Proceedings of the 2003 Winter Simulation Conference
S. Chick, P. J. Sánchez, D. Ferrin, and D. J. Morrice, eds.

SYNCHRONIZATION AND MANAGEMENT OF SHARED STATE IN HLA-BASED DISTRIBUTED SIMULATION

Boon Ping Gan
Malcolm Yoke Hean Low

71 Nanyang Drive
Singapore Institute of
Manufacturing Technology
Singapore 638075, SINGAPORE

Junhu Wei
Xiaoguang Wang
Stephen John Turner
Wentong Cai

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

ABSTRACT

The HLA Runtime Infrastructure can support a conservative simulation protocol for its time management service. However, the performance of conservative simulation protocols is very much dependent on lookahead that one can extract out of a simulation model. Also the most conservative value has to be taken in order to ensure the causality constraint. In this paper, we propose two algorithms, namely pullRO and pushRO, that allow one to replace some of the timestamp order (TSO) messages (possibly those causing zero lookahead values) with receive order (RO) messages. This removes the time constraint that these messages impose on the lower bound timestamp (LBTS) calculation, which in turn will improve the time advancement rate of federates. The algorithms still ensure the causality constraint and a middleware approach is used to preserve the semantics of the RTI APIs. The performance of the two algorithms is compared against a baseline model where no TSO messages are replaced.

1 INTRODUCTION

Distributed simulation is an emerging technology for collaborative simulation. It enables models to be run at geographically dispersed sites. Each participant constructs their own model, and agrees upon the messages that are going to be exchanged among the models. Sensitive data about a model is encapsulated within the model itself. This is particularly useful in an application in which the sharing of sensitive data to build a single centralized model is not an option. One example of such an application is supply chain simulation (Gan et al. 2000). To build a meaningful centralized supply chain model, companies need to share a tremendous amount of data, including sensitive data, such as a dispatching rule at the machine level. With distributed simulation, sharing of sensitive data is not necessary as each company builds their own model, and the data is well encapsulated within the company.

The High Level Architecture (HLA) standard (Kuhl, Weatherly, and Dahmann 1999) is the *de-facto* standard for distributed simulation. It facilitates the interoperability and reusability (through the object model template, Lutz 1998) of all types of models and simulations. The standard provides a common technical framework for the integration of simulation models. It comprises three components: the HLA interface specification, federation rules, and the object model template (OMT). The interface specification, implemented by the Run-Time Infrastructure (RTI), defines how federates interact with the federation, and with one another. In HLA terms, a federate is a simulation model while a federation is a collection of federates that form the entire simulation. Each federate defines the objects and interactions that are shared in its simulation object model (SOM). The responsibilities of federates and their relationship with the RTI are described by the federation rules.

In this paper, we focus on extending the implementation of the RTI to relax the time synchronization among federates, particularly focusing on RTIs that support the conservative simulation protocol for their time management service, for example the DMSO's RTI (*www.dmso.mil*). Each federate regulates the time of the federation with a lookahead value. This lookahead determines the next earliest time that the federate will potentially send out an interaction or an object update. Hence, federates that are constrained cannot progress beyond the minimum of the current times of regulating federates plus the corresponding federate lookahead, which is referred to as the lower bound timestamp (LBTS). This ensures that a constrained federate will not receive any updates/messages in its past. How asynchronously federates can progress in

time is thus very much dependent on the lookahead value. A relatively larger lookahead value means less constraint on the federation.

In general, lookahead is chosen by considering all the timestamp-order (TSO) interactions and object updates within a model (TSO interaction/object updates are delivered in time order to the receiving federates). The smallest time increment required by a TSO interaction/object update is chosen as the federate lookahead. In some simulation models, this could mean zero lookahead if there exists an interaction/object update that is scheduled at the current federate time, such as objects that define the state of a federate that is shared across federates. For example, in a supply chain comprising a raw material supplier (RMS), a manufacturer (MANUF), a distribution center (DC), and an order dispatching center (ODC), inventory information of the manufacturer and the distribution center needs to be made visible to the ODC for order dispatching purpose. Inventory information is thus held in the state of the MANUF and the DC.

