Proceedings of the 2004 Winter Simulation Conference
R. G. Ingalls, M. D. Rossetti, J. S. Smith, and B. A. Peters, eds.

DATA DISSEMINATION TECHNIQUES FOR DISTRIBUTED SIMULATION ENVIRONMENTS

Bryan Horling
Victor Lesser

Department of Computer Science
University of Massachusetts
Amherst, MA 01003-9264, U.S.A.

ABSTRACT

Farm is a distributed simulation environment for modeling the performance of large-scale multi-agent systems. It uses a component-based architecture to distribute the computational load of the simulation and improve running time. It also supports a global data repository, which permits both actors running in the simulator and external analysis components to generate and use arbitrary pieces of information. Because the components are distributed, the manner in which this data is accessed can have significant effect on the communication overhead and duration of the simulation. In this paper we explore several different techniques for accessing and disseminating this data. Analytic and empirical models of the system's performance are presented, along with an analysis of which strategy is appropriate under different conditions.

1 INTRODUCTION

Farm (Horling, Mailler, and Lesser 2004a) is a distributed simulation environment designed to facilitate the analysis of the quantitative aspects of large-scale, real-time multi-agent systems. Its purpose is to provide the essential base 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 uses a component-based architecture, where individual components have responsibility 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 use local resources.

Like most agent-oriented simulation environments, Farm provides control flow mechanisms and a configurable communication medium. These create a minimalist environment in which agents can run and interact with one another. Although some environments support data collection or instrumentation tools (Gasser and Kakugawa 2002, Minar et al. 1996, Kahn and Cicalese 2001) Farm incorporates a more general global data repository, separate from the local memory used by individual agents and components, which allows entities to generate, store and access information potentially produced elsewhere in the simulation. This capability can be used in several different contexts. For example, it can facilitate the simulation scenario by allowing tasks or environmental cues to be created by a driver component, and then accessed by the agents. It can also be used by evaluation or visualization components to analyze data generated by the agents. The negative implication of this arrangement is the potential for large amounts of data which need to be transferred and stored. For example, in a naive centralized solution, the storage point can quickly become a bottleneck as the size of the simulation grows. Some of this overhead clearly cannot be avoided – if the producer and consumer of a piece of information are segregated. However, there are some strategies which can be employed to take advantage of long-term access patterns which can minimize communication and storage costs. In this paper we will present several such strategies, and show how the performance of the environment is affected by them.

In the following section we will briefly describe Farm's architecture, which will provide a deeper understanding of the problem that motivates this work. We will then outline the challenges associated with implementing Farm's data repository, and describe the different strategies which have been implemented to address them. Empirical results from each strategy are provided in three different domains, and we will conclude with a discussion comparing these approaches.

2 FARM OVERVIEW

Farm is a distributed, component-based simulation environment, as shown in Figure 1. By distributed, we mean that discrete parts of the environment may reside on physically

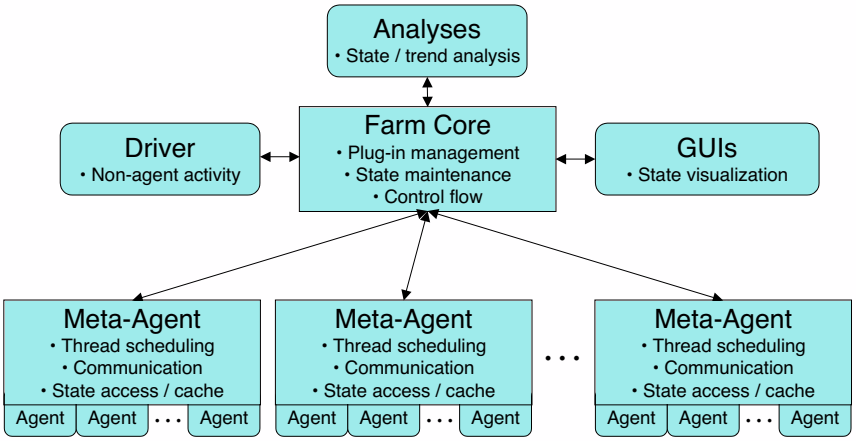


Figure 1: An Example of Farm's Component Architecture

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 those agents or the scenario as a whole. The components responsible for managing agents are called *meta-agents*, as each acts as an interface to and for a cluster of one or more agents. This arrangement is solely for efficiency purposes, it has no effect on the outcome of the simulation, and the individual agents do not know what meta-agent they are controlled by.

