분산 멀티에이전트 시스템의 상호협력 제어

백 순 철[†] · 최 중 민[††] · 임 영 환[†††] · 장 명 욱[†] ·
박 상 규[†] · 이 광 로[†]

요 약

공동 작업 수행을 위하여 상호 긴밀하게 협력하는 분산 멀티에이전트 기반구조상에서 각 에이전트의 기능은 다른 에이전트들에게 전달되어져야 한다. 즉, 멀티에이전트 사회 속에서 각 에이전트는 어떤 다른 에이전트들이 사용 가능한지, 그들이 어떤 문제들을 해결할 수 있는지, 그리고 그들과 정보를 주고받기 위한 방법은 어떤 방법이 있는지 알고 있어야 한다. 이러한 특성은 공동 작업 수행을 위한 에이전트들간의 지역적 또는 광역적 상호협력을 제어하기 위하여 에이전트들간의 통신 방식을 요구한다.

본 논문에서는 컴퓨터 비서로서 개발된 분산 멀티에이전트 기반구조인 MASCOT 플랫폼 상에서 에이전트들간의 상호협력을 제어하기 위한 기법을 제시한다. 이를 위하여 프레임 형태의 통신 패킷을 정의하였으며, 메시지 교환을 위한 프로토콜을 제시하였다. 또한, MASCOT 상에서의 에이전트들간의 상호협력을 제어하기 위한 통신 방식을 설명하기 위해 하나의 시나리오를 제시하였다.

Interaction Control in a Distributed Multiagent System

Soon-Cheol Baeg[†] · Joong-Min Choi[††] · Young-Hwan Lim[†††] · Myeong-Wuk Jang[††††] ·
Sang-Kyu Park[††††] · Gowang-Lo Lee[††††]

ABSTRACT

In a distributed multiagent framework, the capabilities of each agent are known to other agents. Namely, each agent in a multiagent society is aware of what agents are available in the whole society, which is able to solve a query, and how to contact them. This characteristic leads to the simplicity in controlling both local and remote interactions among agents by using a fixed form for communication packets.

This paper presents methods for controlling interactions among agents in this distributed multiagent framework. Agent interactions are described within the platform of MASCOT that is a tightly coupled multiagent system developed for the role of a computer secretary. A frame-like form of a communication packet is defined, and protocols for message exchanges are presented. Also, a scenario is given to demonstrate how the communication mechanism controls agent interactions in MASCOT.

1. Introduction

Controlling interactions among agents in a distributed multiagent framework has been a challenging issue [4, 6, 7, 10]. The types of interactions include requesting jobs, replying with answers, reporting control information such as activation status or error messages, and so on. The control of agent interactions is a difficult

† 정 회 원:한국전자통신연구소 인공지능연구실
†† 정 회 원:한양대학교 전자계산학과
††† 종신회원:숭실대학교 컴퓨터학부
논문접수:1995년 8월 18일, 심사완료:1996년 8월 30일

task mainly because agents are normally heterogeneous and lack the knowledge about other agents. In this loosely coupled system, approaches for controlling agent interactions have been focusing on developing mechanisms for determining who can solve which task. Efforts for standardizing inter-agent communication languages such as KQML belong to this category[3, 4]. Another difficulty lies in the management of common ontology for vocabularies that are exchanged among agents. Namely, it is possible that the same word means different things (ambiguity), or different words mean the same thing in different agents (conflicts) among different agents[5].

Compared to this, this paper presents a mechanism for controlling interactions among agents in a tightly coupled system. In this system, an agent is aware of what agents are available, which is able to solve a query, and how to contact them, albeit the heterogeneity still remains. Typically, a tightly coupled multiagent system is built in a top-down manner by disassembling a monolithic agent into a set of task-specific agents. In contrast, a loosely coupled system is the bottom-up integration of separate agents which already exist or are newly built[7].

