Proceedings of the IASTED International Conference
Parallel and Distributed Computing and Systems
November 3-6, 1999 in Cambridge Massachusetts, USA

Simplifying optimistic distributed simulations in the High Level Architecture

Fernando Vardânega, Carlos Maziero

Programa de Pós-Graduação em Informática Aplicada
Pontifícia Universidade Católica do Paraná
80.215-901 Curitiba - Brazil
e-mail: {*vardanega,maziero*}*@ppgia.pucpr.br*

Abstract

The recent studies in the distributed simulation area are focused in the *High Level Architecture*, defined by the DoD/USA, which proposes a standard environment to develop and run distributed simulations. The HLA components are designed to ensure a high level of interoperability among simulations and, also, to allow maximum component reusability. This paper proposes a new mechanism to help optimist federates perform rollback procedures when needed. This mechanism uses computational reflection techniques to create a rollback manager meta-object that extends the low-level services provided by HLA.

Keywords: parallel simulation, HLA, reflection.

1 Introduction

The research activities in distributed simulation can be classed in two main areas: PADS (*Parallel and Distributed Simulation*) and DIS (*Distributed Interactive Simulation*). Both research areas succeeded to achieve important goals, but several problems remained to be solved, mainly related to the performance aspects, code reuse, and interoperability in heterogeneous environments. The US DoD *High Level Architecture* initiative [6] intends to define and develop a standard software environment in which heterogeneous simulation entities interact using standard interfaces. However, some HLA services are very low-level and hard to use when building simulation models that use peculiar time synchronization schemes. This paper addresses such a problem, and proposes the use of computational reflection techniques to ease building optimistic HLA simulations.

2 Parallel/distributed simulation

The PADS simulation model is roughly composed by an event set, generated by the execution of the processes modeling the system behavior, and variables representing its state [7]. The events are scheduled in a time-ordered list. The sequential scheduler mechanism insures that all causality relations are respected, because events are processed in their chronological order. The main problem in PADS is how to build efficient distributed scheduling algorithms, allowing parallel event processing, and respecting their causality constraints. Two main approaches were proposed to solve this: the pessimistic approach and the optimistic one [8].

The basis of the pessimistic synchronization strategy [2] is to respect the causality constraints local to each process. Considering that processes exchange events through time-stamped messages, one should ensure that all local events older than events received from other processes will be processed before them. This implies on blocking the processing of local events until the causal conditions between them and events received from other processes can be verified [14].

In the optimistic strategy, local and received events can be processed without worrying about local causality constraints. By this, process blocking is avoided, as all available events can be processed. If a message containing an old event arrives at a process, it will be needed to undo the local simulation back to the timestamp of the old event, and then re-do the local simulation considering it. Using this strategy, the events that do not violate causality are confirmed, and the others should be canceled using a rollback mechanism [11, 15].

3 High Level Architecture

In 1995, the US DoD defined a standard architecture for the modeling and simulation of complex systems. It is a high-level, object-oriented architecture, designed to ease the interoperability among different models and to allow component reuse. The HLA - *High Level Architecture*, constitutes a common technical framework for modeling and execution of distributed simulations [6]. Its main components are the *Object Model Templates* [5], the *HLA Compliance Rules* [3], and the *Runtime Infrastructure* [4].

Each HLA simulation is defined by a federation, in which a group of federates interact exchanging data and events. The definition of exchanged data and events is done

using the *Object Model Templates* (OMT), which allow describing the objects that constitute the federation, their attributes and relationships. Each federation defines a *Federation Object Model* (FOM), which describes the shared information used in the federation, and a *Simulation Object Model* (SOM), describing objects, attributes, and interactions used externally in a federation. The HLA compliance rules define the responsibility and relationships among all the federation components [3].

The federates interact using the *Run-Time Infrastructure* – RTI, a distributed framework providing communication and coordination to the federates. The interaction between a federate and the RTI uses method calls from two different classes: *RTIAmbassador* and *FederateAmbassador*. The first one contains all methods offered by the RTI to the federates; its implementation is done by the RTI. The other one is an abstract class, implemented by the simulation programmer, that identifies all methods that each federate should provide to the RTI for callback operations. The HLA services are classed in six categories [6]: *Federation*, *Declaration*, *Object*, *Ownership*, *Time*, and *Data Distribution Management*. The focus of this paper is on time management services, which aims to coordinate the logical time advance and its relationships with the physical time.

