Survivability Simulator for Multi-Agent Adaptive Coordination *†‡

Régis Vincent, Bryan Horling, Thomas Wagner and Victor Lesser
University of Massachusetts
Computer Science Department
Email: {vincent,bhorling,wagner,lesser}@cs.umass.edu

UMass Computer Science Technical Report 1997-60

October 16, 1997

Abstract

The growth of a distributed processing system can increase both the number and likelihood of attacks it may be subject to over its lifetime [14, 7]. This fact, in addition to the complexity inherent in such an environment, makes the survivability of large heterogeneous systems one of the most challenging research areas currently being investigated[1]. Our goal is to create a distributed simulation system to test various coordination mechanisms allowing the elements of the system to detect, react and adapt in the face of adverse working conditions. Our assumption is that the system is composed of a group of autonomous agents. Each agent has its own local view of the world and its own goals, but is capable of coordinating these goals with respect to remote agents. To simulate these complex systems, an environment is needed which permits the simulation of an agent's method execution. To this end, we are developing a distributed event-based simulator capable of simulating the effects directed attacks or a capricious environment have on agent method execution and recovery.
Keywords: Multi-agent systems, discrete simulation, simulator

*A version of this paper appears in the *Proceedings of the First International Conference on Web-Based Modeling and Simulation*, 1998.

†Effort sponsored by the Defense Advanced Research Projects Agency (DARPA) and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-1-0249. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. Disclaimer: The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), Rome Laboratory or the U.S. Government.

‡The effort depicted is partially sponsored by the Dept. of the Navy, Office of the Chief of Naval Research, under Grant No. N00014-97-1-0591 and the content of the information does not necessarily reflect the position or the policy of the Government, and no official endorsement should be inferred.

1 Challenge

With the advent of open computing environments, widely networked information resources, and an equally widely networked client base, applications are evolving from highly centralized structures to distributed decentralized structures. Distribution is required to cope with the complexity of the tasks, to distribute the load, and to utilize the necessarily distributed resources or expertise used by these applications. In this setting, application interactions are also evolving and often a distributed application is actually a collection of separate applications, each possibly having other tasks outside of the given application and possibly under the management or ownership of different controlling entities. With increasing frequency, an agent computing paradigm is used to control these application components where each contributing application is an agent, having its own goals, functionality, and perhaps special computing resources. The agents interact via higher-level interfaces, exchanging high-level information such as goals and partial plans, rather than simply invoking each other's low-level methods via remote-function-call style interfaces. The agent-style control methodology lends itself to this integration task because it enables designers to explicitly represent agent goals, where agents may have *multiple* goals and goals for *different* client applications, and to reason about the benefits or relative importance of these goals from a self-interested perspective [8, 9, 12], a wholly cooperative perspective [6, 5], or a market-driven economic perspective [13].

While this highly networked and distributed environment is giving rise to the next generation of powerful applications, the same decentralized networked characteristics that serve as the growth catalyst also create the greatest hazard to these applications, that is the vulnerability of these applications to attack from outside sources [14, 7]. Applications that are open, or built from individual components that are themselves open systems, are susceptible to virus style attacks and to attacks that disable the network, block communication, or disable member applications. Attacks may be deliberate, i.e., the action of an adversary, or they may simply be system failures that affect the application like a deliberate attack by adversely affecting the application. Attacks may also be more covert in nature, involving misinformation or the confusion of communications or member applications. To address attacks, deliberate or otherwise, the evolving heterogeneous distributed applications must also be *flexible*; they must be able to cope with a component application going down or the loss of communications between various components. The distributed system architecture must adapt dynamically, perhaps finding different routings for communications or other component applications to fulfill a particular set of needs.

Our research focus is on building such survivable, dynamic, heterogeneous distributed systems. We use an agent coordination control scheme, called *generalized-partial-global-planning* (GPGP) [2], to organize the agent members of a distributed application. In GPGP, agents exchange their local views of the group problem solving context and negotiate to determine which agent is to perform which task(s) at what time(s). Individual agents are comprised of three primary modules: 1) the GPGP coordination module, which is responsible for interacting with other agents and maintaining a cohesive distributed application; 2) the local real-time scheduler [16, 15] that performs trade-off analysis of different possible courses of action and decides, in conjunction with the GPGP module, which actions to perform and at what times; 3) the domain problem solver that performs the domain (application) actions. Thus agents consist of two main control problem solving components, the scheduler and the coordination module, and one main domain problem solving component. Agents represent and reason about

problem solving processes using the TÆMS task modeling framework [3, 4]. In TÆMS tasks are hierarchically decomposed into subtasks and finally into primitive actions. TÆMS differs from most traditional modeling frameworks in that it represents both hard and soft interactions between tasks, e.g., enabling or facilitating relationships, and alternative ways to perform tasks. Part of the agent control scheme is to reason about inter-agent and intra-agent task interactions and to reason about the different ways to perform tasks, and the different quality, cost, and duration trade-offs of each alternative way. For example, for the task of gathering information on a competitor cooperation, the agents may be able to achieve the task in short order and little cost, but may sacrifice quality to do so. However, given more time and money, the agents may be able to explore more information resources and produce a higher-quality report.

