used for functional testing of some physical systems like robotics, it is the actions taken in response to events that are of primary concern. The TSO based TM mechanism satisfies the ordering requirements of these simulations but significantly increases the time needed to complete the simulations. In many DVE applications, RO fulfills the performance requirements very well, but it is unable to eliminate the temporal anomalies that may compromise the objectives of the DVE applications, such as a player who sees a tank being destroyed before the tank was hit.

The existing TM services in the HLA RTI provide message ordering schemes that are “all or nothing”. A new approach to TM based on causality is discussed in this paper, which provides a partial message ordering scheme for simulations. This new approach captures the “cause and effect” relationship among the events and enforces an ordered processing of two events only when such a “cause and effect” relationship exists between them.

A middleware approach has been adopted in the development of the CO time management mechanism. The middleware, which encapsulates the CO delivery algorithm, provides exactly the same interface as that defined by the HLA Interface Specification. Federates invoke services using the RTIambassador, but the invocation now goes through the middleware, which appends the appropriate causal information to any event messages sent. Similarly, the RTI invokes callback functions using the FederateAmbassador, but the invocation also goes through the middleware, which uses the causal information to guarantee CO delivery of those event messages.

In a previous implementation of our causal order time management mechanism [Lee et al. 2001], we adopted an embedded approach by modifying the Federated Simulations Development Kit (FDK) [FDK, 1998], an RTI implementation for which the source code is available. This allowed us to utilize the services provided within FDK such as obtaining the destination set of an event message from the underlying module that implements the multicast facilities. However, the disadvantages of modifying the RTI and FDK modules in this way are obvious in that it is very difficult to maintain and update these modules when there are upgrades to the RTI or FDK. In order to make CO time management compatible with any version of RTI, it is better to implement CO delivery in middleware. In addition, experimental results also show that the performance of our middleware approach to CO delivery is almost as good as that of the embedded approach [Turner et al. 2003].

The remainder of this paper is organized as follows: Section 2 contains a for-
mal definition of causal order and describes the new causal ordering scheme, the MSES protocol. Section 3 proposes a causality based TM mechanism, which uses the MSES protocol to realize causal order delivery of messages and is implemented as a middleware on top of DMSO RTI. Section 4 describes the benchmark experiments performed on the new TM mechanism and compares the experimental results against the existing receive order and time stamp order based TM mechanisms. Section 5 discusses the applicability of the new causality based TM mechanism, its drawbacks and possible solutions. Section 6 summarizes the major findings of this work and concludes the paper.

2. A CAUSAL ORDER DELIVERY PROTOCOL

2.1 Causal Order

The notion of causality was first introduced by Lamport in [Lamport 1978] in the general context of a distributed system. A distributed system consists of a group of asynchronous processes\footnote{That is, federates in the context of HLA-based distributed simulations.} cooperating to achieve a common goal by exchanging messages. The processes do not share any memory and there is no common clock. The action of each process in a distributed system can be modeled as three types of events: internal event (INT), message send event (SEND), and message receive event (RECV). The SEND or RECV event of a message $M$ is denoted as $SEND(M)$ or $RECV(M)$. Causality captures the “cause and effect” relationship among the events and establishes a partial order, called “happen before”, which is normally denoted by “$\Rightarrow$”. It is formally defined as follows:

**Definition 2.1. (Happen before relationship)**

(1) If $e$ and $f$ are events occurring at the same process and $e$ occurs before $f$, then $e \rightarrow f$.

(2) If $e = SEND(M)$ is the SEND event of a message $M$ and $f = RECV(M)$ is the RECV event of the same message, then $e \rightarrow f$.

(3) If $e \rightarrow g$ and $g \rightarrow f$, then $e \rightarrow f$.

(4) If neither $e \rightarrow f$ nor $f \rightarrow e$, $e$ and $f$ are said to be concurrent and $e \parallel f$.

Given two messages $M_1$ and $M_2$, if $SEND(M_1) \parallel SEND(M_2)$, then they are concurrent messages. Using the happen before relationship, causal order delivery can be formally defined as follows:

**Definition 2.2. (Causal order delivery)**

If two messages $M_1$ and $M_2$ are sent to the same process $P_i$ and $SEND(M_1) \rightarrow SEND(M_2)$, $M_1$ should be delivered to $P_i$ before $M_2$ (that is, $RECV(M_1) \rightarrow RECV(M_2)$ must be guaranteed).

It is to be noted that there is a distinction between the arrival of a message at a process and the delivery of that message. Messages may arrive at a process out of order but they must be delivered to (or processed by) that process in accordance with the causal order. In this paper, “receive” is used to mean the delivery of a message and the Receive() procedure comprises the actions executed when a message is delivered to a process.
2.2 Previous Work

At the heart of any causal ordering scheme is the computation of control information that needs to be appended with messages to enforce their causal delivery in an asynchronous communication environment. Generally speaking, the size of this information is at least of the order of N^2, where N is the number of processes in the distributed system. The ISIS system [Birman et al. 1991] includes one of the first practical implementations of causal ordering and other representative causal ordering schemes include the algorithm proposed by Raynal-Schipper-Toneg in [Raynal et al. 1991] and the Schipper-Eggli-Sandoz (SES) protocol [Schipper et al. 1989]. The SES protocol sends each message M with a matrix of size N^2, known as the causal vector CV_M. The destination process uses the associated matrix to determine when it is safe to process the message. A major limitation of the SES protocol is that it only works in unicast environment.

To maintain causality, each message M must not be delivered to a process Pi until all other messages sent to Pi whose SEND events happen before SEND(M), called the causal predecessors of M, have been processed. By appending a message M with information about all its causal predecessors, destination processes of M are able to process the message only after all of its causal predecessors have been delivered. This is the traditional approach to causal ordering and has been adopted by many algorithms, including the SES protocol.

By exploiting the topology of the underlying communication network and the communication pattern among the processes in a distributed system, it is possible to decrease the size of the control information to the order of N. For example, in the algorithm proposed by Sun [Sun et al. 1996] for a distributed collaborative editing system where broadcasting is assumed in the inter-process communication, it is enough to attach each message with an array of size N to maintain the causal order among messages.

