4-23-2008

MECA: A Multi-agent Environment for Cognitive Agents

Coleman Phillip
Trinity University

MECA: A Multi-agent Environment for Cognitive Agents

Phillip Coleman

A departmental thesis submitted to the
Department of Computer Science at Trinity University
in partial fulfillment of the requirements for Graduation.

April 23, 2008

Thesis Advisor

Department Chair

Associate Vice President
for
Academic Affairs

MECA: A Multi-agent Environment for Cognitive Agents

Phillip Coleman

Abstract

Many fully functional multi-agent systems have been developed and put to use over the past twenty years, but few of them have been developed to succesfully facilitate social research through the use of social agents. There are three important difficulties that must be dealt with to successfully create a social system for use in social research. First, the system must have an adaptable agent framework that can successfully make intuitive and deliberative decisions much like a human participant would. Secondly, the system must have a robust architecture that not only ensures its functioning no matter the simulation, but also provides an easily understood interface that researchers can interact with while running their simulations. Finally, the system must be effectively distributed to handle the necessary number of agents that social research requires to obtain meaningful results. This paper presents our work on creating a multi-agent simulation for social agents that overcomes these three difficulties.

Acknowledgments

The author would like to thank Dr. Yu Zhang for her inspiration, guidance and support on this project, and Dr. Mark C. Lewis and Dr. Berna L. Massingill for their helpful discussions and feedback as I formulated my ideas.

MECA: A Multi-agent Environment for Cognitive Agents

Phillip Coleman

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

The field of multi-agent research has exploded over the past decade as new applications have been discovered. As technology has evolved, and computers have become faster, the use of agent systems to handle computationally intensive processes has become feasible. Thus, in recent years we have seen numerous and distinct applications of multi-agent systems. While more mainstream than before, the majority of agent systems are utilized to deal with task management in a purely computer or network oriented system, or they are utilized as a machine learning approach to solving computationally hard problems. While worthwhile in themselves, these applications fail to utilize the complete potential of MAS.

The inherent nature of a system composed of large numbers of autonomous decision making agents makes it well suited for social simulations. By utilizing an agent architecture that imitates the human decision making process, social scientists are able to test hypotheses that were untestable in the real world. These virtual societies allow the researchers to analyze and predict complex phenomena that arise in real world societies that often take

too long to develop in the real world. Thus, by controlling the initial settings of these societies, and by modifying the external rules that govern the agents decisions, these multi-agents systems become virtual laboratories for these researchers to play God. With the potential in using multi-agent systems to gain worthwhile results from these simulations, the researchers have to use an agent architecture that has been proven to be adequate and a system that is designed to handle a spatial social system effectively.

An effective architecture for these types of simulations would necessarily need to be able to combine traditional rational models of decision making with a much more cognitive approach that would model humans' ability to make snap decisions. Closely related to this is the need for an intuitive approach to dealing with social situation, instead of relying on formal strategies and top down organization. Currently, there are only a few agent architectures that have shown promise in simulating the human decision making approach; these include CASE (Cognitve Agents for Social Environments) [29], CODAGE (COgnitive Decision AGEnt) [21], and COGENT (COGnitive agENT) [10]. All three of these show potential in accurately modeling the human thought process in group situations. While these satisfy the first requirement, a system that is designed to not only effectively handle these type of agents but to also be approachable for social scientist has yet to be developed.

Of the existing multi-agent systems, such as RePast and Cougaar, few systems have both encompassed the need to be robust and distributed, as well as provide the approachability that the social researchers require. In the end these systems are unable to run the type of complex social systems required of this type of research. The key to social simulations, and the part that makes them the most difficult of multi-agent applications, is their need for realism. If the system does not accurately reflect the real world, then the results are for the most part inconclusive. Therefore the agents not only have to use a decision framework that is complicated enough to handle the peculiarities that we find in humans, as well as handle

enough agents to accurately model many of the social structures that these researchers are interested in. This translates to a need for serious computational power to handle the simulation. With these requirements only a highly distributed system would be able to handle running a simulation the size that is required of this social research.

The system would also need a sufficiently intuitive interface for the researchers as well. Since most social scientist do not have a background in computer science they are unable to deal with large and esoteric systems such as Cougaar. Thus, to allow them to effectively use a multi-agent system as a tool for their research, a simple and user-friendly interface needs to be developed. An important component of this user interface is the visualization of the data. Social science results are often times complicated and numerous, thus a tool to interpret and visualize this data would be required for the system to be truly effective as a Social Multi-Agent Simulation.

1.2 Approach

To create this type of system, a robust design pattern is needed to ensure it's stability and scalability. For this we turn to the Service-Oriented Architecture (SOA) [2], often used through out the industry for large flexible systems. The main principle behind SOA, is to have the different tiers of functions loosely coupled, in other words encapsulated. This modularity allows for the users to easily switch out different functional modulars or to upgrade them separately. Not only does this improve the system's flexibility, allowing the user to adapt it to their needs, but it also provides the system a robustness. By using a modular design as defined by the SOA pattern, we separate the functionality which significantly decreases the chance of problems with interdependency issues. Also this allows us to easily swap out the non-function section and replace it with an existing working

module. For these reasons, the SOA pattern was the most logical choice to base the design of MECA.

We split the functionality of the system into three tiers based off their tasks and locality within the system. These tiers consists of:

- The Client Tier
- The Middleware Tier
- The Data Tier

The client tier encapsulates all the different aspects of the user interface. These aspects include manging simulations, managing the underlying hardware, collecting data, and visualizing and analyzing the data. Together these aspects can easily overwhelm people who are unfamiliar with the system, such as the social researchers it is intended for. With this in mind, the interface has been designed for ease of use. This includes an intuitive layout to allow for ease of access. Also, the client tier consists of the visualization tools that can be used to interpret the data. For this a tool named MASVis (Multi-Agent Systems Visualization) was developed for data analysis and plotting. Similar to its predecessor SWIFTVis, MASVis allows for the flexibility to choose a wide selection of data sources, filters, and plots. Along with this, the system allows for the integration of more traditional social science visualization tools, such as GIS (Geographic Information Systems) software. This interface not only is user friendly for non computer-science users, it allows for access and implementation of the system from any computer on the network, allowing for a user to start a simulation from his office, on a cluster that may not even be in the same building.

The middle tier consists of the main distributed system. Since we wanted the system to be flexible, a simulation can be run over any number of computers as long as they

are connected through a network. This allows users to fully utilize whatever resources are available to them. Also, the system was designed to effectively distribute the work load using a geometric partitioning. This allows the system to scale effectively even when new machines are added while the system is running. Along with this the system uses multithreading when it would improve performance, such as on multicore or multiprocessor machines.

The data tier is separated from the middle tier to ensure the stability of the data. By collecting the data in a separate server, we can avoid data loss in case a problem arises in the main system. Also by having it not integrated into the system it allows for access to the data without the extra step of going through the middle tier. Allowing for multiple back ends for collecting the data, the user can decide what features he would like most, some of which are the use of meta information to make searching through data easier, automatic backups, and the integration with other data analysis software, including MASVis.

When a simulation is started by the end user using the client tier, the middleware tier begins the timestep cycle which runs the number of iterations that was specified by the user. Each time step represents a single unit of time where the agent can percieve and act on the environment in parallel with the rest of the system. Each timestep consists of:

- Delivering messages to the agents
- Updating the agents perceptions
- Allowing the agents to decide on an action
- Implementing those actions
- Collecting any new messages that the agents may have created

These actions are required to be seperated to ensure that agents do not percieve, interpret, or act on an environment that is different from the one available to the other agents. This five part synchronization during each time step is essential to ensure the validity of the over all simulation.

Due to the computational requirements of social simulations, caused by their need for large numbers of complex agents, the use of a distributed system becomes a necessity. By distributing the system, however, multiple concerns arise. The most problematic of these is an imbalance of work load between the servers, where one or more machines carry out much more work per step than the others. Load mismanagement fails to completely utilize the potential of the cluster, and thus slows the simulation down significantly, turning what might have been a few minutes into hours.

Traditionally, designers of multi-agent systems have turned to other disciplines when they ran into problematic aspects of the simulation. This is no different for the load-balancing problem, as most techniques are borrowed or adapted from other fields. While oftentimes a working stopgap to the problem, these techniques failed to take into account the intricacies of a multi-agent system, especially one that is designed for social simulations.

The predominant load-balancing technique for most distributed applications is a job migration strategy, that moves jobs from the overworked machines to the less worked machines. This works well when dealing with large distributed systems, but in simulations agents are not equivalent to jobs. This often means they are dependent on the local data near them, thus moving them would negate their use as it would only serve to severely increase the network communication load, as the agent acts on it's environment which is now on a seperate machine. This schema also does not take into account the abundance of intra-server communication between agents. The network lag caused by this communication oftentimes slows the system down more than the computational load. A migration schema

like this could lead to an overall increase in communication if it purely focused on processor load, thus unbalancing the system even further.

To deal with this we developed a load balancing technique that has four unique components to it:

First, it uses recursive bisections [4] to split the geometric space of the environment between the machines. This ensures that the agents are on the same machine as their local environment, and thus avoids dependency issues of when the agents are removed from the local data they need. Recursive bisections also allow us to easily move the partitions as a way to manage the load, which is not only efficient as it does little computation to determine which agents need to move, and only moves the necessary agents around, but it also preserves the validity of the simulation, which is crucial to the end user.

Second, the schema uses a proactive approach that predicts the movement of the agents based off past vectors. This allows the system to move the partitions incrementally. This approach allows for the system to move fewer agents between machines each time step, and also avoids the situation where the system is merely reacting to unbalanced loads, and thus not preventing them in the first place.

Third, we use a pseudo-migration technique to deal with communication. Since the partitioning is purely spatial it is hard to use it to gauge and make adjustments to the communication work load. Migration is usually used to move agents that communicate often onto the same server, thus eliminating intra-server communication, but since we are dealing with spatial social simulations, the environment and the agent can not be separated like this without risking the validity of the simulation. To deal with this we use a pseudo-migration, where individual servers keep track of the machine location of agents that are communicated with often. This allows the servers to send to communication directly, and avoids having the communication routed through another machine and the extra computation to determine

the machine to send the message. By avoiding this extra processing we significantly reduce the overall communication work load.

Fourth, a multi-agent social system deals with agents that rely heavily on shared data that is either found within the environment or that is global knowledge to the entire system. With any distributed system, shared data of any sort becomes difficult to deal with. To manage this we are using shared data entities that get copied to each machine every step. This allows every agent to have a local copy it can observe and modify; at the end of each step, the changes to the data are synchronized and a new copy is sent to each server. This ensures proper functioning as well as performance.

Together these four components form a robust, efficient, and effective technique to balance the load in highly distributed multi-agent social simulations.

1.3 Context

MECA is a part of the Distributed Intelligent Agent Systems (DIAS) Lab [13]. Dias, consists of professors and students from Trinity University and has the goal of developing systems and agent architectures to aid in cross-disciplinary simulations. While MECA has been based on past work done by other members and has been a group effort, as it plays a crucial role in facilitating these simulations, the author is responsible for the majority of the middleware tier, the entirety of the client tier, and the development and implementation of the load balancing strategy. MECA, along with other projects in DIAS, continues to be used and improved upon to promote the use of multi-agent simulations for social research.

