MPADES: Middleware for Parallel Agent Discrete Event Simulation

Patrick Riley[*]

Carnegie Mellon University, Computer Science Dept., Pittsburgh, PA 15213-3891,
pfr@cs.cmu.edu, http://www.cs.cmu.edu/~pfr

Abstract. Simulations are an excellent tool for studying artificial intelligence. However, the simulation technology in use by and designed for the artificial intelligence community often fails to take advantage of much of the work by the larger simulation community to produce stable, repeatable, and efficient simulations. We present the new system Middleware for Parallel Agent Discrete Event Simulation (MPADES) as a simulation substrate for the artificial intelligence community. MPADES focuses on the agent as a fundamental simulation component. The "thinking time" of an agent is tracked and reflected in the results of the agents' actions. MPADES supports and manages the distribution of agents across machines while being robust to variations in network performance and machine load. We present the system in detail and give experimental results for a simple world model and set of agents. MPADES is not tied to any particular simulation, and is a powerful new tool for creating simulations for the study of artificial intelligence.

1 Introduction

Simulations are an excellent tool for studying artificial intelligence. They can allow the systematic modification of parameters of the environment, execute the large number of trials often required for machine learning, and facilitate the interaction of agents created by different research groups. However, many general simulation environments do not address the special concerns of the artificial intelligence community, such as the computation time of the agent being an integral part of its behavior. On the other hand, many simulators created in the artificial intelligence community fail to take advantage of the vast work in the simulation community for designing stable, repeatable, and efficient simulations.

This paper covers the MPADES (Middleware for Parallel Agent Discrete Event Simulation) system. The system is designed to support simulations for the AI community without being tied to any particular simulated world. MPADES provides support for simulations with agents running in parallel across multiple machines, and for tracking the computation time used by those agents. By taking

[*] I would like to thank Emil Talpes of the Carnegie Mellon ECE department for his contributions to the design and initial implementation of the MPADES system. This research was sponsored by Grants Nos. F30602-98-2-0135 and F30602-00-2-0549 and by an NSF Fellowship. The content of this publication reflects only the position of the authors.

advantage of work in discrete event simulation, the middleware eases the design of a simulation by taking care of many of the system details required to handle distribution in an efficient and reproducible way.

The system is implemented and is described in detail in this paper. In order to test the efficiency of the system, a simulation of a simple world and agents was created. Experimental results with this world model are given, but it should be emphasized that MPADES is not tied to any particular world model.

2 Related Work

The problem of creating efficient simulations has attracted diverse attention from a wide range of sources, including the AI community, scientific computing, industry, and government.

A division in the simulation literature is between discrete event and continuous simulations. A discrete event simulation is one in which every system change takes place in the form of events which occur at a particular time. Continuous simulations simulate worlds which evolve continuously over time, typically by using a small time step and advancing all components of the simulation step by step. MPADES is a hybrid of the two. The agents' interactions with the world are through discrete events, but the underlying world model can be a continuous simulation. When an event is ready to be realized (that is cause a state change in the simulation), the world is first stepped forward to the appropriate time. However, all the reasoning about distributed messages and timing is done in the discrete event framework.

MPADES is a conservative simulator (such as [1]); events are not realized until it can be guaranteed that casual event ordering will not be violated. Optimistic simulations [2], on the other hand, support a back up mechanism in case events are executed out of order. Debates over the merits of conservative and optimistic simulation are common and several surveys discuss the issues [3, 4].

Agent-based or agent-oriented simulation seems to be a relatively new idea for the simulation community. There are still differing ideas about what "agent-based" simulation actually means. MPADES reflects ideas similar to Uhrmacher [5], where the deliberation (i.e. thinking) time of the agent is considered to be part of the simulation events. Uhrmacher also provides an overview of the current state of agent-oriented simulation [6].

Agent simulation has existed for much longer in the AI community. Perhaps the most comprehensive work on simulations that track the thinking time of the agents is the MESS system by Anderson [7, 8]. MESS allows the computation of the agents to be tracked at a level as low as primitive LISP instructions, as well as providing perfectly reproducible results. However, MESS requires that all agents are written in LISP and provides no support for distributed, parallel simulation.

One widely used agent simulation from the AI community is the SoccerServer [9], which provides a simulation of a soccer game among agents. While the SoccerServer supports distribution of agents and effectively limits the computation time that the agents have, there are a myriad of problems with the structure of the SoccerServer.

The SoccerServer operates in fairly large discrete time steps (100ms by default), commonly called a cycle, and all agents take actions simultaneously. Discretizing what is fundamentally a continuous problem is always a difficult task and large discrete steps cause problems with action and interaction modelling. MPADES allows agent actions to take place over many cycles, helping to create a more realistic simulation. The SoccerServer could run at smaller time steps to achieve better simulation. However, the smaller the time step, the more random effects like network performance and machine load can affect the result of the simulation (see below). In addition, the SoccerServer rigidly requires that all agent actions begin at synchronized discrete cycles.

