摘 要

「多個代理程式系統發展方法」對於分散式系統開發工程提供顯著的優點，例如：提高「互通性」、「擴充性」和「可重新組態性」。以代理程式為基礎的解決方案是實用且吸引人的，因為方案中的各種代理程式知道如何做許多事情。例如：代理程式知道如何與其它代理程式溝通，這使得系統發展者不再需要設計通訊協定和訊息傳遞的格式。這類的能力都被當作是基本的代理程式機制而提供。本計畫第一年的工作主要是分析現有的 Belief-Desire-Intention(BDI) 智慧型代理程式平台，並研究如何擴充現有的 BDI 代理程式系統，加入行動能力、名稱解析、代理程式之間的通訊、協調、合作等機制，訂定一套「擁有多個代理程式環境能力」的新 BDI 代理程式架構之規格並加以實作。另一方面，分析現有的開發多個代理程式的方法論，設計一套適合上述新 BDI 代理程式平台的開發多個代理程式系統的方法論。

關鍵字：行動代理程式、智慧代理程式、代理程式發展平台、代理程式導向

Abstract

The multi-agent approach promises significant benefits to programming of distributed systems, such as enhanced interoperability, scalability, and reconfigurability. An agent-based solution is useful and attractive because the agents used in the solution inherently owns several capabilities. For example, agents can communicate with other agents. The system developers need not design communication protocols and message formats, because they are the agent's basic mechanisms. In the first year of the project, we have designed and implemented a new mobile BDI agent architecture and platform providing the essential capabilities, such as mobility, directory service, and inter-agent communication, collaboration, and coordination for multi-agent systems. Furthermore, we also proposed a multi-agent development methodology suited for the new BDI agent architecture and platform.

Keywords: mobile agent, intelligent agent, agent development platform, agent-oriented

1. Introduction

The multi-agent approach promises significant benefits to programming of distributed systems engineering, such as enhanced interoperability, scalability, and reconfigurability. An agent-based solution is useful and attractive because the agents used in the solution inherently owns several capabilities. For example, agents can communicate with other agents. The system developers need not design communication protocols and message formats, because they are the agent's basic mechanisms. Agents have the inherent capability to build the model of their environment, monitor the state of that environment, reason and make decisions based on that state. All the software developer needs to do is to specify what the agents do systematically.

Mobile agent technology has received a great deal of interest over the past few years in both academia and industry. Most models of mobile agent systems are mainly based on the concepts of agents, agent servers (also called agent systems or agent platforms) and places. According to MASIF (Mobile Agent System Interoperability Facility) standard proposed by OMG (Object Management Group), a mobile agent is a software module responsible for executing some tasks that can autonomously migrate from place to place in a heterogeneous network. The state and code of an agent are transferred to the new place when the agent migration occurs. An agent server provides the execution environments, called places, for the safe execution of local agents as well as visiting agents. The server provides various functions, such as agent transport, security, communications of agents with the host, other agents, and their owners, and fault tolerance.

In this report, session 2 discusses naming, location tracking, and inter-agent communication issues in mobile agent systems. Session 3 presents our new mobile BDI agent and agent server architectures with mobility, directory service, and inter-agent communication support. Session 4

introduces our methodology for the development of multi-agent systems based on our platform. In the final session, we draw a conclusion and suggest the future work**.**

2. Design Issues for Supporting Mobility

2.1. Mobility

A mobile agent consists of code, state, and attributes. Mobile agent's code is a program that defines agent's behavior. Besides code, the state of the agent contains all contents and values of the agent's runtime state and object state. The state contains all the information required to resume execution at the suspended point after migration. The third part of a mobile agent consists of attributes that describe the agent, its requirements, and its history for the infrastructure. They include data such as a unique agent identifier, the agent's owner, error messages, and movement history.

An agent can request its host server to transport it to a remote destination. After receiving the commands, the agent server deactivates the agent, captures its state, and transmits it to the server at the remote host. The destination server then restores the agent state and reactive it at the remote host, thus completing the migrating.

In order to let code be executed on heterogeneous machines, there are many languages used for implementing agent system before Java was shown in the world [1]. For example, Agent Tcl [3] and Ara (Agent for remote access) [4] are based on the Tool Command Language, and Telescript [5] is from General Magic [10] Inc. Java is an appropriate choice of languages for agent systems, because it has some features not found in other languages for supporting mobile agent systems. For example, by using object serialization in Java, objects can be easily "serialized" and sent over the network or written to disk for persistent storage.

