Distributed Simulation of Agent-Based Systems with HLA

MICHAEL LEES and BRIAN LOGAN
University of Nottingham
and
GEORGIOS THEODOROPOULOS
University of Birmingham

In this article we describe HLA_AGENT, a tool for the distributed simulation of agent-based systems, which integrates the SIM_AGENT agent toolkit and the High Level Architecture (HLA) simulator interoperability framework. HLA_AGENT offers enhanced simulation scalability and allows interoperation with other HLA-compliant simulators, promoting simulation reuse. Using a simple Tileworld example, we show how HLA_AGENT can be used to flexibly distribute a SIM_AGENT simulation so as to exploit available computing resources. We present experimental results that illustrate the performance of HLA_AGENT on a Linux cluster running a distributed version of Tileworld and compare this with the original nondistributed SIM_AGENT version.

Categories and Subject Descriptors: I.6.5 [**Simulation and Modeling**]: Model Development; I.6.8 [**Simulation and Modeling**]: Types of Simulation—*Distributed, parallel*; I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*Intelligent agents, multiagent aystems*

General Terms: Standardization, Performance

Additional Key Words and Phrases: HLA_AGENT, high level architecture, IEEE 1516, multiagent systems

ACM Reference Format:

Lees, M., Logan, B., and Theodoropoulos, G. 2007. Distributed simulation of agent-based systems with HLA. ACM Trans. Model. Comput. Simul. 17, 3, Article 11 (July 2007), 25 pages. DOI = 10.1145/1243991.1243992 http://doi.acm.org/10.1145/1243991.1243992

1. INTRODUCTION

There has been considerable recent interest in *agent-based systems*, systems based on autonomous software and/or hardware components (agents), which

This work is part of the PDES-MAS project (http://www.cs.bham.ac.uk/research/pdesmas) and was partially supported by EPSRC research grant No. GR/R45338/01.
Authors' addresses: M. Lees and B. Logan, School of Computer Science and IT, University of Nottingham, Nottingham, UK; email: mhl@cs.nott.ac.uk; G. Theodoropoulos, School of Computer Science, University of Birmingham, Birmingham, UK.

cooperate within an environment to perform some task. An *agent* can be viewed as a self-contained, concurrently executing thread of control that encapsulates some state and communicates with its environment, and possibly other agents, via some sort of message passing [Wooldridge and Jennings 1995]. Agent-based systems offer advantages when independently developed components must interoperate in a heterogeneous environment, for example, the INTERNET, and these systems are increasingly being applied in a wide range of areas including telecommunications, the semantic web, business process modelling, control of mobile robots and military simulations [Bradshaw 1997; Jennings and Wooldridge 1998; Luck et al. 2005].

However, the adoption of multi-agent systems (MAS) has been hampered by the limitations of current development tools and methodologies. Multi-agent systems are often extremely complex, so it can be difficult to formally verify their properties. As a result, design and implementation remains largely experimental, and experimental approaches are likely to remain important for the foreseeable future. In this context, simulation has a key role to play in the design and analysis of agent architectures and systems. The use of simulation allows a degree of control over experimental conditions and facilitates the replication of results in a way that is difficult or impossible with a prototype or fielded system. This allows the agent designer or researcher to focus on key aspects of the system. Over the last two decades, a wide range of MAS simulators and testbeds have been developed [Durfee and Montgomery 1989; Pollack and Ringuette 1990; Atkin et al. 1998; Sloman and Poli 1996; Anderson 2000; Schattenberg and Uhrmacher 2001; Riley and Riley 2003], and simulation has been applied to a wide range of MAS research and design problems, from models of complex individual agents employing sophisticated internal mechanisms to models of large-scale societies of relatively simple agents that focus more on the interactions between agents.

However, existing MAS simulations and simulators suffer from two key problems: lack of interoperability and lack of performance. The effort required to develop a new simulation from scratch is considerable. There is therefore a strong incentive to reuse existing simulation components, toolkits and testbeds for a new problem. However, while many simulations (and simulators) have been developed, it is difficult to leverage this investment in the development of new agent simulations. Simulations developed for different agent simulators typically do not interoperate, making it more difficult to reuse simulation components. Combining a simulation of an agent architecture developed for one simulator with a simulation of an environment developed for another, typically involves reimplementation of one or both components. If the agent must be simulated in several different environments, the problem is compounded. In addition, the computational requirements of simulations of many multiagent systems far exceed the capabilities of a single computer. Each agent may be a complex system in its own right (e.g., with sensing, planning, inference etc. capabilities), requiring considerable computational resources, and many agents may be required to investigate the behavior of the system as a whole, or even the behavior of a single agent.

A solution to both of these problems can be found in distributed simulation. Distributed simulation approaches exploit the natural parallelism of MAS, allowing simulation components to be distributed so as to make the best use of available computational resources. The last decade has also witnessed an increasing interest in distributed simulation not only for speeding up simulations, but also as a strategic technology for linking simulation components of various types at multiple locations to create a common virtual environment. The culmination of this activity (which originated in military applications where battle scenarios were formed by connecting geographically distributed simulators via protocols such as the Distributed Interactive Simulation (DIS) protocol) has been the development of the High Level Architecture (HLA), a framework for simulation reuse and interoperability developed by the US Defence Modelling and Simulation Office [Kuhl et al. 1999]. Using HLA, a large-scale distributed simulation can be constructed by linking together a number of geographically distributed simulation components, or *federates*, into an overall simulation, or *federation*. HLA, with minor revisions, has been adopted as an IEEE standard (IEEE 1516) [IEEE 2000] and is likely to be increasingly widely adopted within the simulation community.

The HLA offers an attractive potential solution to the problems of simulation and simulator reuse, and simulation performance, in MAS simulation. The development of HLA-compliant agent simulators and simulation components would facilitate interoperation with other simulations, allowing greater reuse of agent simulation components. In addition, the ability to distribute agent and other simulation components across multiple computers has the potential to increase the overall performance of a MAS simulation. The price of enhanced modularity and interoperability is often increased computational overheads. Distribution offers a way to offset these increased costs, and, given sufficient computational resources and favorable simulation characteristics, to achieve significant reductions in simulation execution time. However agent simulations present particular challenges for distributed simulation. Simulations of situated MAS often have a large shared state, which may be accessed by any of the agents in the simulation [Logan and Theodoropoulos 2001]. In addition, MAS simulations often consist of many relatively lightweight components that access the shared state frequently, relative to the CPU they consume. As a result, it is not clear if the overheads entailed by the adoption of the HLA for MAS simulation can be offset by distribution, and it may be that the benefits of the HLA in terms of ease of simulation development are more than outweighed by the costs of interoperation and/or distribution.

In this article, we investigate the feasibility of the HLA for MAS simulation. We present HLA_AGENT, an HLA-compliant implementation of a legacy agent simulator, SIM_AGENT [Sloman and Poli 1996]. A major goal in the development of HLA_AGENT was backwards compatibility with existing SIM_AGENT simulations. Existing SIM_AGENT simulations run unchanged on HLA_AGENT. By providing additional information specifying which simulation entities are to be simulated by each federate, an agent simulation developer is able to flexibly distribute a legacy SIM_AGENT simulation so as to make best use of available computational resources. Using HLA_AGENT as an example, we illustrate how the

particular challenges of agent simulation can be addressed within the framework of the HLA, and attempt to quantify the overheads of HLA for agent simulation. We show that, even for relatively lightweight agents interacting in a shared environment, it is possible to achieve speedup by distributing the simulation components. The implementation has not been optimized, and we have deliberately focused on relatively simple agent simulations, which have relatively low computational demands for both the agents and their environment. We can therefore be more confident that our conclusions will hold for a wide range of agent simulators and simulations with greater computational requirements.

The rest of the article is organized as follows. In Section 2 we briefly outline some of the key challenges in agent simulation. In Section 3 we briefly describe the SIM_AGENT toolkit and illustrate its application in a simple agent simulation benchmark, Tileworld [Pollack and Ringuette 1990]. In Section 4 we outline how the HLA can be used to distribute an existing SIM_AGENT simulation with different agents being simulated on different federates, and in Section 5 we sketch the changes necessary to the SIM_AGENT toolkit to allow integration with the HLA. In Section 6 we present experimental results to illustrate the performance of HLA_AGENT on a Linux cluster running a distributed version of Tileworld and compare this with the original, nondistributed, SIM_AGENT version of Tileworld. We conclude with a brief description of future work.

