一個在多代理人環境下
去中心化的服務找尋機制

研究生: 杭仲瑋　　　　　　　　　　　　指導教授: 孫春在

國立交通大學
資訊科學系

摘要

多代理人系統提供了在分散式大型環境下複雜問題可行的解決辦法. 每一個在多代理人系統的代理人在只擁有有限的資源, 不完整的資訊, 和有限的能力的條件下, 透過代理人之間的互動來解決複雜問題. 因此, 如何有效率的找尋適合的互動代理人是個很重要的研究主題. 本文提出並說明了一個去中心化的服務找尋機制, 這個機制藉由在建立分散在各個代理人之間的服務目錄, 使得代理人可以利用關鍵字來找尋其他代理人所分享的資源, 能力或者資訊. 另外, 本文提出的這個去中心化的服務找尋機制解決了廣播式的查詢, 單點失敗, 以及效能瓶頸等三個在相關領域的常見問題.

A Decentralized Multi-agent Service Lookup Mechanism

Student: Chung-Wei Hang   Advisor: Chuen-Tsai Sun

Institute of Computer and Information Science
National Chiao Tung University

Abstract

Multi-agent systems (MAS) offer promising solutions to complex problems in distributed, open environments. Each MAS agent uses limited resources, incomplete knowledge, and limited capabilities to solve complex problems via agent interactions. For this reason, how to perform efficient searches for compatible agents is currently an important research topic. The author describes a proposed decentralized service lookup (DSL) mechanism that constructs service directories that are distributed among agents, who use keywords to find resources, capabilities, and knowledge held by other agents. The proposed DSL avoids three common problems in this area: broadcast queries, single point failures, and performance bottlenecks.

Acknowledgement

I am deeply indepted to my advisor, Dr. Chuen-Tasi Sun. All my life, I will remember my first research results from his considerate instruction. I am also very grateful to my dear firends, Hsun-Cheng Lin, Wen-Heng Chung, and Ji-Lung Hsieh, who make up a creative studying environment and really help me overcome many difficulties.

I have never said thanks to Father and Mother, but I would dedicate thesis to them. For Parents' love and encouragement, I can express only an inadequate acknowledgement of my appreciation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Multi-agent systems (MAS) offer promising solutions to complex problems in distributed, open environments. Each MAS agent uses limited resources, incomplete knowledge, and limited capabilities to solve complex problems via agent interactions. This makes efficient searches for compatible agents an important research topic in Information Science. In this thesis, I will give a detailed description of a decentralized service lookup (DSL) mechanism that prompts agents to use keywords for looking up services provided by other agents. The DSL serves as a means for finding compatible agents for purposes of receiving larger rewards and managing resources, capabilities, and knowledge more efficiently.

MASs consist of agents with limited capabilities and incomplete environmental knowledge; the agents solve problems by interacting with each other within loosely-coupled networks. Currently, computer science researchers tend to solve complex problems in distributed, open environments using systems (e.g., RoboCup Rescue [8]) and architectures (e.g., Open Agent Architecture [10]) that view agents as modular components. Individually, agents in a MAS do not have sufficient information for problem-solving.

Research on agent interaction for increasing rewards can be categorized as a) efficient searches for compatible agents, and b) means for multi-agent operations. In the first category, proposed methods include market-based mechanisms [14], middle agents [3], recommendation and request forwarding [5, 7], coalition formation [13, 2], and center agents [8]. Research in the second category is primarily focused on the communication languages (e.g., KQML [4]) used by agents for expressing their needs after they find compatible agents.

There are at least three important challenges to finding compatible agents. The first is referred to as *broadcast queries*—the means by which agents interact. Since messages are broadcast to all agents in a MAS, this method is considered inefficient because it increases both computing and communication overhead. In Matchmakers [17], RoboCup Rescue [8], and similar programs, middle agents must be able to communicate/interact with all other agents. If this task requires excessive amounts of computation and communication resources, it can lead to the second important challenge, referred to as *performance bottlenecks*. If middle agents fail, then all other MAS agents will not be able to find compatible agents—a situation known as *single point failure*, the third challenge.

The DSL described in this thesis avoids these challenges by means of careful information management—that is, by building decentralized service directories. The DSL also provides a keyword system for agents to use when searching for compatible agents. To address the issue of communication efficiency, I will use RoboCup Rescue without center agents as an experimental platform for verifying the proposed DSL's ability to avoid the three challenges.

The outline for the rest of this thesis is as follows: in Section 2 I will discuss research on efficient searches for compatible agents and the current literature on broadcast queries, single point failures, and performance bottlenecks. The proposed DSL mechanism is described in Section 3; the description includes an overview, specific information on the Chord protocol, and DSL operations. In Section 4 I will present the experimental design for testing the proposed DSL and discuss the results of the experiments. A conclusion is offered in Section 5.

Chapter 2

Related Work

2.1 Middle Agent

Middle agents have different names in different systems. In Open Agent Architecture they are known as *facilitators* [10], in InfoSleuth they are called *brokers* [12], and in RETSINA they are referred to as *matchmakers* [17]. Middle-agent MASs also have provider and requester agents. In a typical interaction, providers send advertisements describing their services and requesters send their service requests to middle agents. The middle agents then return matching advertisements to the requesters. Single point failures and performance bottlenecks occur when all agents in a system send their requests and/or post their advertisements to middle agents within a narrow time frame (A.1). The proposed DSL uses a keyword system to manage these interactions.

2.2 Market-based Mechanisms

Another approach makes use of market-based mechanisms to manage agent actions—for instance, bidding for services in auctions using contract net protocols [14]. However, it is still possible for large numbers of agent messages to cause performance bottlenecks, or for unexpected auction breakdowns to cause single point failures.

2.3 Forwarding Recommendations and Requests

In systems based on these two methods, agents send requests to other agents, who either respond with recommendations or forward them to other agents. For example, Iamnitchi and Foster [7] proposed a resource discovery system based on a grid. If A sends a request to B for resource x and B does not have the resource, B uses different request forwarding strategies with C, D, E, etc. until it locates an agent that has the resource. According to an experiment they conducted using different scenarios to test several request-forwarding strategies, the strategy of forwarding requests to agents who have the best track records of satisfying past requests is most successful in environments with unbalanced resource distributions. Forwarding requests to a randomly chosen agent was also identified as a satisfactory strategy, especially when resource distribution is balanced.

2.4 Coalition Formations

In systems based on this method, agents organize coalitions among groups according to agent interests [13, 2]. This allows for quick identification of other agents for interactions based on need. An example is Foner's [5] Yenta Lite system, in which agents use a recommendation system to find other agents who share common interests.

2.5 RoboCup Rescue

RoboCup Rescue [8] is a MAS used to simulate large-scale urban disasters. It utilizes three types of platoon agents and three corresponding types of center agents; these are described in detail in section 4.1. A platoon agent must use a corresponding center agent in order to interact with other types of platoon agents (C.2). If a center agent fails, its corresponding platoon agents will be blocked from interacting with other types of platoon agents, resulting in a single point failure. Performance bottleneck problems arise if all platoon agents of the same type move to interact with other agent types within a short time frame. As I will show in Section 4, the proposed DSL allows platoon agents of different types to communicate with each other without having to rely on center agents, thus avoiding both of these problems.

2.6 Multiple Middle Agents Architectures

Another strategy for avoiding performance bottlenecks and single point failures is to incorporate multiple middle agents into a MAS architecture. Three examples of this approach are Open Agent Architecture [10], RETSINA [18] and DECAF [6]. In RETSINA, after a matchmaker organizes a list containing the IDs of agents whose capabilities match incoming requests, it uses an *agent name server* (ANS) to find them. RETSINA uses a large number of these servers. An ANS can provide a multicast-based discovery service in order to respond to the needs of dynamic environments, and uses a flexible updating method to map agent names to their locations. The availability of more than one middle agent and ANS to serve the needs of individual agents allows this system to avoid single point failure and performance bottleneck problems.

