DEVS/RMI—An Auto-Adaptive and Reconfigurable Distributed Simulation Environment for Engineering Studies

Ming Zhang[1], Bernard P. Zeigler[1], and Phillip Hammonds[3]

[1]Arizona Center of Integrative Modeling & Simulation
The Department of Electrical and Computer Engineering
The University of Arizona, Tucson, AZ 85721
[3]Joint Interoperability Test Command,
Fort Huachuca, AZ, 85670-2798

Abstract

With the increased demand for distributed simulation and testing environments to support large-scale modeling and simulation applications, much research has focused on developing a software environment to support simulation across a heterogeneous computing networks. In this paper, a new implementation of the DEVS formalism called DEVS/RMI system is presented as a natively distributed simulation system based on standard implementation of DEVS. Our objective is to distribute simulation entities across network nodes seamlessly without any of the commonly used middleware. The DEVS/RMI system is also built to support auto-adaptive and reconfiguration of simulations during run-time. Because Java RMI supports the synchronization of local objects with remote ones, no additional simulation time management needs to be added when distributing the simulators to remote nodes. The DEVS/RMI approach is well suited for complex, computationally intensive testing programs such as the Joint Interoperability Test Command's 6016C standards conformance testing project. The effort and resources required to work in other more traditional High Performance computing environments such as MPI or PVM make development of many simulations tests an unacceptably slow process. In contrast, DEVS/RMI provides an extremely flexible and efficient software development environment for rapid development of tests. A test case of a large-scale dynamic 2D-Cell space model is discussed and presented in this paper to show how the dynamic re-configuration capabilities can be applied to simulations of concern to the test and evaluation community. .

Keywords: DEVS, DEVS/RMI, RMI, distributed simulation, test and evaluation, standards conformance

1.Introduction

Distributed Simulation may be called on to support various scientific and engineering studies, including technical (e.g., standards conformance), system level (focus on a single natural or engineered system) and operational (focus on multiple systems, such as families of systems or system of systems). The objectives of such studies may include testing of correctness of system behavior/function, evaluation of measures of performance, and evaluation of measures of effectiveness and key performance parameters. It is important to note that engineering analyses differ from training applications in that the ability to replicate results is critical, the demand for accuracy is typically higher and the causality and relative timing of events must be preserved. For such applications, an ideal distributed simulation environment would have key attributes, including:

Flexibility – must handle a wide range of dynamic, information exchange an dialogic behaviors
Institutionalized Reuse -- support for model reuse and composability, not only at the syntactical, but at the semantic and pragmatic levels as well.
Model Continuity - allow basic development of systems in virtual time-managed mode, while supporting stage-wise transition to real-time hardware in the loop implementation as well.
Quality of Service – should provide acceptable simulation performance at minimum, and increased performance in dimensions such as execution time when required.

Grid computing [1] is an example of an emerging technology that uses loosely coupled network computers to contribute spare CPU time for high performance computing. Simulation-specific middleware such as High Level Architecture (HLA) and test-range-specific middleware such as the Test and Training Enabling Architecture (TENA) provide higher levels of dedicated support for distributed simulation. However, they only provide partial solutions to address the attributes of distributed simulations required for engineering systems just listed. The Discrete Event System Specification (DEVS) formalism [2,7] provides a more complete solution to an ideal distributed simulation environment when implemented over middleware technologies. Distributed simulation based on DEVS include DEVS/GRID[3], DEVS/P2P[4], DEVS/HLA[5], and DEVS/CORBA[6]. However, such solutions provide only limited support for model distribution, in that the mapping of model components to network nodes is largely a manual process. Moreover, although DEVS and its associated simulation protocol are defined abstractly to support migration to other platforms and languages, the coded implementation still has to be redone for a new context. This means that there is still significant work to migrate a simulation application that works well in one environment to work with different middleware on a different operating system or network.

In this paper we describe an implementation of DEVS called DEVS/RMI that was developed from the standard DEVSJAVA [8] package by extending it to execute in distributed fashion using Remote Message Invocation (RMI) facilities. DEVS/RMI fully supports seamless distribution of simulator/model pairs in a heterogeneous network.

Because Java RMI supports the synchronization of local objects with remote ones, no additional simulation time management, beyond that already in DEVSJAVA, needs to be added when distributing the simulators to remote nodes. It also provides an auto-adaptive and reconfigurable environment for dynamic model re-partition and simulator/model migration. The environment simplifies simulator/model distribution across a network without the help of other middleware while still providing platform independence through the use of Java and its Virtual Machine (JVM) implementations. After presenting its implementation features, we proceed to discuss an application that illustrates its abilities to meet the criteria for distributed simulation to support engineering studies. DEVS/RMI allows us to take advantage of current or future clusters or shared memory assets, whatever their performance capabilities. We need the high-performance capabilities to address the computational complexity needed to thoroughly examine complex natural systems and also to test and certify trusted information systems..

2. Background

2.1 DEVS and DEVSJAVA

Discrete Event System Specification (DEVS) is a mathematical formalism [2] to describe real-world system behavior in an abstract and rigorous manner. With the help of modern object oriented language such as Java, the framework for modeling and simulation based on DEVS has reached a mature stage and has been applied in many real-world applications. DEVSJAVA [8] is an implementation in Java of such a framework that has been used for solving real-world simulation problem as well as serving as an openly available teaching tool. However, currently it has limited support of distributed simulation because it uses Java sockets as the simulator/model communication layer. Java sockets cannot directly support object migration with persistence and do not integrate well with the hierarchical object structure of simulator/models in DEVSJAVA. This also makes it hard to implement distributed simulations of variable structure models and/or need dynamic model partitioning to achieve efficiency [13].