There are two alternatives in modeling shared state in an HLA simulation:

1. Request-reply (Pull): Model the inventory as an internal object. When the ODC needs the inventory information, it sends a TSO interaction to both the MANUF and DC, and waits for a TSO reply from each of them.
2. Push: Model the inventory as a TSO HLA object. Whenever the object is updated (inventory), its new value will be reflected at the ODC federate.

Both approaches result in zero lookahead as the current inventory level of the two federates are needed for the order dispatching.

In this paper, we propose a solution to relax the lookahead constraint by replacing TSO interaction/object updates with receive-order (RO) interaction/object updates (RO interaction/object updates are delivered in arbitrary order to the receiving federates.). Even though we formulate the problem around a supply chain simulation, the solution is general enough to be applied to other application domains. Further enhancement to the algorithms, that introduce time guarantee to each update, can be found in Low et al (2003). The paper also gives a more in depth coverage of the implementation issues that have been resolved.

This paper is organized as follows: Section 2 describes some related work, in particular similar work that has been done in parallel simulation. Following that, the proposed approach is discussed in Section 3, with the implementation details being discussed in Section 4. Section 5 presents the performance evaluation study that compares the performance of the proposed approaches to a baseline model where none of the TSO interaction/object updates are replaced by RO updates. Finally, we draw conclusions and outline future work in Section 6.

2 RELATED WORK

Shared state has always been an obstacle to the application of conservative simulation protocols for enhancing the execution speed of simulation. The reason is that shared state introduces zero lookahead to a simulation model as the state variable has to be written and read at an instant of simulation time. A number of studies have been performed to eliminate this constraint. Mehl and Hammes (1993) suggested two general approaches to implement shared variables for a conservative protocol. In the first approach, a history list is kept by the owner of the state variable. When the state variable is requested by a remote entity, say at time t_r, the owner waits until it can guarantee that no future write will be done with timestamp smaller than t_r on the variable. Upon which, the owner retrieves the correct value for the variable from its history list and sends a reply to the requester. Using this approach, the owner can run ahead of the requester in simulation time, but it introduces a disadvantage of suspending the requester during the requesting process. In the second approach, this is avoided by having the requester cache a copy of the variable into its future list. Each cached copy has a time-guarantee associated with it, which indicates the validity of the value. If the requester needs a value, it first looks at its own future list. If a valid copy of the value is found, this value will be taken. Otherwise, a request is sent to the owner. Using this approach, the owner can progress ahead of the requester, and the requester does not have to be suspended to obtain the value if a valid value is cached.

Our approach is very similar to the two approaches proposed by Mehl and Hammes. The difference is that the shared state is only updated locally, and no remote write is allowed in our approach. The owner can thus progress ahead of the requester in simulation time, as no remote update to the state variable needs to be synchronized. This assumption is similar to the work done by Lim et al. (1998). It is particularly useful for shared state variables that are only read by remote entities.

3 THE ALGORITHMS

As discussed earlier, shared state is a critical factor that limits the performance of parallel and distributed simulation that adopts the conservative simulation protocol. The reason is that shared state introduces zero lookahead to the simulation, which means no federates can go ahead in time relative to other federates in a federation. It often results in sequential execution of federates. In general shared state can be implemented using two approaches: 1) request-reply: request for the state value when it is needed, 2) push: keep a copy of the state variable at the requester side, and the owner of the state variable pushes the updates to the requester. Both of these approaches are generally realized using TSO interaction/object updates, which means the

federate that owns the shared variable, and the federate that requests for the shared variable value need to regulate the simulation with zero lookahead.

To resolve this constraint, the TSO interaction/object updates that result in the zero lookahead value can be replaced by RO interaction/object updates, and the timing information is embedded as one of the parameters/attributes of the update. By doing so, owner federates and requester federates no longer need to regulate the federation with zero lookahead as the lookahead value is only computed using TSO messages. They can choose the next lookahead value that is larger than zero to regulate the time advancement of the simulation.