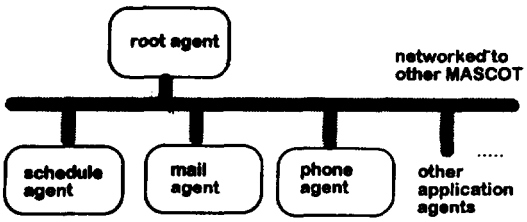
The main characteristic in the tightly coupled multiagent system is the simplicity and efficiency in the control of both local and remote interactions among agents. The form of the communication packet used in information exchange is predetermined and transparent to all agents. The communication packet includes the control part that specifies the sender, the receiver, and the type of the message, and the data part that specifies the action and the content. Although a simple point-to-point protocol might be adopted for this framework, it has a disadvantage that the communication module of each agent ought to take care of synchronization of asynchronously incoming messages. Hence, we employ a coordinator called the root agent whose responsibility is to collect communication messages, check the availability of receiving agents, and route each message to them. Consequently, the communication module of an agent is simplified since the root agent prevent each agent from receiving any message while it is busy performing other task.

In this paper, we present MASCOT (Multi Agent System for COmputer secreTary) which is a tightly coupled multiagent system developed for the role of a computer secretary. The task of a computer secretary is divided into a number of sub-secretaries or sub-agents, namely, the scheduling agent, the mail agent, the phone agent, and so on, and each agent knows what other agents are able to perform for cooperative jobs. We especially focus on its capability on controlling interaction among agents. Other issues including the intelligent user interface are described in [8].

This paper is organized as follows. Section 2 describes the architecture of MASCOT with the description of the root and some application agents. Section 3 presents the data structures and protocols for controlling interactions in MASCOT. Section 4 provides a typical scenario and describes it in terms of MASCOT agent interactions. Section 5 concludes with a summary and future directions.

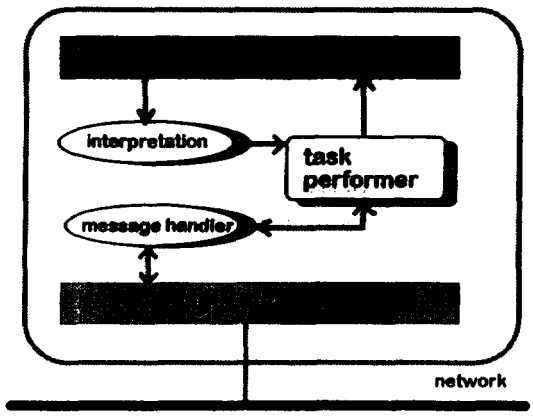
2. System Architecture of MASCOT

MASCOT is a multiagent system developed for the role of a computer secretary. Examples of the capabilities of MASCOT are: 1) handling daily mundane activities such as setting up individual or group meetings, 2) presenting summarized information about newspaper articles or stock trends, 3) reads and replies to electronic mails, 4) keeping track of the current whereabouts of the user to forward urgent messages. To achieve these tasks, MASCOT manages several application agents, and we describe three of them here-the schedule agent, the mail agent, and the phone agent. Interactions among these agents are controlled by the root agent. The system architecture of MASCOT is shown in Fig. 1.



(Fig. 1) MASCOT architecture

An application agent may have its own interface so that the user can issue a query to the application agent directly. Note that the user can also ask a query to the root agent that interprets it and delivers a mission to the proper agent. Fig. 2 shows the functional structure of a typical application agent consisting of the user interface, the interpretation module, the communication module, the message handler, and the task performer. The interpretation module analyzes a user request directly asked through the user interface in a restricted natural language style or in a frame-based form. The communication module receives and sends communication messages. Note that the communication module of an application agent does not maintain a message queue. This is because the root agent takes care of the synchronization of several messages to the same application agent, and as a result, does not send a message to an application agent when it is busy. The message handler is responsible for interpreting incoming communication messages and building outgoing messages. The task performer is the core module that executes domain specific tasks according to the input provided by the interpretation module and the message handler.



(Fig. 2) Functional modules of an application agent

2.1 Application Agents

A few application agents are implemented in MASCOT, and three of them are explained here.

The schedule agent (SA) manages individual schedules, handles group schedules such as team meetings or seminars, detects and resolves time conflicts with existing schedules, and provides automatic scheduling by finding free slots in the user-preferred time zone like the mornings or business hours[1]. The user can issue a command to SA in a restricted natural language style such as "get me all schedules about [meeting] this week". In handling group schedules in particular, SA interacts with a remote SA in other MASCOT to decide whether it should set up a meeting as specified, reschedule it, or reject it. Note that the decision is made depending on some heuristic information including 1) the number of available members for the meeting, 2) the authority level of each member, and 3) the importance of the meeting itself.

