Farm: A Scalable Environment for Multi-Agent Development and Evaluation *

Bryan Horling, Roger Mailler, Victor Lesser
University of Massachusetts
Amherst, MA
{bhorling,mailler,lesser}@cs.umass.edu

Abstract

In this paper we introduce Farm, a distributed simulation environment for simulating large-scale multi-agent systems. Farm uses a component-based architecture, allowing the researcher to easily modify and augment the simulation, as well as distribute the various pieces to spread the computational load and improve running time. Technical details of Farm's architecture are described, along with discussion of the rational behind this design. Performance graphs are provided, along with a brief discussion of the environments currently being modeled with Farm.

1 Introduction

A tension exists in simulation frameworks which trades off the inherent richness of the provided environment, and the flexibility and ease with which that same environment can be used to analyze a particular aspect of a larger solution. On one hand, robust simulation environments can offer many enabling technologies that both increase the fidelity of the simulation, and provide a range of services that the participants may take advantage of. These same features, however, can be an obstacle if the goal is to evaluate a particular technology in the absence of complicating factors.

Our prior work in the area of multi-agent simulation environments [8] resides in the former category; it provides a wide range of services in an attempt to create a realistic environment in which agents can perform and be evaluated. While using this approach is an important step in agent development, our experience has shown that it can also be helpful to extract key technologies from such an environment, and rigorously test them under conditions which have fewer distractions. Performing these more abstract tests has the dual advantages of reducing possible artifacts from unrelated events, and improving the time needed for analysis by reducing the simulation overhead.

Our recent work addressing negotiation-based resource allocation [5] is a good example of this tension. The full-scale solution was developed by implementing fine-grained, sophisticated agents in JAF [8] using a detailed domain-specific simulation tool called Radsim [4]. Test scenarios were quite realistic, where agents were required to manage all aspects of a tracking multiple targets using a distributed network of sensors. This necessitated solutions for a range of interesting issues, such as organizational design, dealing with noisy or uncertain data, managing agent loads, handling unreliable communication, disambiguating targets, etc [4]. While each of these are important in their own right, and some have important effects on negotiation, many are orthogonal to the original resource allocation problem. We found that operating under such conditions not only distracted from this original goal, but also failed to illuminate potential flaws in the negotiation scheme. Negotiation errors were sometimes mis-attributed to related subsystems and we were unable to scale the collection of fine-grained agents using a reasonable number of processors.

In this paper we will present Farm, a distributed simulation environment designed to address the need for a more focused testbed. The core of Farm provides essential functionality needed to drive a multi-agent system, in such a way that elements such as the scalability, real-time convergence rate and dynamics of a particular system can readily be evaluated and compared. Farm has, in some sense, taken a step back by moving to a lighter weight implementation, in order to provide an environment where multi-agent subsystems may be quickly developed and evaluated.

Farm is a component-based, distributed simulation environment written in Java. Individual components have re-

*Effort sponsored in part by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory Air Force Materiel Command, USAF, under agreements number F30602-99-2-0525 and DOD DABT63-99-1-0004. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. This material is also based upon work supported by the National Science Foundation under Grant No. IIS-9812755. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), Air Force Research Laboratory or the U.S. Government.

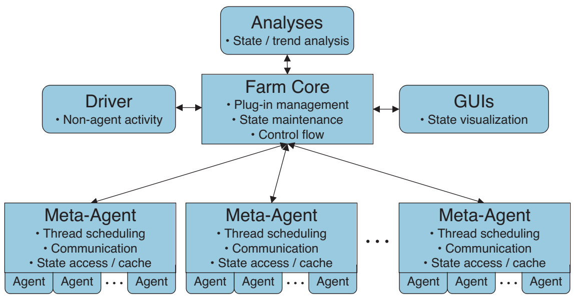

Figure 1. Farm's component architecture.

sponsibility for particular encapsulated aspects of the simulation. For example, they may consist of agent clusters, visualization or analysis tools, environmental or scenario drivers, or provide some other utility or autonomous functionality. These components or agent clusters may be distributed across multiple servers to exploit parallelism, avoid memory bottlenecks, or utilize local resources. In addition, the set of components used in a particular scenario is not fixed - a limited set might be instantiated initially to reduce the simulation overhead, and components may also be dynamically added or removed at runtime as needed.

