The Multi-Agent Distributed Goal Satisfaction System

G. Mitchell Saba and Eugene Santos, Jr.
Intelligent Distributed Information System Lab
Department of Computer Science and Engineering
University of Connecticut
{saba, eugene}@cse.uconn.edu

Abstract

Multi-Agent Distributed Goal Satisfaction, MADGS, is a JAVA-based mobile-agent system under development to facilitate distributed mission planning and execution in complex dynamic environments with a focus on distributed goal satisfaction. We present here an overview of the MADGS system. We discuss the unique aspects of the project including the theoretic design decisions that have led to its current state. The issues explored here include robust and reliable communication protocols, agent design, and a system architecture that facilitates both agent and server autonomy.

Keywords: mobile agents, agent servers, multi-agent systems, mission planning and execution, agent architecture, agent communications

Introduction

Multi-Agent Distributed Goal Satisfaction, MADGS, is a JAVA-based mobile-agent system under development to facilitate distributed mission planning and execution in complex dynamic environments with a focus on distributed goal satisfaction. The MADGS system represents the union of five separate components, Agent-Server (named Carolina), mobile-agents, Distributed Goal Satisfaction (DGS), AgentTool, and Prodigy. Only Carolina and mobile-agents will be primary foci of this paper, with a brief mention to the remaining three components.

In this paper, we present an overview of the MADGS system. We discuss the unique aspects of the project including the theoretic design decisions that have led to its current state. The issues explored here include robust and reliable communication protocols, agent design, and a system architecture that facilitates both agent and agent server autonomy. We outline the dominant features and implementation of the MADGS architecture with discussions regarding the basic components that embody an agent-based system. Our implementation follows standard Object-Oriented and Software Engineering techniques.

The target real-world operation environment for the MADGS system is a network topology that consists of intermittent nodes and uncertain network connections that exist in a large-scale, multi-platform dynamic network. The resulting design developed for this environment addresses the communications issues faced when handling massive numbers of mobile-agents in such a topology. Our development process required the consideration of bandwidth capacities (minimal broadcasts if any), mobile-agent collaboration issues, and server awareness of available resources. In designing a system capable of handling an unknown but unrestricted number of communication and agent migrations over the proposed topology we made an in-depth examination of both agent and server responsibilities. From this examination we developed a premise that there exists a marriage between the functionality of the Carolina agent-server and the agents themselves despite their autonomy. For this reason the MADGS architecture is built around this marriage, maintaining autonomy for both without depreciating the security or performance of the system. The marriage is one built of necessity. In order to minimize agent size some functionality was better placed in the server and offered as a service to the agents.

As such, MADGS is an object-oriented system for the deployment of mobile-agents to facilitate large-scale planning and execution operations in domains such as manufacturing, search and rescue, and military operations. One of the common threads across these domains is the need for distributed goal satisfaction that can work cooperatively with legacy planning systems yet autonomously handle changes in constraints. The ability to autonomously handle changes in the constraints of a plan can mean the success or failure of any distributed operational mission/goal. The need to re-plan or backtrack due to constraint changes in any plan can mean a substantial resource loss; be it lost capital or life the expense is real. Our approach seeks to mitigate a significant amount of this loss by preemptively expecting failure, defining alternative constraint configurations, developing delivery arrangements and in the event of a failure offering an instant solution to the user. This distributed goal satisfaction (DGS) process is a primary background activity of the MADGS system. In the forefront the system is providing an environment for general operation, sub-plan and sub-task execution and user interface. It is this aspect of the MADGS system that facilitates the DGS process. The MADGS system represents every resource (including personnel) by at least one agent. However, this representation allows the MADGS system to facilitate the DGS process by maintaining a more complete view of the current state of the ‘world’. This world-view is constantly shifting in any operation especially large-scale operations. For this reason alone communications becomes a crucial aspect of our system.

In the following sections, we discuss specific aspects of the MADGS system. We begin with an examination of the unique requirements of our domain focusing on agent communications requirements and mobile agents. We then present an overview of
the MADGS architecture and our agent/server environment – Carolina. Next, we briefly discuss the distributed goal satisfaction component of MADGS. Finally, we present our conclusions.

Agent Communications

For MADGS, the critical element for agent communications lies in the necessity of not having a singular point of failure in the system. This obviously means that we cannot afford to use centralized directories or look-up tables. Due to the network constraints we also cannot leave communications up to individual agents since this will effect the size of mobile agents and thereby bandwidth consumption. Another communications issue facing development of the MADGS system was location of agents and resources. Without centralized directories, look-up tables or the ability to farm communication concerns out to agent resources our research led to the quick determination that no available system answered the communication issues, resource tools or adaptability our project called for. For this reason we embarked on the design and implementation of the Carolina agent server and mobile agent system. Unfortunately no generic agent system with sufficient basic communications and resource management abilities could be effectively utilized. Systems identified [1] made different assumptions and focused on a specific problem without offering some base functionality that all agent systems could build off of. It is hoped that one by-product of our work is to define the base functionality needed by all agent systems. This is not to say that all agent systems should or will be based off of our work given all systems do not work off general models [2].

