A LOGIC-BASED MOBILE AGENT FRAMEWORK FOR WEB APPLICATIONS

Shinichi Motomura, Takao Kawamura, Kazunori Sugahara
Tottori University
4–101, Koyama-Minami, Tottori 680–8552, JAPAN
Email: motomura@tottori-u.ac.jp, {kawamura,sugahara}@ike.tottori-u.ac.jp

Keywords: Logic programming, mobile agent, field, XML-RPC.

Abstract: A new logic-based mobile agent framework named Maglog is proposed in this paper. In Maglog, a concept called "field" is introduced. By using this concept, the following functions are realized: 1) the agent migration which is the function that enables agents to migrate between computers, 2) the inter-agent communication which is the indirect communication with other agents through the field, 3) the adaptation which is the function that enables agents to execute programs stored in the field. We have implemented Maglog on Java environment. The program of an agent which is a set of Prolog clauses is translated into Java source code with our Maglog translator, and then it is compiled into Java classes with a Java compiler. In addition, through XML-RPC interface for Maglog which we have also implemented, other systems can easily utilize Maglog. The effectiveness of Maglog is confirmed through the demonstration of an application: the distributed e-Lerning system.

1 INTRODUCTION

Techniques for developing Web applications show an advance recently and dynamic contents provided by programs can be inserted in them. These programs have an inclination to become complicated and some of them are having communication facility with another computers. Service-oriented architecture (SOA) attracts attention as a key technology for this requirement. SOA is an architectural style whose goal is to achieve loose coupling among interacting software agents.

On the other hand, mobile agent systems are discussed for developing distributed applications. It is meaningful to develop Web applications based on mobile agent systems. For realization of the mobile agent systems, the following functions are required to be implemented.

1. Agents should be able to migrate from one computer to another with data and programs.
2. Agents should be able to communicate with other agents.
3. Agents should be able to adapt themselves to environments such as computers they belong to. The adaptation is accomplished by taking data and programs of the environments into themselves.

Considering the above points, a concept of the "field" is proposed to realize the required functions simply.

Agents communicate with other agents indirectly through the field and adapt themselves to the environment by importing data and programs stored in the field. The functions realized by the field are summarized as follows.

1. Migration: Function that enables agents to migrate between computers.
2. Inter-agent communication: Indirect communication with other agents through the field. That is, an agent is able to import data or programs that other agents store in the field.
3. Adaptation: Function that enables agents to execute programs stored in the field.

For the implementation of the mobile agent system with the concept of the field, programs that describe behavior of the agent are written in Prolog language in our system. Since Prolog is logic programming language and has powerful pattern matching mechanism, agents are able to search data and programs stored in fields easily. This powerful pattern matching mechanism of Prolog is called "unification". Unifications between computers are realized to construct mobile

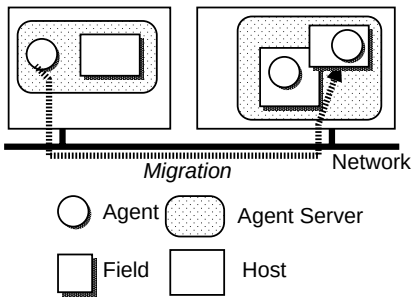

Figure 1: Overview of a mobile agent system on Maglog.

agent system.

In this paper, the mobile agent framework named Maglog on Java environment is proposed to implement the above-mentioned functions. Java is adapted because of its huge class libraries to build network applications. It also should be noted that Java's goal of "write once, run anywhere" is desirable for mobile agent systems.

There are several mobile agent frameworks realized as a set of class libraries for Java such as Aglets(Lange and Oshima, 1998), MobileSpaces(Satoh, 2000), and Bee-gent(Kawamura et al., 2000). The combinations of one of them and a Prolog interpreter written in Java such as NetProlog(de Carvalho et al., 1999) and Jinni(Tarau, 1999) have some similarity to Maglog. The main difference between the combinations and Maglog is the class of mobility. Their mobilities are weak mobility, in which only their clause databases are migrated. On Maglog, all of the execution state including execution stack can be migrated. That is to say, the mobility of Maglog is strong mobility so that agents on Maglog can backtrack and unify variables across the network. That makes programs on Maglog simple and understandable.