Agents are grouped in clusters of one or more entities, and each cluster exists under the control of a meta-agent component which provides access to the rest of the simulation environment. The agents themselves run in pseudo real-time, where individual agents are each allocated a specific amount of real CPU time in which to run. This aspect allows the systems to exhibit a fair amount of temporal realism, where the efficiency of an agent's activities can have quantifiable effects on its performance. Communication actions are similarly modeled and monitored.

Farm has been used to model three different domains, including a variety of agents implementing different types of solutions for these domains. Scenarios consisting of 5000 autonomous agents have been run using 10 desktop-class Linux boxes. These environments will be discussed in more detail later in this paper.

In the following section, we will provide a brief overview of the Farm simulator, followed by a discussion of how Farm relates to other MAS simulation environments. An more detailed look at Farm's architecture is provided in section 4, along with a more in-depth examination of some of those features. We will conclude with examples of the environments that Farm has been used to create, and how they are being used to drive other areas of research.

2 Overview

As mentioned earlier, Farm is a distributed, component-based simulation environment. By distributed, we mean that discrete parts of the environment may reside on physically separate computing systems. In general, no assumptions are made about the type of system a part is run on, with respect to its operating system, memory or disk architecture. In particular, all that is required is a Java interpreter, and a means for that part to communicate with other parts of the environment (e.g. some sort of network connection).

Each part, or component, in this simulation environment is responsible for some aspect of the simulation. Figure 1 shows how a typical set of components in the simulation are related. The hub of activity is the Farm core, which serves as a connection point for components, and also drives the control flow in the system as a whole. The connected components then fall into two categories: those which directly manage the agents running in the system, and those which exist to support of those agents or the scenario as a whole. We refer to these agent managers as meta-agents, as each acts as an interface to and for a cluster of one or more agents. The agents themselves are threads, although this is for performance purposes only - from an agent's perspective they are completely segregated, and are not aware of or share memory with other agents which happen to also be resident at a the same meta-agent.

At runtime, agents are provided time in which to run, and other components are given the opportunity to perform, analyze, or modify the simulation at well-determined times. The run cycle is partitioned such that tasks such as state maintenance or analysis may be performed without adversely affecting the simulation results, even if they require indeterminate time. Such tasks are enabled by the storage of pertinent state information in the Farm core, which then serves as a central location where any given component may interact with a snapshot of the system's current state.

3 Related Work

Attempts have been made [6, 3] to define the set of features and characteristics that a multi-agent simulator should posses. The topics described in these efforts are important, in that they can help guide designers towards a comprehensive, robust solution. Farm, however, is not intended to provide a complete simulation solution - instead it tries to provide a relatively simple environment where agents possess only the most germane functionality. Much of the complexity of a real environment can be either abstracted away, or approximated through the use of black-box style components which provide agents with necessary information, when the actual process of obtaining that information is unimportant. Thus, much of the underlying modeling structures which make other simulators unique is absent in Farm.

3.1 MASS

Our earlier simulation environment, the Multi-Agent System Simulator [8], is quite different than Farm. It provides a quantitative view of the world, where agent activities and their interactions are modeled using TÆMS consumable and non-consumable resources have constraints which can affect behavior, and an agent's beliefs may differ from objective fact. As with the earlier example, agents are built using JAF, which itself has a fair amount of complexity. All of these features are desirable for evaluating sophisticated agents in context, but at the same time they can be distracting when only a subset of behaviors need analysis. In addition, the environmental models and communication mechanisms are centralized, and the agents, while distributed, run as separate processes, so the environment as a whole does not scale well past 40 agents or so. The DECAF [2] agent framework also has a similar character and purpose to JAF/MASS, although it does not have a centralized simulation environment, and it offers built-in brokering and name services which JAF lacks. In most other respects, DECAF compares to Farm in much the same way as JAF.