Plugin components 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 components may be used for a given simulation scenario. 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. Multiple, different analysis components could be used to capture different aspects of the system. Different environmental drivers could be used, depending on what type of scenario is desired.

Agents are implemented as threads, although this is only for performance purposes - from an agent's perspective they are completely segregated, and are not aware of or directly share memory with other agents which happen to also be resident at the same meta-agent. The agents operating in the system are typically (but not necessarily) light-weight, partially autonomous entities that have a defined goal or role to fulfill within a larger domain context. They may be heterogeneous, either by instantiating the same type of agent with different parameters, or by using different classes of agent. To distribute the load incurred by the agent population, Farm organizes them into clusters, where each cluster exists under the control of a meta-agent component that 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 (similar to (Riley and Riley 2003)). 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 performance in domains where the passage of time matters. Communication actions are similarly modeled and monitored in such a way that message delivery times appropriate for the domain can be simulated.

At runtime, agents are provided time in which to run, and other components (such as the drivers, analyses, and GUIs from figure 1) are given the opportunity to perform, analyze, or modify the simulation at well-determined times. The run cycle is partitioned such that non-agent 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 facilitated by the ability to store and retrieve global state information, which allows any given component to interact with a snapshot of the system's current state. The design and analysis of this data storage will be the focus of the remainder of this paper. More details on the Farm simulator can be found in (Horling, Mailler, and Lesser 2004b).

Farm has been used to model several different domains, including a variety of agents and drivers for distributed sensor network, graph coloring, SAT and reinforcement learn-

ing 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.

3 DATA DISSEMINATION TECHNIQUES

The Farm provides the components in the simulation with a data storage facility with characteristics similar to distributed shared memory. This is not inter-agent shared memory (agents are still assumed to interact via messaging), but instead provides an indirect means of interaction between the simulation components and a way to deliver environmental information to the agents. Using this scheme, components may store and retrieve data (properties) from a functionally common repository, enabling the data produced by one part of the simulation to be used by another. 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, each agent could store some notion of it's current state in, for instance, the property `Agent1:State`. An analysis component could then find all properties `*:State` to capture a snapshot of all the agents in the system.

Distributed data storage is accomplished in Farm through the use of a token system. Each globally-accessible property is associated with a token, which may be resident in the simulator core or at any one of the plugins. The owner of the token for a particular property is responsible for its storage; all reads and writes to that value are performed by or through it. The Farm core itself is responsible for keeping track of who owns the token for each property, somewhat like a specialized directory service. When a property is to be read for the first time by an entity, it optimistically assumes that it is stored at the core, and makes a request to the simulator for it. If it is being stored there, the property's data will be delivered and the process continued uninterrupted. If a different plugin is the owner of that property, the core instead provides the requester with the name of that plugin. This can be used to contact the appropriate plugin and retrieve the data. This property-plugin mapping is then cached so future requests can be made directly to the owner plugin. Because plugins may leave, have their tokens removed, or otherwise lose control of a property, such future requests may fail. In this case the requester will again revert to the default assumption that the core itself is storing the property, since the core will either return the data or redirect the reader as needed.

Property writes occur in a similar manner. If the writing entity has a cached property-plugin mapping, then it will contact the appropriate plugin with the new data to be written. If no such mapping exists, or if the property is controlled by the core, then the simulator itself is contacted. As with reading, the simulator may store the data itself, or redirect the writer to the appropriate owner plugin. Because property owners may change over time, writers will again fall back to the default assumption that the property may be controlled by the core if the local knowledge is out of date. We will assume from here on that the plugins fulfill all of the storage responsibilities; simulation core storage is generally only used as a fail-safe mechanism in case a plugin fails or is otherwise unavailable.