2.2. Directory Service: Naming scheme and Name Service

A naming scheme and a name service are needed for addressing and locating various entities in a mobile agent system, such as agent servers, agents, and other global resources. A naming scheme is location transparent [12] if the agent name does not contain any site-specific information. For example, a name comprising the site to which the agent belongs plus an agent identifier (e.g. dssl.csie.nctu.edu.tw/MyAgent) is not location-transparent. On the contrary, a naming scheme according to an agent's properties or functions may be location transparent. An example is the name "MySearchAgent". Furthermore, a naming scheme is location independent if an agent name does not change after being created and might be used to identify and reach the agent independently of its current location. Note that "location-independent" property does not indicate that an agent name cannot contain any site-specific information. For example, an agent name might contain the name of its creator server to record the current location of the agent. Once an agent migrates, its current location record is updated correspondingly. Thus, by contacting the creator server, it is possible to locate the agent. This scheme is location independent but not transparent, as it requires the name presence of the creator server to form an agent name.

Location-dependent naming schemes may allow simpler implementation of name service systems than location-independent ones. Platforms like Agent Tcl, Aglets, and Concordia use location-dependent scheme to name agents. In these systems a mobile agent is named based on the host name, port number, and an identifier. Name resolution is based on DNS. When an agent migrates, its name would change. However, the location-dependent naming schemes make the implementation of agent location tracking cumbersome. On the contrary, a location-independent naming scheme requires a name service to map the symbolic name to the agent's current location. A simpler solution is to use a unique name server to keep track of all agents. However, this is only suitable for small scale systems and lacks of scalability. Another approach taken by Voyager design uses proxies object. A remote mobile agent is located by contacting its creator server to obtain its local references called proxies. Such proxies are updated by the runtime environment when the agent migrates. However, this method creates a strong binding between application level names and network level names and raises the issue of performance if there are a lot of proxies for an agent in the network.

Because entities such as agents are mobile in the network, it is desirable to allow accessing them in a location independent manner.

2.3. Inter-agent Communication

The design challenges for inter-agent communication mechanisms arise due to the mobility of the agents. There are several design choices: connection-oriented communication (such as TCP/IP), connection-less communication (RMI, RPC, or CORBA-IIOP), or indirect communication based on the event publisher/subscriber model and shared mailboxes or meeting objects. In TCP/IP or RMI based communication, agents need to know each other's network address in order to establish communication. Connection-oriented schemes raise the issue of connection disruption due to a participating agent's migration. In other words, connection-oriented communication can be location-independent as long as the agents do not move during communications. Otherwise a new connection for communications has to be established. In comparison, RMI based request-reply model throws an exception when a remote invocation fails due to the migration of the server agent; the client agent only need to re-execute the lookup and binding protocol to re-establish communication with the migrated agent at its new location. However, it may become hard to establish communication if the invoked agent moves very frequently. The indirect communication model using stationary objects to hold events/messages/tuples is more appropriate for such cases. The tuple-space mode is suitable for agent coordination, but not applicable for bulk data exchange.

Several systems (such as Aglets, Grasshopper, and Voyager) have supported synchronous, asynchronous with a reply, and asynchronous without a reply (or one-way) communication models. When an agent sends a synchronous message, its thread blocks until the receiver replies to the message. When sending an asynchronous message, the agent does not block and a return handler is used to get the reply via waiting, polling, or call back. The last type of message is one-way message, i.e. asynchronous without a reply; it is useful when a message sending agent that does not expect reply from message receiving agent.