Another approach to reduce the control information is based on the direct dependency tracking technique. It is obvious that not all causal predecessors need to be known to ensure M's causal order delivery. Only the immediate predecessors of M, the messages on which the delivery of M is directly dependent are necessary. The deliveries of the other causal predecessors of M can be implied by the delivery of the immediate predecessors of M. The formal definition of immediate predecessor is given as follows:

**Definition 2.3. (Immediate Predecessor)**

M' is the causal predecessor of a message M regarding Pi written as M' → p, M if SEND(M') → SEND(M) and both M' and M are sent to Pi. If M' → p, M and there does not exist a message M'' such that M'' → p, M'' → p, M, we say that M' is an immediate predecessor of message M regarding Pi.

Figure 1 explains the direct dependency technique with the help of an example. Here, SEND(M_1) → SEND(M_2), and SEND(M_2) → SEND(M_3). M_1, M_2 and M_3 are all sent to P_3. Both M_1 and M_2 are causal predecessors of M_3 regarding P_3, but M_1 need not be included in the causal vector CV_M_3 since it is no longer an immediate predecessor of M_3. However, M_1 is the immediate predecessor of M_4 regarding P_4 and thus the information of M_1 has to be encapsulated in CV_M_4 to
ensure the causal order delivery of $M_4$ to $P_4$.

The principal advantage of the direct dependency tracking technique is that it can achieve optimality without relying on assumptions over the underlying communication network and thus is feasible for a wider class of applications. Algorithms using the direct dependency tracking technique include the algorithm proposed by Prakash for mobile computing in [Prakash et al. 1997], the algorithm proposed by Kshenmukayani in [Kshenmukayani and Singhal 1998] and the Modified Schiper-Eggle-Sandoz (MSES) protocol [Zhou 2000] discussed in this paper.

The MSES protocol incurs a smaller communication overhead and requires less local storage space than the algorithm proposed by Prakash [Prakash et al. 1997]. Simulation experiments have been conducted to compare it with Prakash’s algorithm and the original SES protocol in various situations and the results attest to the efficiency of the MSES protocol [Zhou 2000]. The MSES protocol is equivalent to the algorithm proposed by Kshenmukayani [Kshenmukayani and Singhal 1998] in terms of eliminating non-immediate or delivered immediate causal predecessors. As a modified version of the SES protocol, it also eliminates the prerequisite of the use of unicast communication media and works in both multicast and broadcast environments.

Another relevant work is the Causal Memory [Alamad et al. 1993], a weak memory model and associated algorithm based on causal order theory for implementing distributed shared memory. The causal memory introduces a similar notion of causality based on read and write operations in a shared memory environment. The associated algorithm guarantees that reads return values consistent with causally related reads and writes. However, there are two major differences between the causal memory algorithm and the protocol studied in this paper:

—The causal memory algorithm uses broadcast. The state information, including vector time, is transmitted to all the processes; whereas, in our MSES protocol, a message may not be sent to all the other processes. Similar to Prakash’s protocol [Prakash et al. 1997], since multicast is supported, the algorithm will be much more complex than those based on unicast or broadcast.

—in shared memory environments, “hidden writes” are always possible. That is, it is always possible for subsequent writes to be applied to a memory location prior to any reader seeing them. However, these semantics are not true for message-passing systems as indicated in [Alamad et al. 1993]. In message-passing systems, all messages sent must be received by the corresponding receivers.

2.3 The Modified Schiper-Eggli-Sandoz (MSES) Protocol

In the MSES protocol, each process $P_i$ keeps an array of size $N$, denoted as $VT_{P_i}$, to track its vector time [Fidge 1988; Mattern 1989], according to the following definition:

**Definition 2.4. (Vector Time)**

For each process $P_k$ in a distributed system, there is an element in the vector time $VT$ (i.e., $VT[k]$) corresponding to it. The vector time $VT_{P_i}$ of $P_i$ in a system comprising $N$ processes is maintained according to the following rules:

1. Initially, $VT_{P_i}[k] = 0$ for $k = 1, \ldots, N$.
2. On sending message $M$, $P_i$ updates $VT_{P_i}$ as follows: $VT_{P_i}[i] = VT_{P_i}[i] + 1$.
3. The new vector is the vector time of $M$ and is propagated to the remote processes with $M$.
4. On receiving a message $M$ with attached vector time $VT_M$ from other processes, $P_i$ increments $VT_{P_i}[i]$ as in (2). Next, $VT_{P_i}$ is further updated to the element-by-element maximum of $VT_P$ and $VT_M$.

```c
# When process $P_i$ sends a message $M$:
/* (1) update process $P_i$'s vector time */
VT_P[i] := VT_P[i] + 1;
/* (2) timestamp and send message $M$ */
VT_M := VT_P;  
CV_M := CV_P;  
send message ($M$, $VT_M$, $i$, Dest($M$), $CV_M$) to all processes in Dest($M$);
/* (3) update $CV_P$'s entries corresponding to the destinations of $M$ */
for all $P_k$ ∈ Dest($M$)
  CV_P[k] := {(i, $VT_M[i]$)};

# When a message ($M$, $VT_M$, $i$, Dest($M$), $CV_M$) arrives at $P_j$:
if CanBeProcessed($CV_M$)
  then {
    /* (1) receive message ($M$, $VT_M$, $i$, Dest($M$), $CV_M$) */
    Receive($M$, $VT_M$, $i$, Dest($M$), $CV_M$);
    /* (2) check all the messages buffered in Buff$_{P_j}$ */
    for all ($M'$, $VT_M'$, $i'$, Dest($M'$), $CV_{M'}$) ∈ Buff$_{P_j}$
      if CanBeProcessed($CV_{M'}$) then
        Receive($M'$, $VT_M'$, $i'$, Dest($M'$), $CV_{M'}$);
    else
      Buff$_{P_j}$ := Buff$_{P_j}$ ∪ {($M$, $VT_M$, $i$, Dest($M$), $CV_M$)};
  }
```

Fig. 2. Pseudo-code of the MSES protocol.