1.4 Layout

The rest of this thesis is organized as follows: Chapter 2 covers the related work in this area, and it's applications to this project. Chapter 3 formally introduces the design of the system and goes into more depth about how the SOA pattern is fully implemented, and its benefits. Chapter 4 lays out the intricacies of our load balancing techniques including its geometric partitioning, proactive load balancing, pseudo-migration, and effective sharing of data. Chapter 5 introduces and discusses a number of experiments run to show the effectiveness of the system and its load balancing capabilities. Chapter 6 concludes the paper and provides direction for potential future work and research.

Chapter 2

Related Work

2.1 Agent Simulations

There have been many successful multi-agent systems that have been designed for numerous reasons, each with its own specific features and benefits. Though numerous, none of these systems provide the necessary features required for social research using large-scale simulations. A run down of the most commonly used systems is provided below.

RePast [33], developed at the University of Chicago, is perhaps one of the most feature filled system available currently. Ran as an open source project, RePast includes templates for easy construction of agents, machine learning libraries, and integrated data visualization tools, and is supported on most of the major operating systems. This extensive development, though, has lead to a very restrictive environment. End users are forced to use the built-in tools to try to model the behavior they are seeking, the process of which is only further complicated by the sheer number of different libraries that are included in the package. Oftentimes, the user will not even need the majority of the features, and is thus forced to use an overly bloated system. MECA on the other hand, using the SOA style, only includes

the features necessary for the users applications. Also unlike RePast, MECA can also be run between machines using different operating systems, thus increasing its flexibility even further.

The multi agent system named MASON [26] is a lightweight system developed at George Mason University. MASON does have the advantage that it is small unlike RePast, and more likely to run faster on most systems. MASON is also package with a 2D and 3D visualizer, so that the user can watch a real time visualization of the system. While definitely a nice feature, the lack of the type of more detailed visualization tools that are used for social research, such as GIS, makes MASON unacceptable for this type of research. Adopting the lightweight mentality through the use of SOA, MECA also provides a built in visualization tool and the ability to add additional tools as needed, and is thus a better option for running social simulations.

To handle the computational complexities of social simulations it is a necessity to have a scalable distributed system. JADE [19], an open source package, was developed with exactly this in mind. Written in Java, the platform can be deployed across any number of machines running any operating system[3]. JADE also provides a GUI that can be run separate from the main system allowing for access to the simulation from anywhere. We included both of these important features into MECA. Like MASON, JADE suffers from a lack of visualization tools, and provides no means to integrate with existing tools. MECA remedies this situation by creating a seamless work flow for the end user. They can go straight from running the simulation to analyzing the results.

Swarm [40], a system developed by the Santa Fe Institute, is another well known and frequently used multi-agent system. Based on the extensive work in agent systems done at the Santa Fe Institute, Swarm consists of a framework and system for designing and implementing agent based modeling experiments using a concurrent object engine allowing

for the continuous running of all agents. This approach creates an intermediate level between the simulation code and the machine that represents the software as swarms and sub-swarms of agents. This approach is great for the simulation and experimentation of swarms, which consists of mostly dumb agents, but is unsatisfactory with dealing with much more complex agents that are found in social simulations. Swarm also lacks the necessary functionality to streamline the research, providing only the system itself.

Cougaar [8] was originally developed by DARPA for use by the military. Designed to allow for massive simulation across large networks, Cougaar is a very robust system. In fact it has very strong fault tolerance built in, allowing it to lose entire sections of the network, without compromising the system. While very robust, Cougaar was not designed for simulations, instead emphasizing the use of multi-agent systems as problem solving tools. Therefore, it lacks the ability to represent and deal with the virtual space involved with a simulation. This, combined with its overly complex interface, makes it an unsatisfactory candidate for social research.

2.2 Architectural Styles

With large scale software systems, it is often useful and sometimes a necessity to use existing and proven architectural styles [5]. Originally introduced by Shaw and Garlan [38], these styles are a way of abstractly representing the components, interactions, and constraints of the system. As a way of formalizing and abstracting reoccurring successful software design practices, these architectural styles are thought to be key to creating successful large complex systems [30]. Many higher level architectural styles such as client-server, pipes and filters, blackboard, and layers have already been adopted and implemented in many existing large scale software systems with much success. These architectures allow for much more

robust systems and improve the overall design process by allowing the reuse of previous patterns and code segment [5]. Once we take into account the scale of the type of highly distributed multi-agent system required for these social simulations and the complexity involved with the interactions between the agents, the use of an architectural style to help design the system becomes a logical conclusion.

Existing large scale multi-agent systems have used a variety of different styles, each with its benefits and drawbacks. Two of the most commonly used styles are the layered approach [22] and the centralized synchronous approach [11].

The layered approach as utilized in multi-agent systems split the system into layers of functionality. Thus, the system might be composed of an object layer, a distributed processing layer, an agent layer, an optimization layer, and a problem-solving layer each building on the structures and functionality of the lower layers. While this style is effective when dealing with isolated problem-solving intelligent agent-systems, its structure makes it difficult to represent aspects crucial for social simulations. In the case of the agent and the environment, neither one is based off the functionallity of the other, but both are mutually important and exclusive groupings of functionality in a social simulation. The layer approach has no way of effectively and easily modeling the relationship between these two. Also, since each layer is completely dependent on the functionality of the previous layer, this style lacks the modularity of functionality that is desired within a large scale social simulation.

Another commonly used approach, the centralized synchronous system, consists of a set of components that are synchronously controlled by a overarching hierarchical component. Each lower level component relies on commands sent from the central component before it is allowed to act. This architectural style is useful when there is a need for exact control over the timing of actions and the use of resources. This centralized control also allows

for better encapsulation of the lower processes, which makes switching them out relatively easy. While both of these features are beneficial in a social simulation setting, the centralized approach suffers from an inability to scale effectively. Since all the computation and communication must go through a single node, this creates a bottleneck effect, which in the end kills the effectiveness of the overall distributed network. Thus, for a social simulation, where scalability is essential, the centralized style becomes an inappropriate approach to the software.

Since the traditional approaches have been shown to be ineffective for systems designed for social simulations, we turn to non-traditional styles, specifically the Service Oriented Architecture approach. The SOA style is built on the principle of modularity, separating functionality into encapsulated modules that can operate independently [2]. By encapsulating the functionality like this it allows for the ability to switch out individual modules depending on the needs of the user. This architectural style is often used to make business and web applications because of its interoperability and flexibility as far as adding and removing services [15]. While rarely used in multi-agent simulation, the SOA model has numerous benefits for a multi-agent social simulation system. 1)*SOA stresses the loose coupling of its encapsulated services*; in other words the system is designed so that each service has as few dependencies on other services as possible. This loose coupling of services allows for individual service to be distributed to different machines without worrying about difficulties with interdependencies [15][2][1]. This is crucial in the system we are designing. Not only does it cater to the need for MECA to be highly distributed, but it also allows the end user to be able to control the system from any other computer, which is highly desirable for the social researchers who will most likely be the end users for the system. 2)*SOA also allows for optimization based on service importance [1].* This allows for certain services to be weighted more highly, and receive more computational time than others.

This is beneficial to a system intended for social research. This way the main system can be given priority during the simulation, while the visualization can be given priority while the results are being viewed and interpreted. The SOA style gives the flexibility as well as the robustness that is needed with a highly-distributed system like MECA.

2.3 Load Balancing

The need for the computational power of a distributed system when dealing with simulations the size required for necessary agent populations for social research is evident [28][34]. Distributing large-scale computations across a network introduces a collection of concerns that not only threaten the validity and performance of the simulation but also the skill of the developer [27]. Out of these problems, the most serious is that of the imbalanced load. This single issue can slow the simulation to a crawl. It comes as no surprise that there has been so many different approaches to the problem of load-balancing.

Traditionally, distributed user systems research has considered the cause of load imbalances to be an inherently user oriented problem [20]. The entire cluster of machines is often considered to be a common computational resource available to and shared amongst a large number of end users each with their own and often competing objectives [18]. Load balancing techniques employed here are based on the concept of job migration, either pausing or moving jobs from the unbalanced machines to the under utilized ones. While there have been a wide array of approaches on how to optimize this method of load balancing, [9][17][23], most have failed to translate to a solution for multi-agent simulations.

One of the most commonly used technique for load balancing is the migration of computationally heavy agents from one machine to another. This relies heavily on the traditional approach to load balancing in non-agent based distributed systems [18]. One of the inherent

aspects of an intelligent agent system is that there is often a large amount of communication between the agents every step, and when these agents are on multiple machines this creates an increased demand on sending message over the network. This intra-server communication, often times, makes up the majority of time delay on the system. Thus, it comes as no surprise that the majority of existing multi-agent load balancing techniques consist of attempts to deal with this communication issue. These techniques treat the agents as if they were jobs in the traditional distributed systems; they migrate agents from one machine to the other. Instead of moving the agents to under utilized machines, migration techniques utilize a heuristic that calculates which agents are sharing the most communications, and attempts to migrate the agents to the same server, thus eliminating the intra-server traffic. While this works well in most multi-agent systems [18], this approach does not take into account the spatial aspects of social simulations. Oftentimes in these social simulations the agents have strong connections with their local environment and the agents and data that it contains. If these agents are migrated away from their local environment, the results may end up not being accurate to the original question.

In the past when faced with difficult problems multi-agent system researchers have always turned to other disciplines for answers: reappropriating tried and proven methods to help solve their own problems[24][32][35][37]. Since traditional approaches are inappropriate for social researchers, we must turn to other distributed spatial systems to see what approaches we might be able to adapt. In distributed physics simulations dealing with particle interactions, the spatial element plays a significant role in the actual interactions, thus new methods of distributing workload based on segmenting the environment have been developed [12]. Out of these approaches two deal with simulations most closely related to those involved with social research: space-filling curves and recursive bisections.

The space-filling curve (SFC) approach [31] consists of using the nth iteration of a space-

filling curve, a curve whose ranges cover the entirety of the 2-dimensional unit square, such that the curve intersects with the position of every agent. The agent's location on the SFC is mapped to a SFC key. These key values are then sorted by value, and then split so that each machine is assigned an equal number of keys. This approaches effectively partitions the geometric space, such that the work load is equally distributed. This approach suffers from an important assumption that can not be translated to distributed multi-agent systems. This assumption is that there is a homogeneous work load between all the agents. While this is very true in a particle system, social simulations consist of heterogeneous agents whose work load may vary over time. Therefore, using the SFC approach is not the best fit for social simulations.

The recursive bisections (RB) approach [4][39], is oftentimes considered one of the simplest approaches, but it is also the fastest [12]. This approach consists of making alternating segmentations of the environment, exponentially growing the maximum number of enclosed partitions. This continues until the number of partitions is reached. Though this is a static load-balancing approach as it creates these segmentations only once when the simulation is starting, these segments can be dynamically moved to balance the workload while the simulation is running. This approach has multiple benefits for our system. First, it does not assume any particular behavior of the agents, thus allowing its application to non-traditional simulations such as found in social research. Secondly, the RB approach's simplicity helps to improve performance and decreases the computational strain of balancing the system, thus allowing it to be done more often. It is for these reasons RB was successfully applied to MECA.