Our first idea was the use of an Ants model [3] where agents leave markers as they migrate through the network. Any agent crossing that path is in turn able to establish contact for collaboration. The Ants system would also insure that all subordinates (group-members) would be able to communicate since they generally originate from the same source. The difficulty with this approach was the growth in the agent's payload as the number of agents it collaborated and communicated with grew. Another problem was the inability to guarantee that communications would be delivered to target agents within some reasonable timeframe, if at all. This stimulated our examination of responsibilities and realization that communications would be better served if Carolina handled all communications. The communication protocols we devised attempts to insure several things:

1. The size of an agent will not increase significantly with increased interactions;
2. Each node will have a consistent view of the 'world';
3. The network load posed by communications can be minimized compared with alternate communication methods;
4. There is no central point of failure in the system; and,
5. All communications can be routed within a reasonable time frame.

These assurances can be made despite the volatile and intermittent nature of our network environment. The first assurance follows from the need for agents to only have an agent's unique assigned name to maintain a communication link with that agent. The second assurance is possible because our system provides for information sharing between nodes that guarantees all nodes will know the exact location of all mobile resources (agents are resources) assuming no non-server resource movement for a time-period not to exceed some constant value \( \lambda \). The third assurance is possible because our communication protocol negates the need for broadcast except for system-wide alerts that are hypothesized to be rare in any system. Since our protocol does not utilize a central look-up table or centrally located directories there can be no single point of failure for communications thus the forth assurance. The last assurance stems from the second assurance, since all nodes are guaranteed to know the location of all mobile-agents it is therefore possible to route ALL communications within a reasonable time frame.

Mobile Agents and MADGS

Mobile-agents are programs representing network users and resources that have the ability to autonomously migrate throughout the network performing tasks and computations on the user's behalf. Unlike classical distributed programming solutions; mobile-agent systems allow the autonomous migration of source code and state information to remote network resources for point of resource execution thereby reducing network traffic. This ability of mobile-agent systems allows for increased asynchrony in client-server interactions. These two abilities are the main advantages of mobile-agent systems [4]. Agent based development of the MADGS system facilitates the implementation of our communication protocols thereby allowing new approaches to old problems such as communication routing without constant collaboration and mobile intermittent network nodes. The communication protocols we developed above provide the ability for a mobile-agent to autonomously locate its extraction point despite the intermittent mobile (laptop) nature of the node.

The communication protocols and the division of responsibilities developed a clear picture of the marriage that exists between our Carolina agent server/environment and the agents operating therein. It is a marriage built out of necessity. The required MADGS system functionality is better served by implementing some 'standard' agent functions as services available from Carolina. This modification of base agent functionality allows us to gain the benefits outlined above. In addition this separation of responsibilities provides a clearer security model for both Carolina and mobile and static agents.

For MADGS, Carolina provides mobile-agents access to system and network resources. These resources can include databases, CPU time, and other agents. Carolina’s primary function is to serve the requests of mobile-agents while monitoring system and agent operations for suspicious actions. MADGS mobile-agents are the workhorses of the system providing the functionality the system users require. Mobile-agents are injected into the system through the AgentTool [5] component responsible for agent creation. The AgentTool component is not a standard component.
The foundation of the MADGS system is the agent server named Carolina. Carolina has a three-tier architecture with several components that are described in this section. There are four main functions of Carolina:

1. Provide an agent execution environment;
2. Insure system integrity through role-based security techniques;
3. Allow access to system resources where appropriate; and,
4. Provide communication services that improve the overall system performance.