In order to limit the computation time of the agents, the SoccerServer relies on a hard limit of wall clock time. Each discrete cycle, the SoccerServer sends sensations in the form of network packets to the agents based on a fixed wall clock time schedule. The agents send actions as network packets, and if the actions do not arrive before the specified wall clock time in the server, the actions are not applied. Thus, any change to the network configuration, network load, or machine load can cause the outcome of the simulation to vary considerably. This makes it extremely difficult to run reproducible simulations unless every aspect of the network can be carefully controlled. Even then, it is not clear whether the results from a simulation in one network and machine configuration are directly comparable to a simulation in another configuration. One of the major goals of MPADES is to avoid this dependency on the network and machine configurations.

Also, the SoccerServer performs very poorly in terms of efficiency of CPU usage. Because of the many wall clock timings in the server, it is difficult to avoid having the CPU idle for lengthy times. When running all components on a single machine, MPADES always achieves 100% CPU usage.

TeamBots (previously known as JavaBots) is another simulation environment which comes from the robotics community [10]. The primary focus of TeamBots is in creating control algorithms which can then be applied directly to robots without rewriting. Similar to the SoccerServer, the world advances in time steps with all agent actions synchronized. TeamBots requires all agents to be written in Java, and while agents could be distributed across machine with Java remote method calls, there is no mechanism for achieving a parallel speedup by distributing the agents. However, the TeamBots interface provides a mechanism for creating reusable control and learning algorithms, which is an issue not addressed with MPADES.

The High-Level Architecture (HLA) [11] is one example of a general architecture for creating distributed simulations. Rather than dealing with specific simulation algorithms, HLA and similar architectures provide interface and mechanism specifications to support reusable and inter-operable simulations. MPADES supports a smaller set of simulations, but provides more structure to aid in the construction of any particular world model. In essence, MPADES makes it easier to write agent based simulation of the sort described here, while HLA is designed to support a broader class of simulations. The algorithms of MPADES could be implemented using an HLA interface.

3 System Features

This section covers the major features of the MPADES simulation system. Briefly, these relevant features:

- Agent based execution, including explicit support for modelling latencies in sensation, thinking, and acting.
- Agents can be distributed among multiple machines.
- The result of the simulation is unaffected by network delays or load variations among the machines.
- The architecture for the agents is unconstrained, and does not require that the agents are written in a particular programming language.
- The agents' actions do not have to be synchronized in the domain.

Before discussing these features, an important point of terminology must be clarified. When we use the term “simulation time” or simply “time”, we will be referring to the time in the simulated world. For example, if we say that event A occurs 5 seconds after event B, we mean 5 seconds in the simulated world, not 5 seconds to a person outside watching the simulation run. If we want to refer the latter time, we will explicitly say “wall clock” or “real world” time.

MPADES supports agent-based *execution*, as opposed to agent-based modelling or implementation [6]. In this context, agent-based execution means that the system explicitly models the sensing, thinking, and acting components (and their latencies) which are the core of any agent. Figure 1 represents a time line of executions of this cycle. Consider the topmost cycle. Time point A represents the point at which a sensation is generated, such as the frame of video from a robot's camera or a snapshot of stock market prices. The time between points A and B represents the time to transfer/process a sensation to be used by the thinking component starting at point B. Between points B and C, some computation is done to determine which actions to take. Between points C and D, the messages are sent to the effectors and those actions begin to take effect at point D. Note that we are not claiming that there is a fundamental difference between the computation that happens in the sense and act components of the cycle and the think component. A simulation system must necessarily create an abstraction over whatever world is being modelled. Since the simulation provides information and receives actions in a processed form, the use of an explicit latency allows the simulation to account for the time that that would be needed in the real world to convert to and from that processed form. Note that MPADES does *not* require that all sensations and actions have the same latency.

In many agents, the sense, think, and act components can be overlapped in time as depicted in Figure 1. MPADES explicitly allows all overlaps except that

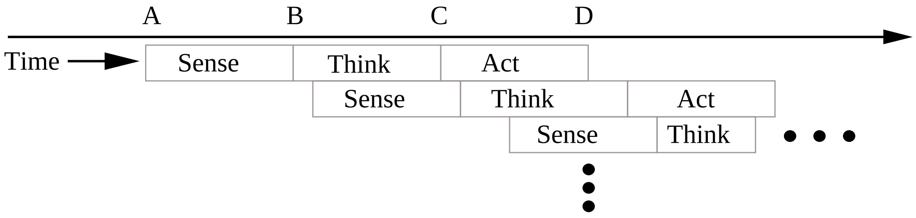

Fig. 1. Example timeline for the sense-think-act loop of an agent

think cycles may not overlap in time because it is assumed that each agent has a single processing unit.