An additional mechanism also exists which allows plugins to be automatically updated with a property's data when it is updated. A list of recipients is attached to the owner's token, so when the property is changed the new value can be automatically pushed to each member of the list. This pushed data is flagged as being cached, so the recipient knows it can safely read from the data, but writes must still be propagated back to the owner. The owner is responsible for ensuring that this remotely cached data is kept up to date.

This token-based scheme is really just a support structure; the more interesting problem is determining which plugin should be assigned to manage a particular property, and what other plugins should be included in that property's recipient list. The difference between the duration of local and remote writes and reads can be as much as two orders of magnitude, caused by the messaging duration and the time needed to encode and decode the information. Testing the average duration of 100,000 sequential, unloaded accesses showed that on average, local reads/writes of a 10 character string took 0.0005/0.0044ms each, respectively. Remote reads/writes took 0.7986/0.8081ms each. As the number of data accesses grows, this overhead has the potential to add a noticeable amount of time to the duration of the simulation. Because the time needed to perform remote accesses can be a significant overhead cost in this framework our objective is to minimize the number of messages needed to support each property, under the assumption that this will similarly minimize the overall time cost associated with that property. The total size of the data being transferred is typically uniformly small enough that it can be ignored. We have explored several different strategies to address this cost. If we view a particular allocation as being constant for the duration of a simulation, this cost C_p^a can be estimated with the following function for a particular plugin a accessing property p.

$$C_p^a = \begin{cases} 0 & a = O_p \\ W_p^a & a \in \overline{T_p} \\ W_p^a + R_p^a & \text{otherwise,} \end{cases} \quad (1)$$

where O_p is the designated owner of the property, and $\overline{T_p}$ is the list of recipients to which the property will be automatically pushed. R_p^a and W_p^a are the number of reads and writes of p by a, respectively. The total cost for property p, including strategy overhead, is then:

$$P_p = (\sum_{a \in A} W_p^a) \times |\overline{T_p}| \quad (2)$$

$$M_p = k\delta(1 + 1|\overline{T_p}|) \quad (3)$$

$$C_p = \sum_{a \in A} C_p^a + P_p + M_p \quad (4)$$

A is the set of available plugins, and $|\overline{T_p}|$ is the cardinality of $\overline{T_p}$. C_p^a only represents the messages needed for a to directly access p; the total cost C_p must also include cost models of the mechanisms which support the distribution architecture. P_p represents the overhead incurred by maintaining p's push list. Each time a value is written to p, that change must be reflected to all members of $\overline{T_p}$. C_p also incorporates M_p, the overhead needed to set up and (eventually) tear down the owner relationship, in addition to the overhead needed to control membership in the push list. k will typically be 2, since a message is needed for both set up and tear down. δ is the number of times ownership is assigned, so if $\delta = 1$ this equation assumes a single owner O_p is assigned and that $\overline{T_p}$ does not change. If $\delta > 1$ and the assignment is allowed to vary, then Equations (2) and (3) are only approximations; a more precise computation of C_p must take into account the costs incurred under each assignment episode, rather than an overall aggregate. Note that δ may also be zero, indicating no remote owner is assigned and the simulation core is the de facto owner.

3.1 Strategies

Our objective is to find an optimal policy for choosing O_p and $\overline{T_p}$ so that we can minimize C_p. Because there are varied access patterns to particular pieces of data, and can be near-continuous access to a single property by different plugins (for example, an agent's state being maintained by one and analyzed by another), it is not clear that a single strategy will be best for all properties. In the extreme, if there is no pattern to the accesses then an equally random assignment of ownership will be as good as any other on average. Fortunately, since particular plugins have specified roles and associated property needs there should be a definite pattern to the accesses. We will evaluate several strategies to see which is appropriate for the types of simulations and plugins we currently use.