3. Mobile BDI Agent Architecture & Platform

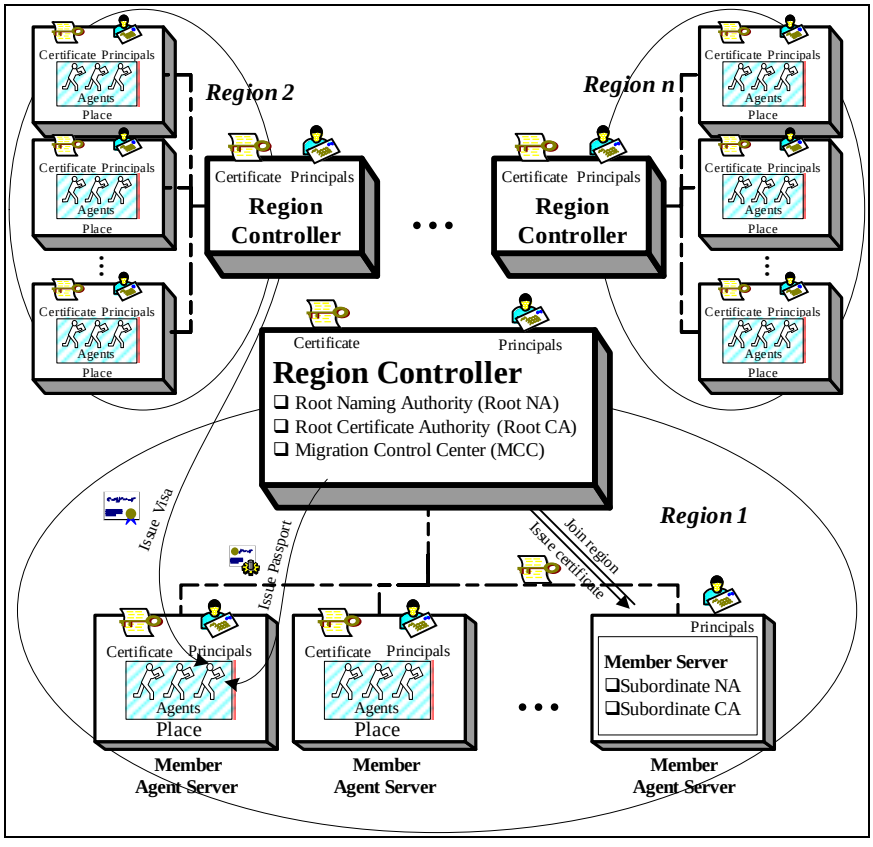

Figure 1. MBDI Platform

3.1. System Architecture Overview

As in Figure 1, MBDI Agent platform contains six major types of entities: *regions*, *agent servers*, *places*, *agents*, *resources*, and *principals*. A region is a logical entity grouping a set of agent servers that have the same authority, but not necessary of the same agent system type. The purpose of regions is to provide central management over all agent servers having the same authority. Agent servers provide the environment called place for safe execution of local agents as well as visiting agents. A place is the context in which an agent can execute. The place provides a uniform set of services that the agent can rely on irrespective of its specific location. An agent is an autonomous entity that performs a set of operations on behalf of a personal or an organization (agent authority) to achieve the user's goal. An agent can provide services, perform task, achieve goals, and utilize resources. Unlike agents, resources are non-autonomous entities provided by agent servers or external programs. Regions, agent servers, places, agents, and resources are associated with principals. A principal is the person or organization whose identity can be authenticated by any system the principal may try to access. In MBDI, there are two main kinds of principals: *Manufacturer* and *Owner*. Manufacturer is the author providing the implementation of the entity. The owner of an entity is the person or organization who creates the entity. The details for main entities are described as following sessions.

3.2. Region

In MBDI, a region defines a set of agent servers that belong to the same authority. The agent server creating a region becomes the *Region's Controller* (RC). By default, each agent server creates a region consisting of itself merely and then becomes the region's controller until it configures to join another region. In a region, the region controller is the top-level (root) directory server and each member agent servers is registered as a lower-level (subordinate) directory server. Within the region, the region controller acts as:

- top-level naming authority: assigning name of entities within region scope.
- top-level name server: entities registration/deregisteration/lookup.
- root Certificate Authority that issuing certificates to member agent servers
- Migration Control Center (MCC) controlling mobile agent between regions.
  - ✓ Passport issuing and management for agent created within the region.
  - ✓ Visa issuing and management for foreign agent from other regions.

A member Agent Servers acts as:

- second-level naming authority: assigning name of the entities created on the server.
- second-level certificate authority: issuing certificates to the agents and users created on the server.
- second-level name server: entities registration/deregisteration/lookup

3.2.1. Naming scheme

To comply with the location independent naming requirement of mobile objects, we propose our MBDI name (MBDIN) scheme. The MBDI naming scheme is as follows:

Region: MBDI://RegionName[:PortNumber]
Agent Server: MBDI://RegionName[:PortNumber]/AgentSystemName
Place: MBDI://RegionName[:PortNumber]/AgentSystemName[:PlaceName]
Agent: MBDI://RegionName[:PortNumber]/AgentSystemName/LocalAgentName
Resource:MBDI://RegionName[:PortNumber]/AgentSystemName/ResourceName
Principals: MBDI://UserName@RegionName[:PortNumber]
MBDI://RegionName[:PortNumber]/UserName@AgentSystemName
MBDI://RegionName[:PortNumber]/AgentSystemName/UserName@LocalAgentName