One major thrust of our current work is on studying and improving the flexibility and adaptability aspects of our agent coordination mechanisms. Recent work in learning coordination mechanisms [10, 11] has shown that it is more effective to learn situation specific coordination strategies rather than using a single strategy for all situations. The work on flexibility and adaptability is related in that a situation specific learning component is needed, perhaps in conjunction with approximate expert knowledge, to learn when and how to adapt to attacks. Accordingly, we must simulate these complex distributed agent-based heterogeneous systems so that we can study the mechanisms in a controlled fashion. We are developing a distributed event-based simulator to simulate agent action execution in the face of system attacks and recovery. The simulator is a single centralized artifact which handles the TÆMS action execution for all agents in the system. The agents are decoupled, running as external processes, as in a real application; we will discuss the complexities of this decoupling in the next section. In addition to action execution, all agent communication is also routed through the simulator. Thus, attacks can change or modify action performance, e.g., causing failure, poor results, or changing resource consumption characteristics, or attacks may change communication characteristics, causing interference, blockages, delays, and lost or garbled messages. From an architectural perspective, the simulator replaces the component of the agent responsible for actual method execution and the coupling is transparent to the other agent control components. Similarly, the communications component is modified so that the simulator controlled routing is transparent. This action/communication coupling between the simulator and the agents enables the simulator to affect system execution performance in a reactive sense. However, to experiment with distributed application prototypes, or to study distributed applications for which the domain problem solving component has yet to be implemented, the simulator must also be able to interact with the agents in a proactive sense. The distinction lies in where the agent plans originate. If the agent plans are generated by a fully implemented domain problem solver, then the simulator's role is limited to modulation of action execution and agent communications. However, if the plans are generated by the simulator and seeded to the respective agents, the simulator can not only control action execution and communications, but it can control the high-level objectives of one or more agent's in the system. This functionality is implemented by replacing the agent's domain problem solving component with a stub that receives TÆMS task structures from the simulator. Thus the agent /simulator coupling is at the action, communication, and, optionally, plan-generation levels.

We will return to these issues in the subsequent sections. Section 2 describes the software design of our simulator and the different attacks and the measures taken by the simulator. Section 3 provides our current implementation status and describes the features of Java that facilitate this work. Section 4 defines the development next phase and our planned future work.

2 Simulator Design

The simulator, Figure 1, is designed as a central process; all agents involved with the model are connected to the simulator using sockets. The agents themselves are independent processes, which could run on physically different machines. Time synchronization is controlled by the simulator, which periodically sends a pulse to each of its remote agents. Each agent has a local *manager* which converts the simulation pulse into real CPU time. The *manager* controls process and records the time spent scheduling, planning, or executing methods. Note, the simulator does not control the agents' activities, it merely allocates time slices and records the activities performed by the agent during the time slide via an instrumentation interface. Once the alloted CPU time has been used, the *manager* halts the agent and sends an acknowledgment message back to the simulator. The simulator waits for all acknowledgments to arrive before sending a new pulse (so all process are synchronized). Using this method, any functions (even computationally long ones) can be executed using whatever pulse-granularity is desired, i.e., if we wish to simulate an agent environment where all planning actions complete in a single time pulse, the pulse-granularity is defined by the longest running planning action. For example, a planning phase may explore a very large space search, but the simulated time required may be just one pulse. Since all remote processes are frozen until the planning is completed, this gives the illusion that the planning was actually performed very quickly. Each agent therefore has its own task-specific time transformation function, which it uses to convert each simulator pulse into local CPU time. However, the clock mechanism works at the other end of the spectrum too. If we wish to study actual planning times, where slower planners take longer, or agents executing on slower machines take longer, then the pulse-granularity is smaller and a tick may correspond to a second in real-time or cpu-time.

The simulator is also a message router, in that all agent communications will pass through it. This scheme permits explicit control over network and communication delays. In this way, if we want to simulate a very fast communication path, the simulator may immediately re-send a message to its destination, but if we want to simulate a compromised network, the simulator may wait n pulses before sending the message. This method also allows an agent to broadcast a message to all other agents without explicitly knowing the number of agents that will receive the message.