2. SIMULATING MULTIAGENT SYSTEMS

In this section we briefly outline the particular challenges in the distributed simulation of multiagent systems.

The notion of an 'agent', though intuitive, is very general. The conventional definition of an agent as a self-contained, concurrently executing thread of control that encapsulates some state and communicates with its environment, and possibly other agents, via some sort of message passing [Wooldridge and Jennings 1995] covers a wide range of software entities. From the point of view of simulation, one of the key aspects of multiagent systems is the way in which agents interact. Agents are embedded in an *environment*—that part of the world or computational system "inhabited" by the agents. The agents sense and act within the environment, which forms a medium of interaction between the agents. As with the notion of 'agent,' there are many different kinds of agent environments. At one extreme, the environment may simply consist of the computational system or network in which the agents are executing. In this case, "sensing" reduces to the arrival of a message (or perhaps monitoring some aspect of the underlying computational system, such as the current CPU load), and "acting" is simply the sending of messages or invoking a Web service. At the other extreme, agents may be embedded in complex environments that they sense via multiple sensory modalities and that support a rich repertoire of actions. The modelling of the environment may itself involve complex physical simulations (e.g., detailed kinematic models), models of sensor and effector noise, and autonomous processes, which continuously update the state of the environment (e.g., weather).

In what follows, we shall focus primarily on the simulation of *situated agents* [Ferber 1999], for example, simulations of agents such as robots situated in a physical environment, or characters in a computer game or interactive entertainment situated in a virtual environment, as these systems present a more challenging problem from the simulation point of view. Situated agents are typically viewed as repeatedly executing a simple *sense—think—act* cycle. Information obtained from the environment via sensing is used, together with the agent's beliefs and goals, to choose one or more actions, which are then executed. The cycle then repeats. The systems of interest typically involve large numbers (thousands or tens of thousands) of agents in complex environments, for example, individual-based ecological modelling or simulations of massively multiplayer online games.

In such simulations, the environment may be *passive*, a simple datastructure recording the public attributes of objects and agents that form the environment, for example, the color, size, shape, position and so on, of an object or agent, and updated directly by the agents, or *active*—managed by one or more 'environment agents,' which are responsible for computing the consequences of the action(s) of the agents, and updating the environment accordingly. Both agents and environment agents (if the environment is *active*) typically also have private data that is not accessible to other agents in the simulation. For example, an agent may have beliefs about the current state of the environment (which may be incorrect), goals representing states of the environment it would like to bring about, and so on, together with information about that part of the environment it can immediately sense and its own internal state.

In the next section we illustrate some of these ideas in more detail, taking the SIM_AGENT toolkit as our example.

3. AN OVERVIEW OF SIM_AGENT

SIM_AGENT is an architecture-neutral toolkit originally developed to support the exploration of alternative agent architectures [Sloman and Poli 1996; Sloman and Logan 1999].[1] It can be used both as a sequential, centralized, time-driven simulator for multiagent systems, for example, to simulate software agents in an Internet environment, or physical agents and their environment, and as an agent implementation language, for example, for software agents or the controller for a physical robot. SIM_AGENT supports both situated and nonsituated agents, and it has been used in a variety of research and applied projects, including studies of affective and deliberative control in simple agent systems [Scheutz and Logan 2001], agents that report on activities in collaborative virtual environments [Logan et al. 2002] and computer games Fielding et al. [2004, 2006] (which involved integrating SIM_AGENT with the MASSIVE-3 VR system and Unreal Tournament respectively), and simulation of tank commanders in military training simulations [Baxter and Hepplewhite 1999] (for this project, SIM_AGENT was integrated with an existing real time military simulation).

[1]See http://www.cs.bham.ac.uk/~axs/cog_affect/sim_agent.html.

In SIM_AGENT, an agent consists of a collection of modules representing the capabilities of the agent, for example, perception, problem-solving, planning, communication and so on. Groups of modules can execute either sequentially or concurrently and with differing resource limits. Each module is implemented as a collection of rules, or ruleset, in a high-level rule-based language called POPRULEBASE. The rule format is very flexible: both the conditions and actions of rules can invoke arbitrary low-level capabilities, allowing the construction of hybrid architectures including, for example, symbolic mechanisms communicating with neural nets and modules implemented in procedural languages. The rulesets that implement each module, together with any associated procedural code, constitute the *rulesystem* of an agent. SIM_AGENT can also be used to simulate the agent's environment, and the toolkit provides facilities to populate the agent's environment with user-defined active and passive objects simulated by one or more environment agents. Within a SIM_AGENT simulation, each object or agent has both external and internal data. The internal data can be thought of as the agent's private state. External data is data that is externally visible to other agents and objects in the environment, for example, color, size, shape and so on.

Simulation proceeds in three logical phases: sensing, internal processing and action execution, where the internal processing may include a variety of logically concurrent activities, for example, perceptual processing, motive generation, planning, decision making, learning and so on.

In the first, sensing, phase each agent's internal database is updated according to what it senses and any messages sent at the previous cycle. For example, if an agent's sensors are able to see all objects within a predefined distance, the internal database of the agent would be updated to contain facts that indicate the visible attributes of all objects that are closer than the sensor range. The agent's database can be thought of as its working memory, which holds the agent's model of the environment, its current goals, plans and so on.

The next, internal processing, phase involves decision making and action selection. The contents of the agent's database are matched against the conditions of the condition-action rules that constitute the agent's rulesystem. Typically, the conditions of more than one rule are satisfied, or the same rule is satisfied multiple times. SIM_AGENT allows the programmer to choose how these rules should run and in what order. For example a given simulation may require that only the first rule matched runs, or that every satisfied rule should run. It is also possible to build a list of all the runnable rules and have a user-defined procedure order this list so that only certain rules (e.g., the more important rules) are run or are run first. These rules will typically cause some internal and/or external action(s) to be performed or message(s) to be sent. Internal actions simply update the agent's database and are performed immediately. External actions change the state of the environment and are queued for execution in the third phase.

The final phase involves sending the messages and performing the actions queued in the previous phase. By definition, external actions change external properties of the agent and/or objects in the environment (e.g., change its location) and these changes can be sensed at the next cycle.

The three logical phases are actually implemented as two scheduler passes for reasons of efficiency. In the first pass, the scheduler runs each agent's sensors and rulesystem. Any external actions or messages generated by the agent in this pass are queued. In the second pass, the scheduler processes the message and action queues for each agent, transferring the messages to the input message buffers of the recipient(s) for processing at the next cycle, and running the actions to update the environment and/or the publicly visible attributes of the agent. While all agents see the same state of the environment at any given timestep, by default, SIM_AGENT makes no attempt to resolve action conflicts. Instead, updates are applied in an arbitrary but reproducible order.

SIM_AGENT provides a library of classes and methods for implementing agent simulations. The toolkit is implemented in Pop-11, an AI programming language similar to Lisp, but with an Algol-like syntax. Pop-11 supports object-oriented development via the OBJECTCLASS library, which provides classes, methods, multiple inheritance, and generic functions.[2] SIM_AGENT defines two basic classes, `sim_object` and `sim_agent`, which can be extended (subclassed) to give the objects and agents required for a particular simulation scenario. The `sim_object` class is the foundation of all SIM_AGENT simulations: it provides slots (fields or instance variables) for the object's name, internal database, sensors, and rule system together with slots that determine how often the object will be run at each timestep, how many processing cycles it will be allocated on each pass and so on. The `sim_agent` class is a subclass of `sim_object` which provides simple message based communication primitives. SIM_AGENT assumes that all the objects in a simulation will be subclasses of `sim_object` or `sim_agent`.

