Autonomic Web-Based Simulation

Yingping Huang and Gregory Madey
Department of Computer Science & Engineering
University of Notre Dame
Notre Dame, IN 46556
{yhuang3, gmadey}@nd.edu

Abstract

Many scientific simulations are large programs which despite careful debugging and testing will probably contain errors when deployed to the Web for use. Based on the assumption that such scientific simulations do contain errors and the underlying computing systems do fail due to hardware or software errors, we investigate and explore robust methods for building reliable systems to support web-based scientific simulations with the presence of such errors. In this paper, we present a framework to build autonomic web-based simulation (AWS). AWS achieves the following features presented in the Vision of Autonomic Computing [15]: self-configuring, self-optimizing, self-healing and self-protecting.

1. Introduction

Web-based simulation is the integration of the Web with the field of simulation and has been growing during the past few years. Developers of large-scale web-based scientific simulations have experienced increased complexity in their software systems due to the complex integration of different software components. Web-based simulations need to be deployed through computing systems. Such computing systems often consist of storage devices (such as local hard drives for simulation data files), databases (data transformed from simulation data files for better data analysis), web servers (user interface for simulation input and output) and simulation servers (on which simulations run). These components often have workflow dependencies and interact among themselves. Managing such systems involves configuring these individual components so that the overall system goals (such as completing a simulation in a specified time interval) can be achieved. In his Turing award speech, "What next? - A dozen IT research goals", J. Gray (Microsoft Research) has emphasized the need for self-manageable systems in which the administrator sets up system goals and high level policies, while the system by itself decides how they can be achieved [9].

In 2001, IBM launched the Autonomic Computing initiative, a vision striving for system self-management. The idea of autonomic computing originates from the human autonomic nervous system. This system tells the heart how fast to beat, checks blood's sugar and oxygen level, etc. All these are done automatically without concious human involvement. That's precisely what is needed to initiate the next era of computing: autonomic computing, also known as self-manageable systems. It's a paradox that to achieve such autonomic features, the system must become even more complex by embedding the complexity into the system infrastructure itself, so that the system management can be automated.

Autonomic Computing includes the following four features: self-configuring, self-healing, self-optimizing and self-protecting [15]. Self-configuring involves automated configuration of components and systems following high-level policies. Self-healing can be accomplished by automatically detecting, diagnosing and repairing localized software and hardware problems. Self-optimizing involves self-tuning of service parameters. Self-protecting means system automatically defends against malicious attacks or cascading failure. It uses early warnings to anticipate and prevent system wide failure.

In this paper, we investigate and explore current technologies to design a robust framework for autonomic web-based simulations. The rest of the paper is organized as follows: Section 2 presents background work related to autonomic web-based simulations (AWS); Section 3 discusses the requirement and design of AWS; Section 4 proposes a robust infrastructure to support AWS; Section 5 concludes the paper with a discussion of future work.

2. Background and Related Work

The architecture of an autonomic computing system is a hierarchical collection of components called autonomic el-

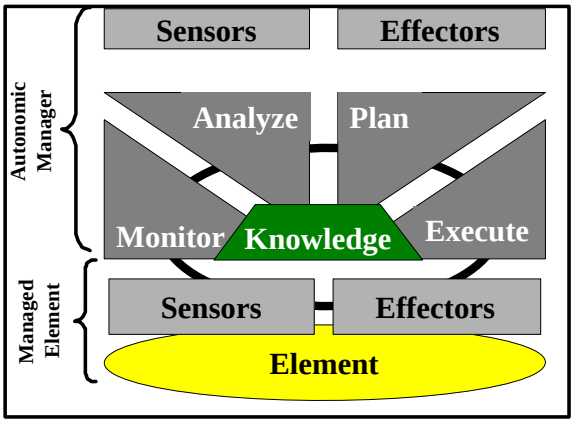

Figure 1. Autonomic Element (Adopted from [15])

ements, which encapsulates autonomic managers and managed elements, as shown in Figure 1. A managed element can be a hardware resource, such as a CPU, or an application software, such as a database server, or an entire system. An autonomic manager is an agent managing its internal behavior and relationships with other agents according to policies. System self-management will arise from both interactions among agents and from agents' internal self-management.

