Proceedings of the 2006 Winter Simulation Conference
L. F. Perrone, F. P. Wieland, J. Liu, B. G. Lawson, D. M. Nicol, and R. M. Fujimoto, eds.

SASSY: A DESIGN FOR A SCALABLE AGENT-BASED SIMULATION SYSTEM USING A DISTRIBUTED DISCRETE EVENT INFRASTRUCTURE

Maria Hybinette
Eileen Kraemer
Yin Xiong
Glenn Matthews
Jaim Ahmed

Computer Science Department
University of Georgia
Athens, GA 30602-7404, USA

ABSTRACT

The PDES literature offers a rich set of techniques for distributed and efficient simulation. However, there is a growing need for simulators that support agent-based applications, and PDES systems are not always well suited for these applications. Example agent-based applications include simulation of biological systems such as ants and bees, multi-robot systems and battlefield simulations. The robotics research community has developed agent-based simulators that provide useful APIs for agent applications. However, such simulators have performance limitations, and they do not scale well. Our approach is to provide middleware between an agent-based API and a PDES simulation kernel. The result is a simulation system that offers an agent-based API for the programmer to a high performance PDES system. Here we describe our design and initial implementation of SASSY, the **S**calable **A**gent**s** **S**imulation **Sy**stem. We describe our initial implementation and compare the design with related approaches.

1 INTRODUCTION

Our goal in this work is to leverage the efficiency, speed, and parallelism available in discrete event simulation (DES) systems for agent-based modeling (ABM). Our approach is to use a standard parallel discrete event simulation (PDES) kernel paired with added middleware to provide an agent-based paradigm for the simulation application developer.

We are not the first to propose the use of PDES for agent-based modeling (see, for instance, Uhrmacher et al. 2000, Logan and Theodoropoulos 2001, Riley and Riley 2003). However, we believe there are several unique aspects of our approach that contribute a novel high-performance design. In particular, we use a "standard" PDES kernel, and we provide a "standard" agent-based model view. Because we use a standard PDES kernel we are able to easily leverage existing and future performance technologies such as optimistic protocols, distributed execution, and advanced efficient Global Virtual Time calculations. Because we provide a standard ABM API, we make the simulation application developer's job easier – she can more directly map her problem to the simulator without having to know the details of PDES.

Why PDES? For more than two decades researchers have focused on improving the efficiency of discrete event simulation systems (DES). Numerous techniques have emerged such as parallel execution on shared memory multiprocessors, distributed execution on multiple machines, optimistic execution, faster algorithms for Global Virtual Time calculation, duplicating (cloning) simulations in progress to aid what-if-scenario analysis and more recently software systems and architectures to improve interoperability between different simulation technologies (e.g., distributed interactive simulation (DIS) and the high-level architecture (HLA) and their extensions) (Jefferson and Sowizral 1985, Das et al. 1994, Fujimoto 1990, Hybinette and Fujimoto 2001, Dahmann et al. 1997).

Why Agent-Based Modeling? Several emerging simulation applications call for an agent-based view that is not well suited to the DES model. Accordingly, agent-based simulators have been developed (e.g., TeamBots (Balch 1998), Swarm (Minar et al. 1996), Mason
(Luke et al. 2005), Player/Stage (Gerkey et al. 2003)), but these simulators suffer from performance and scalability limitations. In order to illustrate these points we will briefly review the DES and agent-based simulation paradigms.

2 RELATED WORK

DES systems are fast and efficient because the systems they simulate are treated as if they proceed forward in discrete time steps – the intervening time is ignored. As compared to continuous time simulation, the discrete nature of time in DES systems enable a reduction in complexity because the requirement for synchronization is reduced. Furthermore,

researchers have considered carefully how to gain speedup by distributing and parallelizing DES across multiple processing elements in **P**arallel **D**iscrete **E**vent **S**imulation (PDES) systems. In general DES implementations use an event list paradigm where events are scheduled at a particular time T by adding them to a priority queue with timestamp T. Events are processed by **L**ogical **P**rocesses (LPs). PDES systems treat scheduling of events as messages that are sent between LPs (the LPs are possibly on different machines).

