A Middleware Approach to Causal Order Delivery in Distributed Simulations

Stephen J. Turner
Wentong Cai
Ji Chen
School of Computer Engineering
Nanyang Technological University
Nanyang Avenue
Singapore 639798
+65 6790 4054, +65 6790 4600
assjturner@ntu.edu.sg, aswtcai@ntu.edu.sg, chenji2k@hotmail.com

Keywords:
distributed simulation, time management, causal order delivery, middleware

ABSTRACT: *This paper describes a middleware approach to the implementation of a time management mechanism based on Causal Order (CO) delivery. CO ensures that messages are delivered to the local federate according to the cause and effect relationship of the events in those messages. The middleware, which encapsulates the CO delivery algorithm, provides exactly the same interface as that defined by the HLA Interface Specification. Experimental results on applications with varying degrees of coupling show that the CO delivery algorithm incurs only a small overhead compared to the Receive Order (RO) mechanism, but achieves a significant performance improvement over the Time Stamp Order (TSO) mechanism. Thus, the CO mechanism can be used in situations where we wish to guarantee the cause and effect relationship between events and also meet the real-time requirements of the simulation. The results also show that the performance of the middleware approach is close to that of an embedded implementation.*

1. Introduction

In a distributed simulation, simulation components (or federates) of various types are executed at possibly different geographical locations, to form a common virtual environment known as a federation. The High Level Architecture (HLA) [1] is designed to promote the reuse and interoperability of federates though its rules, object model template and interface specification. The HLA Runtime Infrastructure (RTI) provides six categories of service, of which one is time management.

Ideally, distributed simulations should reproduce exactly the temporal relations among the events that occur in the real world being modeled. However, the heterogeneous delays associated with the computations and message transmissions over the network may lead to violations of such relations. For example, in a distributed simulation consisting of three federates: a tank, a target and an observer, the observer may see the target destroyed before it was fired upon by the tank, if these different federates are simulated on different computers [2].

To preserve the temporal relations among events, some kind of time management scheme needs to be used in distributed simulations. In an early version of the HLA Time Management Design Document the following five message ordering mechanisms were suggested [3]: *Receive Order* (RO), *Priority Order*, *Causal Order* (CO), *Causal and Totally Ordered*, and *Time Stamp Order* (TSO). In the versions of the RTI currently available, only two time management mechanisms are provided: *Receive Order* (RO) and *Time Stamp Order* (TSO).

RO is often chosen because of the performance requirements of the simulation. In particular, it is the mechanism generally used for real-time simulations. However, RO is unable to eliminate the temporal anomalies that may occur during the execution of a federation as a result of heterogeneous latencies in the communication network. TSO satisfies the ordering requirements, but significantly increases the time overhead as it requires the federates in the federation to be synchronized at regular intervals.

In many distributed simulations, particularly distributed virtual training environments, the "cause and effect" relationship between two events is of more importance than the exact time interval between them. Therefore, this paper describes the design and implementation of a time management mechanism based on Causal Order (CO) delivery. CO ensures that messages are delivered to the local federate according to the cause and effect relationship of the events in those messages. The CO delivery algorithm used is the Modified Schiper-Eggli-Sandoz (MSES) algorithm [4], which is designed to have only a small amount of overhead of causal information and processing time.

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 call-back 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 [5], we adopted an embedded approach by modifying the Federated Simulations Development Kit (FDK) [6], 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.

The objective of the current project is therefore to develop middleware which is inserted between the federates and the RTI and which has the following features:

- *Portability* – the middleware should be able to operate with different versions of the RTI that provide the standard RTI C++ header files.
- *Transparency* – it should be possible to link existing federates with the middleware to use CO time management, with no (or at most a tiny) modification.
- *Performance* – the performance of the middleware CO delivery algorithm should be close to that of an embedded implementation of the CO time management mechanism.

The remainder of this paper is structured as follows: Section 2 outlines the causal order delivery algorithm used in this project. Section 3 describes in detail our middleware approach to the implementation of causal order delivery. Section 4 describes the benchmarking experiments and the experimental results obtained. Finally, Section 5 presents our conclusions and future work.

2. Causal Order Delivery

A distributed simulation is an example of a distributed system where the federates act as asynchronous processes that communicate by exchanging messages. The processes do not share any memory and in general there is no common clock. The idea of causal order is based on the "happened before" relation first defined by Lamport [7]. Causal order delivery can be defined as follows: if the send event of message M_1 "happened before" the send event of message M_2 and messages M_1 and M_2 arrive at the same destination process, then M_1 must be delivered before M_2.