2.2 JAVARMI

Java remote object technology, developed with JDK by Sun [9] several years ago, is easy to use for the programmer and simplifies distributed computing system design. A remote object works just as local object except for the need to maintain a remote reference to locate the remote object. Java RMI hides all low-level communication handling from the programmer. It uses a stub class (the proxy for remote object) to works with other local objects and it automatically supports the synchronization of object method calls while the object resides on a remote JVM. The major problem with Sun's RMI implementation is communication latency due to the inefficient object serialization and marshalling. Some other RMI implementations aimed at high performance have been developed in recent years such as Manta RMI [10], KaRMI [11]. These systems make RMI more attractive for high performance distributed computing including distributed simulation.

3. DEVS/RMI System Description

3.1 System Architecture

DEVS/RMI is a distributed simulation system based on the standard distribution of DEVSJAVA that aims to support seamless distribution of simulation entities across network nodes. DEVS/RMI makes an effort to retain all the existing class structures used in DEVSJAVA while enabling the model and simulator to support java remote object technology (RMI). In this way, distributing the simulator and model can be done without any of the commonly used middleware such as CORBA, HLA, or GRID. Because RMI supports the synchronization of local objects with remote ones, no additional simulation time management needs to be added when distributing the simulators to remote machines. DEVS/RMI maintains all the model and data structures used in DEVSJAVA with expanded capabilities to support remote object technology. This means that a complex model that has been tested on single machine can then be ported to a cluster of computers without any code change. The goal of the DEVS/RMI system is to provide a simulation application with a fully dynamic and re-configurable run-time infrastructure that can handle load balancing and fault tolerance in a distributed simulation environment. A second goal of the DEVS/RMI is to distribute large-scale models to the computing clusters to speedup the simulation or to solve the simulation problems with a size that cannot be handled by single machine's memory or computing power.

As shown on Fig. 3-1, the DEVS/RMI system consists of several key components including simulation controller, configuration engine, simulation monitor and remote simulator. Each of the components will be discussed in more detail in the following sections.