The simulator behavior is directed by a queue containing a time-ordered list of events. Each message it receives either adds or removes events from the queue. At each pulse the simulator selects the correct events and realizes their effects (for example, a network may slow down). Only after the effect of each event has been completely determined is the timing pulse sent to each agent. Primitive actions in TÆMS called *methods* are characterized statistically via discrete probably distributions in three dimensions, quality, cost, and duration. Agents reason about these characteristics when deciding which actions to perform and at what time.

When a agent wants to execute a method, it sends a message to the simulator with the method name. The simulator then retrieves the method from the *objective* TÆMS database. Agents schedule, plan, and interact using a *subjective* view of the methods. The subjective view differs from the objective view because agent's may have an imperfect model of what will actually happen, performance-wise, when the method is executed. For example, for a method that involves retrieving data from a remote site via the WWW, the remote site's performance characteristics may have changed since the agent learned them and thus the agent's view of the execution behavior of that method, namely its duration, is imperfect. In a simulation

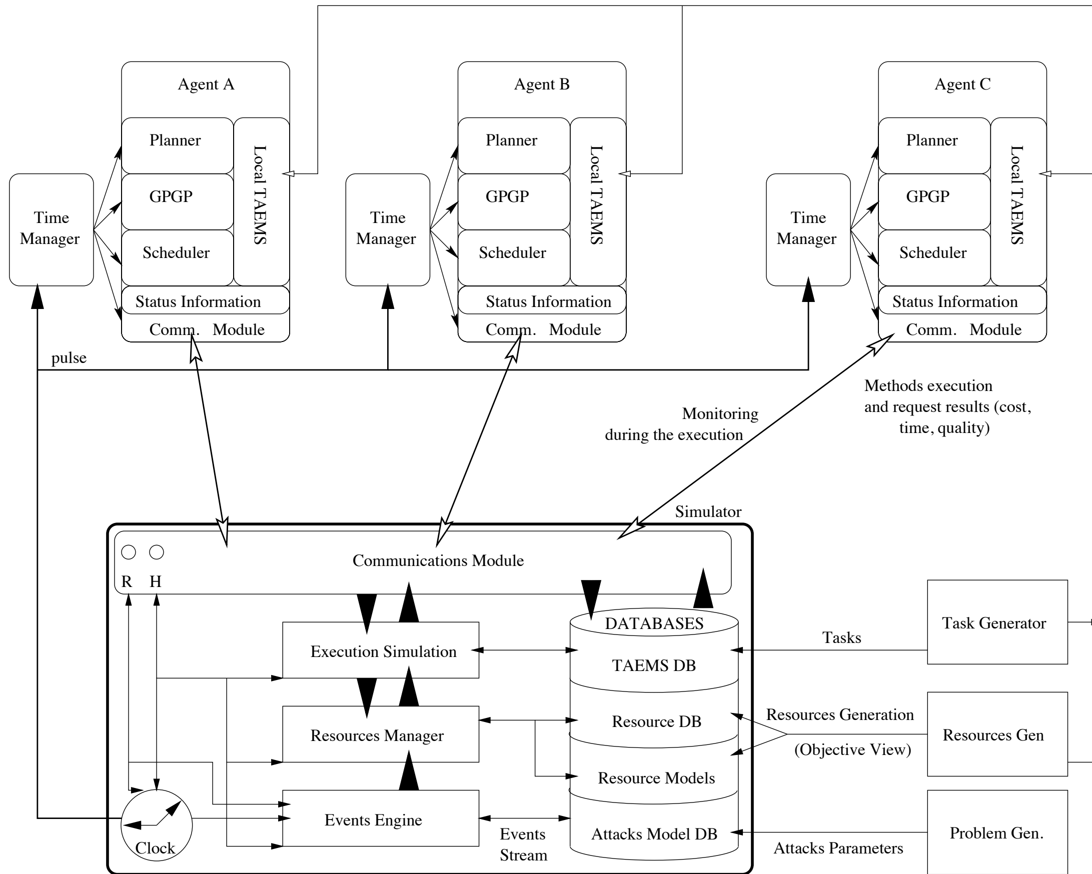

Figure 1: Simulator and Agent Architectures

environment, both the subjective and the objective method views are created by the simulator / task generator and the objective, or true, views of the methods are stored in the simulator's TÆMS database. Thus when an execution message arrives, the simulator must obtain the objective view of the method before any other steps may be taken. The first step of the simulator is to calculate the value of the cost, duration and quality that will be "produced" by this execution. The duration (in pulse time) is used to create an event that sends to the agent the result in terms of cost and quality. Any event realized before the newly queued event is performed may change the results of the newly queued event's "execution." For example, a network breakdown event at the front of the queue may increase the time of all the simulated executions, that follow it in the queue, by 100%. Thus subsequent method completion events are delayed.