3.2 MACE3J

MACE3J [1], like Farm, is primarily intended to simulate large numbers of large-grained agents. It includes mechanisms for a different styles of event and messaging control, data collection, and a lightweight agent definition model. It is also scalable, but does so under a multiprocessor-based scheme, taking advantage of the capabilities inherent in lower level system software to manage many of the inter-processes and inter-thread issues which arise in simulation. While this method is undoubtedly more efficient than the distributed approach we have selected, it also requires additional overhead in the form of an actual multi-processor machine, or a cluster of machines tied together with appropriate software. In this sense, Farm is more closely related to the original MACE, which also employed a distributed architecture.

Farm places more emphasis on the real-time aspects of agent behavior, as progress in a scenario is driven by the passage of time, not events or messages, and the effectiveness of an agent is affected by the duration of its computations. In other ways the two environments are similar: Farm also supports repeatability, varied communication models, transitionable agent models and data collection and display tools. MACE3J's support for randomized event sequences is a feature we intend to add to Farm in the future.

3.3 Swarm

Like Farm, the Swarm [7] simulation environment is a modular domain-independent framework. It offers the ability to probe agents' state and beliefs, and graphically display them, similar to the logging and graphing tools provided with Farm. Fundamentally, the two differ in their representation of time passing. Swarm uses a discrete event system, where a point in time in not reached or seen until some event has been scheduled to take place then. Farm uses a real time approach where time passes regardless of what events are taking place. Both techniques are valid, but serve different purposes. In addition, Swarm agents have a different character to them, as they are generally modeled as a set of rules, or responses to stimuli. Conversely, Farm agents are built more like a conventional program, where the designer develops classes and routines to exhibit agent behavior. Again, both approaches have their merits depending on the problem to be addressed.

3.4 Radsim

Unlike the other environments mentioned here, Radsim is not a general simulation framework. Instead it is a real-time, domain-specific simulator designed to accurately model a collection of radar platforms as they attempt to track targets moving through space[4]. We mention it here because it represents an extreme case in the spectrum of simulators, where the actors in the system must recognize and handle virtually every aspect of a problem with a high level of realism. This complexity is critical for testing comprehensive solutions, particularly if the actual environment the solution will run in is otherwise physically or practically unavailable. However, this same complexity can prevent the environment from scaling, and hinder development of critical subsystems. We feel this argues for a heterogeneous approach, where targeted, scalable simulation results, such as Farm can provide, coupled with the type of detailed experiments that can be done in a complex environment such as Radsim, will in the end produce a higher quality solution.

4 Architecture

As mentioned earlier, a simulation environment built using Farm is comprised of a number of components. Central to this arrangement is the Farm *core*, which handles component, control and state management. The *meta-agents*, specialized components which manage clusters of actual agents, implement much of the architectural-level functionality supporting those agents. More generally, component *plug-ins* provide the remainder of the system's non-agent capabilities, typically including both domain-independent and domain-specific elements which create, manage and analyze the environmental state.

An arbitrary set of meta-agents and plug-ins may be used for a given simulation scenario. Additional meta-agents will better distribute the load incurred by the agent population. Intuitively, in an environment with 100 agents, 5 meta-agents managing 20 agents each will run faster than a single meta-agent with all 100 agents. Experimental results looking at the effects of increasing the number of meta-agents can be seen in Figure 2, which shows its effect on total simulation time. The experimental setup consisted of

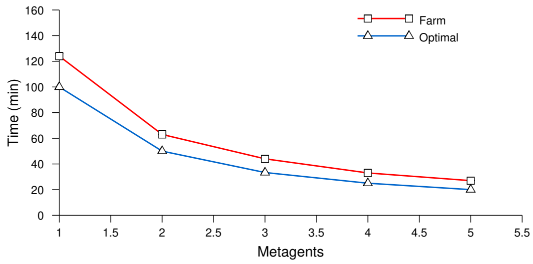

Figure 2. The effect of increasing the number of meta-agents on simulation duration.

a scenario containing 100 agents over a period of 60 seconds, using between one and five meta-agents. An optimal system's duration would be:

$$num\_agents \times scenario\_time / num\_meta\_agents$$

...or 100 minutes with a single meta-agent. This duration would decrease at a rate inversely proportional to the number of additional metagents. Farm closely models the behavior of an optimal system, with differences between the two largely attributable to Farm's environmental modeling and component communication. The number of meta-agents can be increased arbitrarily, although it generally does not make sense to allocate more than one per processor. The set of plug-ins that are used is also quite flexible. For example, one might choose to reduce simulation overhead by running without visualization, or with it to get a clearer picture of the system's state.