IBM has conducted research towards autonomic computing, across all levels of computer management, from hardware to software and some of the work has been published on the IBM System Journal. On the hardware level, systems are dynamically reconfigurable and upgradable, enabling the movement of hardware resources (such as processors, memory and I/O slots) without requiring reboots [14]. On the operating system level, an active operating system allows monitoring code, diagnostic code and function implementations to be dynamically inserted and removed in the system (often called "hot-swapping") [3]. On the application level, database self-validate optimizers [19] and web servers are dynamically reconfigured by agents to adapt service performance [7]. In [19], an autonomic query optimizer automatically self-validates its model to repair incorrect statistics or cardinality estimates. By monitoring queries as they execute, the autonomic query optimizer compares the optimizer's estimates with actual cardinality at each step in a *query explain plan*, and computes adjustments to its estimates that may be used during future optimizations of similar queries. In [7], the authors use an *AutoTune* agent framework under the *Agent Building and Learning Environment (ABLE)* to automatically tune application-level parameters *MaxClient* and *KeepAlive* to control CPU and memory utilization in an *Apache Web Server*.

From the above examples, we see that some of the ideas from autonomic computing have already been implemented in practice. In this paper, we seek to put some of the autonomic computing ideas into the field of web-based simulation. First, we review some of the current technologies that favor autonomic web-based simulation. Then we survey some of the recent web-based simulations.

2.1. Technologies That Favor AWS

Many recent advances in IT are in favor of autonomic web-based simulation. These technologies include grid computing and J2SE (Java 2 Standard Edition) 1.5 (sometimes J2SE 5.0).

Grid computing is distributed computing whose goal is to create the illusion of a simple yet large and powerful self managing virtual computer out of a large collection of connected heterogeneous systems sharing various combinations of resources [8, 13]. In most organizations, there are large amounts of underutilized computing resources. Most desktop machines are busy less than five percent of the time. Even server machines can often be relatively idle. Grid computing provides a framework for exploiting these underutilized resources and thus has the possibility of substantially increase the efficiency of resource utilization. Also, machines may have enormous unused disk drive capacity. Grid computing, more specifically, a *data grid*, can be used to aggregate this unused storage into a large virtual data store, configured to achieve improved efficiency and reliability. Another functionality of grid computing is to better balance resource utilization. For example, grid-enabled applications can be moved to underutilized machines during peak times. In general, a grid can provide a consistent way to balance the loads on CPUs, storages, and many other kinds of resources.

Grid computing provides a natural platform for grid-enabled applications, some of which can present autonomic behavior [16, 2]. Therefore, autonomic web-based simulation can benefit from grid computing.

Another important advance in information technology is the release of J2SE 1.5. The greatest enhancement of J2SE 1.5 from previous editions is monitoring and manageability. Monitoring and manageability is a key component of RAS (Reliability, Availability, Serviceability) in the Java platform. The release of J2SE 1.5 introduces comprehensive monitoring and management support for the Java platform: instrumentation to observe the Java virtual machine (JVM), Java Management Extensions (JMX) framework and remote access protocols. The JVM Monitoring and Management API specifies a set of instrumentations to allow a running JVM and underlying operating system to be monitored. This information is accessed through JMX MBeans and can be accessed locally within the Java address space or remotely using the JMX remote interface.

One of the monitoring features is the low memory detector. JMX MBeans can notify registered listeners when a low memory threshold is crossed. The monitoring and manageability enables self-awareness, the basis of autonomic computing, hence J2SE 1.5 is a contributing platform to develop autonomic web-based simulations. Furthermore, the possibility to efficiently run separate JVMs will provide a basis for isolation of Java processes and achieve self-healing by fast restarting failed or paused processes.

2.2. Web-based Simulations

With the emergence of WWW, many web-based simulations and simulation platforms have been developed, such as [11, 22, 10], to name a few. In [11], the authors present a self-manageable infrastructure to host scientific simulations. The infrastructure integrates web servers, database servers, reports servers, data warehouses and data mining tools to provide a thorough web-based simulation and data analysis environment. In [22], the development of an integrated extensible web-based simulation environment called Computational Science and Engineering Online (CSEO) is presented using some tools from information technology. In [10], the authors report that a metadata tools system and a data services system are undergoing development and integration at Sandia National Laboratories, to provide web-based access to high-performance computing clusters and its associated simulation data.

Some work focusing on simulation data analysis and understanding can be found in [1]. The authors describe scientific simulation data, its characteristic and the way scientists generate and use the data. Then they compare and contrast simulation data to data streams. After that, the authors present a tool called AQSim (Ad-hoc Queries for Simulation Data) system to analyze simulation data.

