IMPLEMENTATION ISSUES FOR SHARED STATE IN HLA-BASED DISTRIBUTED SIMULATION

Malcolm Yoke Hean Low
Boon Ping Gan

Singapore Institute of
Manufacturing Technology
71 Nanyang Drive, Singapore 638075

Junhu Wei, Xiaoguang Wang,
Stephen John Turner, Wentong Cai

School of Computer Engineering
Nanyang Technological University
Nanyang Avenue, Singapore 639798

KEYWORDS

Shared State, Distributed Simulation, High Level Architecture, Zero Lookahead.

ABSTRACT

The problem of shared state is well known to the parallel and distributed simulation research community. In this paper, we revisit the problem of shared state in the context of a High Level Architecture based distributed simulation. A middleware approach is proposed to solve this problem within the framework of the High Level Architecture Runtime Infrastructure. Four solutions to this problem are implemented in the middleware using receive-order messages. We will discuss the implementation issues of these four solutions in the middleware. Experimental results comparing the performance of these four solutions against a simple request-reply approach using time-stamp-order messages are also presented.

1 INTRODUCTION

Simulation has traditionally been used as a tool to perform "what if" analysis in complex systems such as the operations of a manufacturing production facility. In a supply chain scenario with multiple business partners each with its own manufacturing production facility, it is not sufficient for each business partner in the supply chain to optimize its operation using simulation. In order to optimize the performance of the whole supply chain, these simulation models need to be integrated before any meaningful analysis can be made. Physically integrating these simulation models into a single simulation environment is often not possible due to 1) the complexity of individual models; 2) business partners being geographically dispersed; and 3) confidentiality in sharing certain parts of the simulation model. Distributed simulation offers a solution to this problem by allowing existing simulation models to be reused and integrated with other simulation models in the supply chain through well-defined interfaces. Each supply chain simulation can also selectively expose only the necessary data to other partners in the supply chain simulation.

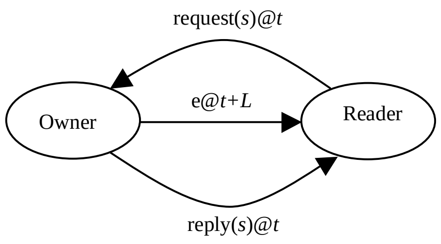

Figure 1: Request-Reply using TSO Messages

An emerging standard for distributed simulation, namely the High Level Architecture (HLA) standard has been proposed by the U.S. Department of Defense (DoD) (Kuhl et al. 1999). The HLA defines the rules and specifications to support reusability and interoperability of different simulators. In HLA terminology, a single simulator is referred to as a federate. A federation is then a set of federates working together to achieve a given goal. Each federate defines the objects and interactions that are shared in its simulation object model (SOM) and interacts with one another over the Runtime Infrastructure (RTI) (DMSO 2002).

The problem of shared state is a well-known problem in both the parallel, as well as the distributed simulation research community. In the context of an HLA-based distributed simulation, the issue involves the following: an "owner" federate updates a local shared variable periodically and multiple remote "reader" federates can access the value of the shared variable instantaneously (at the same simulation time as the read). For the rest of this paper, we assume that the HLA-based distributed simulation runs on an RTI that uses a conservative synchronization protocol for its time management service.

Figure 1 shows a straightforward implementation to support shared state in an HLA-based distributed simulation using a set of time-stamp-order (TSO) request-reply interactions both time-stamped at the simulation time *t*. In the example, the owner federate updates a shared variable *s* periodically, and also sends a TSO event *e* to the reader federate periodically with a lookahead of *L*. At simulation time *t*, the reader federate accesses the value of the shared variable *s* by

Proceedings 15th European Simulation Symposium
Alexander Verbraeck, Vlatka Hlupic (Eds.)
(c) SCS European Council / SCS Europe BVBA, 2003
ISBN 3-936150-28-1 (book) / 3-936150-29-X (CD)

sending a TSO request message time-stamped at *t*. The owner federate, on receiving the request message, will issue a TSO object update with simulation time *t*. We refer to this approach to support shared state on an HLA-based distributed simulation as the *PullTSO* approach.

In order for the *PullTSO* scheme to work, both the owner and the reader federates must be time-constrained and must regulate the federation with a lookahead of zero. However, having zero-lookahead in a federation is often detrimental to the performance of the simulation system whether the underlying RTI is using conservative or optimistic time synchronization. Suppose the request-reply TSO messages in Figure 1 are replaced by receive-order (RO) messages. Since RO messages are not used by the RTI to check lookahead and the time advancement constraint on each federate, the owner federate can regulate the federation with a lookahead of *L* instead.