Another important feature of the agent-based execution of the MPADES system is that the computation resources used by the agent are tracked in order to determine the latency of the thinking of the agent. This provides an incentive for agents to use as little computation as possible in the same way that the real world does: if an agent takes a long time to think, the world will have evolved while it was thinking. Thinking faster leads to a tighter control loop and generally better execution.

Unlike many other simulators which support the tracking of an agent's thinking latency (see Section 2 for a brief discussion), MPADES supports the distribution of agents across machines. Creating efficient parallel simulations is notoriously hard [3], and MPADES provides efficient distributed simulation for environments with the right properties (see Sections 6 and 7 for more details).

In spite of distributing the simulation across machines and in contrast to other AI simulation systems (notably the SoccerServer [9]) MPADES provides a reproducibility property: changes in network topology, network traffic, or machine loads does not affect the *result* of the simulation, only the *efficiency*.

In order to provide maximum inter-operability, MPADES makes no requirements on the agent architecture (except that it supports the sense-think-act cycle) or the language in which agents are written (except that they can write to and from Unix pipes). In the same spirit as the SoccerServer [9], MPADES helps provide an environment where agents built with different architectures or languages can inter-operate and interact in the simulated world.

As discussed in Section 2, MPADES is a discrete event simulator in the interaction with the agents, and a continuous simulator for the world model. This means that all of the distributed reasoning is done in the discrete event framework, while the world model works in small time steps. This notably means that the agents' actions are not necessarily synchronized; any subset of the agents can have actions take effect at a given time step. Many simulations, especially in the AI community, require that all agents choose an action, then the global effect of those actions is then applied, then the agents choose again.

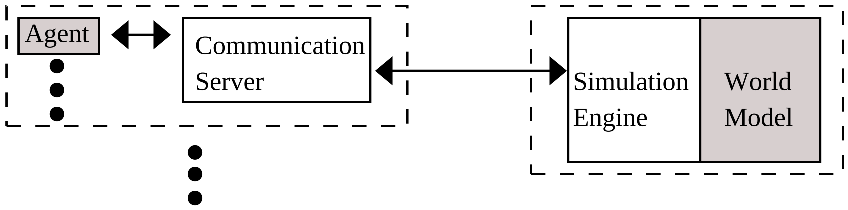

Fig. 2. Overview of the architecture of the MPADES system. The shaded components are provided by the users of the system, not by the middleware itself. The dotted lines denote machine boundaries.

4 System Architecture

Figure 2 gives an overview of the entire MPADES system, along with the components users of the system must supply (shaded in the diagram). The simulation engine and the communication server are supplied as part of MPADES. The world model and the agents are created by a user to simulate a particular environment.

The simulation engine is the heart of the discrete event simulator. All pending events are queued here, and the engine manages all network communication. A communication server must be run on each machine on which agents run. The communication server manages all communication with the agents (through a Unix pipe interface) as well as tracking the CPU usage of the agents to calculate the thinking latency. The communication server and simulation engine communicate over a TCP/IP connection.[1]

The world model is created by a user of the middleware to create a simulation of a particular environment. The simulation engine is a library to which the world model must link, so the simulation engine and world model exist in the same process. The world model must provide such functionality as advancing the state of the world to a particular time and realizing an event (changing the state of the world in response to an event occurring). MPADES provides a collection of C++ classes from which objects in the world model can inherit in order to interact with the simulation engine.

The agents communicate with the communication server via pipes, so the agents are free to use any programming language and any architecture as long as they can read and write to pipes. From the agent's perspective, the interaction with the simulation is fairly simple:

1. Wait for a sensation to be received
2. Decide on a set of actions and send them to the communication server
3. Send a "done thinking" message to indicate that all actions were sent

[1] Since the simulation needs the lowest latency traffic possible in order to achieve efficient simulation, Nagle's algorithm is turned off on the TCP sockets (using the `TCP_NODELAY` socket option in the Linux socket interface). Manual bundling of messages is done to avoid sending an excessive number of small packets.

One of the communication server's primary jobs is to track the thinking time of the agent. When sending a sensation to an agent, the communication server begins tracking the CPU time used by the agent, information provided by the Linux kernel. When the "done thinking" message is received, the communication server puts the agent process into a wait state and calculates the total amount of CPU time used to produce these actions. The CPU time is translated into simulation time through a simple linear transformation[2]. All actions are given the same time stamp of the end of the think phase.

