Critical Causal Order of Events in Distributed Virtual Environments

SUIPING ZHOU, WENTONG CAI, STEPHEN J. TURNER, and BU-SUNG LEE
Nanyang Technological University
and
JUNHU WEI
Xi'an Jiaotong University

We investigate the causal order of events in distributed virtual environments (DVEs). We first define the critical causal order relation among the events. Then, we propose some mechanisms to enhance the prevalent RO (receive order delivery) mechanism in DVEs so that the real-time property of DVEs is preserved while the critical causal order violations are reduced. These mechanisms are implemented as a middleware. Experimental results show that the middleware performs well in reducing the critical causality violations in simulation and incurs little processing overhead.

Categories and Subject Descriptors: I.6.8. [**Simulation and Modeling**]: Types of Simulation—*Distributed*; I.6.7. [**Simulation and Modeling**]: Simulation Support Systems—*Environments*

General Terms: Performance, Human Factors, Algorithms

Additional Key Words and Phrases: Causal order, distributed simulation, virtual environments

ACM Reference Format:

Zhou, S., Cai, W., Turner, S. J., Lee, B.-S., and Wei, J. 2007. Critical causal order of events in distributed virtual environments. ACM Trans. Multimedia Comput. Commun. Appl. 3, 3, Article 15 (August 2007), 20 pages. DOI = 10.1145/1236471.1236474 http://doi.acm.org/10.1145/1236471.1236474

1. INTRODUCTION

With the rapid development of Internet technologies and the fast-growing number of users on the Internet, more and more distributed virtual environments (DVEs) are implemented on the Internet [Balikhina et al. 2002; Singhal and Zyda 1999]. A DVE is a distributed system that allows multiple geographically distributed users to interact simultaneously in a computer-generated shared virtual world. Some typical examples of such systems include NPSNET [Macedonia et al. 1994], MASSIVE [Greenhalgh and Benford 1995], Diamond Park [Anderson et al. 1995; MERL 1995], and DIVE [Frecon and Stenius 1998; Hagsand 1996].

One important issue in DVEs is causality. As a fundamental concept, causality plays an important role in people's perception and knowledge of their living environment and the world. Actually, the

Authors' addresses: S. Zhou (contact author), W. Cai, S. J. Turner, B.-S. Lee, School of Computer Engineering, Nanyang Technological University, Nanyang Avenue, Singapore 639798; email: ASSPZhou@ntu.edu.sg; J. Wei, Systems Engineering Institute, Xi'an Jiaotong University, No. 28 Xianning West Road, Xi'an, Shaanxi, 719949 P. R. China.

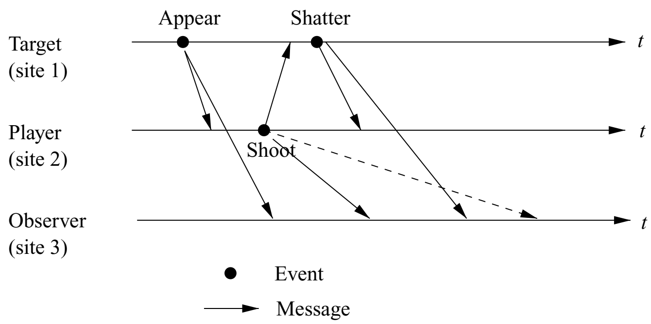


Fig. 1. A simple shooting game.

central aim of many studies in the physical, behavioral, social, and biological sciences is elucidation of the cause-effect relationship among variables or events [Collins et al. 2004; Michotte 1946; Pearl 2000]. Human beings show a strong propensity to perceive causality between cooccurring events. For example, consider a ball (A) that moves toward a stationary ball (B) until they are adjacent, at which point A stops and B starts moving along the same path. In this example, an observer sees A as *causing* B's motion, though what actually reaches the observer's eye is nothing more than the movement of the two balls at different times and locations. In other words, the observer *attributes* causality to objects or events. An interesting consequence of causal perception is that it may create illusions or distort our perception of events. For instance, Schlottmann and Schanks have reported that causal perception can contradict casual judgements 1992. In Scholl and Nakayama [2002], the authors have shown that contextual information can affect causal perception. In another paper [Scholl and Nakayama 2004], they also show that spatial relations between objects could be misperceived by causal perception. It is interesting to note that Cavazza and his collaborators took advantage of causal perception in their VR art project [Cavazza et al. 2005] to create *artificial causality* among events.

As pointed out by many researchers in philosophy and psychology, causal perception is an interesting, yet complex, issue. In this article, we do not aim to discuss causality in a broad sense. Instead, we limit our scope in two ways. First, we focus on the temporal component of causal perception. Although other components such as spatial relations, semantics, etc., may also play a role in causal perception, we believe that temporal relations among events play a major role in a person's causal perception in a DVE. Therefore, our concept of causality in this article is limited to the order of events as proposed by Lamport in a classic paper on distributed systems [Lamport 1978]. Second, instead of studying the empirical evidence for legitimate inference of cause-effect relationships, we focus on mechanisms to reduce the causality violations in DVEs. Readers interested in causality in a broader sense are referred to the book by Collins et al. [2004].

In daily life, it seems clear that things (events) happen according to their causal order; the cause (event) must happen before the effect (event). However, this may not be the case in a DVE. Due to the real-time requirement, *receive order delivery* (RO for short) is widely used in DVEs. With RO, when a site receives a message, the message is processed immediately, that is, messages are processed according to their receiving order. However, due to the large and nondeterministic message transmission delay in the network (e.g., the Internet), the causal order of events may be violated with RO. For example, consider a simple scenario in a distributed shooting game (see Figure 1): The player shoots at the target when it appears, causing it to shatter, and an observer watches the game. Suppose that the "shoot" message is delayed in the network to reach the observer's site, causing it to be received after the "shatter" message at the observer's site (as shown by the dashed arrow). Thus, the observer sees the target being destroyed

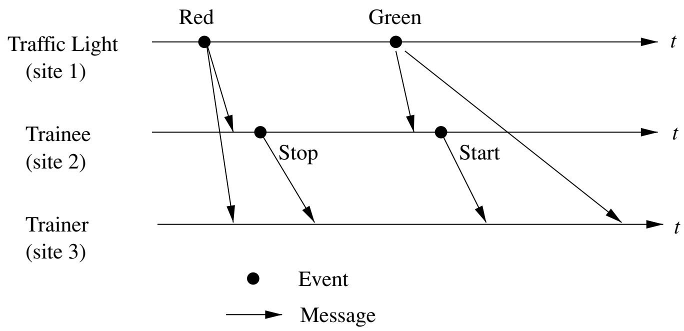


Fig. 2. A distributed training environment.

before the player shot at it, which obviously contradicts the observer's real-life experience. Infrequent occurrences of such phenomena may not be a serious problem to the game. However, if this kind of phenomenon occurs frequently, it may seriously affect a human participant's perception of the virtual environment.

In addition, causal order violations may also damage a participant's decision-making and the interactions among the participants in a DVE. For example, consider a scenario in a DVE for training drivers, as shown in Figure 2: The trainee stops the car when the traffic light turns red, and starts the car when the traffic light turns green. Suppose that the "green" message is delayed in the network, causing it to be received after the "start" message at the trainer's site. Thus, the trainer will think that the trainee has violated the traffic rule, and may issue a warning to the trainee or some of the trainee's points may be deducted. This may damage the training effect and may even puzzle the trainee. For example, the trainee may doubt whether it is right to start the car when the traffic light turns green. Note that the scenario in this example is rather simple. For more complex scenarios, such as when the trainee is learning to act with a new weapon system under various tactical situations, frequent occurrences of wrong warnings due to causal order violations may even cause the trainee to form bad habits and wrong knowledge of the weapon system and tactical rules. Therefore, causal order violations need to be reduced as much as possible in a DVE.