In this paper, we address the issue of shared state in an HLA-based distributed simulation by replacing the TSO request-reply messages in the *PullTSO* approach with RO messages. We propose a middleware approach to support shared state. We will outline four solutions to support shared state within the middleware. While the use of the middleware approach is to hide the implementation of the support for shared state from the users, it must also preserve the semantics of the different RTI APIs that are used by the users. We will explain how this can be achieved in the middleware using an example. The performance of the solutions proposed will also be compared against that of the *PullTSO* approach.

The rest of the paper is organized as follows. Section 2 describes some related work in solving the shared state problem. Our solutions to shared state using the middleware approach will be described in Section 3. In Section 4, we discuss some issues in implementing the solutions in the middleware. Experimental results comparing our solutions to the *PullTSO* approach will be presented in Section 5. We conclude this paper in Section 6 and outline further work in this area.

2 RELATED WORK

The issue of shared state has been explored in the context of several parallel simulation research projects. For example, in the work by Mehl and Hammes (Mehl and Hammes 1993), they proposed two general approaches to implement shared variables using a conservative synchronization algorithm, namely 1) request-reply and 2) cached-copy. In the request-reply approach, the owner keeps a history list of the shared variable. When a reader requests the shared variable at simulation time *t*, the owner will wait until it is certain that no other write messages will be received with simulation time smaller than *t* before it retrieves the value of the shared variable at time *t* from its history list and forwards the reply to the reader.

The request-reply approach frees the owner from the time constraints from its readers and allows the owner to proceed ahead of its readers whenever possible. However, a reader has to always suspend itself whenever it needs to access the value of the shared variable from the owner. The cached-copy approach proposed by Mehl and Hammes solves this problem by having each reader keep a cached-copy of the shared variable. The cached-copy of the shared variable has a time-guarantee associated with it. Whenever a reader needs to access the shared variable, it first checks the validity of its cached-copy of the shared variable. It will send a request to the owner if such copy is not found or the copy is invalid.

Lim et al. (Lim et al. 1998) showed that if the request/write/reply messages are sent using time-stamp order, then the request-reply approach proposed by Mehl and Hammes could function without a history list. While Mehl and Hammes use a cache-on-demand policy to update the reader's local cache copy of the shared variable, Lim et al. explored an always-update-by-writer update policy for the cached-copy approach whereby the owner will forward each update to the shared variable to all its readers.

The issue of shared variables has also been raised during the third meeting of the HLA-CSPIF forum (HLA-CSPIF 2002). The aim of this forum is to create a standardized approach to distributed simulation using HLA to support interoperation of discrete event models created in commercial-off-the-shelf (COTS) simulation packages. The group is currently looking at specifying reference models for testing the interoperability of different simulators. In particular, one of the reference models under discussion requires the implementation of shared variables across two or more different COTS simulation packages.

A preliminary report on the work described in this paper can be found in Gan et al. (Gan et al. 2003) in which two of the solutions presented in this paper are first proposed. In this paper, we extend on our previous work and propose two more solutions to solving the shared state problem. We will also discuss implementation issues to enable support for shared state in a middleware for RTI.

3 THE SOLUTIONS

Both the request-reply and the cached-copy solutions proposed by Mehl and Hammes can be implemented easily under HLA. However, both the request and reply messages, as well as the update message for the cached-copy, will have to be sent using TSO interactions and object updates at the current simulation time. This implies that both the owner and reader federates must regulate the federation with zero lookahead.

In this section, we describe four solutions to solve the problem of shared state. We will refer to these four solutions as: 1) *PullRO*, 2) *PushRO*, 3) *PullROTG* and 4) *PushROTG*. The *PullRO* and *PushRO* solutions will be described briefly as they have previously been described in detail in Gan et al. (Gan et al. 2003). All four solutions involve replacing the TSO interactions and object updates used in the standard request-reply and cached-copy solutions with RO messages. This eliminates one source of zero lookaheads in the federation and allows both the owner federate as well as the reader federates to regulate the federation with the next smallest (possibly non-zero) lookahead. Similar to the work carried out by Lim et al. (Lim et al 1998), our solutions currently assume that the owner is the only writer for the shared variable. We plan to eliminate this assumption in the next stage of our work.