Having said that, the simulation must still be correct as the causality constraint still needs to be satisfied. As RO messages are not considered in the LBTS calculation, it is obvious that the simulation will violate the time order of message processing if no special synchronization mechanism is introduced. This mechanism is realized by introducing a history list to the owner in the request-reply approach, and a future list to the requester in the push approach. These keep a list of values for each state variable, associating a time with each value. This associated time is the simulation time at which the state variable is updated. The requester can obtain the value of the state variable through these lists if a valid value exists. Otherwise, the requester will need to wait until a valid value becomes available. This is how the causality of the simulation is preserved. We are relying on the requester not to move forward in simulation time when its request cannot be satisfied. When the requester does not move forward, it in turn constrains the owner from progressing. This synchronization only happens when the requester needs a value of the state variable, but as the lookahead is no longer zero, it potentially improves the simulation speed.

The history list is used in the request-reply approach to obtain a valid value of the state variable when a value is needed. A requester will issue a request to the owner, say at time t_r. The owner will search through its history list to look for values whose update times satisfy the following condition: $t_l \leq t_r < t_u$, where t_l and t_u are the adjacent update times of the state variable. The update at time t_l is taken, as that is the latest update just before the time of the request. If an update satisfying the condition is found, which means the owner has moved past the time, this value will be pushed to the requester. Otherwise, the request is recorded, and a reply is issued when owner moves beyond t_r. This approach of extending the request-reply approach to use RO messages is known as *pullRO* hereafter.

The future list is used in the push approach to minimize the number of requests being sent out to the owner. A search on the local future list is done before a request is issued to the owner, looking for an update that satisfies the condition of $t_l \leq t_r < t_u$, whereby t_l and t_u are the adjacent update times of the state variable. If the search is successful, the update at time t_l is taken as the reply to the requester. If the search fails, the requester will issue a request to the owner. The push approach will thus work just as the request-reply approach when an update on the state variable cannot be found locally. Such a case happens when the requester runs ahead of the owner in simulation time. This approach of extending the push approach to use RO messages is known as *pushRO* hereafter.

An important issue that needs to be addressed in introducing a history and future list to *pullRO* and *pushRO* respectively is: when shall the state values in the list be discarded, and how frequently this should be done. This is critical as the growing lists will take up too much memory if no fossil collection (discarding old values of shared state that are no longer needed) is done, which in turn might affect the performance of the simulation. However, fossil collection incurs overhead too. Hence, it should not be done too frequently. For the *pullRO*, the owner can discard any shared state values in its history list that have update time less than t_c, where t_c is the minimum federate time in the whole federation. This information can be obtained easily by requesting for the federate time of all federates from the Management Object Model (MOM) (Fullford and Wetzel 1999). As for the *pushRO*, it is very straightforward. Any shared state values in its future list that have update time less than t_c, where t_c is the current federate time of the requester, can be discarded.

4 THE IMPLEMENTATION

In order to avoid the simulation developer from worrying about the detailed realization of the two algorithms discussed in Section 3, a middleware approach is introduced to preserve the API of the RTI. The simulation developer still builds their simulation federate as before, but instead of linking the RTI library to its simulation program, the middleware library (extended RTI), known as RTI+ hereafter, is linked. Figure 1 shows the architecture of the HLA's RTI. The simulation federate talks to the underlying infrastructure through the RTI ambassador, for example sending an interaction or updating an object. The underlying infrastructure talks to the simulation federate through the federate ambassador, for example delivery of interactions or object updates.

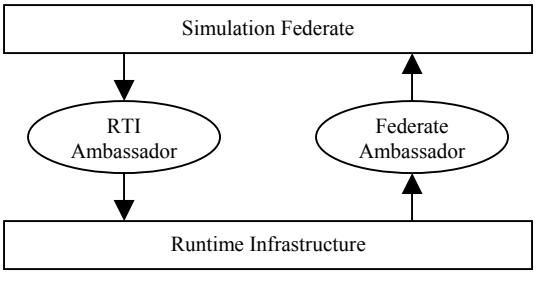

Figure 1: Architecture of RTI

Figure 2 shows the extended RTI architecture, with the incorporation of RTI+. The middleware provides all the RTI ambassador interfaces (known as RTI ambassador+), and filters the federate ambassador callbacks through the federate ambassador+ before passing it to the user's federate ambassador. The RTI+ library is thus comprised of the RTI ambassador+, federate ambassador+, and the original RTI library. Adopting this approach, the only thing that the user needs to do is to alter the TSO interaction/object class that causes zero lookahead to a RO interaction/object class and add a time parameter/attribute to the interaction/object class. All the updates and requests are still carried out as before, and the translation of TSO related calls to RO related calls are transparent from the user.