This interaction effect is also possible because of the interactions between the methods themselves. For example, if one method *enables* another method, and the first method fails, then the other method may no longer be executed or executing the method will produce no result. If both methods are already "scheduled" in the event queue, and the first method fails, then the event associated with the second method's execution must be changed.

The random generator used to calculate the cost, duration and quality values is seeded by a fixed parameter or by a random number (depending of the current time). The solution of seeding by a fixed parameter is used to have a deterministic simulation. Our random generator

produces the same answer if the same seeding value is used. The goal is to compare different agent coordination mechanisms on the same problem using the same simulation. To test a particular coordination mechanism on several problems, we use a seeding based on the current time which guarantees that two simulations do not produce the same solution.

The next section will describe our preliminary implementation in Java, and will describe in details the synchronization problem.

3 Preliminary Implementation

As noted above, the simulation environment was designed to be both distributed and platform independent, in order to provide the most amount of ability and flexibility when modeling complex interactions among heterogeneous agents. Java was therefore the language of choice for its implementation, as it offers robust communication primitives and bytecode interpreters on a wide range of platforms, among other benefits. The 1.1 API was used to create the environment, which consists of a single "server" process, called the *simulator*, and one or more "client" processes, which serve as *managers* to the agents executing within the system.

Each manager has under its control an external agent process, which at this time is a black box written in an arbitrary language. The manager serves as an interface between the agent and the simulation environment; it is responsible for process control, message passing and event delivery. When a manager is started, it is given both the address of the simulation server and the name of the agent which is to be under its control. The manager then spawns a child process (which will become the agent process) and attempts to contact the simulation server, after which it idles until the actual simulation process is initiated. During simulation, the manager waits for a clock pulse to be delivered by the simulator, which indicates the start of a time slot. Upon receipt of the pulse, the manager converts it into its own local time scale by waking the agent process and permitting it to run for a locally specified amount of time. In this way, various processor or code optimizations can be modeled by simply letting the agent run for a different lengths of time. When the manager has determined that the agent's time slot has been used up, the agent process is halted and a pulse acknowledgment is returned to the simulator. Any events or messages generated by the agent are sent to the simulator before acknowledging the pulse, so that they will be recognized as originating during that particular time slot.

For our implementation, native methods were needed to both execute and control the agent, as the process paradigm offered in the Java API does not offer the low level controls required to manage an external process. While this affects the portability of the code, the relatively small number of native methods provide the important ability to make use of previously written agents and also permit a much more diverse agent pool. A simple Java-based threaded agent stub has also been written, which can be used on platforms where process control is not easily available. A second artifact of this approach is that the process monitoring is performed in user space and thus is dependent on the exact preemption time of the agent and manager processes. Because of this, fine grained timing control is not possible, although for our purposes, this is an acceptable tradeoff of complexity, since pulse conversions are typically measured in seconds.

The simulator functions as the central hub of the simulation environment, being responsible for agent synchronization, event simulation and modification, message routing and time management. When the simulator starts up, it initializes an event queue, a local clock, and a communications server which waits for manager initiated connections. As each agent joins, it

is represented locally at the simulator by both agent specification and communications objects. It is through these instances that the simulator communicates with and controls the remote agents. All communications at this time are done with TCP based character streams. During the simulation, agents can join or leave at any time, but typically the server will wait for some number of agents to connect and then begin the simulation process, after which the agent pool remains constant. The simulation time line is discrete, in that all events and agent execution periods fall completely within a specified time slot. Activity during each of these slots begins with the simulator first waiting for all remote agents to complete their activity (as indicated by the pulse acknowledgment), then incrementing the clock and sending a new pulse to each agent. This synchronization step is crucial to the validation of the distributed nature of the simulation environment. Without it, there would be no clear notion of when events take place, leading to race conditions amongst the agents and the simulated or actual events. Once the agents have been notified, all events scheduled to terminate during that time period are retrieved and realized, after which the time slot is considered completed. At this time, events elsewhere in the queue may be modified to simulate resource or behavior changes. For instance, a message delivery may be delayed to mimic a faulty network, or the results of a simulated computation may be altered by a theoretical adversary. The simulation time line then progresses in this manner, with the agent and simulator functioning essentially in lockstep, until the simulator is paused or terminated.