In this paper, we focus on exploring robust methods to design reliable web-based simulation in a naturally unreliable environment.

3. AWS Requirements

A basic requirement to build autonomic web-based simulation is that a simulation is able to restart from near the point of failure. In practice, many scientific simulations are long-running simulations. Such simulations can run in hours, days, or even months. It's costly to restart the simulation from scratch if it dies prematurely. To prevent restarting from the beginning, a mechanism called checkpointing is used to save the state of the simulation periodically [18, 21]. The papers by Chandy [4] and Nicola [20] gave excellent overview of checkpointing and recovery strategies in the literature.

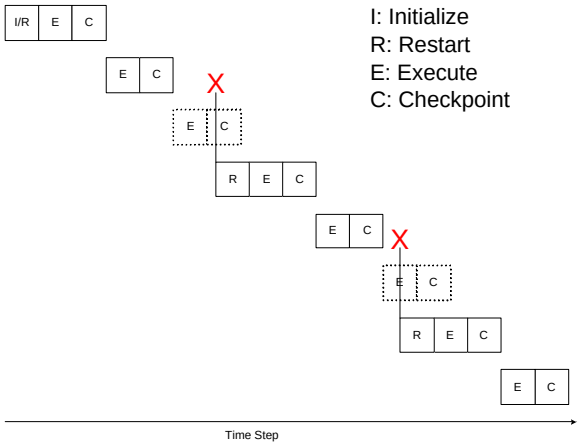

Figure 2. Checkpoint and Restart

Long-running scientific simulations would benefit from this simple checkpointing mechanism that provides automatic restart or recovery in response to faults and failures, and enables dynamic load balancing and improved resource utilization through simulation migration [11, 18]. However it is usually not a trivial task to optimally determine the "interval" between contiguous checkpoints. Excessive checkpointing would result in performance degradation, while deficient checkpointing would incur an expensive recovery overhead. Therefore, a trade-off must be made to determine the checkpoint interval optimally.

The simulation developers control how to choose checkpoint data and how to use the checkpoint data to restart a simulation in the case of failure. For many time-stepped simulations, we can define **checkpoint interval** to be the number of timesteps between two consecutive checkpoints. We show how to determine the checkpoint interval optimally for time-stepped simulations.

3.1. Checkpoint Interval Determination

Checkpoint is the solution for long-running simulations and has been used for system restarting[6]. As shown in Figure 2, the lifecycle of a simulation consists of a series of checkpoints and possibly restarts. Even when a simulation completes, its internal state is checkpointed so that it can resume more time steps when requested by users.

As in Figure 2, each EC or REC is called an execution. Let T be the total (clock) time for a simulation. Let $T_{restart}$ be the restart/initialization time for an execution. Let $T_{checkpoint}$ be the checkpoint time for an execution. Let $T_{execution}$ be the execution time for an execution. Let T_{redo} be the redo time for a crashed execution. Then the total execution time T can be calculated as follows:

$$T = \sum T_{execution} + \sum T_{redo}$$

$$+ \quad \sum T_{restart} + \sum T_{checkpoint}$$

Let $Rel(t)$ denote the reliability (= 1−(probability that a simulation crash at time t)) of a simulation at time t. Let M denote the average time of simulations before failure occurs. Like in many systems [6], to simplify the analysis, we may assume the reliability of a simulation at (wall clock) time t, i.e., the probability that the simulation runs successfully at or before t, is $Rel(t) = e^{-\frac{t}{M}}$.

Let N be the total number of requested time steps for a simulation (N is often a user input, measured in simulation time steps). Let x be the checkpoint interval to be determined. To simply the problem of determining checkpoint interval, we make the following assumptions:

- Execution time of a time step is a constant E.
- Restart time $T_{restart} = R$, where R is a constant.
- Checkpoint time $T_{checkpoint} = C$, where C is a constant.
- $T_{execution} = xE$.
- $T_{redo} = \frac{1}{2}(xE + C)$, i.e., the expected redo time is half the time of an execution. In fact, the distribution of failures occuring at (wall clock) time t after the last checkpoint is $d(t) = \frac{e^{\frac{t}{M}}}{M \cdot \left(1 - e^{-\frac{xE+C}{M}}\right)}$. Thus the expected point of failure between 0 and $(xE + C)$ is $\int_0^{xE+C} t \cdot d(t)dt = M + \frac{xE+C}{1-e^{\frac{xE+C}{M}}} \rightarrow \frac{1}{2}(xE + C)$ as $\frac{E}{M} \rightarrow 0$.

