A Critical Study of COUGAAR Agent Architecture

APPROVED BY:

____

Dr Lawrence Chung

A Critical Study of COUGAAR Agent Architecture

by

Tarun Belagodu

Sasikiran Kandula

Term Paper

THE UNIVERSITY OF TEXAS AT DALLAS

Abstract

Cognitive Agent Architecture (COUGAAR) is a Java-based architecture developed to build large-scale distributed agent systems. The architecture was developed as part of a research program funded by DARPA. Though distributed agent systems have been around for a considerable time and are known to be useful in building large systems, there have been concerns about their scalability, security and survivability in the event of attacks or hardware failures. COUGAAR architecture is expected to address these issues and provide a robust, secure and scalable agent architecture that provides a framework for developing large scale applications.

In this paper we study the presented architecture and analyze how the architecture assures the above features. We will also point out some critical issues which we consider need to be addressed to make the architecture better.

TABLE OF CONTENTS

TABLE OF FIGURES

CHAPTER1

INTRODUCTION

In computer science, a software agent is defined as a piece of autonomous or semi-autonomous proactive and reactive, computer software. To be considered an agent, a software object must be a self-contained program that is capable of making independent decisions and taking actions to satisfy internal goals based upon its perceived environment. Commonly cited main attributes of agents include the following:

- Autonomy: the ability to act autonomously to some degree on behalf of users for example by monitoring events and changes within their environment.
- Pro-activity: the ability to pursue their own individual set goals as well as making decisions.
- Re-activity: the ability to react to and evaluate external events and consequently adapt their behavior and make appropriate decisions to carry out the tasks to help them achieve their goals.
- Communication and Co-operation: the ability to behave *socially*, to interact and communicate with other agents (in multiple agent systems (MAS)) i.e. exchange information, receive instructions and give responses and co-operate when it helps them fulfill their own goals.
- Negotiation: the ability to conduct organized conversations to achieve a degree of co-operation with other agents
- Learning: the ability to improve performance over time when interacting with the environment in which they are embedded.

1.1 Agent architectures

An agent architecture gives a high-level view of the subsystems that make up the agent system, their interactions and the flow of control and/or information among the subsystems. Cognitive agent architecture (COUGAAR) is an agent architecture developed for DARPA under the Advanced Logistics program (ALP), by a consortium of companies during the period 1996-2001. The architecture was developed by ALPINE, a consortium currently composed entirely of BBN Technologies and further developed till 2004 under the DARPA program UltraLog. Its main purpose was to develop techniques to capture and solve problems related to military logistics planning and execution. But COUGAAR essentially describes an approach to building software and can be used in various other domains involving large scale distributed applications. The essential features of COUGAAR that the developers tried to achieve are:

- *Robustness* – the loss of any hardware component or hardware substrate has to result in only minimal loss of functionality.
- *Security* - Maintain information integrity, communication security and to repel all co-coordinated attacks including DoS
- *Scalability* – If the application logic allows for a particular degree of scalability, the underlying COUGAAR architecture should allow that i.e. addition of more agents or hardware components to achieve more or better functionality.

In section 2 of this paper, we give a brief description of the architecture of COUGAAR. Section 3 lists out some of the issues which we feel need to be addressed to make the COUGAAR architecture better, in terms of robustness, security and scalability. The last section tries to conclude and show some possible research directions for the COUGAAR community.

CHAPTER 2

BASIC TERMINOLOGY

In this section we describe some of the terminology related to the COUGAAR architecture. Individual agents in the system are grouped or clustered to form larger subsystems like communities and societies. A *COUGAAR society* is a collection of communities and agents that collectively solve a particular problem or group of problems. The problems are typically related to planning and the plans change dynamically. A COUGAAR society may be made up of one or more communities.