The CPU time used by the process is a reasonable measure for thinking time since it avoids many machine load and memory usage variation issues. However, there may still be slight effects of machine load based on, for example, the number of interrupts and cache performance. Also, the current time slice reported by the Linux kernel is in 10ms increments, a fairly large amount of computation on today's computers. With the randomness in interrupts and other system activity, CPU usage numbers are unfortunately not perfectly reproducible, in contrast to a more language specific time tracking system like [7, 8]. However, tracking the CPU time via kernel-provided information allows much more flexibility in the implementation of the agents and provides a better substrate for interoperability of agents designed by separate groups.

The agents have one special action which MPADES understands: a "request time notify." The agent's only opportunity to act is upon the receipt of a sensation. Therefore if an agent wants to send an action at a particular time (such as a stop turning command for a robot), it can request a time notify. A time notify is an empty sensation. On the receipt of the time notify, the agent can send actions as normal. In order to give maximum flexibility to the agents, MPADES does not enforce a minimum time in the future that time notifies can be requested. However, all actions, whether resulting from a regular sensation or a time notify, are still constrained by the action latency.

While the world model may create new events to be used in the simulation, there are several special kinds of events which the simulation engine understands:

Sense Event When a sense event is realized, a message is sent to the communication server representing that agent. The communication server forwards this message to the agent and begins tracking agent computation time.

Time Notify This event is an empty sensation event, and can be inserted in the queue by the special action "request time notify" of the agent.

Act Event When actions are received from the agents, an act event is inserted into the queue. The simulation event does nothing on the realization of the event, expecting the world model to handle the effects.

[2] The parameters of this linear transformation are a user controllable parameter. The bogoMIPS value of the processor as reported by the Linux kernel is also used. However, if different types of machines are to be used in the same simulation, some benchmarking would need to be done to determine what an equitable setting of these CPU time to simulation time parameters would be.

```
repeat forever
    receive messages
    next_event = pending_event_queue.head
    while (no event will be received for time less than next_event.time)
        advanceWorldTime (next_event.time)
        pending_event_queue.remove(next_event)
        realize (next_event)
        next_event = pending_event_queue.head
```

Table 1. Inner loop for basic conservative discrete event simulator

5 Discrete Event Simulator

This section describes the simulation algorithm used by MPADES. This algorithm is a modification of a basic discrete event simulator.

The inner loop of a basic discrete event simulator is shown in Table 1. The one primary difference is the "advanceWorldTime" call. This supports the one continuous aspect of the simulation, the world model. This function advances the simulated world to the time of the discrete event. The realization of the event causes changes in the world, and notably can cause other events to be enqueued into the `pending_events_queue`.

The key decision in parallel discrete event simulation is whether any event will be received in the future which has a time less than that of the next event. Typically, a "lookahead" value is determined such that the current time plus the lookahead is the smallest time stamp of a message that can be received. This problem is well studied (see [3] for a survey). This section is devoted to the lookahead algorithm of MPADES. We will first cover a simple version which covers some of the fundamental ideas and then describe the MPADES algorithm in full.

An explanation of the events that occur in the normal think-sense-act cycle of the agents must first be given. The nature of this cycle illustrated in Figure 3. First, an event is put into the queue to create a sensation. Typically, the realization of this event reads the state of the world and converts this to some string of information to be sent to the agent. This string is encapsulated in a sense event and put into the event queue. MPADES requires that the time between the create sense event and the sense event is at least some minimum sense latency, which is specified by the world model. When the sense event is realized, this string will be sent to the agent to begin the thinking process. Notice that the realization of a sense event does not require the reading of any of the current world state since the string of information is fixed at the time of the realization of the create sense event. Upon the receipt of the sensation, the communication server begins timing the agent's computation. When all of the agent's actions have been received by the communication server, the computation time taken by the agent to produce those actions is converted to simulation time. All the actions and the think latency are sent to the simulation engine (shown as "Act Sent" in Figure 3). Upon receipt, the simulation engine adds the action latency

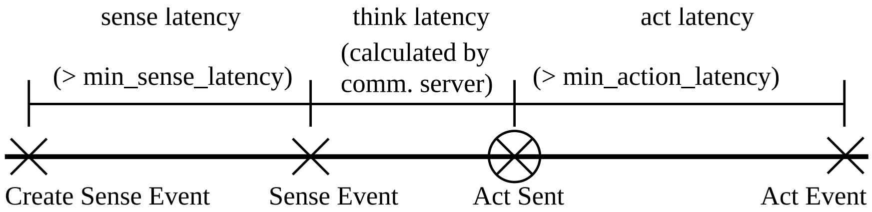

Fig. 3. The events in the sense-think-act cycle of an agent. The "Act Sent" time is circled because unlike the other marks that represent events in the queue, "Act Sent" is just a message from the communication server to the engine and not an event in the event queue.

