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

DISTRIBUTED SUPPLY CHAIN SIMULATION USING A GENERIC JOB RUNNING FRAMEWORK

Haifeng Xi
Heng Cao
Leonard Berman
David Jensen

IBM T. J. Watson Research Center
Yorktown Heights, NY 10598, U.S.A.

ABSTRACT

For supply chain performance simulation that involves aggregating results from multiple runs of the same underlying model, simulation iterations can be distributed to networked computing resources to achieve significant speedup. This paper presents a generic distributed job running framework that facilitates such high performance supply chain simulation. We first introduce a supply chain modeling and simulation tool developed by IBM Research, and summarize the strategy to enhance it. A closer look is then taken at a generic job running framework we designed and how it was used to bring the distributed simulation capability to the tool. After reviewing an ongoing effort to integrate the new tool with the IBM MathGrid environment, we conclude the paper with a brief discussion of our future work.

1 INTRODUCTION

Recent tenors in enterprise evolution, driven and enabled by e-business initiatives, are making our current solutions to supply chain simulation look obsolete. Most notable among those trends is the ever-increasing complexity of enterprise supply chains in terms of both depth and breadth. On one hand, a new round of business process reengineering calls for tighter and more effective integration of a company's supply chain with its various business processes. On the other hand, a company's supply chain is quickly becoming an extended value chain that encompasses its suppliers, customers and partners.

With the mounting need to conduct complex simulation and the steadily decreasing cost of networked computing resources, parallel and distributed simulation is becoming more and more attractive as an effective means to improve simulation performance. See Fujimoto (1999a) for a good survey of distributed simulation strategies. However, almost all the algorithms surveyed attempt to explore the parallelism existing in the simulation models and maintain the causality constraint (i.e., events must be processed in the order specified in their timestamps) when decomposing the simulation model into a collection of logically interdependent processes. These logical processes are executed by distributed simulation engines in parallel. In order to keep the causality constraint, time advancement in those simulation engines needs to be carefully guided by a strategy that ensures proper synchronization among the distributed processes. For example, GRIDS is a generic runtime infrastructure that facilitates simulation of such coupled models on distributed hosts (Sudra et al. 2000).

The synchronization strategies can be classified as being either *conservative* or *optimistic*. The former strategy strictly maintains the causality constraint, making sure that an event will only be processed when no other event with an earlier timestamp will arrive in the future. The latter strategy relaxes the causality constraint in order to explore parallel execution opportunities; nevertheless when constraint violation does happen, the state of involved logical processes must be rolled back. However, unless there are reasonably low causal dependencies among decomposed sub-models, the former strategy can render simulation engines idle for a significant percentage of their running time simply waiting for synchronization messages, while the latter can lead to more resource-wasting rollbacks.

Unfortunately, many rollbacks would be needed for supply chain simulation, where the underlying model consists of objects whose behaviors depend closely upon one another's actions and internal states. Therefore, instead of trying to decompose tightly coupled supply chain models, we chose to exploit the parallelism of the simulation execution from a different and more natural perspective. To fully capture the uncertainties existing in supply chain processes, it is common practice to run a simulation for multiple iterations with different values of involved random variables. These independent iterations can be run in parallel without the need to communicate with each other, causing virtually no overhead at all. In other words, our approach views the complete supply chain simulation process as a coherent job that can be decomposed into a set

of weakly coupled tasks each representing one simulation iteration under a different parametric setting.

The next section gives a brief introduction to an existing Supply Chain Modeling and Simulation tool (sometimes referred to as "the SCMS tool" or simply "the tool") developed within IBM Research, and discusses the design principles for its enhancement. Section 3 presents a generic framework we designed that can be used to build distributed job running systems. Section 4 examines how this framework is employed to bring distributed simulation capability into the SCMS tools, followed by some benchmarking results in Section 5. After reviewing an ongoing effort to integrate the tool with IBM MathGrid Desktop in Section 6, we conclude with an outlook for future work.