A *COUGAAR community* is a notional concept and refers to a group of agents with some common functional purpose or organizational commonality. Each community in turn can have separate individual sub-communities; each community co-coordinating with others for achieving a particular task. An agent community is a notional concept in the sense that it provides some kind of notional interface describing what it does to the society – the services it provides by specifying the inputs it requires and the output it produces (It specifies *what* it can provide but not *how* it does so). For instance, in military logistic transportation can be modeled to be a community. But transportation in turn can be split further into sub-communities in the form of air-transportation, sea-transportation and ground-transportation.

A COUGAAR community contains one or more *nodes.* A node is a single JVM on which one or more agents are deployed and maintained. The grouping of agents into nodes is not domain-related but is done in order to maintain a more equitable distribution of resources among all agents. Agents belonging to different communities can reside on the same node. Agents on the same node share and compete for the resources – CPU time/bandwidth etc.

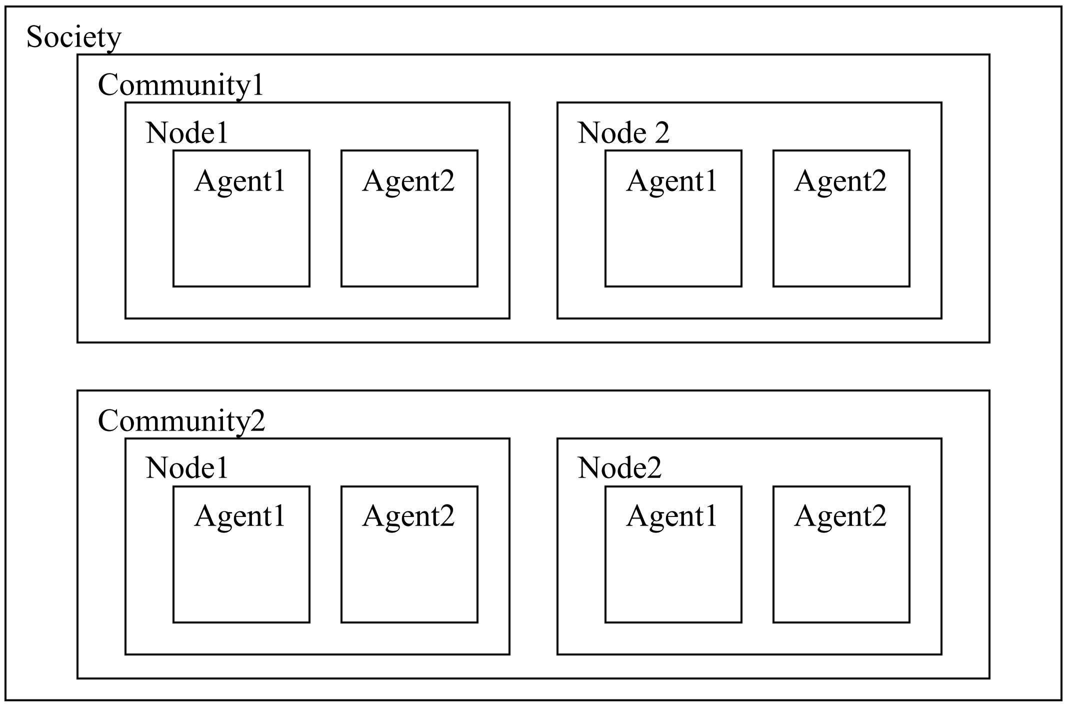

Figure 1: COUGAAR high level view

2.1 Agents in COUGAAR

An agent is the smallest functional unit in the COUGAAR architecture. An agent has two main components