Closely related to the total number of meta-agents is the load placed on each of them, and it generally makes sense to distribute the agent population equally across meta-agents. If there are different classes of agents, which may have different runtime characteristics, then this distribution should also be reflected in the allocation. For example, if we have 5 meta-agents, 80 agents of type x and 20 of type y, the best allocation would place $16x$ and $4y$ on each meta-agent. The environmental driver is generally responsible for this allocation, although it could also be done in a start-up script or by the meta-agents themselves.

4.1 Control Flow

As the system starts up, the core acts as a registry for components, which contact the core as they are invoked. Components start by performing an initialization sequence, after which they wait for direction from the core.

Control in the simulation is concerned with the passage of time. Our ultimate goal in this is to ensure that each agent in the system is provided the same amount of physical CPU time, to evaluate how those agents would perform in a continuous-time environment. In a perfect simulation, all agents would be able to operate asynchronously in parallel. However, competition for the local processor by agents resident at the same meta-agent precludes this option if we wish to ensure fairness among them, and having one processor per agent is clearly infeasible for large numbers of agents. Thus, we approximate this behavior by sequentially assigning individual agents a slice of time in which to run. In between such opportunities an agent thread is paused, so the currently running agent has exclusive access to the CPU, as much as this is possible in a multitasking system. The simulation thus approximates how the agent population as a whole would act under real-time conditions, by breaking the timeline into a series of slices and providing each agent the appropriate amount of time to run for each slice.

This process is separated into two different components of the simulator: the core and meta-agents. The core starts a pulse by notifying all meta-agents in parallel that their agents may run for some duration of time. Each meta-agent then sequentially wakes their local agents for the specified amount of time, after which the core is notified of completion. After all meta-agents have completed the process, the core resumes execution. We refer to this process as the agents receiving a *pulse* of time in which to run. Just before and after this pulse, the components also allowed an indeterminate amount of time in which to run. Thus, before the agents are pulsed, a driver might update environmental data (e.g. a moving target's position). After a pulse, analysis tools might take time to update statistics or visualizations.

Because of this style control flow, interactions do not take place between agents within a single time pulse - the effects of one agent will not be observable by another until that pulse has ended. This can lead to a certain amount of data, communication or behavioral incoherence in the system, the duration of which is bounded by the length of the time slice. Mechanisms for addressing this issue are covered in more detail in section 4.5.

The pseudo real-time nature of this control implies a certain amount of non-determinism to otherwise identical scenarios, as external events may effect the actual amount of processing time an agent receives during a window of real time. If determinism is required, Farm also supports a fixed notion of pulse time, which tracks the agent's execution progress, rather than just elapsed time. For example, instead of allocating 100ms per pulse, agents could be allocated a single pass through their main event loop. Thus, for each pulse, each agent's `pulse` method would be called once, and (assuming the agents themselves are deterministic) the scenario as a whole will be deterministic because the same sequence of activities will be performed for each run. This allows repeatability, but prevents one from drawing strong conclusions about how the agents behave in real time.

4.2 Data Flow

The Farm core also serves as a repository of global data. This is not inter-agent shared memory, instead it provides an

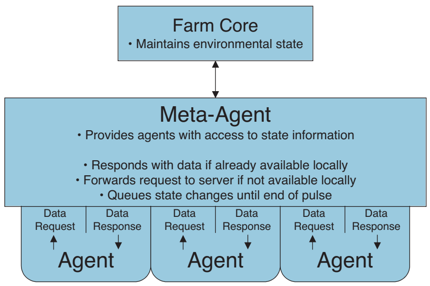

Figure 3. Meta-agent handling of data flow.

indirect means of interaction between components. For example, an environmental driver might be responsible for updating the `Target1:Location` property. Agents needing to know that target's location can then simply access this property. Similarly, an agent could store some notion of it's current state in `Agent1:State`. An analysis component could then find all properties `*:State` to capture a snapshot of all the agents in the system.