Vector time is able to capture the causal relationship among messages, but it cannot guarantee the causal order delivery of messages. So, in addition to the vector time, each process $P_i$ also maintains a causal vector to track the immediate predecessors of the next message $M$ generated by $P_i$. The causal vector $CV_{P_i}$ consists of $N$ entries, each of which includes zero to $N$ tuples. Each tuple $(k, x)$...
kept in the entry of CV_M, at index I (denoted as CV_M[I]) indicates a message M' that is an immediate predecessor of M regarding P_i. M' is generated by process P_k at VT_P_k[0] = x and is not known to be delivered to P_i so far. The tuple (k, x) is called the signature of M'. Figure 2 depicts the pseudo-code of this protocol.

```c
/* This procedure determines whether M can be processed by P_i */
Procedure CanBeProcessed(CV_M)
{
    if (CV_M[j] = Null) \lor (\forall (k, x) \in CV_M[j]((x \leq VT_P_k[0])))
        then return true;
    else return false;
}

/* This procedure is called when P_i processes (M, VT_M, i, Dest(M), CV_M) */
Procedure Receive(M, VT_M, i, Dest(M), CV_M)
{
    /* Step 1: update the entries in CV_P corresponding to M’s destinations */
    for all P_k \in Dest(M) {
        CV_P[k][0] := CV_P[k][0] \cup \{(i, VT_M[0])\};
        CV_P[k][0] := CV_P[k][0] - \{\{l, VT_M[0]\}\} l = 1 \ldots N, l \neq i;
    }

    /* Step 2: update the entry in CV_P concerning P_i */
    CV_P[i][0] := CV_P[i][0] - \{\{l, VT_M[0]\}\} l = 1 \ldots N;

    /* Step 3: update the entries in CV_P for other processes */
    for all P_k \notin Dest(M) \land (P_k \neq P_i) {
        /* perform garbage collection on CV_P[k] */
        for all (l, x) \in CV_P[k][0]
            if (x \leq VT_M[0]) \lor ((l, x) \notin CV_M[l][0])
                then CV_P[k][0] := CV_P[k][0] - \{(l, x)\};

        /* perform garbage collection on CV_M[l][0] */
        for all (l, x) \in CV_M[l][0]
            if (x \leq VT_P_k[0]) \lor ((l, x) \notin CV_P[k][0])
                then CV_M[l][0] := CV_M[l][0] - \{(l, x)\};

        /* combine CV_P[k] and CV_M[l][0] */
        CV_P[k][0] := CV_P[k][0] \cup \max(CV_M[l][0], CV_P[k][0]);
    }

    /* Step 4: update process P_i’s vector time */
    VT_P_i[0] := VT_P_i[0] + 1;
    for all k = 1 \ldots N VT_P_k[0] := \max(VT_M[0], VT_P_k[0]);
}
```

Fig. 3. CanBeProcessed() and Receive() procedures of the MSES protocol.

In Figure 2, the CanBeProcessed() procedure is used by P_i to determine whether a message can be processed and the Receive() procedure is called by P_i to receive or process a message. The pseudo-code of the CanBeProcessed() procedure and the Receive() procedure are further illustrated in Figure 3.
Two operations \( T_{1 - \text{max}} \) used in the \( \text{Receive()} \) procedure are defined as follows:

\[
\begin{align*}
T_1 - \text{max} & \quad T_2 \\
\text{if } \exists (i, x) \in T_1, (i, y) \in T_2, \text{ and } x \leq y, T_1 = T_1 - \{(i, x)\} \\
T_1 \cup \text{max} & \quad T_2 \\
T_1 = T_1 \cup T_2 \\
\text{if } \exists (i, x) \in T_1, (i, y) \in T_1, \text{ and } x \leq y, T_1 = T_1 - \{(i, x)\}
\end{align*}
\]

![Diagram](image)

Fig. 4. Examples of the MSES protocol.

Two examples of the MSES protocol are shown in Figure 4. In Figure 4(a), when \( P_3 \) receives the fire message, the tuple \( (I, I) \) is added into \( CV_{P_3}[I] \). It indicates that the next message sent from \( P_2 \) to \( P_3 \) is causally dependent on the fire message and can only be delivered to \( P_3 \) when the fire message is delivered to \( P_2 \). When \( P_3 \) receives the hit message, it detects that the hit message is dependent on a message with a signature \( (I, I) \), i.e. the fire message. Since \( VT_{P_3}[I] = I \), the fire message has already been processed by \( P_3 \) and the \( \text{CanBeProcessed()} \) procedure allows the hit message to be delivered to \( P_3 \) immediately. The vector time stamp of the hit message is used to update \( CV_{P_3} \) (Step 2 of the \( \text{Receive()} \) procedure). \( P_3 \) compares
the time stamp of the hit message \([1, 2, 0]\) and finds that the fire message with a signature \((1, 1)\) in \(CV_{P_3}[2]\) has been delivered to \(P_3\). Thus \((1, 1)\) is removed from \(CV_{P_3}[2]\).

In Figure 4(b), when the hit message arrives at \(P_3\), the fire message has not been processed by \(P_3\) and \(VT_{P_3}[I] = \emptyset\). In the \texttt{CanBeProcessed()} procedure, \(P_3\) compares \(VT_{P_3}[I]\) with the signature \((1, 1)\) and decides that the hit message has to be placed in the buffer. After the fire message with a time stamp \([1, 0, 0]\) is processed by \(P_3\), \(VT_{P_3}[I] = 1\) and it is now safe to process the hit message in the buffer. It can be seen that the causal order among the messages is maintained. By only maintaining the signatures of immediate predecessors not known to be delivered in the messages’ causal vectors, the MSES protocol reduces the communication overhead and the local storage space needed to ensure causality. The correctness proof and an analysis of the optimality of the MSES protocol can be found in [Zhou 2000].

3. A CAUSALITY BASED TM MECHANISM

3.1 Overall System Structure

In an HLA based federation, every federate interacts with the RTI through the interface composed of the \texttt{RTIambassador} and \texttt{FederateAmbassador}. Before the federate can invoke any RTI services, it must first create an instance of the \texttt{RTIambassador}. It is natural to put the middleware between the federate and the \texttt{RTIambassador}. By doing so, the middleware is able to intercept event messages from the federate, analyze those messages, and add causality information. Thereafter, the middleware passes the messages to the \texttt{RTIambassador} and updates the federate’s causality time management information kept in the middleware.

![System structure with middleware.](image)

Conversely, when the RTI has any event message destined for a certain federate, the RTI will invoke a call-back function on the \texttt{FederateAmbassador}. Before the message can be passed to the federate, it must be analyzed to obtain the causality information to see whether it is safe to pass the message to the federate. Hence, the middleware should be placed between the RTI and the \texttt{FederateAmbassador}. When it is safe to deliver the message, the middleware also updates the federate’s causality time management information accordingly. Figure 5 illustrates the overall system structure.