- Blackboard
- Plugins

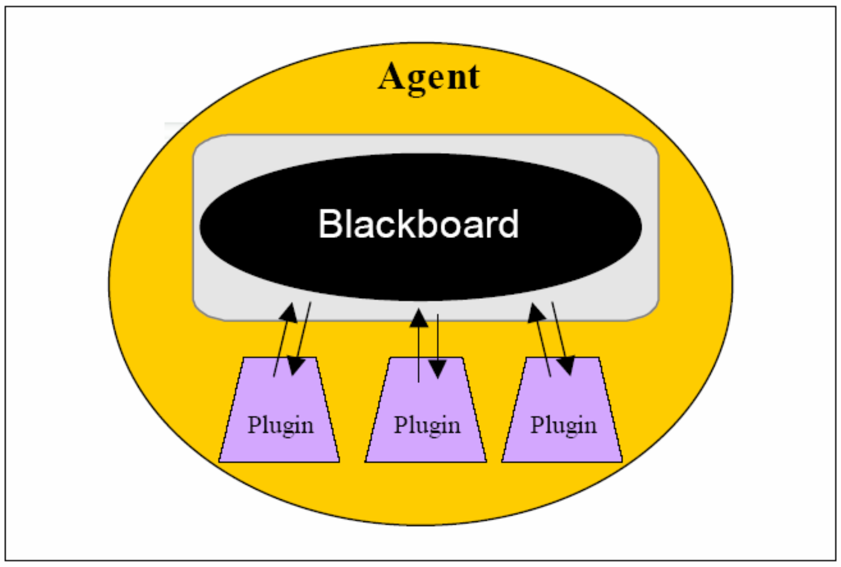

Figure 2. COUGAAR Agent internals

2.1.1. Plugin

Plugins are the software components that provide behavior and business logic to the agent's operations. Depending on the functionality that an agent wants to achieve, it can initialize a certain set of Plugins. Once initialized, a plugin is a conceptually different entity from the agent. Plugins are self-contained elements of software and are not dependent on other plugins. They can communicate with other plugins only through the agent's blackboard. They can publish results to the blackboards and react to events from the blackboard.

2.1.2 Blackboards

A blackboard is an agent-local memory store that supports subscribe/publish semantics. The components (plugins) of the agent can be assigned objects. The components can add/delete/update objects from the blackboard. At the same time, they can subscribe to add/change/remove notification for certain objects. The COUGAAR blackboards are local to an agent and this provides for scalability. If a global blackboard had been used it would have been a single point of failure (as in JMS).

Access to the blackboard is transaction-controlled. But this transaction management is only for the addition and removal of objects and not for any changes made at the sub-object level. This transaction management is done by proxy object entities called 'Subscribers'. Each plugin is associated with a s*ubscriber* which serves as an interface and manages all interaction between the plugin and the blackboard. This *subscriber* is also responsible for other plugin functionality like querying and publication of plan changes to the blackboard.

2.2 Agent Interactions

Though the blackboard allows the plugins within an agent to interact, the architecture should allow for the agents to interact with each other. For this the COUGAAR supports various features:

- Naming: The communication to a particular agent should essentially require that each agent is identified uniquely with a name. The naming service in an application based on COUGAAR architecture can be modeled as per the developer's preference. The developer can choose to name an agent so that the name reflects the functionality of the agent. Alternatively, the name of the agent can be generated randomly, for instance, with the nodes' network id concatenated by a very large randomly generated number. The architecture does not attach any specific significance to the name but only requires them to be unique. This is analogous to naming of files.

- White pages: is a distributed table that maps the agent names to network addresses. The use of a distributed table makes it more robust since we do not have a single point of failure and also scalable as the addition of new agents or agent organizations is not limited by the restriction on the size of the central table. Efforts are on to replace the white pages with a JNDI lookup service.

- Yellow Pages: is an attribute based service, something similar to that in a phone directory. Each agent registers itself depending on its' set of capabilities. An agent looking for a service from another agent can query the yellow pages depending on the capabilities it is looking for.

Inter-agent communication is necessary in querying and delegating tasks. For this COUGAAR provides two features, Relays and Attribute-based Addresses.

2.2.1 Relays:

These consist of two interfaces, *source and target*, that the blackboards objects can use. They ensure that an object in one agent can have manifestation on the object of another agent with which it wishes to communicate. Objects on the source blackboard implement the *Relay.source* interface and this ensures that data at the source can appear on the target. Similarly *Relay.target* interface should be implemented by the object on the target blackboard so that any response from the target can appear on the source. It is allowed for the same object to act as a source and target for different agents simultaneously, by implementing both the interfaces.