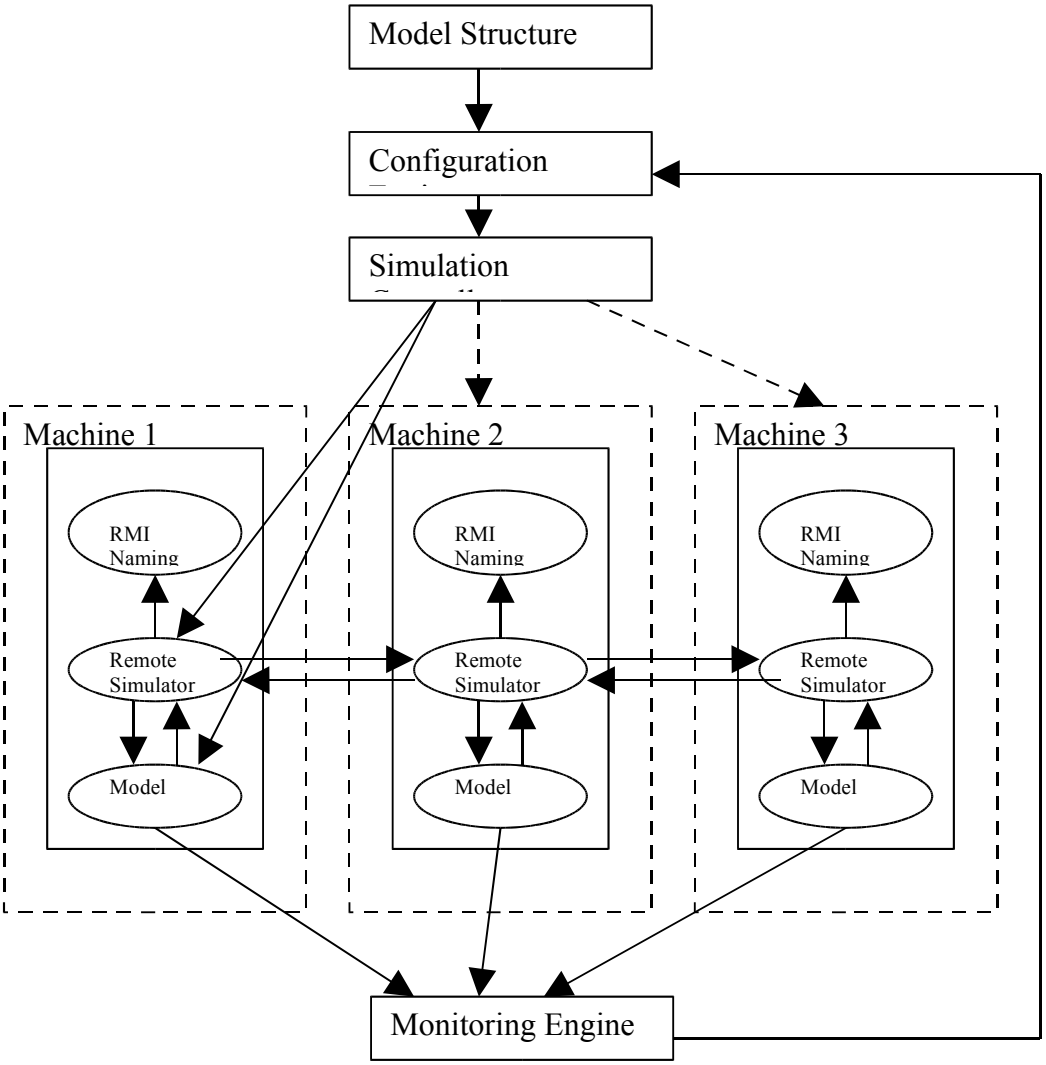


Fig 3-1 DEVS/RMI System Architecture

3.1.1 *Simulation Controller and Configuration Engine*

The simulation controller is the key control unit in the DEVS/RMI system. Its main function is to provide the dynamically generated partition plan from the configuration engine and then create or migrate the necessary simulators/models remotely in the network. The simulation controller can stop and restart/continue the simulation at any stage. The configuration engine is the "brain" of the system that analyzes the model information obtained from the simulation monitor and then applies the built-in partition/repartition algorithm. For example, if the configuration engine decides a new partition plan is needed and then sends this information to the controller during simulation run-time, this controller can then stop the current execution and re-configure the simulation environment. This might involve creating a new set of simulators on selected nodes and migrating existing simulator and model to those nodes. The key concern here is to maintain the model status during the migration. Java RMI supports persistent object migration natively and therefore, the controller does not have to restart

the simulation from time zero and the simulation continues seamlessly. Although not required, the simulation controller and configuration engine can be implemented as DEVS models In general, to implement them as DEVS models simplifies the coupling with other DEVS models in the system.

3.1.2 Simulation Monitor

The simulation monitor is another important component in the DEVS/RMI and provides important information about each running model in the network. The simulation monitor can be implemented also as a DEVS model which has input ports for each local or remote model. The simulation monitor collects the information from those models, measures their activities and then conveys the information to the configuration engine to determine the new partition plan at run-time if necessary. Other ways to implement the simulation monitor are allowed such as using a relational database structure.

3.1.3 *Remote Simulator*

The remote simulator operates according to the same concept and hierarchical structure used in DEVSJAVA. However, the simulator related interfaces and classes are redefined to support remote objects.

Remote simulator classes are created by making the CoreSimulatorInterface and AtomicSimulatorInterface extend the Remote interface. In this way, all the other simulators or coordinators can then be remote objects because they extend the CoreSimulatorInterface and AtomicSimulatorInterface level by level. The following are the code segments for the remote simulator interface and coupledSimulator:

```
public interface RMICoreSimulatorInterface extends Remote{
public void initialize()throws RemoteException;
...
}
public interface RMIAtomicSimulatorInterface extends
Remote,RMICoreSimulatorInterface{
public void initialize()throws RemoteException;
public void initialize(Double d)throws RemoteException;
...

}

public class RMICoupledSimulator extends RMIAtomicSimulator
        implements RMICoupledSimulatorInterface {
...
}
```

In DEVSJAVA, any message object passed among simulators is inherited from the *entity* object. In order to pass messages among distributed simulators in DEVS/RMI, the *entity* interface has to extend Java *Serializable* so that any inherited message class can be transferred by RMI. Following are code pieces for affected entity interface and entity class.

```
public interface EntityInterface extends Serializable{
```

```
...
}
public class entity extends Object implements Serializable,EntityInterface{
...
}
```

3.2 Remote simulator creation and registration

As shown on the architecture in Fig. 3-1, the remote simulators are created by the simulation controller, or more precisely the coordinator. After that, the newly created simulator/coordinator registers itself with the RMI naming server with a unique name for later lookup by the simulation controller. The controller uses the partition/repartition plan from the configuration engine to dynamically create and register these simulators and then parse the corresponding models to the simulators. As shown in the following code pieces in the *RMICoordinator* class, the method *regRemoteSimulator (*IOBasicDevs *model,string machine_name)* accepts the model name and machine name as parameters and then makes a remote method call using the remote reference "ts" to create a simulator with passed model at remote machine. The remote machine then registers the simulator and return the registered RMI URL back to the *RMICoordinator*. The *RMICoordinator* can then use this URL to add a remote reference for the newly created remote simulator using *addRemoteSimulator(iod,regUrl)* method.

```
public void setSimulatorsAuto()throws Exception,RemoteException{
...
if(iod instanceof atomic)
{
String regUrl=regRemoteSimulator(iod, "machine name");
addRemoteSimulator(iod,regUrl);
}
...
}
Public String regRemoteSimulator(IOBasicDevs model,String machine_nm)
throws Exception,RemoteException{
testserverinterface ts = (testserverinterface)Naming.lookup("rmi://" +
                         machine_nm+":1199" + "/testserver");
String url=ts.regSimulator(model,machine_nm);
System.out.println("Remote simulator url "+url);
return url;
}
```

The other way to create remote simulator is to use a static approach, which means to separate the process of creating simulators on local and remote machines. For example, a simulator with corresponding model is created and registered at a remote machine by itself, not through remote method call from *RMICoordinator*. The *RMICoordinator* can then create a remote reference for that remote simulator using predefined URL, in general, a RMI server address plus the model name.

The static approach is more practical for creating remote simulators for large-scale models such as in 2D or 3D cell spaces due to the time required to pass a large number of model components by *value* as required in the dynamic approach.