3.1 An Example: SIM_TITLEWORLD

In this section we briefly outline the design and implementation of a simple SIM_AGENT simulation, SIM_TILEWORLD. Tileworld is a well established testbed for agents [Pollack and Ringuette 1990]. It is an environment consisting of tiles, holes, and obstacles, and an agent whose goal is to score as many points as possible by pushing tiles to fill in the holes. The environment is dynamic: tiles, holes, and obstacles appear and disappear at rates controlled by the simulation developer. Tileworld has been used to study commitment strategies (when an agent should abandon its current goal and replan) [Pollack et al. 1994] and in comparisons of reactive and deliberative agent architectures [Pollack and Ringuette 1990]. SIM_TILEWORLD is an implementation of a multiagent Tileworld [Ephrati et al. 1995], which consists of an environment and one or more agents (see Figure 1). The environment is simulated by one or more environment agents, each of which is responsible for simulating the tiles, holes, and obstacles in a rectangular, nonoverlapping region of the Tileworld.

For the SIM_TILEWORLD example, three subclasses of the SIM_AGENT base class `sim_object` were defined to represent holes, tiles, and obstacles, together with two subclasses of `sim_agent` to represent the environment and the Tileworld agents. The subclasses define additional slots to hold the relevant simulation attributes, for example, the position of tiles, holes, and obstacles, the

[2]OBJECTCLASS shares many features of the Common Lisp Object System (CLOS).

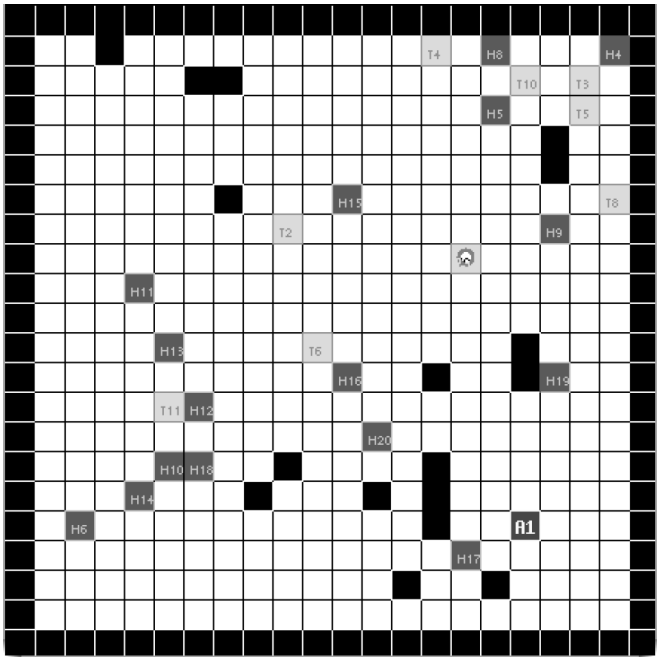

Fig. 1. A screen shot of SIM_TILEWORLD.

types of tiles, the depth of holes, the tiles being pushed by the agent and so on. By convention, external data is held in slots, while internal data (such as which hole the agent intends to fill next) is held in the agent's database.

The simulation consists of two or more active objects (the environment and the Tileworld agent(s)) and a variable number of passive objects (the tiles, holes, and obstacles). At simulation startup, instances of the environment and agent classes are created and passed to the scheduler. At each cycle the scheduler runs the environment agent(s) to update the Tileworld agents' environment. In SIM_TILEWORLD the environment agent(s) have a simple rule system with no conditions (it runs every cycle), which causes tiles, obstacles, and holes to be created and deleted according to user-defined probabilities. The scheduler then runs the Tileworld agents, which perceive the new environment, and updates their internal databases with the new sense data. The sensors of an agent are defined by a list of procedures and methods. Any object in the simulation objects list which 'satisfies' these procedures or methods (in the sense of being an appropriate method for the object class in the case of methods, or returning sensor data in the case of procedures) is considered 'sensed' by the agent. The agents then run all rules that have their conditions satisfied (no ordering of the rules is performed). Some of the rules may queue external actions (e.g., moving to, or pushing, a tile) that are performed in the second pass of the scheduler at this cycle. This completes the cycle, and the process is repeated.

4. DISTRIBUTING A SIM_AGENT SIMULATION WITH HLA

As indicated in Section 1, we can identify two main roles for the HLA in connection with a MAS simulation such as SIM_AGENT. The first, which we call the *distribution* of SIM_AGENT, involves using HLA to distribute the agents and objects comprising a SIM_AGENT simulation across a number of federates. The second, which we call *interoperation*, involves using HLA to integrate SIM_AGENT with other simulators. As our concern is primarily with the computational overhead of HLA for situated MAS simulations, in the remainder of this paper we concentrate on the former, namely distributing an existing SIM_AGENT simulation using SIM_TILEWORLD as an example.

4.1 The High Level Architecture

The High Level Architecture (HLA) allows different simulations, referred to as *federates*, to be combined into a single larger simulation known as a *federation* [DMSO 1998; IEEE 2000]. The federates may be written in different languages and may run on different machines. A federation is made up of:

—one or more federates;

—a Federation Object Model (FOM);

—the Runtime Infrastructure (RTI).

Each federate can model a single entity, such as an agent, or a number of entities, or it may have some other purpose. For example, a federate might be a data logger or a viewer used to 'steer' a simulation, or it might act as a surrogate for a human participant in a simulation, reflecting the state of the larger simulation to some user interface and conveying decisions from the participant to the rest of the simulation.

The FOM defines the types of and the relationships between the data exchanged by the federates in a federation. Each FOM consists of a set of object classes and a set of interaction classes. Each object class defines a possibly empty set of named data, called attributes. Instances of these object classes and their associated attribute values are created by the federates to define the persistent state of the simulation. Federates evolve the state of an object instance in simulation time by supplying new values for its attributes. An interaction is a set of named data, called parameters, which forms a logical unit within the federation, for example, an event within the simulation model. The data comprising an interaction is sent as a unit by a federate to the other federates in the federation. Unlike objects, interactions have no continued existence after they have been received. Object and interaction classes are organized into separate inheritance hierarchies, in which each class inherits the attributes (for objects) or parameters (for interactions) of its superclasses. Each federate must typically translate from its internal notion of simulated entities to HLA objects and interactions as specified in the FOM. The structure of all FOMs is defined by the Object Model Template (OMT), which ensures federations can communicate with one another.

The RTI is middleware that provides common services to the federates. All communication between the federates in a federation, and between federations,

is accomplished via the RTI. Each federate contains an RTI Ambassador and a Federate Ambassador along with the user simulation code. The RTI Ambassador handles all outgoing information passed from the user simulation to the RTI. Each call made by the RTI Ambassador typically results in a corresponding callback on other federates. For example, updating the value of an attribute of an instance of an object class defined in the FOM on one federate will result in a callback containing the new value on federates which subscribe to the attribute. It is the task of the Federate Ambassador to handle these callbacks and invoke appropriate code in the user simulation, for example, update a the value of a field or variable representing the attribute. The FOM is passed to the RTI at the beginning of an execution and effectively parameterizes the RTI for that federation. The FOM is supplied as data to the RTI at the beginning of an execution.

Based on the SIM_TILEWORLD implementation outlined in Section 3.1, we choose to distribute the simulation across n agent federates and a single environment federate.

The HLA provides services in six areas, namely Federation Management, Object Management, Declaration Management, Ownership Management, Time Management, and Data Distribution Management. In the remainder of this section, we illustrate the role of these services in distributing a SIM_TILEWORLD simulation.[3]

4.2 Object and Declaration Management

In the HLA, information about objects in the simulation is not held centrally; rather each federate is responsible for maintaining its own local information about objects of interest simulated by other federates. A federate declares its interest in objects and attributes at the beginning of a simulation by *publishing* any attributes it may update during the simulation and *subscribing* to attributes for which it would like to receive updates. A federate that is subscribed to an attribute of a certain class will also be informed of any instances of this class created or deleted by another federate.

In our SIM_TILEWORLD federation, communication between the federates updates the persistent state of the simulation (the positions of the Tileworld agents and the configuration of the objects in the enviroment), and is most naturally performed through the creation and deletion of object instances, and the updating of their attributes.[4] We define two main classes of simulation objects: *Agent* and *Object*, with the *Object* class having *Tiles*, *Holes*, and *Obstacles* as subclasses. The attributes are those relevant to sensing and acting in the Tileworld, for example, the positions of the agents, tiles, holes, and obstacles, the type and depth of holes and so on. The classes and attributes in the FOM can be mapped in a straightforward way onto the classes and slots used by SIM_AGENT. For example, the *depth* attribute of the *Tile* class in the FOM maps to the `depth` slot of the `sim_tile` class in SIM_AGENT.