2.2.2 Attribute-based Addresses:

In some cases, the communication needs to be sent to a community but the source is not sure which particular agent in the target community is responsible for this kind of tasks. For instance, an agent might have certain critical sensor information, like breach of security at a particular plant, which should be conveyed to an agent community responsible for taking corresponding action, like raising an alarm. By using attribute-based addressing the source can look up which of the agents in the target community can handle this. Also in cases where there can be more than one targets, the source can multi-cast the information to all the target agents.

2.3 Inter-agent relationships

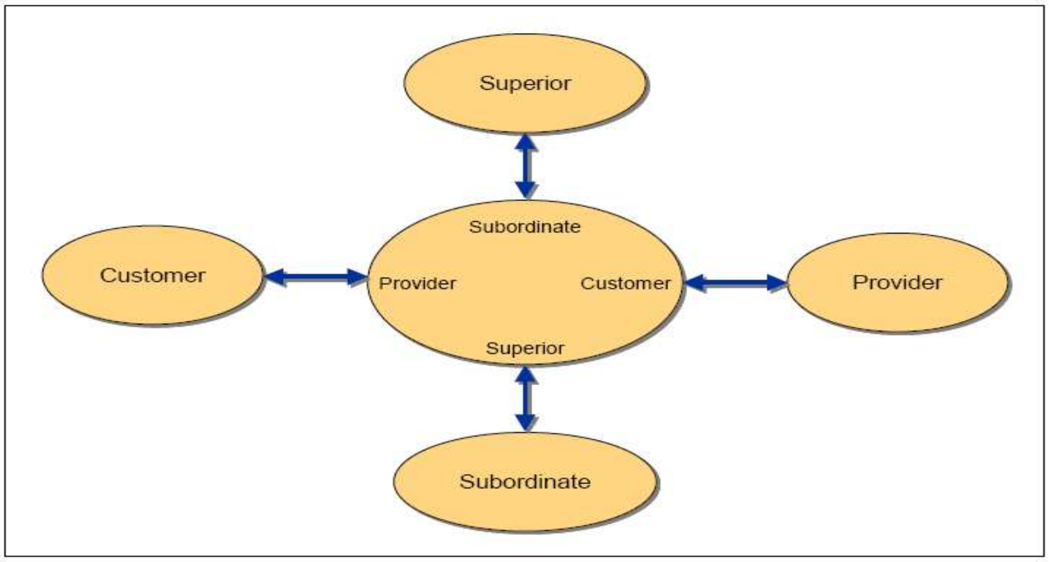

Figure 3. COUGAAR Inter-agent relationship

For each agent in Cougaar, roles can be defined depending on the services it can offer to other agents. Efficient communication between the agents is possible if the relationships between agents are clearly specified. An agent that is required to perform a particular operation can delegate the work to other agents that is related to. These inter-agent relationships can be of various kinds but the most important among them are:

- Superior/sub-ordinate relationship, in which there is a long-term relationship between the agents. The designated supervisor can ask its subordinate to perform a certain function. The sub-ordinate agent can make use of its resources and negotiate with other agents and work to achieve the goal. It keeps reporting to the supervisor agent at regular intervals.
- Customer/provider relationship, in contrast is a short-lived relation in which the customer requests for a service and the provider replies with the results of the service.

An agent on realizing that it does not have enough resources to reach a particular goal can request the services of agents it is related to. The agent on receiving the request can either choose to perform the job or delegate it to one of the agents that it is related to. This would create a chain-like structure that would transfer the jobs as well as the results of the job once it is completed.

CHAPTER 3

ISSUES

COUGAAR architecture was designed to provide a framework for developing large-scale distributed application which can ensure high security, scalability and robustness. There are specific features in the COUGAAR that have been designed for this purpose. The COUGAAR architecture indeed appears to be good in ensuring these. But our study of the architecture has raised some concerns which might not go well with the features that COUGAAR wants to achieve. Some of these concerns are described.