The MiddleRTIambassador class inherits the standard RTIambassador class and overrides appropriate methods. The MiddleFederateAmbassador class is a "wrapper" class for the user's FederateAmbassador class implemented as part of the application code. We cannot use simple inheritance here since the MiddleFederateAmbassador class must be independent of the application. In particular the name of the user's FederateAmbassador class is not known, only that it is a subclass of the abstract class FederateAmbassador. Instead of using inheritance, we therefore use the technique of forwarding method invocations from the MiddleFederateAmbassador class to the user's FederateAmbassador class, allowing us to redefine methods as appropriate.

The only change to the federate code, as a result of the insertion of these middleware classes, is that the instance declaration of RTIambassador is replaced by an instance declaration of MiddleRTIambassador. The method calls made by the federate remain exactly the same. When the federate joins the federation, the MiddleRTIambassador initializes the instance of MiddleFederateAmbassador by providing a reference to the user's FederateAmbassador as a parameter. The MiddleFederateAmbassador is then passed to the RTI as the call-back reference to the federation instead of the user's FederateAmbassador.

To ensure CO delivery both the MiddleRTIambassador and MiddleFederateAmbassador need to share causality time management information for the local federate. In this implementation, the shared information is held in a class called CausalOrderModule. An instance of this class is shared by both the middleware ambassador classes by means of a "has a" relationship. The sharing of the CausalOrderModule is implemented as follows. The instance of the CausalOrderModule is created in the MiddleRTIambassador at the time the federate joins the federation and a reference to this instance is passed to the MiddleFederateAmbassador at the same time as the user's FederateAmbassador.

There are other supporting classes that provide services to the CausalOrderModule. By organizing the system in this way, encapsulation and maintainability of code can be achieved. Section 3.2 describes the use of these classes, which are as follows:

— VectorTime - information about messages sent and received by local and remote federates, together with methods for manipulating vector time.
— CausalVector - information that is used to determine the message delivery order, together with methods for manipulating the causal vector.
— UnwrappedMessage - the original event message and its associated causality information (held as individual components), together with encoding and decoding methods.
— FederateManager - federate information, together with methods for managing that information.

Figure 6 gives an overview of the class diagram. The relationship between the MiddleRTIambassador and the MiddleFederateAmbassador classes is as described above. The CausalOrderModule "has a" VectorTime and CausalVector and inherits from FederateManager. The FederateManager is used to assign a process identifier \( i \ (1 \leq i \leq N) \) to the local federate and to collect and provide knowledge of remote federates such as joining the federation, publishing, subscribing, resigning, etc.
management of the federate information is described in Section 3.3.

The *MiddleFederateAmbassador* class has multiple instances of *UnwrappedMessage* that are held in a *MessageBuffer*. This is the queue of messages that cannot be delivered at the current time because of the causality check. Since an undeliverable message contains not only the original message but also extra causality information that will be used later to recheck if the message can be delivered, the data structure of *MessageBuffer* is a vector of *UnwrappedMessage*. Insertion, deletion, and iteration are simple and fast, because of the features provided in the *Vector* class. A further data structure used when sending and analyzing a message is *DestinationList*. This is a set whose elements are process identifiers.

3.2 Implementation of the CO Delivery Algorithm

The CO algorithm is implemented in middleware using the classes described in Section 3.1. There are three steps for sending a message (see Figure 2). These steps are implemented within the overridden methods in the *MiddleRTIambassador*: sendInteraction() and updateAttributeValues(). When the federate invokes these methods, it will invoke the overridden methods instead of the original methods in *RTIambassador*. The *MiddleRTIambassador* adds the appropriate causality information to give an encoded message and invokes the corresponding method in the *RTIambassador*. In addition, it will update the causality time management information held in the middleware. The following paragraphs describe in detail how these three steps are implemented.

(1) Update process $P_i$’s vector time $VT_P$.

The *MiddleRTIambassador* invokes a method in the *CausalOrderModule*. This invokes a further method in *VectorTime* to perform the increment.

(2) Timestamp and send message $(M, VT_P, i, Dest(M), CV_P)$.
First, the local vector time and causal vector are collected through the `CausalOrderModule`. Before they are attached to the original message and sent through the RTI, they must be encoded into string format. The encoding functions are provided in the `VectorTime` and `CausalVector` classes respectively. In addition, the local process identifier and the destination list of the particular message are obtained from the `CausalOrderModule`. The above four items of causal information are attached to the original message and sent to the RTI by invoking the corresponding method in the `RTLambassador`.

(3) Update $CV_p$’s entry corresponding to the destination of $M$

The `MiddleRTLambassador` invokes a method in the `CausalOrderModule` to collect the appropriate causality information. This method invokes a further method in `CausalVector` to add this information.

As described above, there are four items of causal information that need to be attached to the original message: the vector time, causal vector, destination list and source process identifier. There are a number of possible approaches to the way that they are attached and sent through the RTI. One approach is to pass the causal information as parameters of a meta message that also contains the original message. This requires the definition of additional interaction and object classes in the Federate Execution Definition (FED) file. Although methods of the `RTLambassador` could be overridden to subscribe to these classes automatically, the code required would be complicated.

A simpler approach is to attach the causal information to the message’s tag which then becomes a meta tag. In the HLA interface specification, every interaction or attribute update message has a tag associated with it for user defined purposes. As the tag is a character string, all four items of causal information have to be converted to string format. Since there may already be a tag present in the original message, this must also be appended. The four items of causal information and the original tag are separated by the delimiter ampersand. Figure 7 gives an example of a string encoded meta tag. This is for a message sent from process 0 to destination processes 1 and 2. The advantage of this approach is that there is no need to declare any additional interaction or object classes in the FED file. There is also much less code required, compared with the meta message method.

![Fig. 7. An example of a meta tag.](image-url)

Receiving a message in the CO delivery algorithm (see Figure 2) can be divided into two parts: a causality check and the delivery procedure. When message $(M, VTM, i, Dest(M), CV_M)$ from $P_i$ arrives at process $P_j$, the algorithm must first check whether the message can be delivered by comparing $CV_M[j]$ with the vector...
time \( VTP_i \) of process \( P_i \) \((\text{CanBeProcessed}())\) procedure in Figure 3). If it is safe to deliver the message, the steps in the \textit{Receive()} procedure in Figure 3 are executed, otherwise the message must be buffered until it can be guaranteed that there will be no causality violation.