3.1 Solution 1: *PullRO*

In the *PullRO* approach, whenever a reader needs the latest value of a shared variable, it sends an RO interaction to the owner to request for the value at a specific request time. The owner, on the other hand, maintains a history list of all the updated values of the shared variable with their associated update times. Suppose the owner federate is at simulation time t_1 and it receives a request from a reader with request time t_2. If $t_1 \geq t_2$, the owner federate searches its history list for two consecutive entries with update times t_j and t_k such that $t_j \leq t_2 < t_k$. An RO object update with the value of the shared variable at simulation time t_j will be sent back to the reader. If $t_1 < t_2$, the request will be buffered and serviced when the owner's simulation time reaches t_2.

3.2 Solution 2: *PushRO*

In the *PushRO* solution, we use a cached-copy approach with an always-update-by-writer update policy similar to that used by Lim et al. (Lim et al. 1998). Whenever the owner updates the shared variable, it also sends the update to each of its readers. This update is associated with the timestamp of the update and is sent using RO interactions or object updates. As the owner may be sending an update with timestamp greater than the simulation time of a reader, each reader must also keep a future list to store the updates received from the owner.

In situations in which one of the readers runs ahead of the owner, the reader will not find any valid entry in its future list. In this case, the reader adopts a *PullRO* approach and sends the owner an RO interaction to request for the value of the shared variable. This request will be serviced as in the case of *PullRO* when the owner's simulation time reaches the request time.

3.3 Solution 3: *PullROTG*

In the *PullROTG* approach, we augment the *PullRO* solution with a time-guarantee feature. Each update entry in the owner's history list is now associated with a time-guarantee for the valid duration of the entry. This time-guarantee is automatically determined by our middleware and requires no input from the users. For example, if the owner updates a shared variable A at time t_1, a time-guarantee is also associated to the length of validity for the value of A. Initially, without any additional information from the user, the middleware cannot assign any effective time-guarantee to the value of A at the point of the update. Hence, the update entry for A at time t_1 is associated with a time-guarantee of t_1, effectively representing a time-guarantee of zero.

However, if the owner subsequently updates the shared variable A at time t_2, the entry of A at time t_1 in the history list can then be modified to be associated with a time-guarantee t_2. This means that the update at time t_1 is valid up to time t_2. Each reader, on the other hand, will also keep a copy of the shared variable and its associated update time and time-guarantee. Suppose a reader requests the shared variable at time t_3 ($t_1 \leq t_3 < t_2$), it will receive an update from the owner with the value of the shared variable at time t_1 and an associated time-guarantee t_2. The reader can use this local copy of the update for subsequent requests with time smaller than t_2.

If the owner receives a request with time $t > t_{curr}$, where t_{curr} is the current simulation time of the owner, the request will first be buffered. The request will be serviced when the owner's simulation time reaches t. The value of the shared variable at time t, together with the associated time-guarantee, t_g, will be forwarded to the reader. Note that in this case, the time-guarantee, t_g, for the update message is computed as follows:

$$t_g = \min(t_{NER}, t_{MNE}) \qquad (1)$$

where t_{NER} is the cutoff time specified by the owner federate when invoking any of the RTI time advancement APIs; and t_{MNE} is the time obtained by invoking the `RTI:queryMinNextEventTime()`. One assumption made here is that if the owner federate is at simulation time t and asks the RTI for time advance to time t_{NER}, it will not update its shared variables between t and t_{NER}.

3.4 Solution 4: *PushROTG*

Similarly, we also augment the *PushRO* solution with the time-guarantee feature. In the *PushROTG* approach, each update entry in the future list maintained by a reader is also associated with a time-guarantee. Note that each of these updates received from the owner initially carries no effective time-guarantee, i.e. the time-guarantee is the same as the

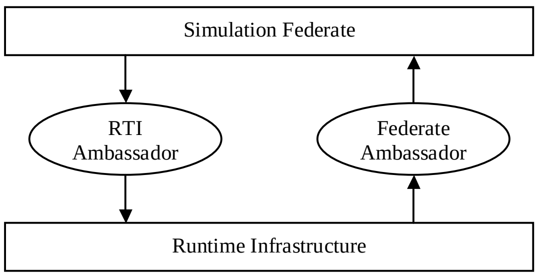

Figure 2: Architecture of RTI

update time. However, whenever a new update is received from the owner, the time-guarantee for the previous entry in the future list is associated with the update time of the newly received update. We note that this form of time-guarantee offers no distinct advantage over the *PushRO* approach whenever the reader runs behind the owner in simulation time since the search for a match between two consecutive entries in the future list already uses the concept of time-guarantee implicitly.

However, whenever a reader runs ahead of the owner, it adopts the *PullRO* approach and sends a request to the owner. The update message it receives from the owner has a time-guarantee computed using equation 1. This update message will be placed at the top of the future list. The time-guarantee provided by this update can further reduce the number of request messages being forwarded to the owner.