3.3 Local Simulator vs. Remote Simulator

It is not always effective to create a simulator as a remote reference to a remote simulator. In some case, if the model is set in the same machine as the *RMICoordinator*, it is more efficient to create the simulator just as local object. Fig. 3-2 shows the relation ship among local and remote *CoupledSimulators/CoupledCoordinators.*

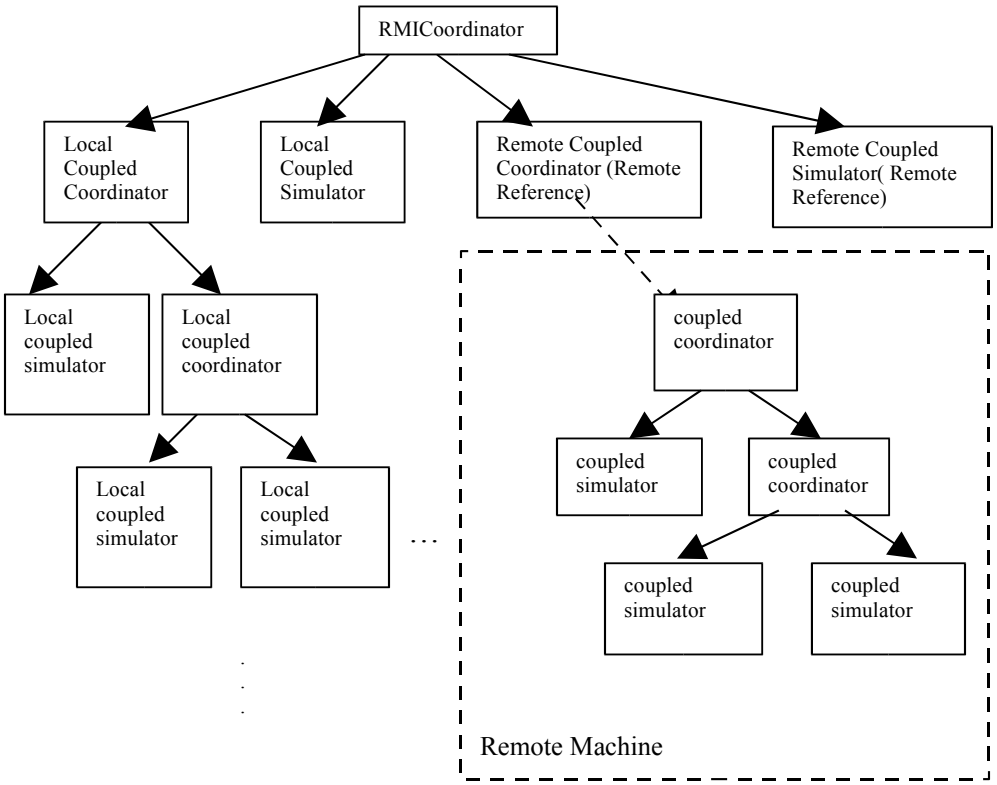


Fig. 3-2 Local vs. Remote Simulator

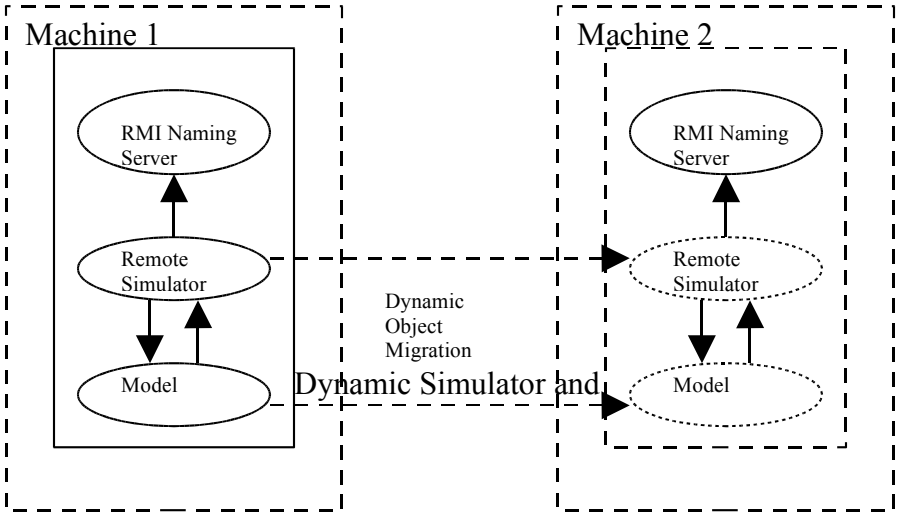
It should be noted that in the *RMICoordinator* construction phase, a simulator reference can be created either as local object reference or as remote object reference. The difference here is that the local simulator object is created and initialized when a local simulator object reference is created; however, when a remote simulator object reference is created, it points to the remote object created in different address space or JVM, which either can be created by dynamic or static way mentioned in section 3.2.
Following is a piece of code that illustrates the creation of local simulator or remote simulator depending on the model's *putWhere()* attribute.

```
public void setRMISimulators() throws Exception,RemoteException{
....
 if(iod instanceof atomic)
     {
```

```
                if(iod.putWhere().equals(localHost.getHostAddress()) ||
                    iod.putWhere().equals(localHost.getHostName())){
                 addSimulator(iod);
                }
                else{
                 String url="rmi://"+iod.putWhere()+":1199/"+iod.getName();
                 addRemoteSimulator(iod,url);
                }
                }
            ....
        }
        tellAllSimsSetModToSim();
    }
```

3.5 **Dynamic simulator and model migration across machine**

The key technology used in DEVS/RMI to make the run-time reconfiguration of simulation possible is Java RMI object persistence. RMI supports the object serialization and reconstruction in remote JVM with persistent data which is the key concern when a model is migrated to another machine during simulation run-time. As shown in Fig. 3-2, The remote simulator/model pair on machine 1 can be dynamically migrated to machine 2 using predefined RMI procedure. The data consistency of the pair will be maintained with only the change of their remote reference in the simulation controller.