The PDES paradigm is well suited for simulation applications that consist of multiple computational or processing nodes with packets or messages passing between them. Networking simulators, for instance, represent routers as LPs, and packets as messages/events. Other examples include simulation of air traffic with airports as LPs and aircraft as messages, and road systems as intersections (LPs) and cars (messages). In this paradigm, computation occurs at the fixed LPs – the messages that move between them have no computational capability. Most applications for PDES involve a large number messages in comparison to the number of LPs.

The standard PDES API for simulation developers is not well suited to agent based applications because it does not offer the programming model these researchers expect. For example, multi-agent system (MAS) researchers expect to treat agents as objects that move around in an environment (like messages in DES, but with the ability to compute). In most PDES simulations LPs don't move, they represent geographically static objects such as network routers, airports, sectors in the airspace, intersections. So, in these simulators the objects that perform computing don't move. In contrast, in physical agent simulations the agents move around. In general, ABM researchers expect their agents to Riley and Riley (2003), Balch (1998), Gerkey et al. (2003):

- **Use the Sense-Think-Act cycle** – agents sense their environment, consider what to do, then act. This is the predominant computational paradigm for agents; it stands in contrast to the message/event paradigm for PDES.
- **Compute** – Agents have computing capability and state; again, in contrast to messages in PDES, which provide no computing function.
- **Proliferate** – MAS simulations typically involve hundreds or thousands of agents.
- **Persist** – Agents are persistent members of the environment, in contrast to messages that exist only for a short periods.

For these reasons, a number of MAS and multi-robot systems researchers have devised their own simulation systems for their research. From a software engineering and ease of use point of view their simulators are well suited to the research tasks they pursue, but these simulators are not

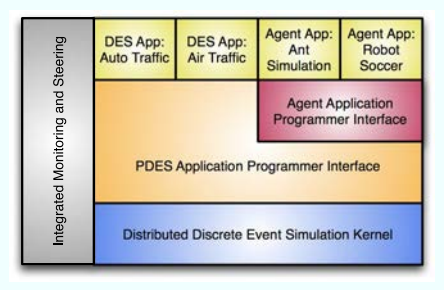

Figure 1: SASSY: The (S)calable (A)gent(s) (S)imulation (Sy)stem: Our Distributed Simulation System Supports Both General Simulation Applications and Agent Based Simulations

"high performance" in the same sense that state of the art PDES systems are. In fact, some agent based simulation systems face serious performance limitations. These limits prevent MAS researchers from investigating systems with thousands or millions of agents.

We feel the best solution is to provide middleware between a PDES kernel and agent-based API. This will enable MAS researchers to program using a model that is comfortable for them, while they leverage the high performance of an underlying PDES kernel.

3 SYSTEM DESIGN

We have implemented an architecture like that illustrated in Figure 1. In this architecture, a faster than real time simulation runs on one of our servers (or on several in the case of a parallel and distributed simulation). Faster than real time simulation allows models to advance ahead of the corresponding wall-clock time. Through a web server, Internet users are able to query and steer the simulation. In some cases, as in a traffic and a multi-robot simulation that we have developed, users can request specific simulation results for their personal use. At the same time, the researchers who have designed the simulation will be able to revise it while it is running.

4 THE PHYSICAL AGENT MODEL

In the standard physical agent model, an agent senses its environment, considers what to do, then acts (see Figure 2). This is frequently referred to as the sense-think-act cycle (Riley and Riley 2003, Uhrmacher et al. 2000, Logan and Theodoropoulos 2001).

Multi-agent simulators are typically configured as shown in Figure 3. The code for each agent connects to a process that maintains world state for the simulation. An Application Programmer's Interface (API) allows agents to query the simulator for sensor information and to send actuation commands to the simulator. The simulator updates

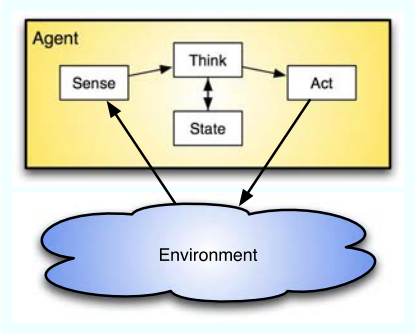

Figure 2: The Physical Agent Model

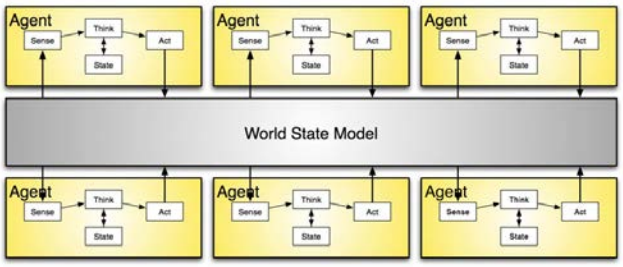

Figure 3: An Agent-Based Simulation

the world state accordingly. The simulator checks for possible physical interactions that would prohibit a requested action. The simulator also moderates interactions between agents (such as communication).