Finally, since the MADGS system has been designed for use in a large-scale dynamic network architecture with massive numbers of mobile agents carrying out communications, point-to-point communications were not an option. The protocol we developed earlier extends existing work [8]. Agents are not responsible for maintaining an address book of any kind. If communication is needed with a specific agent the agent only needs that agent’s unique identification number which is provided when the agent is created or from the server after a request for dependent-collaboration. Location of and routing of messages to agents in the network is performed as a joint effort between the servers and a communications agent. The server logs all agents entering or passing through the server in route to another node in the network. Carolina logs the agent’s name, identification number, class, location or destination and time stamp. Communication agents use a random algorithm to canvas the network moving from server to server collecting the server’s agent directory and comparing it to their ‘view’ of the network. This agent then modifies (or cleans) the server directory and comparing it to their ‘view’ of the network. This agent then modifies (or cleans) the server directory and comparing it to their ‘view’ of the network. This agent then modifies (or cleans) the server directory and comparing it to their ‘view’ of the network. This agent then modifies (or cleans) the server directory and comparing it to their ‘view’ of the network. This agent then modifies (or cleans) the server directory and comparing it to their ‘view’ of the network. This agent then modifies (or cleans) the server directory and comparing it to their ‘view’ of the network.
The decision to not allow agents to directly communicate departs strikingly from the standard agent-based system protocols. The reason we take this unique stance is the limitations that occur when you allow an unlimited number of agents to freely migrate and communicate over a constantly changing network topology. Under such conditions the network traffic increases significantly as mobile-agents attempt to maintain the current location of mobile-agents they are collaborating with. Given our network environment, bandwidth usage must be kept to a minimum, therefore management of this cannot be left to the individual agents. It becomes an issue of control and performance. Let us begin to examine the Carolina architecture (See Figure 2).

Carolina receives agents through its AgentServer Port. The AgentManager controls this port. The AgentManager receives incoming agents, checks their intended destination (IP), if it is local then the AgentManager registers the agent in the AgentDirectory, deserializes it and passes it to the ExecutionContainer where the agent is provided with a thread for execution. If however, the agent's intended IP is not local, the AgentManager simply reroutes the serialized agent through the AgentClient Port after registering the transient agent in the AgentDirectory. The AgentDirectory is one of the key components of Carolina. It maintains data on all agents that the resident Carolina server has seen. Information stored in the AgentDirectory includes typical information including the agent's unique name assigned at creation, the agent's class, source IP, current IP, and goal. Additionally, AgentDirectory stores a pointer to the messages stored in the Message Directory for the agent.

**Architecture of Agents in Carolina**

There are few super-agents in the 'real-world' that are not accountable to some authority. In the agent world this authority is typically the server. Carolina serves as an overseer of the system resources and the actions against them. The model used to construct the relationships between agents within the MADGS system is based on a military hierarchy. This hierarchy model allows the tight control over agent interactions and a clean object model. All MADGS agents report to some entity. This entity can be a system user or another agent. Agents are all sub-classes of a base agent class thereby ensuring common core functionality for all agents. Agents have a rank and group that determines the type of communications they can issue. Agents interacting with agents of higher rank within the same group must comply to 'order' communications they receive. Agents from different groups or of equal rank request assistance or collaboration. This hierarchy lends itself to the assignment of security or priority settings for each agent dependent on a given task. A low ranking task agent that receives priority orders from a superior inherits some of the ranking agent's authority (rank) while completing the task. This type of communicative inheritance is critical if a system is going to allow for mission-critical tasks as ours does. Figure 3 illustrates the overall interactions between system components, agents, and the command flow.

The designated Command Agent Group consists of high-ranking agents or users. Using a Group of Command Agents protects against a single point of failure. The provided model easily fits the needs of a commercial, manufacturing, search and rescue or military model since all share a common theme, the accomplishment of a goal by following a predetermined plan of operations (i.e.: business plan, production plan, disbursement plan or battle plan). In relation to inter-agent communications Carolina follows the allowed agent communication model previously described. Each agent in the Command Agent Group actively generates sub-plans for assignment and execution. In parallel, these same Command Agents are processing DGS alternative sub-plan resource configurations (models) for their respective primary sub-plan. The DGS models typically are not shared unless there is a resource failure that threatens the primary sub-plan. Future plans for the DGS models includes pooling these models to learn which models offer more success and are more frequently selected by system users.
Presently there are three agents fully designed in the MADGS system, the Command, Communication, and The Wanderer agents. The architecture of these three agents are illustrated in Figures 4-6 and described here. The Command agent (Figure 4) is a problem-solving agent that incorporates DGS into its architecture. The DGS cache is used to store alternate configurations in rank order, to provide for the failure of alternate resources. Data on the relevant alternate resources, surrounding a particular plan, are stored in the agent's data storage area. This resource data is acquired through sharing, collaboration or provided directly by the user when necessary. The agent's Control module coordinates the execution, problem solving, communications, and DGS modules. State information and an execution log is also maintained. The execution log is used to justify exhibited behavior in the event a problem occurs during execution.