Our baseline strategy (C) employs a simple but inefficient centralized assignment. All properties are stored at the simulator core, so all accesses will be made remotely. In this case, $\forall a, p, C_p^a = W_p^a + R_p^a$. We will then evaluate the last writer (LW) strategy, where the plugin which makes the last write access will be assigned ownership of the property. In this case, the owner will change over time. Two analogous techniques are evaluated where the first reader (FR) or writer (FW) is assigned ownership, which does not change over the remainder of the simulation. In other tests, the last writer, first reader and first writer strategies will be augmented with an aggressive push mechanism (LWP, FRP, FWP), where all remote reads will result in the reading plugin being added to the automatic push list for that property. A first-access, pushed strategy was also evaluated, but dropped from these experiments because it was in all cases nearly identical to FWP, since the vast majority of properties are written before a read is attempted.

Another strategy will add a so-called "adjustable" push to the first reader and writer strategies (FRAP, FWAP) using an exponential, recency-weighted usage estimation as a form of online learning. This is done in constant space for each property using the formula:

$$\Omega_{p_{t+1}}^a = \Omega_{p_t}^a + \alpha(\omega_{p_{t+1}}^a - \Omega_{p_t}^a) \quad (5)$$

Recall that during each time step in the simulation a specific amount of processing time is allocated to each agent, and multiple data accesses can be performed by each agent during that time. Let $\omega_{p_t}^a$ represent the amount of overhead-inducing activity associated with property p by plugin a at time t. This value, reset to zero at the beginning of each time step, is incremented by 1 each time p is read, and decremented by 1 each time it is pushed to a. A nonnegative ω indicates that p is being read locally at least as much as it is being pushed to the agent (a desirable condition). α is the step-size parameter ($0 < \alpha \leq 1$), which controls how much the historical values affect the estimate. In these experiments $\alpha = 0.3$. Then, $\Omega_{p_{t+1}}^a$ is the estimated amount of activity at time $t + 1$, with which a can determine if it should change its membership status on p's push list. If that estimate shows that a push mechanism would reduce or exceed remote accesses and overhead, then a push is requested or rejected respectively. In practice, a change is made only when $0 > \Omega_{p_{t+1}}^a > 0.3$ to avoid thrashing. This strategy is similar to the *PoP* (push-or-pull) strategy described in (Deolasee et al. 2001), although we use a somewhat simpler switching heuristic. Note that because $\overline{T_p}$ varies under this strategy, agents may be added to or removed from this push list multiple times. Thus, the cardinality of this set in Equation (3) must refer to the total number of additions made to the list, and not just the number of members it has at any one time. Equation (2) is also approximate in this case.

We will also look at a technique which post-processes the access pattern (S), in an attempt to learn what the most appropriate static assignment should be. In this case, the owner and push lists are calculated as follows to minimize the total number of remote accesses per property:

$$\overline{T_p} = \{a \in A | R_p^a > \sum_{x \in A} W_p^x\} \quad (6)$$

$$O_p = a \in A | (\forall x \in A) C_p^a \geq C_p^x \quad (7)$$

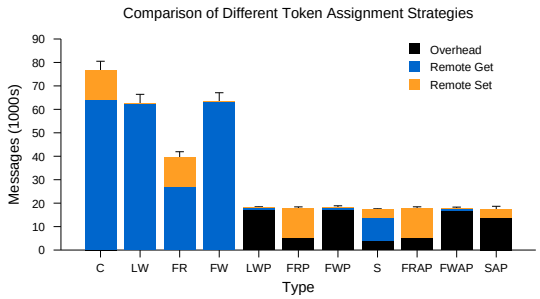
Equation (6) specifies push targets as those which have a greater number reads than the total number of writes, which equals the number of pushes they would receive. Equation (7) selects the plugin which has the greatest cost, taking into account the benefits of pushing which have just been calculated. After selecting O_p, it is removed from $\overline{T_p}$ if necessary. Because this information is known a priori, we can avoid the overhead needed to set up individual push requests. This information is instead bundled along with the ownership message, resulting in fewer messages, at the expense of making a single message somewhat larger. We will also examine the effects of using this learned policy to prime the initial state of the adjustable technique described earlier (SAP).