The agents may be implemented in a number of ways. In TeamBots, for instance, agents are Java objects with "call back" methods the simulator calls to give them an opportunity to run (Balch 1998). In SPADES and Player/Stage the agents are separate processes that connect to a single-threaded simulation engine (Riley and Riley 2003, Gerkey et al. 2003).

5 DISCRETE EVENT SIMULATION

Discrete event simulations typically maintain data structures of state variables, an event queue of forthcoming time-stamped events and a global clock that indicates the progress of the simulation (see Figure 4). The simulation advances by repeatedly processing the event containing the smallest time stamp from the event list. Processing an event may cause one or more state variables to be modified, and/or new events to be scheduled. For example, a discrete event simulation may model an air traffic system where state variables indicate the number of airplanes at each airport. Departure and arrival events modify these variables as new aircraft arrive or depart from the airport (Wieland 1998).

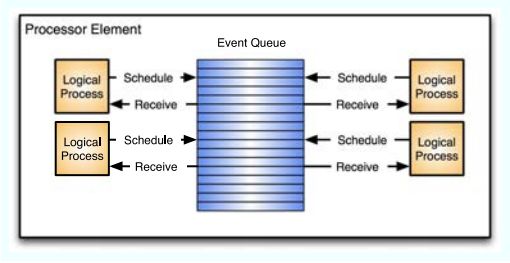

Figure 4: A Discrete Event Simulation System in Which Several LPs Schedule and Execute Events

Distributed parallel simulation provides two advantages. First, multiple processors can be used to reduce the execution time of the simulation. Second they may be required to support distributed personnel or resources (e.g., a combat simulator with multiple human participants at different locations). Distributed simulation also facilitates linking existing simulators developed for different platforms to model large systems.

In this work, a parallel simulation is composed of distinct components called logical processes or LPs. Each LP models some portion of the system under investigation. For example in an air traffic simulation each airport might be represented by an LP. The logical processes may be mapped to different processors. As in a sequential simulation, a change in system state is defined by an event. The "scheduling" of an event is accomplished by sending a message from one LP that may request the destination LP to change its state or schedule additional events.

A synchronization mechanism is used to ensure each LP processes events in time-stamp order. The two leading classes of synchronization protocols are *conservative* and *optimistic* approaches. A conservative protocol enforces consistency by avoiding the possibility of an LP ever receiving an event from its past (as measured in simulated time). The optimistic protocol, in contrast, uses a detect-and-recover scheme. When an event is received in an LP's past, an LP recovers by rolling back previously processed events with later time-stamps than the one that was just received.

6 API TO THE SASSY PDES KERNEL

SASSY provides a PDES API (application programmer interface), as well as an Agent-Based API that is implemented as middleware on top of a PDES kernel (see Figure 1). In this section we describe the PDES API.

The API for the PDES simulation application programmer is simple and easy to use. SASSY is implemented in Java, and therefore benefits from object-oriented system design. The kernel provides an abstract class *Logical Process* that implements the features of a generic logical process.

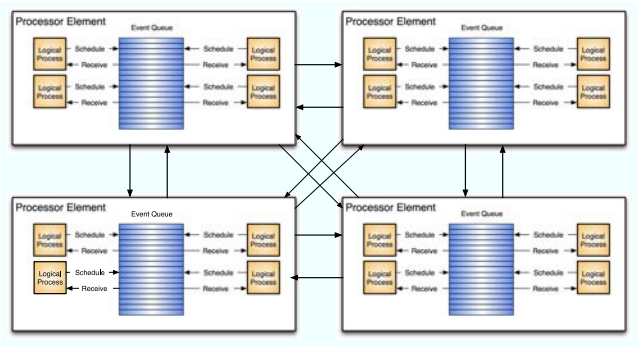

Figure 5: A Parallel Discrete Event Simulation System in Which Several PEs Support Multiple LPs to Schedule and Execute Events