Then we have the following:

- The probability that an execution complete successfully is $Rel(xE + C)$.
- The number of (successful) executions is $\frac{N}{x}$.
- The expected number of executions (both successful and crashed) is $\frac{N}{x \cdot Rel(xE+C)}$.
- The number of crashed executions is therefore $\frac{N}{x}(\frac{1}{Rel(xE+C)} - 1)$. And thus,
- $\sum T_{execution} = N \cdot E$.
- $\sum T_{redo} = \frac{N}{x}(\frac{1}{Rel(xE+C)} - 1) \cdot \frac{1}{2}(xE + C)$.
- $\sum T_{checkpoint} = \frac{N}{x} \cdot C$.
- $\sum T_{restart} = \frac{N}{x}(\frac{1}{Rel(xE+C)} - 1) \cdot R$.

Therefore, the total execution time (wall clock time) of a simulation is

$$\begin{aligned} T(x) &= N \cdot E + \frac{N}{x} \cdot C \\ &+ \frac{N}{x}(\frac{1}{Rel(xE + C)} - 1) \cdot (\frac{1}{2}xE + \frac{1}{2}C + R) \end{aligned}$$

Then,

$$\begin{aligned} T(x) &= N \cdot E + \frac{N}{x} \cdot C \\ &+ \frac{N}{x}(e^{\frac{xE+C}{M}} - 1) \cdot (\frac{1}{2}xE + \frac{1}{2}C + R) \\ &\geq N \cdot E + \frac{N}{x} \cdot C \\ &+ \frac{N}{x} \cdot \frac{xE + C}{M} \cdot (\frac{1}{2}xE + \frac{1}{2}C + R) \\ &= NE + \frac{NE}{M}(\frac{C}{2} + R) + \frac{NCE}{2M} \\ &+ \frac{NC + \frac{NC}{M}(\frac{C}{2} + R)}{x} + \frac{NE^2}{2M} \cdot x \\ &\geq NE + \frac{NE}{M}(\frac{C}{2} + R) + \frac{NCE}{2M} \\ &+ 2 \cdot \sqrt{(NC + \frac{NC}{M}(\frac{C}{2} + R)) \cdot \frac{NE^2}{2M}} \\ &= \frac{NE}{M} \cdot (M + C + R + \sqrt{2MC + C^2 + 2CR}) \end{aligned}$$

The "=" of the second inequality above holds, if and only if

$$x = \frac{\sqrt{2MC + C^2 + 2CR}}{E} \approx \frac{\sqrt{2MC}}{E},$$

since $C, E, R \ll M$.

We determined the checkpoint interval of a simulation is approximately $\frac{\sqrt{2MC}}{E}$ time steps and does not depend on the restart time R. The above calculated total execution time T can be used to predict the execution time of a simulation. The checkpoint interval x can also be written as $x = \sqrt{2 \cdot \frac{M}{E} \cdot \frac{C}{E}}$. Therefore, to determine x, it suffices to determine $\frac{M}{E}$ and $\frac{C}{E}$. But $\frac{M}{E}$ is the average number of time steps before crashing, and it can be determined by running sufficiently many (20, for example) simulations without checkpoint. $\frac{C}{E}$, the ratio of checkpoint time to execution time of one time step, is simulation program dependent.

3.2. Proactive Failure Detection using J2SE 1.5

Another (optional) requirement to achieve autonomic web-based simulation is to be able to detect failure protectively. It is an optional requirement because the companion infrastructure, which is presented in Section 4, provides utilities for autonomic failure detection.