(determined by querying the world model) and puts an act event in the pending events queue. Similar to the minimum sense latency, there is a minimum action latency which MPADES requires between the sending time of an action and the act event time. The realization of that act event is what actually causes that agent's actions to affect the world.

Note that a single agent can be have multiple sense-think-act cycles in progress at once, as illustrated in Figure 1. For example, once an agent has sent its actions (the "Act Sent" point in Figure 3), it can receive its next sensation even though the time which the actions actually affect the world (the "Act Event" point in Figure 3) has not yet occurred. The only overlap MPADES forbids is the overlapping of two think phases.

Note also that all actions have an effect at a discrete time. Therefore there is no explicit support by MPADES for supporting the modelling of the interaction of parallel actions. For example, the actions of two simulated robots may be to start driving forward. It is the world model's job to recognize when these actions interact (such as in a collision) and respond appropriately. Similarly, communication among agents is handled as any other action. The world model is responsible for providing whatever restrictions on communication desired.

The sensation and action latencies provide a lookahead value for that agents and allows the agents to think in parallel. When a sense event is realized for agent 1, it cannot cause any event to be enqueued before the current time plus the minimum action latency. Therefore it is safe (at least when only considering agent 1) to realize all events up till that time without violating event ordering.

The quantity we call the "minimum agent time" determines the maximum safe time over all agents. The minimum agent time is the earliest time which an agent can cause an event which affects other agents or the world to be put into the queue. This is similar to the Lower Bound on Timestamp (LBTS) concept used in the simulation literature. The calculation of the minimum agent time is shown in Table 2. The agent status is either "thinking," which means that a sensation has been sent to the agent and a reply has not yet been received, or "waiting," which means that the agent is waiting to hear from the simulation engine. Besides initialization, the agent status will always be thinking or waiting. The current time of an agent is the time of the last communication with the agent (sensation sent or action received). The receipt of a message from a communication server

```
calculateMinAgentTime()
    ∀i ∈ set_of_all_agents
        if (agent_i.status = Waiting) agent_time_i = ∞
        else agent_time_i = agent_i.currenttime + min_action_latency
    return min_i agent_time_i
```

Table 2. Code to determine the minimum time that an agent can affect the simulation.

```
repeat forever
    receive messages
    next_event = pending_event_queue.head
    min_agent_time = calculateMinAgentTime()
    while (next_event.time < min_agent_time)
        advanceWorldTime (next_event.time)
        pending_event_queue.remove(next_event)
        realizeEvent (next_event)
        next_event = pending_event_queue.head
        min_agent_time = calculateMinAgentTime()
```

Table 3. Main code for parallel agent discrete event simulator.

cannot cause the minimum agent time to decrease. However, the realization of an event can cause an increase or a decrease. Therefore, the minimum agent time must be recalculated after each event realization.

Based on the calculation of the minimum agent time, we can now describe a simple version of parallel agent discrete event simulator, which is shown in Table 3. The value `min_agent_time` is used to determine whether any further events can appear before the time of the next event in the queue.

While this algorithm produces correct results (all events are realized in order) and achieves some parallelism, it does not achieve the maximum amount of possible parallelism. Figure 4 illustrates an example with two agents. When the sense event for agent 1 is realized, the minimum agent time becomes A. This allows the create sense event for agent 2 to be realized and the sense event for agent 2 to be enqueued. However, the sense event for agent 2 will not be realized until the response from agent 1 is received. However, as discussed above, the effect of the realization of a sense event does not depend on the current state of the world. If agent 2 is currently waiting, there is no reason not to realize the sense event and allow both agents to be thinking simultaneously.

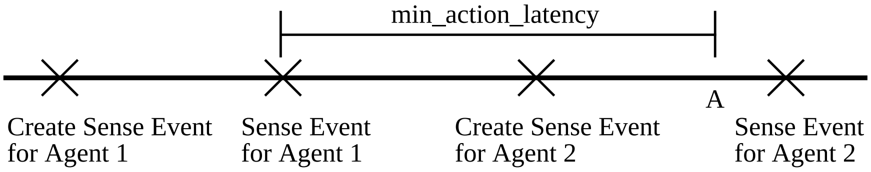

Fig. 4. An example illustrating possible parallelism that the simple parallel agent algorithm fails to exploit.

However, this allows the realization of events out of order; agent 1 can send an event which has a time less the time of the sense event for agent 2. Certain kinds of out of order realizations are acceptable (as the example illustrates). In particular, we need to verify that out of order events are not causally related. The key insight is that sensations received by agents are casually independent of sensation received by other agents. In order to state our correctness guarantees, we will define a new sub-class of events "fixed agent events" which have the following properties:

1. They do not depend on the current state of the world.
2. They affect only a single agent, possibly by sending a message to the agent.
3. Sense events and time notify events are both fixed agent events.
4. Fixed agent events are the only events which can cause the agent to start a thinking cycle, but they do *not* necessarily start a thinking cycle.