3.5.1 Dynamic model reconfiguration in Distributed Environment

Dynamic model reconfiguration has been studied and implemented by Hu [14] in DEVSJAVA as so-called variable structure. It is a very useful method to express the dynamic model structure change for modeling complex and dynamic system. DEVS/RMI natively supports this feature without change of the original implementation if the relevant models are local for *RMICoordinator*. Furthermore, DEVS/RMI can also support the model reconfiguration even if the models are remote to the *RMICoordinator* , by which way the dynamic model structure reconfiguration in a distributed environment is straightforward.

As shown in the following piece of code in atomic class, the model in remote machine can locate the *RMICoordinator* by the attribute *coordWhere,* and then remotely call the *addRMICoupling(...)* or *removeRMICoupling(...)* methods in the *RMICoordinator*, by which way the *RMICoordinator* is updated with new model structure information.

```
public void addRMICouplingR(String src,String p1,String
dest,String p2)throws RemoteException{
digraph P = (digraph)getParent();
if(P!=null){
      P.addPair(new Pair(src,p1),new Pair(dest,p2));
try{
RMICoordinatorInterface rc = (RMICoordinatorInterface)
Naming.lookup("rmi://"+coordWhere+":1200/co");
rc.addRMICoupling(src,p1,dest,p2);
}catch(Exception e1){}
```

3.4 Model Partition in DEVS/RMI

In DEVS/RMI, model partitions can be performed either in a static way or in a fully dynamic way. Models can pass a parameter to remote simulator or they can pre-exist on the remote machine.

3.4.1 Static model partition

Static model partition is implemented in the model construction phase and then manipulated by the corresponding simulator. As shown in the following code, a new ViewableAtomic model is created with name “rainFall” and is put to machine “t2”. In such a way, the model can be assigned to computing node during the initialization phase of simulation. To partition a large-scale cell space using some predefined manner is also straightforward by assigning selected columns or rows of cells to certain node.

```
ViewableAtomic rain_proc = new rainfall("rainFall","t2");
add(rain_proc);
```

3.4.2 Dynamic model partition

Dynamic model partition means that the model partition can be changed during simulation runtime. In dynamic model partition, the models need to be dynamically passed to remote machine as a parameter. The simulation controller can apply a new partition plan with the help of dynamic simulator/model migration supported by DEVS/RMI. The simulation loop can temporarily stopped during this process and then resume its execution after the new partition plan is applied.

3 Experimentation

3.5 Introduction

In order to verify the effectiveness of DEVS/RMI on simulating large and complex dynamic system, it is necessary to select a model system that can represent one of such

kind of system. In this experiment, a valley fever model [15] is chosen to be simulated on a Linux based cluster of computers with the support of DEVS/RMI. Relevant experimental results are collected and analyzed.

3.6 Valley fever model

The Agent-based Valley Fever Model valley fever model initially designed by Bultman, Fisher and Gettings [15] is a 2D dynamic cell space model to represent how the fungal spores grow on a patch of field over a long period of time with given environmental conditions such as wind, rain, moisture and etc. As shown on Fig 4-1, it consists of several individual model components: wind model, rainfall model, coupling control model and patch model. All these components are DEVS atomic models except patch model, a DEVS coupled model consisting of an atomic model called "Sporing Process" and another atomic model called "environment". All these models are put to a 2D cell space DEVS diagraph and the patch models are in fact have x-y coordinator in the cell space. The wind model and the rainfall model are both statistic models which can generated wind data and rain data periodically. The output from them are then sent to the coupling control model which uses the input rain data and wind data to determine the dynamic coupling of rain model with patches as well as the dynamic coupling among patches. This model structure highly dynamic and changes its structure every simulation step.