Chapter 3

Design and Implementation of MECA

The diversity of systems and problems in the social sciences that can be explored using multi-agent systems requires that software be developed that will flexibly deal with change and allow for the easy integration of new agent types and governing rules into the architecture. This architecture must also be designed with scalability in mind, allowing for the system to be effectively distributed across a parallel network. Along with this, the system must allow for multiple types of analysis tools to connect to the simulations and process the outputs from them, as to be applicable to the specific needs of the end users.

3.1 Service-Oriented Architecture

The first step in designing the system was deciding on an architectural style; for this we decide on the Service-Oriented Architecture (SOA). SOA is an architectural style commonly used for large scale software systems. SOA has been used in many large corporations for

their enterprise software, including such companies as IBM and NASA. It provides a flexible and stable architecture for large scale software systems. For these reasons SOA provides a good fit for a distributed multi-agent social simulation.

SOA requires that services should be loosely coupled, in other words encapsulated with as few dependencies on other services as possible. This encapsulation allows for services to be easily swapped and their implementation to be changed or upgraded. This modularity is beneficial to MECA as it allows for services to be swapped out as they are needed for specific simulations or experiments.

For our system we used a three tier approach as seen in Figure 3.1.

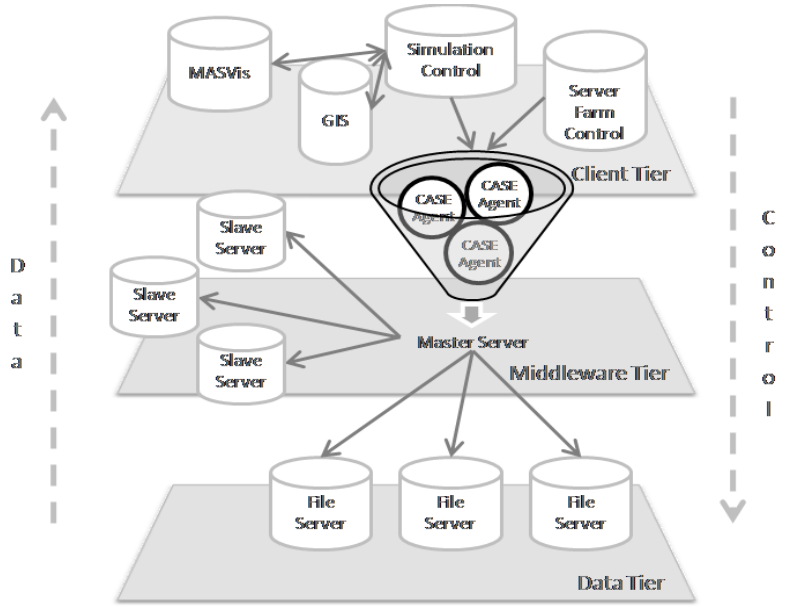


Figure 3.1: Shows a three-tier SOA diagram. The services that we are constructing are highly component based so that they can be easily extended.

The client tier consists of elements that are used to control the server and simulation aspects of the middleware and data tiers. These elements are loosely bound to the simulation engine through the use of the Bridge design pattern[16]. The bridge design is usefull here

as it utilizes interfaces to bind seperate and unique processes, thus avoiding hardcoding dependencies between the two services. This allows for other GUIs to be built and used in place of the ones that we have currently constructed as response to specific needs of the end user. Currently we have constructed four interfaces.

1. MASVis, a tool that was developed for visualization and data analysis, since they are two of the most significant challenges in dealing with large scale simulations [7].

2. A Geographic Information System platform, GIS, that will be interfaced with our middleware. This interoperability with GIS is essential for many social science projects where the use of GIS has grown explosively.

3. A simulation control unit that provides feedback on what is happening in a particular simulation, including load balancing and spatial decomposition between the slaves. This user interface also also provides a method for displaying settings specific to the experimental domain code so that users can alter those settings or explore certain aspects of the simulations while they are running.

The client tier will be explored in more detail in Section 3.3.

The middleware tier contains the core engine and infrastructure for the multi-agent system. The core system is made up of the collection of master and slave servers as well as the CASE agents that occupy them. By encapsulating the rules for the agents into modules, the can be dynamically loaded into the system at run time. This makes it easy to explore new problem domains, an essential requirement for this project because of the diversity of problems that are of interest in the social sciences. The middleware tier will also consist of the service providers that help the middleware tier communicate with both the client tier as well as the data tier. These consist of extensive use of the Bridge pattern to create

interfaces to facilitate communication between the main tiers of the architecture. When entire tiers are switched out, these interfaces are the only portion of the code that will need to be marginally modified. We have developed three experimental domains: Foreclosure, the Stock Market, and a Load Balancing Stress simulation. Foreclosure is a domain we developed for helping social scientists to analyze the nationwide foreclosure crisis problem. The Stock Market simulation simulates the interactions of intelligent agent brokers, while they are buying and selling in a virtual stock market (refer to Chapter 5 for more details).

The data tier is the last tier of the system. The data tier helps encapsulate the data, and stores it separately from the simulation running in the core system; this allows for the physical separation of the data tier from the rest of the system by placing it on a separate server. This allows us to ensure the stability and validity of the data. Even if the system crashed the data would still be preserved and remain uncorrupted. The encapsulation also allows for direct access to the data using visualization tools, which stream lines the over all research process. The data and results from the simulation are saved for later analysis as binary files. This format is faster and less space intensive than using text, which is essential when you are dealing with massive amounts of data from large-scale social simulations.

Using these three tiers will help ensure that our simulation is robust enough to allow for any part of it to be updated with little or no difficulty. Also the modular nature of the SOA allows for other researchers to use only the parts of the system that they find to be appropriate for their own project.

Next, we explain in detail the middleware tier, which focuses on the system distribution, and the client tier, which describes the user interface.

3.2 Middleware Tier

For the system to be effective at dealing with the large number of agents required for social research, it has to be highly distributed both in processing as well as memory. To achieve this we developed a Master/Slave system, as shown in Figure 3.2.

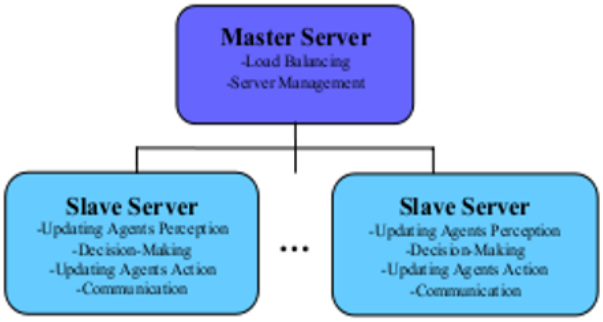


Figure 3.2: Master/Slave Relationship

The master server is in charge of initializing and synchronizing the other servers (slaves). The master's responsibilities include synchronizing each step, facilitating agent communication and agent movement from one server to another, and most importantly load balancing. This process consits of:

- Start the slave servers' timesteps
- Manage and deliver communications
- Balance the system through the use of geometric partitioning (see Section 4.2)
- Synchronized the shared data and redistribute it to the slave servers

The master server after receiving connections to all the slave servers through an RMI connection, begins the simulation, by sending a synchronized start signal to all of the

slaves. The master server then waits until all servers have sent the signal signifying that they had finished that time step. This ensures that all the slave servers are on the same time step. This is a significant requirement for social simulations, as often there are interactions between agents on separate servers, and if they are not synchronized these interaction would be invalid.

The master server also acts as a delivery system for communications. The slave servers return a list of messages along with the end signal at the end of each steps. The master server then sorts these messages by their intended recipient, and using a hash map, which maps agents ID's to their current servers and which is updated every step, sends the messages to their appropriate servers for delivery to the specified agents.

The other important task the master server handles is the balancing of the overall network. It achieves this through proactively moving around geometric partitions, thus changing the overall work load. This subject is covered in much more detail in Chapter 4.

The slave server is initialized with a given bounds and a list of agents, which effectively distributes the memory initially and avoids the bottlenecking effect that often happens if the master server is forced to pass the agents to the slave servers initially. After this the slave server starts the time step cycle every time it recieves the start signal from the master server, this cycle consists of:

- Deliver communications
- Update all the agents' knowledge for what they perceived that step
- Have all the agents implement their decided action for that step
- Collect and new communications that need to be delivered
- Manage Ghosting (see Section 4.4)

- Check Communication Threshold Levels
- Create new ghosts
- Remove unneccesary ghosts

The slave server utilizes the maximum number of threads that the machine it resides in can handle to run this cycle, often updating multiple agents in parallel. After this cycle finishes the time to complete the step, the messages gathered, and the end signal is sent back to the master server.

3.3 Client Tier

In creating the user interface, the primary consideration is that users will not be concerned with all of the system's features at once. Furthermore, there will not be enough screen space to conveniently and aesthetically portray all of these features, specifically the settings, server, data, graphical, and simulation windows, at once. To house each element of the system's features in detached windows, able to be hidden and then restored whenever needed, is therefore clearly plausible and arguably necessary for such an open-ended agent simulation. This can be argued for each individual element of the system's features.

Once a user calibrates the appropriate settings for their simulation implementation, they may not need to keep these settings in focus throughout the running of the simulation; rather, the user may be more focused on the system's output and the results of the simulation. To keep these settings on the screen for the duration of the simulation would force the user to work with cluttered space, constantly having to move windows around in order to see and collect data. To house all of the simulation settings in detached window, able to be hidden and then restored whenever needed, is, therefore, appropriate for such a wide-breadth multi-agent simulation.

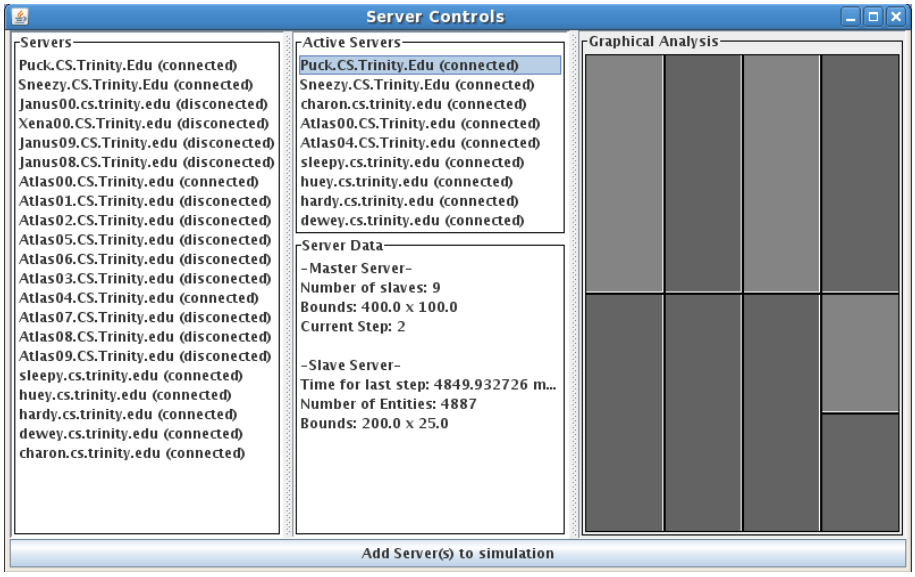


Figure 3.3: The interface for controlling the servers

This same logic can apply to the server window, where server hosts can be added, managed, or booted. Clearly, server management is not the primary focus of any type of simulation. While servers need to be added at the beginning of a simulation and monitored and/or booted throughout the duration of the simulation, the server window need not be opened and visible the entire time. Again, the detached and hide-able window system works effectively in this situation. This approach can be seen in Figure 3.3.