4 Time management in HLA

Each federate can use a different time policy with respect to the federation logical time. A federate can be *regulating*, *constrained*, *regulating and constrained*, or *not regulating nor constrained*. In a given federation, it is possible to have federates using any of these time policies, and they can be changed by calling RTI methods. Two main aspects of time management in HLA are *message ordering* and *logical time advance*.

Much of the time management is done by the correct ordering of messages. The messages coming from the federates are queued by the RTI according the existence of timestamps (TSO - *Timestamp Ordered messages*) or not (RO - *Receive Ordered messages*), and according the time policies used by the sender and the receiver. RO messages are put in the federate's FIFO input queue, and are available to it. TSO messages are time-stamped with their sending times. They are put in the federate's time-ordered input queue, and delivered to it in a non-decreasing timestamp order. A TSO message can be delivered to a federate only when the RTI can insure that no more messages having smaller timestamps will be received by that federate.

The logical time advance in the federates is done explicitly, that is, the federate requests the RTI to advance its logical time (through *RTIAmbassador* class methods) and then waits for an approval (through a callback on the *FederateAmbassador* class). This procedure is needed to insure that the federate will not receive any TSO message with a timestamp smaller than its local logical time.

The most common approaches for time management in HLA are *time stepped* (synchronous), *event driven* (pessimistic) and *optimistic* [9]. In the pessimistic approach, the events should be processed according their timestamp order. This approach corresponds to the *event-driven* mechanism in HLA. In the optimistic approach, the events can be processed out of timestamp order. The RTI offers services for message delivering without considering timestamps of TSO messages, and basic rollback mechanisms. However, the rollback mechanisms provided by the RTI cover only the RTI state recovery (message queues, etc). All the management for state saving and recovery in the federate should be implemented by the simulation programmer.

4.1 Rollback actions

In the optimistic approach, the messages carrying events are given to the federates without considering their timestamp order. The *RTIAmbassador.flushQueueRequest* method asks the RTI to give all the queued messages to the federate. After this, the RTI invokes the callback method *FederateAmbassador.timeAdvanceGrant*, authorizing the federate's logical time to progress. If the federate receives an out-of-order (older) message, some procedures should be executed to rollback the simulation, canceling this message and the other messages consumed after it. This recovery procedure includes unrolling the simulation to a execution point before the wrong message's timestamp, re-processing events, canceling scheduled events, and canceling messages erroneously sent to other federates. The message cancellation is done using the RTI method *Retract*, used with the *flushQueueRequest* service (figure 1).

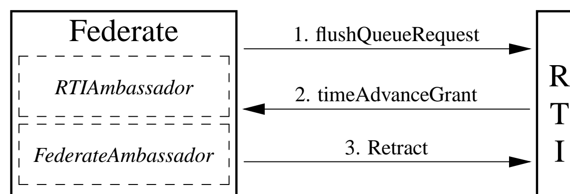

Figure 1: Optimistic federate retraction

If the message to be canceled was already delivered to another federate, its execution should also be rolled back. The RTI calls its *FederateAmbassador.requestRetraction* method, and the federate should then undo any processing done for events received improperly. All these actions should be implemented by the simulation programmer.

5 Computational reflection

Computational reflection is a development technique that allows a system to interact with itself, through a self-representation. Using this, the system can control its own

behavior, allowing a clear separation between the functionality provided by the system to end users and the functions provided to configure and manage the system. This is done through a set of structures used by the system to represent its own aspects, both structural and computational [13].

According [13], a reflexive architecture computational system is constituted by two levels: a *base level* and a *meta level*. The base level is responsible for solving problems belonging to an external domain, normally related to the system's functionality. The meta level is in charge of the control and management of the base level. This allows a better modularity, separating the application code (base level) from the management code (meta level). These concepts can be easily applied to object-oriented systems, associating a *meta-object* to each system object, now called a *base object*. This meta-level structure does not need to be applied on every system objects, but only to those which need to be managed or controlled.