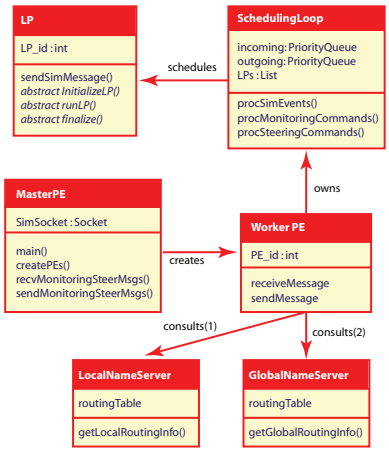

Figure 6: SASSY: The (S)calable (A)gent (S)imulation (Sy)stem: Java Implementation

These methods are implemented as private; the application programmer does not need interact with them. Methods requiring application-specific implementation are designated *abstract* and are to be implemented by the application programmer.

To implement a simulation application to run on SASSY a programmer extends the LogicalProcess class by implementing three abstract methods: `initializeLP`, `runLP`, and `finalizeLP` (see Figure 6). These methods respectively initialize simulation data structures, describe an execution handler that is run when the logical process is scheduled, and call routines that are activated when the logical process leaves the simulation.

The LPs schedule events both remote and local (including events to itself) via messages. A SASSY message is a Java object that specifies the destination of the message and is sent via the `sendSimMessage()` method.

To run a simulation application on SASSY, two configuration files are specified. One file specifies simulation system configuration information (e.g., number of PEs, available machine IP numbers). The other file specifies application specifics such as the names of the simulation objects and their corresponding numeric identifiers. Each line in this file provides information about one LP in the application.

Consider as an example a road traffic simulation consisting of road segments, implemented so that each road segment corresponds to a logical process. A line of the application configuration file might look something like:

```
Segment(tab)I-85N-S(tab)I-85
north       from       I-285
to               I-75/85(tab)
9250/3/0,I-7585N-S
```

As the simulation kernel parses the line, it creates an instance of the Segment class (which is a subclass of LogicalProcess), sets its application-level name ID to "I-85N-S", sets its description to "I-85 north from I-285 to 1-75/85", and passes the string "9250/3/0,I-7585N-S" into the Segment's `setConfigData()` method. This method, implemented by the application programmer, parses the string to define this Segment (LP) as 9250 meters long, 3 lanes wide, initially containing 0 cars, and sending all cars leaving this Segment to another Segment (LP) identified as "I-7585N-S". The simulation kernel then assign this newly created LP to one of the PEs.

A monitoring and steering capability permits us to dynamically add (and remove) logical processes (simulation objects) through the monitoring and steering console.

6.1 Initialization, Execution and Wrap-Up

Because the simulation application class objects are created by SASSY according to the information contained in the configuration file, the initialization phase for the application is simple. Typically, at least one logical process sends a message in the initialization phase.

After the simulation is initialized, it enters the execution phase. The simulation application runs as specified by the application programmer in the `appRunLP()` method.

In the `appRunLP()` method the application programmer specifies the behavior of the simulation: what the LP should do upon receiving a message. Typically, the LP responds to the receipt of a message by updating its local state based on the content of the message and optionally generating one or more messages to be sent either to itself or to other LPs.

It is up to the simulation application to end the simulation. Currently, there are two ways to specify the end of the simulation: at a specified simulation time, or when some amount of real time has elapsed in which no messages have been sent. The monitoring and steering system provides

direct control of termination and can stop/pause/rewind and restart the simulation.

7 IMPLEMENTATION OF THE PDES KERNEL

The simulation kernel is implemented as a package of Java classes. Initially, the kernel is composed of two or more PEs (processing elements). One PE serves as the MasterPE, the remainder serve as WorkerPEs. The MasterPE gets execution information from the user at the command line, creates WorkerPEs accordingly, processes the configuration information, and allocates LPs to the WorkerPEs. The LPs are initialized by the WorkerPEs according to the configuration information.

The SASSY kernel uses a hierarchical name service structure, similar to DNS (Mockapetris 1987). A global name server runs on the MasterPE and contains a mapping of the PE's IDs to their physical addresses. Each WorkerPE runs a local name server that maintains LP ID to physical address mapping information for all the LPs running on this PE.

