Federate Migration in HLA-Based Simulation

Zijing Yuan[1], Wentong Cai[1], Malcolm Yoke Hean Low[2], and Stephen J. Turner[1]

[1] School of Computer Engineering
Nanyang Technological University, Singapore 639798
{p144328830,aswtcai,assjturner}@ntu.edu.sg
[2] Singapore Institute of Manufacturing Technology
71 Nanyang Drive, Singapore 638075
yhlow@simtech.a-star.edu.sg

Abstract. The High Level Architecture promotes simulation interoperability and reusability, making it an ideal candidate to model large-scale systems. However, a large-scale simulation running in a distributed environment is often affected by the imbalance of load level at different computing hosts. Migrating processes from heavily-loaded hosts to less-loaded hosts can overcome this shortcoming. We have previously developed a SimKernel framework to execute HLA-based simulations in the Grid environment with migration support as a prominent design feature. In this paper, we will introduce a transparent migration protocol for SimKernel-based federates that minimizes migration overhead.

Keywords: HLA-based simulation, federate migration, load management, Grid computing

1 Introduction

The High Level Architecture (HLA) was developed by the U.S. Defence Modeling and Simulation Office (DMSO) to provide simulation interoperability and reusability across all types of simulations and was adopted as an IEEE standard [1]. The Runtime Infrastructure (RTI) is an implementation of the HLA standard that provides the platform for simulation.

In HLA, a simulation is called a federation and a simulation component is called a federate. Federates communicate with their peers by sending interactions or updating object attributes. The interaction is the formal HLA definition of a transient message of parameter compounds. In this paper, we will use interaction, event and message interchangeably. Federates do not communicate directly with each other and all communication is administrated by the RTI based on each individual federate's interest.

While HLA/RTI aims to provide a software platform for simulation interoperability, the Grid [2] provides an ideal hardware infrastructure for the high performance computing community. There has been research effort to integrate the HLA simulations with the Grid environment to solve large-scale compute-intensive problems [3,4,5,6].

M. Bubak et al. (Eds.): ICCS 2004, LNCS 3038, pp. 856–864, 2004.
© Springer-Verlag Berlin Heidelberg 2004

Large-scale simulations running in a distributed environment are often affected by the imbalance of dynamic load level at individual participating hosts, and this leads to poor performance. Load balancing is a technique that can improve hardware utilization and shorten the execution time. To achieve a balanced load, a process is migrated from a heavily-loaded host to a less-loaded one.

However, process migration generally incurs a large overhead mainly due to poor migration protocol design. This is especially true with HLA-based simulations where not only analyzing and extracting the federate execution state requires tremendous effort, but also migration coordination may undesirably halt non-migrating federates in the simulation. Hence, it would be particularly beneficial to minimize the undesirable large migration overhead.

We have previously developed a SimKernel framework for modelers to design and deploy parallel and distributed simulations in the Grid environment using HLA/RTI. In this paper, we will introduce a migration mechanism based on SimKernel that could minimize migration overhead.

2 Related Works

Federate migration can be achieved at various levels. Obviously, general purpose process migration schemes could also be used to migrate HLA federates. In this paper, however, we focus only on application level federate migration.

The easiest approach to migrate a federate is to utilize the HLA standard interfaces: *federationSave* and *federationRestore*. The drawback is apparent: federation wide synchronization is required. Another side-effect is that all non-migrating federates are required to participate in the federation save and restore process for every migration request. As seen in [5], the migration overhead increases almost proportionally with the number of federates in the simulation.

Other implementations generally adopt the checkpoint-and-restore approach. In both [3,4], the migrating federate's essential state is checkpointed and uploaded to an FTP server. The restarted federate will reclaim the state information from the FTP server and perform a restoration. These implementations introduce further latency since communicating with the FTP server is more time consuming.