This type of mass data storage presents particular problems to our environment. First, it is clearly a bottleneck, as all agents might be reading or writing to this space during a given time period. Second, it presents the potential for inconsistency, as other entities might attempt to access a data element while another is writing to it.

The bottleneck problem cannot be wholly resolved, as there are frequently instances where one component requires access to data who's size grows with the number of agents, however we can try to mitigate the effects. First, meta-agents are used as a intermediaries, which intercept property requests from agents and cache any results, as see in Figure 3. Subsequent attempts to access the same data will hit the cache first, and thus avoid accessing the core itself. Writes back to the simulator are also intercepted, stored locally in the meta-agent, and only written back to the simulator when the pulse has ended. This has the effect of consolidating multiple writes to the same variable. Secondly, the core offers an API where components may exploit Java's remote object passing capabilities to perform functions locally in the core. For example, if the analysis component above simply needs to compute an aggregate statistic, it may instead elect to send the core an object capable of performing the task. Then, instead of transferring a potentially large amount of data from the core to be analyzed, the relatively small object is sent to the core to do the job. The data is processed locally, and only the final statistic is returned to the analysis component, avoiding unnecessary data transfer[1].

Data consistency problems can occur if an agent or component attempts to access the same data while another is

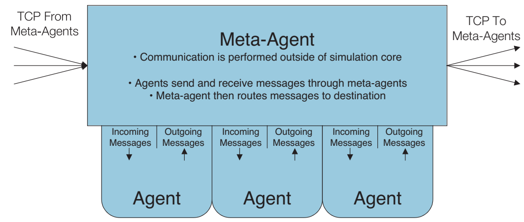

Figure 4. Farm communication flow.

writing to it. These types low level interactions are handled through synchronization, however higher level problems can still arise. For example, if an analysis tool computing an aggregate statistic intermingles its reads with writes from an agent, the statistic will not necessarily be correct. Because agents are not supposed to be directly sharing data they produce (they are assumed to do this via message passing), inter-agent coherency issues are not as significant a problem. We currently resolve these issues by sequentializing the activation of plug-ins. Because no more than one may be active at a time, such race conditions cannot occur. This also has the unfortunate side effect of eliminating the benefits of parallelism among plug-ins, although generally the potential speed improvement is negligible compared to the gains obtained by running the meta-agents in parallel.

Centralization of the data also has the benefit of providing a central location where this data may be easily stored. The core has built-in functionality to continuously save this data over time. Such save files can then be played back again, and even interacted with as though it were live. For example, one might complete and save an example run without any analysis components. Later, this run can be replayed, this time with only analysis components, which will behave exactly as they would have had they been present when it was recorded. Similarly, one might play back that same run with only visualization present to achieve a movie-like effect. This is especially useful for demonstrations, when sufficient computational power for a real simulation may be unavailable. From a testing standpoint, one could also record the bare environment over a period of time without any agents being present. Then that exact scenario could be played back multiple times with agents present, to help debug and evaluate changes made to the agents.

4.3 Communication

To improve scalability, communication takes place entirely outside of the core. Instead, communication occurs between meta-agents, and individual agents send and receive messages via their managing meta-agent, as shown in Figure 4. When a new meta-agent registers with the core, all existing meta-agents are told about the addition, and the new meta-agent is given a list of all other members - thus a fully-connected graph of meta-agents is maintained. When an agent sends a message, it is added to a per-agent outgoing queue. The meta-agent selects ready messages from these

[1]This may beg the question, why distribute such components in the first place, since the real computation is being done locally in the core? Even though the raw statistic is computed at the core, this data will still need to be stored and potentially visualized. Thus resource requirements remain which can be satisfied by distributing the component on a remote host.

queues and checks its address table to determine what meta-agent the recipient belongs to. If the meta-agent is found, the message is delivered. If it is not found, it uses the list of known meta-agents to find the appropriate one, and that mapping is then recorded in the address table. Thus each meta-agent will learn a mapping for only necessary destination agents. As messages are received by a meta-agent, they are added to a per-agent incoming message queue, which is polled by the agent as necessary.