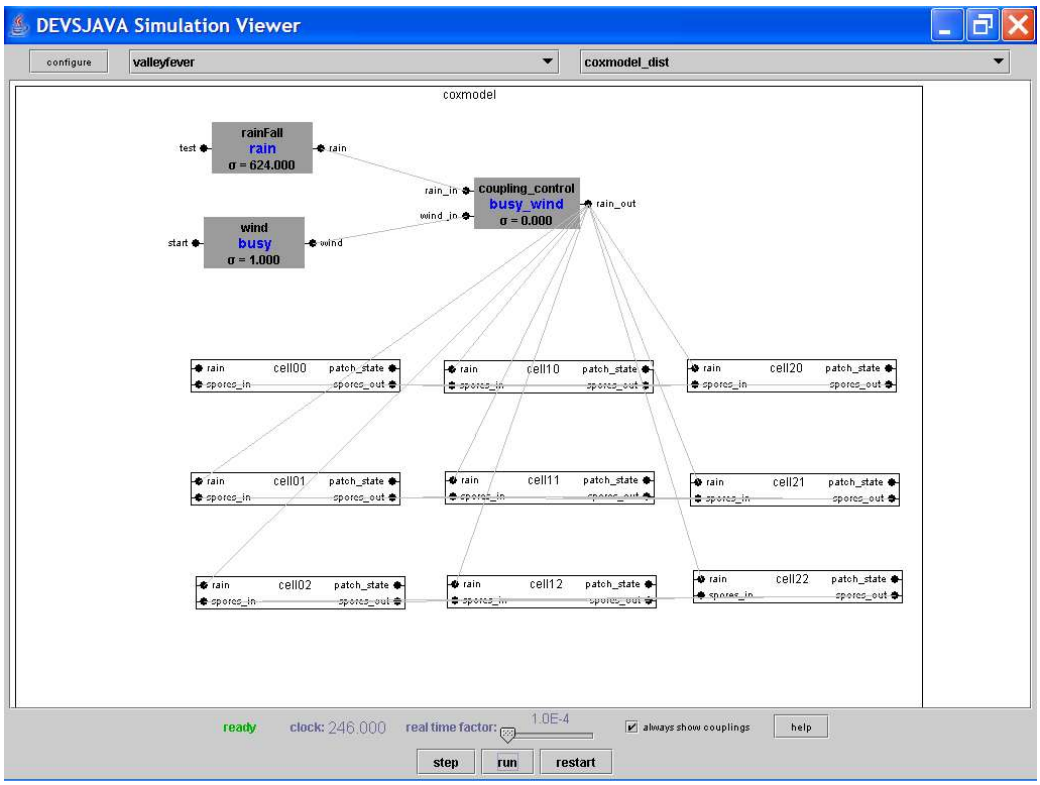


Fig. 4-1 Valley Fever Model in DEVSJAVA SimView

3.7 Distributed Simulation of Valley Fever Model

3.7.1 Linux Cluster

In this experiment, 10 nodes of a 40 node Linux cluster are used to test the valley fever model under DEVS/RMI. Each node in the cluster has a AMD Athlon XP 2400+ with 2GHz CPU and 512M physical memory and all the nodes are connected by 100M Ethernet switch. The operating system of each node is GNU/Linux 2.4.20 with Java Runtime 1.4.1-01 installed.

4.3.2 Partition Model Components to Clusters

In this experiment, the static model partition mentioned above in section 3.7.1 is used in order to reduce the cost of dynamically creating remote simulators with model passing as parameter. Several different partition methods are implemented for testing, it was found that the distributing other model component except patch cells did not have much difference regarding the simulation execution time. With the increase of the patch cells for the model, it is easier to see that it is necessary to partition these cells.

Therefore, the patch cell space are evenly divided by columns according to the number of computing node, for example, to partition a 10 by 10 patch spaces on 5 nodes, every two columns of cells will be put to one node. All the other model components are put with the *RMICoordinator*. On each computing node, a program called *"testserver"* is started to start RMI registry by itself and create a set of simulator/model pairs which belong to this node according the model's *putWhere()* attributes, then the references of the simulators are registered with RMI registry for the lookup by the *RMICoordinator*. After above mentioned initialization of each node, the *RMICoordinator* is started at the main simulation control node.

As shown in Fig. 4-2, a 10 by 10 patch space is divided into 1, 2, 5, 10 nodes respectively to measure the total simulation time spent in terms of 100 simulation loops. It can be seen that the simulation time has a significant increase with the increase of the computing nodes due to the added communication overhead, however, there is no significant execution time increase for simulating the model among two nodes, five nodes and ten nodes.

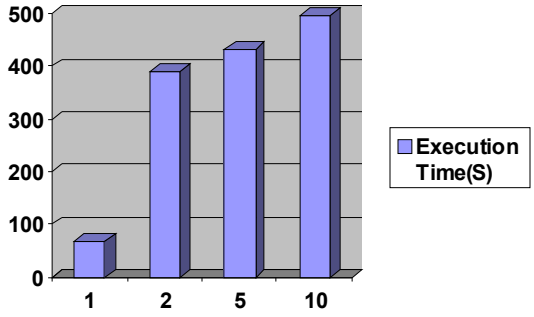


Fig. 4-2 Simulation execution time(seconds) vs. No. of Computing node in original model

3.8 Workload injection to the distributed cells

From the experiment result obtained from section 4.3.2, it can be inferred that the increase of simulation execution time is due to the increased communication load among the nodes. It is worthwhile to examine what can happen if the workload is increased on distributed cells without increasing the communication load such as number of RMI calls through network. A 5 by 5 cell space is tested in terms of 400 simulation loop because larger cell spaces can cause unacceptable delays for collecting results. Fig 4-3 shows the situation changes when different workload is injected to the distributed cells. For the left figure, whenever each patch cell gets an external event, it will calculate the sum of integer from 1 to 100, it can be seen that the total execution time of overall simulation is slightly increased for 5 nodes compared with for 1 node. For the right figure, whenever each patch cell gets an external event, it will calculate the sum of integer from 1 to 150, the total execution time of overall simulation is greatly reduced when using 5 nodes compared with using only 1 node. It can be then seen that the workloads on the distributed cells play an important role in affecting the performance of the distributed simulation with DEVS/RMI. For the original valley fever model, the distributed cells do not have enough workload to compensate the cost of increased communication incurred by RMI calls across network.