Hence, minimizing the migration latency would be of great interest. As migration overhead stems mainly from synchronization and communication with a third party (such as FTP), mechanisms avoiding these issues would be desirable. Such algorithms exist in other migration studies. An interesting *freeze free* algorithm for general purpose process migration was proposed by Roush [7]. In this algorithm, the source host receives messages before the communication links are transferred. Any message arriving while the communication links are in transit will be held temporarily and will be sent to the destination when the links are ready at the new host. Message receipt is only delayed while the communication links are in transit. This greatly reduces process freeze time since non-migrating processes are not involved in the migration. The *migration mailbox* is another approach [8] where a predefined address called a migration mailbox receives mes-

sages for the migrating process. After the process is successfully migrated, it will retrieve the messages from the mailbox and inform other processes to send messages directly to it. The shadow object approach used in avatar migration [9] also achieves the same target. In this approach, each server monitors an interest area and neighboring servers' interest areas overlap in a migration region. An avatar is maintained by a server. When an avatar moves into the migration region, a shadow object is created on the neighboring server. Once the avatar is out of the original server's scope, the shadow object is upgraded to an avatar on the neighboring server and the old avatar at the original server is destroyed.

We adopt a similar approach to the shadow object concept. However, we need to ensure that the restarted federate should be identical to the original federate when a successful migration is achieved. The main concern is to ensure that no event is lost or duplicated. Federate cloning addresses the same problem, and relies on event history logging [10] to ensure the integrity. The approach requires logging every event and may result in a large overhead. We will use an alternative method for message integrity enforcement.

3 Federate Migration

We proposed a *SimKernel* framework for easy development of parallel and distributed simulation using RTI and for deployment of simulation in the Grid environment in [11]. The framework consists of three major components, a federate execution model named SimKernel, a GUI that allows modelers to specify essential simulation information at process level and an automatic code generation tool that translates the modeler's design into executable Java codes for deployment. The SimKernel framework is designed with the following characteristics:

- Simulation design is allowed to be specified at the Logical Process (LP) level.
- Each federate is abstracted to a main simulation loop with event queues inQ and $outQ$ holding incoming and outgoing events respectively (Figure 1).
- All federates adopt the same execution pattern.
- Event interest of a particular federate is specified in a configuration file.
- Event processing detail is defined in a user-defined $consume()$ routine.
- Each federate is identified by a unique literal name at the LP level.

These features facilitate easy federate migration at a higher abstraction level. As the SimKernel is application independent, information transferred at migration time is drastically reduced. In the aspect of code transfer, if the standard library of the SimKernel framework is placed at the destination hosts, the migrating federate can be dynamically reconstructed at the destination with the LP specifications and the received events restored.

Migrating a federate, or process in general, requires the transfer of the program executable and essential execution state. With reference to the features identified above, migrating a SimKernel-based federate can reduce the size of the program executable to be transferred at migration time since the SimKernel code library is application independent. Essential execution state includes any

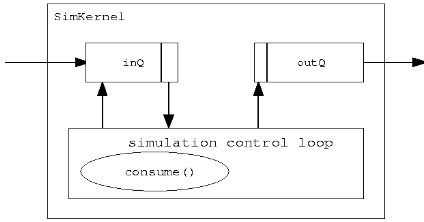


Fig. 1. SimKernel Federate Model

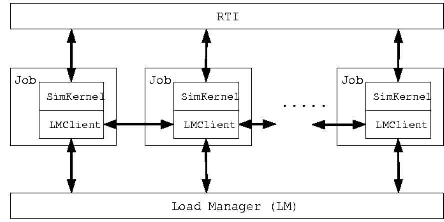


Fig. 2. Migration Architecture

local variables and events in both *inQ* and *outQ*. At migration time, the events in *outQ* will be delivered to the RTI and thus will not be transferred. Hence, only the *inQ* information and the local attributes need to be transferred to the destination. This also reduces the amount of data to be transferred. In this section, we will describe the system architecture and protocol to migrate SimKernel-based federates. In the following text, we will refer to the federate before a particular migration operation as the original federate or migrating federate and the federate created at the destination for migration purpose as the restarting federate.