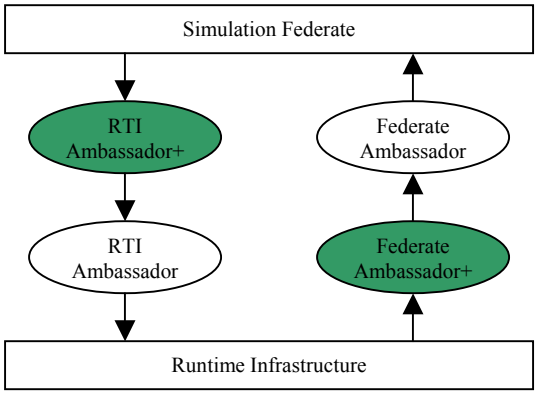

Figure 2: Middleware Approach

4.1 RTI Extension for Shared State

To facilitate the implementation of shared state in a HLA based distributed simulation, we introduce two new RTI ambassador methods that allow a simulation federate to request for an object or class update at a specific simulation time. In the original RTI, only *requestObjectAttributeValueUpdate* and *requestClassAttributeValueUpdate* method calls are available. These two calls do not take in time as their argument, which means a requester will only receive an update at a time that the owner receives the request, rather than at a time that the requester needs the value. Contrary to these two methods, the two new methods take in time as an argument. Figure 3 below shows the API of these methods.

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

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

Figure 3: Shared State APIs for RTI

Both methods translate the call to a TSO interaction that is sent as a request to the owner of the state variable. This interaction contains the object/class handle that the requester is requesting, and the simulation time at which the value is needed. Figure 4 illustrates this mechanism. A call to the request method in the middleware is translated to a TSO *sendInteraction* call in the original RTI ambassador at the requester side. A *receiveInteraction* callback is triggered in the middleware's federate ambassador at the owner side, which in turn triggers the owner to reply through a call to *updateAttributeValue* of the original RTI ambassador. One important point to note here is that the *receiveInteraction* is not allowed to call the *updateAttributeValue* within the federate ambassador+. Hence, the middleware needs to record the request, and initiates the reply phase once the control is returned to the middleware. During the reply phase, the owner will reply to the requester once a valid value for the requested shared state is available. In the meantime, the requester will wait for *reflectAttributeValueUpdate* to be called before it makes any further progress in time.

The naming convention that will be used in the remaining sections of this paper is as follow: method calls in *italic* font are calls to the RTI+ library, while method calls in standard font are calls to the original RTI library. Method calls with a *t* in the bracket are TSO method calls.