The meta-object is causally connected to the base object, then any changes in the meta-object are be reflected in the base object. When an object is reflected, all its methods are also reflected on the correspondent meta-object. Thus, an invocation on a method in the basis object will be deviated to the corresponding meta-object method. The meta-object methods do their normal processing and eventually call their corresponding methods in the base object.

6 A reflective rollback manager

The mechanism proposed here provides an automatic and generic way to deal with the *requestRetraction* callbacks, freeing the optimistic federate programmer of this complex task. Our proposal uses computational reflection techniques [13, 12] to create a time management meta-level between the RTI and each federate. The time management method calls between them are intercepted (reflected) by a *rollback manager*, which implements the rollback management in behalf of the federate (figure 2).

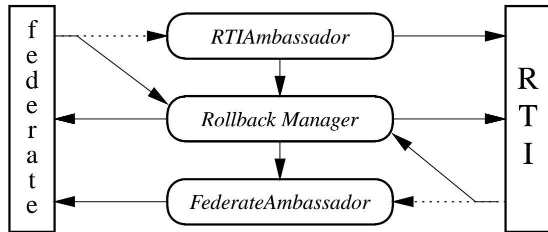

Figure 2: The rollback manager

Using this approach, the rollback manager takes the control of the federate's state rollback, including canceling received or sent messages. The federate will continue calling the same *RTIAmbassador* methods to interact with the RTI, and will receive RTI callbacks through the same *FederateAmbassador* methods. However, some of these methods will be intercepted by the rollback manager. Only some time management methods, mostly related to retraction operations, are intercepted; all the other methods are passed directly to the *RTIAmbassador* and *FederateAmbassador* implementations. The methods to be reflected are those related to time management operations in optimistic federates, as *flushQueueRequest* and *timeAdvanceGrant*. Using this approach, the federate can adopt an optimistic behavior without worrying about possible rollbacks.

The rollback manager should keep snapshots of its federate's state, in order to restore a previous state when a rollback occurs. However, as the manager has no direct access to the federate's internal state, this access should be provided by the federate. This is done through two callback methods, implemented by the federate, that give controlled access to its internal state[1]: a *getState*(S_c,t_c) method, which returns the federate's state at current time t_c in the $[S_c, t_c]$ state vector, and the *setState*(S_s, t_s), which restores the federate's state to the state saved in the state vector $[S_s, t_s]$, with $t_s < t_c$. The rollback manager uses the *getState* method to maintain a list of previous states of the federate, and the *setState* method to restore a previous state, when a rollback is needed.

Using the state saving methods, the rollback manager can save the federate's state at given times in which all TSO events sent to the federate are guaranteed. An event is considered guaranteed if it can be processed without any cancellation risk in the future, unless its retraction is explicitly requested[2].

The calls to the *flushQueueRequest* method are intercepted by the rollback manager, which interacts whit the RTI to obtain the TSO messages. This is done in two phases: initially the rollback manager uses a pessimistic approach to receive the TSO messages from the RTI. Through the method *nextEventRequest*, it requests that RTI deliver all the messages RO available inside its queue FIFO and all the messages TSO with time stamps smaller than the federate's current time. When there are not more TSO messages that match this requirement, the RTI authorizes the federate's time advance, through a callback to the *timeAdvanceGrant* method. This callback passes a future time value t_f as a parameter, to indicate that the federate's logical time can be advanced to t_f.

At this point, the manager had received all the safe messages as stated in the pessimistic approach (section 2), so RTI can guarantee that all the TSO messages with time stamps smaller than t_f had been delivered. This time t_f can be considered as a checkpoint time, indicating a point in the simulation time where the state of the federate is safe, with no rollback risks. Thus, the manager saves the federate's state at t_f as a checkpoint, using the *getState* call defined above.

[1] These two operations are inspired from the Isis system [1], for the replica's state management in groups of fault tolerant processes.

[2] The retraction of events is used inside of some discrete event simulations to model behaviors of interruptions and preemption.