3.1 Architecture

Our simulation execution support system consists of two subsystems, namely simulation subsystem and load management subsystem. A SimKernel component performs the simulation activity using the HLA/RTI and the Load Manager (LM), with the LMClients at each individual hosts, performs the load management activity (Figure 2).

The LM determines the destination host for the federate to be migrated. The LMClients at the source and destination hosts will communicate until a successful migration is achieved.

The LMClient at each host performs three major tasks. First, it monitors the load level at the host and reports the information to the LM. The information will be used by the LM to determine the federate and hosts involved in migration. Second, on receiving a migration request from the LM, the LMClient will migrate the selected federate using the protocol described in the next subsection. Third, the LMClient will create, suspend, communicate with, and destroy the federate when necessary.

To support migration, the SimKernel main simulation loop is adapted to a state-based process model (Algorithm 1). A set of valid states is illustrated in Figure 3. Most state names are self-explanatory except that state “collected” means that the execution state information is already extracted and transferred to the destination. State “joined” means that the restarting federate has joined the federation execution and has successfully subscribed all events of its interests. State transition is described in the migration protocol. The state “restarting” is not shown in the loop because a restarting federate will join the federation, subscribe and publish its event interests and set the state to “joined” before executing the main simulation loop.

Algorithm 1 SimKernel Main Simulation Loop

```
while (notEndOfSimulation()){
   switch(fedStatus) {
       case running:     processEvent(); // identical to main loop in [11]
                         break;
       case suspended:   flushOutgoingEvents();
                         waitForExecutionStateRequest();
                         performFlushQueueRequest();
                         collectStateInfo();
                         setFedState("collected"); break;
       case collected:   waitForSuccessfulMigrationAck();
                         setFedState("terminating"); break;
       case joined:      waitForExecStateInfo();
                         setFedState("restoring"); break;
       case restoring:   performFlushQueueRequest();
                         checkAndRemoveDuplicates();
                         restoreStateInfo();
                         setFedState("running"); break;
       case terminating: break;
       default:          System.out.println("invalid state.");
   } }
```

3.2 Migration Protocol

Our migration protocol (Figure 4) begins with the LM issuing a migration request to the LMClients at both source and destination hosts. The LMClient at the source host will set the federate state to “suspended”. After the current event if any is processed, the migrating federate sends out all outgoing events in its *outQ*. Then, the federate waits for the LMClient at the destination host to request transfer of execution state. The LMClient will request execution state from the federate only when its peer at the migration destination sends a “requestInformation” request. The LMClient at the destination host will create a new instance of the federate with state “restarting” upon receiving the migration request from LM. The *restarting* federate will proceed to join the federation execution and subscribe and publish any event of its interests with the same configuration of the original federate. After the new federate successfully completes the event subscription (i.e., “joined”), the new federate starts to receive messages of its interest. Note that the new federate is identical to the original one. After the new federate subscribes to the events, both federates will receive messages from the same set of federates. The LMClient at the destination will send “requestInformation” to its peer at the source host when the restarting federate reaches the “joined” state.

When the migrating federate is instructed to collect execution state, it first invokes *flushQueueRequest()* with the parameter of its current logical time, causing the RTI to deliver all messages by calling its *receivedInteraction()* callback regardless of time management constraints. Received events will be stored in the federate’s *inQ*.