[3]We do not consider Federation Management in HLA_AGENT, as this is similar to other HLA federations.

[4]In this example, SIM_AGENT's interagent message based communication is not used. Message passing is also handled by the RTI, using interactions.

Table I. Attribute Publications and Subscriptions

		Federate	
Object	Attribute	Environment	Agent
Agent	*privilegeToDelete*	publish	publish
	position	subscribe	publish
Tile	*privilegeToDelete*	publish	publish
	position	publish	publish
	life	publish	subscribe
	type	publish	subscribe

Table I illustrates the publications and subscriptions for the *Agent* and *Tile* classes in the Tileworld FOM. The *position* attribute of the *Agent* class is published by the agent federate(s) as these federates update the position of the agents they simulate, for example, when the agents move in the Tileworld. The environment federate subscribes to the *position* attribute of the *Agent* class. It is therefore informed of changes in the agents' positions, but is unable to update the position of any agent. In contrast, the *position* attribute of the *Tile* class is published by both the environment and the agent federates. This is because initially, when a tile is created, the environment federate will set the tile's position. However, when an agent (simulated by an agent federate) pushes the tile, it will start to update the *position* attribute of the appropriate instance of the *Tile* class. *privilegeToDelete* is a special attribute, predefined by the HLA: a federate that owns the *privilegeToDelete* attribute of an object can delete the object from the simulation.

4.3 Ownership Management

In addition to publishing an attribute of an object class, for a federate to be able to update the value of a particular attribute instance, the HLA requires that the federate *own* the attribute instance. Ownership of an attribute indicates that the owning federate is responsible for simulating the corresponding property of the object instance and can be transferred from one federate to another during federation execution.

In HLA_AGENT we use ownership management to manage conflicts between actions performed by agents simulated by different federates. In SIM_AGENT, conflicting actions result in the updates associated with a single arbitrarily chosen action being performed. By requiring that attribute ownership is only transferred at most once per simulation cycle, and that a federate relinquishes ownership of an attribute only if it has not already been updated at the current cycle, we can achieve the same effect in a distributed setting.[5] (If multiple attributes are updated by the same agent action, we require that federates acquire ownership of the attributes in a fixed order to avoid deadlock.) For example, if two agents running on different federates try to move a given tile at the same cycle, whichever agent's action is processed first will acquire ownership of the tile and succeed, while the other will be denied ownership and fail.

[5]Note that while the action performed is arbitrary in both cases, in the distributed setting the action chosen will vary from run to run, as it depends on the real time at which federates request ownership of the attributes.

4.4 Time Management

Time Management services coordinate the advancement of logical time within the federation and ensure that time-stamped data is delivered to each federate at the appropriate logical time. Federates can be *time-regulating* or *time-constrained* or both. A time-regulating federate is one whose advance of logical time regulates the rest of the federation (specifically those federates that are time-constrained). A time-constrained federate is one whose advance of logical time is constrained by the rest of the federation (specifically those federates that are time regulating).

SIM_AGENT is a centralized, time-driven system where simulation advances in timesteps, or cycles. At the end of a cycle external actions may change aspects of the simulation. These changes are then perceived by all agents at the beginning of the next cycle. We therefore synchronize the HLA_AGENT federation at the end of each cycle, by making all the federates both time-regulating and time-constrained. This ensures that all federates advance logical time at the same rate, alternating between performing external actions and perceiving changes.

4.5 Data Distribution Management

The aim of data distribution management (DDM) in HLA is to limit the amount of data exchanged between the federates in the simulation. This is achieved through the specification of publication and subscription regions in *routing spaces*, with each region implicitly defining an interconnection pattern between federates, and the assignment of multicast groups to these regions. Due to the complexity of router configuration and the limited availability of multicast groups, the assignment of multicast groups is static and is based on a priori knowledge of the federates' interconnection patterns [Morse and Zyda 2000]. However, in complex agent-based systems it is often difficult, if not impossible, to determine an appropriate simulation topology a priori; static interest management schemes are therefore less effective in agent-based simulations than in other application domains [Logan and Theodoropoulos 2001]. Various efforts have been made to define alternative dynamic schemes for DDM, e.g., [Morse and Zyda 2000], and in Logan and Theodoropoulos [2001] we have proposed an approach that combines dynamic DDM and load balancing for the simulation of agent-based systems.

While DDM would reduce the amount of broadcast communication in some HLA_AGENT simulations it has not been utilized in the integration exercise described in this article. In the current version of HLA_AGENT, federates send information to one another based on the publication and subscription information and it is up to the individual agents to filter the relevant information (e.g.,, based on sensor range). In Section 6 we present some results that indicate the effect the reduction in broadcast communication with DDM could have on the performance of the simulation.

5. EXTENDING SIM_AGENT

In this section we briefly sketch the extensions necessary to the SIM_AGENT toolkit to allow an existing SIM_AGENT simulation to be distributed using the HLA.

Together, the extensions constitute a new library, which we call HLA_AGENT. Our aim was to allow maximum flexibility in the distribution of the user simulation. Existing user simulation code runs unchanged, with the user providing additional information specifying the number of federates in the federation and how the objects and agents in the simulation are to be assigned to federates so as to make best use of available computing resources. The distribution of the user simulation is also symmetric in the sense that no additional management federates are required.

In what follows, we assume that we have an existing SIM_AGENT simulation (e.g., SIM_TILEWORLD) that we want to distribute by placing disjoint subsets of the objects and agents comprising the simulation on different federates. Each federate corresponds to a single SIM_AGENT process and is responsible both for simulating the local objects forming its own part of the global simulation, and for maintaining proxy objects that represent objects of interest being simulated by other federates. Each federate may be initialised with part of the total model or all federates can run the same basic simulation code and use additional information supplied by the user to determine which objects are to be simulated locally. For example, in SIM_TILEWORLD we may wish to simulate the agent(s) on one federate and the environment on another. We assume that the class definitions for the objects and agents comprising the simulation are available to all federates that publish and subscribe to the corresponding FOM classes and attributes, and that all federates can create instances of these classes to represent agents being simulated by the federate and as proxies for agents being simulated by other federates.

The overall organization of HLA_AGENT is illustrated in Figure 2. Each SIM_AGENT federate requires two ambassadors: an RTI Ambassador, which handles calls to the RTI and a Federate Ambassador, which handles callbacks from the RTI. Calls to the RTI are processed asynchronously in a separate thread. However, for simplicity, we have chosen to queue callbacks from the RTI to the Federate Ambassador for processing at the end of each simulation cycle. SIM_AGENT has the ability to call external C functions. We have therefore utilized the reference implementation of the RTI written in C++ developed by DMSO, and defined C wrappers for the RTI and Federate Ambassador methods needed for the implementation. We use Pop-11's simple serialization mechanism to handle translation of SIM_AGENT data structures to and from the byte strings required by the RTI. All RTI calls and processing of Federate Ambassador callbacks can therefore be handled from SIM_AGENT as though we have an implementation of the RTI written in Pop-11.

In the remainder of this section, we briefly describe the changes to SIM_AGENT in more detail. It turns out that the changes to the scheduler are confined to the first (sensing) and third (action) phases. The second phase involves only the internal operation of the agent and updates to the agent's private database. Such updates are typically invisible to other agents, and can be ignored for purposes of distribution.[6]

[6]We have not considered the distribution of the components of a single agent across multiple federates.