The discovery service [9] provided by an ANS is built using a multicast-based protocol such as Gnutella (global discovery) or UPnP (local discovery). When an ANS performs a search, it sends a multicast message to an entire network in order to contact all other ANSs; this can result in a broadcast query problem (A.2). By avoiding the use of extra agents, the proposed DSL system is better able to respond to changing agent behaviors with much fewer messages being exchanged.

Chapter 3

Decentralized Service Lookup

3.1 Chord Protocol and Chord API

The Chord protocol addresses object location problems using a peer-to-peer network environment without centralized or hierarchical control organizations. Chord supports one simple operation, expressed as "given a key, Chord maps the key onto a node" [15]. Assuming n number of nodes, Chord protocol takes $O(\log(n))$ messages to look up the location of a key with $O(\log(n))$ nodes, with each node having $O(\log(n))$ amount of information stored in it. Chord protocol maintains information even when nodes join and leave the system; at a high level of probability, each event results in no more than $O(\log^2(n))$ messages. The Chord protocol uses load-balanced, decentralized, scalable, and flexible naming properties, all of which are common charac-

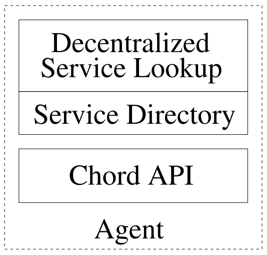


Figure 3.1: Agent architecture with DSL

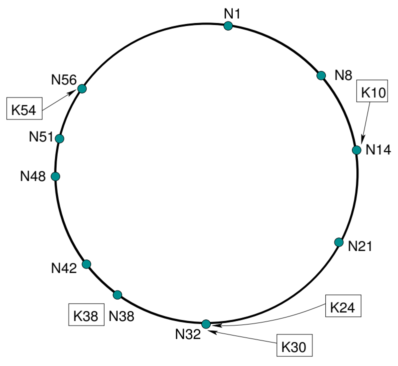


Figure 3.2: Chord ring

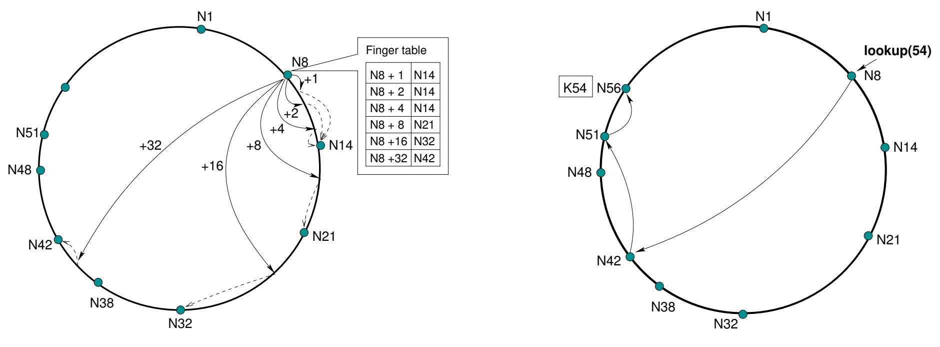
teristics of large-scale, dynamic, and open peer-to-peer environments. Other decentralized protocols that have been suggested for locating objects in peer-to-peer systems include Pastry, Tapestry, and Freenet. Chord is considered more simple and reliable [15].

Chord's key-mapping feature provides a method for MAS agents to build decentralized service directories in the absence of global information or centralized control. Chord defines the means for a) locating a key, b) finding new nodes to enter a system, and c) handling and recovering local node information if an agent fails unexpectedly.

The agent architecture in the proposed DSL MAS is shown in Figure 3.1. Using Chord API (Appendix B), agents build and maintain local information in the form of finger tables, successor lists, and service directories. They use DSL operations and local information to locate services and to find agents that provide them.

3.1.1 Consistent Hashing

Using a hash function (e.g., SHA), Chord assigns an ID to each node and assigns a key to each object; in both cases, the range is from 0 to $2^m - 1$.



(a) Finger table

(b) Key lookup with finger table

Figure 3.3: Chord protocol

Chord also organizes system nodes into Chord rings. X is said to be the successor of Y (either an object or node) if it is next to Y in clockwise order in a Chord ring. A example of a Chord ring containing 10 nodes and 5 keys with identifiers ranging from 0 to 63 is shown in Figure 3.2.

3.1.2 Scalable Key Location

Chord ring nodes maintain their successors' locations as part of their local information. The nodes also maintain finger tables that contain m entries. The i^{th} entry in the finger table of node X (called the i^{th} finger of X) stores the successor of $(x + 2^{i-1})$, where $1 \leq i \leq m$. The complete finger table of a node (named node 8) is presented in Figure 3.3(a). This scheme has two characteristics: a) each node only stores information on a small number of other nodes, and b) the information stored in one node is insufficient for finding a key's successor. In other words, finding a key's successor requires several steps involving several nodes.

In Figure 3.3(b), for instance, node 8 wants to look up an object with key 54. First, node 8 searches its finger table and learns that it must contact node 42. According to its local information, it knows that node 52 is node 42's successor. Then node 42 addresses node 56 by finding key 54's successor in its local information. Finally, key 54 is located as part of node 56's local information. This operation takes $O(\log(n))$ steps involving $O(\log(n))$ nodes. Detailed proofs and pseudo codes are presented in [15, 16].

3.1.3 Dynamic Operations and Failures

In dynamic and open network environments, Chord has two responsibilities: a) maintaining correct local information when nodes join, leave, or fail unexpectedly; and b) ensuring that finger tables and node successors are updated by periodically calling the *stabilize* and *fix_fingers* functions of Chord API. Furthermore, nodes maintain successor lists of size r in order to store their first r successors, thus ensuring that Chord will succeed even if r successors fail. However, if a node fails with probability p, the probability that r successors will fail at the same time is only p^r.

3.2 Agent Local Information

The proposed DSL provides a method for MAS agents to look up services using local information instead of a combination of global information and centralized control. As part of their local information, agents store a) Chord-required finger tables and successor lists and b) service directories (Fig. 3.4) that maintain service-location pairs using the format <*service keyword, agent's name*>. Service keywords can be defined as names, descriptions, or any other parameter as determined by the application.

Service Keyword	**Agent's Location**
"gcd"	"140.113.88.55"

Figure 3.4: Service directory

Table 3.1: DSL operations

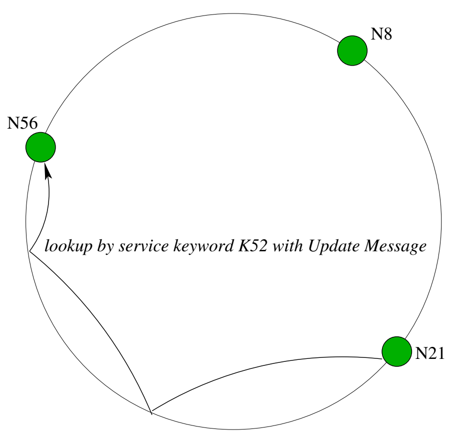
Operation	**Description**
$n.Join(n')$	agent n joins Chord ring which has agent n' already
$n.Update(keyword)$	agent n updates its service
$n.Lookup(keyword)$	agent n looks up service by keyword
$n.Recovery()$	agent n recovers local information periodically