3.2 Results

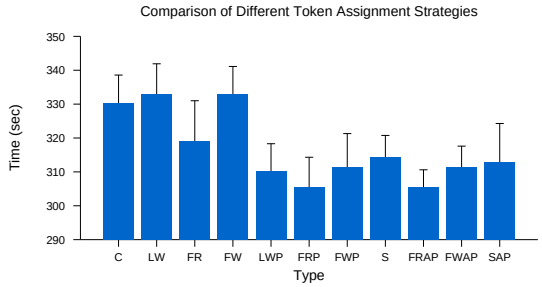
The strategies were evaluated using a distributed sensor allocation simulation, consisting of nine plugins with varying access patterns. Each strategy was tested in 50 different random scenarios, and the total cost in terms of remote accesses and overhead messages determined. Roughly 100 properties, accessed a total of 75,000 times, were managed by the system. Remote accesses include any reads or writes to non-local data, while overhead is the collection of messages needed to support the various schemes, such as ownership notification, push requests and pushed data. The results are shown in Figure 2.

The most obvious conclusion that can be drawn from these results is the reduction in message activity permitted by the push-based technique. All six strategies which employed it produced nearly half the messages of the best technique which did not, and close to four times fewer messages than the simple centralized solution. This was caused by the fact that there were some components (such as the analysis and graphing plugins) which read from properties on every time pulse, even though they were changed less frequently. For example, if a property were updated only every other pulse, the push technique cuts the number of messages in half compared to one which read the value on every pulse.

Also apparent from Figure 2b was the fact that lowering the total level of messaging does not strictly correspond to a reduction in time, even where computational overheads are similar at first glance. The differences lie in the manner remote accesses have been apportioned over the timeline. For example, the learned technique (S), although it has at most an equal number of total messages and minimal runtime computational requirements, takes longer to complete than its counterparts which more aggressively pushed data (e.g. LWP, FRP, FWP). In some cases, an additional measure of parallelism is created, when a group of plugins set or push data to a plugin which would otherwise need to retrieve that data serially. Other differences are observed because of the cumulative effects of minor duration variances between strategies. In these tests, the allocations generated by the S and FRP strategies produced a set of variables which were owned by one of two plugins. One was a reader, the other a writer, so one of the two would be forced to make a remote access for each property at each time pulse. In this particular instance, it was observed that remote sets were slightly slower than remote gets for the same data. These small delays accumulated to produce a significant portion of the overall duration differences between these two strategies.



a)

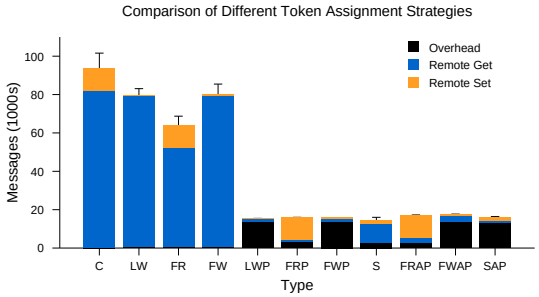


b)

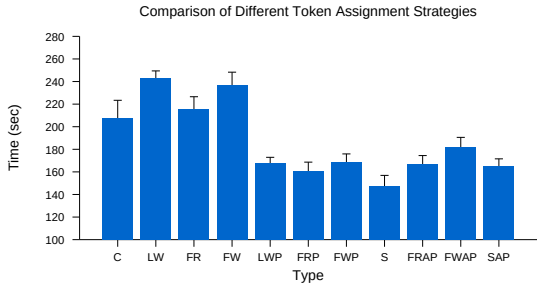
Figure 2: Comparison of Message Activity and Time Between Property Ownership Strategies in the DSN Domain

A Student's t-test was performed across all pairs of strategies. This showed that the FRP and FRAP strategies took significantly less time than other strategies in this domain ($p < 0.005$), while they were indistinguishable from each other. In fact, none of the adjustable strategies FRAP, FWAP, SAP took significantly less time ($p > 0.5$) than their static counterparts. This is a result of the relatively fixed access patterns that take place for each property.