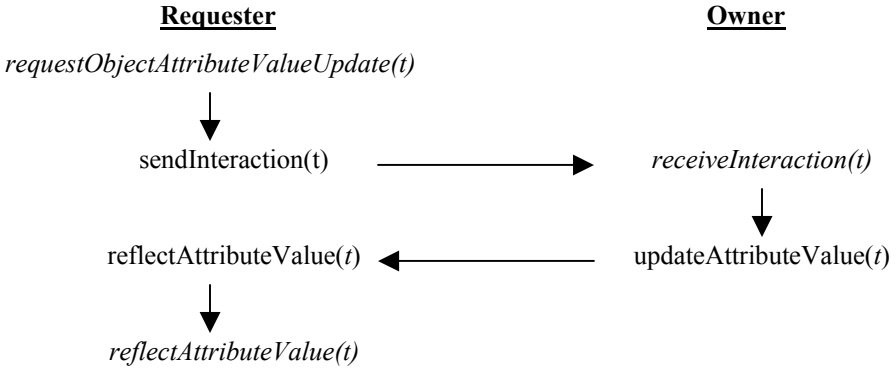

Figure 4: Sequence of Calls for Requesting Object Update – The TSO Approach

4.2 PullRO

Figure 5 illustrates the sequence of method calls to implement the *pullRO* approach. Comparing this to the sequence of method calls in Figure 4, one would notice that the entry and exit point to this sequence of method calls are the same at the requester side. This means that the implementation of the *pullRO* approach is totally transparent to the user. The RTI+ translates the TSO request call to a RO *sendInteraction* at the requester end, and replies to the requester using a RO *updateAttributeValue*. As discussed earlier, the *updateAttributeValue* cannot be called at *receiveInteraction* as program control is still within the federate ambassador. Hence, the request is recorded, and the reply is deferred until the control returns to the middleware. Once the middleware obtains the control, it first looks through the history list to see if a valid value can be found. If a value is found, the *updateAttributeValue* is initiated straightaway. Otherwise, the request is processed when the simulation time of the owner moves past the request time.

At the requester end, once a request is issued, the requester 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 time advance request (a primitive in RTI that federates use to try to advance their time) until the request is received.

4.3 PushRO

Figure 6 illustrates the sequence of method calls to implement the *pushRO* approach, where requested shared state is found in the future list. The *pushRO* approach reverts back to a *pullRO* approach (as illustrated in Figure 5) if the shared state value at the requested time is not available in the future list. All updates to the shared state are pushed towards the requester even if the requester does not need the value. As can be seen from Figure 6, the RTI+ translates the TSO *updateAttributeValue* call to a RO *updateAttributeValue* at the owner side. At the requester side, the RO *reflectAttributeValue* is called to deliver the object update. The middleware at the requester side will then record the object update, and associate the update time to the update. When the requester issues a request for a shared state value, the local future list is first searched. If a valid value is found, the TSO *reflectAttributeValue* is called to deliver the value to the requester. However, it should be noted that the user's *reflectAttributeValue* is not called immediately when a valid value is found. It is deferred until the simula-

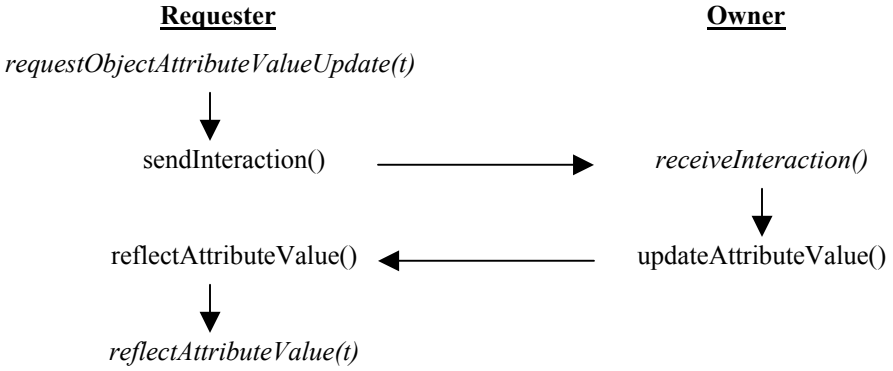

Figure 5: PullRO – Sequence of RTI-Middleware Interaction

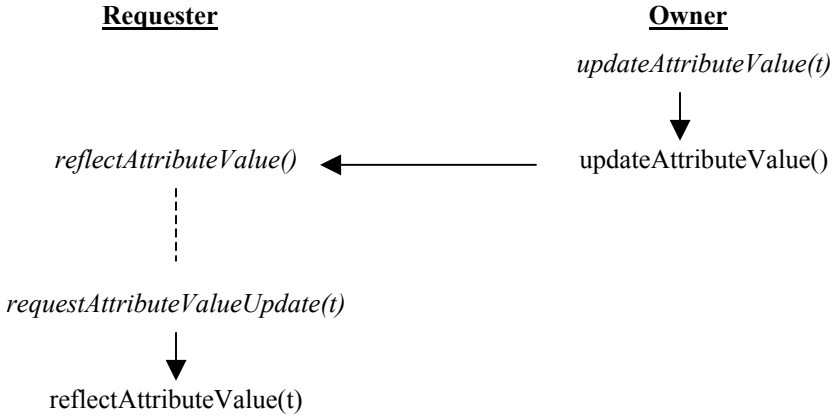

Figure 6: PushRO – Sequence of RTI-Middleware Interaction (Local Copy)

tion federate attempts to pass control to the RTI. This is to ensure consistency between the RTI+ and RTI, as messages are only delivered to a simulation federate when the control is with the RTI.