The mail agent (MA) is needed when the user wants to perform some actions for specific mail messages. To do this, the user can specify monitoring conditions in terms of the sender, the subject, and the arrival time of incoming mails, and register them to MA. Each monitoring condition is associated with an action. Implemented actions include forwarding, deleting, printing, alarming, and reading.

MA monitors the arrival of a new mail and, if any of the monitoring conditions is satisfied for the new mail, it performs the associated action. Especially for the forwarding action, MA interacts with SA to know where the user currently is, and also with PA to forward the message to the user by phone.

The phone agent (PA) makes phone calls, checks

the pre-defined password to identify the correct recipient, and delivers a given message to the recipient. The voice synthesizer converts a text message into voice before sending. PA is normally given a number to call, but when the number is not specified, PA refers its phone directory to find out the phone number of a person or a place.

All agent interactions are performed by message exchanges among agents. The root agent serves as a router for these communication messages. In other words, all messages are first sent to the root agent even though the destination is known at the time of sending. Then, the root agent relays it to the destination agent. By this coordination, the communication module of an application agent is simplified, since the root agent is responsible for synchronizing the processing of several messages going to the same application agent.

The root agent is also responsible for activating and deactivating application agents. Initially, only the root agent is running, and the application agents become activated as needed by the control of the root agent. This scheme is especially effective in an environment with limited resources since all application agents need not be running all the time as separate processes.

3. Controlling Agent Interactions

3.1 Communication Packets

A communication packet has a frame-like form with attribute-value pairs, and consists of the following fields: **FromAgent**, **FromHost**, **ToAgent**, **ToHost**, **MsgType**, **Action**, and **Content**. **FromAgent** specifies the sending agent. **FromHost** is the symbolic host name on which **FromAgent** is operating. **ToAgent** specifies the receiving agent. **ToHost** is the symbolic host name on which **ToAgent** is operating. **MsgType** distinguishes whether the type of message is a query or a reply. **Action** specifies the action to be performed by **ToAgent**. Finally, **Content** is the data needed to perform the specified action.

For example, when Hana wants to set up a meeting with Duri, Hana's SA constructs and sends **packet1** to the root agent by which it is relayed to Duri's SA. Here, **sched1** denotes the actual schedule object, not a pointer, that is passed to Duri's SA. Another example is when MA asks SA to obtain the user's location, denoted by packet2.

```
packet1
        FromAgent : SCHED
        FromHost : Hana
        ToAgent : SCHED
        ToHost : Duri
        MsgType : Query
        Action : RegisterSched
        Content : sched1
```

```
packet2
        FromAgent : MAIL
        FromHost : Hana
        ToAgent : SCHED
        ToHost : Hana
        MsgType : Query
        Action : GetLocation
        Content :
```

The main feature in this form of communication packets is that the destination agent is predetermined. This is possible since in a tightly coupled system an agent knows which agent is responsible for the specified action. Without the need for determining the destination, this leads to a simple communication module in both the root agent and the application agents. The data structure for the **Content** field is not fixed, and is dependent upon the type of action specified in the **Action** field. For example, it can be a schedule instantiation for the *RegisterSched* action, but it can

be just a string for the reply of the *GetLocation action*.

3.2 Message Exchange Protocol

For the exchange of communication messages, a communication module (CM) is managed by the root agent and each of the application agents. Since all communication messages are routed via the root agent, CM of each application agent sends all outgoing packets to the root agent regardless of the final destination. CM of the root agent relays incoming messages in two ways by examining the **ToHost** field of the communication packet. If **ToHost** is the same as the local host, the root agent sends the packet to **ToAgent** in the local MASCOT. If **ToHost** is a remote host, the root agent sends the packet to the root agent of MASCOT running on **ToHost**. Upon receiving a packet from a remote root agent, CM of the root agent sends it to the agent specified as **ToAgent**. When CM of the application agent receives the packet from its root agent, it checks **Action** and **MsgType** and executes an appropriate routine for which **Content** serves as the data. The protocol of message routing in the CM of the root agent is represented by a pseudo code in Fig. 3.

```
// 'packet' is a variable that holds the packet data
// sent from an application agent to the root agent
// 'send_packet (p, a, h)' is a function that sends a
// packet 'p' to the agent 'a' in the host machine 'h'
// 'LocalHost' holds the name of the local host machine

If packet.ToHost = LocalHost
        then send_packet(packet, packet.ToAgent,
        LocalHost)
        else send_packet(packet, 'ROOT', packet.
        ToHost)
```