3.3 Decentralized Service Lookup (DSL) Operations

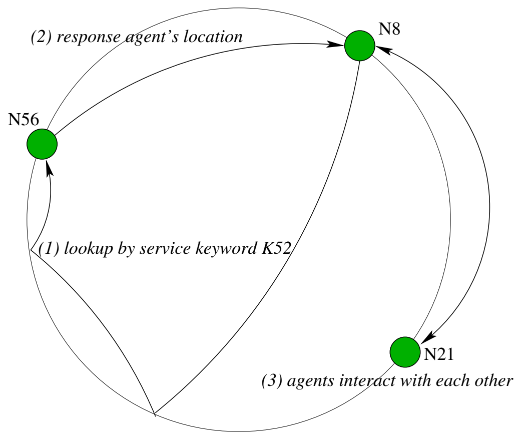
3.3.1 Join Operation

Initially, a MAS agent must call a join operation in order to become part of a Chord ring. The join operation triggers the construction of an agent's local information and updates the appropriate Chord ring. As a join operation parameter, the agent must then pass another agent that is already in the Chord ring. The join operation also performs the *stabilize*, *fix_fingers*, and *check_predecessor* functions of Chord API before building the Chord-required finger tables and successor lists. Join operation steps for agent a_1 with service key s_1 and a second agent (a_2) that is already in the Chord ring can be illustrated as

Steps of Join Operation
$a_1.join(a_2)$
$a_1.stabilize()$
$a_1.fix\_fingers()$
$a_1.check\_predecessor()$



(a) DSL update operation


(2) response agent's location
N8
N56
(1) lookup by service keyword K52
N21
(3) agents interact with each other

(b) DSL lookup operation

Figure 3.5: DSL operations

3.3.2 Update Operation

This operation updates information on the service provided by an agent to the directory. The agent must first identify the service key through consistent hashing, then look for the key in the Chord ring using the Chord API *find_successor* function. The agent finds the key's successor agent, which is responsible for maintaining the service location in its service directory. After the agent locates the key's successor, the agent sends that successor an update message that contains a service directory entry in the format <*service keyword, agent's location*>. After the successor receives the message, it updates the corresponding entry in its service directory.

An example of this operation is presented in Figure 3.5(a). In the figure, $N21$ looks for $K52$'s successor using the Chord API *find_successor* function. After several steps, $K52$'s successor—$N56$—receives the update message and updates its service directory.

3.3.3 Lookup Operation

In order to find a specific service, agents must know the service keyword before using the lookup operation to locate the service. This operation returns the service locations found in the directories of key successors. After receiving these locations, agents interact directly with the agents that provide the services. An example is given in Figure 3.5(b). In the example, $N8$ wants to find the agent that provides the service *gcd* with the hash value $K52$. First, the lookup operation calls *find_successor* several times to find $K52$'s successor, $N56$. After querying the service directory of $N56$, the lookup operation returns the location of $N21$ (which provides service *gcd*) to $N8$. $N8$ then interacts directly with $N21$.

3.3.4 Recovery Operation

In a dynamic, open MAS, agents regularly join, leave, and fail. These events often result in outdated local information. The DSL recovery operation helps agents handle these ongoing events. Chord API defines several methods (including *stabilize*, *fix_fingers*, and *check_predecessor*) to recover finger tables and successor lists. The recovery operation also triggers the update operation to update service directories using data from finger tables and successor lists.

Chapter 4

Experiments and Evaluations

To verify that the proposed DSL helps in the search for compatible agents, an attempt was made to improve the efficiency of platoon agent interactions in RoboCup Rescue without the benefit of center agents. Experimental results (e.g., fire spreading, message distribution, hops for finding successors) were analyzed in order to prove that the DSL avoids broadcast queries, single point failures, and performance bottlenecks.

4.1 RoboCup Rescue

The various agent categories in RoboCup Rescue are shown in Figure 4.1. Ambulance teams are responsible for rescuing injured civilians in disaster areas, fire crews are responsible for extinguishing fires, and police units are responsible for clearing roads that are blocked by collapsed buildings. These

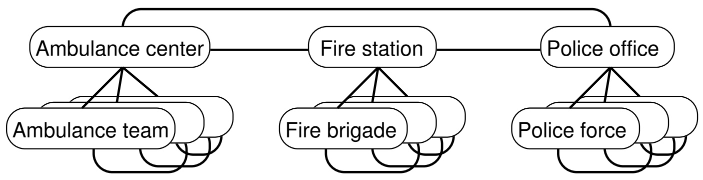


Figure 4.1: RoboCup Rescue telecommunication

three agent types come under the category of *platoon agents*. They have the ability to interact with each other over long distances using a) *telecommunication* via center agents, or b) in-person *oral communication*. For instance, if a fire crew cannot move through a blocked area, it can orally communicate its needs to any police units within 30 meters. RoboCup Rescue also has three corresponding *center agents*: ambulance centers, fire stations, and police stations. As shown in Figure 4.1, fire crews located in fire stations must use telecommunications to contact police units located in police stations. The respective stations are responsible for forwarding information from individual units in the field (C.2).

The main task of fire crews—extinguishing fires—is dependent on two key points: cooperation with and between crews, and the ability to quickly determine where fires or other disasters are occurring. They cannot get to fires or other emergencies if access roads are blocked. In such situations, they must inform police units of the blockages; in other situations, police units must inform fire units of the locations of burning buildings. If fire crews and police units are limited to oral communications, then the efficiency of firefighting crews will be severely restricted. In sections 4.3 and 4.4 I will describe simulations of these types of scenarios using RoboCup Rescue with and without center agents.

4.2 Rescue without Center Agents using DSL

Without the benefit of center agents, platoon agents in RoboCup Rescue are only able to telecommunicate with other platoon agents of the same type; in contrast, they can use oral communication with all types of platoon agents. Oral communication is limited to platoon agents who are within 30 meters of each other. For example, if fire crew B and police units C and

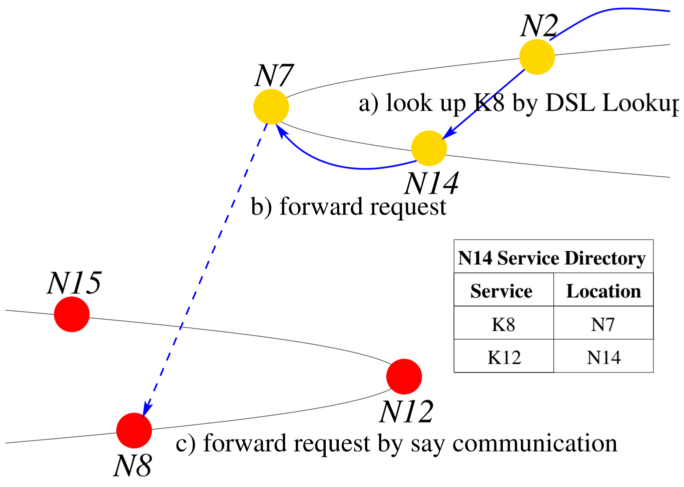
D are within 30 meters of fire crew A, then any communication sent by A will be immediately received by B, C, and D. In the real world, this is a very unusual situation. In order to facilitate communication between platoon agents of different types within 30 meters of each other, the proposed DSL system creates directories for service sharing—meaning that platoon agents of different types can interact without having to go through center agents.

In the example presented as Figure 4.2,

1. Platoon agents of the same type are organized into a Chord ring by a join operation.
2. Fire crew $N1$ needs help from police unit $N8$, whose service is represented by key $K8$.
3. $N1$ looks for $K8$'s successor via the lookup operation and learns that the successor is $N14$ (Fig. 4.2 (a)).
4. $N14$ queries its service directory and learns that agent $N7$ provides service $K8$.
5. $N14$ forwards a request to $N7$ (Fig. 4.2 (b)).
6. $N7$ receives the request and performs the service for $N1$—that is, it forwards the request to $N8$ via oral communication (Fig. 4.2 (c)).