The two parts in receiving a message are implemented by means of redefined methods in the \textit{MiddleFederateAmbassador}. When the RTI invokes \textit{receiveInteraction()} or \textit{reflectAttributeValue()} , these redefined call-back methods will be invoked instead of those in the user's \textit{FederateAmbassador}. Before any actions can be performed on the message, the meta tag has to be decoded. The vector time and causal vector are converted back into instances of their respective classes (the decoding is performed by the classes themselves). The destination list and the source process identifier are held in appropriate data structures. The last token in the meta tag is restored as the original tag. All these items, together with the original message, are held as individual components of the class \textit{UnwrappedMessage}.

As the causal vector and vector time are both converted back to their class format, checking whether the message can be delivered is easily done by calling appropriate methods on these classes. If the condition for delivery is not satisfied, it is necessary to buffer the complete message as an \textit{UnwrappedMessage} in the \textit{MessageBuffer} queue. If the message passes the check, the original message is first forwarded to the user federate by invoking the corresponding call-back method \textit{receiveInteraction()} or \textit{reflectAttributeValue()} in the \textit{FederateAmbassador} of the user's application.

Then the causality information in the message is used to update the local causality information held in middleware. To do this, the \textit{MiddleFedAmbassador} invokes a method in the \textit{CausalOrderModule} to execute the four steps of the \textit{Receive()} procedure in Figure 3 as follows:

1. Update \( P_j \)'s causal vector \( CV_{P_j} \) concerning destinations of \( M \)
2. Update \( P_j \)'s causal vector \( CV_{P_j} \) concerning the sender \( P_i \)
3. Update \( P_j \)'s causal vector \( CV_{P_j} \) concerning other entries
   The \textit{CausalOrderModule} invokes methods in \textit{CausalVector} to perform each of the three substeps described above.
4. Update \( P_j \)'s vector time \( VTP_i \)
   The \textit{CausalOrderModule} invokes methods in \textit{VectorTime} to perform the increment and to update the local vector time from the vector time in the message.

Note that the messages in the \textit{MessageBuffer} queue will be rechecked whenever any message passes the check and is delivered, because delivery of a message updates the local causality information and this may mean that further messages in the queue can now be delivered.

3.3 Federation Information Management

An important issue in the implementation of this algorithm is how to collect, maintain, and utilize the federation information. This information is mainly required to construct the destination list for each message. Some of this federation information is maintained by the RTI and is available in the Management Object Model (MOM). The MOM [Fullford and Wetzel 1999] is a universal model which identifies federations, objects and interactions used in a federation. However, ad-
ditional information must also be kept such as the mapping from federate handles to process identifiers. Such issues were solved differently in the previous embedded version of the causal order time management mechanism [Lee et al. 2001]. In the embedded version, all the required federation information is available from the FDK itself. However, in the middleware version, since every federate has its own copy of the middleware that keeps the federation information, every federate has to communicate with the RTI and peer federates regularly to keep this information updated.

The advantage of obtaining information from the MOM is apparent in that it is simple, as there is only one source with which the middleware needs to interact. However, there is a limitation to the service that the MOM provides, which is that the federate has to poll the MOM regularly in order to keep the information updated. This wastes network bandwidth and processor time. An alternative approach is to get information from the MOM at initialization time only. Subsequently, information is exchanged directly between federates by means of sending interactions. Therefore, the information is updated only when it changes. Network bandwidth and processor time are saved significantly, especially when the number of federates is large.

The main data structures for federation information management are FedToPID and two instances of SubscriptionTable for interactions and objects respectively. FedToPID maps the federate handle to the process identifier. The federate handle is the identity assigned by the RTI that is common across the federation, but which may not be consecutive. The process identifier is the identity assigned by the middleware that is common across the federation. It starts from zero and is consecutive. The advantage of having a process identifier defined in this way is that it reduces the size of the causal vector and improves the efficiency of the causal order delivery algorithm. The two SubscriptionTables give the destination set (as a list of process identifiers) for each interaction class or object class.

When a federate joins the federation, it requests the MOM for the federate handles of all the federates currently in the federation. The initial FedToPID table is constructed from this data and the process identifiers are deduced from the ordering of the handles. The federate also requests the MOM for the subscription information of both interaction and object classes and uses this information to construct the initial entries of the two SubscriptionTables. Note that the initial tables correspond to information about federates that have already joined the federation. The federate then sends out information about itself to all other federates in the federation by means of special interactions. The interaction handle for these special interactions is known to the middleware, but is transparent to the user federate. (However, these classes need to be added into the FED file).

During execution, the FedToPID table is updated as further federates join the federation. The information is provided by newly joined federates which send their federate handle and process identifier after their initialization stage. The information is simply inserted in the FedToPID table according to the order of federate handle. Similarly, when other peer federates subscribe or unsubscribe to certain classes, this information is sent to all federates by means of special interactions. The SubscriptionTables are updated using this information. Finally, when a federate re-
signs from the federation, it needs to notify the other federates of its resignation. These will remove the federate from the FedToPID table and delete all subscription information related to that federate.

4. BENCHMARK AND EXPERIMENTAL RESULTS
Experiments have been conducted to evaluate the efficiency of the proposed causality based TM mechanism and to compare it with the existing TM mechanisms implemented in the DMSO RTI (i.e., the TSO and RO based TM mechanisms). In the first set of experiments (i.e., Section 4.2), a network of up to seven PCs connected by fast Ethernet running DMSO RTI 1.3NG-V5 was used. In the second set of experiments (i.e., Section 4.3), a network of 36 high-end PCs connected by 100 Mbps switch running DMSO RTI 1.3NG-V6 was used.

4.1 Performance Benchmarks
To benchmark the performance of the TM Mechanisms under different situations, three simulation scenarios are created. Each of these scenarios is designed with a different degree of inter-federate dependency. The first scenario depicts a loosely coupled federation, where each of the federates is running autonomously. The second scenario depicts a closely coupled federation, where federates interact with each other frequently. The third scenario depicts a mixed federation, which comprises a mixture of autonomous federates and interacting federates.