After this pessimist phase, the manager calls the *flushQueueRequest* method on the RTI. At this point, the RTI will deliver all other TSO messages sent to the federate, without worrying about their timestamps. These messages are considered unsafe and can suffer rollback, since the RTI doesn't guarantee that messages with smaller timestamps won't be sent to that federate in the future. If a rollback occurs, the rollback manager has access to all the needed information to undo the processing improperly done, to cancel scheduled events and to restore the last safe state of the federate.

6.1 The rollback procedure

If the federate receives a message older than its current logical time t_c, the federate's state should be rolled back to a previous safe state, in order to guarantee the causality constraints. The need of a rollback operation can be be detected by the rollback manager, because it receives all the messages addressed to the optimistic federate.

In HLA, there are four major event types that can change objects and their attributes. These events should be managed separately by the rollback manager, to allow it to maintain the whole control on all modifications performed in the federate. These events will be described in the next items of this text; at this point we can consider all the received events in a generic way.

When receiving a TSO message the manager will compare its timestamp t_m with the current logical time t_c (the rollback manager is at the same simulation time as the federate it manages). If $t_m < t_c$ a causality violation is detected, and the manager should restore the federate's state to a previous safe state $[S_s, t_s]$ with $t_s < t_m$.

The rollback manager should also keep track of all messages sent by the federate during $t_s < t \leq t_c$, i.e. after the S_s checkpoint, to be able to cancel them. Therefore, the manager can invoke the *Retract* method on the RTI to cancel messages sent to other federates. For doing this, the manager should keep track of all the message handles (*EventRetractionHandles*) for the messages sent during the time interval $[t_s, t_c]$.

In the same way, the manager can receive cancellation requests for messages improperly sent by other federates. The RTI will forward to the receiver the cancellation requests through the *requestRetraction* callback. Normally it is up to the federate to implement the needed procedures to deal with these cancellation messages. In our proposal, the rollback manager will take in charge this task.

6.2 The rollback manager operation

In our schema, the messages received are passed to the rollback manager and later forwarded to the optimistic federate. The messages received with attributes (*Attribute Handle Value Pair Set*) or parameters (*Parameter Handle Value Pair Set*) can be two: *ReflectAttributeValues* (RAV) and *ReceiveInteraction* (RI). In this case, before forwarding the messages to the federate, the rollback manager should save the old attribute values and the federate state to allow a possible rollback.

The mechanism operation can be summarized through the following algorithm, in which F stands for the federate, M for the rollback manager and R for the RTI:

```
while (t_c < t_max) do
  F:next_event_time = timestamp of the
      next local event ;
    flushQueueRequest(next_event_time) ;
  M:intercept the flushQueueRequest() call ;
    nextEventRequest (next_event_time) ;
  R:send RAV/RI events to the federate ;
  M:intercept the RAV/RI events ;
  R:timeAdvanceGrant () ;
  M:advance its logical time ;
    save the federate's current state ;
    forward the RAV/RI events to the federate ;
  F:receive the RAV/RI events ;
    store them in the queue of pending events ;
  M:flushQueueRequest (next_event_time) ;
  R:send remaining (unsafe) RAV/RI events ;
  M:intercept the unsafe events ;
  M:save attributes and recovery handles ;
    rollback the federate if needed ;
  R:timeAdvanceGrant () ;
  M:advance the logical time ;
    forward the unsafe events to the federate ;
  F:receive the RAV/RI events ;
    store them in the queue of pending events ;
  M:timeAdvanceGrant () ;
  F:advance the logical time ;
    process the events present in the queue of
    pending events ;
end;
```

This mechanism can also be presented through a time diagram with all the interactions between the entities (federate, manager and RTI), as shown in the figure 3. This figure shows the interactions during the normal execution of an optimistic federate. When the manager detects a message older than the current time ($t_m < t_c$), it interacts with the federate and the RTI to execute the rollback.

The RTI normally calls the *requestRetraction* method on the federate when a message already delivered to it should be canceled. The event handler *EventRetractionHandle* for that message is passed with the request, which is intercepted by the manager. Using this handler, the rollback manager can recover the old values for the attributes or parameters. The old values were passed to the federate through the methods *ReflectAttributeValues* and *ReceiveInteraction*. Therefore, the federate doesn't need to worry about the cancellation of this event. If an improper processing has resulted in sending some messages to other federates, the manager will request their cancellation through the RTI method *Retract*. The rollback manager keeps track of all messages sent and their handles.