The correctness guarantees that MPADES provides are:

1. All events which are not fixed agent events are realized in time order.
2. All events which send sensations to the agents are fixed agent events.
3. The set of fixed agent events for a particular agent are realized in time order.

In order to achieve this, several new concepts are introduced. The first is the notion of the "minimum sensation time." This is the earliest time that a *new* sensation (i.e. fixed agent event) *other than a time notify* can be generated and enqueued. The current implementation of MPADES requires that the world model provide a minimum time between the create sense event and the sense event (see Figure 3), so the minimum sensation time is the current simulation time plus that time.

The time notifies are privileged events. They are handled specially because they affect no agent other than the one requesting the time notification. MPADES also allows time notifies to be requested an arbitrarily small time in the future, before even the minimum sensation time. This means that while an agent is thinking, the simulation engine cannot send any more fixed agent events to that agent without possibly causing a violation of correctness condition 3. However, if an agent is waiting (i.e. not thinking), then the first fixed agent event in the pending event queue can be sent as long as its time is before the minimum sensation time.

To insure proper event ordering, one queue of fixed agent events per agent is maintained. All fixed agent events enter this queue before being sent to the agent, and an event is put into the agent's queue only when the event's time is less than the minimum sensation time.

There are two primary functions for the agent queue. First, enqueueAgentEvent puts a fixed agent event into the queue. The doneThinking function is called when an agent finishes its think cycle. Both functions call a third function checkForReadyEvents. Psuedo-code for these functions is shown in Table 4. Note that in checkForReadyEvents, the realization of an event can cause the agent status to change from waiting to thinking.

```
checkForReadyEvents(a: Agent)
    while (true)
        if (agent_a.status = thinking)
            return
        if (agent_a.pending_agent_events.empty())
            return
        next_event = agent_a.pending_agent_events.pop()
        realizeEvent(next_event)
```

```
enqueueAgentEvent(e:Event)
    a = e.agent
    agent_a.pending_agent_events.insert(e)
    checkForReadyEvents(a)
```

```
doneThinking(a: Agent, t:time)
    agent_a.currenttime = t
    checkForReadyEvents(a)
```

Table 4. Code for maintaining the per agent fixed agent event queues

Using these functions, we describe in Table 5 the main loop that MPADES uses. This is a modification of the algorithm given in Table 3. The two key changes are that in the first while loop, fixed agent events are not realized, but are put in the agent queue instead. The second loop (the "foreach" loop) scans ahead in the event queue and moves all fixed agent events less that the minimum sensation time into the agent queues. Note that in both cases, moving events to the agent queue can cause the events to be realized (see Table 4).

6 Empirical Validation

In order to test the efficiency of the simulation and to understand the effects of the various parameters on the performance of the system, we implemented a simple world model and agents and ran a series of experiments. We tracked the wall clock time required to finish a simulation as a measure of the efficiency.

The simulated world is a two dimensional rectangle where opposite sides are connected (i.e. "wrap-around"). Each agent is a "ball" in this world. Each sensation the agent receives contains the positions of all agents in the simulation, and the only action of each agent is to request a particular velocity vector. The dynamics and movement properties are reasonable if not exactly correct for small omni-directional robots moving on carpet, except that collisions are not modelled. The world model advanced in 1ms increments.

We varied three parameters of the simulation:

- The number of machines. We used five similar Linux machines with varying speeds. While this variation in machines is undesirable, we did not have a uniform cluster available for testing. The machines were behind a firewall to control the network traffic, but the use and processing of the machines was not totally controlled. The machines used were all running Linux RedHat 5.2 with the following processors: Pentium II 450MHz, Pentium Pro 200MHz, Pentium Pro 200MHz, Pentium II 233MHz, and Pentium II 233MHz.
- The number of agents, varying from 2 to 14.

```
repeat forever
    receive messages
    next_event = pending_event_queue.head
    min_agent_time = calculateMinAgentTime()
    while (next_event.time < min_agent_time)
        advanceWorldTime (next_event.time)
        pending_event_queue.remove(next_event)
        if (next_event is a fixed agent event)
            enqueueAgentEvent(next_event)
        else
            realizeEvent (next_event)
        next_event = pending_event_queue.head
        min_agent_time = calculateMinAgentTime()
    min_sense_time = current_time + min_sense_latency
    foreach e (pending_event_queue) /* in time order */
        if (e.time > min_sense_time)
            break
        if (e is a fixed agent event)
            pending_event_queue.remove(e)
            enqueueAgentEvent(e)
```

Table 5. Code for efficient parallel agent discrete event simulator as used by MPADES

– Computation requirements of the agents. To simulate agents that do more or less processing, we put in simple delay loops (not sleeps, since we need the processes to actually require CPU time) to change the amount of computation required by the agents. We used 3 simple conditions of fast, medium, and slow agents. Fast agents simply parse the sensations and compute their new desired velocity with a some simple vector calculations. The time this takes is smaller than the resolution of CPU time reported by the Linux kernel. The medium and slow agents add a simple loop that counts to 500,000 and 5,000,000 respectively. On a 1GHz Pentium III, this translates to approximately 1.2ms and 10.6ms average response time.