We have also tested these strategies in two other domains with different load patterns. The first is a graph coloring scenario consisting of 12 plugins and 80 agents, which attempt to solve a similarly sized 3-coloring graph using a distributed constraint satisfaction algorithm. Approximately 350 properties were accessed 100,000 times over 100 pulses. The results in Figure 3 show the FRP strategy works well, as does the S strategy. In this domain, the access patterns were such that the post-process analysis produced an assignment that naturally exploited parallelism, by having the analysis component own many of the properties it required. This meant the agent plugins frequently had to remotely set

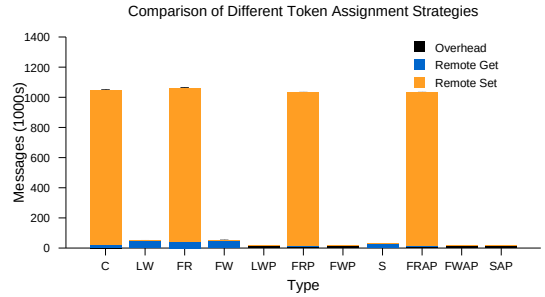


a)

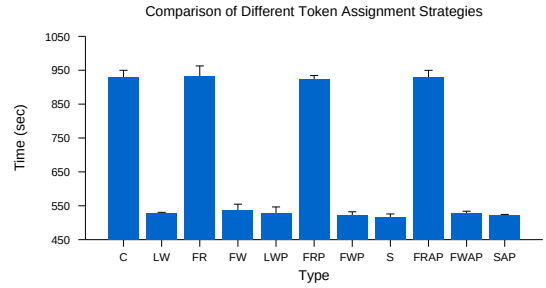


b)

Figure 3: Comparison of Message Activity and Time Between Property Ownership Strategies in the Graph Coloring Domain



a)



b)

Figure 4: Comparison of Message Activity and Time Between Property Ownership Strategies in the Learning Domain

their data, but the bulk reads performed by the analysis component were local, resulting in a net savings in time. The learned (S) strategy which produces this allocation does best here ($p < 0.005$), followed by FRP which also assigned ownership of these properties to the analysis component.

The second is a learning domain consisting of 14 plugins and 500 agents, which are each learning an appropriate policy for a simple, individually random n-armed bandit problem. This managed 1000 properties, accessed over 1 million times over 1000 pulses, but with a simpler usage pattern. Specifically, the use of mobile analysis code allows the bulk of these properties (97%) to be used by just a single plugin. From Figure 4, one can see how dramatically property relocation can reduce the amount of messaging needed to support it, with corresponding decreases in total running time. The peculiar fact that no components read from those properties leads to little differentiation from strategies that push data, and poor performance of techniques where ownership is based on reading. In the latter case the absence of readers means the properties stay resident in the simulation core, so those accesses are all remote. In this case, the previously good FRP strategy is no better than centralizing the data.

3.3 Analysis

The quantitative reductions in time observed under each strategy are shown in Table 1. Based on these tests, the learned (S) strategy, seems to offer the most consistent performance in our typical scenarios, with time reductions of around 5%, 29% and 44% across the three domains when compared to the default centralized strategy. However, it does not it does uniformly produce the best results, it requires the user to generate data needed to perform the analysis, and it will not adapt well to cases where the correct allocations vary widely from one episode to the next. The first reader push (FRP) strategy offers similar or better performance in two out of the three domains with a 7% and 22% reduction in time in DSN and graph coloring respectively, and one might argue that the conditions in the learning domain are atypical. In our opinion, this strategy is the suitable choice for most domains, although there is clearly a need for an alternative under some conditions. The first writer push (FWP) exhibits lower performance in two of the domains, and also suffers from an worst case analogous to that seen in FRP (an absence of writers). This also holds for the writer (LWP) strategy, with the additional caveat that thrashing may occur in domains where two separate writers coexist. Although not seen in these tests, we suspect that in domains where the usage pattern for properties change over time that the adjustable reader strategy (FRAP) may prove best, because it can adapt to changing conditions. However, this is not the case in our existing domains, and the additional overhead usually causes a slight reduction in performance. Both this and the learned technique would benefit