We use a hierarchical naming where sub name spaces are separated by a slash, '/'. Name of entities are assigned by naming authorities that are responsible for a subtree of the name space and naming responsibility of sub name spaces is delegated to sub-authorities. Each naming authority corresponds to a name resolution service. The following takes Agent name as an example for explanation. The leading "`MBDI`" indicates that the name conforms to the MBDI naming scheme. `RegionName` represents the *home region* (or *creation region*) where the agent was created and is named as the Full Qualified Domain Name (FQDN) of the corresponding region controller. By doing so, we can use the existing resolution framework of the Domain Name System (DNS) to locate the *home region* that is used as a hint to locate an entity. PortNumber is the optional field used to specify port number where the region controller provides the name service. Default port number is 7000. `AgentSystemName` specifies the name of the agent server where the agent is created. `PlaceName` is optional name of the place and if omitted, it indicates the default place. `LocalAgentName` is the local name of the agent chosen by generator or programmer. The string expressing the agent's characteristics can be used for agent's `LocalAgentName` for better readability. The following is a sample agent name: MBDI://rc1.csie.nctu.edu.tw:8000/MyAgentServer:sales/MySellerAgent.

Our naming scheme contains three characteristics:

1. Adopt hierarchical naming that provides easy maintenance and delegation of namespaces.
2. Provide location independent naming for mobile object.
3. Use existing name resolution infrastructures (DNS).

3.2.2. Directory Service (Name Service)

The top-level naming authority of a region is its region controller and each agent server is the subordinate naming authority for entities created on the server. Each naming authority provides directory services including white pages services for locating and providing information about the entities created within the name space of the authority and yellow pages services for searching services provided by these entities. Directory service maintains the mapping between the MBDIN of an entity and its characteristics. Depending on the type of the entity, different attributes are stored in the directory database. The common attributes for all types of entities include the entity's string name and public key for unique identification, current location for locating the entity, and an access control list (ACL) for enforcing security policies. Furthermore, each entry keeps some status information, such as "last modification time" and "last access time". An entity with a name needs to register itself with a directory service and then may update its entry when necessary. For example, when a mobile agent migrates to another agent server, it updates its current location.

3.2.3. Certificates Management

Each entity is associated with a pair of keys—public key and private key for data encryption/decryption, and digital signatures (for authentication and data integrity). Each region controller acts as a root Certification Authority (CA) within the region and is responsible for issuing digital certificates to agent servers when they join the region. Each member agent server is registered as lower-level (subordinate) CA that issues certification to agents created on the server. A certificate contains the holder's name, a serial number, expiration dates, the holder's public key (used for encrypting messages and digital signatures), and the digital signature of the certificate-issuing authority so that a recipient can verify that the certificate is real.

A region controller may accept or reject the join request from other agent servers based on its security policies. If the region controller accepts the request, a certificate is issued to the request server. The certificate signed by the region controller contains the server's public key and a variety of other identification information. The region controller will distribute the certificates of all member agent servers so that every member server knows the certificate of all the others. The certificate is revoked when the agent server disjoins from the region or when its content is subject

to change or suspected of being tampered with before it is expired. The region controller will push the revocation information to all the other member agent servers.