2 SUPPLY CHAIN MODELING AND SIMULATION TOOL

IBM Research has been active for several years in pursuing effective solutions to supply chain simulation. Supply Chain Analyzer (SCA) achieved significant success in both internal supply chain reengineering and external consulting business (Bagchi et al. 1998). Although SCA was sold to a supply chain vendor in 2000, simulation has remained a powerful methodology in IBM to predict supply chain performance and to facilitate business process transformation. Recently, a new Java-based supply chain modeling and simulation tool (Figure 1) has been developed in IBM Research and successfully applied to a variety of internal processes. For example, IBM Enterprise Server Group has used the tool to simulate days-of-supply to optimize its inventory policy (Cao et al. 2003); a few Sense-and-Respond pilot projects have used it for supply chain performance evaluation and risk analysis (Lin et al. 2002). The tool has the following main features:

2.1 Interactive GUI Driven Modeling

The modeling environment comes with visual widgets corresponding to supply chain building blocks, and enables drag-and-drop model composition. The building blocks include both supply chain entities like *Manufacturer*, *Supplier* and *Customer* etc., and logic nodes like *merge* and *switch*. For each building block, input pads and output pads are defined according to its event handling capabilities. The relationships among supply chain entities are established through the *Arc* widget which links one supply chain entity's output pad to another one's input pad.

2.2 Agent-Based Model Representation

The tool follows the Java *Delegation Event Model* (Sun Microsystems 1999) design pattern. Behind those visual widgets are software agents that have different event dispatchers (source) and handlers (sink). Each dispatcher or handler has an internal queue, where unprocessed events are ordered by their timestamps. Once a user links two supply chain entities with an Arc widget, the underlying event handler in the sink entity will be registered with a corresponding event dispatcher in the source entity.

2.3 Discrete Event Simulation Engine

The simulation engine maintains an internal clock for the "simulated time". During any simulation step, the engine will first determine the earliest timestamp of all the unprocessed events in the model, and then update its clock with this timestamp and broadcast it to all the simulation agents. Upon receipt of such a time update, an agent will process its queued events that have the same timestamp as the one received. As part of the event processing, new events can be generated and queued up. This iterative process will go on until there are no unprocessed events in the model, or the end of the specified simulation duration is reached.

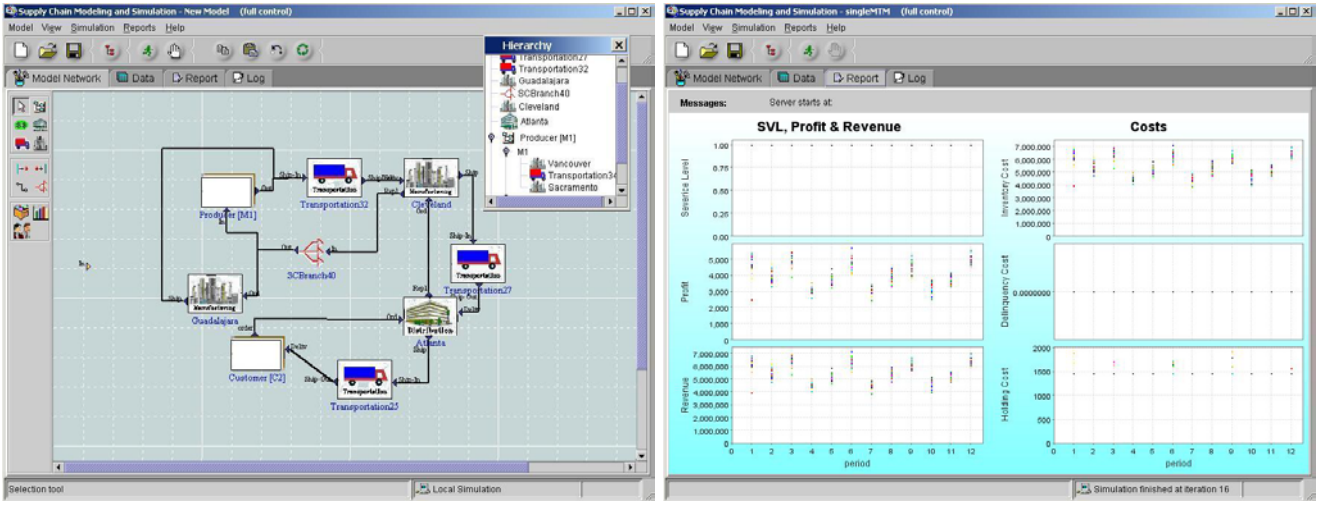

Figure 1: Supply Chain Modeling and Simulation Tool

2.4 Animation for Visual Design Validation

The tool has animation capability built in to visualize event passing during simulation, as a quick way to validate the model design and parameter setting.