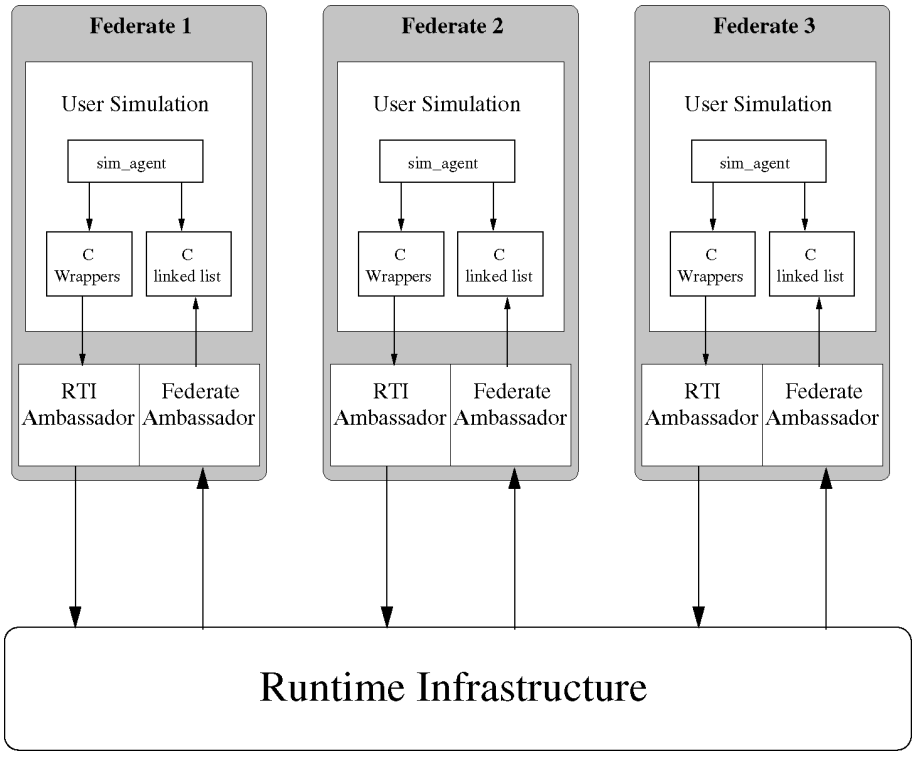

Fig. 2. The Organization of HLA_AGENT.

5.1 Creating and Deleting Objects

The creation and deletion of objects and agents by a particular federate (SIM_AGENT process) must be propagated to the other federates in the simulation that subscribe to the corresponding class in the FOM. OBJECTCLASS provides *wrappers*, closures around existing methods that extend or even replace the functionality of the method. We extend the existing SIM_AGENT code with 'new' and 'destroy' wrappers, which intercept calls to class constructors and destructors respectively. When a new object is created by the user simulation, the new wrapper registers it with the RTI, triggering object discovery callbacks and proxy creation on other federates. A similar pattern is used with object deletions. When an object in the user simulation becomes garbage, the 'destroy' wrapper is run. The wrapper 'undeletes' the object by creating a new, non-weak, reference to it, ensuring that the object persists until the end of the current simulation cycle and allows the object to be deleted on all federates at the same time. If the federate does not own the object being deleted (e.g., if it is a proxy), the 'destroy' wrapper also negotiates with the federate that owns the object for permission to delete it. By convention, in HLA_AGENT, a federate will always grant permission to delete an object unless it intends to delete the object itself at this cycle or has already given permission to another federate. Requesting permission to delete an object is therefore sufficient to ensure that the object will be deleted.

5.2 Propagating the Effects of External Actions

As stated in Section 3, agents can perform two different types of actions: internal actions that update the agent's private database, and external actions that update publicly visible attributes of an object. Internal actions only affect the state of the agent and are processed immediately, since the effects of the action (changes to the contents of the agent's database or working memory) typically form part of a larger decision-making process within the agent. However, in the case of external actions, it is necessary to propagate the update to other federates that subscribe to the attribute. This involves calls to the RTI to acquire ownership of the attribute (if it is not currently owned by this federate) and to do the update.

In SIM_AGENT, attribute instances are represented by slot values. Each slot has two predefined methods, an *accessor*, which returns the current value, and an *updater*, which sets the value. In HLA_AGENT, we use 'update' wrappers to "intercept" calls to the slot updater methods and propagate the new value to other federates by making the appropriate RTI calls.

The wrapper for a slot updater first checks to see if the slot corresponds to an attribute of a class in the FOM that is published by the federate. If so, the federate requests the RTI to propagate the update to other federates that have subscribed to this attribute. If the attribute instance is not owned by the federate, this involves requesting ownership of the attribute instance from the owning federate. Updates to slots not corresponding to published attributes are assumed to be local to the federate and are not propagated.

To avoid multiple updates to the same attribute at the same cycle being received out of order by other federates, updates to attributes owned by a federate are queued until all the updates have been processed, and only the last update to each attribute is propagated to the RTI. (This is necessary, for example, during object creation to avoid values set by superclass initializers overwriting a value set by a class initializer.)[7] Updates to attributes not owned by a federate are further delayed until ownership of the attribute is transferred to the federate. If the federate is unable to obtain ownership of the attribute—if the attribute has already been updated by another federate at this cycle—then the local update is discarded. Agents on all federates therefore always see the same simulation state.

5.3 Partitioning the User Simulation

To distribute a simulation, a user defines the classes and attributes that constitute the federation object model and, for each federate, provides a mapping between the classes and attributes in the FOM and the SIM_AGENT classes and slots to be simulated on that federate. If the user simulation is partitioned so that each federate only creates instances of those objects and agents it is responsible for simulating, then no additional user-level code is required. In the case in which all federates use the same user simulation code, the user must

[7]It also solves the problem that two agents running on the same federate can update the attribute, since ownership is per federate, not per agent.

define a procedure that is used to determine whether an object should be simulated on the current federate. The user therefore has the option of partitioning the simulation into appropriate subsets for each federate, thereby minimizing the number of proxy objects created by each federate at simulation startup, or allowing all federates to create a proxy for all nonlocal objects in the simulation. For very large simulations, the latter approach may entail an unacceptable performance penalty, but has the advantage that distributed and nondistributed simulations can use identical code.

6. EXPERIMENTAL RESULTS

To evaluate the performance of HLA_AGENT and the overhead of HLA for MAS simulations, we implemented a distributed version of SIM_TILEWORLD using HLA_AGENT, and compared its performance with the original, nondistributed SIM_AGENT version.

The hardware platform used for our experiments is a Linux cluster, comprising 64 2.6 GHz Xeon processors each with 512KB cache (32 dual nodes) interconnected by a standard 100 Mbps fast Ethernet switch.[8] Our test environment is a Tileworld 50 units by 50 units in size with an object creation probability (for tiles, holes, and obstacles) of 1.0 and an average object lifetime of 100 cycles: objects are created and destroyed at approximately the same rate. The Tileworld initially contains 100 tiles, 100 holes and 100 obstacles and the number of agents in the Tileworld ranges from 1 to 64. In the SIM_TILEWORLD federation, the environment was simulated by a single environment agent running on a single environment federate while the Tileworld agents were distributed in one or more federates over the nodes of the cluster.[9] The results obtained represent averages over 5 runs of 100 SIM_AGENT cycles.

We would expect to see speedup from distribution in cases where the CPU load dominates the communication overhead entailed by distribution. We therefore investigated two scenarios: simple reactive Tileworld agents with minimal CPU requirements and deliberative Tileworld agents that use an A^* based planner to plan optimal routes to tiles and holes in their environment [Logan and Alechina 1998]. The planner was modified to incorporate a variable "deliberation penalty" for each plan generated, and the agents replanned whenever the region of the Tileworld they could sense was changed by the actions of another agent or by the environment itself. In the following experiments the deliberation penalty was arbitrarily set at 10 ms per plan. This is towards the lower end of the deliberation times reported in the MAS literature. For example, the results reported in Riley [2003] are for agents with cycle times in the range 95–105 milliseconds, and Schattenberg and Uhrmacher [2001] report experiments with planning agents that require from 2 seconds to 20 hours of CPU time per cycle.

For comparison, Figure 3 shows the total elapsed times when executing 1, 2, 4, 8, 16, 32, and 64 reactive and deliberative SIM_TILEWORLD agents and their

[8]For our experiments, only one processor was used in each node.

[9]Note that this is not required, and the environment can be decomposed into multiple federates, each responsible for simulating for example, a region of the Tileworld. Such decomposition can partially alleviate the bottleneck of a single environment federate.