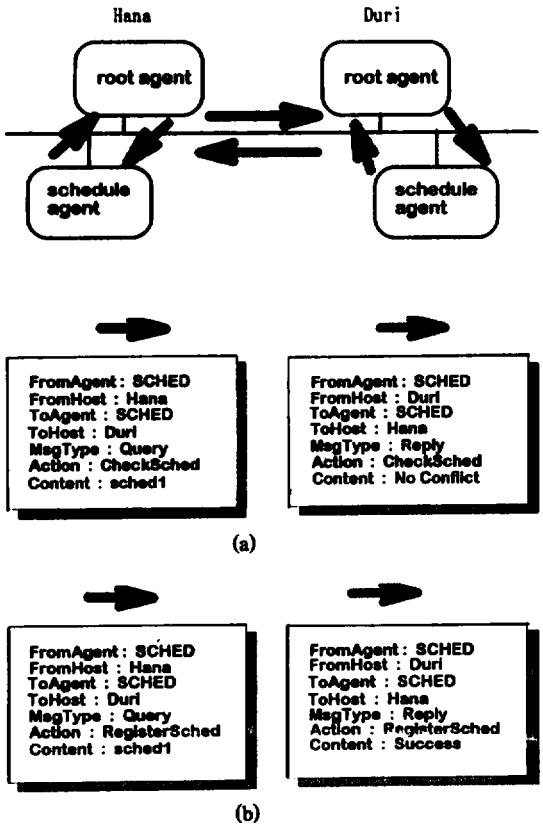
(Fig. 3) Message routing protocol of the root agent

It is possible that an agent can send the same packet to several agents. For example, if a person needs to make a group meeting with three people, the SA of that person would send a requesting packet three times to the SA of each person who participates in the meeting. Since a group schedule can be made only when all participants have confirmed. a counter should be maintained in the sending agent.

Since the agents are running asynchronously, the coordination of communication messages is needed when several communication messages are sent back and forth among application agents. For this purpose, the root agent keeps track of the activation status of the application agents. There are three types of activation status for an agent: EXIT, READY, and BUSY. EXIT denotes that the agent is not activated or initiated, READY denotes that the agent is initiated but is currently idle, and BUSY denotes that it is activated and currently processing a job. Initially, all application agents have the EXIT status. When an application is needed to be activated, the root invokes it, changes its status to READY by sending a control communication message to the application agent. An agent with the READY status can process the message from the root or other agents. If an application agent wants to communicate with other application agent that is currently BUSY, the root agent puts the message in a message queue that will be dispatched and sent to the destination agent as soon as the status is changed to READY.

The message exchange protocol described above is illustrated by an example. Suppose Hana wants to set up a group schedule with Duri. Two actions are involved in this task; *CheckSched* for checking time conflicts with existing schedules, and *RegisterSched* for actually adding the new schedule to the schedule DB. First, Hana's SA sends a *CheckSched* message to Duri's SA through the root agent. Then, Duri's SA acknowledges back by saying that there are no conflicts. Now, Hana's SA sends a *RegisterSched* message to Duri's SA. Duri's SA then adds the new schedule to the schedule DB and acknowledges that the new

schedule is successfully added. Fig. 4 (a) and (b) shows the flow of message exchanges for *CheckSched* and *RegisterSched* actions, respectively, with the description of actual messages.



(Fig. 4) Message exchanges for setting up a new group schedule

4. Interactions in the Scenario

In this section, we present a typical scenario that involves a number of local and remote agent interactions.

"Hana, Duri and Sena are managers in a company. Hana wants to have a business meeting with Duri at 3 in the afternoon. Hana asks his secretary SecA to contact with Duri's secretary SecB whether Duri is free at 3. SecB confirms that Duri is free then and both secretaries put a new schedule "business meeting-Hana with Duri" in the 3 PM slot. At 2 PM, Hana informs SecA that he will be at Sena's office and be back by 3 for the meeting with Duri. At 2:30, the president of the company calls Duri that they should meet at 3. Since it has a higher priority, Duri asks SecB to cancel the meeting with Hana. Now, SecB informs SecA of this situation, and SecA looks up the telephone number of Sena's office in the phone directory, calls the number, and tells Hana that the meeting is canceled so he doesn't have to be back by 3. Hana confirms and continues to work with Sena."