4 IMPLEMENTATION ISSUES

In this section, we discuss the implementation issues of providing shared state support using the four solutions in a middleware for HLA-based distributed simulation. We will first describe the RTI+ middleware that supports shared state in Section 4.1. The implementation of the four solutions in RTI+ will be described in Section 4.2. Section 4.3 describes how some of the methods in RTI+ should be implemented in order to preserve the semantics of the HLA-RTI APIs. Section 4.4 discusses the implementation issues for fossil collection and late arriving federates.

4.1 Middleware to Support Shared State

Figure 2 shows how a simulation federate is typically integrated with the HLA-RTI. The simulation federate can send interactions or object updates to other federates through the RTI using the RTI ambassador. Conversely, the RTI delivers interactions or object updates to the simulation federate via the federate ambassador (implemented as callback functions by the simulation federate).

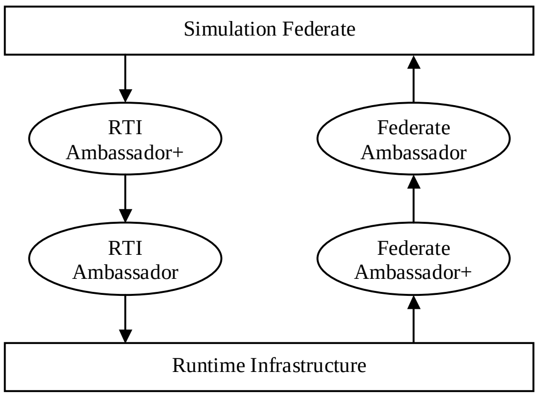

Figure 3: RTI+ Middleware

We extended the HLA-RTI architecture with a middleware layer, which we refer to as the RTI+. Figure 3 shows the extended HLA-RTI architecture. In the extended architecture, all outgoing interactions from the simulation federate to the RTI are routed through the RTI ambassador+. All incoming interactions from the RTI to the simulation federate are routed through the federate ambassador+. Both the RTI ambassador+ and the federate ambassador+ support the full set of interfaces in RTI ambassador and federate ambassador respectively. All four solutions described in the previous section, together with the *PullTSO* solution, are implemented in the RTI+.

For the rest of this paper, we will use the format RTI:xxx and FebAmb:xxx to denote methods in the original RTI library; and RTI+:xxx and FebAmb+:xxx to denote methods in the RTI+ middleware. Some of the interfaces are extended in the RTI+ library in order to support shared state. For example, the RTI+:registerObjectInstance method is extended to create a history list for objects at an owner federate, and a future list for objects at a reader federate.

Two additional methods are also provided in the RTI ambassador+ to allow a simulation federate to request for an object or class update at a specific simulation time. Figure 4 shows the APIs of these two methods.

```
void requestObjectAttributeValueUpdate(ObjectHandle
                        theObject, RTI::FedTime theTime)

void requestClassAttributeValueUpdate(ClassHandle
                        theClass, RTI::FedTime theTime)
```

Figure 4: Shared State APIs for RTI+

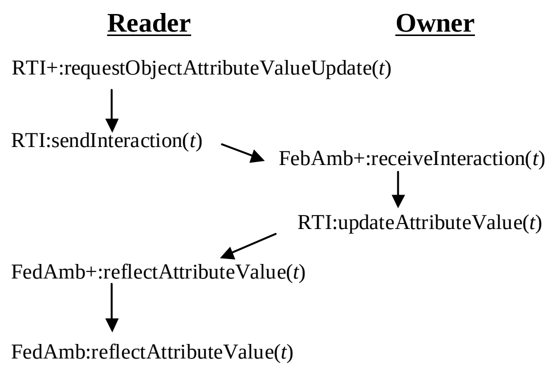
Reader
Owner
RTI+:requestObjectAttributeValueUpdate(t)
RTI:sendInteraction(t)
FebAmb+:receiveInteraction(t)
RTI:updateAttributeValue(t)
FedAmb+:reflectAttributeValue(t)
FedAmb:reflectAttributeValue(t)