In order to reduce PE to PE communication, each local name server also caches the mapping information concerning LPs on other PEs whenever they acquire such information.

Each WorkerPE is responsible for receiving and sending messages. It periodically checks its two queues: the incoming queue and the outgoing queue. For an incoming message, the WorkerPE checks the message type and processes it accordingly. For an outgoing message, the WorkerPE consults the local name server to find the physical address of the destination. If the local name server does not contain this information, the WorkerPE consults the global name server to get the physical address and then sends the message to that PE.

8 THE AGENT-BASED MODEL API

In this section we describe SASSY's Agent-Based Modeling API. The ABM API is implemented as middleware between the PDES kernel and the agent-based model application code (see Figure 1). Each agent is provided a PDES proxy LP that serves to process and create messages. Figure 7 shows how this works for one agent.

Each agent proxy maintains a model of relevant objects in the environment near the corresponding agent it serves as a proxy for. When agents move or act in the world they generate an event that is sent to the other nearby agents so they can track the movements and state of others. The agent proxy LPs keep their state current for the agent they support.

As mentioned above, the Agent Proxy LP (APLP) keeps track of the world's state that is relevant to the agent it serves. In our approach, there is no central representation of world state. Instead, the world state relevant to each

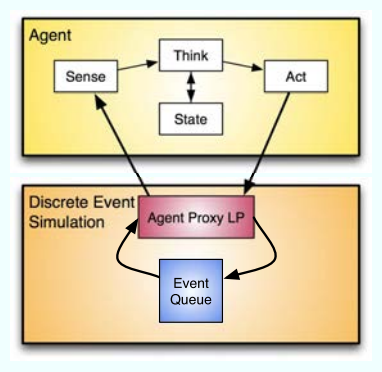

Figure 7: An LP in the PDES System Serves As a Proxy for a Simulated Physical Agent

agent is maintained by that agent's proxy LP. As an agent's state changes, it notifies other agents using a state message reflection mechanism. Message reflection is accomplished by a distributed publish/subscribe mechanism implemented by a set of LPs arranged in a grid. These LPs are referred to as Interest Monitoring LPs (IMLPs). Each agent registers interest in (subscribes to) the activities that occur within specific cells. Agents that move within a specific cell periodically publish their state by sending a message to the relevant IMLP; then the IMLP reflects those messages to other interested agents.

Our approach is similar to HLA interest manager approaches that use conservative clocks (e.g., Tacic and Fujimoto's work reported in (Tacic and Fujimoto 1998) and Wang, Turner and Wang's work in (Wang et al. 2003)). Tacic and Fujimoto's work focuses on reducing network traffic in a simulation using a conservative protocol (HLA) while Wang, Turner and Wang describes how to integrate agents using different interest management schemes into an HLA-based distributed simulation. In contrast, our approach supports optimistic simulation, and our focus is on reducing the number of rollbacks.

Logan and Theodoropoulos also propose a related approach in (Logan and Theodoropoulos 2001). Logan and Theodoropoulos implemented interest management for an optimistic simulator. In their approach world state is maintained in "environmental LPs" (similar to our IMLPs). In our approach the agents track world state themselves, there is no centralized representation of any agent's state.

We believe our approach enables more efficient parallel execution, as well as allowing the agent LPs to advance optimistically. Further, it reduces the necessity for rollbacks. We illustrate the approach with an example below. Recall that our focus is to reduce rollback while minimizing bandwidth requirements.

Consider the scenario illustrated in Figure 8. In this case we are concerned with agents A, B, C, and D. A and B will

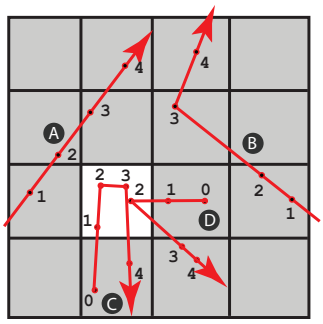

Figure 8: Four Agents: A, B, C and D That Roam About a 2 Dimensional Space; the Light Colored Region Is an Interest Region Maintained by IMLPj; Positions of the Agents and Their Directions Are Denoted by Dots and Arrows (the Numbers Refer to Instants in Simulated Time)