This detached-window argument can also be applied to output-based elements of the simulation user interface. Consider when a user is focused on only the raw textual data output of a simulation. Clearly, any visual representation of the data in question, whether it is a graphical representation of this data or a step-by-step visualization of the individual agents in the system, would be extraneous and unneeded in the given situation. Obviously, this scenario can be flipped into a situation in which the user is only focused on visuals, in which case the raw textual data output would be unneeded. Similar to the case of the

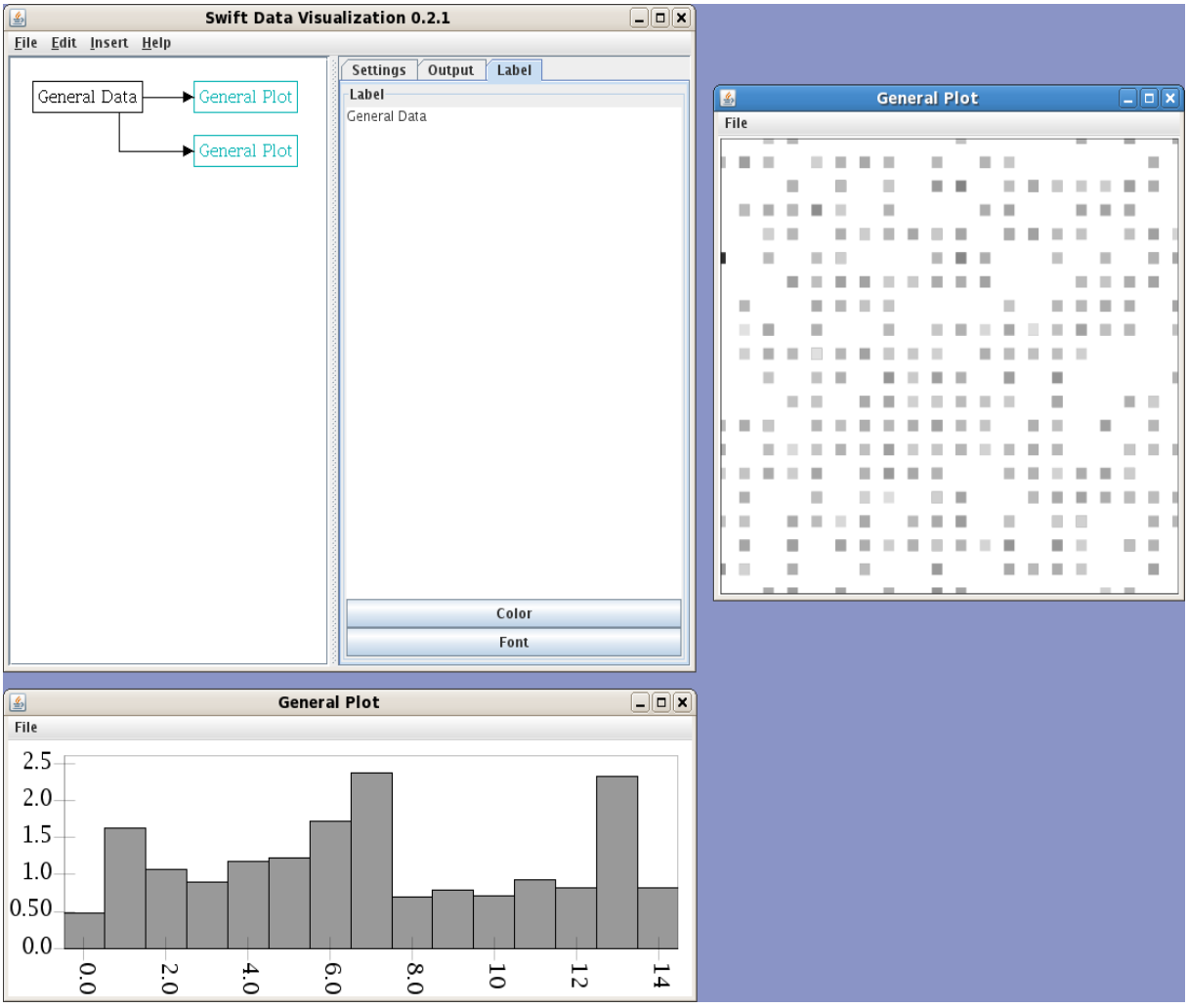


Figure 3.4: MASVis - The included tool for data visualization through the use of plots and filters.

settings window, housing both graphical data representations and textual data output in detached, escapable and restorable windows, is a necessary feature.

To keep almost every aspect of the simulation in separate, hide-able windows requires the use of a main menu, or a master controller, that cannot be closed. This controller handles not only the core, system-wide commands for MASVis but also the hide and restore functions for all other windows, while still remaining as small as possible and preserving screen space as can be seen in Figure 3.4.

Without a doubt, users of MASVis benefit greatly from a user interface system with separate, movable, hide-able windows. Such a system allows for a maximization of the valuable resource of screen real estate and also puts the systems focus on what currently matters to the user.

Overall the client interface has been designed with the SOA principles in mind, by making each element as modular as possible. This not only streamlines the overall work flow, but also makes the interface very flexible and effective.

Chapter 4

Load-Balancing in Social Simulations

There has been extensive research into effective load-balancing in highly distributed software systems. While a few of these have been adapted to multi-agent systems with mixed results, many of these traditional approaches and modified adaptations suffer from inherent difficulties.

4.1 Difficulties in Multi-agent Load Balancing

Agents are often embedded in an environment. Traditionally the actions of intelligent agents are executed within some environment to allow for the observation of their outcomes. The individual agents themselves become tied to the environment that surrounds them including other nearby agents. This environment is usually spatial in nature and can be described with simple geometric coordinates depending on the number of dimensions of the space. For efficiency reasons any load balancing policy should aim to store agents and their associated

environmental data on the same machine. Such a policy would encapsulate a majority of an agents computation within one machine rather than across several.

Multi-agent systems utilize communication extensively. Unlike the traditional distributed computing model where each machine carries out its workload independently until the simulation terminates, at which point results from each are aggregated to form a final result; the distributed multi-agent model requires intermediate communication between machines. In addition to messages sent between agents, machines may also communicate amongst each other about a shared data source. The cost communication should play a crucial role in determining the load of an individual machine; factoring in message length, network latency as well as the frequency of the number of messages sent/received.

Workloads are heterogeneous and potentially volatile. The amount work carried out by intelligent agents during each interval of a simulation is by no means homogenous nor stable. An individual agents workload is a function of the number of messages it has to process and/or send the degree of change in its environment and the complexity of its decision making. A load balancing policy should be able to rapidly and accurately adapt to fluctuations in both the workloads of individual machines and the entire system.

Multi-agent systems rely heavily on shared data. Achieving a complete decomposition of the environment agents share is impossible for many problem domains. Rather, a small portion of shared data must be maintained by each machine and synchronized at end of each interval of the simulation. A proper load balancing policy should manage this overhead by avoiding scheduling data synchronization during peak workload times or on machines that are already or close to reaching an overloaded state. While similar in many respects it is evident that distributed multi-agent systems diverge from the traditions of distributed and concurrent computing along a number of aspects. It is because of this divergence that an all inclusive solution towards solving the load balancing problem in distributed multi-agent

systems cannot be found directly in the traditional literature on distributed and concurrent systems. However, this literature does offer a set of advances made and lessons learned that as a field multi-agent systems can utilize to develop its own approach towards solving the load balancing problem.

We propose a load balancing policy for distributed multi-agent systems that accounts for their inherent limitations and simultaneously exploits their unique advantages. For efficiency and validity reasons entire spatial regions of agents comprise the workload assigned to each machine. Drastic changes in the distribution of workload are anticipated and proactively prepared for by predicting the direction and magnitude of agent movement within the simulation space. The resulting architecture ensures a well managed load using minimal system resources and the intuitive design allows developers to focus their energies on the internals of the agents and not related sub-problems.

In traditional dynamic load balancing schemes the two most important policies are the selection and location policy. The selection policy determines which task needs to be transferred to another machine either to alleviate load or share data with a running task on another machine. The location policy determines the ultimate destination of those tasks chosen earlier by the selection policy [6]. Our model combines these two policies into a single step that geometrically partitions the simulation space along a set of bounds and maps those partitions to the physical hardware available to the system.

4.2 Geometric Partitioning

When carrying out the selection and location policy there are two primary concerns:

- How many agents should be selected to be transferred relative to the change in the load desired?

- Is the ultimate destination of each agent in keeping with the integrity of the underlying model?

Relative to the computing power available on the market today individual agents use only a relatively minute fraction of the total resources of the machine they are executing on. Therefore, in vast majority of cases migrating an individual agent from machine to machine has little effect on the overall system's load and is extremely expensive in terms of added computation. To be truly effective, agents should be redistributed in single system wide batch operation.

To maintain the integrity of the underlying model, the ultimate destination of each agent should be relative to it's location prior to the redistribution process. Moreover, agents that were not located next to each other spatially beforehand should not be placed beside each other just for the sake of reduced computation. If an agent A expects to observe agent B to its immediate right this should remain true regardless of changes to the load distribution. As it is assumed here and is true in most cases[1] that two agents on separate machines have no mechanism for observing each other apart from communication over a network, then either agent A and B has to be moved to another machine together, or a remote reference over a network has to be created between the two.

To address these two concerns a very old and conceptually simple technique called geometric partitioning [12] is used. Using only the geometric coordinates of each agent, regions of space are assigned to individual processors so that the workload in each region matches the resources available to each processor. Partitions are computed using recursive bisection [4] which divides the simulation space into two partitions, each with half of the simulations workload. This cutting process is applied recursively until the expected workload of each

[1]Excluding shared memory multiprocessors.

partition matches the desired granularity. This method is extremely effective when geometric locality is important as in a multi-agent system where most of an individual agent's interaction occurs with its immediate neighbors. In addition, geometric methods because of their simplicity generally run faster and are easier to implement as well [39].

As Figure 4.1 indicates the bounds of each partition should change both through time as the simulation progresses and agents move throughout the space and as the resources of the physical hardware increase and decrease.