We wish to have a relatively realistic network model, so care is taken when sending messages. A potential race condition also exists for message delivery, as one agent's message may reach another agent before it has technically been sent in the global time line. As messages are added to an agent's outgoing queue, they are marked with a delivery time. The delivery time of a message will be that of the prior message in the queue, plus a bounded random transit duration which can be weighted by the length of the message. A message loss rate probability may also be set. At the end of a pulse, each meta-agent searches the outgoing message queues of its local agents, and sends messages if permitted by the assigned delivery times. These messages are queued for delivery at the destination meta-agent. At the beginning of the next pulse, those received messages are delivered to the appropriate incoming queue for each agent. The agent is then responsible monitoring its queue and handling new messages. We are investigating other potential communication paradigms, such as a defined routing network or a distance-limited broadcast scheme, to provide additional communications scenarios for agents to explore.

While this decentralized communication mechanism scales very well, it prevents other components from directly observing or analyzing message traffic. Gross statistics, such as total incoming and outgoing messages, are currently computed and stored as global properties by individual meta-agents. Other statistics can be computed in a similar manner to compensate for this design decision.

The exact messaging protocol is left intentionally unspecified and abstract. Agents simply send `Message` objects, which can be extended as needed. The destination agent then receives `Message` objects in its incoming queue. Parsing of the object is performed automatically.

4.4 Scalability

Some discussion of the scalability of Farm has been mentioned earlier. The component architecture of Farm, and specifically its ability to segregate the agent population into groups under the control of distributed meta-agents, leads the environment to large scale scenarios. Because the agents effectively run in parallel because of this distribution, the primary constraint is having available computing power to run such simulations in a reasonable amount of time.

Figure 5 shows a sample of Farm's scalability characteristics, from the results of a series of repeated trials with a scenario length of 60 seconds. The number of meta-agents was fixed at five, and the number of agents gradually increased from 100 to 1000, with a 1:4 ratio of targets to sensors. The distributed resource allocation domain from section 5.1 was used, because the movement of targets provides a need for continual re-evaluation and activity. The agents in this domain do actual problem solving, and use communication to negotiate over areas of contention. The initial results seen in this graph are promising.

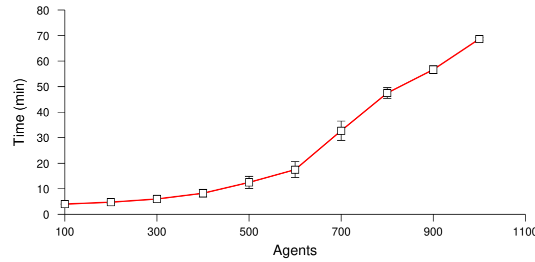

Figure 5. The effect of increasing the number of agents on simulation duration.

In comparing Figure 5 with 2, one might also note a significant difference in simulation duration. For example, the 100 agent case here took only 6 minutes, as compared to 124 minutes previously. Both experiments used the same machines, domain and agent population, but the trials from Figure 5 allowed the agents to signal their meta-agent if they have no additional work to do. This allows the agent's pulse cycle to be ended prematurely, with potentially large savings in actual running time without loss of precision in the simulation results. In our scenario, if there are more agents in an environment of constant size, there is a higher probability that additional computation will be needed to resolve the correspondingly larger number of conflicts. This is seen in non-linearity of the data in Figure 5, where disproportionately more time is used in larger populations. In this way, the system avoids expending effort simulating agents' "idle" time, which gives Farm some of the benefit that a strictly event-based simulation environment would posses.

Perhaps more interesting than "how large can it get?" is the question "what prevents it from getting large?". No design is perfect, and parts of Farm's architecture can inhibit scale in order to permit other features. The most constraining is the centralization of data storage, as outlined in section 4.2. This is necessary to facilitate state analysis, but excessive usage can accumulate a large time penalty. In general, such scenarios will just take longer to process than if the data storage were completely distributed. Data caching and delayed write-backs ensure agents are not unduly penalized for the time required to update data to the core, and we are currently investigating methods to account and compensate for the time required to fetch information.