3.3. Agent Server and Place

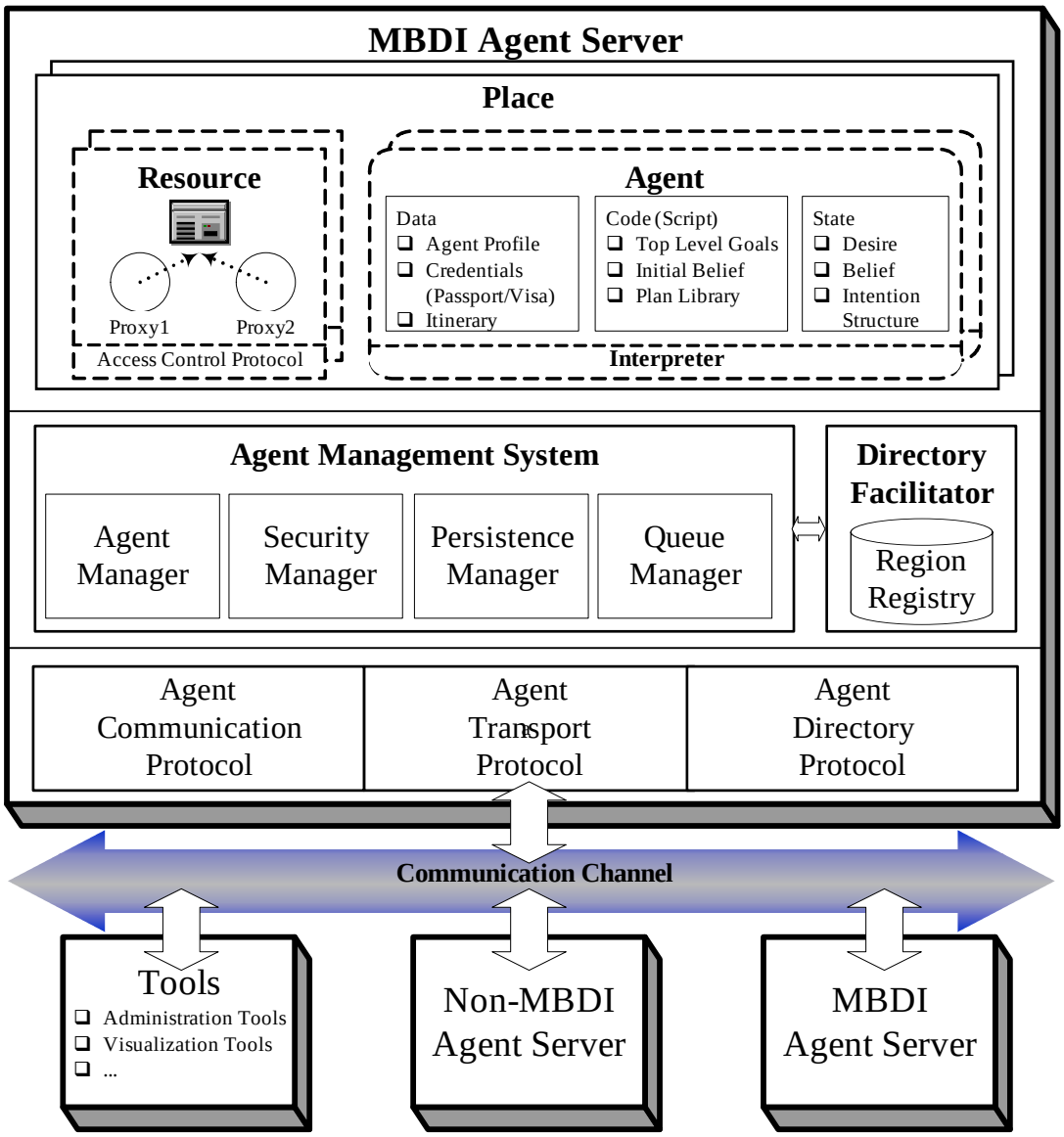

Figure 2. MBDI Server Structure

Our Mobile BDI Agent Platform, named MBDI Server, provides the environment called *place* for the safe execution of local agents as well as visiting agents. A place is the context in which an agent can execute. The place provides a uniform set of services that the agent can rely on irrespective of its specific location. The MBDI Server provides various functions, such as agent management, transport, security, communications of agents with the host, other agents, and their owners, and fault tolerance. The MBDI Server accepts incoming agents, authenticates the identity of the owner, and passes the authenticated agent to the execution environment. The MBDI Server also keeps track of the agents running on its machines and answers queries about their status. Furthermore, the MBDI Server allows an authorized user to suspend, resume and terminate a running agent, and allows agents to communicate with each other through message passing and direct connections. As in Figure 2, MBDI Server consists of the following components:

- **Agent Manager (AM)** is responsible for providing fundamental agent management operations. The operations include the creation, suspending, resuming, transferring, receiving, termination, and getting status of agents.

- **Directory Manager (DM)** provides access to directory service. DM maintains two lists of agents, one is for the agents that are created locally and the other is for visitor agents. DM provides white pages (address books) services to agents. Each newborn agent created locally must register with local DM in order to get a valid and unique name and the DM is also in charge of keeping the location of agents created locally. An agent may register with the DM to announce its address or query DM to find out current locations of other agents created locally. When the agent migrates to another MBDI Server, the agent informs the DM to update its location information. DM also provides yellow pages services to agents. An agent may register with the DM to announce its capabilities or query the DM to find out what capabilities are offered by other agents. For an MBDI Server joining a region, the global directory service is described in latter session.
- **Security Manager (SM)** is responsible for protecting hosts and agents from malicious entities. SM provides mechanisms for identifying users, authenticating and authorizing their agents, and data encryption.
- **Persistent Manager (PM)** is required to ensure that the agents can recover from the MBDI Server crash successfully. It maintains the state of agents; thus, it allows for the checkpoint and restart of agents in case of failure of MBDI Server. Additionally, it can snapshot the state of objects upon request by agents, to provide better reliability guarantees for critical procedures.
- **Queue Manager (QM)** is responsible for the scheduling and guaranteed delivery of mobile agents between MBDI Server.
- **Message Manager (MM)** is responsible for managing incoming/outgoing messages and events. The MM has two queues, one for incoming messages/events, and the other for outgoing messages/events. For incoming messages, the MM forwards these messages one at a time to the corresponding object. It ensures that the next message is not forwarded until the current message has been handled. For an outgoing message, it will be kept in the outgoing queue until it has been delivered successfully or failurely in a fixed time.