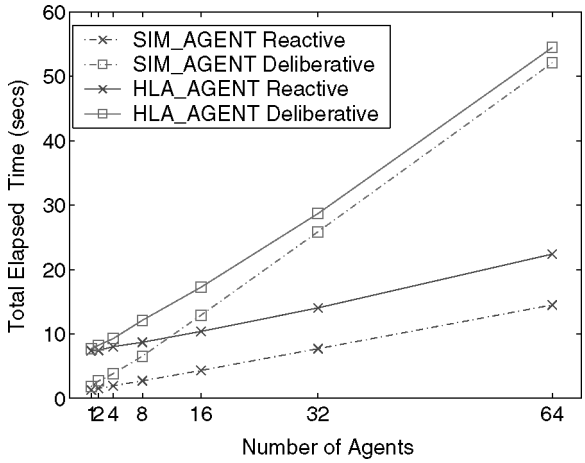

Fig. 3. Total elapsed times for 1–64 reactive and deliberative agents in SIM_AGENT and HLA_AGENT (single federate).

environment on a single cluster node using SIM_AGENT and a single HLA_AGENT federate.[10] This gives an indication of the overhead inherent in the HLA_AGENT library itself independent of any communication overhead entailed by distribution. As can be seen, the curves for SIM_AGENT and HLA_AGENT are quite similar, with the HLA overhead diminishing with increasing CPU load. For example, with 64 reactive agents, the HLA introduces a significant overhead: with SIM_AGENT, the average elapsed time is 14.5 seconds compared to 22.4 seconds with HLA_AGENT, giving a total overhead for the HLA of approximately 54.5%. For agents that intrinsically require more CPU, the overhead is proportionately smaller. With 64 deliberative agents, the average elapsed time is 52.08 seconds with SIM_AGENT and 54.45 seconds with HLA_AGENT, giving a total overhead for the HLA of just 4.6%. This is more or less what we would expect: with a single federate, the HLA adds a more or less fixed overhead to specific operations (simulation startup, the creation and deletion of objects, updating attributes, and synchronization). As the total CPU required by the user simulation increases, the additional time required for these operations with HLA becomes less significant.

We also investigated the effect of distributing the Tileworld agents across varying numbers of federates. Figure 4 shows a breakdown of the total elapsed time for an agent federate when distributing 64 reactive and deliberative agents over 1, 2, 4, 8, and 16 nodes of the cluster.[11] In each case, the environment was simulated by a single environment federate running on its own cluster node. As expected, the elapsed time drops with increasing distribution, and with four

[10]We report elapsed times as these account for network latencies, avoid ambiguities in allocating time to multiple threads, and give a true indication of the speedup that can be obtained in practice. Our test case is CPU bound; on an otherwise unloaded cluster node, the CPU and elapsed times are very similar.

[11]Unfortunately, it was not possible to obtain exclusive access to all the nodes in the cluster for our experiments.

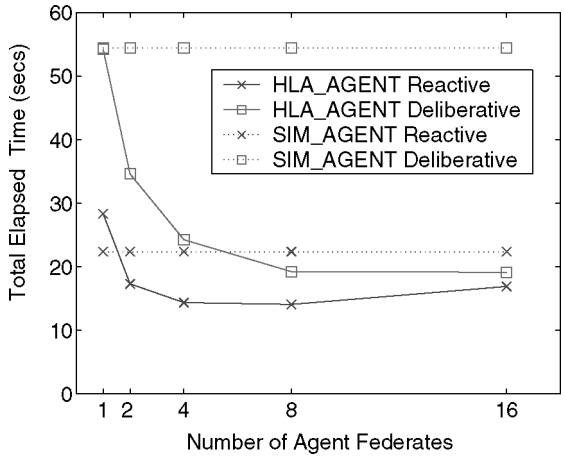

Fig. 4. Total elapsed time for an Agent Federate (64 Reactive and Deliberative Agents distributed over 1–16 nodes).

nodes the elapsed time is comparable to the nondistributed case for the reactive agents. (To aid comparison, the total elapsed times for the nondistributed case: 64 reactive and deliberative agents simulated on a single cluster node using SIM_AGENT, are also shown.) For the more computation-bound deliberative agents a greater speedup is achieved, and with four nodes the elapsed time is approximately half that of the nondistributed case. However in both the reactive and deliberative cases, as the number of nodes (and hence the communication overhead) increases, the gain from each additional node declines.

The total elapsed times for each simulation cycle for the distributed HLA_AGENT experiments can be further broken down into the time for the simulation phase (running the user simulation plus object registration and deletion, attribute ownership transfer requests, and queueing attribute updates for propagation at the end of the user simulation cycle) and the RTI phase (flushing queued attribute updates to the RTI, applying incoming attribute updates to the slots of local objects and proxies, processing object discoveries and deletions, and synchronizing with other federates). Figure 5 shows a breakdown of the total elapsed time for both the simulation and RTI phases of HLA_AGENT for an agent federate for each distribution of 64 reactive agents over nodes of the cluster. As expected, the simulation time per federate initially decreases when using more nodes before levelling off, when the overhead inherent in the HLA_AGENT library starts to dominate the time spent in the user simulation. However, with more than two agent federates, the time spent in the RTI phase increases as the communication between federates (and hence callbacks from the RTI) increases. In the case of reactive agents, with more than eight agent federates the increased RTI overhead offsets the relatively modest gains that can be made by distributing such lightweight agents. With deliberative agents, which intrinsically require greater (though still fairly modest) CPU, we continue to see small improvements in overall elapsed time up to 16 agent federates.

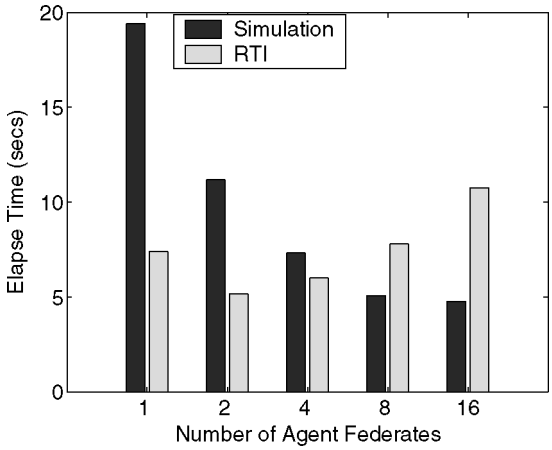

Fig. 5. Simulation and RTI phase times for an Agent Federate (64 Reactive agents distributed over 1–16 Nodes).

To test this hypothesis, we modified the configuration of the agent federates so that they did not subscribe to the *position* attribute of the Agent class. This reduces the total number of agent position updates propagated by the RTI, from linear in the number of federates, to constant. (With a relatively small number of tiles to push and low environment dynamism, agent position updates constitute the bulk of the RTI traffic. For example, with a single agent federate we would expect to see 128 Tileworld agent position attribute updates per cycle, since Tileworld agents update their `x` and `y` positions every cycle. With 16 agent federates, the environment federate still receives 128 attribute updates per cycle, but in addition each agent federate receives 120 Tileworld agent position updates from the 60 Tileworld agents on the other agent federates. As a result, the number of callbacks processed by the RTI in each cycle grows from 128 with 1 agent federate to 2048 with 16 agent federates.) As expected, this has no effect with a single agent federate, since in this case the agent positions are only sent to the environment federate. However it makes a noticeable difference with 16 agent federates, reducing the elapsed time for 64 reactive agents from 16.92 to 13.82 seconds, (a reduction of 22.4 %) and for 64 deliberative agents from 19.14 to 16.13 seconds (a reduction of 18.7%).

In practice the speedup achievable by limiting update propagation is less dramatic. Some agent position updates need to be propagated to agents simulated by other federates, the positions of those agents which are within sensor range of a given agent, and HLA's static DDM scheme is incapable of limiting the propagation of updates to only those that are actually required by the other agents. We would therefore expect to see a reduction in elapsed time somewhere between the no subscription case and the broadcast case. However such gains can still be worthwhile, and in future work we plan to exploit HLA's DDM services in HLA_AGENT.