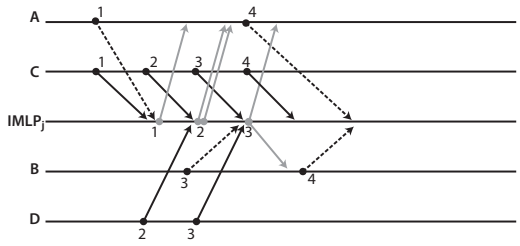

Figure 9: Event Message Timeline for Agent LPs A, B, C and D and IMLPj

register interest in the light colored cell (for convenience we will refer to it as IMLPj). C and D will move through the J cell and post state messages to IMLPj. In a real simulation all four agents would subscribe to multiple IMLPs and they would also all post state information to the IMLP cell they travel within. In this example we focus on the messages related to A and B acting as subscribers and C and D acting as publishers.

In Figure 9 we show an example timeline of event messages sent to and from IMLPj. In this timeline, all four agents are roughly synchronized. Events occur as follows: Agent A *subscribes* to information from IMLPj at time 1, and *unsubscribes* at time 4 (all times are given in simulation time). Agent B *subscribes* at time 3, and *unsubscribes* at time 4. Agent C enters cell J at time 1, it sends an *enter* message at time 1, then *state* messages at time 2 and 3. Agent C leaves cell J at time 4, and sends a corresponding *leave* message at that time. IMLPj receives the state messages from C, and "reflects" them to A and B at the appropriate times. Agent D enters cell J at time 2 and leaves at time 3; it sends appropriate *enter* and *leave* messages at those times. IMLPj reflects the time 2 state message from D to agent A at time 2. Note that Agent B does not have to be notified of D's activities because it

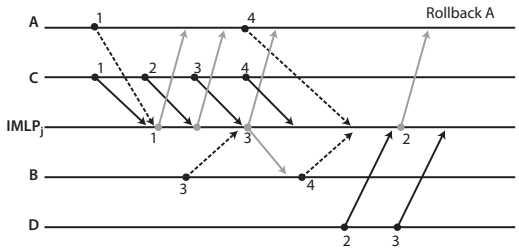

Figure 10: Time Lines of Agents (Agent LPs): A, B, C and D and Interest Manager LP That Corresponds to a Cell in the Grid (in This Example There Is One Cell per LP); This Example Shows That Agent A Needs to Roll Back

was not interested in IMLPj at time 2. In this scenario no rollbacks were necessary.

IMLPs maintain a record of state messages and subscription windows back to Global Virtual Time (GVT). If a state message arrives within the subscription window of a particular agent, the IMLP will reflect that message to the interested agent. Also, if later in time, an agent registers interest in events during a time window in the past, the IMLP will reflect those old messages from that time to the agent.

Now consider the timeline in Figure 10. Events move forward in a manner similar to the earlier example, except that agent D is somewhat behind the other agents. In real time agent D arrives in cell J after C has come and gone, and after A and B have unsubscribed to information about cell J. D's message at time 2 is reflected to agent A. A is forced to rollback because it had already advanced in simulated time to time 4. Note that agent B does not have to rollback because it is not affected by D's activities.

Note that $IMLP_j$ never had to rollback. In fact, it is never necessary for an IMLP to rollback (a proof of this assertion is left for future research).

9 MONITORING AND STEERING MODULE

Different simulation applications have different needs for external runtime input and control (steering) and output and display (monitoring). SASSY has a powerful and flexible built-in monitoring and steering (M/S) architecture that can accommodate these varying needs.

When initialized, the simulation creates a socket and listens on a (user-specified) port for connections from a monitoring/steering client program. This M/S client can be implemented in several ways – it can be interactive (as might be needed for run-time adjustment of the simulation parameters either by a human researcher or by a custom steering module such as a machine learning algorithm) or non-interactive (such as feeding in sensor data at periodic intervals for comparison with the simulation's prediction) and can be implemented in whatever programming lan-

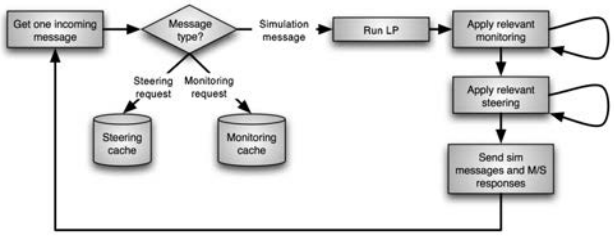

Figure 12: Incoming Steering and Monitoring Messages are Kept in a Cache; the Kernel Manages the Cache and Schedules Applies Monitoring and Scheduling Requests; Monitoring and Steering Events Can Be One-Time, On-Going, Periodic or Conditional Basis

guage the user prefers, so long as it is capable of sending and receiving through a socket connected to the simulation kernel.