2.5 Dynamic and Consolidated Reporting

After a supply chain model is designed and before it is executed, the modeler can select a number of performance metrics of interest to him/her, such as customer serviceability, inventory in terms of safety stock or WIP, and various cost measures. During the model execution process, simulation agents gather relevant information and send them back to a reporting component, which calculates values for those metrics and present them graphically in dynamic reports. At the end of the simulation, results from all the iterations will be consolidated into a final report.

As model complexity keeps growing, a decision was made to extend the SCMS tool to take advantage of supply chain simulation parallelism as explained in Section 1. There were two concerns when we set out with the enhancement tasks. First, since the current code base is pretty stable, we would like to minimize code refactoring by following the "if it isn't broken, don't fix it" rule. Specifically, we are interested in making distributed simulation look transparent to other parts of the tool, meaning that any existing Java event types and Java listener subscription relationships between the simulation engine and the front-end GUI should remain untouched. Section 4 explains in detail how we managed to do that.

The other concern is that we don't want the implementation to be tied with any particular distributed protocol at compile time. Instead, we would like to define an abstraction of the distributed communication layer and to be able to "plug in" a particular implementation at runtime through some configuration mechanism. The benefit of doing so is runtime configurability which allows the tool to use the most appropriate implementation based on runtime requirements such as network security, scalability, and performance etc. The following section discusses a generic framework designed for that purpose.

3 GENERIC DISTRIBUTED JOB RUNNING FRAMEWORK

The framework is composed of a set of Java classes, interfaces, and abstract classes (Figure 2). It is generic in the