Based on our experience, one of the major causes of simulation crashes is due to low memory. A useful feature of J2SE 1.5 is the low memory detector. When a low memory threshold is set either manually or by learning, JMX MBeans can notify registered listeners when it is crossed. Once such a threshold is crossed, the simulation can checkpoint itself and terminate gracefully. The simulation dispatcher (which is discussed in Section 4) can restart the sim-

```
import java.lang.management.*;
import java.util.*;

public class LowMemDetector{
   public boolean isMemoryLow(){
      List<MemoryPoolMXBean> memoryPoolMXBeans =
          ManagementFactory.getMemoryPoolMXBeans();
      for (MemoryPoolMXBean memoryPoolMXBean: memoryPoolMXBeans){
          float usedMemory=memoryPoolMXBean.getUsage().getUsed();
          float totalMemory=memoryPoolMXBean.getUsage().getMax();
          float freeMemoryPct=(totalMemory - usedMemory) / totalMemory;
          if( freeMemory < threshold)
            return true;
      }
      return false;
   }
}
```

Figure 3. Low Memory Detector Using the Management API

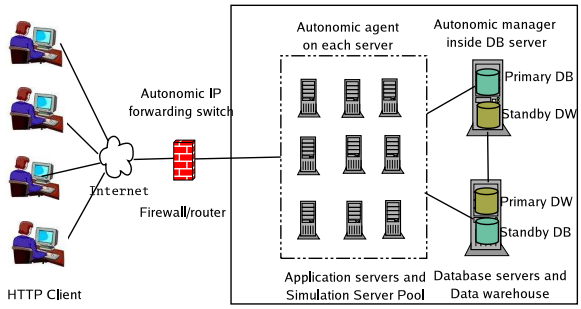

Figure 4. Self-Manageable Infrastructure for AWS

ulation on an appropriate simulation server. Figure 3 shows an example of low memory detector. The J2SE 1.5 Management and Monitoring API can be used to monitor the Java Virtual Machine and underlying operating system.

Web-based simulations must be deployed through some computing system. To achieve autonomic web-based simulations, the computing system must be a self-manageable system, i.e., the system must be self-configuring, self-healing, self-optimizing and self-protecting.

4. Autonomic Infrastructure for AWS

Figure 4 shows a simple multi-tiered architecture of the computing grid, where Primary DB is a general purpose database server and Primary DW is a data warehouse. The firewall/router forwards incoming network traffic to appropriate servers. The server pool consists of application servers and simulation servers. The application servers host the front-end web applications including visualization tools, while the simulation servers host the actual simulations.

Autonomic agents are installed on these servers to

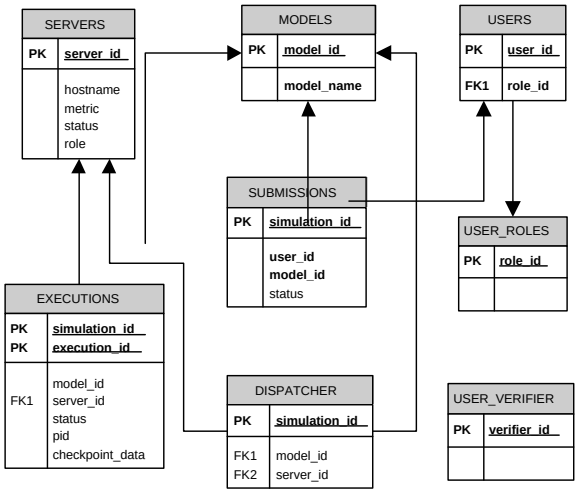

Figure 5. Core Data Model for AWS

achieve self-manageability of the infrastructure. For example, when a simulation server runs short of storage, the autonomic agent purges old local data files using the unix ”find” command:

```
find /home/nom/data -atime +7 \
 -exec rm -rf {} \;
```

The autonomic agents are data driven in the sense that they contact the database server frequently to retrieve appropriate information to direct their behavior. Therefore, an efficient data model is mandatory to build autonomic agents. Meanwhile, an autonomic manager is residing inside the database to monitor all autonomic agents in the server pool. Figure 5 shows the core data model to support autonomic web-based simulations. Because of space limitations, only a few columns for each table are presented in the model. We explain the usage of the ER diagram when discussing the self-management features.

4.1. Self-configuring

Self-configuring involves autonomic incorporation of new components and autonomic component adjustment to new conditions. The autonomic agents can self-configure the server pool, the database server and the data warehouse in the following ways:

- Autonomic discovery of new servers - a new server can be discovered if the autonomic agent is running on it. More precisely, the autonomic automatically inserts a record into the SERVERS table and periodically uploads the underlying operating system properties such as load average to corresponding columns of the SERVERS table. The operating system proper-

ties will be used by the dispatcher to distribute simulation jobs to appropriate simulation servers.