![Figure 8: Execution of a loosely coupled federation.](image)

In the first scenario (i.e., loosely coupled federation), the dependencies among the federates are reduced to the minimum so that only the order of the messages sent by the same sender to the same receiver needs to be maintained. This is the best case where federates are executed as fast as possible and dependencies among the federates are ignored. The execution of a loosely coupled federation with three federates is further illustrated in Figure 8.
While the first scenario creates the least causally related federates in a federation, the second scenario (i.e., *closely coupled federation*) creates the most causally related federates. In this scenario, the causal orderings are exploited to the extreme in that each message sent out by a federate is dependent on the previous messages received from all other federates.

![Diagram of mixed federation execution](image)

**Fig. 9.** Execution of a mixed federation

The aforementioned two scenarios illustrate the performance of federations in two extreme cases: one with zero inter-federate dependency, the other with maximum inter-federate dependency. Since these two scenarios are rarely seen in practical simulations, another scenario (i.e., *mixed federation*) is studied with moderate inter-federate dependency. In this scenario, three federates make up a "warfare" federation. One federate is simulating an aircraft, another federate is simulating a tank, and the third federate is simulating an observer. A typical execution of the "aircraft" federate, "tank" federate and "observer" federate is shown in Figure 9.

The "aircraft" federate broadcasts an update message to the other two federates periodically. It also "fires" at the "tank" federate sometimes. When receiving a "fire" interaction, the "tank" federate broadcasts a "hit" update message to the other two federates. Except when reacting to the "fire" interaction initiated by the "aircraft" federate, the "tank" federate does not send any messages. The "observer" federate receives messages from the "aircraft" and "tank" federates and does not send messages itself. Figure 9 shows the scenario when the fire interval of the aircraft federate is 100 update messages.

In all of the three scenarios, communication between federates uses reliable TCP/IP connections. A fixed spin loop is executed at the federates after each outgoing message to account for the CPU time used by local simulation tasks, so as to make the simulation more realistic. In federations using the TSO mechanism, each federate requests to advance its time by 1.0 second after it sends an interaction.
and it will continuously call \texttt{tick()} until its time advancement request is granted by the RTI before it sends the next interaction. The lookahead of federates is set to 0.1 second. The time stamp of interactions is set to the current time of the local federate plus 1.0 second.

4.2 Comparison of Time Management Mechanisms

In this set of experiments, the performance of the RO, CO and TSO mechanisms was compared for the three different scenarios using a dedicated network of PCs. For each scenario, experiments were carried out to measure the execution time of federates over 5000 time steps. The number of federates in the federation was also varied from 2 to 7 in the first two scenarios with each federate on a separate PC. For each combination of time management mechanism and federation size, the simulation was repeated five times and the average execution time taken.

In the loosely coupled federation, activities of different federates are independent of each other. Thus, for a federate \( P_i \), temporal relationships of two messages received from different remote federates are of no interest. The only requirement for \( P_i \) is that the ordering of messages from the same federate must be maintained so that it could view the activities of the federate in a timely manner. Both the CO mechanism and TSO mechanism satisfy the ordering requirements of federates in loosely coupled federations. Although RO does not guarantee the ordering of messages from the same source explicitly, such ordering requirement is satisfied when TCP/IP is used as the underlying network communication protocol.

![Graph](image)

Fig. 10. Execution time of federations in the loosely coupled federation.

Figure 10 shows the results for the loosely coupled scenario. The results show that the CO mechanism incurs only a slightly higher overhead compared with the RO mechanism. However, the overhead for the TSO mechanism is much larger and increases according to the number of federates in the federation.

For the closely coupled scenario, in a federation execution using the RO or CO mechanism, federates will execute similar programs as the first scenario except that...
after sending an interaction a federate will continuously call \texttt{tick()} until it receives an interaction from each of the other federates before it sends the next interaction. In a federation execution using the TSO mechanism, it guarantees that federates will not receive messages occurring in the past. If all the federates are time-stepped federates with the same time-step size, dependency among federates in the closely coupled federation is implicitly realized in this case. A federate will ask for simulation time advancement to the next time step after sending an interaction without explicitly waiting for replies from other federates. In this scenario, federates using the TSO mechanism execute exactly the same program as that in the first scenario.

Figure 11 shows the results for the closely coupled scenario. As explained above, the RO and CO mechanisms therefore have additional code in the application to guarantee the causal requirements. Although the TSO mechanism satisfies the constraint without the use of extra code, the execution time is much larger. Again, the overhead in using the CO mechanism is only slightly larger than that of the RO mechanism.

For the mixed federation scenario, experiments were conducted with the inter-fire interval of the aircraft federate changing from 100 update messages to 10 update messages. Similarly as before, a fixed spin loop is executed at the “aircraft” federate after each message is sent. As the “hit” message is causally dependent on the “fire” message, it is ensured by using the CO mechanism that the observer sees the aircraft “fire” and the tank being “hit” thereafter. It is ensured by the CO mechanism as well that the updates from the “aircraft” are received by the “tank” and “observer” in order. Thus the CO mechanism fulfills all the ordering requirements of federates in this scenario. Since the observer may see the “hit” event of the “tank” occurring before the “fire” event with the RO mechanism, the RO mechanism cannot be used for this scenario. However, the TSO mechanism is applicable in this scenario, since it provides a totally ordered delivery service of messages. In addition, it guarantees that the
update messages from the “aircraft” and the “hit” messages from the “tank” are intermixed correctly, specifically, in time stamp order, when they are delivered to the “observer”.

Figure 12 shows that similar results are obtained for the mixed scenario. The CO mechanism demonstrates more than 50% performance improvement over the TSO mechanism. Thus, it can be seen that the CO mechanism outperforms the TSO mechanism and allows federates to enhance their communication efficiency in federations with moderate inter-federate dependencies.

4.3 Scalability Study

In this second set of experiments, we consider an “aircraft”, a “tank” and an “observer” federate in the third scenario (as described in Section 4.1) as one group. To investigate the scalability of the CO and the TSO mechanisms, the number of groups in the experiment is increased from 1 to 12 (i.e., number of federates in the federation varies from 3 to 36). The “aircraft” federate broadcasts an update message to all the federates in the federation in every simulation step. It also sends a “fire” message to the “tank” federate and the “observer” federate in its group in every 10 update messages. When the “tank” federate receives a “fire” message, it sends a “hit” message to the “aircraft” federate and the “observer” federate in the same group. The “aircraft” federate executes a fixed spin loop after it sends out a “fire” message. The “tank” federate and the “observer” federate also execute the fixed spin loops after receiving the “fire” message. The “aircraft” federate terminates after executing 2000 simulation steps. The “tank” and “observer” federates terminate after executing 2000 simulation steps as well as receiving all the “fire” messages sent by the “aircraft” federate in the same group.