Figure 5: The *PullTSO* Approach

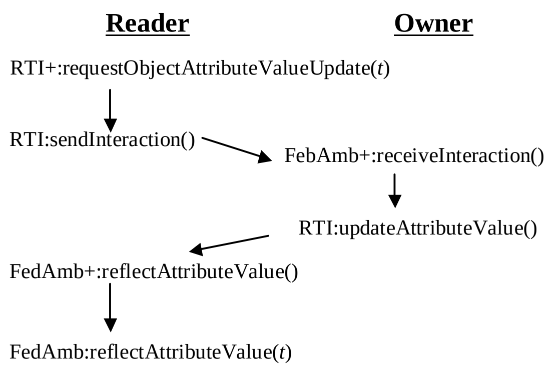
Reader
Owner
RTI+:requestObjectAttributeValueUpdate(t)
RTI:sendInteraction()
FebAmb+:receiveInteraction()
RTI:updateAttributeValue()
FedAmb+:reflectAttributeValue()
FedAmb:reflectAttributeValue(t)

Figure 6: The *PullRO* Approach

Using the *PullTSO* approach, both methods will be translated to a TSO interaction that is sent from the reader to the owner. This interaction contains the object/class handle that the reader is requesting, and the simulation time at which the value is needed. Figure 5 illustrates this mechanism. A call to the RTI+:requestObjectAttributeValueUpdate method in the middleware is translated to a TSO RTI:sendInteraction call in the original RTI ambassador at the reader federate. A FebAmb+:receiveInteraction callback is triggered in the middleware's federate ambassador at the owner side. The middleware at the owner federate processes the request by replying through a call to RTI:updateAttributeValue. Note that the FebAmb+:receiveInteraction is not allowed to call the RTI:updateAttributeValue within the federate ambassador+. Hence, the middleware needs to first record the request, and processes it once control is returned to the RTI ambassador+. Control is transferred to the middleware when the user invokes the RTI+:tick method. The RTI+:tick method needs to perform two tasks: 1) process all pending requests from external readers in the method RTI+:processSysInteraction; and 2) yield control to the RTI by calling the original RTI:tick method.

4.2 Implementation using RO Messages

In this section, we discuss the implementation using RO messages to deliver the request-reply messages between owners and readers of shared variables. While the details of the implementation are illustrated using the *PullRO* solution, the same principle applies to the other three solutions.

Figure 6 illustrates the sequence of method calls to implement the *PullRO* approach. Note that the entry and exit points to the sequence of method calls are the same as those in Figure 5. This means that the underlying solutions used to support shared state in the middleware is transparent to the user. The middleware at the reader federate translates the TSO request call to an RO RTI:sendInteraction. The owner federate in turn replies to the reader using an RO RTI:updateAttributeValue.

At the reader end, once a request is issued, the reader is not allowed to progress in time, until it receives the update that corresponds to its request. This is realized by keeping track of pending requests, and withholding the time advance request (a primitive in RTI that federates use to try to advance their time) until the request is received.

4.3 Semantics of RTI APIs

The semantics of RTI APIs has to be preserved when implementing support for shared state in the RTI+ middleware no matter which solutions the user chooses to use. A naive implementation may alter the semantics of the original RTI APIs. We illustrate this issue using the implementation of RTI+:tick. As mentioned in the previous section, the RTI+:tick method needs to perform two specific tasks, RTI+:processSysInteraction and call RTI:tick. However, the order in which these two tasks are carried out will potentially violate the specification of the RTI APIs.

Suppose the tick method is implemented such that the RTI+:processSysInteraction task is executed first, followed by the call to the original RTI:tick method. Consider the example in Figure 7 showing a sequence of method calls from an owner federate using the *PullTSO* approach. The owner federate first requests for time advance to simulation time 4. The RTI+ middleware will forward the request to the RTI. The owner federate subsequently yields control to the RTI+ middleware by calling the RTI+:tick method. The RTI+:processSysInteraction subtask in the RTI+:tick method does nothing since there is no pending request. However, when the original RTI:tick method is called, a TSO interaction carrying a request from a reader to access a shared variable at simulation time 3 is delivered. Note that this request will not be processed until the next call to RTI+:processSysInteraction. The owner federate is subsequently granted a time advance to simulation time 4.

Suppose the owner next issues a time advance request to advance to simulation time 8, and proceeds to yield control to the RTI+ middleware by calling the RTI+:tick