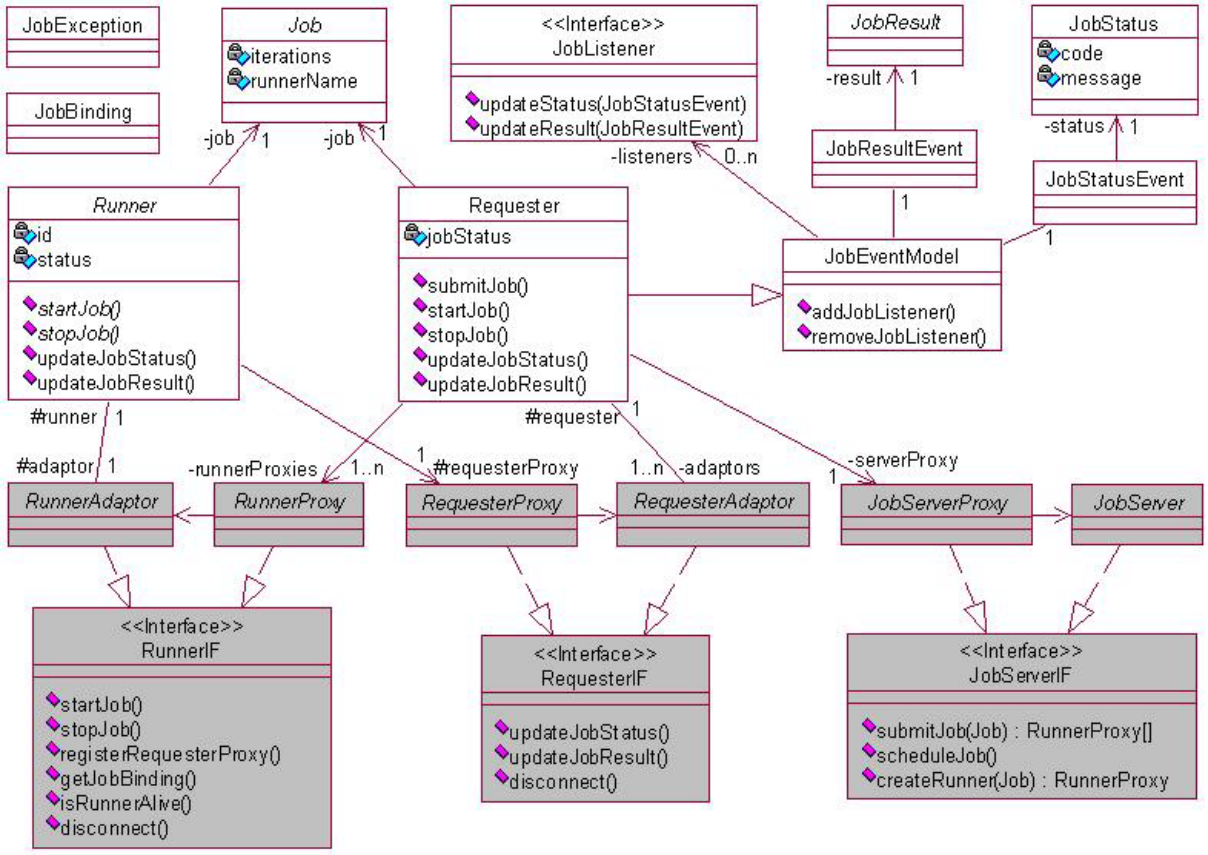

Figure 2: Framework Class Diagram

sense that it can be used to implement any kind of distributed job-oriented system, of which distributed supply chain simulation is simply a special case. The classes and interfaces in the framework can be classified as *job-layer* (unfilled rectangles) or *communication-layer* (color-filled rectangles) entities. The job-layer entities constitute a high-level API for developers of distributed applications; they represent an abstraction at the job level and facilitate distributed job submission, control, and results collection. The communication-layer entities represent an abstraction at the network communication level; they are used by the job-layer entities and are transparent to application developers.

Communication-layer entities are either Java interfaces or abstract classes, and therefore need system developers to furnish concrete implementations based on different network protocols. The framework comes with a default implementation in Java RMI, and bindings to other protocols such as TCP/IP and Globus are possible as well.

There are three job-layer abstract classes that application developers must implement. `Job` represents the structure of a class of tasks that need to be executed repeatedly for a given number of iterations; developers specialize it to model domain tasks of interest to a particular application. Similarly, subclasses of `JobResult` need to be defined to encapsulate domain specific job running results. The abstract class `Runner` represents the operation required to run a certain type of jobs. Separation of the structure of a job and the operation needed to execute it follows the *Visitor* design pattern (Gamma et al. 2000) which lets you define a new operation without changing the classes of the elements on which it operates. For any concrete `Job` subclass, an application developer must provide at least one `Runner` subclass (visitor).

`Requester` is a concrete class mediating between the user application and the communication layer. It is the most prominent class in the framework because it is where the distributed job semantics becomes visible to the user application. A `Requester` object needs to talk to remote job servers and `Runner` instances in the servers to fulfill its functionality, therefore we have adopted the *Proxy* design pattern (Gamma et al. 2000) to introduce three remote proxy classes, `JobServerProxy`, `RunnerProxy` and `RequesterProxy`, which enable bi-directional communication between `Requester` and `JobServer` and between `Requester` and `Runner`. The original Proxy pattern would have required `Runner` and `Requester` to implement the same interface as `RunnerProxy` and `RequesterProxy` respectively, which would have coupled these application-layer classes with the communication layer. To avoid this, we have used the *Adapter* design pattern (Gamma et al. 2000) to introduce two adaptors, `RunnerAdaptor` and `RequesterAdaptor`, which implement `RunnerIF` and `RequesterIF` respectively and forward incoming calls from remote proxies into corresponding job-layer adaptees.

Now let us take a detailed look at how `Requester` interacts with other entities in the framework to effect job submission. Once this becomes clear, job control and status/result callback mechanism will fall into place. Job submission involves a three-way handshake: `Requester` → Gatekeeper, Gatekeeper → `Requester`, and `Requester` → `Runners`, which are illustrated in Figure 3, 4 and 5 respectively.