Various event delivery mechanisms have been proposed to preserve the causal order of events in distributed systems. For example, *causal order delivery* (CO, for short) [Cai et al. 2002] and other schemes [Birman et al. 1991; Sun et al. 1998] are proposed to preserve the causal order of events. However, most of the work in this area is focused on logical time systems and based on Lamport's *happened before* relation [1978] and the vector clock. According to the *happened before* relation, the *causal history* [Schwarz and Mattern 1994] of an event e includes all *potential* causal events of event e. Thus, in order to preserve the causal order of events, upon receiving an event, a site must make sure that all the events in the causal history of the received event have already been processed before processing the received event, which may cause a large delay in processing the received event. Therefore, the existing mechanisms are not generally suitable for DVEs. Interestingly, to the best of our knowledge, little work has been done on the event delivery mechanisms to preserve the causal order of events in DVEs.

In general, a tradeoff between real-timeness and causal order of events is necessary for DVEs. In this article, we aim to develop some novel event delivering mechanisms for DVEs which preserve the real-time property of the prevalent RO mechanism in DVEs, while simultaneously reducing the causal order violations in the DVEs. We argue that *potential* causes are different from *real* causes. The real cause of an event is the necessary condition (event) of the event, that is, "event e_1 is the cause of event

e_2" actually means that "event e_2 cannot happen unless event e_1 has already happened" or "event e_2 is dependent on event e_1". In DVEs, only the real cause-effect relation is significant to human participants. For example, in the shooting game shown in Figure 1, the causal order of the "shoot" and "shatter" events is important to the observer, thus needs to be preserved. However, before shooting at the target, the player may change his/her posture for a better aim. At the observer's site, whether the "change posture" event is processed before or after processing the "shatter" event does not have significant impact on the observer's perception of the shoot-shatter relation. This suggests that the *happened before*-based causal relation is too strong in characterizing the cause-effect relation in DVEs. Note that the real cause of an event is only a subset of the causal history of the event. Thus, if we only preserve the real causal relation, that is, we only make sure that the real causal events (rather than the whole causal history of the received event) have already been processed before processing the received event, the processing delay of a received event will be reduced, which is very beneficial to DVEs.

The rest of the article is organized as follows. In Section 2, a causal order model for DVEs is proposed. In Section 3, causal-receive order delivery is proposed to preserve both the real-time property of DVEs and the critical causal order relations specified by the causal order model. We also propose some mechanisms to enhance the prevalent RO mechanism in DVEs so that the real-time property of DVEs is preserved while are critical causal order violations are reduced. A middleware implementation of the proposed mechanisms is described in Section 4. Section 5 describes experiments with the middleware. Related work is discussed in Section 6. Conclusions are summarized in Section 7.

2. A CAUSALITY MODEL FOR DVEs

2.1 System Model

We regard a DVE as a distributed system comprising N different *sites, that is, site* 1 *site* 2, ... , *site* N, and assume that a single simulation *process* runs on each site.[1] Each simulation process is responsible for maintaining the state of one or more *objects/entities* during the simulation execution, as well as representing the state of other objects/entities (maintained by other simulation processes) that are relevant to it.

Each simulation process progresses *autonomously* and *asynchronously* according to a fixed frame rate (e.g., 25 frames per second). In each frame, various computation and communication tasks are conducted, for example, receiving user's inputs, updating states, and sending updates to other processes. A process need not synchronize with other processes to advance to the next frame. A simulation process may receive inputs from the user and models the behavior of entities in response to inputs. When this behavior causes updates that are relevant to entities maintained by other simulation processes, the simulation process will generate messages to notify other related simulation processes about theses updates.

Occurrences of actions performed at a site are called *events*. Users' inputs, collisions of objects, explosions, etc., are examples of events. However, periodic updates of the states of objects/entities in the virtual environment are not regarded as events. As these updates often occur periodically and incrementally (i.e., one such update has little impact on the system), in the following discussions we do not consider the messages containing state updates of objects, and focus instead on the causal order relation between events. We further classify the events in a DVE as either *local* or *global*, as follows.

Definition 2.1. A local event is an event which is generated at a site and not sent to other sites. A global event is an event which is generated at a site and sent to other sites.

[1]In this article, the terms *site* and *process* are used interchangeably.

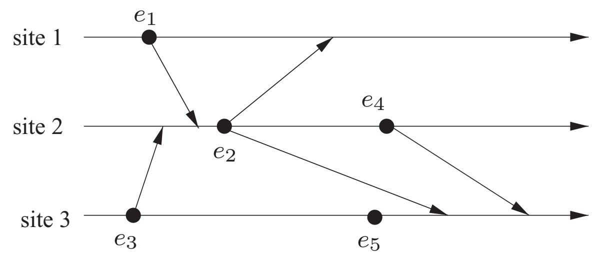


Fig. 3. Critical causal pairs in DVEs.

As an example, e_1, e_2, e_3, and e_4 in Figure 3 are global events, and e_5 is a local event at site 3.

We assume that the processing time of an event is negligible as compared to the frame time. An event is generated at a particular instant of time and assumed to be atomic, namely, at any moment, only one event may occur at a site.

Processes communicate with each other solely by *asynchronous message passing* through the network, that is, a process does not wait for the acknowledgement of message delivery when sending a message. We assume that multicast[2] is supported, thus the same message can be sent to an arbitrary set of destinations in one sending operation. Messages may be processed/delivered according to a certain order after they arrive at their destinations. We further assume that the underlying communication channels are reliable and adopt FIFO (i.e., messages arrive according to the order in which they are sent). Messages in transmission may suffer from arbitrary, but finite, nonzero delays.

2.2 Critical Causal Pair

As we pointed out in the Introduction, the causal order preservation schemes adopted in logical time systems are not generally suitable for DVEs. The reason is that *happened before*-based causal relations among events are too strong for DVEs. Current causal order preservation schemes try to detect and preserve *all* the potential causal relations among events, even though some are so weak that violations of these relations have little or no impact on users' perception. This suggests that in DVEs, only the actual causal relations need to be preserved. Furthermore, some psychological studies have demonstrated that the causal link between two consecutive events, as perceived by human subjects, would break if the two events were separated by a large delay (several hundred milliseconds) [Buehner and May 2003; Michotte 1946]. Thus, it is necessary and possible to define a new causal relation in DVEs which relaxes the *happened before*-based causal relation.

It is easy to know that all local events are properly ordered. Thus, we only need to consider the ordering of global events. Based on the assumption of FIFO message delivery, we have the following observation.

Observation 2.1. The ordering of global events generated at the same site will be preserved at those sites where these global events are received.

Thus, as far as the ordering of events is concerned, we only need consider the ordering of the global events that are generated at different sites. For the rest of this article, all the discussions on causality are focused on the causal relations among these events.

Following Lamport's *happened before* relation [1978], we define a causal order relation on the events in a DVE as follows.

[2]We do not assume a broadcast environment.

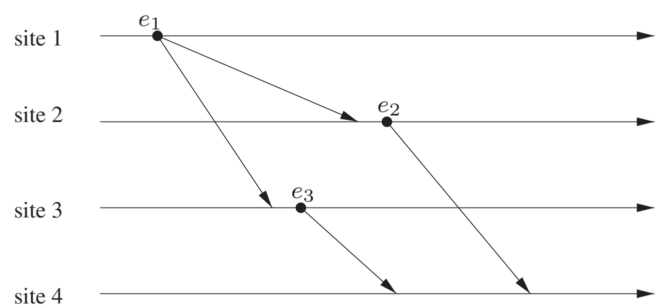


Fig. 4. Multiple responses.