In addition, without load balancing, the speedup that can be obtained is limited by the elapsed time for the slowest federate. Figure 6 shows the total

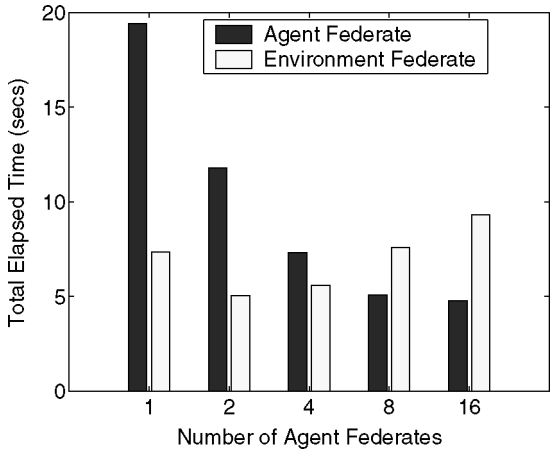

Fig. 6. Total elapsed simulation phase time for Agent and Environment Federates (64 Reactive Agents Distributed Over 1–16 Nodes).

cycle elapsed times for the simulation phase of HLA_AGENT for the environment federate and an agent federate for each distribution of 64 reactive agents over nodes of the cluster. As can be seen, with more than four agent federates, the simulation phase time for the environment federate is greater than that for any single agent federate. Prior to this point, the environment federate spends part of each cycle waiting for the agent federate(s) to complete their simulation phase, and after this point agent federates spend part of each cycle waiting for the environment federate. With eight agent federates, the elapsed time of the environment federate forms a lower bound on the elapsed time for the federation as a whole, and further speedup can only be obtained by distributing the environment across multiple federates. Without the communication overhead of distribution, we would therefore expect the total elapsed time to reach a minimum between four and eight agent federates and thereafter remain constant.

Our experiments show that significant speedup can be achieved by distributing agent federates across multiple cluster nodes, even in the case of lightweight agents with modest computational requirements. As expected, since lightweight agents are communication bound, the broadcast communication overhead starts to offset the reduction in simulation elapsed time, ultimately limiting the speedup that can be achieved. We would expect to see more favorable results with heavyweight agents, which are intrinsically computation bound.

7. RELATED WORK

In this section, we briefly outline some related work in distributed agent simulation and compare it to HLA_AGENT.

DGensim [Anderson 2000] is a distributed simulator for multiagent systems. Simulations are distributed over n node processors, $n - 1$ of which execute an *agent manager* and one or more agents, while the remaining node executes an *environment manager*. Agents send timestamped actions asynchronously to the

environment manager. The environment manager is a time-driven simulation that processes each action when the environment's simulation time reaches the timestamp associated with the action. Several strategies for handling agent actions that arrive after the environment simulation has advanced to after the timestamp of the action are discussed in Anderson [2000], but none is enforced by DGensim. Agents are environment, are written in Lisp, and the agents can have any architecture. The central aim of DGensim was improved simulation fidelity, with increased speed of simulation execution viewed as a useful side-effect. Interoperability with other simulators is not an aim.

[Schattenberg and Uhrmacher 2001; Uhrmacher 2001] have developed a Java-Based Agent Modelling Environment for Simulation (JAMES). In JAMES, agents are modelled as situated automata, and the agent program is compiled into transition rules. JAMES extends the DEVS formalism [Zeigler et al. 2000] and utilizes a hybrid *discrete event simulation* approach rather than the time-stepped simulation used by HLA_AGENT. Agent components with difficult to predict latencies (e.g., deliberation) are not modelled but called directly from the simulation, with the actual computation time being used to determine the time of the next event. The main objective of the JAMES work is to facilitate small- and large-scale testing of multiagent systems, and the system has been used to implement a version of Tileworld for testing simple planning agents. As with DGensim, interoperability with other types of simulation was not a design goal.

MACE3J [Gasser and Kakugawa 2002] is a Java-based simulation, integration and development testbed, which aims to fulfil the need for tools to support distributed collaborative scientific research in large-scale, large-grain multi-agent systems. MACE3J places minimal constraints on the architecture of the agents. Agents are viewed in terms of the services they utilize. As with HLA_AGENT, agents are distributed via proxies. This allows the interaction of simulated and 'real' components and the flexible transfer of functionality from simulated to real components. MACE3J has been used in a number of experimental applications, including a workflow system with 5000 agents, and the distribution of a legacy agent application written in Java.

SPADES [Riley and Riley 2003; Riley 2003] is a conservative parallel discrete simulator for multiagent systems. As with DGensim, simulations are distributed over $n - 1$ agent nodes, all of which communicate with a centralized simulation engine that executes the world model. SPADES provides no direct support for the distribution of the world model (which must be written in C++), whereas HLA_AGENT supports the decomposition of the environment into multiple federates. Agents can have any architecture (so long as the sense, think, and act phases are discrete) and be implemented in any language. Like JAMES, SPADES uses a *software-in-the-loop* methodology to track the computation time of the agents. Agent actions do not need to be synchronized, and sensing is by broadcast communication (every update to the world model is sent to every agent). In contrast, in HLA_AGENT, agent actions are synchronous, but updates are distributed to federates only if they subscribe to the attribute being updated, and can be further restricted by using the DDM services of the RTI (though DDM is not used in the current implementation of HLA_AGENT). Like

HLA_AGENT, SPADES supports the interoperation of agents written by different groups. However HLA-compliance was not a goal and the interface is at the level of event messages rather than objects and attributes of the FOM.

Direct comparisons of the performance of HLA_AGENT with DGensim and JAMES are difficult, due to a lack of published data and/or differences between the systems. For MACE3J, Gasser and Kakugawa [2002] cite almost linear speedup for agents, which are "relatively independent", that exchange few messages. The closest comparison is probably with SPADES, and the results reported in Riley and Riley [2003] (2–26 agents distributed over 1–13 machines) show a similar pattern to those in Section 6. With fast agents, similar to our reactive agents, only modest speedups are achieved beyond 5 nodes. With slow agents (similar to our deliberative agents) the speed up tails off around 9 nodes.

8. SUMMARY

In this article, we have presented HLA_AGENT, a tool for the distributed simulation of multiagent systems, which integrates the SIM_AGENT toolkit and the HLA. We showed how HLA_AGENT can be used to distribute an existing SIM_AGENT simulation with different agents being simulated by different federates, and briefly outlined the changes necessary to the SIM_AGENT toolkit to allow integration with the HLA. No additional management federates are required and existing SIM_AGENT simulations run unmodified. Simulations developed using HLA_AGENT are by definition capable of interoperating with other HLA-compliant simulators and the objects and agents in the simulation can be flexibly distributed across multiple federates so as to make best use of available computing resources.

We are currently investigating the performance of HLA_AGENT in a number of SIM_AGENT applications. While further work will enable us to analyze the RTI overhead and characterize the performance of HLA_AGENT with different kinds of agents and environments, the results of experiments in which we used HLA_AGENT to distribute an existing SIM_AGENT simulation across the nodes of a Linux cluster show we can get speedup even with agents that have minimal computational requirements. However, with lightweight agents, the broadcast communication overhead with large numbers of federates rapidly starts to offset the reduction in simulation elapsed time, limiting the speedup that can be achieved. With agents that intrinsically require more CPU, the situation is more promising, and for such computationally intensive simulations, we believe the HLA represents a viable approach from the point of view of both interoperability and distribution.

The efficient propagation of updates to the shared environment is a key problem for simulations of multiagent systems [Lees et al. 2004]. Our approach currently makes no use of the data distribution management services provided by the RTI, and we plan to extend our current implementation of HLA_AGENT to utilize DDM. We are also investigating more radical approaches to the problem of scalability, which attempt to avoid the bottleneck introduced by a centralized communication component (in our case the RTI). The bottleneck results from the need to ensure time-synchronization and mutual exclusion when the

agents interact with a shared environment, and is a phenomenon encountered to some degree by any centrally managed distributed simulation. We are therefore investigating the application of interest management techniques to fully distributed MAS simulators without a centralized component, where both the location of data and its propagation is adaptively managed to optimize network performance. This work is the subject of the PDES-MAS project [Logan and Theodoropoulos 2001].

Another problem is that the speedup depends on the initial allocation of agents to federates. If this results in unbalanced loads, the slowest federate will constrain the overall rate of federation execution. It should be relatively straightforward to implement a simple form of code migration to support coarse grain load balancing by swapping the execution of a locally simulated object with its proxy on another, less heavily loaded, federate. We also plan to investigate the performance implications of distributing the simulation across multiple (geographically dispersed) clusters.[12] Together, these extensions will form the first step towards a GRID-enabled HLA_AGENT.

Another area for future work is the *interoperation* of simulations developed using HLA_AGENT with other simulators. This would allow the investigation of different agent architectures and environments using different simulators in a straightforward way. In a related project [Minson and Theodoropoulos 2004], we are developing an HLA-compliant version of the RePast agent simulator [Collier 2004], which will form part of a combined HLA_AGENT/RePast federation.