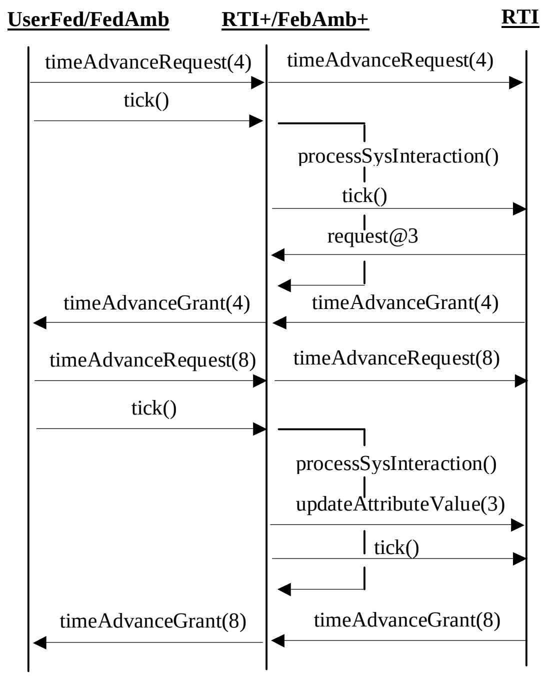

Figure 7: Semantics of RTI APIs

method. In the RTI+:processSysInteraction method, the request from the reader at simulation time 3 is processed. This results in a TSO object update event time-stamped at 3 being sent out to the reader.

However, sending out the TSO event time-stamped at 3 will immediately cause the RTI to generate an "*InvalidFedTime*" exception. The reason for this is that after the middleware invokes the method RTI:timeAdvanceRequest with request time 8, it "may not generate TSO events whose time stamps are less than the requested time plus the federate's lookahead" (DMSO 2002). Thus, the request at simulation time 3 should not have been sent out whilst the RTI:timeAdvanceRequest with request time 8 is in progress.

The correct approach to implementing the RTI:tick method would be to reverse the order of executing the two tasks. In the example above, executing RTI:tick before RTI+:processSysInteraction would allow RTI to deliver the request at time 3 to the owner federate. This request would be processed by the RTI+:processSysInteraction that executes next. The middleware at the owner federate will generate an object update at simulation time 3 and send it to the reader before the owner federate is granted an advance to simulation time 4.

4.4 Late Arriving Federates and Fossil Collection

Other implementation issues that have to be addressed are the problem of late arriving federates and fossil collection. We will describe how these two issues are resolved in this section.

In the initial implementation of the *PushRO* and *PushROTG* solutions, an assumption is that the reader will receive all updates from the owner federate. The only situation in which a reader actively requests values of a shared variable from the owner is when the reader runs ahead of the owner and all the cached values of the shared variable in the reader's future list have time-stamp (or time-guarantee) smaller than its simulation time.

However, the RTI also allows federates to join at any time an existing running federation. Such federates are termed as late arriving federates. Suppose one of the reader federates is a late arriving federate which joins the federation at simulation time t. If the owner federate is at simulation time t_1 ($t_1 \geq t$), some of the updates sent from the owner between t and t_1 may be "lost" to the reader federate.

To handle this problem, an owner federate employing the *PushRO* or *PushROTG* scheme must also keep a history list for all its shared variables. A new reader federate which joins the federation midway will request for the values of shared variables using the same approach as in the case of *PullRO* and *PullROTG*.

The entries for each update to a shared variable are kept in a history list at an owner federate or in a future list at a reader federate. The memory used by the outdated entries has to be fossil collected periodically. While an owner federate has to keep those entries with time-stamp greater than the smallest federate time of all its readers (in all four RO solutions), a reader federate needs only to keep those entries with time-stamp greater than its own federate time. The federate times of individual reader federates can be obtained by accessing services provided by the Management Object Model (Fullford and Wetzel 1999) in order to determine the reader federate with the smallest time-stamp.

5 EXPERIMENTAL RESULTS

Experiments were carried out to compare the performance of the four solutions using RO messages against the *PullTSO* approach. The experiments were carried out using the DMSO RTI 1.3-NG version 5. The simulation model consists of two federates: an owner federate and a reader federate. The owner federate updates a shared variable every 100 time units, while the reader federate accesses the shared variable every 100r time units, where r is the request ratio. We experimented with different request ratios r = (0.1, 0.2, 0.5, 1.0, 2.0, 5.0, 10.0). We also experimented with different lookaheads between the two federates. The lookahead is modelled by sending a TSO user

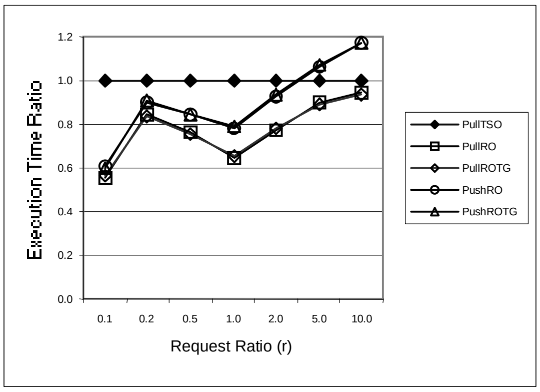

Figure 8: Execution timeio vs Request Ratio (Lookahead = 10)