Definition 2.2 (*Causal Order Relation "$\rightarrow$"*). Given two global events e and e' generated at site i and site $j(\neq i)$, respectively, $e \rightarrow e'$ iff either: (1) The message containing e is processed at site j before the generation of e'; or (2) there exists a global event e'' such that $e \rightarrow e''$ and $e'' \rightarrow e'$.

Now we define the *critical causal pair* and *critical causal order* "$\stackrel{c}{\rightarrow}$" as follows.

Definition 2.3 (*Critical Causal Pair* and *Critical Causal Order* "$\stackrel{c}{\rightarrow}$"). Let e and e' denote two global events generated at sites. i and $j(j \neq i)$, respectively. If e' is the first global event generated at site j after processing event e and there is no other global event processed between event e and event e', then we call e and e' a critical causal pair, and e is called the *predecessor* of e'. The temporal relation between e and e' is called critical causal order, denoted as $e \stackrel{c}{\rightarrow} e'$.

As a simple example, events e_1 and e_2 in Figure 3 are a critical causal pair. However, events e_1 and e_4, as well as events e_3 and e_2, are not critical causal pairs.

The philosophy behind the preceding definitions is: (1) At a site j, after processing an event e sent from another site, generally, the first global event e' generated at site j is the direct "effect" corresponding to event e; and (2) supposing that e'' is another global event generated after event e' at site j, the order between event e and event e'' is preserved at all sites if the order between e and e' is preserved at all sites.

It should be noted that the critical causal order is defined only on a critical causal pair, thus the critical causal order relation is not transitive, that is, $e_1 \stackrel{c}{\rightarrow} e_2$ and $e_2 \stackrel{c}{\rightarrow} e_3$ does not imply $e_1 \stackrel{c}{\rightarrow} e_3$.

In a distributed virtual environment, one (cause) event may be multicast to multiple sites causing multiple (effect) events at different sites, as shown in Figure 4. In this case, since the manner in which a site will respond to the processed event e_1 is dependent on the site itself, e_2 and e_3 are regarded as two different events. Thus, $e_1 \stackrel{c}{\rightarrow} e_2$ and $e_1 \stackrel{c}{\rightarrow} e_3$ in Figure 4 are treated as two different critical causal pairs.

Obviously, for any critical causal pair $e \stackrel{c}{\rightarrow} e'$, e is processed before e' at the site where event e' is generated. In order to avoid a critical causal order violation, the ordering of e and e' needs be preserved at all other sites where these two events are processed. Thus, we characterize the causality problem in DVEs as follows.

Definition 2.4. The critical causal order is preserved in a DVE if for any critical causal pair $e \stackrel{c}{\rightarrow} e'$, e is processed before e' at all sites where e and e' are received. Otherwise, the critical causal order is violated in the DVE.

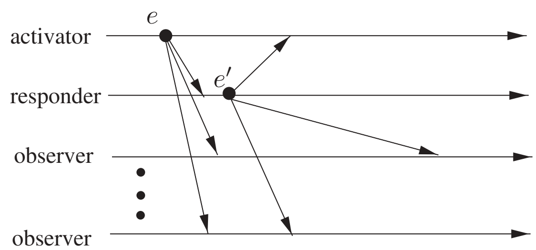


Fig. 5. A typical scenario in DVEs.

3. EVENT DELIVERY IN DVES

3.1 Causal-Receive Order Delivery

As mentioned in the Introduction, the causal order of events may be violated with RO due to the large and nondeterministic message transmission delay in the network. This may seriously affect a human participant's perception and decisions. On the other hand, the CO mechanism may induce large processing delays, thus are not generally suitable for DVEs. In this section, a new delivery order namely *causal-receive order delivery*, is proposed to reduce the influence of causal order violations on users' perceptions, meanwhile keeping the real-time property of the system.

Definition 3.1. A DVE is called a *causal-receive order delivery* (CRO, for short) system if the following properties are satisfied:

(1) Upon receiving a message, the message is processed immediately; and
(2) critical causality is preserved, that is, for any critical causal pair $e \xrightarrow{c} e'$, e is processed before e' at all sites where e and e' are received.

Remark 3.1. CRO can be regarded as a tradeoff between RO and CO. With RO, messages are processed according to their receive order, namely, processed immediately without caring about the causal relations among events. With CO, messages are processed according to the causal relations among the events, namely, causal relations among the events are preserved at all sites. However, CO may cause a large delay in processing a received message. Property 1 of CRO is to meet the real-time requirement of DVEs, and property 2 is to help avoid causality violations in the systems. Thus, with CRO, we attempt to preserve only the most significant causal relations, while keeping the real-time property of DVEs.

3.2 Enhanced Receive Order Delivery Mechanisms for DVEs

As it is difficult (if not impossible) to achieve CRO, we propose some novel event delivering mechanisms which preserve the real-time property of the prevalent RO mechanism in DVEs, simultaneously, reducing the critical causal order violations in DVEs. We call these mechanisms *enhanced receive order delivery mechanisms* (ERO for short) in this article.

Consider a typical scenario in a DVE, as shown in Figure 5: A site (*activator*) generates an event e (cause) causing another event e'(effect) at another site (*responder*), and other sites (*observers*) receive both of these two events.[3] Obviously, $e \xrightarrow{c} e'$. We further denote $m\{e\}$ and $m\{e'\}$ as the messages carrying event e and event e', respectively.

[3]The roles of activator, responder, and observer may change, that is, they are dynamic according to different actions.

Suppose that RO is adopted at all sites. It is easy to know that at the activator's or responder's the causal order of this critical causal pair is preserved, namely, e is processed before e'. Thus, we need only consider the order of e and e' at an observer's site.

At an observer's site, there are two cases regarding the arrival order of $m\{e\}$ and $m\{e'\}$:

Case 1. $m\{e\}$ arrives first. In this case, the causal order of e and e' is preserved simply by using the RO.

Case 2. $m\{e'\}$ arrives first. In this case, RO will violate the causal relation of e and e'.

To address the causal order violation problem in Case 2, we introduce an event counter to each site. The counter is initialized to zero, and is increased by one whenever a global event is generated at the site. We then introduce a new type of protocol message $m\{e, e'\}$ at the responder's site: Instead of sending $m\{e'\}$, this new type of message $m\{e, e'\}$ is sent to the sites which are common destinations of e and e'. Specifically, $m\{e, e'\}$ is defined as follows:

e	$S(e)$	$T(e)$	e'	$S(e')$	$T(e')$

Protocol message

Note that $S(e)$ is the identification number of the site where event e is generated, and $T(e)$ is event number of event e at site $S(e)$.

Regarding the protocol message, there are two special cases: (1) If event e' is the first global event generated at a site without processing any event from other sites, then it has no predecessor; (2) if a site is the destination of event e' but not the destination of event e, then it is meaningless for the responder to carry event e in the protocol message. In these two cases, we define the protocol message as follows.

NIL	NIL	0	e'	$S(e')$	$T(e')$

Note that the NIL event is assumed to be generated at a virtual site (NIL) and needs not be processed.

Upon receiving a message $m\{e, e'\}$, a site processes the events carried in this message *immediately* in the following way:

> If e is NIL, process event e' immediately. Otherwise, process event e, then process event e' immediately.

Thus, in Case 2, with the help of the protocol message, we can still process e before e' at an observer's site, that is, the critical causal relation $e \xrightarrow{c} e'$ is preserved.