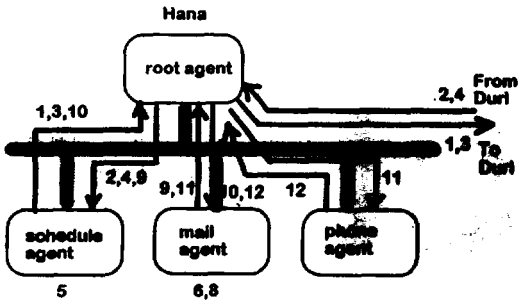
We can describe this scenario with a sequence of interactions performed by the application agents in MASCOT. For each step, the information about the action and the message type is described in a parenthesis.

1. To set up a meeting with Duri, Hana's SA sends a message to Duri's SA to check if Duri is available at 3PM. (*CheckSched Query*)
2. Duri's SA replies to Hana's SA that Duri is free at that time. (CheckSched Reply)
3. Hana's SA sends a message again to Duri's SA to put a new schedule titled "business meeting (Hana and Duri)" to the 3 PM slot. (*RegisterSched* Query)
4. Duri's SA changes its schedule DB and acknowledges to Hana's SA. (*RegisterSched* Reply)
5. Hana adds a new schedule to SA that he will be in Sena's office until 3.
6. Hana stores a monitoring condition to MA by expressing forward [urgent] mails to me.
7. Since Duri has to attend a more important meeting, Duri sends an email to Hana with the subject urgent situation saying that he can't make the meeting.
8. Hana's MA recognizes that the arrived mail matches the monitoring condition for forwarding

by comparing the subject part of the mail with monitoring conditions.

9. Hana's MA asks its SA to get the current whereabouts of Hana. (*GetLocation* Query)
10. Hana's SA replies with the information that he is in Sena's office. (*GetLocation* Reply)
11. Hana's MA asks PA to deliver Duri's message to Hana. (*ForwardMsg* Query).
12. Hana's PA calls at Sena's office, checks the password to make sure Hana picked up the phone, and replays the mail message by voice synthesis. After calling, PA replies to MA that it successfully forwarded the message. (*ForwardMsg* Reply)

Fig. 5 shows the flow of message exchanges for this scenario. In this figure, each number denotes the sequence of actions and message flows as described in the above scenario.



(Fig. 5) Message exchange flows for the scenario

5. Conclusion

We have presented a message-based communication method for interactions among agents in a distributed multiagent framework in which the capabilities of agents are known to other agents. This approach leads to the simplicity in controlling both local and remote agent interactions using a fixed form for communication packets. However, it has difficulties in exchanging messages with other loosely coupled systems, and we are currently working on building a multiagent framework based on an open agent architecture[2] to overcome this drawback and support more heterogeneity and openness.

Another topic we are also interested in is the separation of the user interface from the agent core engine to facilitate a portable agent system installed in a small computer such as a notebook or a PDA. This topic produce many intersesting issues in the control of interactions between the user interface that has a minimal processing overhead focusing on the display and the remotely-running agent core that is actually hidden from the user but performs most operational tasks[9].

REFERENCES

[1] Joongmin Choi and Sang-Kyu Park, "An Agent-based Automatic Schedule Management System (in Korean)," Proceedings of the 21st Korean Information Science Society Fall Conference, Seoul, Korea, pp. 715-718, 1994.

[2] Philip R. Cohen, Adam Cheyer, Michelle Wang, and Soon Cheol Baeg, "An Open Agent Architecture," Working Notes of AAAI Spring Symposium on Software Agents, pp. 1-8, 1994.

[3] Tim Finin, Richard Fritzson, Don McKay, and Robin McEntire, "KQML as an agent communication language," Proceedings of CIKM'94, The ACM Press, 1994.

[4] Michael R. Genesereth, "An Agent-based Approach to Software Interoperability," Technical Report Logic-91-6, Logic Group, CSD, Stanford University, 1993.