- Autonomic resize of the server pool - each autonomic agent frequently makes connections to the database server. It is not necessary to keep the autonomic agents running when the system is idle. The autonomic manager automatically shuts down and restarts autonomic agents based on the workload of the system. The load of the system is recorded in the SUBMISSIONS table, which is monitored by the autonomic manager.

- Autonomic configuration of the firewall/router - the computing grid is a private subnet. The front end web application and visualization is installed on some of the servers in the server pool. Typically, only one server needs to be the application server at a time. The firewall forwards incoming HTTP traffic to the application server. In the case that the application server is down for whatever reason, the autonomic agent automatically starts another application server and the firewall forwards HTTP requests to the new application server by running an ”iptables” script.

- Autonomic configuration of the application servers and simulation servers in the server pool - autonomic agents automatically install the most recent version of web applications to application server and install the most recent version of simulation program on simulation servers. The autonomic agents also mount NFS if local storage is crossing a certain threshold. The SERVERS table in the data model records the status (UP or DOWN) and the role (APPLICATION SERVER or SIMULATION SERVER), which are updated frequently by the autonomic agents.

- Autonomic configuration of the database server and the data warehouse - one of the basic requirement for autonomic computing is self-awareness. The database server records all information about the computing grid. Such information includes the database capacity (such as tablespace quota and maximal number of database connections) and other server capacity (such as total memory and disk quota). This information is accessed by both the autonomic manager inside the database and the autonomic agents on servers in the server pool. The data warehouse initialization parameters are automatically configured to support data warehousing operations. For data warehousing technologies, such as dimension tables, fact tables, star schema and snowflake schema, please refer to Inmon [12] and Kimball [17].

```
#!/bin/sh

URL=username/password@service
SERVER=server
STATUS=`wget -q -s -O - http://${SERVER}:8888 | head -1 | cut -d' ' -f2`
if [ $STATUS != "200" ]; then
sqlplus $URL >/dev/null <<EOF
  update servers set status='DOWN' where hostname = '${SERVER}';
  commit;
  exit
EOF
```

Figure 6. Autonomic Agent Pings an Application Server

4.2. Self-healing

Some degree of redundancy is required to achieve self-healing. There is a hot standby data warehouse for the primary data warehouse and a standby database for the primary database. The database and data warehouse are designed on two physical hosts as in Figure 4. The server pool ensures that when an application server or simulation is down for any reason, other servers can pick up its tasks and resume the service directed by the autonomic manager and autonomic agents. Next we show how this is implemented.

4.2.1. Autonomic Failure Detection The non-functional period of a failed service or node is comprised of two distinct phases: periods when a system is unaware of a failure (failure-detection latency) and periods when a system attempts to recover (failure-recovery latency) [5]. It is vital to develop autonomic failure detectors for the purpose of self-healing.

4.2.2. Self-healing Application Servers There are two monitors to monitor an application server. A local monitor, part of the autonomic agent, monitors the execution status of an application server. If responses from the application server are not received in a timely fashion, the application server is considered as ”failed”, another application server is started automatically by a corresponding autonomic agent, and the IP forwarding is changed by the autonomic agent on the firewall/router by issuing an ”iptables” script.

Another monitor resides on the firewall/router. It periodically ”pings” the application server using the ”wget” command. If the return code is not 200 (a HTTP return code 200 indicates a successful HTTP connection), the application server is reported as failed, another application server is started, and the IP forwarding is changed the same way as above. Figure 6 shows how to ”ping” an application server and update the status of the application server in the SERVERS table.

4.2.3. Self-healing Simulation Servers Simulations run on simulation servers. The autonomic agents frequently

uplode the underlying operating system metrics such as load average, free memory and CPU usage to the autonomic manager through the SERVERS table. This operation also serves as a heart-beat of the simulation server. If the autonomic manager does not receive the heart-beat in a timely fashion, the simulation server is reported as failed. All simulations currently running on the simulation server are marked as CRASHED. The corresponding rows in the SUBMISSIONS table are updated so that the dispatcher, which is implemented as Java stored procedures and running in the database as part of the autonomic manager, distributes the crashed simulations to other appropriate simulation servers by restarting from the checkpoint right before the occurence of failure.

4.2.4. Self-healing Database Servers The database server is the heart of the computing grid, because the autonomic manager resides in it. The autonomic manager controls all autonomic agents running in the server pool. Meanwhile, simulation generated data is also loaded into the database server and eventually extracted, transformed and loaded into the data warehouse. It is vital to keep the database server running 24×7. However, in a naturally unreliable environment, database servers still can be out of service due to hardware or software errors, no matter how carefully it was designed.