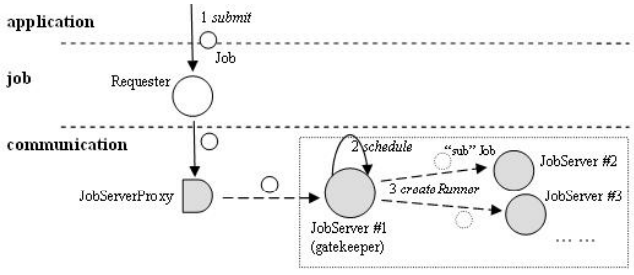

Figure 3: Job Submission: Requester → Gatekeeper

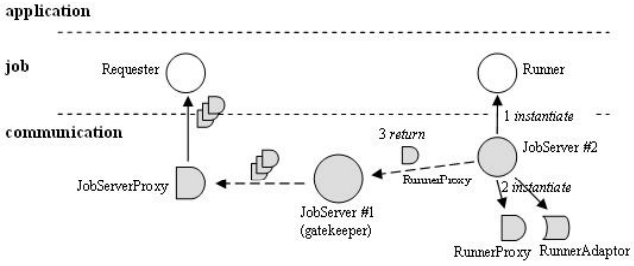

Figure 4: Job Submission: Gatekeeper → Requester

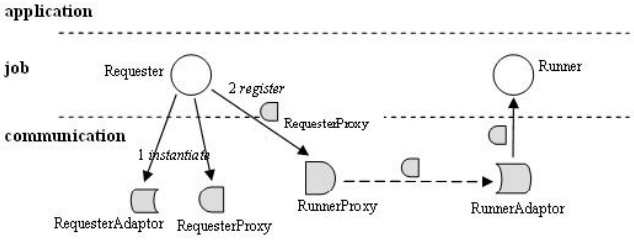

Figure 5: Job Submission: Requester → Runners

`Requester` → Gatekeeper. The `Requester` accepts a job from the user application (client) and forwards it to the Gatekeeper (a `JobServer` in the server pool that is configured to accept and schedule user jobs); the Gatekeeper schedules the job, i.e., determines how to split the job to a number of sub jobs based on available resources in the server pool, and sends each sub job to the server where it is scheduled to run.

Gatekeeper → `Requester`. Upon receipt of a sub job from the Gatekeeper, a job server instantiates the corresponding `Runner` subclass (specifically, the job server retrieves the fully-qualified `Runner` class name from the submitted `Job` object and dynamically loads the `Runner` subclass from an HTTP class server); the server then instantiates a `Runner`-

`Proxy-RunnerAdaptor` pair for the `Runner`, and returns the `RunnerProxy` object to the Gatekeeper; finally, the Gatekeeper returns all the `RunnerProxy` objects it collects from job servers to the `Requester`.

`Requester` → `Runners`. With remote proxies (`RunnerProxy` objects) at hand, the `Requester` can send job control commands to remote `Runners`. Now it needs to establish the connection in the other direction to allow `Runners` to send back job running statuses and results. To accomplish this, the Requester instantiates a `Requester-Proxy-RequesterAdaptor` pair for each `RunnerProxy` received, and registers the `RequesterProxy` with the corresponding remote `Runner`.

Application developers don't need to understand the communication-layer details, yet they have to know how to configure the framework to use a certain implementation. There are two configuration files serving that purpose, one for the job layer and the other is communication-layer implementation specific. The job-layer configuration file is used by the client to specify `Job-Runner` mapping, and to plug in a communication-layer implementation by specifying a factory class (*Abstract Factory* design pattern in Gamma et al. 2000); it is used by the job server to find out about the class server location. Here is a snippet of the properties file:

```
#
# Client-side config: Job-Runner mapping
#   format:
#       job_alias=runner_class_name
#
job.foo=com.ibm.job.runner.FooRunner
job.scms=com.ibm.scms.job.SimuRunner

#
# Client-side config: plug in a communication
# layer implementation via Abstract Factory
# design pattern
#   creates:
#       JobServerProxy
#       RequesterProxy
#       RequesterAdaptor
#
factory=com.ibm.job.factory.RMIFactory
#factory=com.ibm.job.factory.SocketFactory
#factory=com.ibm.job.factory.GlobusFactory

#
# Server-side config: class server URL
#
classserver=http://huashan:9099/
```