The problem with the protocol message is that an event may be sent to a more than once (from different sites). Thus, an event may be processed at a site for more than once. Virtual clocks can be used to address this problem. As shown in [Zhou et al. 2002], the virtual clock C^i at site i is defined as a vector with $N-1$ elements: $(C^i_{j_1}, C^i_{j_2}, \cdots, C^i_{j_{N-1}})$, where $j_1 < j_2 < \cdots < j_{N-1}$ are integers between 1 and N, and $j_k \neq i, k = 1, 2, \cdots, N-1$. At the beginning of the application, all elements of C^i are set to zero. At site i, upon receiving a message $m\{e, e'\}$ where event e is generated at site j, if $C^i_j \geq T(e)$, then event e has already been processed and therefore will be discarded; otherwise, after processing event e, C^i_j is set to $T(e)$.

Though the aforementioned mechanism ensures that an event is processed no more than once at any site, it may introduce another problem: An event may be discarded without being processed, as shown in Figure 6. In the figure, after processing event e_2 (carried in message $m\{e_2, e_4\}$) at site 4, $C^4_1 = T(e_2)$. Upon receiving $m\{e_1, e_3\}$, since $C^4_1 \geq T(e_1)$, event e_1 will not be processed and thus missed at site 4.

To address this problem, instead of using the aforementioned virtual clock, each site maintains $N-1$ vectors which correspond to the other $N-1$ sites. For example, at site i, the vector that corresponds to

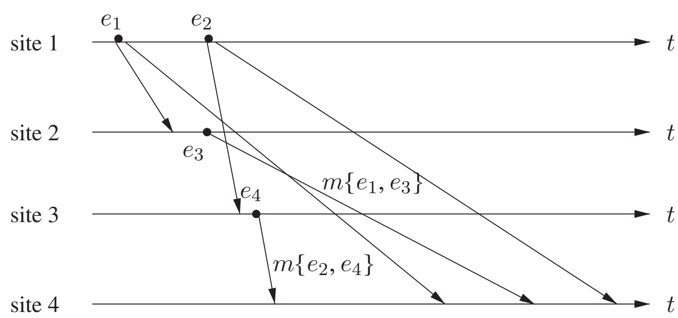


Fig. 6. Miss of an event at a site.

site j (denoted as T_j^i) is defined as follows:

$$(T_j^i(1), T_j^i(2), \cdots)$$

The length of T_j^i is always no less than two. The elements of T_j^i record the event numbers of the events that are sent from site j and have been processed at site i. To minimize the length of T_j^i, $T_j^i(1)$ and $T_j^i(2)$ are used to record the lower bound and upper bound on the event number for the events generated from site j: An event with event number less than $T_j^i(1)$ must have already been processed at site i; an event with event number greater than $T_j^i(2)$ must not have been processed at site i; for an event with event number between $T_j^i(1)$ and $T_j^i(2)$, other elements of T_j^i need to be checked to decide whether the event has been processed at site i. At the beginning of the application, T_j^i contains only two elements, with both $T_j^i(1)$ and $T_j^i(2)$ set to zero. At site i, after processing an event, the corresponding vector may need to be updated depending on whether the event has already been processed. We propose the following mechanism to decide on whether an event has been processed and to update the corresponding vector.

Mechanism 3.1. Suppose event e is generated at site j and received at site i.

—If $T(e) \leq T_j^i(1)$ or $T(e) = T_j^i(2)$, then e has been processed at site i.

—If $T(e) > T_j^i(2)$, then e has not been processed at site i. After processing e, append the value of $T_j^i(2)$ to the end of T_j^i if $T_j^i(1) < T_j^i(2)$, and set $T_j^i(2)$ to $T(e)$.

—If $T_j^i(1) < T(e) < T_j^i(2)$ and $T(e)$ is not an element of T_j^i, then e has not been processed at site i. After processing e, append $T(e)$ to the end of T_j^i.

—If $T_j^i(1) < T(e) < T_j^i(2)$ and $T(e)$ is an element of T_j^i, then e has been processed at site i.

—Simplify the vector T_j^i recursively as follows: If an element of T_j^i equals $T_j^i(1) + 1$, set $T_j^i(1)$ to $T_j^i(1) + 1$, then remove this element from T_j^i.

Based on Mechanism 3.1, we have the following ERO mechanism.

Remark 3.2. Mechanism 3.1 guarantees that an event is processed exactly once at a receiving site. By examining the event number of a newly arrived event and the corresponding vector, the problem as identified in Figure 6 can be avoided. For example, in Figure 6, after processing e_2 at site 4, $T_1^4 = (0, 2)$. Upon receiving e_1, since $T_1^4(1) < T(e_1) = 1 < T_1^4(2)$ and $T(e_1)$ is not an element of T_1^4, e_1 has not been processed at site 4, according to Mechanism 3.1. Thus, e_1 should be processed. After processing e_1, T_1^4 is updated to $(2, 2)$.

Mechanism 3.2. At a site i, upon receiving a message $m\{e, e'\}$, where e an event generated at site p, and e' is an event generated at site q, do the following.

```
if (e has already been processed)
   if (e' has not been processed)
   {
      Process event e';
      Update T_q^i;
   }
   else { /* C1 --- Possibility of CRO violation */
     do nothing;
   }
else if (e' has not been processed)
      {
         Process event e;
         Update T_p^i;
         Process event e';
         Update T_q^i;
      }
      else
      {
         /* C2 --- Violation of CRO */
         Process event e;
         Update T_p^i;
      }
```

When a global event e' is generated at site i, do the following:

```
Process event e';
For a destination d of e', if it is also the destination of e
where e is the predecessor of e', send message m{e, e'} to d;
otherwise, send message m{NUL, e'} to d.
```

Remark 3.3. Mechanism 3.2 is aimed only at preserving critical causality rather than preserving causality in the general sense. Thus, even though critical causality is preserved in an application, causality violations may still exist in the application.

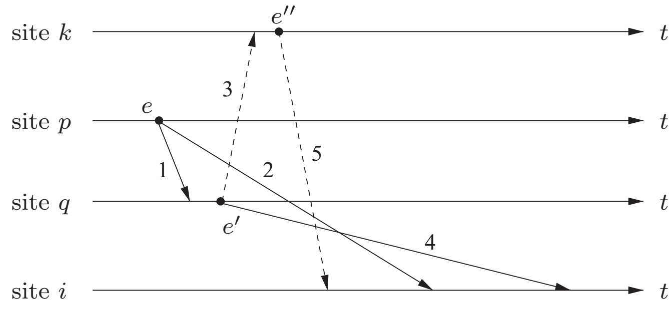
Now we discuss an important property of Mechanism 3.2.

Property 3.1. Mechanism 3.2 keeps the real-time delivery property of RO, and compared with RO, it reduces the critical causality violations in a DVE application.

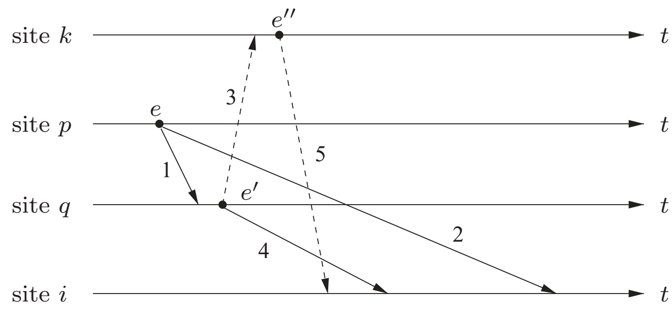
The first part of Property 3.1 is obvious, since Mechanism 3.2 processes any received message immediately. Now we show the second part of the property.