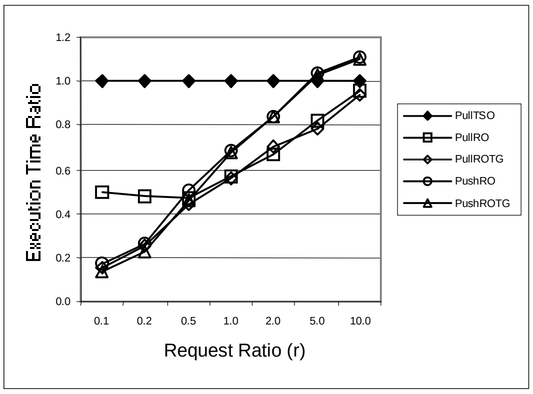

Figure 9: Execution time ratio vs Request Ratio (Lookahead = 100)

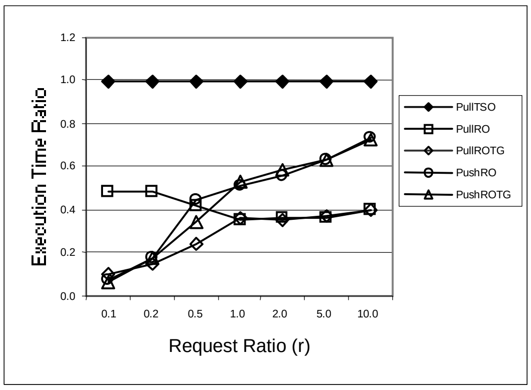

Figure 10: Execution time ratio vs Request Ratio (Lookahead = 1000)

interaction from the owner to the reader each time the owner updates the shared variable. We experimented with different values of lookaheads *l* = (0, 10, 100, 1000). If the owner updates the shared variable at simulation time *t*, then the user interaction is time-stamped with the simulation time *t+l*.

Figures 8, 9 and 10 show the execution time ratios using the four solutions for different values of lookaheads. The execution time ratio is the ratio between the execution time of the respective RO solutions against the execution time using *PullTSO*. Hence the *PullTSO* version is depicted with an

	Lookahead=100		
Request Ratio	**0.1**	**0.2**	**0.5**
PullRO	99999	49999	19999
PullROTG	19744	21041	17307
PushRO	25122	23720	19900
PushROTG	13972	13607	10698
	Lookahead=1000		
Request Ratio	**0.1**	**0.2**	**0.5**
PullRO	99999	49999	19999
PullROTG	10000	10000	10000
PushRO	34	12620	19730
PushROTG	35	10000	10000

Table 1: No. of Requests Received by the Owner Federate

execution time ratio of 1.0. Table 1 shows the number of requests received by the owner federate for request ratios 0.1, 0.2 and 0.5.

From the three graphs, we see that the execution times of the four RO solutions generally decrease compared to the *PullTSO* approach as the minimum lookahead of the system is increased from 10 to 1000. This shows that the RO solutions free the owner and reader federates from one source of zero lookahead caused by the sending of a TSO request-reply message and allows them to regulate the federation with the lookahead imposed by the user interaction sent from the owner to the reader federate.

For small request ratios, the reader federate requests the values of the shared variable more frequently than the updates by the owner federate. This results in a large number of request messages being sent from the reader to the owner if *PullRO* is used. However, this scenario favors the *PushRO* solution since updates only need to be sent out infrequently. Most of the accesses to the shared variable by the reader federate can be fulfilled by the middleware using entries in the future list without the need to request new values from the owner federate. The graphs also show that the *PullROTG* solution is also able to deliver performance comparable to that of the *PushRO*. The time-guarantee provided by the owner federate is used effectively to satisfy requests from the reader federate without the need to request updates from the owner federate.

Table 1 confirms that the number of requests received by the owner federate is significantly fewer using the *PushRO* version compared to the *PullRO* version for small request ratios. The number of requests using the *PullROTG* solution is also comparable to that of the *PushRO* version for lookahead=1000. The additional time-guarantee provided by the *PushROTG* solution further reduces the number of requests received by the owner federate.

For large request ratios, the performance of both *PullRO* and *PullROTG* solutions are consistently better than the *PushRO* and *PushROTG* solutions. In fact, for the runs with lookaheads = 10 and 100 and request ratio $r \geq 5.0$, both *PushRO* and *PushROTG* yield worse performance compared to the *PullTSO* solution. In this case, the owner updates the value of the shared variable more often than the reader accesses it. Thus, most of the updates from the owner received by the reader in the *PushRO* or *PushROTG* versions are redundant.