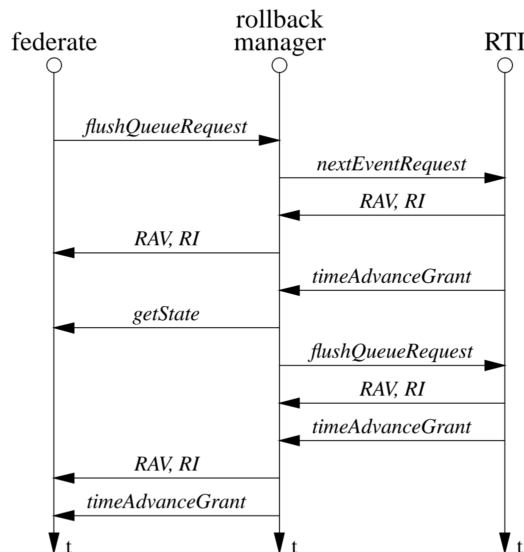

Figure 3: Time diagram for the interactions

7 Conclusion

The use of computational reflection techniques helps the building optimistic federates. All the aspects related to rollback operations can be taken in charge by the rollback manager in behalf of the federate, hiding the complexity of the optimistic approach from the simulation model programmer. The manager is capable to identify the need for a rollback, as well as to take all the proper actions to ensure that the federate returns to a safe state before the causality violation. The rollback manager is generic: all the optimistic federate should have is the correct implementation of the *getState* and *setState* operations.

To validate this proposal, a simple federation is being developed, involving several optimistic federates. In this work we are using RTI version 1.3 for C++. A Java binding package is also being used for the development of Java federates. Our development platform is a Solaris 2.6 Sun workstation with the Java Development Kit 1.1.6. The prototype federation is currently being built, and some preliminary measurements are being done. They show that the impact of the proposed mechanism on the system performance remains acceptable, but more extensive measurements should be done before giving concrete results.

The tests carried out with the rollback manager presented here were done by manually substituting the RTI method calls, to the meta-object methods. There are several programming languages supporting meta-object protocols [12, 10], that could be used to implement the rollback manager, to transparently deal with the reflection aspects.

References

[1] K. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, December 1993.

[2] K.M. Chandy and J. Misra. Distributed simulation: a case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, 5(5):440–452, September 1979.

[3] Defense Modeling and Simulation Office - US DoD. *HLA Compliance Rules*, 1996. http://www.dmso.mil/dmso/docslib/.

[4] Defense Modeling and Simulation Office - US DoD. *HLA Interface Specification*, 1996. http://www.dmso.mil/dmso/docslib/.

[5] Defense Modeling and Simulation Office - US DoD. *HLA Object Model Templates*, 1996. http://www.dmso.mil/dmso/docslib/.

[6] Defense Modeling and Simulation Office - US DoD. *HLA Overview*, 1997. http://www.dmso.mil/dmso/docslib/.

[7] A. Ferscha and S. K. Tripathi. Parallel and distributed simulation of discrete event systems. Technical report, University of Maryland, August 1994.

[8] R.M. Fujimoto. Parallel discret-event simulation. *Communications of the ACM*, pages 31–53, October 1990.

[9] R.M. Fujimoto. Time management in the high level architecture. *SCS Simulation Magazine*, December 1998.

[10] M. Golm. Metaxa and the future of reflection. In *OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, Vancouver, Canada, October 1998.

[11] D. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[12] G. Kiczales, M. Ashley, L. Rodriguez, A. Vahdat, and D. Bobrow. Metaobject protocols: Why we want them and what else they can do. *Object Oriented Programming: The CLOS Perspective*, 1993.

[13] P. Maes. Concepts and experiments in computational reflection. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 147–156, october 1987.

[14] J. Misra. Distributed discrete-event simulation. *ACM Computing Surveys*, 18(4):39–65, March 1986.

[15] H. Sowizral and D. Jefferson. Fast concurrent simulation using the time warp mechanism. In *Distributed Simulation, SCS, Simulation Councils*, pages 63–69, La Jolla, California, 1985.