Table 1: Percentage Reduction in Simulation Duration Versus Centralized, Observed Under Each Strategy in the Distributed Sensor Network, Graph Coloring and Learning Domains

Domain	LW	FR	FW	LWP	FRP	FWP	S	FRAP	FWAP	SAP
DSN	−0.8	3.4	−0.8	6.1	7.5	5.7	4.8	7.5	5.7	5.3
GC	−17.1	−5.1	−14.2	18.6	21.6	18.5	28.5	18.9	11.3	19.1
L	43.3	−0.5	42.1	43.0	0.7	43.7	44.4	−0.1	43.1	43.9

from a richer representation of the potential benefits of parallelism and more accurate message delivery costs, which could address some of the deficiencies noted above.

4 OTHER STRATEGIES

There are other techniques that have been developed for content distribution networks and large scale storage systems which in theory are applicable to this environment, but in practice we suspect that the cost-benefit ratio would not be favorable. For instance, hierarchical distribution frameworks (Rodriguez, Spanner, and Biersack 1999) are an effective way of disseminating information. The facilities needed to support such a technique would not be difficult to implement in the existing framework as an extension of the push mechanism. However, we feel that given the typically small number of plugins which need to access a particular property that the reduction in load provided by such a hierarchy would be minimal. Multicast notification would be a useful way to reduce the cost of pushed information if it were supported by the underlying network, but relying on this limits where Farm can be used. A potential solution to this is application-layer multicast (Banerjee, Bhattacharjee, and Kommareddy 2002), but because the underlying protocols are usually still unicast and the distribution target pool is small, the benefits would seemingly be minimal.

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 can lose some measure of coherence. Because Farm is attempting to replicate the behavior of a centralized simulation, data storage should exhibit strong coherence, and components always use the most up-to-date property when possible. In the system described above, coherence is maintained by the single owner for each property, which handles all read and write accesses for the property. These accesses are serialized by the owner, ensuring that consistency is maintained. In the case where data has been pushed to and cached by remote components, the owner is also responsible for immediately updating those components with the new information. Owner coherence is maintained by locking the property for the duration of the update, while remote coherence is limited by the network and processing delay incurred by the update. One could envision a system where a weaker form of coherence might be acceptable to some components (e.g. visualization) in order to reduce overheads. In this case, techniques used to minimize the cost associated with maintaining such a heterogeneous population could be employed, as shown in (Shah et al. 2001). However, due to the relatively small population of plugins in a typical scenario, the costs associated with maintaining such special cases may outweigh the potential benefits.

As alluded to above, Farm also uses Java's RMI and serialization services to support mobile code, for cases where data retrieval bottlenecks are otherwise unavoidable. With this technique, instead of remotely retrieving and locally processing a potentially large amount of data, a specialized function is delivered to and run directly by property owners. This function can be written to use only local data, with the intention of returning a more concise view to the originator. This is particularly useful for analysis components, which frequently need to access data that scales in number with the agent population. For example, the analysis component used in the learning domain computes averages of two properties set by each agent. Because the analysis and meta-agent components are distinct, when running with a population of 500 agents, this will immediately result in 1000 remote property gets or sets regardless of the strategy being employed. We use mobile functions to avoid this by first computing local averages at each plugin, and then transferring only these values. The dominating costs of this technique scale with the number of plugins instead of the number of agents, resulting in substantial savings in bandwidth and time.