4.3 Scenario Design

Morimoto [11] developed a Java API named Yab API to help speed up the process of designing agent AIs in RoboCup Rescue. Ako [1] et al. used it to design an agent AI named YowAI that won second place in the 2003


N2
N7
a) look up K8 by DSL Lookup
N14
b) forward request
N15
N12
N8
c) forward request by say communication

N14 Service Directory	
Service	**Location**
K8	N7
K12	N14

Figure 4.2: DSL lookup operation in RoboCup Rescue

RoboCup Rescue Contest. These YowAI agents were used in the experiments described in this section.

Three RoboCup Rescue scenarios were designed using different conditions; ambulance teams and the ambulance center were deleted for purposes of simplification and to enhance the focus on interactions between fire crews and police units (C.1,C.3,C.4). The scenarios used the default configuration of RoboCup Rescue: a Kobe City location with four initial burning buildings, a 300-second simulation period, and the number of agents shown in Table 4.1. The three scenarios were as follows:

YowAI with center agents showing a normal (i.e., non-DSL) result and a performance bottleneck problem.

YowAI without center agents showing that single point failures can cause agents to perform their jobs poorly.

YowAI+DSL without center agents showing that the proposed DSL improves agent performance, and highlighting the performance characteristics of the DSL system.

Table 4.1: Number of agents in different scenarios

	Police Office	Fire Station	Police Force	Fire Brigade
YowAI with center agents	1	1	10	10
YowAI without center agents	0	0	10	10
YowAI+DSL without center agents	0	0	10	10

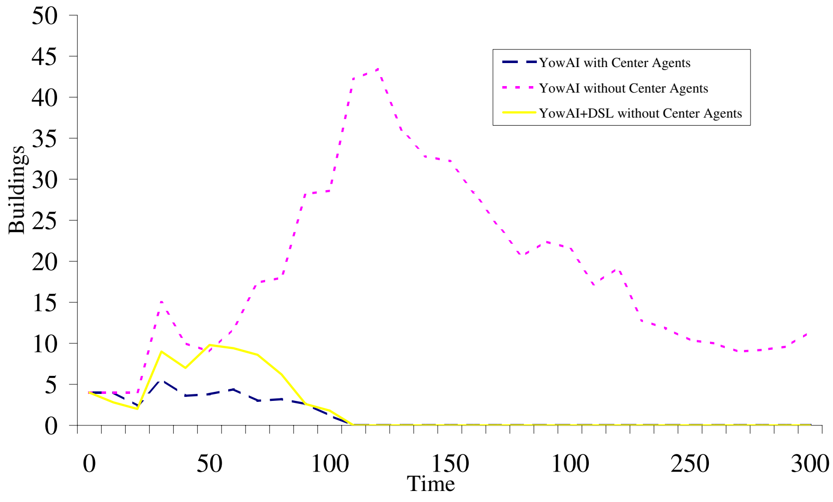


Figure 4.3: Number of burning buildings in the three scenarios

4.4 Results

4.4.1 Spread of Fire

Statistics from a number of burning buildings were used to verify if the fires in the scenarios spread or were extinguished. In each of the three scenarios, four buildings are burning as the simulation starts. The fires spread through the buildings very quickly, and become more and more difficult to control. If the fire crews and police units can work together quickly to locate and reach the fires, there is greater potential for extinguishing them before they spread.

The number of burning buildings during single fires in different scenarios (average numbers of simulations= 5) are shown in Figure 4.3. In the YowAI scenario with center agents, the number of burning buildings= 4; the fire

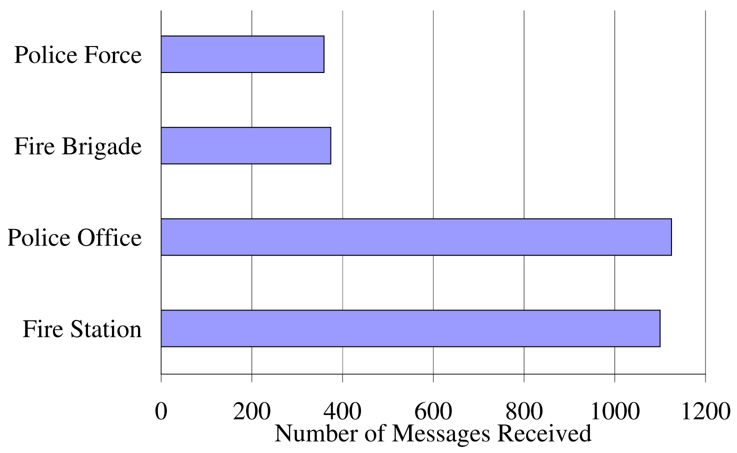


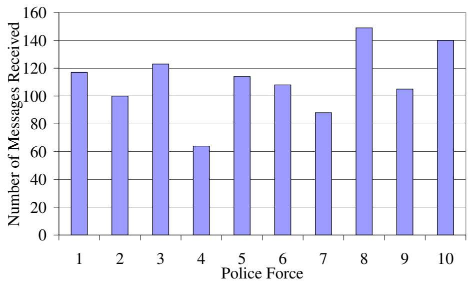
Figure 4.4: Number of messages received by agents in YowAI with center agents

is completely extinguished by the end of the 100-second simulation. In the YowAI scenario without center agents, the fire crews are not able to extinguish the fire quickly, and it spreads through more than 40 buildings by the time the simulation ends at 300 seconds. This result is due to a single point failure problem, since different types of platoon agents could not interact with each other without the assistance of center agents.

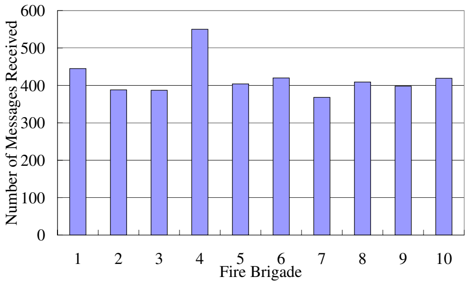
Finally, in the YowAI+DSL scenario without center agents, the fire crews extinguish the fire in 100 seconds, after it spreads through only 10 buildings. The proposed DSL system apparently offers an efficient way for different types of platoon agents to find each other and achieve a common goal without having to address a single point failure problem.

4.4.2 Message Distribution

A comparison was made between messages received by agents in the YowAI scenario with center agents and the YowAI+DSL scenario without center agents to analyze bottleneck problems and the load balance property of the proposed DSL system. As shown in Figure 4.4, in the YowAI scenario with



(a) Police forces



(b) Fire brigades

Figure 4.5: Number of messages received

center agents, the number of messages received by platoon agents (359 by police units and 374 by firefighting crews) was much greater than the number of messages received by center agents (1, 125 police-related messages and 1, 100 firefighter-related messages). According to these results, performance bottleneck problems occurred because the center agents had to handle a much larger number of messages than the platoon agents, thus increasing computation and communication overhead.

In the YowAI+DSL scenario without center agents, the biggest difference in the percentages of messages received by police units (Fig. 4.5(a)) and firefighters (Fig. 4.5(b)) was no more than 7 percent. It is therefore suggested that the reduced number of bottleneck problems associated with the proposed DSL is due to its balanced load characteristic.

4.4.3 Hops of Finding Successors

The term *hops for finding successors* refers to the number of messages that are used to look up a service location by its keyword. The term can be used to discuss system efficiency—that is, the fewer the number of messages, the fewer the number of agents involved, the less time required, and the more efficient the system. The proposed DSL system maps the hash values of service keywords onto a Chord ring agent. After five simulations, the average number of hops was 2.63 over $2,176$ lookup operations. In the YowAI scenario with center agents, 3 messages are required for a platoon agent to send a request to another type of platoon agent. In comparison, the DSL system requires 2.63 messages to look up a service location, plus one oral communication message and one telecommunication message to forward the request—in other words, it needs 4.63 messages for a platoon agent to send a request to another type of platoon agent within a 10-agent Chord ring. As the number of agents grows, the number of required hops increases at a rate of $\log(number\ of\ agents)$. The DSL system requires $O(\log(number\ of\ one\ type\ platoon\ agents))$ hops for a platoon agent to look up a service location. The proposed DSL system thus uses more messages, but avoids the problems that are associated with a system that uses center agents, and is therefore considered more efficient.