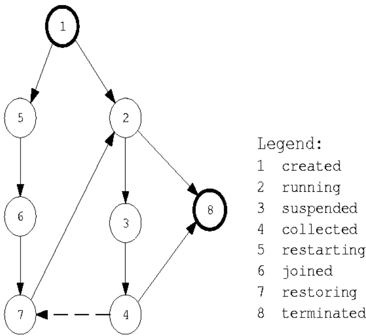


Fig. 3. Federate States

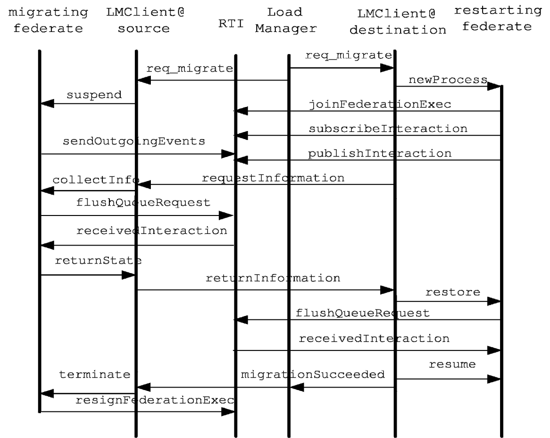


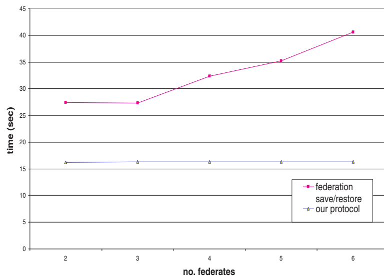
Fig. 4. Federate Migration Protocol

Upon the completion of flush queue request, the migrating federate encodes its internal execution state including events in the *inQ* and local attributes in the attribute table into a formatted string and in the meantime, sets a flag to indicate that execution state data is ready for migration. The federate state is also set to "collected". The LMClient at the host periodically checks the federate until the flag is set and starts to transfer the information to its peer at the destination host. The "collected" migrating federate will set its state to "terminating" after its *migrationSucceededFlag* is set by the LMClient at the host on receiving a "migrationSucceeded" message. Later the information transferred to the destination host is restored by the *restarting* federate.

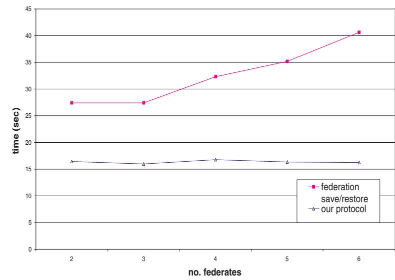
The *restarting* federate begins restoration after it receives the state information. A dynamic class loading technique [12] is used to reconstruct event objects from the string specification transferred from the source host. Reconstructed event objects are inserted to *inQ*. Subsequently, the *restarting* federate invokes *flushQueueRequest()* with parameter of its current logical time to obtain all events sent by RTI since it registered its event interests. When restoring the received information, duplicates are removed.

When the restarting federate has successfully restored its execution state, normal simulation execution resumes. The LMClient at the destination will notify the LM and its peer at the source that the federate is successfully migrated.

Note that the LMClient at each host is regularly updating the load level information to the LM. When an LMClient fails to do so, the host is assumed inactive and not eligible for migration. If the selected destination host is down after the migration decision is made, no socket channel to the host can be successfully created and the LM has to select another destination for migration.



(a) Benchmark Used in [5]



(b) Two-way Super-ping

Fig. 5. Migration Latency Comparison

4 Implementation Results and Discussion

To evaluate the protocol's performance, experiments were carried out in a LAN environment using 2.20GHz Pentium 4 PCs with 1GB RAM. Two test cases were investigated. The first case implemented a two way super-ping, with the total number of nodes (federates) varied from 2 to 6. The second case was identical to the one used in [5], where a federate publishes events, and all other federates subscribe and consume the received events. In both setups, migration requests were issued by the LM every 30 seconds and the event processing time was simulated using random delay. Similar results were obtained in both cases. The migration latency using our protocol was plotted in Figure 5 in comparison to the federation save/restore approach provided by the HLA/RTI standard interface.

Unlike the federation save/restore approach, the time taken for the migration process using our protocol remains constant with increasing number of federates. Migration overhead spans from the time when the original federate is suspended to the time when the migrated federate resumes normal execution. In comparison to other migration approaches, our protocol results in reduced migration overhead due to the following factors:

No explicit federation-wide synchronization is required. Federate migration that employs federation-wide synchronization suffers from poor performance since federates not involved in the migration must also be synchronized.