5 EXPERIMENTAL RESULTS

The performance of the proposed approaches, *pullRO* and *pushRO*, are compared using a simple request-reply simulation model. The simulation model consists of an owner federate and a requester federate running on two computers interconnected by Ethernet. The owner federate periodically updates a shared variable and sends out a user interaction to the requester federate. The shared variable and interaction are updated or sent periodically, with a 100 time unit interval. Requests are issued with an interval of 0.1, 0.2, 0.5, 1.0, 2.0, 5.0, and 10.0 times the update interval, as this ratio will have a significant impact on the performance. Intuitively, a lower request-to-update interval ratio will favour the *pushRO* as requests happen more frequently than updates. On the other hand, a higher request-to-update interval ratio will favour the *pullRO* as it generates much fewer requests than updates. Also, lookahead of 10, 100, and 1000 are used for sending other interactions to mimic models that have other TSO messages to be sent/updated. In the case where the request-reply is sent through a TSO message (known as the *pullTSO* approach), the lookahead will be zero no matter what lookahead value is used, as the request-reply is carried out with zero time increment. But the lookahead of the interaction can be used when the TSO request-reply messages are replaced with RO messages using the *pullRO* and *pushRO* approach. The experiments were run with 10,000 updates.

Figures 7, 8, and 9 show the execution time achieved with varying request-to-update interval ratio, for lookahead values of 10, 100, and 1000 respectively. As can be seen, the execution times for the *pullRO* and *pushRO* implementations generally decrease as the lookahead is increased from 10 to 1000. The two approaches also perform consistently better than the *pullTSO* approach. This improvement can be attributed to the fact that the owner can run ahead (with a larger time grant) of the requester due to the larger lookahead. Whenever the requester needs a value at a specific simulation time, the value is already available either in the history list of the owner in the *pullRO* approach, or the future list of the requester in the *pushRO* approach.

Another general trend that can be observed from the figures is that the *pushRO* approach outperforms the *pullRO* approach consistently with small request-to-update interval ratio. Although there are a large number of requests from the requester compared to the number of updates from the owner, only a small proportion of these are actually sent to the owner. The requester always finds the state value in its future list, as the owner runs much further ahead in simulation time as compared to the requester. For

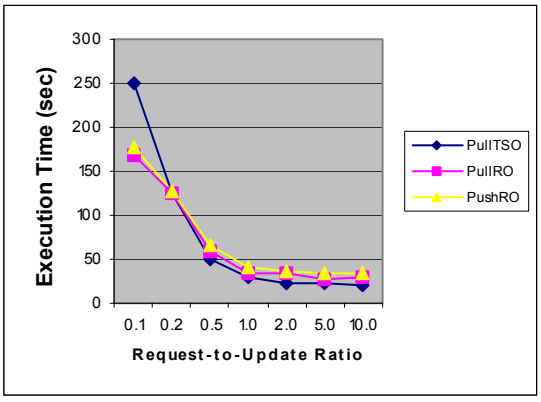

Figure 7: Execution Time vs Request-to-Update Interval Ratio (Lookahead=10)

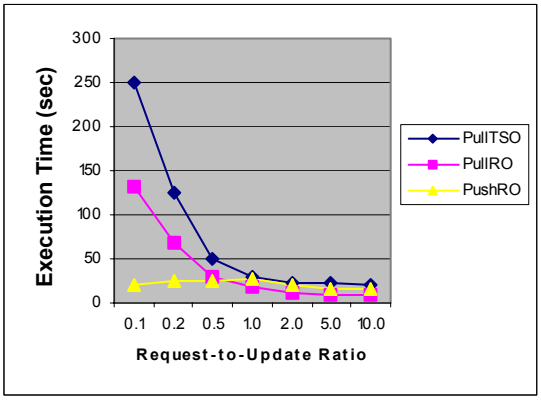

Figure 8: Execution Time vs Request-to-Update Interval Ratio (Lookahead=100)