3.1 Robustness

To ensure robustness the Cougaar application should be able to survive an attack or loss of components with minimal loss of functionality. However, to achieve this objective each agent or cluster of agents reacts independently when it detects a potential loss of hardware components. Though this tends to locally optimize the performance in the event of an attack, the overall performance may be adversely affected. A much better alternative prescribed, would be to have a coordinated recovery which can ensure that the overall performance suffers minimal hindrance. This can be handled in two ways;

- When an agent or group of agents, identify a threat, they can query whether any other agents are facing the same threat. In case the attack is being faced by a group of agents, then these agents can negotiate with each other to determine the most appropriate action for each of the agents.

However, in some cases the agents may not have similar goals and the negotiation process should prioritize the goals, which may not be easy to achieve. Further, for the agents to negotiate there should be an underlying communication mechanism. But if the communication network is itself facing the attack, then the negotiation may induce extra traffic into the network and may not be a viable alternative.

- A separate agent can be assigned to take the necessary action to ensure survivability. A specialized agent, says a co-coordinator agent, can be assigned for each group of agents. This agent needs to be informed by any agent that perceives a threat and it is the responsibility of the coordinator agent to use other coordinator agents and arrive at a possible course of action.

However, having a single agent can cause scalability problems eventually. As such, both the coordinated recovery methodologies have some drawbacks and the independent recovery mechanism may result in poor performance. The architecture does not suggest any better mechanism to ensure robustness.

3.2 Security at agent-level

An agent in the COUGAAR architecture can belong to more than one COUGAAR community at the same time. If an agent belongs to more than one community then agents of one of the community can access the information of the other community through the agent. In the event of a security threat when agents of a particular community or node have been compromised, it is better if the damage is localized and other communities are not affected. The architecture does not clearly describe how this is going to be achieved.

3.3 Blackboard-related issues

The blackboards used in the agents are local only to the agent i.e. contents of the blackboard are visible only to the agent in which it resides. As discussed earlier other agents that are interested in the objects of an agent's blackboard can subscribe to the information. These blackboards are independent of each other and there is no existing control over the contents of the blackboard or the consistency between the elements of the blackboard. However, the blackboard is consistent in itself i.e. all the objects of the blackboard are consistent with each other.

The architecture specifies that any COUGAAR application should incorporate the necessary logic to maintain the consistency. We feel that achieving consistency would not be easy especially in application that requires a certain amount of synchronous behavior among the agents and it would have been better if the architecture had addressed this issue. For applications that require the agents to be asynchronous and more independent, this architecture can provide a good framework.

3.4 Constructing societies and communities:

The efficiency of a Cougaar community depends on how well the agents are organized into societies and communities. One significant requirement for a community is that it should provide single-coherent interface to other communities in the society. Though agents within the same community are more tightly coupled to each other, they do not share a same state and the blackboards in the agents are also independent. Due to this, providing consistency in a community is hard. One way around would be to make a single agent as a gateway to a set of agents within the community. Since all interaction with this group should be through the same agent, a coherent system is possible. However, this poses severe scalability problem. This singe agent which serves as an interface would be a bottleneck and the number of agents that can be added to the community would be limited by the capability of the agent to provide the necessary interactions.

Another important consideration is the distribution of the agents of a community. As discussed earlier, the architecture does not place any restriction on the agents that reside on the same agent node. These agents share the same set of hardware resources but can belong to different communities and provide different functionalities. However, the agents of the same community need to have greater communication and distributing them arbitrarily might mean additional traffic on the communication network connecting the nodes. An alternative proposed was to deploy all the agents of the same community on the same LAN. This would provide for better and faster communication between the agents.

However, this might cause some survivability problems. The localization of all the agents of a community can cause problems if there is any disruption in the locality of the community. For example, if the LAN on which the community is deployed is down due to some reason, the entire community might be unreachable from any other community of the society. Considering that each community depends on another to provide some service, the complete non-availability of an entire community can cause severe performance degradation and in some cases bring the entire agent society to a halt. Deploying redundant communities to provide the same functionality will be very expensive and might not be feasible for most applications. The architecture does not prescribe an alternative.