No communication with third party is required. In our design, migrating a federate requires only peer to peer communication between the source and the destination hosts. This greatly reduces the migration time.

Our framework for simulation design is completely transparent to modelers. Modelers only need to specify at LP level the LPs in the simulation, the events between LPs and the processing details of each event. The design is translated into Java codes by our automatic code generator. This allows modelers to concentrate on the simulation model rather than low level implementation.

Our protocol further benefits the non-migrating federates with complete migration transparency. During the entire migration period, non-migrating federates that interact with the migrating federate continue to send and receive events. These federates have no knowledge whether the federate is processing a message or migrating to another host.

5 Conclusion

In this paper, we have presented a migration protocol for federates based on our SimKernel framework. The protocol adopts a shadow-object like model and reduces the influence on other non-migrating federates to the minimum. Our protocol also achieves complete transparency and reduces overhead without violating the simulation constraints. Although the protocol reduces migration overhead, it still needs to be improved to guarantee complete message consistency. There is a potential problem of message loss if the network is heavily congested. We are working on an improved algorithm to guarantee complete message consistency by providing a counter for each interaction class and verifying the continuity of the counter values. The improved algorithm will be presented in our future publication.

References

1. IEEE: P 1516, Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) - IEEE Framework and Rules (1998)
2. Foster, I., Kesselman, C., Tuecke, S.: The Anatomy of the Grid: Enabling Scalable Virtual Organizations. International J. Supercomputer Applications **15**(3) (2001)
3. Cai, W., Turner, S.J., Zhao, H.: A Load Management System for Running HLA-based Distributed Simulations over the Grid. In: Proceedings of the Sixth IEEE International Symposium on Distributed Simulation and Real Time Applications (DS-RT '02). (2002) 7–14
4. Lüthi, J., Großmann, S.: The Resource Sharing System: Dynamic Federate Mapping for HLA-based Distributed Simulation. In: Proceedings of Parallel and Distributed Simulation, IEEE (2001) 91–98
5. Zając, K., Bubak, M., Malawski, M., Sloot, P.: Towards a Grid Management System for HLA-based Interactive Simulations. In: Proceedings of the Seventh IEEE International Symposium on Distributed Simulation and Real Time Applications (DS-RT '03), Delft, The Netherlands (2003) 4–11
6. Zając, K., Tirado-Ramos, A., Zhao, Z., Sloot, P., Bubak, M.: Grid Services for HLA-based Distributed Simulation Frameworks. In: Proceedings of the First European Across Grids Conference. (2003)
7. Roush, E.T.: The Freeze Free Algorithm for Process Migration. Technical Report UIUCDCS-R-95-1924, UIUC (1995) Available online at `http://www.cs.uiuc.edu/Dienst/UI/2.0/Describe/ncstrl.uiuc_cs/UIUCDCS-R-95-1924`.
8. Heymann, E., Tinetti, F., Luque, E.: Preserving Message Integrity in Dynamic Process Migration. In: Proceedings of Euromicro Workshop on Parallel and Distributed Processing (PDP-98). (1998) 373–381

9. Huang, J., Du, Y., Wang, C.: Design of the Server Cluster to Support Avatar Migration. In: Proceedings of The IEEE Virtual Reality 2003 Conference (IEEE-VR2003), Los Angeles, USA (2003) 7–14
10. Chen, D., Turner, S.J., Gan, B.P., Cai, W., Wei, J.: A Decoupled Federate Architecture for Distributed Simulation Cloning. In: Proceedings of the 15th European Simulation Symposium (ESS 2003), Delft, The Netherlands (2003) 131–140
11. Yuan, Z., Cai, W., Low, M.Y.H.: A Framework for Executing Parallel Simulation using RTI. In: Proceedings of the Seventh IEEE International Symposium on Distributed Simulation and Real Time Applications (DS-RT '03), Delft, The Netherlands (2003) 12–19
12. Liang, S., Bracha, G.: Dynamic Class Loading in the Java Virtual Machine. In: Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'98). (1998) 36–44