Redundancy is required to ensure self-healing, therefore, a hot standby database is configured for the primary database, as in Figure 4. Whenever the primary database is DOWN, database connections can be failed over to the standby database and the standby database will take the role of the primary database. The original primary database can be recovered by the database administrator and reconfigured as the standby database for the new primary database.

We use the following information to protectively self-heal the database server.

- The table V$SESSION - The maximal number of concurrent database connections is setup by the initialization parameter PROCESSES (for example, 300). The autonomic manager is aware of this capacity and queries the number of rows (current connections) in the V$SESSION table, no further database connections are allowed if the number of current connections is approaching this maximum. This preventively removes the ”maximum number of connections exceeded” error.
- The table DBA_FREESPACES - The database capacity is monitored so that when a threshold (90% for example) is exceeded, certain actions such as purging old data or increasing database size are performed by the autonomic manager.
- The alert log file - The alert log file records significant events occurred in the database. It can be monitored so that the primary database can switch over to the standby database and the primary database can bounce itself to prevent cascading system failure.

4.2.5. Self-healing Simulations Using the checkpointing mechanism, simulations can achieve self-healing. The EXECUTIONS table in Figure 5 records information about executions of simulations. As described in Section 3, each simulation consists of a series of executions. Each execution can be on a different simulation server (server_id), with different process ID (pid). Checkpoint data is used to restart the simulation if required.

The J2SE 1.5 Management and Monitoring API can be used to monitor the JVM which starts the simulations. The autonomic agents on the simulation servers also monitor the execution of simulations. If an execution dies prematurely, the SUBMISSIONS table in Figure 5 is updated for the corresponding execution by setting the status to CRASHED. The dispatcher, part of the autonomic manager, in the database will re-distribute the crashed simulation to an appropriate simulation server. The corresponding autonomic agent restarts the crashed simulation using the same command (for example, java -jar ReactionBatchModel.jar -simulation_id 1234) used to start the same simulation. The checkpoint_data in the EXECUTIONS table will be used to restart the simulation.

An execution of a simulation is monitored as follows: the autonomic agent retrieves status (EXECUTING or COMPLETED) and pid from the EXECUTIONS table for its underlying simulation server. If the status of an execution is EXECUTING, it then queries the underlying operating system to see whether the simulation is running with the corresponding pid. If such pid does not exist, the execution is reported as CRASHED.

4.3. Self-optimizing

Self-optimizing involves self-tuning service parameters by measuring resource usage and performance autonomously. One such implementation for autonomic web-based simulation is load balancing the server pool. Load balancing is achieved by both the dispatcher (part of the autonomic manager inside the database) and the autonomic agents as described in the following steps:

- A new simulation (or a crashed simulation) is assigned to a simulation server with lowest OS load metric that is below a present threshold by the dispatcher.
- The autonomic agents check the DISPATCHER table periodically to see whether a simulation job is assigned and start the simulations if any.
- At each checkpoint, the autonomic agents contacts the autonomic manager about whether migration is necessary taking the latency of restart into account.

Figure 7. The NOM Simulation Research Portal

- Simulations on heavily loaded servers are checkpointed and restarted on lightly loaded servers through the dispatcher.

Using the above load balancing technique, the server pool is able to self-optimize to improve performance.

4.4. Self-protecting

Self-protecting the system will prevent large-scale correlated attacks or cascading failures that permanently damage the system. Besides careful configuration of the firewall, the following configuration is setup for the computing grid. Users of the computing grid must register and be verified by the system administrator. The system administrator assigns an appropriate role to the user. This involves three tables USERS, USER_ROLES and VERIFIER in the data model as shown in Figure 5.

5. Conclusions and Future Work

This paper presents a prototype of autonomic web-based simulation, based on the vision of autonomic computing. The basic requirement to achieve autonomic web-based simulation is proposed. The implementation of an autonomic infrastructure to support autonomic web-based simulation is discussed. This work is initiated by the NSF funded interdisciplinary project "Stochastic Synthesis: Simulating the Environmental Transformations of Natural Organic Matter". The purpose of the work described in this paper is to let our collaborators access the simulation models any time from anywhere. More autonomic features will be implemented in the future to achieve autonomic web-based simulations. Figure 7 shows the portal for the project which hosts autonomic web-based simulations.