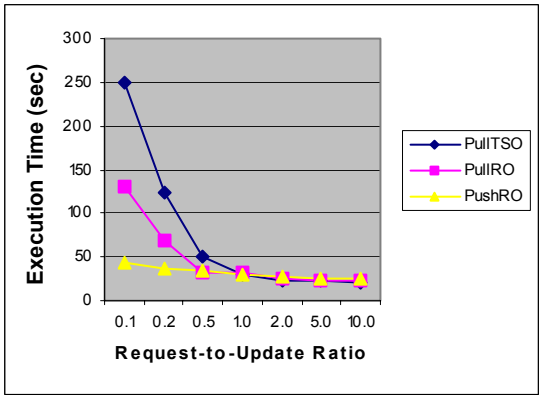

Figure 9: Execution Time vs Request-to-Update Interval Ratio (Lookahead=1000)

example, the number of requests for lookahead of 1000, and ratio of 0.1 is only 25 in the *pushRO* approach but 100,000 in the *pullRO* approach.

Conversely, the *pullRO* outperforms the *pushRO* for large request-to-update interval ratio (refer to Table 1 for the number comparison as the difference in performance is

Table 1: Execution Time Achieved for High Request-to-Update Interval Ratio

	Lookahead=10		
Request-to-Update Ratio	2.0	5.0	10.0
PullRO	33.66	28.34	30.43
PushRO	35.89	34.29	34.04
	Lookahead=100		
Request-to-Update Ratio	2.0	5.0	10.0
PullRO	24.59	21.94	21.79
PushRO	28.08	25.36	24.92
	Lookahead=1000		
Request-to-Update Ratio	2.0	5.0	10.0
PullRO	12.12	9.86	8.94
PushRO	19.95	16.69	16.10

not visible from Figures 7, 8, and 9). This is due to the small number of requests from the requester compared to the number of updates from the owner. This means that the requester is synchronized less often with the owner with a large ratio. For example, the number of request for lookahead of 1000, and ratio of 10.0 is only 1000 in *pullRO* but the number of updates is 10,000 for *pushRO*.

6 CONCLUSIONS

As can be seen from the benchmarking test, the *pullRO* and *pushRO* consistently outperform simulations that do not try to eliminate zero lookahead updates/interactions even though the two approaches introduce the overhead of managing the history and future lists through the middleware. Apparently, the overhead is minimal. We intend to perform scalability tests on *pullRO* and *pushRO* to evaluate if the performance achieved is sustainable with more shared state variables. Also, both approaches will be enhanced to include time guarantee information to each update, such that the requester will have information on the validity duration of an update. This can help to cut down the number of requests being sent by the *pullRO* approach, and also benefits the *pushRO* as the *pushRO* approach reverts back to *pullRO* when the requester always runs faster than the owner.

REFERENCES

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 Spring Simulation Interoperability Workshop*, 99S-SIW-196.

Gan B.P., S.J. Turner, W. Cai, L. Liu, S. Jain and W.J. Hsu. 2000. Distributed Supply Chain Simulation Across Enterprise Boundaries. In *Proceedings of the 2000 Winter Simulation Conference*, ed. J.A. Joines, R.R. Barton, K. Kang, and P.A. Fiswick, 1245-1251. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers.

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*, ed. D.J. Medeiros, E.F. Watson, J.S. Carson, M.S. Manivannan, 1591-1597. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers.

Low Y.H., B.P. Gan, J.H. Wei, X. Huang, S.J. Turner and W. Cai. 2003. Implementation Issues for Shared State on HLA-Based Distributed Simulation. Submitted to *European Simulation Syposium*.

Lutz, R. 1998. High level architecture object model development and supporting tools. *Simulation Special Issue on High Level Architecture* 71 (6): 401-409.

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

BOON PING GAN is a Research Fellow 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>.

MALCOLM YOKE HEAN LOW is a Research Fellow with the Production and Logistics Planning Group at the Singapore Institute of Manufacturing Technology. He received his doctorate 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>.

JUNHU WEI is working with Nanyang Technological University (Singapore) as a Research Fellow, under the sponsorship of Research Manpower Development Pro-

gramme of Singapore Institute of Manufacturing Technology. 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>.