3.4. Mobile BDI Agent Architecture – MBDI Agent

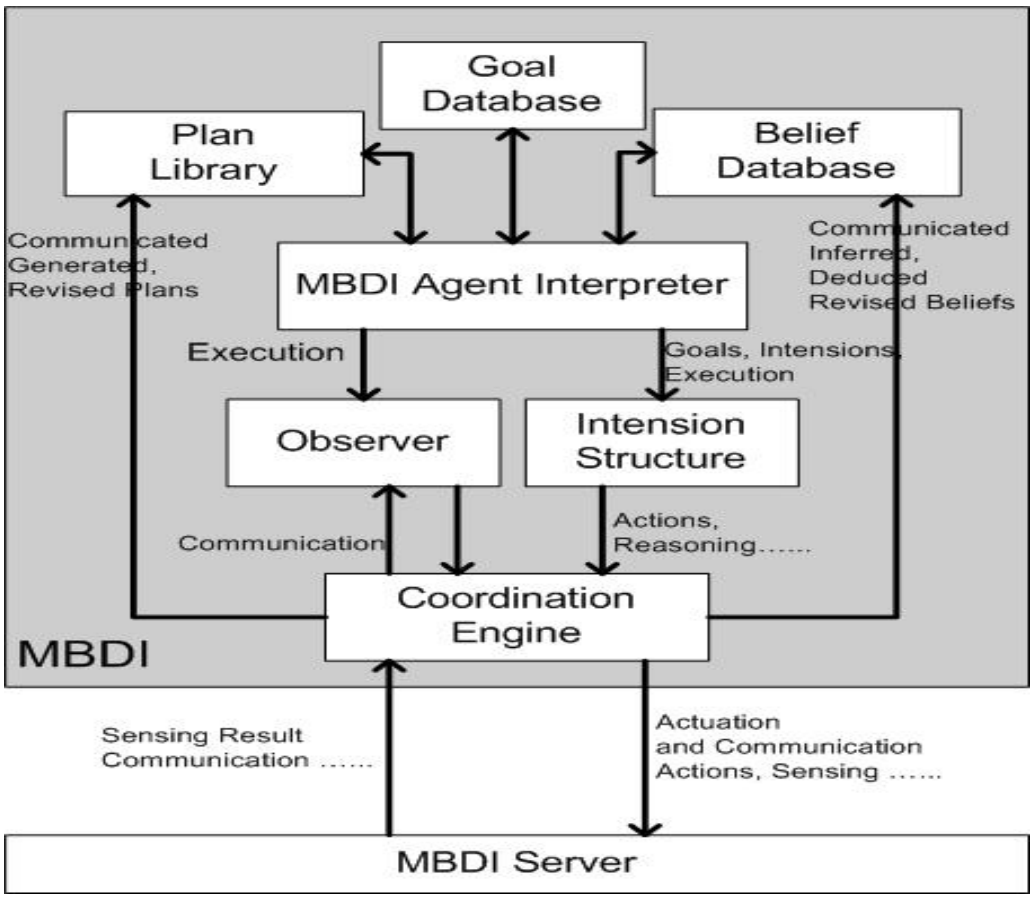

Figure 3: MBDI agent architecture

The intelligent agent architecture we have proposed is based on BDI-theories [16]. A generic agent is shown in Figure 3. The generic agent includes the following components:

- **Belief Database** represents what the agent knows about itself and the world. For example, the belief may contain information that describes the agent's relationships with other agents and the capabilities of those agents.
- **Goals** specify what the agent is trying to achieve.
- **Plan Library** defines the sequences of actions and tests to be performed to achieve a certain goal or react to a specific situation.
- **Intention Structure** contains those plans that have been chosen for eventual execution.
- **Interpreter (Reasoner)** selects appropriate plans based on agent's current belief, goals, and intentions. Then the interpreter places the selected plans on the intention structure, and executes them.
- **Co-ordination Engine (CE)** is a lightweight plan that the agent executes between plan steps. CE is responsible for processing the incoming/outgoing messages and events that coordinating the agent's interactions with agent servers and other agents by using the information stored in the belief database. The CE manages the agent's problem solving behavior, especially for those involving multi-agent collaboration, i.e. team plans. It provides several predefined co-ordination protocols, such as master-slave for delegating tasks to subordinate agents, contract-net for contracting tasks out to peer agents, and various auction protocols for buying or selling resources. The CE also provides a number of methods that enable to customize the behavior of the CE.