A simple application protocol, modeled after HTTP, is used for the exchange of M/S requests and responses between the M/S client and the simulation. Four types of messages are exchanged: monitoring requests from the client, the corresponding reports from the simulation, steering requests from the client, and the corresponding acknowledgments from the simulation. Although these messages occur as request-response pairs, they are not synchronous, as the client may make a request to be carried out at some future simulation time, in which case no corresponding response will be received from the simulation until the monitoring request is fulfilled. Further, note that the monitoring request may initiate a continuing stream of responses rather than a single response. SASSY provides a number of built-in M/S features at the kernel and simulation levels as well as an API for adding additional M/S capability at the application level.

Incoming requests whose types are recognized by the kernel are handled at that level, while unrecognized requests are assumed to be application-specific and are forwarded to the application for handling (See Figure 11 and Figure 12 for details).

Handling at this points consists of scheduling the monitoring and steering requests in the appropriate request cache. Monitoring and steering actions occur only at their scheduled times and when their conditions, if any, are satisfied. This approach permits monitoring and steering actions to be performed on a one-time, ongoing, periodic or conditional basis.

The M/S architecture is linked with SASSY from the lowest levels of the simulation kernel to the application level allowing monitoring and steering flexibility. It can monitor and steer at various levels, the application level variables, simulation level logical processes (e.g., scheduling semantics) and the simulation itself (e.g., stop, pause, and replicating the whole simulation). In other words in addition to fine-grained application-level monitoring and steering, e.g., observing and adjusting individual application variables, it is also possible to monitor and steer the simulation itself at a very coarse-grained level, even to the point of controlling and observing powerful kernel features such as load-balancing, cloning, and merging of simulations.

10 CONCLUSION AND FUTURE WORK

We have presented the design and implementation details of SASSY, a hybrid simulation system that supports agent-based simulation on top of a high-performance PDES kernel. SASSY also provides monitoring and steering support. The PDES kernel and monitoring and steering components have been implemented. The agent-based modeling API is under development. We expect soon to evaluate SASSY using standard applications such as PHOLD and our traffic and agent simulators that are under development.

REFERENCES

Balch, T. 1998. *Behavioral diversity in learning robot teams*. Ph. D. thesis, College of Computing, Georgia Institute of Technology.

Dahmann, J. S., R. Fujimoto, and R. M. Weatherly. 1997. The department of defense high level architecture. In *Proceedings of the 1997 Winter Simulation Conference(WSC-1997)*, 142–149.

Das, S., R. Fujimoto, K. Panesar, D. Allison, and M. Hybinette. 1994, December. GTW: A Time Warp system for shared memory multiprocessors. In *Proceedings of the 1994 Winter Simulation Conference Proceedings (WSC-1994)*, 1332–1339.

Fujimoto, R. M. 1990, October. Parallel discrete event simulation. *Communications of the ACM* 33 (10): 30–53.

Gerkey, B. P., R. T. Vaughan, and A. Howard. 2003, Jul. The player/stage project: Tools for multi-robot and distributed sensor systems. In *Proceedings of the International Conference on Advanced Robotics*, 317–323. Coimbra, Portugal.

Hybinette, M., and R. M. Fujimoto. 2001. Cloning parallel simulations. *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 11 (4): 378–407.

Hybinette, M., and R. M. Fujimoto. 2002. Latency hiding with optimistic computations. *Journal of Parallel and Distributed Computing* 62 (3): 427–445.

Jefferson, D. R., and H. Sowizral. 1985. Fast concurrent simulation using the time warp mechanism. In *Distributed Simulation 1985*, Volume 15 of *Simulation Council Proceedings*, 63–69. Society for Computer Simulation (SCS).

Logan, B., and G. Theodoropoulos. 2001. The distributed simulation of agent-based systems. *http://www.cs.bham.ac.uk/research/pdesmas/*.