3.5 Agent Initialization and Node Agents

Cougaar provides two ways for creating or initializing new agents: static and dynamic.

- In static initialization, each agent node is provided with an XML file which contains information regarding the agents that have to be created and the plugins that have to be loaded to each agent. A node initializes an agent by granting it some resources and specifying the part of the XML file which is relevant to that agent. The agent then reads the XML file and loads the plugins it is meant to load.
- In dynamic initialization, each node can be run as a generic agent server i.e. it creates or remove agents from the node depending on the requests from other nodes.

Each node has exactly one node agent, which provides management functionalities at a node. This agent is a super-agent in the sense that it has control over all the objects at the node irrespective of which agent it belongs to. We feel that this poses a serious threat to the security aspect of the node. A malicious agent gaining control of the single node agent can gain complete access to all the agents and also control the hardware resources of the node. Further, the node agent is a single point of failure and in the event of failure of this agent; the nodes can no longer function.

3.6 White pages and Yellow Pages

White Pages are implemented as distributed tables and can provide the mapping from an agent name to the agent's physical network address. *Yellow Pages (YP)* provide a look-up service based on the services that each agent needs or provides. Yellow pages are more complex to implement as the queries for service can be quite demanding (for instance, an *and* relation describing the set of services). Implementing the YP as a distributed system would be very hard since that would imply updating a large number of tables whenever an agent starts to offer or stops offering a service. On the other hand, YP cannot be implemented as a global database due to scalability reasons.

One alternative would be to build a kind of hierarchical system of YPs. An agent looking for a service would look in the YP of the lowest hierarchical level. If the required service is not found then it can move upwards along the hierarchy. However, this would mean multiple queries for looking up a single service

3.7 Bandwidth

COUGAAR architecture is suitable to build large scale applications that have high-bandwidth availability. Due to the large number of inter-agent interactions that are required, an application that cannot provide a decent bandwidth would eventually have only a marginal performance.

3.8 Steep Learning Curve

As mentioned earlier, COUGAAR architecture is suitable for large scale applications. As a result of which, there is a steep learning curve associated with its use.

3.9 Asynchronous Communication

The architecture has been designed to make the agents independent and asynchronous. This can cause serious consistency and coherence problems. Providing synchronous communication is definitely an issue to consider.

CHAPTER 4

CONCLUSIONS

COUGAAR architecture is a robust, highly scalable, dependable agent architecture that has undergone years of development and evolution. Though, we consider COUGAAR to be one of the most complete and detailed agent architectures available today, there is still room for improvement. In most cases, the architecture introduces a single agent as a super-agent for coordinating the activities of a group of agents. This would severely effect the scalability and security requirements that COUGAAR aims to achieve. The architecture should be improved to eliminate such bottlenecks. Some of the cases to be considered have been pointed in the previous section.

Further, the architecture is suitable for large scale applications, especially, ones that can guarantee very high bandwidth. This limits the number of applications that can *afford* to provide such high bandwidth requirements. We strongly believe that a scaled down version of the architecture can be designed that can be suitable for small scale applications and thus provide greater flexibility.

One important advantage in COUGAAR is that it evolving continuously and has been able to draw the attention of a considerable number of research organizations and communities. It is currently the focal point of many research organizations and we believe its relevance to applications in various domains is bound to increase.

BIBLIOGRAPHY

[1] *Cougaar Architecture Document*, BBN Technologies, 2004.

[2] Christopher Matthews, *An Expose of Autonomous Agents in Command and Control Planning*, Command and Control Research and Technology Symposium, 2004.

[3] Aaron Helsinger, Michael Thome, Todd Wright, *Cougaar: A Scalable, Distributed Multi-Agent Architecture*, IEEE SMC, 2004.

[4] Dana Moore, Aaron Helsinger, David Wells, *Deconfliction in Ultra-large MAS: Issues and a Potential Architecture*, Proceedings of the first COUGAAR conference, 2004.