This cause and effect relationship between messages can be preserved by maintaining a data structure at each process to keep track of the latest messages sent by that process to all other processes and by attaching this information to each message transmitted [8]. However, a simple approach may incur a great amount of communication overhead, which is in general of the order of N^2 where N is the number of processes. By exploiting the topology of the underlying communication network, it is possible to simplify the implementation of causal order delivery [9], thus reducing communication overhead. However, such algorithms are applicable only to a small range of applications.

The CO delivery algorithm used in this project is the Modified Schiper-Eggli-Sandoz (MSES) algorithm, which is described and proved in [4]. The MSES protocol is based on the previous SES (Schiper-Eggli-Sandoz) protocol described in [8]. The SES protocol sends each message M with a matrix of size N^2 (where N is the number of processes), known as the causal vector CV_M. The destination process uses the associated matrix to determine when it is safe to deliver the message. A major limitation of the SES algorithm is that it only works in a unicast environment. Two main modifications to SES are made in MSES. The first one is to extend it to allow multicast. The second one is to reduce control information [4].

The approach to reduce the control information is based on the direct dependency tracking technique. It is obvious that not all causal predecessors of an event need be known to ensure causal order delivery of a message *M*. Only the immediate predecessors of *M*, the messages on which the delivery of *M* is directly dependent are necessary. The delivery of the other causal predecessors of *M* can be inferred by the delivery of the immediate predecessors of *M*. Hence the size of the causal vector CV_M can be reduced dramatically (in [4] the causal vector CV_M is referred to as a causal barrier CB_M in order to compare the algorithm with that described in [10]). 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.

3. A Middleware Approach to CO Delivery

3.1 Overall System Structure

Every federate interacts with the RTI through the interface composed of the *RTIambassador* and *FederateAmbassador*. Before the federate can invoke any RTI services, it must first create an instance of the *RTIambassador*. It is natural to put the middleware between the federate and the *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 *RTIambassador* and updates the federate's causality time management information kept in the middleware.

Conversely, when the RTI has any event message destined for a certain federate, the RTI will invoke a call-back function on the *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 *FederateAmbassador*. When it is safe to deliver the message, the middleware also updates the federate's causality time management information accordingly. Figure 3.1 illustrates the overall system structure.

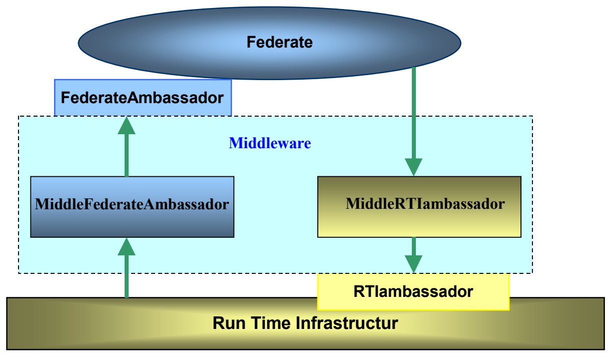

Figure 3.1 System Structure with Middleware

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 *FederateAmbassado*r. 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 federate 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 casual vector (causal barrier).
- *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 3.2 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* ($1 \le i \le N$) to the local federate and to collect and provide knowledge of remote federates such as joining the federation, publishing, subscribing, resigning, etc. The management of the federate information is described in Section 3.3.

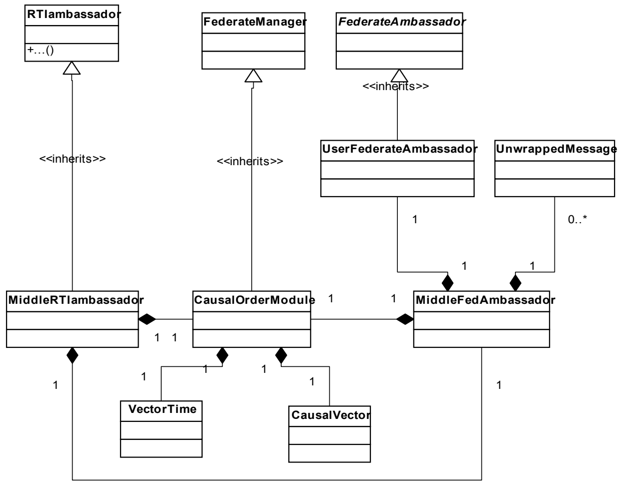

Figure 3.2 Class Diagram of Causality Middleware

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 *vector*. 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