Chapter 5

Conclusion

The proposed DSL system allows for a decentralized method of service lookup in a MAS by building service directories that are distributed among agents. The system offers an efficient lookup operation for one agent to find compatible agents for service provision and goal-oriented interactions. Most importantly, the proposed DSL system avoids three problems that are very common in systems designed to bring together compatible agents: broadcast queries, performance bottlenecks, and single point failures. By simulating three firefighting scenarios with RoboCup Rescue, it was possible to show that the DSL system can avoid the single point failures and performance bottlenecks that are associated with the use of center agents.

Data on message distribution in the three scenarios shows that the proposed DSL system is capable of balancing message loads and improving interaction efficiency among different types of platoon agents without the use of center agents, while avoiding single point failures and performance bottlenecks. Each DSL operation requires only a few messages among a small number of agents—a significant improvement over the broadcast query method used by other systems. In addition, the proposed DSL system has a recovery operation feature for handling changing behaviors and situations associated with agents joining, leaving, or failing in a scenario. The proposed system is built on a Chord protocol that makes use of peer-to-peer methodology—in other words, no agent is considered more important than others in a Chord ring. The DSL system described in this paper is clearly decentralized.

Appendix A

Related Work

A.1 Middle Agents

Agents in MAS with middle agents can be categorized into three types: providers, requesters, and the middle agent. Providers are agents who provide services to others, and requesters are agents who need services. The interactions between these tree types of agents can be described as follows:

1. Providers advertise their services to the middle agent.

2. The middle agent stores these advertisements.

3. Requesters request services from the middle agent.

4. The middle agent finds if there are well-matched services and return an agent name list.

For example, *matchmakers* and *brokers* are two kinds of middle agents. In fact, the interaction between agents and the matchmaker is a little different from the interaction between agents and the broker. Figure A.1 shows the interaction between a single agent and the matchmaker.

(a) The provider advertises/unadvertises its service to the matchmaker.

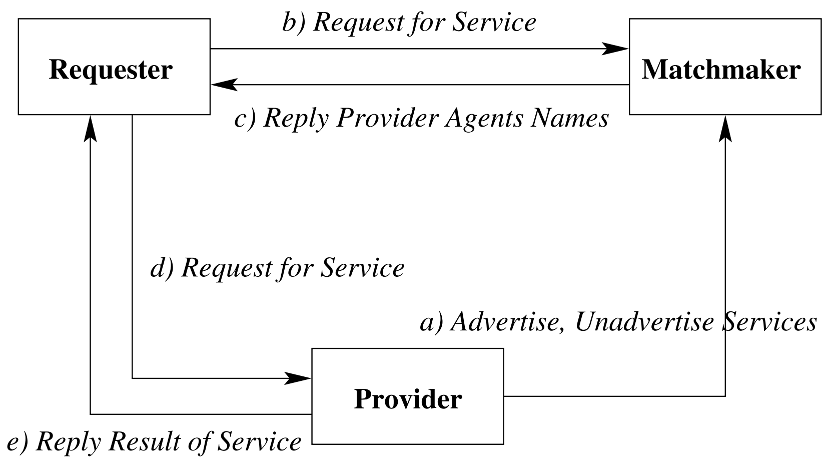

b) Request for Service
Requester
Matchmaker
c) Reply Provider Agents Names
d) Request for Service
a) Advertise, Unadvertise Services
Provider
e) Reply Result of Service

Figure A.1: Interaction between a single agent and the matchmaker

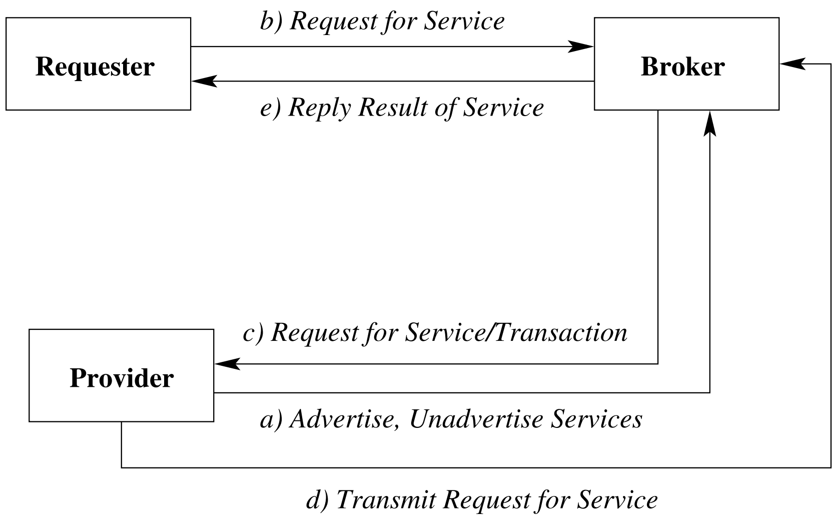

b) Request for Service
Requester
Broker
e) Reply Result of Service
c) Request for Service/Transaction
Provider
a) Advertise, Unadvertise Services
d) Transmit Request for Service

Figure A.2: Interaction between a single agent and the broker

(b) The requester requests service it needs to the matchmaker.

(c) The matchmaker replies the provider well-matched agent names.

(d) The requester requests service it needs from the provider who advertises the service.

(e) The provider replies results of service.

After the requester receives the agent names from the matchmaker, it interacts with the provider directly. Instead of interacting with the provider directly, the requester delegates request to the broker as follows (Figure A.2).

(a) The provider advertises/unadvertises its service to the broker.

(b) The requester requests service it needs to the broker.

(c) The broker requests service the requester needs from the provider who advertises the service.

(d) The provider replies results of service to the broker.

(e) The broker forwards results of service to the requester.

However, middle agents, such as matchmakers and brokers, may cause performance bottleneck because all agents (e.g., providers and requesters) in MAS have to interact with it. Moreover, single point failure occurs when the middle agent fails unexpectedly.

For instance, the interaction between the matchmaker and agents can be shown in Figure A.3

Both providers and requesters need to interact with the matchmaker directly. There could be heavy computation and communication overhead on the matchmaker. Thus, it may cause performance bottleneck, which makes

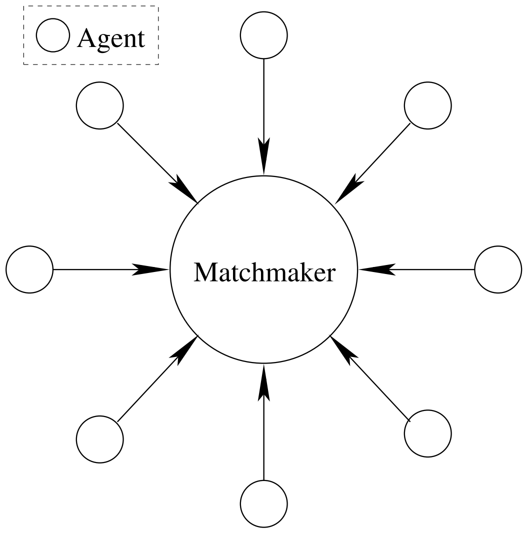


Figure A.3: Interaction between a single agent and the matchmaker

MAS have low performance because a few agents have heavy computation and communication overhead. Furthermore, when the matchmaker fails unexpectedly, providers cannot advertise their services, and requesters cannot find well-matched providers either. Single point failure occurs.