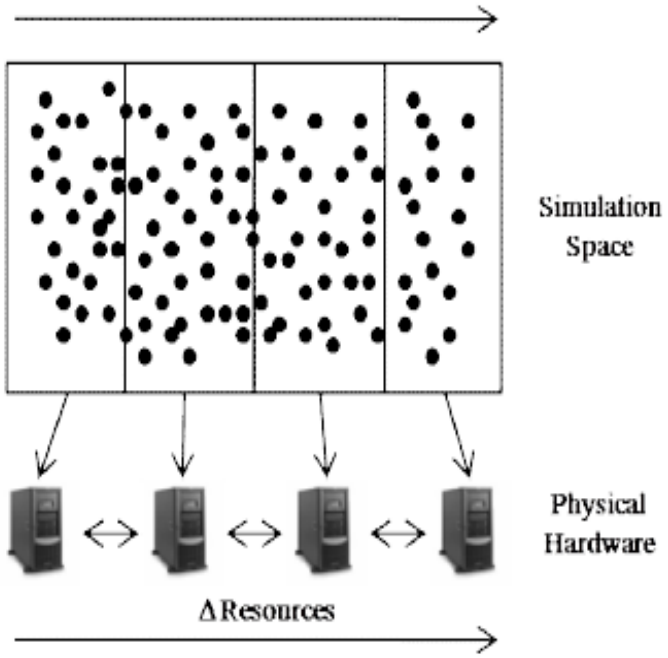


Figure 4.1: A representation of the distribution of partitions among slave servers.

This is implemented within the system as a KD-Tree, a data structure that recursively bisects a spatial region until the number of desired partitions is met, in this case the specified number of machines. Each internal node in the tree represents the location of one of the n-1 dimensional cuts to the n-dimensional simulation space and contains the children created

by that cut. These children are sub-regions if further cuts have been made to them. If no further cuts have been made or no more are possible these children exist at the level of the leaves of tree and are considered to be the partitions that are eventually mapped onto the physical hardware. New internal nodes are added to the tree until the desired number of granularity is achieved. By storing the partitions in this format, it becomes a $O(n * logn)$ operation to move a cut, which is a significant improvement over the $O(n^2)$ complexity of individual computation per agent that accompanies traditional migration techniques [25].

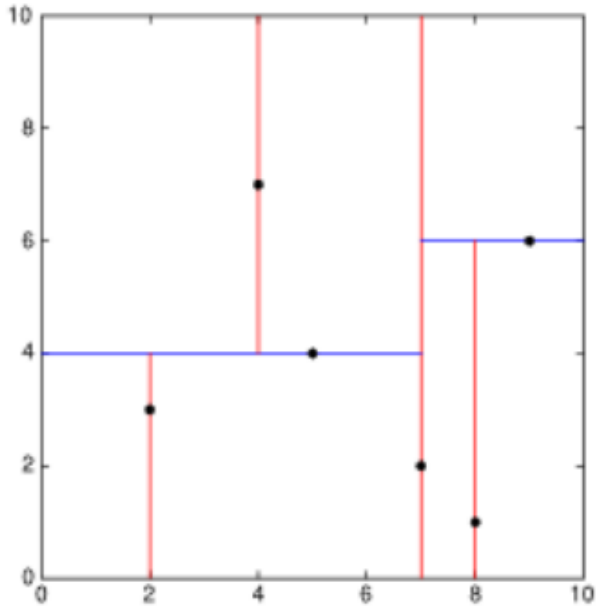


Figure 4.2: A graphical representation of a KD Tree.

The KD-Tree is also used to store the individual agent's locations on each of the slave servers. By utilizing this data structure to store the location, the system sees a significant performance improvement when looking for each agents local environment, which as was mentioned earlier plays an essential role in individual agent decision making in multi-agent

simulations.

The load balancing algorithm examines the workload of space that is partitioned on either side of an individual cut and then using the formula below creates a new partitioning of this space. The method for determining the degree by which the bounds is presented below.

$$X' = \frac{T_2 * X}{T_1 * (1 - X) + T_2 * X} \tag{4.1}$$

where X represents the current location of the cut and X' represents the new location for this particular cut as a percentage of the distance between the midpoint in the first sub-region and the midpoint in the second sub-region. T_1 and T_2 are the workload for those individual spaces. This equation determines the position of X' by dividing the percentage workload of one section by the percentage workload of the entire region, thus determing the position to reach the average workload. This workload can be measured in several ways; in our implementation we explore both the maximum time and average time it took all partitions to complete one time step as a measure of workload. This formula is applied recursively to all the cuts of the simulation space to balance out the workload among the partitions.

4.3 Proactive Behavior

Bounds can either be changed proactively in anticipation of changes in the system to come or passively in response to existing changes. While proactive behavior utilizes predicted models of future states of the system that have the possibility of being erroneous, it avoids a more serious problem of bounds thrashing that can occur when bounds are only passively changed. Bounds thrashing occurs when there are large rapid changes in the partition

bounds from time step to time step in response to highly dynamic agent movement and can lead to new bounds that are unsuitable for the current state, leaving the system unbalance and leading to additional work to remedy this error. Whereas when bounds are proactively changed, agent movement can be anticipated and bounds thrashing avoided by moving the bounds slowly and incrementally through time thus decreasing the total amount of resources consumed when load balancing. In proactive load balancing the direction of agent movement is predicted by using a linear regression of the agent's past location in both the x and y dimension on a per time step basis. By examining r units further along the line created, the agents position r steps from now can be predicted.

$$Y = \frac{n * \Sigma(xy) - \Sigma x \Sigma y}{n * \Sigma(x^2) - (\Sigma x)^2} X + \frac{\Sigma y - \frac{n*\Sigma(xy) - \Sigma x \Sigma y}{n*\Sigma(x^2) - (\Sigma x)^2} * \Sigma x}{n} \tag{4.2}$$

where x and y are the set of past positions of the agent and n is the number of those past positions. This knowledge can be collected for each agent within a partition and then averaged to find the aggregate movement of agents within that partition. The average position of all the agents is used as a reference point for this line. Using this we can then predict the average postion of the agents in any given number of steps. As agents approach the edge of a partition the load balancer should be triggered to adjust the bounds to compensate for the agent movement according to a set percentage representing the predicted number of agents that would have moved between the servers.

4.4 Pseudo-Migration

Relying on geometric partitioning solely as a means for load balancing would overlook one of the largest contributors to the load in any multi-agent system communication. Agent to agent communication is inherently non-spatial as agents may communicate with any other

agent no matter how large the spatial divide is between them.

A technique for reducing inter-server communication especially across high latency networks would be to create local ghost copies of agents that frequently receive messages from agents located on that node (see Figure 5.5 below). Messages sent to these ghosts are then batched by the node they reside on and sent in packets at the end of each time step to their corresponding host node. Throughout this process low volume periodic communication is allowed between agents residing on different nodes. Once inter-node communication between two agents reaches a critical level, however, pseudo-migration and ghosting is forced until the communication level between the two declines.

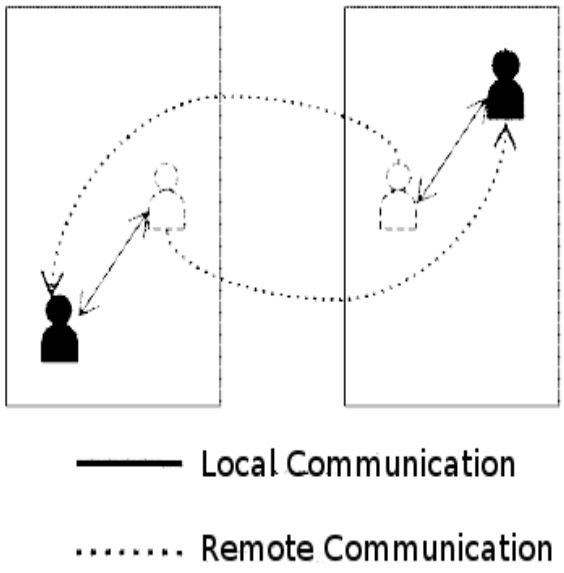


Figure 4.3: A representation of Psudo-Migration "Ghosting".

To achieve this each slave server maintains a data structure containing the frequencies of communication past and present between it's agent and any other agents within the simulation. If this frequencies ever cross an established threshold for a certain pair of

agents, the slave server requests a ghost agent from the master server.

```
for all messages
  increment frequency for sender/recepient pair
for all sender/recepient pairs
  if frequency >= threshold
    request ghost of recepient
```

Once the master server creates a ghost of the desired agent, it repeats the process it creates a ghost of the other agent on the other corresponding slave server. This effectively creates a pipeline between the two agents through the use of remote references. Using RMI these agents can effectively communicate directly over the network. Even with the ghosts in place, the slave server continues to monitor the communication level between these and all the other agents. If the communication level of these agents ever drops below the threshold the slave server removes the ghost from the server and requests that the other slave server to do the same.

```
for all sender/recepient pairs
  if frequency <= threshold and sender/recepient already ghosted
    remove ghost
    request reciprocal ghost's removal
```

This limitation on life span helps ensure that the technique is only used on agents that are contributing most to the communication load, thus avoiding overload the slave server's memory. Too much and too frequent ghosting can be just as detrimental as allowing intensive inter-server communication as local memory once reserved for real agents is quickly consumed by their copies.

One of the inherent aspects of multi-agent simulations is that the agents are mobile, and can easily move from one server to the other. If this happens the ghost is left without a corresponding agent. In this case, the next time the ghost is used to communicate, the server removes the ghost and then routes the message allong the traditional route through the master server. The ghost between the two agents is created again when the frequency between the new servers reaches the threshold.

```
recieve ghost message
if agent exists
  route message
else
  remove ghost
  request reciprocal ghost's removal
  route message to master server
```

This method effectively sidesteps sending messages up to a master node to be relayed back down to a receiving slave node by allowing agents to directly communicate with the slave node that contains the agent they wish to correspond with. This in effect, trades a small increase in local computation for a serious reduction in the communication load. In a cluster of workstations connected by high latency network, of the type commonly used by MAS researchers, this transfer of work from network down to the individual nodes can lead to drastic decreases in simulation running time.

4.5 Shared Data

In any distributed system it becomes necessary to provide some mechanism for sharing data amongst the multiple separate nodes that comprise the system. Though there exist multiple

approaches towards sharing data [24] we choose to represent data to share as an agent within the system. A copy of this shared data agent is distributed by the master node to each slave node at the beginning of each time step. During the execution of that step slave nodes not only utilize the data within this agent but contribute additional data to share in future time steps. This specialized agent differs from the standard agents that comprise the system in that they make no decisions and carry out no actions and most importantly each node's local copy is synchronized by the master node at the end of each time step to include data added and/or edited by the slave nodes. For agent-based researchers this approach towards managing shared data not only exploits existing features of their systems but is intuitive to understand and easy to implement.

Chapter 5

Experiments and Results

To effectively test MECA's capability as a large-scale multi-agent system for social simulation a variety of different test must be run. This includes a test of the ease of use, a test of its correctnest using a standard social test domain, and finally its ability to scale must be examined by testing it's load balancing capabilities. These load balancing tests include testing the geometric, psuedo-migration, and proactive aspects.