3.4.1. The Mechanism to Track an Agent in a Region

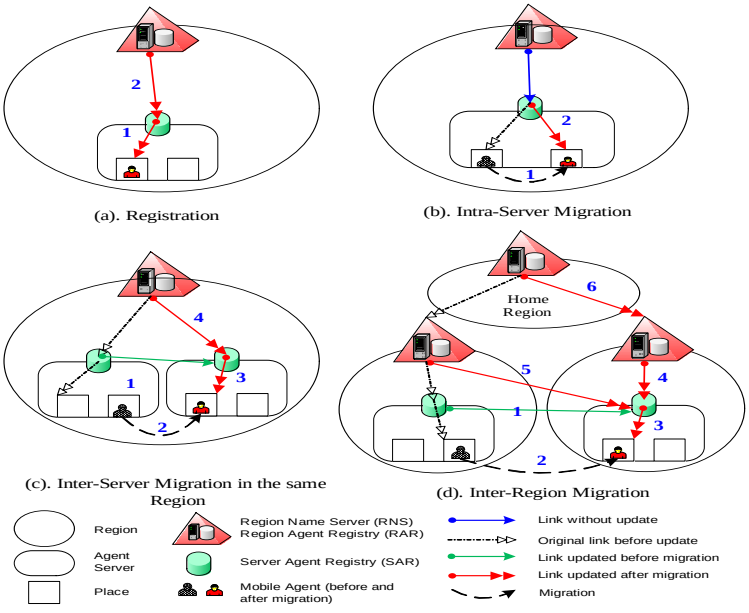

Figure 4: Location management

In each Region, the region controller acts as Region Name Servers (RNS) for that Region to provide global naming service. Each RNS contains a Region Agent Registry (RAR), the entry in which provides the location where the target agent is to be found. Similarly, each agent server's Directory Manager (see session 3.2) contains a Server Agent Registry (SAR) used to store the place name where the target agent is to be found. Both RAR and DAR contain information about all the agents that have been created in their scope or have transited through their scope.

An entry of RAR is in the form of {AgentName, CurrentLocation, IsHome, IsMigrating}. AgentName is our URN name of an agent. IsHome indicates that whether the agent is created in this region or just a visitor transiting through this region. Moreover, IsMigrating is used to

indicate whether the agent have started to migrate or not. If IsMigrating is false, the CurrentLocation stores URN name of the region or agent server where the agent stays currently; otherwise, it indicates the target region or agent server to where the agent have started to migrate. Similarly, an entry of SAR is also in the form of {AgentName, CurrentLocation, IsHome, IsMigrating}. The only difference is that the CurrentLocation of SAR stores the URN name of place where the agent is to be found.

The management of agents' location information in our system could be described in four separate phases: *registration*, *migration*, *getLocation*, and *deregistration* phase.

- *Registration(Deregistration)*
  When an agent is created or arrives at an agent server, the agent will perform a registration operation to declare its existence. The agent registers its birth in the former case (birth registration) while it registers as a visitor in the latter case (visitor registration). Both cases use the same procedure. The difference is the information used to register: the field IsHome is set to True in the former case while it is set to False in the latter. The registration operation is as follows. First, the agent registers its name and current place name in the SAR of current agent server and then registers its name and current agent server name in the RAR of current region. Note that for birth registration, the SAR will check the agent name to guarantee the requirement of name uniqueness (see Figure 4(a)).
- *Migration*
  The operations carried out during this phase depend on whether the source place and destination place belong to the same agent server, different agent servers but the same region, or different agent servers and regions. For intra-server (the same agent server) migration, only the SAR is updated with the new place name (Figure 4(b)). For inter-server migration in the same region, before migration, the agent updates the SAR with the URN name of destination agent server. After migration, the agent performs a visitor registration (Figure 4(c)). For inter-region migration, firstly, the agent updates the SAR with the URN name of destination agent server. After successful migration, the agent performs a visitor registration. Finally, the agent updates the RAR of source region with the URN name of destination agent server and the RAR of the agent's home region with the URN name of destination region (Figure 4(d)).
- *getLocation*
  The *getLocation* procedure follows the links that the agent has left in the agent registry on the regions or agent servers it has visited.
  1. The name of agent's home region is extracted from its URN name and the existing resolution framework of the Domain Name System (DNS) for URN resolution is used to find the relative RNS of the agent's home region.
  2. The RNS is contacted and the RAR entry for the agent is retrieved. This (Home) RAR entry contains an indication of
     A. The target agent server the agent is on if it is still within its home region.
     B. The target region the agent is on if it has migrated to the outside of its home region.