4 Future Directions

As we have just completed a prototype of the simulation environment, much work remains to be done. We anticipate the following features to be integrated into the system in the near future:

- Problem and attack generators to create, direct and modify the agents' goals. A related feature is the ability to quantify the survivability test-level and the degree of adaptation required of the agent coordination mechanisms.
- "Real" agents, capable of working on a given task structure and communication with other peer agents.
- Agent communication interception and routing.
- Communication implementation using serialization, CORBA or Java RMI.

The main advantage of our simulator will be:

- Platform independent distributed implementation.
- Domain independent architecture.
- Deterministic behavior on the part of the simulation environment. If the agents themselves are deterministic, two executions will produce the same event stream and results.
- Attacks are clearly defined as events, permitting the modeling of complex coordinated attacks.

Our goal is to construct a multi-agent simulator based on an object-oriented model in Java. The simulator is currently in an early stage of development. Our design's main advantage in using Java are robust network capabilities and platform independent implementation. We expect that a preliminary version will be available by January 1998.

References

[1] DARPA. http://www.ito.darpa.mil/ResearchAreas/Information_Survivability.html, 1996.

[2] Keith S. Decker. *Environment Centered Analysis and Design of Coordination Mechanisms.* PhD thesis, University of Massachusetts, 1995.

[3] Keith S. Decker. TÆMS: A framework for analysis and design of coordination mechanisms. In G. O'Hare and N. Jennings, editors, *Foundations of Distributed Artificial Intelligence*, chapter 16. Wiley Inter-Science, 1995.

[4] Keith S. Decker and Victor R. Lesser. Quantitative modeling of complex environments. *International Journal of Intelligent Systems in Accounting, Finance, and Management*, 2(4):215–234, December 1993. Special issue on "Mathematical and Computational Models of Organizations: Models and Characteristics of Agent Behavior".

[5] Keith S. Decker and Victor R. Lesser. Designing a family of coordination algorithms. In *Proceedings of the First International Conference on Multi-Agent Systems*, pages 73–80, San Francisco, June 1995. AAAI Press. Longer version available as UMass CS-TR 94–14.

[6] E. H. Durfee, V. R. Lesser, and D. D. Corkill. Cooperative distributed problem solving. In A. B. Barr, P. Cohen, and E. Feigenbaum, editors, *The Handbook of Artificial Intelligence*, volume 4, pages 83–147. Addison Wesley, 1989.

[7] M. Eichin and J. Rochis. With microscope and tweezers: An analysis of the internet worm of november 1988. In *IEEE symposium on Research in Securiry and Privacy*, 1989.

[8] Terry M. Moe. The new economics of organization. *American Journal of Political Science*, 28(4):739–777, November 1984.

[9] Charles Perrow. *Complex Organizations.* Random House, New York, 1986.

[10] M. V. Nagendra Prasad and Victor R. Lesser. Learning problem solving control in cooperative multi-agent systems. In *Workshop on Multi-Agent Learning, AAAI-97*, Providence, Rhode Island, 1997.

[11] M. V. Nagendra Prasad and Victor R. Lesser. The use of meta-level information in learning situation_specific coordination. In *IJCAI-97*, Nagoya (Japan), 1997.

[12] J. S. Rosenschein and J. S. Breese. Communication-free interactions among rational agents: A probabilistic approach. In L. Gasser and M. N. Huhns, editors, *Distributed Artificial Intelligence, Vol. II.* Pitman Publishing Ltd., 1989.

[13] Tuomas Sandholm and Victor Lesser. Issues in automated negotiation and electronic commerce: Extending the contract net framework. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 328–335, 1995.

[14] S. Staniford-Chen, S. Cheung, R. Crawford, M. Dilger, J. Frank, Hoagland J., K. Levitt, C. Wee, R. Yip, and D. Zerkle. Grids-a graph based intrusion detection system for large networks. Technical report, NISSC, 1996. http://seclab.cs.ucdavis.edu/arpa/-grids/welcome.html.

[15] Thomas Wagner, Alan Garvey, and Victor Lesser. Complex Goal Criteria and Its Application in Design-to-Criteria Scheduling. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, July 1997. Also available as UMASS Department of Computer Science Technical Report TR-1997-10.

[16] Thomas Wagner, Alan Garvey, and Victor Lesser. Criteria-Directed Heuristic Task Scheduling. *International Journal of Approximate Reasoning, Special Issue on Scheduling*, To appear 1997. Also available as UMASS Department of Computer Science Technical Report TR-97-59.