The other file is used by the implementation specific `JobServerProxy` to find out about gatekeeper location information. Here is a snippet of the properties file that comes with the Java RMI implementation:

```
#
# RMIJobServerProxy config:
#   Gatekeeper location
#
gatekeeper.url=rmi://huashan:1099
gatekeeper.name=JobServer
```

Apparently, this layered design has managed to address the second concern in Section 2 and offers the expected runtime configurability. What needs to be pointed out is an additional benefit that enables developer role separation, i.e., an application developer can code against the job-layer API and focus on his domain issues, while a system developer with expertise in distributed system can concentrate on providing high-quality communication-layer implementations.

Something also worthy of mention is the built-in fault-tolerance in `Requester` and `Runner` classes. During a job run, they keep monitoring the availability of the party at the other end of the communication link, and in the case of a link failure, the connection will eventually be shut down on both ends, and any allocated resources will be gracefully released.

4 SCMS ENHANCEMENT FOR DISTRIBUTED SIMULATION

In the original SCMS tool, when a user requests the simulation of a constructed supply chain model (a `SimuModel` object), the application will create a simulation engine (a `SimuEngine` object), assign the model to the engine, and start it up; events generated from the engine (`SimuEvent` objects) are fired to the simulation event listener interface `SimuEventListener` implemented in the main JFrame window `SCMSMainFrm`. This is fine when running `SimuEngine` locally is the only option for the tool to perform simulation. However, with distributed simulation as another possibility, that is no longer the case.

The *Strategy* design pattern is perfect for such situation where a family of algorithms are interchangeable and can vary independently from clients that use it (Gamma et al. 2000). In our case, a base strategy class `Simulator` and two concrete strategy classes `LocalSimulator` and `DistributedSimulator` have been introduced accordingly (Figure 6). `LocalSimulator` is nothing but an adapter of `SimuEngine` based on object composition; `DistributedSimulator` is more complicated and will be discussed later. Now, all that needs to be changed in the SCMS tool is the way a simulation run is initiated, i.e., instead of directly starting a `SimuEngine` instance, the tool will use one of the two `Simulator` strategies. In either case, all the simulation-related presentation logics, such as animation and dynamic reporting, remain unchanged, which has addressed the first concern outlined in Section 2.

`DistributedSimulator` is implemented by using the generic framework and simulation domain specific subclasses of `Job`, `Runner` and `JobResult` (Figure 7). `SimuJob` and `SimuJobResult` are simply wrapper classes of `SimuModel` and `SimuEvent` respectively. `SimuRunner` has been designed to use a `SimuEngine` object as adaptee, in a similar way as `SimuEngine` is adapted by

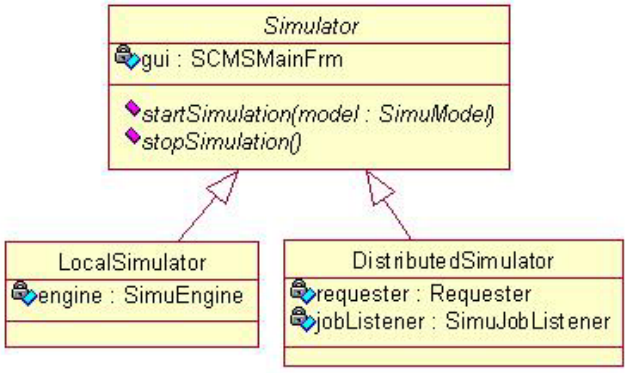

Figure 6: Simulator Strategy

`LocalSimulator`. However, there is a major difference: `LocalSimulator` runs in the same JVM as `SCMSMainFrm`, while `SimuRunners` generally don't. As a result, `SimuRunner` can not send simulation events (fired by the adapted `SimuEngine` object) directly to the SCMS tool, as `LocalSimulator` does; instead, it has to intercept those simulation events with an inner class that implements the `SimuEventListener` interface, and wraps such an event in a `SimJobResult` object which is carried as "payload" in a `JobResultEvent` object. This job-layer event travels the communication layer and gets back to the original `Requester`. The `SimuJobListener` registered with the `Requester` then retrieves the wrapped simulation event and forwards it to the GUI component.