Consider a typical critical causal pair $e \xrightarrow{c} e'$, where e and e' are two global events generated at site p and site q, respectively. We consider the processing order of e and e' at another site i. There are only two possible cases regarding the arrival order of the two messages containing event e and e' at site i, as shown in Figure 7(a) and 7(b). Note that messages in the figures have different meanings for RO and ERO. For example, message 4 in Figure 7(a) only contains event e' for RO, while for ERO, it is the protocol message which contains both e and e'.

In Case 1, the critical causal order between e and e' is preserved at site i according to RO, while it *might* be violated at site i using our ERO mechanisms. However, this can only occur if there exists a



(a) Case 1



(b) Case 2

Fig. 7. Critical causal order violations.

site k such that an event e'' is generated at site k after processing e', and the message containing e'' (message 5) arrives at site i before the arrival of the message containing e (message 2), as shown by the dashed arrows in Figure 7(a) (refer to comments C1 in Mechanism 3.2). Obviously, in this case, the relation of $e' \xrightarrow{c} e''$ is violated using RO. Therefore, in Case 1, whenever there is a critical causal order violation using the proposed ERO mechanisms, there must exist a critical causal order violation using RO.

In Case 2, the critical causal order between e and e' is violated at site i according to RO. In this case, if ERO is used, $e \xrightarrow{c} e'$ is violated only if there exists a site k such that an event e'' is generated at site k after processing e', and the message containing e'' (message 5) arrives at site i before the arrival of the message containing e' (message 4), as shown by the dashed arrows in Figure 7(b) (refer to comments C2 in Mechanism 3.2). Again, in this case, the relation of $e' \xrightarrow{c} e''$ is violated using RO.

It should be noted that our mechanism has another advantage over RO. In case there is a strong reason at the application level to discard event e if e' has already been processed, we can simply drop e, as our mechanism actually checks whether e' has been processed upon receiving the protocol message $m\{e, e'\}$. However, this cannot be done by RO, as the relation between e and e' is not captured.

To summarize the analysis on the previous two cases, we conclude that the proposed ERO mechanism will reduce the critical causality violations in a DVE application.

Since our ERO mechanism does not strictly enforce CRO, experiments are needed to evaluate and compare its performance with RO in reducing the critical causal order violations and preserving the real-time property of DVEs. In Section 5, we will describe these experiments.

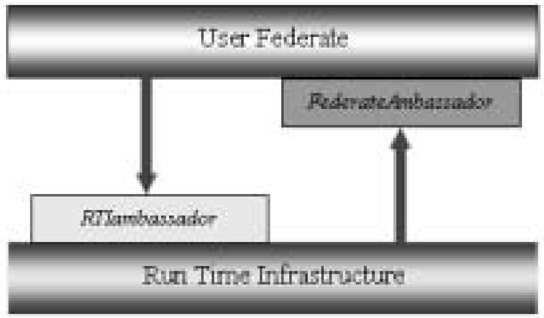


Fig. 8. The architecture of RTI-based simulation.

4. A MIDDLEWARE APPROACH TO ERO DELIVERY

As demonstrated by our experimental results in the next section, ERO helps to reduce the critical causality violations in a DVE, while keeping the real-time delivery property of RO. Thus, it is a better delivery mechanism for DVE applications as compared with both RO and CO. However, implementing Mechanism 3.1 and 3.2 is not trivial, especially for existing applications. To facilitate the use of these mechanisms and to minimize the modifications to an existing application, we implemented the proposed mechanisms as a middleware so that they can be easily incorporated into an application. Considering the widespread adoption of high level architecture (HLA) [Dahmann et al. 1998] in distributed applications, our middleware implementation of the ERO mechanisms is built on HLA services. In this section, we describe the implementation details of the middleware.

4.1 Introduction to HLA

The HLA provides an architecture for linking distributed simulation components (known as *federates* in an HLA *federation*). The main objective of the HLA is to foster the interoperability and reusability of simulation components.

The HLA is a standard rather than an implementation. The HLA run'time infrastructure (RTI) [Department of Defense 2002] implements the HLA interface specification, and is the supporting software that allows federates to interact with each other via RTI services. Figure 8 shows the architecture of an RTI-based simulation. Federates do not communicate with each other directly. Instead, a federate invokes the RTI services to exchange information with other federates. More specifically, each federate has a federate ambassador (*FederateAmbassador*) and an RTI ambassador (*RTIambassador*). A federate invokes methods on the *RTIambassador* whenever it needs certain RTI services (e.g., sending attribute updates or interactions). Conversely, the RTI invokes methods on the *FederateAmbassador* whenever it needs to call the federate (e.g., informing the federate of attribute updates or interactions sent from other federates).

In current RTI implementations [Department of Defense 2002; Pitch 2004], two time management mechanisms are supported, namely, *receive order* (RO) and *time stamp order* (TSO). RO is often chosen for simulations that do not require strict ordering of events or real-time simulations where responsiveness is the major concern. As pointed out in Section 3, RO is unable to eliminate the temporal anomalies that may occur in a DVE application due to the nondeterministic message transmission delay in the network. On the other hand, TSO guarantees the ordering of events, but may induce significant overhead, as it requires the federates to be synchronized frequently to ensure that all events are processed according to their time stamp orders at all federates. Therefore, TSO is not generally suitable for DVE applications.

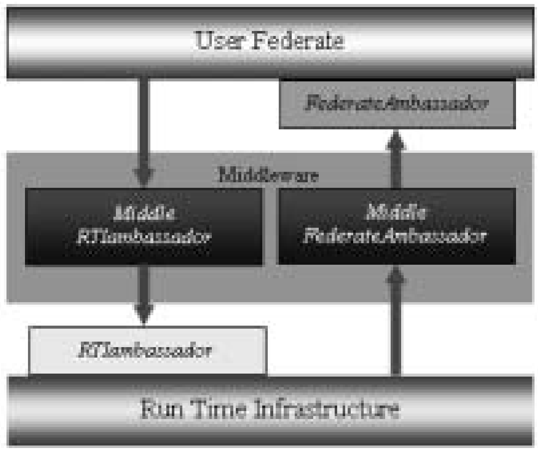


Fig. 9. System architecture with the middleware.

Our ERO mechanisms can be regarded as a tradeoff between RO and TSO. The following sections describe the middleware implementation of the ERO.

4.2 System Structure with ERO Middleware

In an HLA-based application, federates exchange information using only the services provided by RTI, which makes it possible to introduce a middleware between the RTI and federates to implement the ERO mechanisms. We extend the HLA RTI architecture by adding a middleware layer, as shown in Figure 9. The middleware consists of a *MiddleRTIambassador* class which extends the standard *RTIambassador* class and a *MiddleFederateAmbassador* class which extends the standard *FederateAmbassador* class. As shown the figure, all outgoing messages from a federate to the RTI are through the *MiddleRTIambassador*, and all incoming messages to a federate from the RTI are through the *MiddleFederateAmbassador*. The middleware intercepts these messages and performs operations as required by the ERO mechanisms.

The middleware hides the implementation complexity of the ERO mechanisms. The only change to a user federate source code is to replace the instance declaration of *RTIambassador* with an instance declaration of *MiddleRTIambassador*. All the method calls made by the federate remain exactly the same. Therefore, the middleware is almost completely transparent to a user federate.

4.3 Implementation of the ERO Mechanism

The major considerations in implementing the ERO mechanisms are: (1) how to send protocol messages; (2) how to receive and process protocol messages; and (3) how to determine the predecessor of an event. It should be noted that an event is uniquely identified in the federation by an event number and the handle of the federate that generates the event. A federate is assigned a unique handel by the RTI when it successfully joins the federation. An event is assigned an event number when the event is generated at a federate. In our implementation, the event number represents the sequential number of events that are generated in the same federate.