A.2 Agent Name Server

Agent name servers (ANSs) are special agents who map agent names to their locations in RETSINA [18]. ANSs must have ability to ask other ANSs when agents ask the mappings it does not know. Therefore, a mechanism is needed for ANSs to locate all the other ANSs in the system. Langley [9] et al. provide a discovery mechanism to ANSs, and the mechanism could be divided into two parts: locating ANSs in the same LAN and in WAN. When ANSs want to locate other ANSs in the same LAN, they use UPnP protocol, which is broadcast-based. Although broadcast-based protocol may cause broadcast query problem, it is reasonable because ANSs are in the same LAN. On the other hand, when ANSs want to locate other ANSs in WAN,

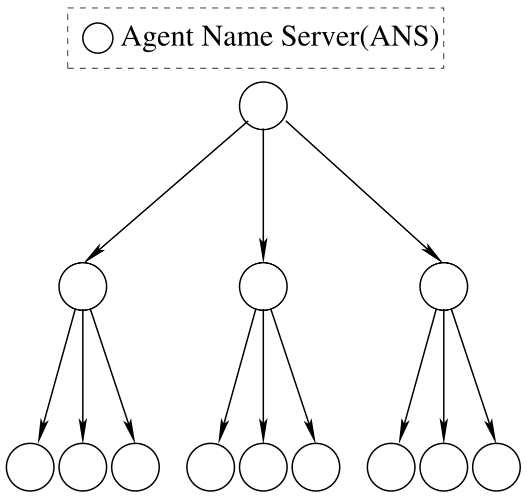


Figure A.4: The interaction between ANSs in WAN

they use Gnutella protocol. By Gnutella protocol, besides the mappings from agent names to their locations, each ANS keeps a few locations of other ANSs. When an ANS receives requests, the ANS asks other ANSs if it cannot satisfy the requests (Figure A.4). When an ANS wants to answer agent names it does not know, it asks the ANSs it knows. If the successive ANSs do not know the agent names either, they will ask other ANSs they know until the agent names are known by some ANS. Gnutella protocol takes lots of messages and involves lots of ANSs while performing such an operation. It may cause broadcast query problem.

Appendix B

Chord API

Chord API assists in the development of Chord-based applications. Chord API contains several methods for a node n to perform operations as the following:

Operation	**Description**
$n.create()$	construct a new Chord ring
$n.join(n')$	join Chord ring which has node n' already
$n.stabilize()$	update successor list and notify predecessor
$n.notify(n')$	notify n' to check predecessor
$n.fix\_fingers()$	check finger table entries if they are up-to-date
$n.check\_predecessor()$	check predecessor if it is up-to-date
$n.find\_successor(key)$	lookup *key*'s successor
$n.closest\_preceding\_node(id)$	lookup *key*'s successor from local information

Appendix C

Experiment Details

C.1 RoboCup Rescue Simulation Platform

This chapter will introduce the RoboCup Rescue simulation platform used in this paper, in which ambulance teams and the ambulance center were deleted for purposes of simplification and to enhance the focus on interactions between fire crews and police units. There are several components in this RoboCup Rescue simulation platform[1]:

1. RoboCup Rescue Agents (RCRAgents)
2. Sub-simulators
3. Geographical Information System (GIS)
4. Viewer
5. Kernel

In these components, platoon agents and center agents are called RCRAgents, sub-simulators simulate phenomenon in disaster area, such as spread

[1]All the components, such as GIS, kernel, and all sub-simulators can be executed at different computers, which are connected by network.

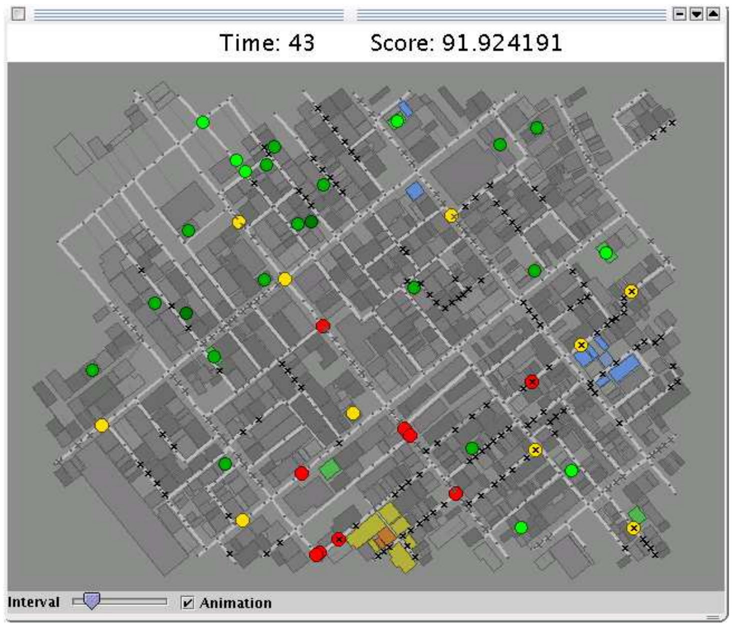


Figure C.1: Graphical view of disaster area provided by viewer

of fire and collapse of buildings, and GIS provides initial conditions of disaster area as simulation starts. Kernel manages network connections among these components, including inter-communication between RCRAgents, and viewer provides graphical view of disaster area during runtime (Figure C.1).

RCRAgents in the disaster area are shown as color circles: fire brigades are in red, police forces are in yellow, and civilians are in green. The colors of RCRAgents become dark when they get hurt, and the colors turn to black if they die. Besides, square blocks are buildings, and there are in several colors, which mean the burning time of buildings:

- **Gray** The buildings are in normal state without fire.
- **Yellow and Red** The buildings are burning, and the color turns yellow to red if the burning time of buildings becomes longer.

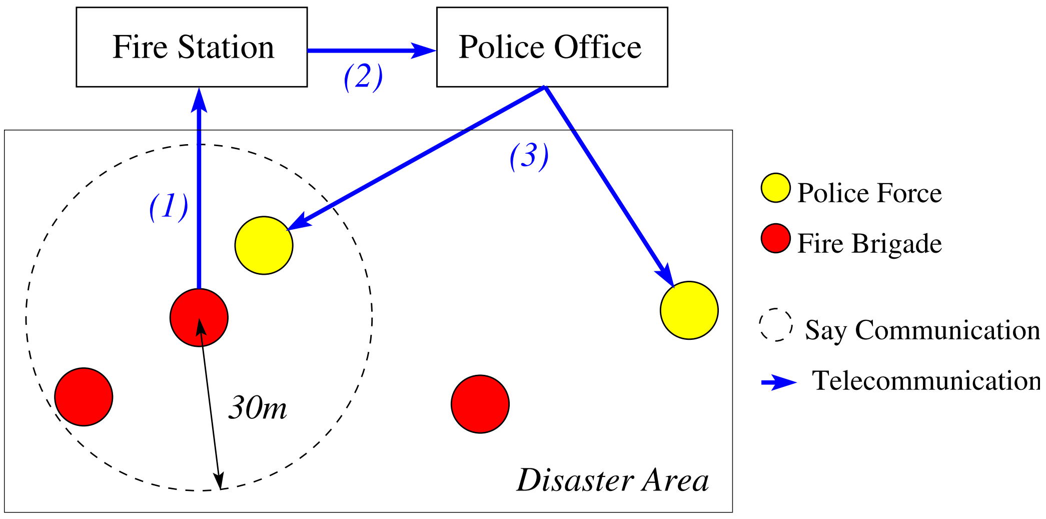


Figure C.2: Agent communication in RoboCup Rescue

- **Blue** The buildings were burning before, but the fire is put out before they are burnt out.
- **Dark Gray** The buildings are burnt out.