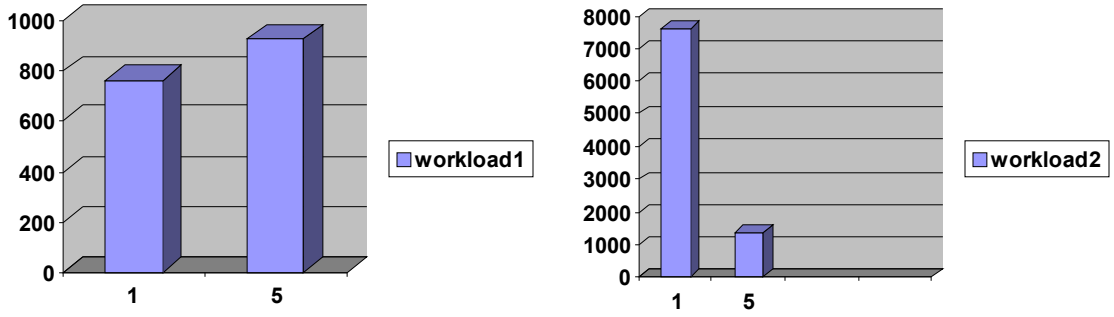


Fig. 4-3 Simulation execution time(seconds) vs. No. of Computing node under different workload on distributed cells

4 DEVS/RMI Performance Issues

3.9 Speedup of the Simulation with Increased Workload

3.10

From the experiment performed on the cluster, it was found that the performance of DEVS/RMI highly depends on the model component partition, especially the model component workload partition. For example, if the distributed components or cells have less workload, the performance of the simulation can become worse compared with that in single machine due to the latency of the network and the remote method calls among distributed simulators. When the workload on cells increases without increasing the number of RMI calls, such as in the testing case, the speedup of the simulation is significant when running the model on cluster with DEVS/RMI.

3.11 Dynamic Model Reconfiguration in Distributed Environment

Dynamic model reconfiguration in distributed environment is fully supported in DEVS/RMI. However, the change of model structure such as coupling information among distributed machines is costly due to the RMI calls and network latency. For instance, if the coupling control model in the experimental case is placed with the *RMICoordinator*, the dynamic coupling is then determined by local calls instead of remote method calls, thus, the reduced cost from RMI calls can then benefit the overall performance of the simulation on cluster.

3.12 Simulation of Large Models

Another important issue that needs to be considered is that it is impossible to simulate a valley fever java model in single machine with a cell space larger than 85 by 85 because of limited memory. This implies that there is a limitation of problem size when simulating large-scale cell space on single machine, which can be solved by distributing the large size model to a computer cluster with DEVS/RMI. In such situations, distributing the model to multiple computing nodes is the only solution so long as the performance is not overly degraded by a communications burden.

3.13 Other Performance Considerations

In general, Sun's RMI used in DEVS/RMI should not be the best implementation for high performance distributed simulation. However, a large-scale model still achieve performance advantages using DEVS/RMI if the distributed model components can be designed to have a large workload. If an alternative high-speed RMI protocol is implemented in DEVS/RMI, it can be expected that a high performance fully object-oriented distributed simulation environment can be built to solve very complex and large simulation problem.

7 Applications to Test and Evaluation

We have developed the DEVS/RMI approach because it is well-suited for complex, computationally intensive workload testing programs such as the Joint Interoperability Test Command's 6016C standards conformance testing project. This testing project is a semi-automated suite of tools that translates the natural language standard and converts it to an XML expression of the standard that can then be evaluated and translated into a series of automated tests. The analysis of the dependencies of the standard and the subsequent automated test case generation require "higher" performance or traditional high performance computational platforms in the form of clusters or shared memory systems in order to produce comprehensive scientific experimentation and robust standards conformance testing. Other applications of DEVS/JAVA based RMI HPC include automated test case generation for other standards, the local and distributed analysis of test data, and development of monologic and dialogic data for testing scenarios such as those used in Joint Distributed Engineering Plant (JDEP). The effort and resources required to work in other more traditional HPC environments such as MPI

or PVM make development of time critical experiments and tests an unacceptably slow process. In contrast, DEVS/RMI provides an extremely flexible and efficient software development environment for rapid development of tests.

5 Conclusions

In this paper, DEVS/RMI system is presented as a natively distributed simulation system based on standard version of DEVSJAVA with support for dynamic structure models, auto-adaptive and reconfigurable simulation. This system reduces the overhead that is added by middleware solution for the distributed simulation. A test case of a large-scale 2D dynamic cell space model was studied and analyzed to demonstrate the effectiveness of employing the DEVS/RMI system on a cluster of computers.

We have noted that scientific and engineering analyses differ from training applications in that the ability to replicate results is critical, the demand for accuracy is typically higher and the causality and relative timing of events must be preserved. Let us review to what extent the DEVS/RMI system can address the requirements of a highly demanding distributed simulation environment as expressed earlier.

The DEVS formalism affords the ***flexibility*** to handle a wide range of dynamic, information exchange an dialogic behaviors. Its rigorous framework provides for accurate modeling and simulation and time management that is provably correct.

DEVS's modular and hierarchical model construction and associated abstract simulator protocol support model reuse and composability with dynamic system semantics thus enabling ***institutionalized reuse*** of simulation model assets.