Informally, the CO delivery algorithm can be described as follows (a formal description is given in [4]). When a process P_i ($1 \le i \le N$) sends a message *M,* it must execute the following steps:

1. Increment P_i*'s* vector time VT_{Pi}.

2. Send message $(M, VT_{Pi}, i, Dest(M), CV_{Pi})$ *to* all the processes that are included in the destination set *Dest(M)*, where CV_{Pi} is the causal vector of process P_i.

3. Update P_i*'s* causal vector CV_{Pi} to indicate that *M* is a direct dependent predecessor of any future message *M'* which is destined to a process in *Dest(M)*.

Receiving a message in this algorithm can be divided into two parts: a causality check and the delivery procedure. When message $(M, VT_M, j, Dest(M), CV_M)$ from P_j arrives at process P_i the algorithm must first check whether the message can be delivered by comparing $CV_M[i]$ with the vector time VT_{Pi} of process P_i. If it is safe to deliver the message, the following steps are executed, otherwise the message must be buffered until it can be guaranteed that there will be no causality violation:

1. Update P_i*'s* causal vector CV_{Pi} using the causal information VT_M and CV_M in message *M*. There are three substeps that perform different updating actions: (i) update CV_{Pi} for each process P_k in *Dest(M)*, (ii) update CV_{Pi} for *M's* source process P_j, and (iii) update CV_{Pi} for each process P_k that is neither a destination of *M* nor the source process of *M*.

2. Update P_i*'s* vector time VT_{Pi} by incrementing the value for the local process and then updating for remote processes using the information in VT_M.

The CO algorithm is implemented in middleware using the classes described in Section 3.1. There are three steps in the pseudocode for sending a message. 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. Increment P_i*'s* vector time VT_{Pi}

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

2. Send message *(M,* VT_{Pi}*, i, Dest(M),* CV_{Pi}*)*

   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 *RTIambassador.*

3. Update P_i*'s* causal vector CV_{Pi}

   The *MiddleRTIambassador* 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 *RTIambassador* 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 5.5 gives an example of a string encoded meta tag. This is for a message sent from process 0 to destination processes 1 and 2. (For details of the format of the vector time and causal vector, see [4]). 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.

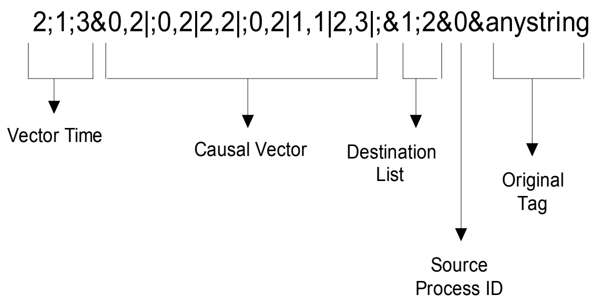

Figure 3.3 An Example of a Meta Tag.

In the above pseudocode for receiving a message, there are two parts. One is for checking to see whether the message can be safely delivered. The other is for updating local causality information. Both these parts are implemented by means of redefined methods in the *MiddleFederateAmbassador.* When the RTI invokes *receiveInteraction*() or *reflectAttributeValues*(), these redefined call-back methods will be invoked instead of those in the user's *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 *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 *UnwrappedMessage* in the *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 *receiveInteraction*() or *reflectAttributeValues*() in the *FederateAmbassador* of the user's application. Then the causality information in the message is used to update the local causality information held in middleware. For this, the *MiddleFederateAmbassador* invokes a method in the *CausalOrderModule* to execute the two steps as follows:

1. Update P_i*'s* causal vector CV_{Pi}

   The *CausalOrderModule* invokes methods in *CausalVector* to perform each of the three substeps described above.

2. Update P_i*'s* vector time VT_{Pi}

   The *CausalOrderModule* invokes methods in *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 *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 [11] is a universal model which identifies federations, objects and interactions used in a federation. However, additional 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. 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 resigns 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. Experimental Results

The benchmarking of the middleware version of the causal order time management mechanism was conducted in a similar way to that for the embedded version [4]. Three simulation scenarios were used to benchmark the performance. Each of these scenarios is designed with a different degree of dependency between federates. The first scenario is a loosely coupled federation, where the interactions of different federates are independent. The second scenario is a closely coupled federation, where the interactions of different federates have a "cause and effect" relationship. The third scenario depicts a mixed federation, which comprises a combination of independent and causally related interactions. A network of up to seven PCs connected by fast Ethernet running RTI 1.3NG-v5 was used for the benchmark experiments.