The Wanderer (Figure 5) and Communication (Figure 6) agents share some functionality. Both agents serve to facilitate robust communications in our network by migrating, comparing and sharing information regarding other servers or agents respectively. There are two main differences though. The Communication agent is a single task agent that never dies. It's function is to randomly move through the network gathering each nodes AgentDirectory, comparing the current agent location and status information with its own and merging the two from the most accurate information. Once completed, the node's AgentDirectory is updated and the Communication migrates to the next node. This process never stops. The Wanderer however, has a limited life span with two tasks. Unlike the Communication agent that has no 'home' node in the network, The Wanderer agent is tied to one Carolina server. For each Carolina server there can only be ONE Wanderer agent. This singularity is necessary to insure that a rogue Wanderer agent cannot be injected into the network. The Wanderer's first task, once instantiated, is to create a Server list for its host Carolina server. This is accomplished by a polling process, the Wanderer attempts to bind to other potential Carolina server's AgentServer. If successful the Wanderer migrates to that Carolina server and announces the presence of its host Carolina in the network. The Wanderer then collects foreign host information from the local Carolina server list and migrates to the next available Carolina server (not previously visited) ultimately returning to its host Carolina server. This process is performed once when the Carolina server is instantiated. Once complete the Wanderer lies dormant (sleeps) until the host Carolina server initiates a shutdown event. The second task of The Wanderer is to announce the death of its host Carolina server. When a Carolina server is shutdown The Wanderer detects this event and (prior to the server shutdown) migrates to another server where it corrects the available server log of that local server. The Wanderer then visits all servers that where listed in its host Carolina server prior to shutdown.

These three agents demonstrate the base functionality of the MADGS system. Each of these agents were created without using AgentTool however excluding The Wanderer all future agents will be instantiated from AgentTool and be sent to a Carolina server for execution. The modules present in these agents will be selectable from AgentTool for custom point and click creation of new agents to meet new tasks.

**Distributed Goal Satisfaction**

Even though some agent-based mission planning and execution systems have been developed [9] they do not fully use the power of agent programming. Our approach is to use the strengths of legacy systems in conjunction with the strengths of agent programming coupled with our own approaches to communications (as outlined above) and resource location and allocation. Most large-scale operations create a plan offline by formulating a problem or suggested outcome and then determining an optimal plan for the realization of this global goal. While an agent system could use such an operable legacy
systems in existence optimized for this purpose. This is why Prodigy is used to create a Master Plan for the MADGS system. This plan is then provided as input to the MADGS system. The MADGS system then uses command agents to decomposes the plan (if necessary) into sub-plans which are further decomposed into tasks by sub-command agents (Command agents with a lower rank) that assign the tasks to subordinate task agents. This process is not dissimilar to those present in existing agent systems. Our approach is only unique in how we plan for alternative courses of action in the event a plan fails during execution due to changes in the present state of the ‘world’.

Resolving to one course of action (or plan) in a real-time system poses great difficulty due to the volatile nature of the constraints and the conditions a plan is based on. A change in a constraint or condition of a sub-plan could lead to its total or partial failure that in turn can lead to a rippling effect, thereby negating the validity of the initial plan. To overcome these points of failure, a robust and flexible planning system is needed. The DGS agent module seeks to provide a surrounding technique to improve the robustness and flexibility of the overall planning and execution process. We accomplish this by acting on the resources required to accomplish a given goal, plan or task. A resource is any commodity that is necessary to facilitate the completion of a goal. (i.e.: Goal A requires resource X, quantity 3) The DGS agent module receives the local version of the plan and a list of the required constraints (primary resources). With this input a tree is constructed of the alternative resource configurations for each required primary resource. This data is then rated based on alternative resource availability to the local plan (taking into consideration other pending or current local plans). This information is then stored and the DGS agent module monitors the primary resource statistics. In the event that a primary resource fails or is exhausted the DGS agent module suggests alternative resource configurations to complete the current plan. If accepted the resources are set into action in place of the primary (failed) resources. During this process DGS collects and records data on the selections that users make to give weight to certain configurations and more importantly to learn new resource alternatives and configurations. This process when successful negates the need to replan or backtracking furthering the maintenance of a real-time system.

Conclusions

We have described MADGS, a JAVA-based system permitting mobile agent programs to execute, communicate, access resources and migrate. These activities occur in a network topology that consists of intermittent nodes and uncertain network connections that exist in a large-scale, multi-platform dynamic network. The unique features of the MADGS system include communication protocols that offer performance assurances and a new approach to distributed goal satisfaction that negates the need for traditional backtracking or replanning. We have also introduced, in the Carolina server, a functional architecture for generic agent systems with clear responsibilities for the server and agents that preserve role-based security and system integrity. In the MADGS system's Carolina server we provide a robust execution environment for distributed mission planning and execution with a base agent design that facilitates the completion of the provided goals.

In the future, we plan to include empirical data for the DGS agent module, continued integration of the Carolina server, AgentTool, and Prodigy. Another aspect of our remaining work is the collection of empirical data on the agent communication and location process with comparison to proven methods. Additionally, exploration into the construction of a DGS knowledge base from the distributed DGS modules will be made.

References


http://www.media.mit.edu/~nelson/research/routes/sigmobile.ps