Every experimental condition was run five times and the median of those five times is reported. Each simulation was run for 90 seconds of simulation time. In all experiments, the agents received sensations every 95–105 milliseconds (actual value chosen uniformly randomly after each sensation). The sensation latency and action latency was chosen uniformly randomly between 30 and 40 milliseconds for each sensation and action.

Figure 5 shows the wall clock time required to finish the simulations with various parameters. First, note that the speed of the agent gives fundamentally different trends in the parallel performance. With fast agents, distribution actually slows down the overall simulation. By distributing agents, the round trip times between the world model and the agents increase. This latency increase slows the simulation, and that effect will only be outweighed if a significant amount of computation is done in parallel on the machines. The medium speed agents clearly express this trend. With two machines, the simulation slows (usu-

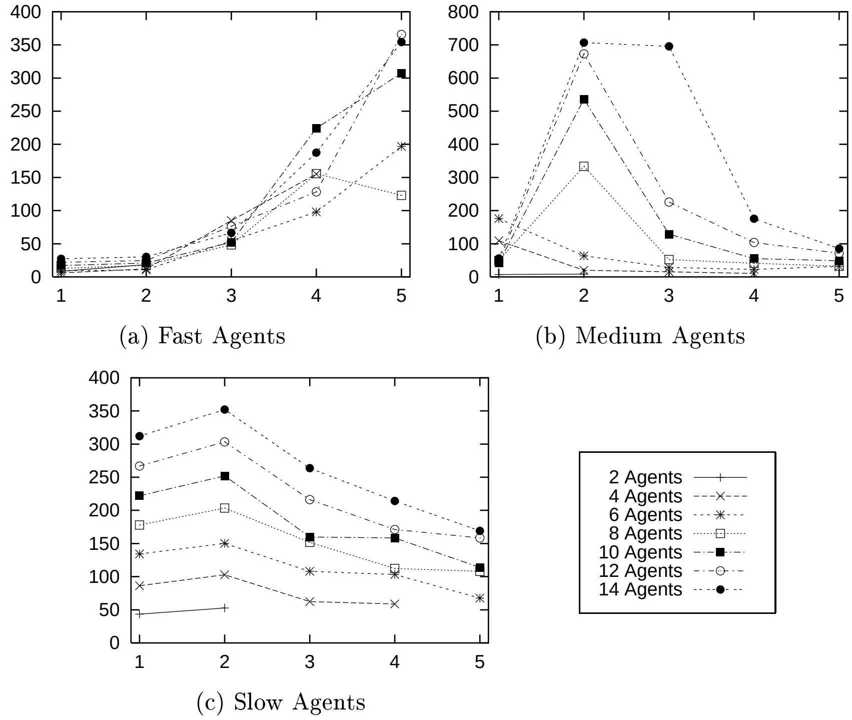

Fig. 5. Timing results with the sample world model and agents. The x-axis is the number of machines, the y-axis is the median wall clock time required to run the simulation.

ally quite significantly), only improving once the number of machines becomes large enough to balance the increased latency. The slow agents exhibit the same trend, except that parallel speedups are finally achieved with the larger amount of computation distributed.

We cannot currently adequately explain the large times required for the medium and fast speed agent simulations. We hypothesize that the network performance becomes very poor with many small packets being transmitted frequently. It should be noted that in spite of this, the simulation results are still correct.

In order to more directly compare with the SoccerServer [9], we also ran a small set of experiments with similar parameters. 22 agents were used for 90 seconds on simulation time. For the SoccerServer running at full speed, this would take 90 seconds. We timed the ChaMeleons01 agents [12] and discovered their average response time was 5ms, so we set the artificial delay in the agents to that length.

Table 6 shows the wall clock run times with the parameters similar to the SoccerServer. Note the slow speeds of the machines used here (see above), and to run this simulation on 1GHz Pentium III, the server has to be slowed down

# Machines	1	2	3	4	5
Median Wall Clock Time	446.8	317.7	231.8	190.5	164.3

Table 6. Wall clock time to simulate 90 seconds of time, with parameters similar to the SoccerServer

to 3 times slower than normal (which gives a run time of 270 seconds). Here, significant parallel speedups are achieved. There are a significant number of agents available to be distributed, and each one has reasonable computation requirements.