5 CONCLUSIONS

Farm's ability to provide global data storage to the simulation participants has both benefits and drawbacks. On one hand, it facilitates state analysis, visualization, and allow agents to easily access environmental or scenario information. However, this can lead to a dramatic increase in the demand placed on the underlying network, with a corresponding decrease in the simulation's speed. This behavior is not unique to Farm, but may be observed in almost any distributed system where nodes are related through dependencies on rapidly-changing data. In this work, we have shown that one can exploit existing, but *a priori* unknown data access patterns to reduce the overhead caused by such

an arrangement. In particular, by using reference locality and push-updates one can significantly reduce the computational and network load incurred by the process.

Simulations in general, and Farm in particular, are differentiated from typical programs in that their data access patterns may be substantially different depending on the components and agents which take part in the simulation, as well as the scenario in which they are run. Consequently, we have observed that no single strategy stood out from the others under all circumstances. Under more dynamic conditions it is possible that a valid strategy may later become unacceptable, although this did not occur frequently in our experiments. We have explored adaptive and learned strategies to address these problems, both of which have strengths and weaknesses. Ultimately, a balance must be found between the complexity and overhead of the solution and the benefits it can provide. The results presented in this paper can provide inspiration and guidance when making this decision.

ACKNOWLEDGMENTS

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.

REFERENCES

Banerjee, S., B. Bhattacharjee, and C. Kommareddy. 2002. Scalable application layer multicast. In *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, 205–217. ACM Press.

Deolasee, P., A. Katkar, A. Panchbudhe, K. Ramamritham, and P. J. Shenoy. 2001. Adaptive push-pull: disseminating dynamic web data. In *World Wide Web*, 265–274.

Gasser, L., and K. Kakugawa. 2002. 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*, 745–752. ACM Press.

Horling, B., R. Mailler, and V. Lesser. 2004a. Farm: A Scalable Environment for Multi-Agent Development and Evaluation. In *Advances in Software Engineering for Multi-Agent Systems*, ed. A. G. C. Lucena, J. C. A. Romanovsky, and P. Alencar, 220–237. Springer-Verlag, Berlin.

Horling, B., R. Mailler, and V. Lesser. 2004b. The Farm Distributed Simulation Environment. Computer Science Technical Report 2004-12, University of Massachusetts.

Kahn, M. L., and C. D. T. Cicalese. 2001. CoABS grid scalability experiments. In *Proceedings of the Second International Workshop on Infrastructure for Scalable Multi-Agent Systems at Autonomous Agents*.

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

Riley, P., and G. Riley. 2003. SPADES — a distributed agent simulation environment with software-in-the-loop execution. In *Proceedings of the 2003 Winter Simulation Conference*, ed. S. Chick, P. J. Sánchez, D. Ferrin, and D. J. Morrice, 817–825. IEEE, Piscataway, NJ.

Rodriguez, P., C. Spanner, and E. W. Biersack. 1999. Web caching architectures: Hierarchical and distributed caching. In *Proceedings of the 4th International Web Caching Workshop*.

Shah, S., A. Bernard, V. Sharma, K. Ramamritham, and P. Shenoy. 2001. Maintaining temporal coherency of cooperating dynamic data repositories. Computer Science Technical Report TR01-52, University of Massachusetts at Amherst.

AUTHOR BIOGRAPHIES

BRYAN HORLING is a Ph.D. candidate at the University of Massachusetts, working in the Multi-Agent Systems Lab. He received his MS from UMass in 1998, and has bachelor's degrees in computer science and biology. His research interests include organizational design, agent control architectures, simulation frameworks and distributed computing. His email address is <bhorling@cs.umass.edu>.

VICTOR LESSER received his Ph.D. from Stanford University in 1972 and has been a professor of computer science at the University of Massachusetts at Amherst since 1977. He is a founding fellow of AAAI, and the founding president of the International Foundation for Multi-Agent Systems. His major research focus is on the control and organization of complex AI systems. He has been working in the field of Multi-Agent Systems for over 25 years. Prior to coming to the University of Massachusetts, he was a research scientist at Carnegie-Mellon University where he was the systems architect for the Hearsay-II speech understanding system, the first blackboard system developed. His email address is <lesser@cs.umass.edu>.