5.1 Ease of Use

A crucial goal of MECA was to provide social researchers an intuitive and easy way of creating, running, and interpreting their simulations. Along with the user interface, that was discussed in Section 3.3, we simplified the process of creating fully functional simulations to three easy steps:

- Create the agent's decision making process
- Create the builder to setup the environment and number of agents on the slave servers

- Create a menu to allow for modifying parameters during runtime.

The first step for the end user to do is to formulize the agent's decison making process. The system facilitates this by taking care of the rest of the mundane functions of the agent that the researcher is not interested in. To guide the user in formulizing this process we provided an agent template that seggregates the different functionality that the agent has, including percieving, communication, and acting.

After the agent has been created and formulized, the next step is to create the builder. Based on a template, this class allows the social researcher the flexibility of deciding exactly how they will set up the slave servers, including the bounds of the environment and the number of agents. Though this step complicates the process a bit, it serves a crucial role of allowing for additional flexibility.

Finally the end user is able to create a user interface for their simulation to allow for manipulation of the parameters while the system is running. Why this is not required for the simulation to run, it does allow for a much more streamlined work flow, when the social researchers are tweaking and exploring the intricacies of their simulations. For example, the user could create a UI that was able to change the number of agents, number of resources, bounds, and the range of sight for the agents. This significantly increaed the overall work flow, as it no longer is neccesary to return to the code to hard code the parameters over and over again.

MECA provides an easy three step process of taking a simulation from concept to implementation, thus allowing for a quicker and less threatening system for the social researchers to use. This process, along with the documentation on the system, allows for even completely unfamiliar users to pick up the system

5.2 Correctness

The most important step in testing any multi-agent system is showing it's correctness. The system must be able to handle processing the agents, messages, and servers in the intended and proper way. To have the system produces valid results, it must be proven to act according to its design. To test this we used two seperate test domains and compared the results the system provided against those that were expected.

5.2.1 Sugarscape

The sugarscape domain was chosen to test the system's correctness because it covered the necessary functionality and it's results are also very well documented. This allowed us to create the simulation according to the specified approach, as well as allowed us to easily and quickly vailidate the system's results.

The sugarscape was first proposed by Thomas Schelling as a way of modeling what he thought was an inherent pattern within segregation [36]. The sugarscape domain was later greatly expanded by Joshua Epstein and Robert Axtell [14]. We used their extended rules for our simulation. Sugarscape is a simple simulation consisting of agents and sugars. Two concentrations of sugar are placed in opposite corners. The sugar continues to grow until it reaches a set capacity set in the environment. The agents have a range of vision that allows them to see sugar. The agents also spend one unit of sugar a day, but they can gather more sugar than they can consume, if their sugar level ever drops below a certain threshold the agent dies and is removed from the simulation. The agent decision making process is simple; the agents move towards the highest value collection of sugar they can see and then proceed to collect all of it. Since the agents are allowed to move around, the simulation shows interesting patterns that form from clumping.

For our simulation we used 100 agents, with two main concentrations of sugars. These agents used the above rules, and were distributed across 4 machines with the bound of 200 by 200. The sugar growth rate was set so that each source of sugar would instantly regenerate. The results of this simulation can be seen in Figure 5.1 below.

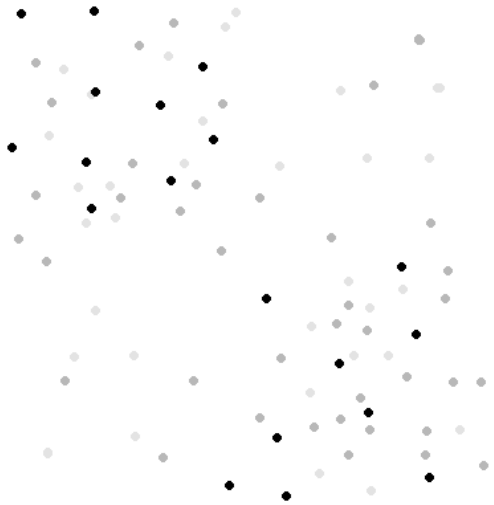


Figure 5.1: Scatterplot of a simple sugarscape simulation.

5.1 clearly shows clumping of agents, black dots, in the two areas where the concentration of sugar was highest. The agents level of sugar is shown by the color of the dot, white meaning no sugar and black implying an abundance of sugar. In fact the agents inhabiting the areas where no sugar was growing either died out or moved into the more prosperous regions. This result accurately reflects the original result that Epstein and Axtell originally produced using this simulation. This correlation between results helps prove the correctness of the system, and helps support the system's ability to correctly run simulations.

5.2.2 Stock Market

While the Sugarscape domain is useful for showing the systems correctness, it does not reflect the type of simulation the system will most likely be used for. To test the systems capability of handling the more complex decision making and additional communication involved with social simulations we turned to another test domain; the stock market. Based off the New York Stock Exchange this simulation provides a complex environment that is much more along the lines of social reseach than the Sugar Scape domain.

We modeled 20,000 agents who would select from among 30 unique stock indices for a time frame of 25 rounds. Every agent began the simulation with an initial 10,000 dollars cash, and no limitations were set on the amount of stock they could purchase each round as long as they had cash available to make a desired purchase. Stock prices changed each round based on traditional microeconomic supply and demand curves that accounted for the volume of buying and selling that occurred in the previous round. The more shares of a stock that were purchased, indicative of a higher demand for that stock and a dwindling supply, the higher the price was driven up and vice versa. Agents bought and sold stock only to the market and did not engage in inter-agent purchases, sales or trades for simplicity purposes. The agents were allowed to communicate and influence each others decisions, thus simulating the social influences found in the actual system.

This experiment was run using the Cognitive Agent for Social Environments (CASE) architecture, an architecture that models the human individual and social decision making process [29]. The architecture combines an intuitive and delibaritive individual decision making process with the social influence involved in neighborhoods, groups, and networks. This combinations helps model the impurities of realistic human decision making. For this reason, CASE is a good representative of the type of architectures and decision making

processes that will be used in most social simulations, and is thus a good benchmark test for MECA.

Using these settings we were able to run the simulation distributed over 4 machines and were able to produce results that intuitively reflected what was expected from the simulation. The results in themselves, though, help show MECA's capability to handle this type of simulation, effectively dealing with all of the intricacies involved with social interactions.

5.3 Load Balancing

To test for the system's effectiveness and robustness, we had to strain the load balancing method and the system as a whole to see if it would be able to handle the scaling required for large scale simulations. To accomplish this we tested the three seperate aspects of the load balancing strategy; geometric partitioning, proactive balancing, and pseudo-migration.

Each node in the cluster had the following characteristics:

Processor	Intel Pentium 4 512 cache
Memory	1 GB
Diskspace	80 GB
Network Bandwidth	Gigabit Ethernet

5.3.1 Geometric Partitioning

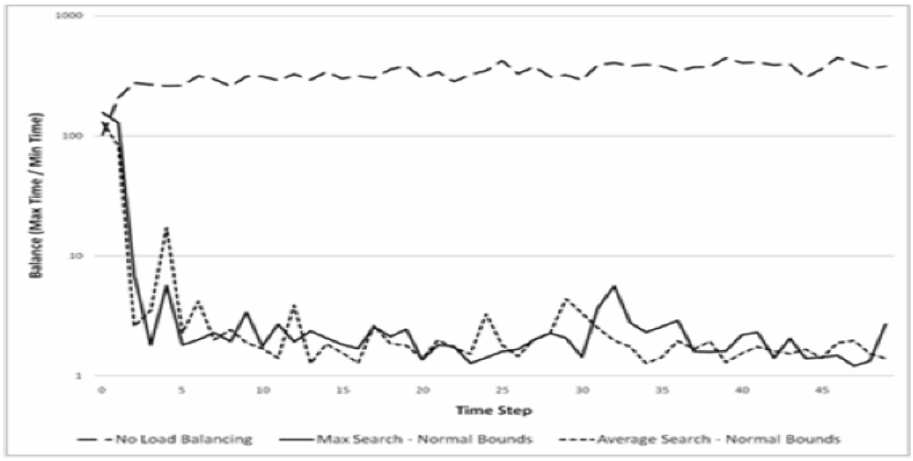
For our first experiment we used non moving agents who computed the average location of all the agents around them every step. This provided us with a group of computationally intensive agents, which helped us strain the system and force the load balancing strategy to be used. The workload was measured using the wall clock time for each step on each

node, thus encompassing the entirety of the load the individual agents created as well as any overhead created by other processes running concurrently with the simulation. This allowed us to modify the location of the bounds by simply recursively running through the tree, thus achieving O($logn$) performance for the load balancing, where n is equal to the number of nodes to balance. Since many of the upper level cuts actually created two sub-regions that contained multiple partitions within them, we tested our load balancing scheme using both the maximum time to complete a step for all the partitions in the sub-region as well as average time to complete a step as an indicator of the work load placed on each particular sub-region. We also attempted multiple variations of how far we would modify the bounds, both by a fractional amount of the intended move, in this case of our experiments 50%, by a fixed incremental move of 5 units and the standard modification provided by Formula 4.1.

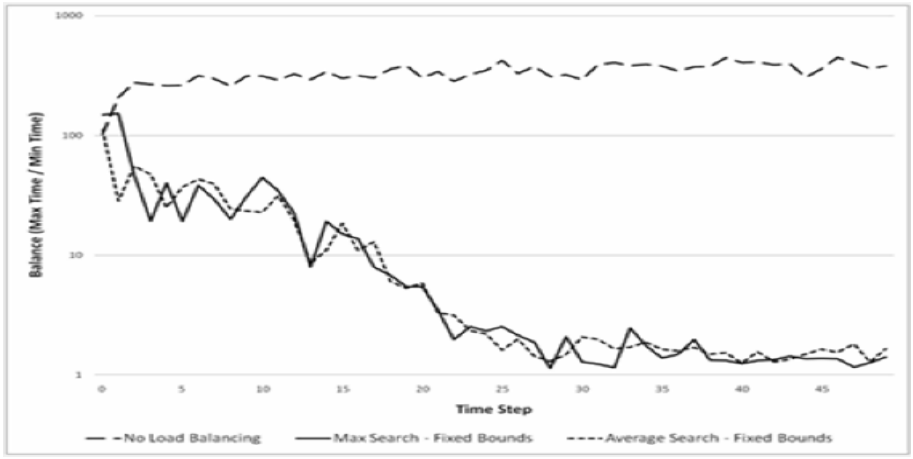
We ran this simulation with 40,000 agents for 50 time steps distributed over 4 seperate machines. Using this simulation, we were able to effectively test the benefits of all 6 combinations of time preferences and partition movement amounts, as described above. We measured balance by dividing the maximum server time by the minimum time. Therefor the closer to 1 the more balanced the system is.

As you can see from the Figure 5.2, the load balancing technique showed a 300% improvement over the performance of the simulation without load balancing. Figure 5.2 also shows that the load balancing in general was able to find a balance in n number of steps, where n is equivalent to the number of partitions. This linear relationship between the scale of the simulation and the time needed to achieve effective load balancing, shows that this technique is scalable to even large simulations than tested here.