To address these issues, the *joinFederationExecution*(), *sendInteraction*(), and *tick*() services in the RTIambassador are overridden in the *MiddleFederateAmbassador*, and the callback service *receiveInteraction*() in the *FederateAmbassador* is overridden in the *MiddleFederateAmbassador*. Figure 10

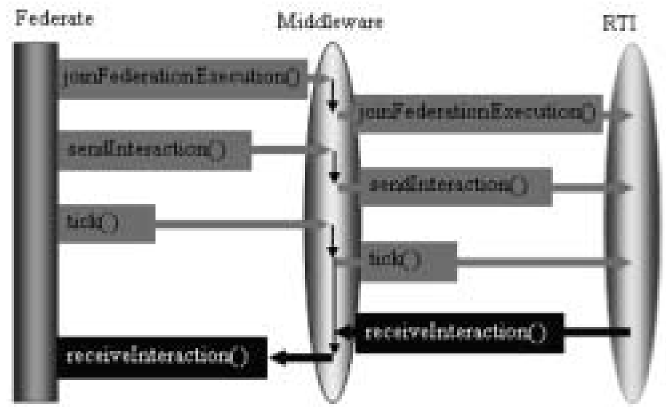


Fig. 10. Major services involved in the implementation.

summarizes the major services involved and their relationship. Following sections describe the major considerations in implementing these services in the middleware.

4.3.1 *Joining Federate Execution.* To join a federation, a federate invokes the following RTI service:

```
joinFederationExecution(
        const char* federateName,
        const char* executionName,
        RTI::FederateAmbassadorPtr userFedAmb)
```

The purpose in overriding this method is to get the federate handle and pointer to the user *FederateAmbassador*. By invoking this method, a federate tells the RTI the federate name, the federation name, and pointer to an instance of the user *FederateAmbassador* class. The RTI assigns a handle to the federate if the method invocation is successful.

4.3.2 *Sending Protocol Messages.* To send an event, a federate invokes the following RTI service:

```
sendInteraction(
        RTI::InteractionClasshandle theInteraction,
        const RTI::ParameterHandleValuePairSet& theParameters,
        const char* theTag)
```

In the *MiddleRTIambassador* class, the *sendInteration*() method is overridden to send the protocol message $m\{e, e'\}$. According to the ERO mechanisms, an event e' needs to be sent together with its predecessor e. In addition, the event number and source federate of an event need to be attached to the event. All this information is encoded in the argument *theTag* when sending $m\{e, e'\}$. When a global event e' is generated at a federate, the federate invokes the *sendInteraction*() method in the *MiddleRTIambassador* to do the following:

(1) Increment the event number by one.
(2) Construct protocol message $m\{e, e'\}$, where e is the predecessor of e'.
(3) Send $m\{e, e'\}$ by invoking the *sendInteraction*() method in the *RTIambassador*.

4.3.3 *Receiving Protocol Messages.* To notify a federate to receive an event, the RTI uses the following callback function in the user *FederateAmbassador*:

```
receiveInteraction(
        RTI::InteractionClasshandle theInteraction,
        const RTI::ParameterHandleValuePairSet& theParameters,
        const char* theTag)
```

In the *MiddleRTIambassador* class, the *receiveInteration*() method is overridden to receive and process the protocol message $m\{e, e'\}$ as follows:

(1) Get the event number and source federate of e and e' from the argument *theTag*.
(2) Check whether e and e' have already been received according to Mechanism 3.1. Ignore an event if it has already been received before. Otherwise, put the event into an event buffer which is implemented as an FIFO queue.[4]
(3) Update the vectors as described in Mechanism 3.1.

4.3.4 *Determining the Predecessor of an Event.* With the RTI architecure, a component called the local RTI component (LRC) in the user federate is in charge of buffering and delivering incoming events. By invoking *tick*(), a federate temporarily passes execution control to the LRC. When the LRC obtains control, it conducts various activities. A typical activity of the LRC is to drain the inbound event queue and provide callbacks to the federate through the *FederateAmbassador*. Thus, when a federate *ticks*, the RTI may deliver more than one event to the federate. For the purpose of transparency, the predecessor of an event needs to be determined by the middleware rather than by a user federate, hence the middleware cannot pass more than one event to a federate when the federate ticks once. Otherwise, the middleware may have difficulty determining the predecessor of an event. For example, supposing that a global event e is generated at a federate after the middleware delivers event e_1 and e_2 to the federate, the middleware has no idea of whether e is generated immediately after e_1 or e_2. Therefore, in the *MiddleRTIambassador*, the *tick*() method is overridden to do the following:

(1) Invoke the *tick*() method in the *RTIambassador*.
(2) Get the first event in the event buffer and deliver this event to the user federate. This event is the predecessor of the next new global event that is generated at this user federate.

5. EXPERIMENTS

Causality violations may occur in distributed applications, especially those deployed over the wide area network (WAN), for example, the Internet. In Section 3, we proposed the ERO mechanisms which help to reduce the critical causality violations in distributed applications. In Section 4, we implemented the ERO mechanisms as a middleware based on HLA. To evaluate the effectiveness of the proposed mechanisms and the performance of the middleware, a distributed real-time battlefield simulation was developed, and simulation experiments conducted. In this section, we describe the experiments and summarize their results.

5.1 Experiment Design

Using the RTI and our ERO middleware, we developed a distributed battlefield simulation on a PC cluster in the Parallel and Distributed Computing Center at Nanyang Technological University. The simulation model is specifically designed for the evaluation of ERO mechanisms and middleware. The emphasis of the simulation model is on the causal orders among different events, rather than on

[4]In the next section, we will explain the reason for putting the event into a buffer, rather than forwarding it directly to the federate.

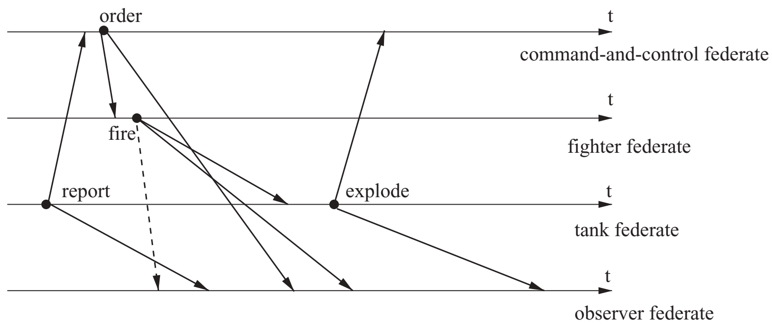


Fig. 11. A battlefield simulation.

the fidelity of the battlefield. There are four types of federates in the simulation: a *command-and-control* federate which simulates a spy plane, a *fighter* federate which simulates a group of fighter jets, several *tank* federates, each simulating a group of tanks, and an *observer* federate. Each federate runs on one computer in the PC cluster.

Each tank group consists of 100 tanks. To simulate the detecting behavior of the spy plane, each tank group periodically send reports to it. Besides other information, each report includes a tank's name as well as that of the federate to which the tank belongs. When the spy plane detects a tank, that is, receives a report from a tank group, it sends orders to the fighter group to fire a missile at the tank. Besides other information, a "fire" event includes the names of both the target tank and the federate that simulates the tank. All tank federates can receive the "fire" event, but only the federate that contains the specified tank responds to it, while other tank federates simply ignore this event. When a tank (federate) receives the "fire" event, there is 50% probability the tank will be destroyed, which will result in an "explode" event. If the tank is not destroyed, then a "miss" event will be sent to the command-and-control federate. The observer federate receives all the events in this engagement. Figure 11 illustrates various events in the simulation where there is only one tank federate. The simulation terminates when all tanks are destroyed. Simulation data collection and result computation are conducted at the observer federate.