DEVS/RMI supports DEVS ***Model Continuity***. *Although restricted models developed in DEVSJAVA*, DEVS/RMI supports basic development of systems in virtual time-managed mode through their model implementations in a single machine, and enables easy distribution of such models over networks and clusters to gain the advantages of distributed simulation. Although currently DEVS/RMI does not support stage-wise transition to real-time hardware in the loop implementation, the work by Hu[14] to support model continuity in DEVSJAVA can be extended using the same RMI approach as discussed here.

We have shown how DEVS/RMI can provide ***Quality of Service*** in that it can provide acceptable simulation performance when migrating DEVSJAVA from single machine to distributed form. For future work, better RMI implementations other than Sun's, such as Manta RMI system, needs to be considered to get higher performance. Further, repartition algorithms for hierarchical DEVS models, such as defined by Park [13], need to be implemented in DEVS/RMI and compared with their implementations on existing distributed DEVS simulation environments. With the native support of RMI, no additional simulation time management needs to be considered when distributing the simulators to remote

nodes because RMI supports the synchronization of local objects with remote ones. Also migration of models with persistent data can be done seamlessly. For these reasons, we believe that comparison with other implementations will establish performance advantages for DEVS/RMI.

It is interesting to note that rapid technological advances continually change the definition of high performance computing. For example, the HPC capabilities of the 60s and 70s are surpassed by today's laptop computers; even a relatively small Linux cluster can now out perform most massively parallel systems from the 80s and early 90s. Being based on a mathematical formalism, such as DEVS, the methodology we describe in this paper is not hardware technology dependent. Indeed, using DEVS/RMI, we can take advantage of current or future clusters or shared memory assets, whatever their performance capabilities. We need the high-performance capabilities to address both complexity and temporal demands of experimentation and testing. In the applications described above, multiple processors are needed to handle computations that are interdependent, and we need the results to provide standards conformance or interoperability certification as quickly as possible to deliver critical experimental results thoroughly tested systems.

References

1. www.gridforum.org/
2. Bernard P. Zeigler, Tag Gon Kim and Herbert Praehofer, "Theory of Modeling and Simulation", Academic Press, 2000.
3. Chungman Seo, Sunwoo Park, Byounguk Kim, Saehoon Cheon, Bernard P. Zeigler, "Implementation of Distributed High-performance DEVS Simulation Framework in the Grid Computing Environment" , 2004 High Peformance Computing Symposium.
4. Saehoon Cheon, Chungman Seo, Sunwoo Park, Bernard P. Zeigler, "Design and Implementation of Distributed DEVS Simulation in a Peer to Peer Network System" , 2004 Military, Government, and Aerospace Simulation.
5. Jong-keun Lee, Min-Woo Lee, Sung-Do Chi, "DEVS/HLA-Based Modeling and Simulation for Intelligent Transportation Systems", SIMULATION, Vol. 79, No. 8, 423-439 (2003).
6. Bernard P. Zeigler, Doohwan Kim, Stephen J. Buckley, "Distributed supply chain simulation in a DEVS/CORBA execution environment", December 1999 Proceedings of the 31st conference on Winter simulation: Simulation---a bridge to the future - Volume 2.
7. Bernard P. Zeigler, Yoonkeon Moon, Doohwan Kim, Jeong Geun Kim, "DEVS-C++: A High Performance Modelling and Simulation Environment", January 1996, Proceedings of the 29th Hawaii International Conference on System Sciences (HICSS'96) Volume 1: Software Technology and Architecture.
8. http://www.acims.arizona.edu/SOFTWARE/devsjava_licensed/DevsJavaUserGuidev1.1.zip.
9. http://java.sun.com/docs/books/tutorial/rmi/
10. Jason Maassen, Rob van Nieuwpoort, Ronald Veldema, Henri E. Bal, and Aske Plaat. "An efficient implementation of Java's remote method invocation", In Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP, pages 173-182, May 1999.
11. Michael Philippsen, Bernhard Haumacher and Christian Nester, "More Efficient Serialization and RMI for Java", In Concurrency: Practice and Experience 12(7):495-518, John Wiley & Sons, Ltd., Chichester, West Sussix, May 2000.
12. Ntaimo, L., Khargharia, B., Zeigler, B.P, and Vasconcelos, M., "Forest fire spread and suppression in DEVS", SIMULATION, Vol 80, No. 10.
13. Park, Sunwoo, "Cost-based partitioning for Distributed Simulation of Hierarchical Modular DEVS Models", Ph. D. Dissertation, The University of Arizona, 2003

14. Xiaolin Hu, Bernard P. Zeigler and Saurabh Mittal *Dynamic Reconfiguration in DEVS Component-based Modeling and Simulation,* Simulation: Transactions of the Society of Modeling and Simulation International, November 2003
15. R. Jammalalika, et. al., Re-implemenation of an Agent-based Valley Fever Model (Originally Developed by Bultman and Fisher by Gettings) in DEVS, DEVS Symposium, April, 2005.
16. X. Hu, Ph. D. Dissertation: A Simulation-based Software Development Methodology for Distributed Real-time Systems, Fall 2003, Electrical and Computer Engineering Dept., University of Arizona

Acknowledgement and Thanks

The authors would especially like to acknowledge and thank Dr. Mark Gettings and Dr. Mark Bultman of the USGS for the use of the USGS Beowulf cluster and for the opportunity to use their experimental frame and data. We look forward to cooperating and supporting their projects more in the future.