2 OVERVIEW OF MAGLOG

Figure 1 shows an overview of a mobile agent system on Maglog executing an example. In the figure, two computers (hereafter referred as hosts) are connected to a network and agent servers are running on each of them to activate agents and to provide fields for them.

The rest of this section describes three basic components of Maglog, that is, agent, agent server and field.

2.1 Agent

An agent has the following functions.

1. Execution of a program that describes behavior of the agent,
2. Execution of procedures stored in a field where the agent currently locates,
3. Communication with other agents through a field,
4. Creation of agents and fields,
5. Migration to another host in a network,
6. Cloning of itself.

An agent of Maglog executes its program sequentially. The class of agent migration is strong migration which involves the transparent migration of agent's execution state as well as its program and data. In order to realize unifications between computers, Maglog supports strong mobility.

For creation of a child agent, a parent agent executes the following built-in predicate.

```
create(AgentID,File,Goal)
```

In this predicate, `File` corresponds to the filename in which the behavior of the agent is described. If the execution of the predicate is successful, an agent is created and its globally unique identifier `AgentID` is returned. The created agent immediately executes the goal specified by the argument `Goal` and disappears when the execution is accomplished.

An agent can obtain its identifier by executing the following built-in predicate.

```
get_id(Agent)
```

An agent can create its clone agent by executing the following built-in predicate. If `CloneAgentID` is the empty set, it is original agent. In the case of the cloned agent, `CloneAgentID` is the cloned agent identifier.

```
fork(CloneAgentID)
```

Each agent contains Prolog program and its interpreter. Initial behavior of the agent is described in the Prolog program given by `File` in the predicate of its creation. Since Prolog language treats programs and data identically, the agent behavior might be modified during execution.

Figure 2 shows an example of an agent behavior. agentA is assumed to have a clause `in((clause(p(x),Y),assert(p(X):-Y)), fieldA)` in its program. Behavior of agentA is described as follows,

1. agentA enters fieldA.
2. agentA executes a predicate `clause(p(X),Y)` and retrieves a clause whose head matches `p(X)` from fieldA as a result. Here `Y` is bounded to `q(X),r(X)` which is the body of the clause.
3. agentA executes a predicate `assert(p(X):-Y)`, and then a clause `p(X):-q(X),r(X)` is added to its own program.

That is to say, an agent is able to import clauses from fields so that it can change its behavior dynamically.

The built-in predicate `in/2` will be described in Section 3.1. Here the notation `Name/Arity` is the predicate indicator (hereafter referred as PredSpec) which refers one or several predicates. `Name` and `Arity` correspond to the name of predicate and its number of argument respectively.

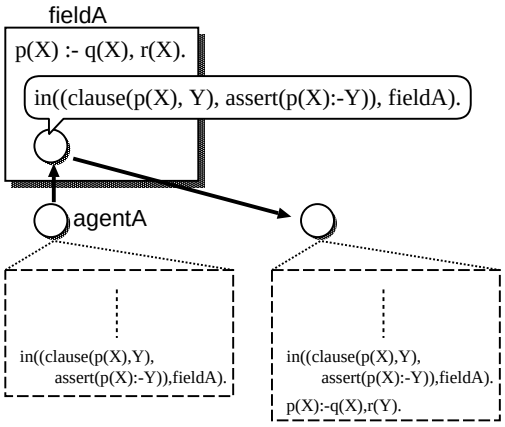

Figure 2: Dynamic change of program that describes behavior of the agent by asserting a new clause.