Another constraint, related to data flow, is environmental maintenance. The task of creating and maintaining the simulation environment (e.g. placing sensors, moving targets, etc.) is typically the responsibility of a single component. Like any other, this component may be distributed for load balancing purposes, but it is still a single process limited to the resources present at its local processor. Like the agents, it also accesses state data, but since it has the responsibility of maintaining the entire state, the potential burden is much more concentrated. Extremely large, complex or dynamic environments might therefore benefit from separating the environmental maintenance into separate components, much as the agents themselves are separated. Thus, one might have a target component, a sensor component, and the like, each with a specific, tractable responsibility.

4.5 Coherency

Whenever an environment is distributed, the problem of coherency arises because entities on one processor may have data inconsistent with that on another. One must try to make sure that interactions between processors are as faithfully represented as those occurring on the same processor.

The data consistency problem in Farm manifests itself in the time between when one agent changes a value to when that change can be observed by another. In between those events, the system has lost some measure of coherence. To avoid potential race conditions, the meta-agents cache value modifications during a pulse, and write those values back when the pulse ends, as discussed in section 4.2. Thus, the end of a pulse acts as a synchronization point, and the duration of each pulse represents the maximum length of the duration of incoherence. This value can be specified to whatever is appropriate (a typical value is 100ms for real time applications). More generally, because agent interactions do not occur within a pulse, the pulse duration is the effective granularity of the simulation. A smaller value will lead to finer granularity and greater coherence, but will increase the overhead associated with running the system because of the additional synchronization points.

Communication coherency is also important. Farm must ensure that a message is delivered when appropriate, and from the recipient's perspective, not before it was actually sent. As outlined in section 4.3, this is accomplished in a manner similar to the data flow. Messages from the agents are queued for delivery during the pulse, and only sent after the pulse has completed. The receiving meta-agent queues incoming messages, which are delivered to their final recipient when the specified delivery time has been reached.

A more insidious form of incoherency occurs when the meta-agents are distributed across a heterogeneous set of machines. Because the agent's computational effort is measured in seconds, one group of agents may effectively be allocated more time simply because the processor they happen to reside on can perform more computations in the same amount of time. A few strategies can be employed to compensate for this problem. One could compute a processor-to-real time ratio for all machines in the pool, and use that to scale the actual time allocated by individual meta-agents. One could also statistically remove the problem through repeated trials where the agent population is shuffled between hosts. A third option (clearly requiring much less effort) is to simply ensure your server pool is homogeneous, or accept the performance differences as a byproduct of working in a realistic environment. For the results presented in this paper, the experiments were performed using a group of similarly configured workstations.

5 Environments

Several computational environments have been implemented using Farm, each taking about two days to implement the environment itself, and the agents taking from a day to a week depending on their complexity and the availability of source code. Each environment generally consists of a *driver*, which instantiates and maintains the environment, and *analysis* components, which generates statistics at runtime, and a set of one or more types of agents. In addition, several generic components have been developed which may be used across all environments. These include a graphing component, property log, and time driver.

5.1 Distributed Resource Allocation

The distributed resource allocation environment is an abstraction of the distributed sensor network problem [4]. A complete solution would reason about a range of issues, from role assignment to low level task scheduling to multi-target data association. The underlying problem, however, is much more straightforward. The environment consists of a number of sensors and mobile targets, and the high level objective is to use the sensors to track them. Each sensor has limitations on its range and usage. This then reduces to a resource allocation problem, where the attention of the sensors must be allocated so that all the targets are tracked.

Our comprehensive solution to this problem is implemented as a homogeneous collection of sophisticated JAF agents, which run in real time in both the Radsim simulator and hardware. In the simpler Farm environment, there are two simpler types of agents: sensor agents, each of which controls a single sensor, and tracking agents, each of which is responsible for tracking a single target. The driver provides the track managers with a list of candidate sensors, i.e. those which are in range of its target, and the track manager must determine which sensors it wants to use. The track managers must then coordinate to resolve conflicts so all targets are tracked. The SPAM negotiation protocol [5] was implemented to solve this problem.

This domain was the incentive behind Farm's creation, and has shown itself to be particularly useful in debugging and evaluating SPAM. Because Farm scales to much greater numbers, and also eliminates most of the complicating, but ultimately tangential factors (relative to resource allocation), development of the protocol was much more tractable. We were also able to directly use almost all the code from the original JAF-based implementation, so improvements made in the Farm environment were easily mapped back the more realistic Radsim and hardware environments.