To change the network load, a network traffic generation program was developed. In each cycle, it sends 16 KB of data to each of the other computers in the experiment using TCP connections and then sleeps for some time. When the other computers receive the data, they simply send back the same data to the sender.
Fig. 13. Execution time of federates in scalability study.
In the first experiment, the network traffic generation program was not used. In the second and third experiments, a traffic generation program was executed on each computer used in the experiment. The sleep time is dynamically adjusted so that 0.5 MB and 1 MB of data are sent and received per second by each traffic generation program in experiments 2 and 3 respectively.

Experimental results were collected using a dedicated network of 36 Pentium 4 machines each with 2.4 GHz CPU and 512 MB DDR RAM. They are connected by 100 Mbps switch. The OS in each computer is Windows 2000. The number of federates in the federation was varied from 3 to 36 with each federate running on a separate machine.

The federation execution time of the three experiments are shown in Figure 13 (a), (b) & (c) respectively. It can be seen from the figure that the execution time of CO is only marginally affected by the background network traffic. It is fairly consistent in all the three experiments. However, the background network traffic has a significant effect on the execution time of TSO. For example, with three federates, it takes 30.44 seconds to finish when network traffic generation program is not used. The execution time is increased to 45.96 seconds and 75.27 in experiments 2 and 3 when the network traffic generation program is executed concurrently with different traffic load generated. This registers 50% and 150% increment respectively. The increment of the execution time also increases with the size of the federation. When the network generation program is used, more background traffic will be generated with an increased number of federates in a federation. In experiment 2, for example, when the number of federates in a federation increases to 36, the increment of execution time, compared to the case where the network traffic generation program is not used (i.e., experiment 1), becomes 250%.

The negative effect of the background network traffic on TSO is mainly due to the periodic Lower Bound Time Stamp (LBTS) calculation. In the experiments, the interval to calculate LBTS is set using the default value (i.e., 0.05 seconds). Although the size of message increases with the federation population when CO is used, the number of communications required is much less than that required by TSO. Thus, as shown in Figure 13, CO is much more scalable than TSO especially under heavy network load situations.

5. DISCUSSION

5.1 Limitation of Causal Ordering

The principal limitation of the causality based TM mechanism is that it only provides a partial ordering scheme and is unable to eliminate the anomalies that occur due to the fact that federates can process the causally concurrent events in a different order. For example, in the mixed federation discussed in Section 4.3, if the tank is moving and issuing updates, it is not ensured by the causality based TM mechanism that the causally unrelated update messages from the tank and aircraft are processed in total order at the observer. Suppose there are two aircraft, A and B. Both of them try to fire at a tank. Aircraft A fires earlier than B and destroys the tank. Since the messages from the two aircraft are causally concurrent, it is not guaranteed that they are processed in order by the observer. It could cause the observer to believe that the tank is destroyed by aircraft B. This is a situation
where ordering of concurrent events may influence the execution of the federates and may be important to the objectives of the federation.

![Diagram](image_url)

**Fig. 14.** Mechanism to buffer outgoing messages.

Causal order is defined based on the sending of the event. It presents a problem, however, for the traditional paradigm of discrete event simulation (DES) that can schedule events into the future. For example, a simulation may schedule a message $m_1$ at time $t_1$ with timestamp $t_1 + \Delta t_1$ and subsequently schedule a message $m_2$ at $t_2 > t_1$, with timestamp $t_2 + \Delta t_2$, $t_2 + \Delta t_2 < t_1 + \Delta t_1$ (as shown in Figure 14(a)). In this case, if the messages are processed according to the causal order, it will violate the order stipulated by messages’ timestamps. However, this problem can be resolved by using lookahead information and a buffering mechanism as shown in Figure 14(b). With lookahead, a process can delay the sending of a message to a later time. As shown in Figure 14(b), since $t_2 + \Delta t_2 - L_1 < t_1 + \Delta t_1 - L_1$, $m_2$ is now sent before $m_1$. Thus, the causal order between $m_1$ and $m_2$ are effectively guaranteed to be consistent with their timestamps.

In addition, the causality based TM mechanism is incapable of handling the situation when there are hidden dependencies among the events [Fujimoto and Weatherly 1996]. Suppose a federation has one aircraft and two tanks. The aircraft is ordered to fire at the first tank moving into its vision. Within a certain short period of time, the two tanks move into the vision of the aircraft one after another. Since the update messages from the tanks are causally unrelated, the aircraft may perceive the arrival of two tanks in a different order, resulting in a situation where the aircraft fires at the wrong target.

The order in which the messages are delivered to federates using the causality based TM mechanism is not fully deterministic or repeatable. It depends on properties such as communication delays for concurrent events, and may be different from one execution to another, since communication delays may change from one execution to the next. However, it is sometimes important that the repeated execution of simulations produce the same result. For example, for many analytic simulations, the repeatability of the simulation results is required to verify their authenticity [Fujimoto 2000].

The ordering limitations of the causality based TM mechanism can be addressed by extending causal order to total order. Additional information such as a time stamp can be appended with messages to specify explicitly dependencies among the events that cannot be expressed by the “happen before” relationship [Birman

---

2Assume that $L_1$ is process $P_1$’s lookahead so that $\Delta t_1 > L_1$ and $\Delta t_2 > L_1$. 

et al. 1991]. However, the determination of when to process a message and how to ensure a federate will not receive a message with a time stamp less than its current time is not straight-forward in general and would require additional synchronization among the federates.

The communication overhead of the proposed causality based TM mechanism, which is at least of the order of $N$ for a federation with $N$ federates, can be still high for large distributed simulations with many constituent simulators. Further reductions on the communication overhead for ensuring causality could be achieved through compression of vector clocks, or other efficient implementations of vector time. For example, Singhal proposes an efficient technique to maintain vector time, which reduces the communication overhead due to propagation time of vector time by sending only incremental changes in the time stamp [Singhal and Kshemkalyani 1992].

The size of causal vector can be further reduced as well, by exploiting the topology of the underlying communication network. For instance, in distributed systems where communications among the processes are through message broadcast, immediate predecessors of a message corresponding to different processes are identical. In this case, the causal vector of the message only needs to contain one entry of less than $N$ tuples and the average size of causal vectors is less than the order of $N$.