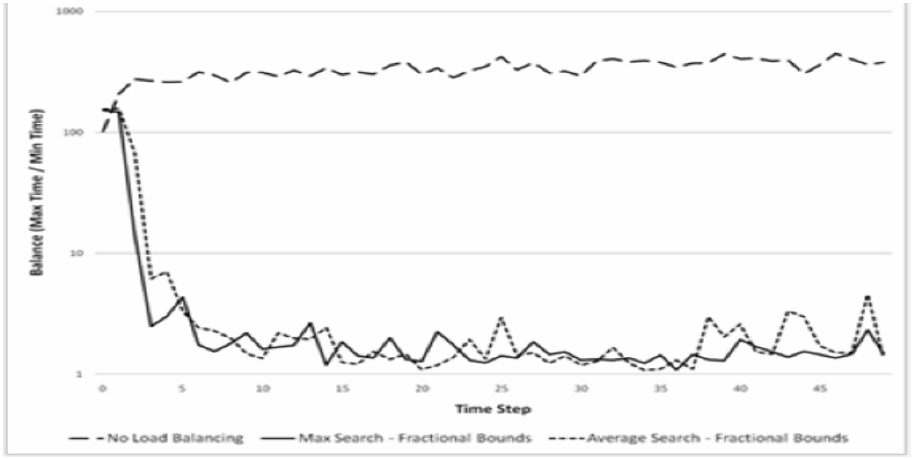
The second experiment introduces a dynamic aspect to the simulation. Using the same agents as in the previous experiment we allowed the agents to move as a group along the bounds of the simulation. This was done to test the robustness and the effectiveness of our



(a) Normal Bounds



(b) Fixed Bounds



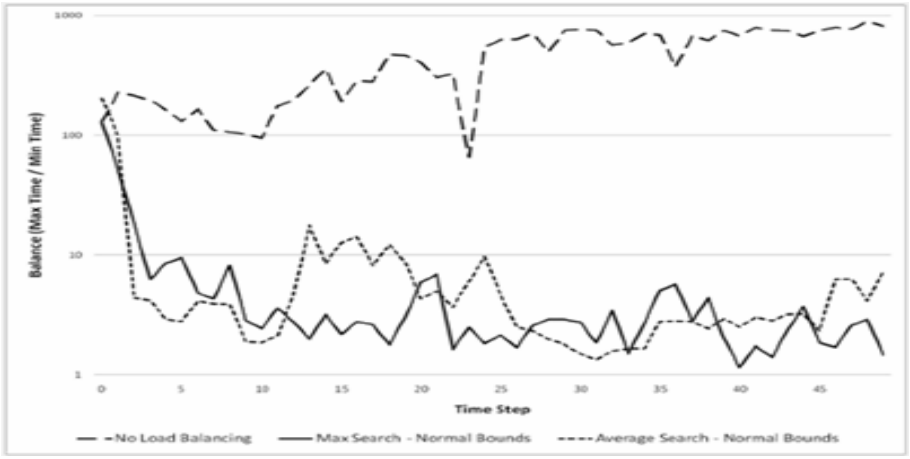
(c) Fractional Bounds

Figure 5.2: Max vs. Average Search for Bounds when Load Balancing one Concentrated Group of Agents

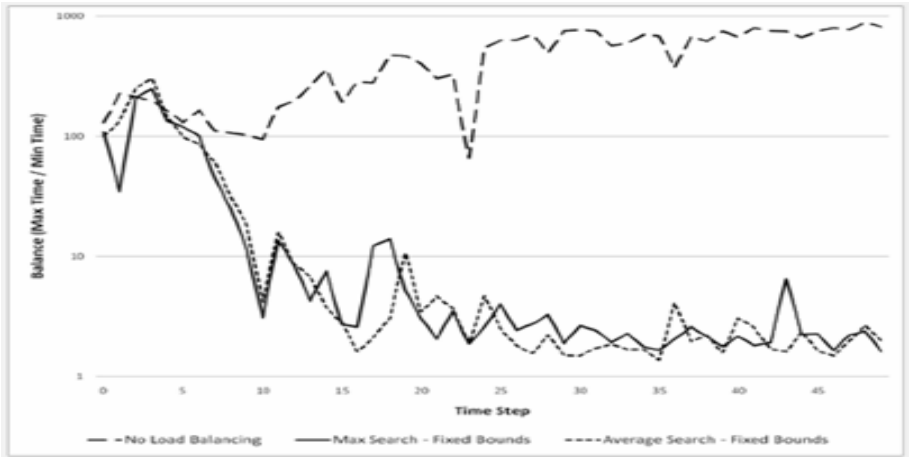
load balancing scheme under dynamic and uncertain conditions. We used the same settings for the simulations as before. Figure 5.3 illustrates that the improvement in balance (max time / min time) remains significant when compared to a simulation that employs no load balancing schemes. An optimum balance in the load is still reached rather quickly, here in $c * n$ steps, where c represents a constant and n equals the number of partitions.

In both Figure 5.2 and Figure 5.3 there is a noted difference between modifying the bounds normally by way of formula 4.2 (Figure 5.2(a) and 5.3(a)), by a fixed amount (Figure 5.2(b) and 5.3(b)) or by a fractional amount (Figure 5.2(c) and 5.3(c)) of the result given by 4.2. Modifying the bounds appeared to have very little effect on the overall rate at which the system converged to an optimum balance. Figure 5.2(c) illustrates that in a static simulation where agent movement is minimal moving the bounds of the partitions fractionally leads to a high degree of stability in the overall balance of the system. In the case where agents exhibited a high degree of movement, Figure 5.3, modifying the bounds both normally (Figure 5.3(a)) and by fractional amounts (Figure 5.3(c)) actually lead to reduced stability. Whereas Figure 5.3(b), which modified the bounds on a fixed basis, showed greater stability but at the cost of an unacceptably slow convergence to an optimum balance. We considering the results of Figure 5.2 and Figure 5.3 it is important to note that reaching an optimal balance quickly is more desirable than overall stability in the balance in reducing the running time of a simulation. The changes in stability of the balance over the course of the simulation are insignificant when compared to the severity of change from a complete imbalance initially to an optimal balance within the early portions of the simulation

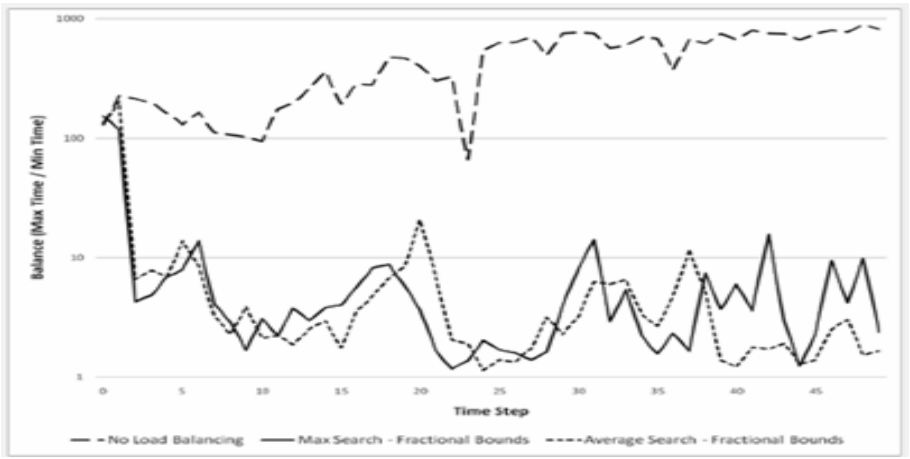
In both experiments the results showed a clear and distinct improvement over the system without load balancing. This improvement helps MECA's ability to scale to handle larger simulations consisting of large number of agents.



(a) Normal Bounds



(b) Fixed Bounds



(c) Fractional Bounds

Figure 5.3: Max vs. Average Search for Bounds when Load Balancing a Moving Concentrated Group of Agents

5.3.2 Proactive

Geometric partitioning plays a crucial role in the load balancing strategy, but it can be significantly improved by the inclusion of proactive elements. These proactive elements allows the system to gradual change the bounds to balance the system instead of over reacting when the system is already unbalanced. This is done by modifying the geometric partitioning by the predicted load based of an average linear regression.

The experiment uses the same agents as the second geometric partitioning experiment. In this experiment the agents move around the bounds of the simulation while doing relatively computationally extensive calculation. Since, a proactive approach is most beneficial in a dynamic and changing workload, these agents are perfect for testing our algorithms capabilities.

The simulation was ran using 40,000 agents for a total of 100 timesteps. The simulation was distributed across 4 machines. The proactive algorithm was set to predict 20 steps into the future, and used 10% of the number of agents to predict the overall future work load.

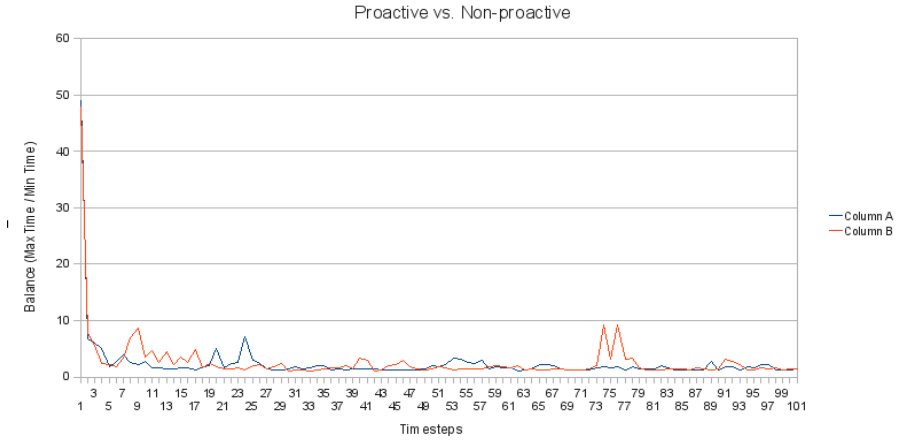


Figure 5.4: Proactive Approach (Column A) vs Non-Proactive Approach (Column B) when Load Balancing a Moving Concentration of Agents.

Figure 5.4 clearly shows the stabilizing effect of the proactive approach on the system. The proactive approach, column A, had far fewer peaks while attempting to balance a dynamically shifting work load. While the over all balance improvement is minimal, though still existant, the main purpose for the proactive approach was to allow for more gradual changes in the partition, thus avoiding the problem of thrashing. It is important to note that maintaining a constant balance is more beneficial. The computation involved with shifting the large number of agents involved in the large peaks seen in Figure 5.4 is far more detrimental, than the little benefit that the Non-proactive approach may have gained from the few times it was more balanced. In this light the effects the proactive approach has on the overall stability of the system are significant.

5.3.3 Pseudo-migration

The final aspect of the load balancing strategy must be tested is the pseudo-migration approach to balancing the communication load. Though, seperate from the other two aspects, the pseudo-migration algorithm plays a crucial role in minimizing the overall communication load. Since multi-agent systems and social simulations in particular stress the communication aspect, this part of the load balancing strategys is of upmost importance.

For this experiment we used agents with heterogeneous communication patterns. Specifically they communicated with 5% - 50% of the agents on the other server every timestep. Other than this the agents did not move nor make any other decisions, thus focusing the experiment onto only testing the psuedo-migration approach.