Note that the event relations in the simulation model are general in the sense that various critical causality violations as identified in Figure 7 could occur in the simulation. For example, as shown by the dashed arrow in Figure 11, if the "fire" event is received earlier than the "report" event at the observer federate, the critical causal relation between the "report" event and "order" event is violated at the observer federate, even if the ERO mechanism is used (refer to Case 2 in Figure 7).

To evaluate the effectiveness of the ERO mechanism in reducing the critical causality violations and overhead of the ERO middleware, we implement the same battlefield simulation model with two different message delivery mechanisms:

(1) *Case_RO*. In this case, RO is used to deliver messages, and the federation is built directly on RTI without using the middleware.
(2) *Case_ERO*. In this case, the ERO middleware is used in the implementation.

Both of the two implementations use RTI 1.3NG V6 [Department of Defense 2002].

To simulate the message transmission delay in the Internet, a latency emulator is implemented between the middleware and RTI. When the emulator receives an event from the RTI, it delays the event for some time before delivering it to the middleware. The delay of an event is modeled as a random variable x which has an exponential distribution [Bertsekas and Gallager 1992; Kato et al. 1999], that is, $f_x(t) = \lambda e^{-\lambda(t-\mu)}, t \geq 0$, where $f_x(\cdot)$ stands for the probability density function of x.

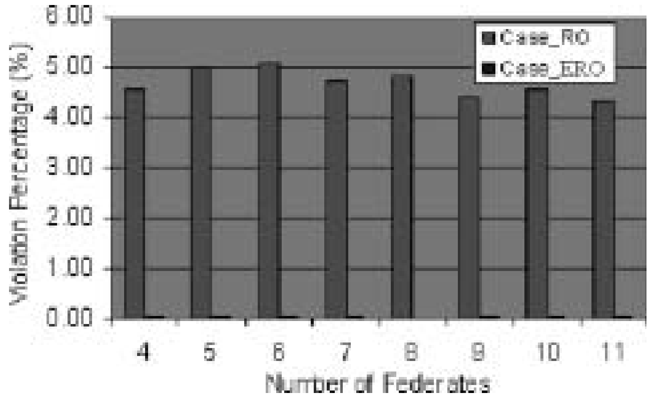


Fig. 12. The average percentage of critical causal order violations.

Table I. Percentage of ERO and CO Violations

	ERO Violation (%)		CO Violation (%)	
Implementation	Case_RO	Case_ERO	Case_RO	Case_ERO
Average	4.34	0.03	5.25	0.82
Confidence Interval	(4.26, 4.42)	(0.026, 0.034)	(5.17, 5.33)	(0.76, 0.88)

(sample size = 90, confidence level = 90%, number of tank federate = 8)

5.2 Experimental Results and Analysis

A series of experiments as conducted to compare the ERO implementation with the RO implementation under different scenarios. Besides a command-and-control federate, a fighter federate, and an observer federate, different scenarios have different numbers of tank federates. Up to 8 tank federates are used in these experiments. Due to the randomness involved in the experiments, each scenario is executed 90 times with the two implementations. Statistical analysis is conducted to compare the performance of the two implementations based on the collected data.

Figure 12 shows the average percentage of critical causality violations with the two implementations. The percentage of violations is computed as the number of critical causal order violations divided by the total number of received events at the observer federate. The results show that the ERO middleware is very efficient in reducing the number of critical causal order violations in the simulation.

Since the critical causality order relation is a subset of the causal order relation, by reducing the number of critical causal order violations, we expect that the number of causal order violations will also be reduced. Table I shows the effect of the middleware in reducing both critical causal order violations and causal order violations.

To evaluate the overhead of the middleware, the two implementations are compared by their execution times. Figure 13 shows the average execution time in simulation of the two implementations with different numbers of federates, and Figure 14 shows the 90% confidence interval of the execution time. The results show that the execution time of the two implementations increases slowly and nearly linearly with the number of federates, and that the middleware incurs only slight overhead compared with the RO delivery mechanism that doesn't use the middleware. It is interesting to note that in some (rare) cases, the execution time of the middeware implementation is even less than that of the RO implementation. This is because with the middleware implementation, an event may arrive at a destination federate more quickly due to the “event-piggyback” effect of the protocol messages of ERO mechanisms.

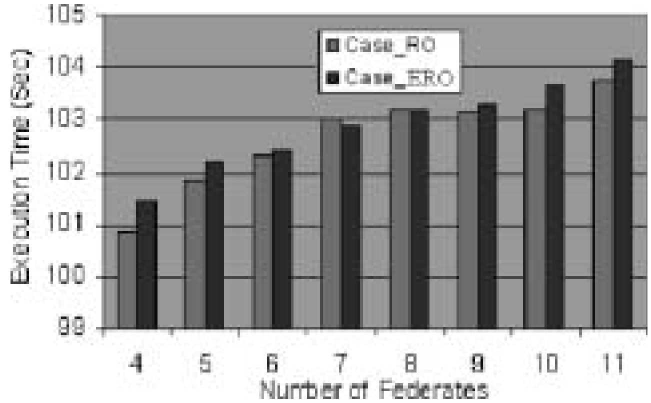


Fig. 13. The average execution time of the simulation.

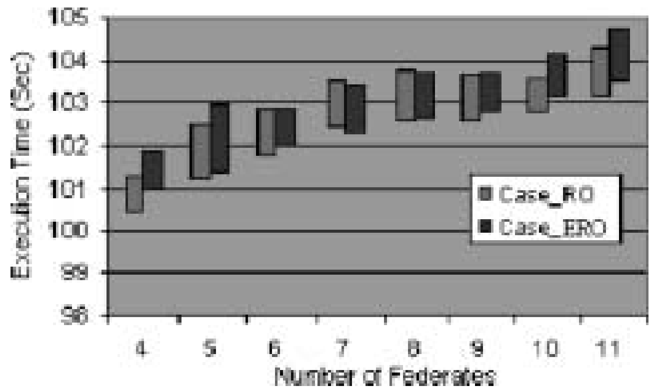


Fig. 14. The confidence interval of the execution time.

To summarize, the experimental results show that the ERO middleware is very efficient in reducing the number of critical causal order violations (as well as the number of causal order violations) in distributed applications, and incurs only slight overhead compared with the RO delivery mechanism.

6. DISCUSSION OF RELATED WORK

Due to the importance of the causal order of events in parallel and distributed systems, many schemes have been developed to capture the causal relationship among the events in such systems. In Schwarz and Mattern [1994], the authors gave a good survey on causal order and various techniques to detect the causal relations in distributed computations. Generally, *vector time* can be used [Ellis and Gibbs 1989; Sun et al. 1998] to capture the causal relationship among events in a broadcast environment. In a multicast environment, in addition to the vector clock, other mechanisms need to be introduced to capture the causal relationship among events [Birman et al. 1991]. In [Cai et al. 2002], the authors used a causal barrier to capture the causal relationship among events and developed a set of schemes to preserve the causal relation among events. These schemes address the causal order of events from a logical-time system point of view, and are generally difficult to implement in DVEs. Besides, these schemes may cause a large delay in processing a received event, thus are not generally suitable for DVEs. In [Cai et al. 2005], a causal order delivery was proposed for time management in federated simulation, which has a better performance than the widely-used time stamp order delivery mechanism in terms of reducing the processing delay of a received event. However, when the number of federates in the federation becomes large, the messages transmitted among federates need to carry a large amount of control information, which will put a heavy workload on the network and on each federate. Moreover, the processing delay of an event also increases as the number of federates in the federation increases, which may greatly degrade the performance of many DVEs with a tight constraint on the response time.