There are some buildings are inflammable, such as refuges which are in green on the map, and "x"'s are blockages which are caused by collapses of buildings.

C.2 RCRAgent Communication

There are two ways for RCRAgents to inter-communicate (Figure C.2).

1. **Say Communication** platoon agents talk via natural voice.
2. **Telecommunication** platoon agents talk via telecommunication devices.

Say communication prompts platoon agents to communicate with all types of platoon agents within 30 meters of disaster area. When kernel receives a message of say communication sent by a platoon agent, it duplicates the message and forwards duplicate messages to other platoon agents within 30 meters.

On the other hand, telecommunication prompts platoon agents to communicate with same type of platoon agent and the corresponding center agent. In addition, telecommunication prompts center agents to communicate with corresponding type of platoon agents and all other center agents. Therefore, when kernel receives a message of telecommunication sent by a fire brigade, it duplicates the message and forwards duplicate messages to all fire brigades and fire station. When kernel receives a message of telecommunication sent by a police office, it duplicates the message and forward duplicate messages to all police forces and fire station.

Thus, when a RCRAgent sends one messages to some type of platoon agents, and platoon agents of that type receive the same message. In order to implement DSL and verify its properties, it makes no sense because it is difficult to verify that DSL avoids broadcast query. For example, when DSL calls lookup operation, DSL searches successor of the key among the agents in a Chord ring, which consists of platoon agents in one type (e.g., fire brigades). DSL prefers that if a RCRAgent sends one message, then there will be only one RCRAgent that can receive it. So, I make small modifications to the kernel. The new kernel receives a message of telecommunication sent by a fire brigade, it forwards the message to a specified fire brigade or fire station instead of duplicating the message. With the new kernel, when a platoon agent sends a message of telecommunication, it has to specify which one it would like to send to. The new kernel makes it possible to observe the number of messages received/sent by platoon agents in the same type (e.g., fire brigades) and verify that DSL is load-balanced and decentralized.

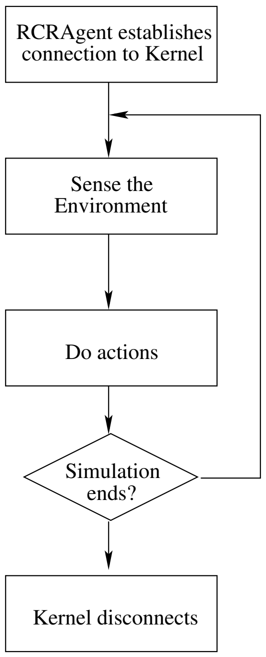


Figure C.3: RCRAgent life cycle

C.3 RCRAgent Life Cycle with DSL

DSL provides four operation: join, update, lookup, and recovery for agents to find other suitable agents to interact with by sharing services.

In RoboCup Rescue, RCRAgents sense information from the environment and take actions continuously during the simulation. Figure C.3 shows the life cycle of RCRAgents. After the simulation starts, RCRAgents establish connections to the kernel. Then, RCRAgents perform their jobs by sensing information from the environment and taking actions continuously. After the simulation ends, kernel disconnects all RCRAgents.

On the other hand, when RCRAgents interact with each other with DSL, they need some extra steps to maintain their local information needed by DSL. After establishing connections to the kernel, RCRAgents have to invoke join operation to initialize their local information and inform their successors

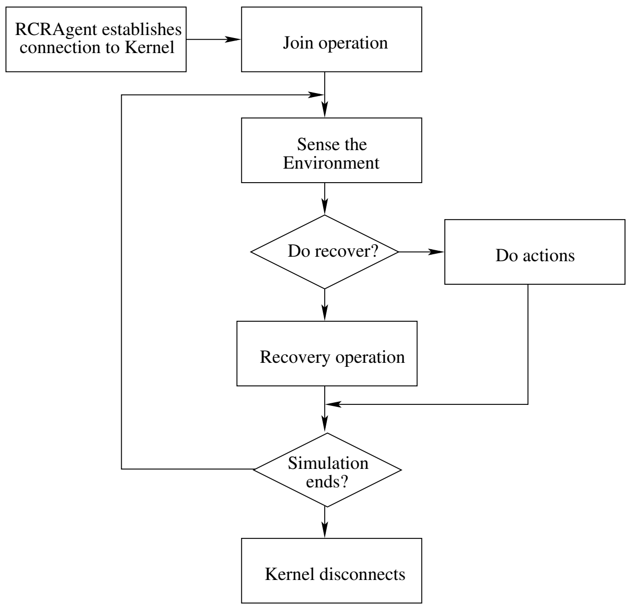


Figure C.4: RCRAgent life cycle with DSL

that they join the system. Besides, after RCRAgents sense information from environment, they have to decide if they have to update service they provide (update operation) and local information (recovery operation) or take actions as Figure C.4. In experiments of this paper, RCRAgents call update operation every 10 simulation cycle.

C.4 Parameter Configuration

In Section C.3, DSL makes RCRAgents perform additional operations in their lift cycles in order to interact with others without center agents. In fact, there are some constraints in RoboCup Rescue, which is a real-time system. For example, RCRAgent communication is asynchronous, and RCRAgents can take only one action in one simulation cycle. It means RCRAgents can send as many messages as they want no matter how many cycles does it take. As a result, RCRAgents take no action if they sends too many messages in

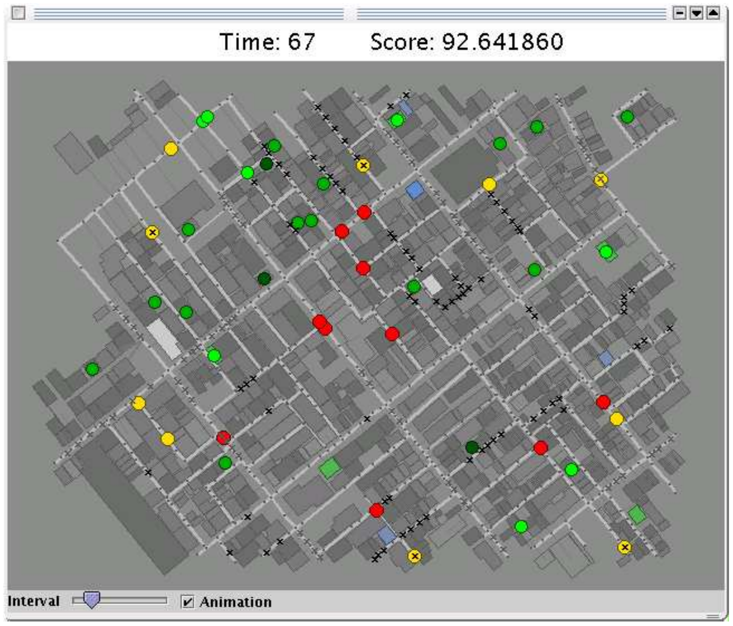


Figure C.5: Result in YowAI with center agents

one cycle. Therefore, when RCRAgents use DSL to interact with each other, they must not perform too many DSL operations. Although DSL operations need only a few messages to complete the job, these messages are too many for such a real-time system.

This paper uses several parameters to control RCRAgent behaviors to keep a balance between performing DSL operations and doing their jobs.

- **NumDiff** The number of messages which have to be sent if platoon agents want to communicate another type of platoon agents is limited, and other messages will be discarded.
- **DSLLimit** The number of DSL messages that are allowed to be handled in one simulation cycle is limited, because considering too many messages causes platoon agents not to take at least one action in a simulation cycle.