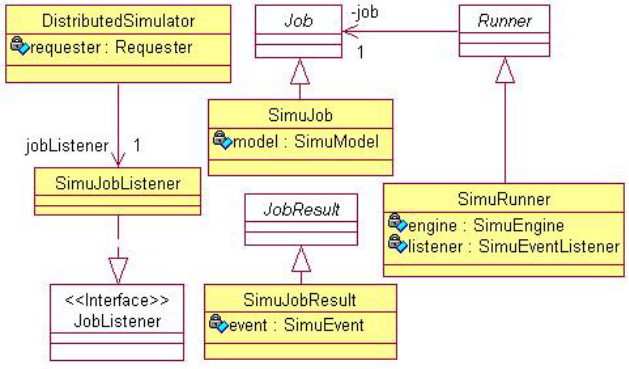

Figure 7: Simulation Job Class Diagram

Using a framework and proper design patterns has made it fairly easy to add a complex feature to the existing tool in a non-intrusive way.

5 BENCHMARKING

In this section, we will look at what kind of speedup distributed simulation can achieve over local sequential simulation. We developed a simple simulation model for benchmarking purpose that has one Customer and one Manufacturer. The Manufacturer can produce one type of end product that is assembled from two part types. The Customer provides 12-month demand forecast to the Manufacturer. Four computers were used and their relevant configuration information is listed in Table 1. The benchmarking result is presented in Table 2.

Table 1: Computer Configurations

	CPU	Memory	OS
Computer 1	Pentium 4 3.06GHz	512 MB	Windows XP
Computer 2	Pentium 4 2.2GHz	512 MB	Windows XP
Computer 3	2 x Pentium III 500MHz	768 MB	RedHat Linux
Computer 4	Pentium III 930MHz	768 MB	Windows 2000

Table 2: Benchmarking Result

Iterations	**Time (seconds)**		
	Local Simulation		**Distributed Simulation**
	Computer 1	**Computer 4**	
20	2	5	5
50	7	19	6
100	10	27	10
200	20	51	20
500	50	128	45
1000	97	252	81

The following observation can be made from Table 2: when the computer used for local simulation (Computer 1) has above-average computing power (compared with all the machines involved in the distributed simulation), the local simulation performance is comparable with that of the distributed simulation up to 500 iterations; when the machine used for local simulation has below-average computing power (Computer 4), distributed simulation exhibits a significant performance speedup. Considering the fact that the current RMI job server has implemented a very primitive job scheduling algorithm, which is to distribute iterations evenly across available computers, our benchmarking result is easy to interpret: the performance of distributed simulation is largely determined by the slowest computer in the server pool (Computer 4 in our case).

Two quick conclusions can be drawn:

- When the servers are heterogeneous, an intelligent scheduling algorithm should be used to balance job loads among fast and slow machines to minimize the overall running time.
- If we have a supercomputer that has far more superior computing power than that of the server pool average, then, depending on the size of the server pool and the amount of distributed communication

overhead involved, we may be better off by running simulation jobs locally on the supercomputer.

6 INTEGRATION WITH IBM MATHGRID DESKTOP

One weakness of our generic job running framework is lack of deployment support. For example, in order to conduct the benchmarking discussed in Section 5, we had to manually deploy the framework and start the RMI job server on each computer involved, which can become a maintenance headache in real-world application.

To solve this problem, we have been seeking to integrate the enhanced SCMS tool with IBM MathGrid Desktop (IMGD). IMGD is a GUI toolkit developed in the Math Science Department of the Watson Research Center to simplify the development and distribution of grid-enabled applications (both legacy and newly written). It is built on the Java Community Grid (CoG) Kit and interacts with a back-end grid built on Globus Toolkit 2.0. To accomplish its goal, IMGD has provided:

- A visual desktop with which developers can advertise, distribute, install, and demonstrate their application
- A client, containing the desktop, which can hold any Swing based application GUI
- Server technology that allows functionality from any server based shared library to be accessed from the client

By wrapping our job framework according to IMGD's server-side specification , we can use IMGD to deploy the framework to MathGrid computers with just a few mouse clicks. In addition, we can run the Swing based SCMS GUI from the same environment and submit distributed supply chain simulation requests to those MathGrid computers.