Even though the parameters were set similarly, there are a few important differences in these simulations. The MPADES based simulation has a step size of 1ms, where the SoccerServer has a step size of 100ms. The MPADES based simulation has a simpler world model. The agents have fewer actions (just a velocity request), where the SoccerServer has movement, ball control, communication, and internal state updating actions. The SoccerServer likely sends more data across the network because of the larger action set and especially the broadcast communication. Therefore, while these results are suggestive about the comparison in efficiency, no hard conclusions can be made.

7 Conclusion

We have presented the design of and experimental results for the MPADES system to support efficient simulation. The system supports the tracking of computation time of the agents in order to model the thinking time as part of the simulation. Unlike other AI simulators, this is accomplished in a distributed environment while being tolerant of network and machine load variations, and without requiring the agents to be implemented in a particular programming architecture or language. This provides an open agent environment where agents designed by different groups can interact.

Further, MPADES supports simulation where the smallest time step of the simulation is much smaller than the typical sensation and action latencies of the world being modelled. Agent actions do not have to be synchronized. These features allow a much closer approximation to the continuous time frame underlying most simulations.

With the implementation of a sample world model, we empirically tested the system. The results show that good parallel speedups can be achieved when the computation time of the agents is significant. Otherwise, the price paid by the network latency can slow down the overall simulation. However, we have not yet fully explored the issue of how the sensation and action latencies affect the parallel speedups.

The system does suffer from several drawbacks. The simulation engine is a centralized component and the system provides no direct support for the distribution of the simulation of the world model, only the distribution of the agents themselves. This makes it unlikely that the simulation will scale well to a large number of agents in its current form. Support for such distribution could draw on the extensive experience of other simulations, but has the potential to considerably complicate the system.

MPADES provides no structure for the design of the agents other than interface requirements. While this is a benefit in allowing a greater degree of inter-operability, it does place more of a burden on the agent designers. Also, the current implementation uses the CPU usage reported reported by the Linux kernel. Since those reports are fairly coarse and can vary with the other activity on the system, MPADES does not exhibit perfect reproducibility. There is a degree of uncontrolled randomness in the tracking of the thinking latencies of the agents. However, better tracking and reporting from the kernel could potentially alleviate these problems.

MPADES provides a solid foundation for producing high-quality agent based simulations. It handles many system and distribution details so that they can be largely ignored by the world model and agent designers, while still maintaining efficient and largely reproducible results. MPADES is a powerful new tool for creating simulations for the study of artificial intelligence.

References

1. Misra, J.: Distributed discrete-event simulation. ACM Computing Surveys **18** (1986) 39–65
2. Jefferson, D.: Virtual time. ACM Trans. Prog. Lang. and Syst. **7** (1985)
3. Ferscha, A., Tripathi, S.: Parallel and distributed simulation of discrete event systems. In Zomaya, A.Y., ed.: Parallel and Distributed Computing Handbook. McGraw-Hill (1996) 1003 – 1041
4. Fujimoto, R.M.: Parallel discrete event simulation. Communications of the ACM **33** (1990) 30–53
5. Uhrmacher, A., Gugler, K.: Distributed, parallel simulation of multiple, deliberative agents. In Bruce, D., Donatiello, L., Turner, S., eds.: Proceedings of the 14th Workshop on Parallel and Distributed Simulation (PADS 2000). (2000) 101–108
6. Uhrmacher, A.M.: Concepts of object- and agent-oriented simulation. Transactions of SCS **14** (1997) 59–67
7. Anderson, S.D.: Simulation of multiple time-pressured agents. In Andradóttir, S., Healy, K.J., Withers, D.H., Nelson, B.L., eds.: Proceedings of the 1997 Winter Simulation Conference. (1997) 397–404
8. Anderson, S.D.: A simulation substrate for real-time planning. Technical Report 95-80, University of Massachusetts at Amherst Computer Science Department (1995) (Ph.D. thesis).
9. Noda, I., Matsubara, H., Hiraki, K., Frank, I.: Soccer server: A tool for research on multiagent systems. Applied Artificial Intelligence **12** (1998) 233–250
10. Balch, T.: Behavioral Diversity in Learning Robot Teams. PhD thesis, College of Computing, Georgia Institute of Technology (1998) (available as tech report GIT-CC-98-25).
11. U.S. Department of Defense: High level architecture interface specification, version 1.3 (1998)
12. Riley, P., Carpenter, P., Kaminka, G., Veloso, M., Thayer, I., Wang, R.: ChaMeleons-01 team description. In Birk, A., Coradeschi, S., Tadokoro, S., eds.: RoboCup-2001: Robot Soccer World Cup V. Springer, Berlin (2002) (forthcoming).