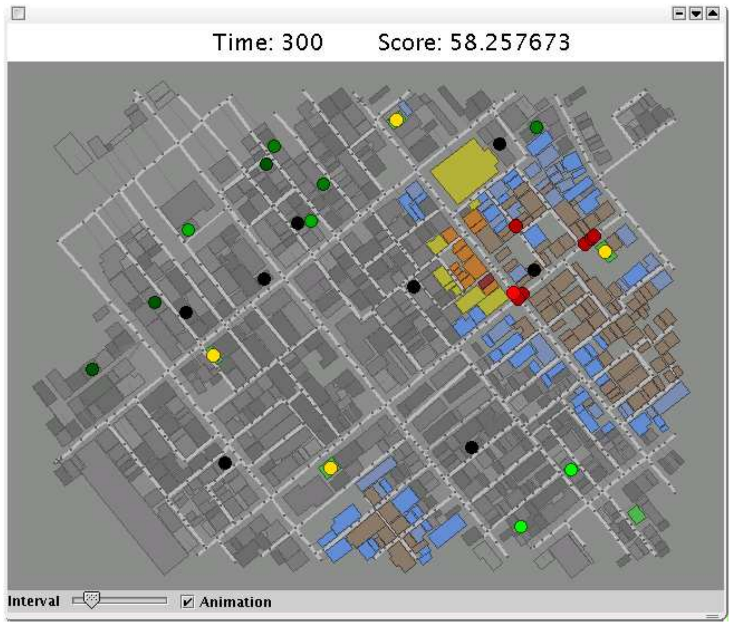


Figure C.6: Result in YowAI without center agents

- **NonDSLLimit** The number of non-DSL messages allowed to sent by a platoon agent in one simulation cycle is limited, and it can make sure that platoon agents have time to take their actions.
- **DSLUpdate** Finally, the period for platoon agents to do update or recovery operations of DSL is controlled, because doing these operations too frequently not only does not improve the efficiency of platoon agents but causes platoon agents to have no time doing their jobs.

After we simulate different combinations of parameter configurations and find out the best one. It is $(NumDiff, DSLLimit, NonDSLLimit, DSLUpdate) = (1, 4, 2, 10)$. We use this parameter configuration in all simulations of this paper.

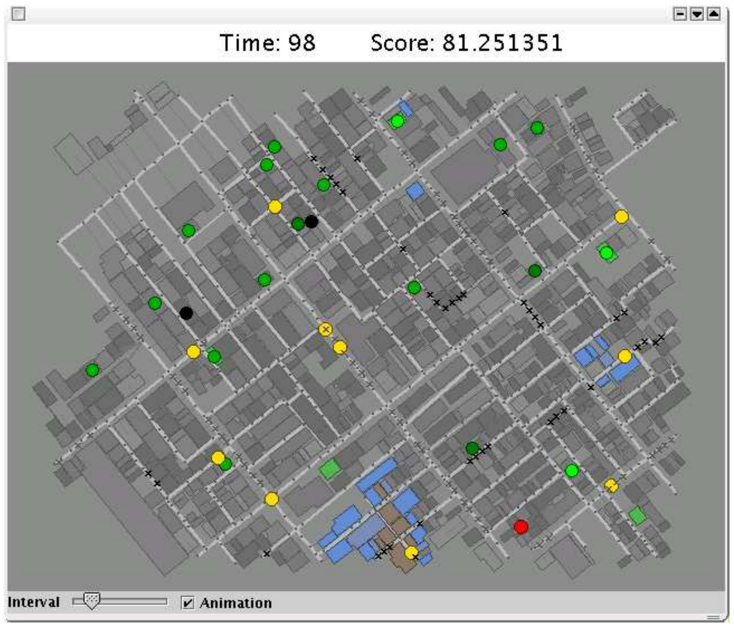


Figure C.7: Result in YowAI+DSL without center agents

C.5 Spread of Fire Screenshots

Figure C.5 shows the result of YowAI with center agents scenario. After 67 seconds, the 4 initial fires are all put out without spreading through other buildings. This shows that platoon agents have very good performance with the benefit of center agents.

In the scenario of YowAI without center agents, the fires are not put out, and spread through many other buildings. A lot of buildings are burnt out. The result is shown in Figure C.6. Platoon agents cannot perform their jobs well without center agents.

In the scenario of YowAI+DSL without center agents, platoon agents can use DSL to help them find others to interact with. The result is shown in Figure C.7. Although the fires spread through a few buildings, platoon agents put out the fires after 98 seconds. It shows that DSL provides an efficient way for platoon agents to find others to interact with even if there are no center agents.

Bibliography

[1] Takuya Ako, Toshihiro Suzuki, Shin'ichi Takahashi, and Ikuo Takeuchi. YowAI2003 team profile. `http://ne.cs.uec.ac.jp/~taku-ako/YowAI2003.html`.

[2] Christopher H. Brooks and Edmund H. Durfee. Congregation formation in multiagent systems. *Autonomous Agents and Multi-Agent Systems*, 7:145–170, September 2003.

[3] Keith Decker, Katia Sycara, and Mike Williamson. Middle-agents for the internet. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, Nagoya, Japan, 1997.

[4] Tim Finin, Yannis Labrou, and James Mayfield. KQML as an agent communication language. In N. Adam, B. Bhargava, and Y. Yesha, editors, *Proceedings of the 3rd International Conference on Information and Knowledge Management*, pages 456–463, Gaithersburg, MD, USA, 1994. ACM Press.

[5] Leonard N. Foner. Yenta: A multi-agent, referral-based matchmaking system. In *The First International Conference on Autonomous Agents*, February 1997.

[6] John R. Graham, Keith S. Decker, and Michael Mersic. DECAF - A flexible multi agent system architecture. *Autonomous Agents and Multi-Agent Systems*, 7:7–27, July 2003.

[7] Adriana Iamnitchi and Ian Foster. On fully decentralized resource discovery in grid environments. In *International Workshop on Grid Computing*, Denver, Colorado, USA, November 2001. IEEE.

[8] Hiroaki Kitano. Robocup Rescue: A grand challenge for multi-agent systems. In *MultiAgent Systems, 2000. Proceedings. Fourth International Conference on*, pages 5–12, July 2000.

[9] Brent K. Langley, Massimo Paolucci, and Katia Sycara. Discovery of infrastructure in multi-agent systems. In *Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 1046–1047, Melbourne, Australia, 2003. ACM Press.

[10] David J. Martin, Adam J. Cheyer, and Douglas B. Moran. The Open Agent Architecture: A framework for building distributed software systems. *Applied Artificial Intelligence*, 13(1–2):91–128, 1999.

[11] Takeshi Morimoto. YabAPI: API to develop a RoboCup Rescue agent in Java. `http://ne.cs.uec.ac.jp/~morimoto/rescue/yabapi/`.

[12] Marian H. Nodine and Amy Unruh. Facilitating open communication in agent systems: The InfoSleuth infrastructure. In *Agent Theories, Architectures, and Languages*, pages 281–295, 1997.

[13] Onn Shehory and Sarit Kraus. Methods for task allocation via agent coalition formation. *Artificial Intelligence*, 101(1–2):165–200, May 1998.

[14] R. G. Smith. The Contract Net Protocol: High-level communication and control in a distributed problem solver. *IEEE Trans. On Computers*, 29(12):1104–1113, 1980.

[15] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.

[16] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking*, 11:17–32, February 2003.

[17] Katia Sycara, Matthias Klusch, Seth Widoff, and Jianguo Lu. Dynamic service matchmaking among agents in open information environments. *SIGMOD Record (ACM Special Interests Group on Management of Data)*, 28(1):47–53, March 1999.

[18] Katia Sycara, Massimo Paolucci, Martin van Velsen, and Joseph Giampapa. The RETSINA MAS infrastructure. *Autonomous Agents and Multi-Agent Systems*, 7:29–48, July 2003.