6. Acknowledgments

This research was partially supported by a NSF ITR Grant No. 0112820. We acknowledge the contributions of X. Xiang, R. Kennedy and T. Schoenharl for their respective discussions.

References

[1] G. Abdulla, T. Critchlow, and W. Arrighi. Simulation data as data streams. *SIGMOD Record*, 33(1):89–94, 2004.

[2] M. Agarwal and M. Parashar. Enabling autonomic compositions in grid environments. In *Proceedings of the Fourth International Workshop on Grid Computing*, pages 34–41, 2003.

[3] J. Appavoo, K. Hui, C. Soules, R. Wisniewski, D. D. Silva, O. Krieger, M. Auslander, D. Edelsohn, B. Gamsa, G. Ganger, P. McKenney, M. Ostrowski, B. rosenburg, M. Stumm, and J. Xenidis. Enabling autonomic behavior in systems software with hot swapping. *IBM Systems Journal*, 42(1):60–76, 2003.

[4] K. Chandy. A survey of analytic models for rollback and recovery strategies. *Computer*, 8(5):40–47, 1975.

[5] C. Dabrowski, K. Mills, and A. Rukhin. Performance of service-discovery architectures in response to node failures. In *Proceedings of the International Conference on Software Engineering Research and Practice*, pages 95–101, 2003.

[6] J. Daly. Presenting a model for predicting the optimum checkpoint interval for restart dumps. In *Proceedings of the International Conference on Computational Science*, pages 3–10, 2003.

[7] Y. Diao, J. Hellerstein, S. Parekh, and J. Bigus. Managing web server performance with autotune agents. *IBM Systems Journal*, 42(1):136–149, 2003.

[8] L. Ferreira, V. Berstis, J. Armstrong, M. Kendzierski, A. Neukoetter, M. Takagi, R. Bing-Wo, A. Amir, R. Murakawa, O. Hernandez, J. Magowan, and N. Bieberstein. *Introduction to grid computing with Globus*. IBM Corporation, 2003.

[9] J. Gray. What next?: A dozon information-technology research goals. *Journal of the ACM (JACM)*, 50(1):41–57, 2003.

[10] V. Holmes, W. Johnson, and D. Miller. Integrating metadata tools with the data services archive to provide web-based management of large-scale scientifi c simulation data. In *Proceedings of the 37th Annual Simulation Symposium*, pages 72–79, 2004.

[11] Y. Huang, X. Xiang, and G. Madey. A self manageable infrastructure for supporting web-based simulations. In *Proceedings of the 37th Annual Simulation Symposium*, pages 149–156, 2004.

[12] W. Inmon. *Building the data warehouse*. John Wiley & Sons, 1996.

[13] B. Jacob, L. Ferreira, N. Bieberstein, C. Gilzean, J.-Y. Birard, R. Strachowski, and S. Yu. *Enabling applications for grid computing with Globus*. IBM Corporation, 2003.

[14] J. Jann, L. Browning, and R. Burgula. Basic building blocks for autonomic computing on ibm pseries servers. *IBM Systems Journal*, 42(1):29–37, 2003.

[15] J. Kephart and D. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[16] B. Khargharia, S. Hariri, M. Parashar, L. Ntaimo, and B. uk Kim. Vgrid: A framework for building autonomic applications. In *Proceedings of the International Workshop on Challenges of Large Applications in Distributed Environment*, pages 19–27, 2003.

[17] R. Kimball. *The data warehouse toolkit*. John Wiley & Sons, 1996.

[18] J. Kohl and P. Papadopoulas. Effi cient and flexible fault tolerance and migration of scientifi c simulations using cumulvs. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, pages 60–71. ACM Press, 1998.

[19] V. Markl, G. Lohman, and V. Raman. Leo: An autonomic query optimizer for db2. *IBM Systems Journal*, 42(1):98–106, 2003.

[20] V. Nicola. *Checkpointing and the modeling of program execution time*. John Wiley & Sons, 1995.

[21] F. Quaglia. Combining periodic and probabilistic checkpointing in optimistic simulation. In *Proceedings of the thirteenth workshop on Parallel and distributed simulation*, pages 109–116. IEEE Computer Society, 1999.

[22] T. Truong. An integrated web-based grid-computing environment for research and education in computational science and engineering. In *Proceedings of the 37th Annual Simulation Symposium*, pages 143–148, 2004.