Each of the three scenarios is a step-based simulation in which all federates have the same step size. The federates are put into a sleep state for 10 milliseconds in each step to model the CPU time used by local simulation tasks in an actual application model, so as to make the simulation more realistic. For the TSO mechanism, a time advance request is made at the end of each step with an increment of 1.0. Also, for this mechanism, the lookahead time is 0.1, and the time stamp of each interaction is set to the current time plus 1.0 so that messages sent in one step are received in the next step. In all three scenarios, federates communicate using reliable TCP/IP connections.

4.1 Comparison of Time Management Mechanisms

First the performance of the RO, CO and TSO mechanisms were compared for the three different scenarios. For each scenario, experiments were carried out to measure the execution time of federates over 5000 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.

Figure 4.1 shows the results for the loosely coupled scenario. In each step, a federate broadcasts a message to all other federates, but the activities of federates are independent of each other and do not depend on the messages received. Thus, the temporal relationship between two messages received from different federates is not important. 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.

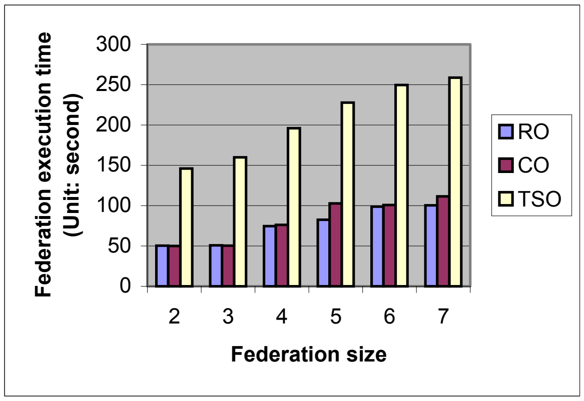

Figure 4.1 Execution Time of Federates in the Loosely Coupled Federation

Figure 4.2 shows the results for the closely coupled scenario. Each message sent out by a federate is dependent on the previous messages received from all other federates. The RO and CO mechanisms therefore have additional code in the application to provide this causal relation. Although the TSO mechanism satisfies this 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.

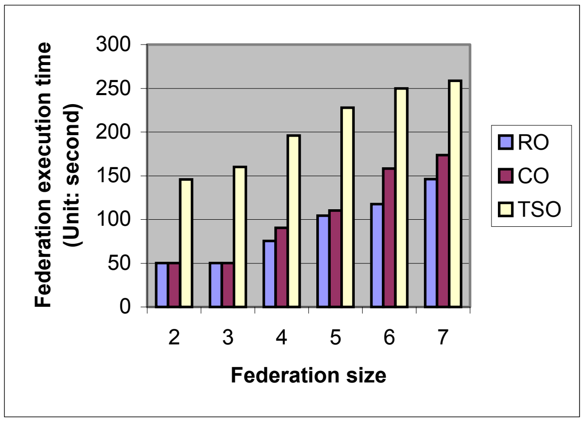

Figure 4.2 Execution Time of Federates in the Closely Coupled Federation

Figure 4.3 shows that similar results are obtained for the mixed scenario. This represents an intermediate and more realistic situation. There are exactly three federates in this scenario: an aircraft, a tank and an observer. In each step, the aircraft sends an update message to the other two federates. After a larger interval, it also fires at the tank federate. When the tank receives a fire message, it sends a hit message to the other two federates. The observer receives messages from the aircraft and tank, but does not send messages itself. The interval between fire messages was varied from 10 to 100 steps.