[5] Thomas R. Gruber, "Ontolingua: A Mechanism to Support Portable Ontologies," Technical Report KSL 91-66, Knowledge Systems Laboratory, Stanford University, 1992.

[6] R. V. Guha and Douglas B. Lenat, "Enabling Agents to Work Together," Communications of ACM, vol. 37, no. 7, pp 127-142, 1994.

[7] Henry A. Kautz, Bart Selman, and Michael Coen, "Bottom-up Design of Software Agents," Communications of ACM, vol. 37, no. 7, pp. 143-146, 1994.

[8] Sang-Kyu Park, Gowang-Lo Lee, Joongmin Choi, Myeong-Wuk Jang, Young-Hwan Lim, Chee-Hang Park, and Ji-Young Choi, "An Agent-based User Interface System (in Korean)," Proceedings of UNIEXPO'94, Seoul, Korea, pp. 23-27, 1994.

[9] T. Rodden, P. Sawyer, and I. Sommerville, "Vista:A User Interface For a Distributed Object-Oriented Software Engineering Environment," Software Engineering Journal, pp. 25-34, 1992.

[10] Eric Werner, "Cooperating Agents:A Unified Theory of Communication and Social Structure," in Distributed Artificial Intelligence, vol 2, Les Gasser and Michael N. Huhns (eds.), Pitman, London, pp. 3-36, 1989.

장 명 욱

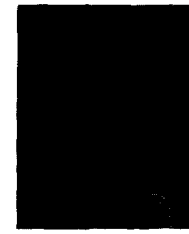
1990년 고려대학교 전산과학과 졸업(학사)
1992년 한국과학기술원 전산학과 졸업(석사)
1992년~현재 한국전자통신연구소 인공지능연구실 연구원
관심분야:에이전트 시스템, 패턴 인식

최 중 민

1984년 서울대학교 컴퓨터공학과 졸업(학사)
1986년 서울대학교 컴퓨터공학과 졸업(석사)
1993년 State University of New York at Buffalo, Computer Science 졸업(박사)
1993년~1995년 한국전자통신연구소 인공지능연구실 선임연구원
1995년~현재 한양대학교 전자계산학과 조교수
관심분야:인공지능, 에이전트 시스템, 지식 표현 및 추론

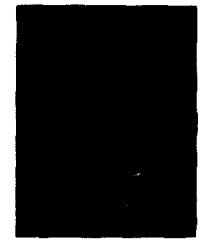
박 상 규

1982년 서울대학교 컴퓨터공학과 졸업(학사)
1984년 한국과학기술원 전산학과 졸업(석사)
1984년~1987년 대림산업 전산실 근무
1989년~현재 한국과학기술원 전산학과 박사과정
1987년~현재 한국전자통신연구소 인공지능연구실 선임연구원
관심분야:에이전트 시스템, 정보 검색, HCI



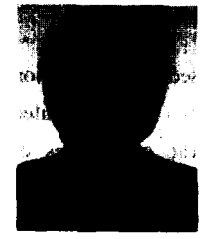
백 순 철

1986년 연세대학교 전자공학과 졸업(학사)
1988년 연세대학교 전자공학과 졸업(석사)
1988년~현재 한국전자통신연구소 인공지능연구실 선임연구원
1993년~1994년 미국 SRI International (International Fellow)
관심분야:에이전트 시스템, 전문가 시스템



이 광 로

1986년 Fukuoka 공업대학 전자기계공학과 졸업(학사)
1988년 Ritsumeikan 대학원 전기공학과 졸업(석사)
1988년~현재 한국전자통신연구소 인공지능연구실 선임연구원
1994년~1995년 미국 SRI International (International Fellow)
관심분야:에이전트 시스템, 멀티미디어, 신경망



임 영 환

1977년 경북대학교 수학과 졸업(학사)
1979년 한국과학기술원 전산학과 졸업(석사)
1985년 Northwestern 대학 전산학과 졸업(박사)
1979년~1982년 한국전자기술연구소 선임연구원
1983년~1985년 Argonne Lab. 연구원
1985년~1996년 한국전자통신연구소 멀티미디어연

구부 책임연구원
1993년~1994년 미국 SRI International (International Fellow)
1996년~현재 숭실대학교 컴퓨터학부 교수
관심분야: 멀티미디어, 에이전트 시스템