ACKNOWLEDGMENTS

We would like to thank Matt Ismail of the Centre for Scientific Computing at the University of Warwick for access to their Task Farm cluster.

REFERENCES

ANDERSON, J. 2000. A generic distributed simulation system for intelligent agent design and evaluation. In *Proceedings of the 10th Conference on AI, Simulation and Planning, AIS-2000*, H. S. Sarjoughian, F. E. Cellier, M. M. Marefat, and J. W. Rozenblit, Eds. Society for Computer Simulation International, 36–44.

ATKIN, S. M., WESTBROOK, D. L., COHEN, P. R., AND JORSTAD., G. D. 1998. AFS and HAC: Domain general agent simulation and control. In *Software Tools for Developing Agents: Papers from the 1998 Workshop*, J. Baxter and B. Logan, Eds. AAAI Press, 89–96. Tech. Rep. WS–98–10.

BAXTER, J. AND HEPPLEWHITE, R. T. 1999. Agents in tank battle simulations. *Commun. ACM 42*, 3, 74–75.

BRADSHAW, J., Ed. 1997. *Software Agents*. AAAI Press, Menlo Park, CA.

COLLIER, N. 2004. RePast: An extensible framework for agent simulation. http://repast.sourceforge.net.

DMSO 1998. High level architecture interface specification, version 1.3.

DURFEE, E. H. AND MONTGOMERY, T. A. 1989. MICE: A flexible testbed for intelligent coordination experiements. In *Proceedings of the 9th Distributed Artificial Intelligence Workshop*. 25–40.

EPHRATI, E., POLLACK, M., AND UR, S. 1995. Deriving multi-agent coordination through filtering strategies. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, C. Mellish, Ed. Morgan Kaufmann, San Francisco, 679–685.

FERBER, J. 1999. *Multi-Agent Systems*. Addison Wesley Longman.

[12]See www.cs.bham.ac.uk/research/projects/dsgrid.

FIELDING, D., FRASER, M., LOGAN, B., AND BENFORD, S. 2004. Extending game participation with embodied reporting agents. In *Proceedings of the ACM SIGCHI International Conference on Advances in Computer Entertainment Technology (ACE 2004)*. ACM Press, New York, NY, USA, 100–108.

FIELDING, D., LOGAN, B., AND BENFORD, S. 2006. Balancing the needs of players and spectators in agent-based commentary systems. In *Proceedings of the 5th International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2006)*, P. Stone and G. Weiss, Eds. IEEE Press, Hakodate, Japan, 996–998.

GASSER, L. AND KAKUGAWA, K. 2002. MACE3J: Fast flexible distributed simulation of large, large-grain multi-agent systems. In *Proceedings of the 1st International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2002)*, C. Castelfranchi and W. L. Johnson, Eds. ACM Press, Bologna, 745–752.

IEEE 2000. IEEE Standard for modeling and simulation (M&S) High Level Architecture (HLA)—Framework and rules. IEEE. (IEEE Standard No.: 1516-2000).

JENNINGS, N. R. AND WOOLDRIDGE, M. 1998. Applications of intelligent agents. In *Agent Technology: Foundations, Applications, Markets*, N. R. Jennings and M. Wooldridge, Eds. Springer-Verlag, 3–28.

KUHL, F., WEATHERLY, R., AND DAHMANN, J. 1999. *Creating Computer Simulation Systems: An Introduction to the High Level Architecture*. Prentice Hall.

LEES, M., LOGAN, B., MINSON, R., OGUARA, T., AND THEODOROPOULOS, G. 2004. Modelling environments for distributed simulation. In *Environments for Multi-Agent Systems: Proceedings of the the 1st International Workshop (E4MAS'04)*, D. Weyns, H. V. D. Parunak, and F. Michel, Eds. Number 3374 in LNAI. Springer, 150–167.

LOGAN, B. AND ALECHINA, N. 1998. A* with bounded costs. In *Proceedings of the 15th National Conference on Artificial Intelligence, AAAI-98*. AAAI, AAAI Press/MIT Press, Menlo Park CA & Cambridge MA, 444–449.

LOGAN, B., FRASER, M., FIELDING, D., BENFORD, S., GREENHALGH, C., AND HERRERO, P. 2002. Keeping in touch: Agents reporting from collaborative virtual environments. In *Artificial Intelligence and Interactive Entertainment: Papers from the 2002 AAAI Symposium*, K. Forbus and M. S. El-Nasr, Eds. AAAI Press, Menlo Park, CA, 62–68. Tech. Rep. SS–02–01.

LOGAN, B. AND THEODOROPOULOS, G. 2001. The distributed simulation of multi-agent systems. *Proceedings of the IEEE 89*, 2 (Feb.), 174–186.

LUCK, M., MCBURNEY, P., SHEHORY, O., AND WILLMOTT, S., EDS. 2005. *Agent Technology Roadmap: A Roadmap for Agent Based Computing*. AgentLink III.

MINSON, R. AND THEODOROPOULOS, G. 2004. Distributing RePast agent-based simulations with HLA. In *Proceedings of the 2004 European Simulation Interoperability Workshop*. Simulation Interoperability Standards Organisation and Society for Computer Simulation International, Edinburgh. Paper No. 04E-SIW-046.

MORSE, K. L. AND ZYDA, M. 2000. On line multicast grouping for dynamic data distribution management. In *Proceedings of the 2000 Fall Simulation Interoperability Workshop*. Simulation Interoperability Standards Organisation and Society for Computer Simulation International. Paper No. 00F-SIW-052.

POLLACK, M. E., JOSLIN, D., NUNES, A., UR, S., AND EPHRATI, E. 1994. Experimental investigation of an agent commitment strategy. Tech. Rep. TR 94–31, University of Pittsburgh, Pittsburgh, PA 15260.

POLLACK, M. E. AND RINGUETTE, M. 1990. Introducing the Tileworld: Experimentally evaluating agent architectures. In *Proceedings of the 9th National Conference on Artificial Intelligence*. 183–189.

RILEY, P. 2003. MPADES: Middleware for parallel agent discrete event simulation. In *RoboCup-2002: The Fifth RoboCup Competitions and Conferences*, G. A. Kaminka, P. U. Lima, and R. Rojas, Eds. Number 2752 in Lecture Notes in Artificial Intelligence. Springer Verlag, Berlin, 162–178.

RILEY, P. AND RILEY, G. 2003. SPADES—a distributed agent simulation environment with software-in-the-loop execution. In *Proceedings of the Winter Simulation Conference*, S. Chick, P. J. Sánchez, D. Ferrin, and D. J. Morrice, Eds.

SCHATTENBERG, B. AND UHRMACHER, A. M. 2001. Planning agents in JAMES. *Proceedings of the IEEE 89*, 2 (Feb.), 158–173.

SCHEUTZ, M. AND LOGAN, B. 2001. Affective vs. deliberative agent control. In *Proceedings of the AISB'01 Symposium on Emotion, Cognition and Affective Computing*. AISB, The Society for the Study of Artificial Intelligence and the Simulation of Behaviour, 1–10.

SLOMAN, A. AND LOGAN, B. 1999. Building cognitively rich agents using the SIM_AGENT toolkit. *Comm. ACM 42*, 3 (Mar.), 71–77.

SLOMAN, A. AND POLI, R. 1996. SIM_AGENT: A toolkit for exploring agent designs. In *Intelligent Agents II: Agent Theories Architectures and Languages (ATAL-95)*, M. Wooldridge, J. Mueller, and M. Tambe, Eds. Springer–Verlag, 392–407.

UHRMACHER, A. M. 2001. Dynamic structures in modeling and simulation: A reflective approach. *ACM Trans. Model. Comput. Simul. 11*, 2, 206–232.

WOOLDRIDGE, M. AND JENNINGS, N. R. 1995. Intelligent agents: Theory and practice. *Knowledge Engineering Review 10*, 2.

ZEIGLER, B. P., PRAEHOFER, H., AND KIM, T. G. 2000. *Theory of Modeling and Simulation*, 2nd ed. Academic Press.

Received October 2004; revised May 2006, July 2006; accepted July 2006