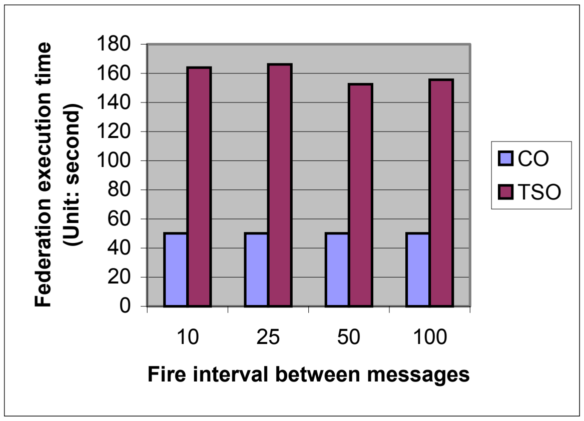

Figure 4.3 Execution Time of Federates in the Mixed Federation

As the hit message is causally dependent on the fire message, the CO mechanism ensures that the observer sees the aircraft fire before the tank is hit. Thus the CO mechanism satisfies the ordering requirements of this scenario. Similarly, the TSO mechanism is also applicable in this scenario, since it delivers messages in time stamp order and the time stamp of the fire message will be less than that of the hit message. However, the RO mechanism cannot be used for this scenario since it cannot guarantee the cause and effect relationship between the fire and hit messages.

4.2 Comparison with Embedded Approach

The comparison of the different time management mechanisms confirmed the results obtained from our previous version of the CO delivery algorithm implemented by modifying the FDK implementation of the RTI. However, to compare the performance of the middleware approach to CO delivery with that of the embedded implementation is more difficult. Although the same benchmarking programs were used, the experiments were carried out on different machines using different RTI implementations and operating systems (Sun Solaris in the case of FDK, Windows NT in the case of RTI-NG). It is therefore not appropriate to make a direct comparison. Instead the execution time for the CO mechanism is normalized against the execution time for the RO mechanism. Figures 4.4 and 4.5 show the normalized execution time for the CO delivery algorithm using both approaches. Results are given for the loosely coupled and closely coupled scenarios respectively (as RO cannot be used for the mixed scenario).

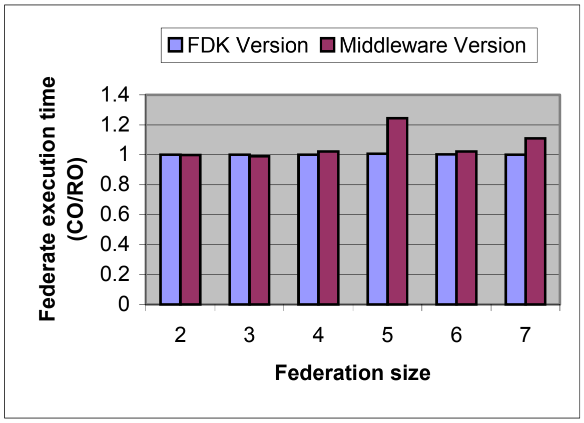

Figure 4.4 Comparison of Approaches in Loosely Coupled Federation

It can be seen that in some cases the overhead of the middleware version is slightly larger than that of the embedded version. This is mainly due to the time taken in encoding and decoding the causality information that is appended to the event messages. In the middleware approach, there is also the overhead of federation information communication. When federates join the federation, they need to obtain information from the MOM. During execution, further information may be exchanged between federates. In the embedded version, such federation information is managed within the FDK itself. However, despite its slightly larger overhead, the significant advantages of portability, maintainability and transparency offered by the middleware version make it a more attractive approach.

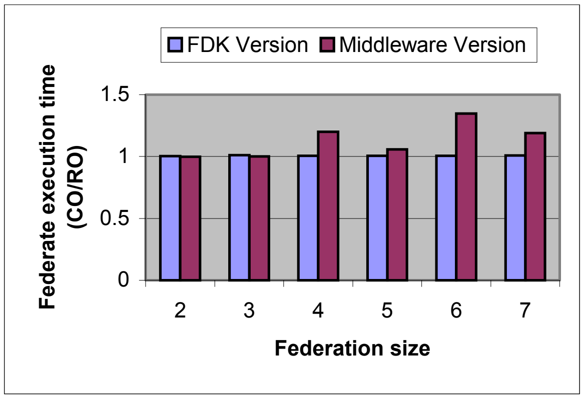

Figure 4.5 Comparison of Approaches in Closely Coupled Federation

5. Conclusions

This paper has described the implementation of a time management mechanism based on causal order delivery using a middleware approach. Experiments have been carried out to benchmark the performance of this mechanism against that of the two existing time management mechanisms provided in the HLA RTI. The results show that the CO mechanism has only a small overhead compared with the RO mechanism in various simulation scenarios with different degrees of dependency. The CO mechanism significantly outperforms the TSO mechanism regardless of scenario and federation size. The results also show that the performance of our middleware approach to CO delivery is almost as good as that of an embedded approach.