6 CONCLUSION

In this paper, a middleware approach to support shared state in an HLA-based distributed simulation has been described. Detailed discussions for some issues in implementing the four RO solutions in the middleware are also presented. Our experiments show that the four RO solutions proposed and implemented in the RTI+ middleware are indeed much more efficient compared to the *PullTSO* approach. The experimental results also show that the additional time-guarantee provided by the *PullROTG* solution allows further performance improvement compared to the *PullRO* solution when the request ratio is small.

This middleware approach for shared state will also be tested on more realistic simulation models to better evaluate the effectiveness of the four solutions. Further work will also be carried out to explore the possibility of applying similar techniques to address the issue of remote write.

REFERENCES

DMSO. 2002. RTI 1.3-Next Generation Programmer's Guide Version 5, DoD, DMSO, Feb 2002.

Fullford D. and D. Wetzel. 1999. "A Federation Management Tool: Using the Management Object Model (MOM) to Manage, Control, and Monitor a Federation". In *Proceedings of the 1999 Spring Simulation Interoperability Workshop*, 99S-SIW-196.

Gan B.P.; M.Y.H. Low; J.-H. Wei; X.-G. Wang; S.J. Turner and W.-T. Cai. 2003. "Synchronization and Management of Shared State in HLA-Based Distributed Simulation". To appear in the 2003 Winter Simulation Conference.

HLA-CSPIF 2002. 3rd Meeting of HLA-CSPIF Forum, 20-21 November 2002, Savill Court Hotel, London. `http://www.cspif.com`.

Kuhl F.; R. Weatherly and J. Dahmann. 1999. "Creating computer simulation systems: An introduction to the High Level Architecture". Prentice Hall PTR.

Lim C.C.; Y.H. Low; B.P. Gan and S. Jain. 1998. "Implementation of Dispatch Rules in Parallel Manufacturing Simulation". In *Proceedings of the 1998 Winter Simulation Conference*, 1591-1597.

Mehl H. and S. Hammes. 1993. "Shared Variables in Distributed Simulation". In *Proceedings of 7th Workshop on Parallel and Distributed Simulation*, 68-75.

AUTHOR BIOGRAPHIES

MALCOLM YOKE HEAN LOW is a Research Engineer with the Production and Logistics Planning Group at the Singapore Institute of Manufacturing Technology. He received his Bachelor and Master of Applied Science in Computer Engineering from Nanyang Technological University, Singapore in 1996 and 1997 respectively, and a D.Phil. in Computer Science from Oxford University in 2002. His research interests are in the areas of adaptive tuning and load-balancing for parallel and distributed simulation systems, and the application of multi-agent technology in supply chain logistics coordination. His email address is `yhlow@SIMTech.a-star.edu.sg`.

BOON PING GAN is a Research Engineer with the Production and Logistics Planning Group at Singapore Institute of Manufacturing Technology (formerly known as Gintic Institute of Manufacturing Technology). He is currently leading a research project that attempts to apply distributed simulation technology for supply chain simulation. He received a Bachelor of Applied Science in Computer Engineering and Master of Applied Science from Nanyang Technological University of Singapore in 1995 and 1998 respectively. His research interests are parallel and distributed simulation, parallel programs scheduling, and application of genetic algorithms. His email address is `bpgan@SIMTech.a-star.edu.sg`.

JUNHU WEI is working with Nanyang Technological University (Singapore) as a Research Fellow. He received his BE in Automatic Control and ME in System Engineering and PhD in Control Engineering from Xi'an Jiaotong University (China). His current research interests include parallel and distributed simulation, Simulation, Planning and Scheduling of Manufacturing. His email address is `asjhwei@ntu.edu.sg`.

XIAOGUANG WANG is currently a Ph.D student at School of Computer Engineering (SCE), Nanyang Technological University, Singapore. She received her B.Sc in Computer Science from Nanjing University of Aeronautics and Astronautics, China in 1997. Her research interests lie in Distributed Simulation and High Level Architecture, which is also her Ph.D topic currently being developed. Her email address is `PG02355670@ntu.edu.sg`.

STEPHEN J. 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. His email address is `assjturner@ntu.edu.sg`.

WENTONG CAI is currently an associate professor and Head of Software System Division 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 is a member of IEEE and his current research interests are mainly in the areas of parallel and distributed computing, particularly, Parallel & Distributed Simulation and Grid Computing. His email address is `aswtcai@ntu.edu.sg`.