This experiment was ran using 200 agents for a total of 100 timesteps. The simulation was distributed only over 2 servers, in an attempt to test the specific relationship of creating ghost on the servers of agent pairs. Each agents communication levels dynamically changed over time, thus forcing the system to both created and destroy ghost copies. The threshold

was set at 1 message between the specified agents per time step.

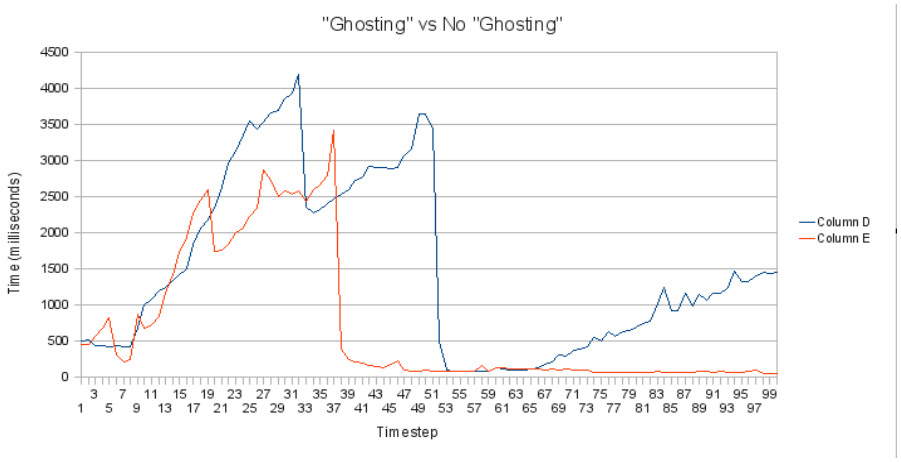


Figure 5.5: Ghosting (Column E) vs No Ghosting (Column D) when Load Balancing a large group of communicating agents.

Figure 5.5 clearly shows the improvement ghosting has on the over all communication workload, as seen by the comparison of column e, ghosting, vs column d, non-ghosting. The use of pseudo-migration had an obvious effect of lowering the over all wall clock time significantly as well as stabalizing it compared to the non-ghosting approach. The initial large spike in both lines was most likely due to an overall increase in communication within the system. The delay for the pseudo-migration approach to fully stabilize is most likely due to the extended time it took to get a significant number of ghost to decrease the communication load. The speed of stabilization could probably be improved by decreasing the threshold, but at the cost of extended use of the memory. Pseudo-migration though showing little improvement in the beginning of the simulation, it was either as good as or much better than the non-ghosting approach. Overall Psuedo-migration significantly reduces the communication load is an essential part of the load balancing strategy.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

The work in this thesis is the first step in developing a robust and flexible multi-agent system that could be used for a variety of social research. This system has been shown to be not only a valid system for running social simulations, but also robust and effective enough to handle large scalable systems.

One of the defecits of current multi-agent systems is that they are not developed to facilitate social research. Instead of streamlining the research process, they complicate it by adding unneeded features, or leaving out important tools. MECA is designed to be as approachable as possible for social researchers, by using a highly modular interface with data visualization tools built in. The system underneath is also developed to better handle the social and spatial aspects of the simulation. The system was also shown to be valid, as well as intuitive to use, both crucial to it's use in social research. These aspects, along with the encapsulated data tier, provide a robust system that is easily adaptable for the specific needs of the end user, and that facilitates the research process from beginning to end.

As multi-agent systems are increasingly used to identify and analyze the properties of complex systems, both real and artificial, the need to draw on more computing power is ever pressing. Multi-agent researchers have only begun to explore how to exploit the capabilities of networked machines to power their systems. Our work represents an integral first step in crafting a truly agent-based solution towards balancing the load in a multi-agent system. Exploiting the unique attributes of multi-agent systems such as the concept of geometric space and inter-agent communication our load balancing scheme attempts to proactively maintain a well-balanced multi-agent system. The load balancing strategy was shown to be both effecient and robust. The geometric partitioning quickly balanced the system, while the proactive aspect helped stabalize the overall balance. The benefits from this combined with the pseudo-migrations reduced communication load has helped the overall system become much more scalable, and in turn much more suitable for social simulation. While seemingly inconsequential on smaller scales the benefits of an effective load balancing scheme drastically reduce the running time of large-scale simulations. Simulations of this scale are becoming ever more common as multi-agent researchers apply their technology to new and more complex domains, and MECA is designed to handle their ever increasing complexity.

6.2 Future Work

The future work on MECA is multi-faceted. This includes expanding on the current interface and integrating it into a web based engine for access to the system from anywhere. This integration would also allow for multiple users to keep track of the simulations they are running, as well as provide more extensive integration into Google Map and GIS. The proactive aspects of the load balancing is also being further developed to find better hueris-

tics for predicting the overall trends of the simulation. Along with this the data tier will be expanded to allow for easier searching and filtering of data. Finally, a manual/tutorial is planned for the system, to allow future users to easily use it and hopefully expand it further.

Bibliography

[1] Yvonne Balzer. Improve your SOA project plans. *IBM*.

[2] Michael Bell. chapter Introduction to Service-Oriented Modeling, page pp. 3. Wiley and Sons, 2008.

[3] Fabio Luigi Bellifemine, Giovanni Caire, and Dominic Greenwood. *Developing Multi-Agent Systems with JADE*. Wiley and Sons, 2007.

[4] M. J. Berger and S. H. Bokhar. A partitioning strategy for nonuniform problems on multiprocessor. *IEEE TRANSACTIONS ON Computers*, 36(5):pp570–580, 1987.

[5] F. Buschmann, C. J Kel, R. Meunier, H. Rohnert, and M. Stahl. *A Pattern-Oriented Software Architecture — A System of Patterns*. Chichester UK, 1996.

[6] Ka-Po Chow and Yu-Kwong Kwok. On load balancing for distributed multiagent computing. *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS*, 13(8):pp.787–801, 2002.

[7] Phillip Coleman, Michael Pellon, and Jason Leezer. A multi-agent simulation for social agents. In *Proceedings of the National Conference of Undergraduate Research*, 2008.

[8] Cougaar. `cougaar.org/`.

[9] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Journal of Parallel and Distributed Computing*, 7:pp.279–301, 1989.

[10] S. Das and D. Grecu. Cogent: Cognitive agent to amplify human perception and cognition. In *Proceedings of AAMAS*, pages pp443–450, 2000.

[11] Paul Davidsson, Stefan J. Johansson, and Mikael Svahnberg. Characterization and evaluation of multi-agent system architectural styles. In *Proceedings of the Fourth International Workshop on Software Engineering for Large-Scale Multi Agent Systems (SELMA'05)*, pages 497–503, 2005.

[12] Karen D. Devine, Eric G. Boman, Robert T. Heaphy, Bruce A. Hendrickson, James D. Teresco, Jamal Faik, Joseph E. Flaherty, and Luis G. Gervasio. New challenges in dynamic load balancing. *Applied Numerical Mathematics*, 52(2):pp.133–152, 2005.

[13] Dias labarotory. `http://cs.trinity.edu/~yzhang/research/dias.html`.

[14] Joshua M. Epstein and Robert Axtell. *Growing Artificial Societies.* The Brookings Institute, 1951.

[15] Thomas Erl. *Service-oriented Architecture: Concepts, Technology, and Design.* Prentice Hall, 2005.

[16] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1977.

[17] D. Grosu and A. T. Chronopoulos. Algorithmic mechanism design for load balancing in distributed systems. *IEEE Trans. on Systems, Man and Cybernetics*, 34:pp.77–84, 2004.

[18] K. Hwang and Z. W. Xu. *Scalable Parallel Computing:¡Technology, Architecture*. McGraw-Hill, 1998.

[19] Jade. `jade.tilab.com`.

[20] M. Kafeel and I. Ahmad. Optimal task assignment in heterogeneous distributed computing systems. *IEEE Concurrency*, 6:pp.42–52, 1998.

[21] Jean-Daniel Kant and Samuel Thiriot. Modeling one human decision maker with a multi-agent system: the codage approach. In *Proceedings of the 2006 Autonomous Agents and Multi-Agent Systems Workshop (AAMAS'06)*, pages 50–57, 2006.

[22] Rajiv Khosla, Nikhil Ichalkaranje, and Lakhmi Jain. *Design of Intelligent Multi-Agent Systems: Human-Centredness, Architectures, Learning and Adaptation*. Springer, 2005.

[23] P. Krueger and M. Livny. A comparison of preemptive and nonpreemptive load distributing. In *Proceedings of the IEEE International Conferance for Distributed Computing Systems*. Springer, 1988.

[24] S. Kumar and P. R. Cohen. Towards a fault-tolerant multi-agent system architecture. In *Proceedings of the International Conference for Autonomous Agents and Multi-Agent Systems*, pages pp. 459–466, 2004.

[25] Tyng-Yeu Liang, Ce-Kuen Shieh, and Jun-Qi Li. Selecting threads for workload migration in software distributed shared memory systems. *Parallel Computing*, 28(6):893–913, March 2002.

[26] Mason. `cs.gmu.edu/~eclab/projects/mason`.

[27] T.G. Mattson, B.A. Sanders, and B. L. Massingill. A pattern language for parallel programming. *Addison Wesley Software Patterns Series*, 2004.

[28] S. Moss and P. Davidsson. Multi-agent-based simulation:second international workshop. In *MABS 2000*. Springer, 2001.

[29] Michael Pellon, Phillip Coleman, and Jason Leezer. Simulating social agents with multi-agent systems. In *Proceedings of the National Conference of Undergraduate Research*, 2008.

[30] D. Perry and A. Wolf. Foundations for the study of software architectures. *ACM SIGSOFT Software Engineering*, 17:pp.40–52, 1992.

[31] J. R. Pilkington and S. B. Baden. Partitioning with spacefilling curves. *CSE Technical Report CS94349*, 1994.

[32] O. F. Rana and K. Stout. What is scalability in multi-agent systems? In *Proceeding of the International Conference for Autonomous Agents and Multi-Agent Systems*, pages pp.56–63, 2004.

[33] Repast. `repast.sourceforge.net`.

[34] R. K. Sawyer. *Social Emergence: Societies as Complex Systems*. Cambrige University Press, 2005.

[35] A. Schaerf, M. Shoham, and M. Tennenholtz. Adaptive load balancing: A study in multi-agent learning. *Journal of Artificial Intelligence Research*, 2:pp.475–500, 1995.

[36] Thomas C. Schelling. Models of segregation. *The American Economic Review*, 59(2):488–493, May 1969.

[37] T. Schlegel, P. Braun, and R. Kowalczyk. Towards autonomous mobile agents with emergent migration behavior. In *Proceeding of the International Conference for Autonomous Agents and Multi-Agent Systems*, 2006.

[38] M. Shaw and D. Garlan. *Software Architecture — Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[39] H. D. Simon. Partitioning of unstructured problems for parallel processing. In *Proceedings of the Conference on Parallel Methods on Large Scale Structural Analysis and Physics Applications*, 1991.

[40] Swarm. `swarm.org`.