7 FUTURE WORK

While continuing the effort to integrate with IMGD, we will concentrate our future work on improving the generic job running framework from two aspects:

7.1 Better Fault Tolerance

Fault-tolerance currently built into the framework aims at releasing resources and restoring communication-layer integrity upon link failures; however, it does not recover the lost sub job(s) due to the same failures. We would like to enhance the `Requester` implementation to enable automatic job re-scheduling when a link failure occurs. Another potential issue is the single point of failure at the Gatekeeper, which might be addressed in `JobServer` implementations with a distributed leader election protocol.

7.2 Communication Layer Binding to Globus

The communication layer has only one default implementation as of now, we would like to build another one using Globus Toolkit 3.0. Applications using this protocol binding will be able to take advantage of standard Globus features such as Grid Security Infrastructure and Grid Information Services. For instance, Grid Information Services will make it easy to incorporate intelligent job scheduling into the `JobServer` implementation, while it is quite difficult to do so with Java RMI which doesn't offer a similar resources information service.

REFERENCES

Bagchi, S., S. J. Buckley, M. Ettl, and G. Y. Lin. 1998. Experience Using the IBM Supply Chain Simulator. In *Proceedings of the 1998 Winter Simulation Conference*, ed. J. M. Charnes, D. M. Morrice, D. T. Brunner, and J. J. Swain, 65–72. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers.

Cao, H. et al. 2003. A Simulation-based Tool for Inventory Analysis in a Server Computer Manufacturing Environment. In *Proceedings of the 2003 Winter Simulation Conference*, ed. S. Chick, P. J. Sánchez, D. Ferrin, and D. J. Morrice.

Fujimoto, R. M. 1999a. *Parallel and Distributed Simulation Systems (Wiley series on parallel and distributed computing)*. John Wiley & Sons, New York, NY.

Gamma, E. et al. 2000. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Lin, Y. G. et al. 2002. The New Frontier: Sense and Respond System for Value Chain Optimization. *ORMS Today*: April 2002.

Mascarenhas, E., V. Rego, and J. Sang. 1995. DISplay: A System for Visual-Interaction in Distributed Simulations. In *Proceedings of the 1995 Winter Simulation Conference*, ed. C. Alexopoulos, K. Kang, W. R. Lilegdon, and D. Goldsman, 698-705.

Sudra, R., S. J.E. Taylor, and T. Janahan. 2000. Distributed Supply Chain Simulation in GRIDS. In *Proceedings of the 2000 Winter Simulation Conference*, ed. J. A. Joines, R. R. Barton, K. Kang, and P. A. Fishwick, 356-361.

Sun Microsystems, Inc. 1999. Delegation Event Model. <http://java.sun.com/productsjdk/1.1/docs/guide/awt/designspec/events.html>.

AUTHOR BIOGRAPHIES

HAIFENG XI is an advisory software engineer at the IBM T. J. Watson Research Center. He received the M.S. degree in Electrical and Computer Engineering from the University of Maryland. His current interests include Web ap-

plication architecture, business integration, grid computing, and supply chain simulation. He can be reached by e-mail at <haifengx@us.ibm.com>

HENG CAO is a staff software engineer at the IBM T. J. Watson Research Center. She received the M.S. degree in Robotics from Carnegie Mellon University. In addition to modeling, analysis and simulation of supply chain systems, her interests include artificial intelligence, data mining and business integration. She can be reached by e-mail at <hengcao@us.ibm.com>

LEONARD BERMAN is a research staff member at the IBM T. J. Watson Research Center. He received his Ph.D. in Computer Science from Cornell University in 1977 and has been with IBM since then. His current research is in the area of heterogeneous computing. He can be reached by e-mail at <namreb@us.ibm.com>

DAVID JENSEN is a senior manager at the IBM T. J. Watson Research Center. He received his Ph.D. in Operations Research and Industrial Engineering from Cornell University, and joined IBM in 1987. His interests include linear and nonlinear optimization and Web based deployment of analytical methods. He can be reached by e-mail at <davjen@us.ibm.com>