  In case A, the agent then contacts the target agent server's SAR to find on which place the agent stays, whereas in case B the RNS of target region is contacted and then the similar procedure in case A is repeated until the place on which the agent stays is found.
- *Deregistration*
  When the agent terminates, it informs the home SAR and the home RAR to clear the entry of the agent.

If all the update operations in the migration phase have been successful, the target agent can be found by at most six messages (Request/Reply with home RAR, current RAR, and current SAR).

3.4.2. Communication

In our system, agents can send messages either synchronously or asynchronously, locally or remotely, peer-to-peer or multicast. Messaging is done through the passing of message objects. The message object can contain anything, from a simple data type to a serializable object. The

actual message passage is performed by obtaining a reference to the receiver agent object via the name service and then calling *sendMessage* method with the message object as an argument. A *messenger* component is responsible for reliable message delivery. The receiver collects the messages in a queue managed by the receiver's *messageManager* component. Through the messageManager, priority levels of messages can be set for faster processing for messages of more importance. When the receiver invokes its *handleMessage* method, the message at the head of the queue is dequeued and processed. After the message response has been determined, the receiver invokes one of several *sendReply* methods of the message object. These methods take the reply as an argument and send it automatically back to the original sender; addressing and location are transparent and handled automatically by the name service.

The messaging system supports three types of message modes: (1) synchronous, (2) asynchronous with a reply, and (3) one-way, i.e. asynchronous without a reply. These messaging mechanisms work not only with the agents running in the same place, but between remote agents as well. However, care must be taken to ensure that the sender knows where the intended receiver is and the message is guaranteed to be delivered to the receiver, which might even migrate anytime. It is done in our system through the name service and messenger.

4. Conclusions

In the first two years of the project, we have analyzed the existing BDI agent architectures and mobile agent platform to design and implement a new mobile BDI agent architecture and platform that provides essential capabilities in multi-agent environments, such as mobility, directory service, and inter-agent communication, collaboration, and coordination. Furthermore, we have analyzed existing methodologies for analysis and design of multi-agent systems and have proposed our own methodology suited to the new platform.

The major work of the next year is to: (1) continue accomplishing implementation and testing of the new platform; (2) use our methodology to analysis and design several example applications to verify its practicability; (3) implement the examples on the new platform to verify the platform's practicality; (4) analyze existing development environments for multi-agent systems; (5) design and implement a visual programming environment which supports analysis, design, implementation, and simulation for multi-agent systems.

Reference

[1]"Software Agents: A review", Shaw Green, Leon Hurt etc.

[2] Programming and Deploying Java Mobile Agents with Aglets, Danny B. Lange and Mitsuru Oshima

[3] Agent Tcl was developed by Robert S. Gray and colleagues at the Dartmouth College Computer Science Department.

[4] Ara is a project within the Distributed Systems Group in the Computer Science Department of the University of Kaiserslautern, Germany.
http://www.uni-kl.de/AG-Nehmer/Projekte/Ara/index_e.html

[5] James E. White; Telescript technology: The foundation for the electronic market place; General Magic White Paper.

[6] IBM Aglets, http://www.trl.ibm.co.jp/aglets

[7] Voyager 3.1, http://www.objectspace.com

[8] Concordia, http://www.meitca.com/HSL/Projects/Concordia

[9] MASIF-The OMG Mobile Agent System Interoperability Facility; Mobile Agents–Second International Workshop, MA'98 (Stuttgart, Germany, September 1998); Published as Kurt Rothermel and Fritz Hohl, editors, Lecture Notes in Computer Science, 1477. Springer, September 1998.

[10]Mobile Agents White Paper, General Magic, http://www.genmagic.com/technology/techwhitepaper.html/

[11] Agent system Development Method Based on Agent Pattern; Yasuyuki Tahara, Akihiko Ohsuga and Shinichi Honiden; 21st International Conference on Software Engineering, 16-22 May 1999.
[12] Distributed Systems: concepts and Design; George Coulouris, Jean Dollimore, and Tim Kindberg; second edition 1994
[13] Distributed Operation Systems & Algorithms; Randy Chow and Theodore Johnson at university of Florida; publisher Addison Wesley 1997
[14] HOSTNAME Server; Tech. Report RFC 953; ftp://nic.ddn.mil/user/pub/RFC
[15] Mobile Objects and Agents (MOA); Dejan S., William LaForge and Deepika Chauhan; The Open Group Research Institute.
[16] Marcus J. Huber. JAM: A BDI-theoretic mobile agent architecture. In Proceedings of the Third International Conference on Autonomous Agents (Agents'99), pages 236--243, May 1999.