One area of future work is to investigate better methods of encoding and decoding the causality information in order to reduce the overheads further. It may also be possible to improve the federation information maintenance mechanism by better utilizing the services provided by the MOM and RTI. Currently the subscription table is built on an object rather than an attribute basis. This needs to be modified to calculate the destination sets correctly when federates publish or subscribe only some of the attributes of an object. The calculation of destination sets also need to be modified for federates that use Data Distribution Management (DDM) services.

6. References

[1] J. Dahmann, F. Kuhl, and R. Weatherly: "Standards for Simulation: As Simple As Possible But Not Simpler - The High Level Architecture for Simulation", Simulation, Vol. 71:6, pp. 378-387, 1998.

[2] R.M. Fujimoto: "Time Management in the High Level Architecture", Simulation Vol. 71:6, pp. 388-400, 1998.

[3] R.M. Fujimoto and R.M. Weatherly: "Time Management in the DoD High Level Architecture", Proc. 10th Workshop on Parallel and Distributed Simulation (PADS 96), pp. 60-67, 1996.

[4] W. Cai, B.S. Lee and J. Zhou: "Causal Order Delivery in a Multicast Environment: An Improved Algorithm", Journal of Parallel and Distributed Computing, Vol. 62, pp. 111-131, 2002.

[5] B-S Lee, W. Cai and J. Zhou: "A Causality Based Time Management Mechanism for Federated Simulation", Proc. 15th Workshop on Parallel and Distributed Simulation (PADS 2001), pp. 83-90, 2001.

[6] R.M. Fujimoto: Federated Simulations Development Kit (FDK), College of Computing, Georgia Institute of Technology, Georgia 30332-0280 http://www.cc.gatech.edu/computing/pads/fdk.htm , 1998.

[7] L. Lamport: "Time, Clocks and the Ordering of Events in a Distributed System", Communications ACM, Vol.21:7, pp. 558-565, 1978.

[8] A. Schiper, J. Eggli, and A. Sandoz: "A New Algorithm to Implement Causal Ordering", Proc. International Workshop on Distributed Algorithms, LNCS Vol. 392, pp. 219-232, Springer-Verger, 1989.

[9] C. Sun, Y. Zhang, Y. Yang, and C. Chen, "Distributed Concurrency Control in Real-time Cooperative Editing Systems, Proc. 1996 Asian Computing Science Conference," pp. 84-95, Singapore, 1996.

[10] R. Prakash, M. Raynal and M. Singhal: "An Adaptive Causal Ordering Algorithm Suited to Mobile Computing Environments", Journal of Parallel and Distributed Computing, Vol. 41, pp. 190-204, 1997.

[11] D. Fullford and D. Wetzel: "A Federation Management Tool: Using the Management Object

Model (MOM) to Manage, Control, and Monitor a Federation", Proc. 1999 Spring Simulation Interoperability Workshop, 99S-SIW-196, 1999.

Author Biographies

STEPHEN JOHN TURNER joined Nanyang Technological University (Singapore) in 1999 and is currently an Associate Professor in the School of Computer Engineering and Director of the Parallel and Distributed Computing Centre. Previously, he was a Senior Lecturer in Computer Science at Exeter University (UK). He received his MA in Mathematics and Computer Science from Cambridge University (UK) and his MSc and PhD in Computer Science from Manchester University (UK). His current research interests include: parallel and distributed simulation, distributed virtual environments, grid computing and multi-agent systems.

WENTONG CAI is currently an Associate Professor at School of Computer Engineering (SCE), Nanyang Technological University (Singapore). He received his B.Sc. in Computer Science from Nankai University (P. R. China) and Ph.D. also in Computer Science from University of Exeter (U.K.). He was a Post-doctoral Research Fellow at Queen's University (Canada) from Feb 1991 to Jan 1993, and joined SCE as a lecturer in Feb 1993. Dr. Cai has served as a program committee member in many international conferences (e.g., PADS, DSRT and PDCS). He is a member of IEEE and his current research interests are mainly in the areas of parallel and distributed computing (particularly, Parallel and Distributed Simulation and Grid Computing).

JI CHEN received his B.Eng. (Honours) in Computer Engineering from Nanyang Technological University (Singapore) in 2002. He is currently a systems analyst in PSA Corporation Limited (Singpore).