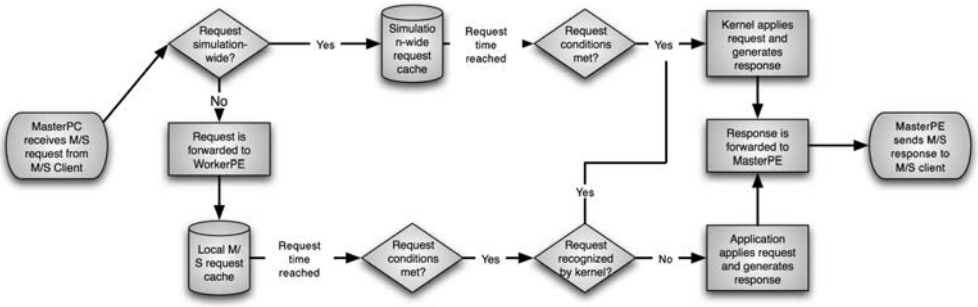

Figure 11: Monitoring and Steering Messages Are Routed Either to the Application Level or to the Simulation Control Level; For Application Control Level Messages the Kernel Makes Sure That the Message Is Routed to the Appropriate Machine (Local or Remote)

Luke, S., C. Cioffi-Revilla, L. Panait, K. Sullivan, and G. Balan. 2005. MASON: A multiagent simulation environment. *SIMULATION* 81:517–527.

Minar, N., R. Burkhart, C. Langton, and M. Askenazi. 1996. The swarm simulation system: A toolkit for building multi-agent simulations. *Santa Fe Institute*.

Mockapetris, P. V. 1987, November. RFC 1034: Domain names — concepts and facilities. Obsoletes RFC0973, RFC0882, RFC0883. See also STD0013 Updated by RFC1101, RFC1183, RFC1348, RFC1876, RFC1982, RFC2065, RFC2181, RFC2308. Status: STANDARD.

Riley, P. F., and G. F. Riley. 2003, December. SPADES – a distributed agent simulation environment with software-in-the-loop execution. In *Proceedings of the 2003 Winter Simulation Conference (WSC-2003)*, 817–825.

Tacic, I., and R. Fujimoto. 1998. Synchronized data distribution management in distributed simulations. In *Proceedings of the* 12^{th} *Workshop on Parallel and Distributed Simulation (PADS-98)*, 108–115.

Uhrmacher, A. M., P. Tyschler and D. Tyschler. 2000. Modeling and simulation of mobile agents. *Future Generation Computer Systems* 17 (2): 107–118.

Wang, L, S. J. Turner and F. Wang. 2003. Interest management in agent-based distributed simulations. In *Proceedings of the Seventh IEEE International Symposium on Distributed Simulation and Real-Time Applications(DS-RT 2003)*, 20–29.

Wieland, F. 1998, December. Parallel simulation for aviation applications. In *Proceedings of the 1998 Winter Simulation Conference(WSC-1998)*, 1191–1198.

AUTHOR BIOGRAPHIES

MARIA HYBINETTE is an assistant professor in the computer science department at the University of Georgia (UGA). Her research is focused on distributed simulation, multi-agent-based simulation and experimental systems. She completed her Ph.D. at Georgia Tech and then was a research scientist also at Georgia Tech. Before joining UGA she was employed as a staff simulation & modeling engineer at the MITRE Corporation. She now directs the Distributed Simulation Laboratory (DSL) at UGA with Eileen Kraemer.

EILEEN KRAEMER is an associate professor in the computer science department at the University of Georgia. Prior to joining the faculty at UGA, she served on the faculty at Washington University in St. Louis in the Computer Science Department of the School of Engineering and Applied Science, she is a co-director of the Distributed Simulation Laboratory and served as director of the Computer Visualization Laboratory. She received her Ph.D. in Computer Science in September of 1995 from the College of Computing at the Georgia Institute of Technology in Atlanta.

YIN XIONG is is a Ph.D. student in Computer Science department at the University of Georgia. Her research interests include developing mechanisms to enable efficient and scalable simulation systems for large scale agent based simulations. She received her M. S. in Computer Science in 2001 from the University of Georgia.

GLENN MATTHEWS is an MS student and was a research assistant at the Distributed Simulation Laboratory at the University of Georgia. His current research interest is in the area of machine learning.

JAIM AHMED is an MS student in Computer Science at the University of Georgia. His research interests include agent-based simulations. He joined the Distributed Simulation Laboratory at the University of Georgia in 2005.