5.2 Applicability of the Causality Based TM Mechanism

The CO and TSO mechanisms are two alternatives to eliminate temporal anomalies occurring in distributed simulations. The TSO mechanism provides a total message ordering services to federates, but incurs a large amount of communication traffic and imposes constraints on the simulation progress of federates. The CO mechanism can speed up their execution, but only provides partial message ordering services. Thus, there is a trade-off of degree of message ordering and execution speed of simulations between the CO and TSO mechanisms. The choice between the two alternatives would therefore depend on how accurately the temporal relationships among the events need to be reproduced and how fast the federates are expected to be executed.

For some DVE applications, the cause and effect relationships among the events are important to achieve the objectives of those applications. But they do not need to reproduce the temporal relationships among the events exactly and the anomalies occurring due to the different process order of causally concurrent events are not perceptible, or if perceptible, may not compromise the overall simulation objectives. The causality based TM mechanism may be a better choice than the RO or TSO mechanisms for those applications.

In collaborative applications or groupware real-time applications, such as multiplayer Internet games and shared window systems, participants need to exchange real-time data (messages) over a communication network. The execution speed and communication efficiency of such applications cannot be reduced, as this would cause a great degradation of their services. For example, in teleconferencing systems, analog video signals are sampled at a source site and produce a continuous flow of messages. In order to reconstruct a high quality analog signal at the destination site and to achieve good interactivity, messages should be delivered as quickly as possible and in order. This kind of application needs good interactivity and

real-time performance of their execution. They may tolerate a message ordering less stringent than time stamp ordering. The TSO mechanism cannot be used in these applications due to its high latency and communication overhead and the RO mechanism is not sufficient to ensure the service quality of those applications. The CO mechanism is able to cope with their ordering requirements efficiently.

Time stamp ordering has been used in analytic simulations for a long time, for its accuracy in reproducing the temporal relationships among the events, despite its high latency and communication overhead. However, in some analytic simulations such as those used in functional testing of robots, circuits and chips, the before and after relationship among the events are more important than the exact time interval between events. Suppose a simulation is testing the functionality of the robots. This simulation is made up of simulators modeling the robots and physical entities that can trigger actions on the robots in the real world. The purpose of the simulation is to test whether the robots represented by the corresponding simulators take appropriate actions when receiving external events. The cause and effect relationships are more important than the exact time interval between events, for evaluating the functionality of the robots. In addition, a large arbitrary delay in message transmission cannot be tolerated, as this would cause the simulation to deviate from the real world. This kind of analytic simulation may use the CO mechanism instead of the TSO mechanism.

In most other analytic simulations where ordering of causally concurrent events is critical or there are important hidden dependencies among the events, the TSO mechanism should be used instead. It should also be applied to simulations requiring repeatability for event process order. Scenarios where a deterministic or total process order of events is needed and/or the CO mechanism cannot meet the ordering requirements of simulations have been discussed in the previous section.

Another principal advantage of the CO mechanism over the TSO mechanism is that it does not require simulations to specify the value of lookahead. The TSO mechanism performs poorly when applied to simulations with zero lookahead. In some analytic simulations and real time DVE applications, such as simulations testing the functionality of robots and the teleconferencing system cited above, messages generated by a participant federate represent events that take place immediately at that federate and should be assigned a time stamp equal to the current time of the federate. Such applications have zero lookahead and it is more appropriate to use the CO mechanism in these cases.

6. CONCLUSIONS

Time Management is an important issue that must be addressed in distributed simulations. It has been shown that the existing TM mechanisms implemented in the HLA RTI fall short when applied to some categories of simulations. A new causality based TM mechanism is proposed in this paper. It uses a modified SES protocol to provide simulations with causal order delivery service of messages. Using the direct dependency tracking technique, the MSES protocol minimizes the size of the causal control information propagated with messages.

The MSES causality based TM mechanism has been implemented with the DMSO RTI. Experiments have been conducted to benchmark the performance of
the mechanism against that of the two existing TM mechanisms provided in the
DMSO RTI in various situations. The results show that the CO mechanism incurs
only a small amount of latency and performs almost as well as the RO mechanism
in all simulation scenarios with different degrees of inter-federate dependency. The
TSO mechanism only performs better than the RO or CO mechanisms in a closely
coupled federation of two federates. When the federation population increases from
two, the latency incurred by the TSO mechanism increases much faster than that
of the RO or CO mechanisms, in all of the three scenarios.

The scalability of the CO mechanism is also studied in comparison with the TSO
mechanism. The experimental results show that the CO mechanism is not affected
by the network load situation; whereas background network traffic has a significant
impact on the TSO mechanism. The results also show that CO is much more
scalable than TSO especially under heavy network load situations.

The trade off between degree of event ordering and execution speed of simulation
for the CO and TSO mechanisms have been discussed in this paper. The CO mechan-
ism does not require federates to specify the value of lookahead and allows them
to enhance their execution speed by reducing the synchronization activities carried
out among them. However, it only provides partial message ordering services and
cannot be used by applications where dependencies among the causally concurrent
messages are important for simulation executions.

Two possible dimensions for future research work have been suggested in this
paper. One is to extend causal order to total order through appending messages
with time stamps as well as causal control information. The second dimension is
toward further reductions on the control information, by incorporating efficient
implementations of vector time or decreasing the size of causal vectors according
to the topology of the underlying communication media.

ACKNOWLEDGMENTS
The authors are grateful for the help provided by Mr. J CHEN who implemented
the CO mechanism as a middleware on top of the DMSO RTI, Mr. Hanfeng ZHAO
who collected most of the data used in the paper, and Mr Yi ZENG who provided
help on formatting the final version of this paper. They would also like to extend
their gratitude to the reviewers of the early version of this paper for providing
valuable comments and suggestions on this research work.

REFERENCES
Georgia Institute of Technology, September.
Multicast. ACM Transactions on Computer Systems 9, 3 (August), 272-314.
FDK, 1996. Federated Simulations Development Kit (FDK), Georgia Institute of Technology.
Australian Computer Science Communication 10, 1 (February), 56-66.
(December), 398-400.
Parallel and Distributed Computing. Wiley.


Received Month Year; revised Month Year; accepted Month Year.