In [Zuberi and Shin 1996], the causal order of events in distributed embedded real-time systems was investigated, and a causal message ordering algorithm given. The proposed scheme extends the Δ-protocol so that instead of delaying all messages for a fixed period, each message is delayed according to its deadline. However, as the authors pointed out in [Zuberi and Shin 1996], the algorithm requires too much preknowledge of the system and requires the message transmission delays among different nodes to remain stable, thus is only suitable for small-sized real-time systems. In [Roberts et al. 1999], the authors address the causal order of events in a virtual reality system. In order to make a tradeoff between the real-time requirement of DVEs and causal order, the authors proposed a scheme aimed at maintaining only those important causal relationships in distributed virtual ball games, while ignoring those in which the application is not interested. However, the scheme needs to know all the objects that an event *will* affect, which is not applicable to many DVEs, since generally, whether an event will affect an object is nondeterministic.

7. CONCLUSIONS

The causal order of events has been widely studied in the parallel computing and distributed system community. However, most of the research has been from a logical-time system point of view, which is not generally suitable for DVEs due to the high cost of the mechanisms to preserve the causal order.

In this article, we investigate causal order of events in DVEs. First, the critical causal order $e \xrightarrow{c} e'$ is proposed. Critical causal order is the most significant causal relation in DVEs, which should be preserved at all sites. It should be noted that our critical causal order differs from *happened before*-based causal order in that we only consider the processing order for a critical causal pair. In this sense, the critical causal order relation can be regarded as a relaxation of the *happened before*-based causal order relation. Based on the critical causal order relation, causal-receive order delivery is proposed to preserve both the real-time property of DVEs and the critical causal order relations among the events. We developed enhanced receive order delivery (ERO) mechanisms which preserve the real-time property of the prevalent RO mechanism in DVEs, while reducing critical causal order violations. We also implemented the proposed mechanisms as a middleware using the HLA. Experiments have been conducted to evaluate the performance of the middleware based on a distributed battlefield simulation. The experimental results show that the middleware is very efficient in reducing the critical causality violations in the simulation, and only incurs slight processing overhead.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their very valuable comments and insightful suggestions, which helped to improve the quality and presentation of this article.

REFERENCES

Anderson, D. B., Barrus, J. W., Howard, J. H., Rich, C., Shen, C., and Waters, R. C. 1995. Building multi-user interactive multimedia environments at MERL. *IEEE Multimedia 2*, 4, 77–82.

Balikhina, T., Ball, F., and Duce, D. 2002. Distributed virtual environments—An active future? In *Proceedings of the 20th Eurographics UK Conference (EGUK)* (Leicester, UK, Jun. 11–13).

Bertsekas, D. and Gallager, R. 1992. *Data Networks.* Prentice-Hall, Englewood Cliffs, NJ.

Birman, K., Schiper, A., and Stephenson, P. 1991. Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst. 9,* 3(Aug.), 272–314.

Buehner, M. J. and May, J. 2003. Rethinking temporal contiguity and the judgement of causality: Effects of prior knowledge, experience, and reinforcement procedure. *Quart. J. Exper. Psychol. 56A*, 5, 865–890.

Cai, W., Turner, S. J., Lee, B. S., and Zhou, J. 2005. An alternative time management mechanism for distributed simulations. *ACM Trans. Modeling Comput. Simul. 15,* 2, 109–137.

Cai, W., Lee, B., and Zhou, J. 2002. Causal order delivery in a multicast environment: An improved algorithm. *J. Parallel Distrib. Comput. 62*, 1, 111–131.

Cavazza, M., Lugrin, J.-L., Crooks, S., Nandi, A., Palmer, M., and Le Renard, M. 2005. Causality and virtual reality in art. In *Proceedings of the 5th Conference on Creativity and Cognition* (London, Apr. 12–15).

Collins, J., Hall, N., and Paul, L. A. 2004. *Causation and Counterfactuals.* MIT Press, Cambridge, MA.

Dahmann, J. S., Kuhl, F., and Weatherly, R. 1998. Standards for simulation: As simple as possible but not simpler, the high level architecture for simulation. *Simul. 71,* 6, 378–387.

DMSO. 2002. RTI 1.3NG programmers guide, version 6. http://www.dmso.mil/public/.

Ellis, C. A. and Gibbs, S. J. 1989. Concurrency control in groupware systems. In *Proceedings of the ACM SIGMOD Conference on Management of Data* (May). 399–407.

Frecon, E. and Stenius, M. 1998. DIVE: A scalable network architecture for distributed virtual environments. *Distrib. Syst. Eng. J. 50*, 3.

Greenhalgh, C. and Benford, S. 1995. MASSIVE: A collaborative virtual environment for teleconferencing. *ACM Trans. Comput.-Hum. Interact. 2,* 3, 239–261.

Hagsand, O. 1996. Interactive multiuser VEs in the DIVE system. *IEEE Multimedia 3,* 1, 30–39.

Kato, J., Shimizu, A., and Goto, S. 1999. Active measurement and analysis of delay time in the Internet. In *Proceedings of the International Workshop on Parallel Processing* (Wakamatsu, Japan). 21–24.

Lamport, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21,* 7 (Jul.), 558–565.

Macedonia, M. R., Zyda, M. J., Pratt, D. R., Barham, P. T., and Zeswitz, S. 1994. NPSNET: A network software architecture for large scale virtual environments. *Presence 3,* 4.

MERL. 1995. The diamond park project. http://www.merl.com/projects/dp/.

Michotte, A. 1946. *The Perception of Causality.* Basic Books, New York.

Pearl, J. 2000. *Causality: Models, Reasoning, and Inference*. Cambridge University Press, New York.

PITCH. 2004. pRTI 1516, version 2.3 product sheet. http://www.pitch.se/prti1516/files/ptri1516_product_sheet.pdf.

Roberts, D. J., Worthington, B. G., and Sharkey, P. M. 1999. Influence of the supporting protocol on the latencies induced by concurrency control within a large scale multi-user distributed virtual reality system. In *Proceedings of the SCS Western Multi-Conference* (Jan.).

Schlottman, A. and Shanks, D. R. 1992. Evidence for a distinction between judged and perceived causality. *Quart. J. Exper. Psychol. 44*, 2, 321–342.

Scholl, B. J. and Nakayama, K. 2004. Illusory causal crescents: Misperceived spatial relations due to perceived causality. *Perception 33,* 4, 455–469.

Scholl, B. J. and Nakayama, K. 2002. Causal capture: Contextual effects on the perception of collision events. *Psychol. Sci.13,* 6 (Nov.).

Schwarz, R. and Mattern, F. 1994. Detecting causal relationships in distributed computations: In search of the holy grail. *Distrib. Comput. 7,* 3, 149–174.

Singhal, S. and Zyda, M. 1999. *Networked Virtual Environments: Design and Implementation.* Addison-Wesley.

Sun, C., Jia, X., Zhang, Y., Yang, Y., and Chen, D. 1998. Achieving convergence, causality-preservation, and intention-preservation in real-time cooperative editing systems. *ACM Trans. Comput.-Hum. Interact. 5*, 1(Mar.), 63–108.

Zhou, S., Cai, W., Turner, S., and Lee, B. 2002. Critical causality in distributed virtual environments. In *Proceedings of the 16th Workshop on Parallel and Distributed Simulation* (Washington DC, May 12–15).

Zuberi, K. M. and Shin, K. G. 1996. A causal message ordering scheme for distributed embedded real-time systems. In *Proceedings of the Symposium on Reliable and Distributed Systems* (Oct.).

Received May 2005; revised December 2005, May 2006; accepted July 2006