5.2 Graph Coloring

The well-known graph coloring domain was implemented as a means of both testing the generality of SPAM in a new domain, and also to compare its performance against reference protocols known to work on graph coloring. As above, a driver was implemented which creates the nodes and edges of the graph. We use the layout procedure described in [9] to produce satisfiable graphs of arbitrary size using a defined number of colors. Currently the resulting graph is static, although we will add an additional dynamic component to it in the near future. A separate analysis component evaluates the possible and actual number of coloring constraints, which is then visualized using the graphing component. Three agents have been implemented in this domain, using protocols derived from descriptions in [9].

5.3 SAT

The boolean satisfiability problem, or n-SAT, is another well-studied domain. Our driver for this environment constructs random SAT problems with a specified number of variables, clauses and clause length. It can also read and instantiate CNF formula encoded in the DIMACS format, allowing researchers to make use of the large collections of SAT benchmark materials available online. Individual agents correspond to the variables from the formula, which must then interact and exchange information in some way to find a solution if one exists. As with other domains, an analysis component also exists which can monitor the progress of the search over time.

6 Summary

Farm is a multi-agent simulation environment designed to handle large scale simulations and custom designed analysis, visualization and content while tracking agent activity in simulated real time. The main simulation entity acts as a hub, by accepting and managing connections from distributed plugins, providing execution prompts to those plug-ins, and maintaining a common, globally accessible data repository. Agents in the system are implemented as threads, but are autonomous in character, require communication to interact, and do not share memory. These agents are organized in groups on distributed processors, where their real-time CPU usage is monitored and rationed in accordance with the simulated environment's design. By distributing both the agents and analysis tools in this fashion, Farm is able to exploit available computing power to model very large environments, while retaining the ability to effectively model real world performance.

Issues relating to scale and coherency are closely tied to the distributed nature of the system. On one hand, data flow can hinder scale because state information is stored centrally at the simulation core. On the other, because agents are distributed across processors, care must be take to ensure temporal, data and communication consistency. Different strategies for managing these issues were covered.

The environment so far has been used to create scenarios containing more than 5000 individual agents. Several domains have also been implemented, including a distributed sensor network, graph coloring and SAT. Further information on Farm can be found at:
`http://mas.cs.umass.edu/research/farm`

References

[1] L. Gasser and K. Kakugawa. Mace3j: fast flexible distributed simulation of large, large-grain multi-agent systems. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 745–752. ACM Press, 2002.

[2] J. R. Graham, K. S. Decker, and M. Mersic. Decaf - a flexible multi agent system architecture. *Autonomous Agents and Multi-Agent Systems*, 2003.

[3] S. Hanks, M. E. Pollack, and P. R. Cohen. Benchmarks, test beds, controlled experimentation, and the design of agent architectures. *AI Magazine*, 14(4):17–42, Winter 1993.

[4] V. Lesser, C. Ortiz, and M. Tambe. *Distributed Sensor Networks: A multiagent perspective.* Kluwer Publishers, 2003.

[5] R. Mailler, R. Vincent, V. Lesser, J. Shen, and T. Middlekoop. Soft real-time, cooperative negotiation for distributed resource allocation. In *Procceedings of the 2001 AAAI Fall Symposium on Negotiation*, 2001.

[6] M. G. Marietto, N. David, J. S. Sichman, and H. Coelho. Requirements analysis of multi-agent-based simulation platforms: State of the art and new prospects, 2002.

[7] N. Minar, R. Burkhart, C. Langton, and M. Askenazi. The swarm simulation system: A toolkit for building multi-agent simulations. Web paper: http://www.santefe.edu/projects/swarm/, Sante Fe Institute, 1996.

[8] R. Vincent, B. Horling, and V. Lesser. An agent infrastructure to build and evaluate multi-agent systems: The java agent framework and multi-agent system simulator. In *Lecture Notes in Artificial Intelligence: Infrastructure for Agents, Multi-Agent Systems, and Scalable Multi-Agent Systems.*, volume 1887. Wagner and Rana (eds.), Springer,, January 2001.

[9] M. Yokoo. *Distributed Constraint Satisfaction*. Springer Series on Agent Technology. Springer, 1998.