2.2 AgentServer

An agent server is a runtime environment for agents and it provides required functions for agents. The above-mentioned predicates, such as `create/3` and `get_id/1`, are the examples of the functions.

An agent server creates and deletes agents. An agent server assigns `AgentID` to the created agent. `AgentID` consists of host's IP address and the time the agent created, so that it becomes globally unique. In addition, an agent server also provides an agent migration function. When an agent migrates from hostA to hostB, the agent server on hostA suspends the agent's execution and transports the agent to hostB. And after that the agent server on hostB resumes the execution of the agent.

An agent server also manages fields and provides functions for an agent to utilize them.

2.3 Field

A field is an object managed by an agent server to hold Prolog clauses, and it is created when an agent executes the following built-in predicate.

```
fcreate(Field)
```

If `Field` is an unbound variable, a field which has a unique identifier is created, and its identifier is bound to the argument `Field`. If `Field` is a symbol, the action of this predicate depends on whether the field whose identifier is the symbol exists or not. If it does not exist, a field whose identifier is the symbol is created, otherwise nothing is done.

Important features of Maglog realized with the concept of the field will be described in the following section.

3 FEATURES REALIZED WITH FIELD

3.1 Predicate Library

An agent enters a field and executes a goal by the following built-in predicate.

```
in(Goal, Field)
```

The agent exits from `Field` automatically whether the execution succeeds or not. This built-in predicate is re-executable, i.e. each time it is executed, it attempts to enter the field and executes the next clause that matches with `Goal`. When there is no more clause to execute, this predicate fails.

When an agent enters into a field, it imports procedures of the field and combines them with procedures of itself. Therefore, an agent needs not hold all of the program by itself to solve a problem, but rather enters the appropriate fields which provide necessary procedures. An agent can change its behavior dynamically according to the field which it entered. In this way, an agent can adapt its behavior to the environments.

3.2 Inter-agent Communication

Agents entering the same field can be considered of forming a group. The procedures within the field are shared by the agents. Moreover, by adding/removing procedures within the field, agents can influence the behavior of other agents.

Updating procedures in a field can be done by the following built-in predicates.

```
fasserta(Clause, Field)
fassertz(Clause, Field)
fretract(Clause, Field)
```

The first argument `Clause` of these predicates is a clause to be added or removed from the field specified by the second argument `Field`. `fasserta/2` inserts the clause in front of all the other clauses with the same functor and arity. Functor and arity mean the name of predicate and its number of the argument respectively. On the other hand, `fassertz/2` adds the clause after all the other clauses with the same functor and arity. `fretract/2` removes the next

unifiable clause that matches with the argument from the field. This built-in predicate is re-executable, i.e. each time it is executed it attempts to remove the next clause that matches with its argument. If there is no more clause to remove, then this predicate fails.

By using these predicates, an agent can communicate with other agents not only asynchronously but also synchronously. An agent has two modes for execution of procedures stored in a field. In the fail mode, the execution fails when an agent attempts to execute or to retract a non-existent clause in a field. In the block mode, an agent that attempts to execute or to retract a non-existent clause in a field is blocked until another agent adds the target clause to the field. For agents in the block mode, a field can be used as a synchronous communication mechanism such as a tuple space in Linda model(Carriero and Gelernter, 1989)

Figure 3 shows an example of the synchronous inter-agent communication.

1. `PARENT` creates `fieldA`.
2. `PARENT` creates `CHILD` and makes it execute `main('fieldA')`. `PARENT` attempts to remove the clause that matches `ans(ID,X)` from `fieldA` and it is blocked until a unifiable clause is added by `CHILD`.
3. `CHILD` executes `calculate(X)` and the result is bound to `X`. The identifier of `CHILD` is bound to `ID` by the execution of the built-in predicate `get_id(ID)`. `CHILD` adds `ans(ID,X)` to `fieldA`.
4. `PARENT` wakes up and removes `ans(ID,X)` from `fieldA`.

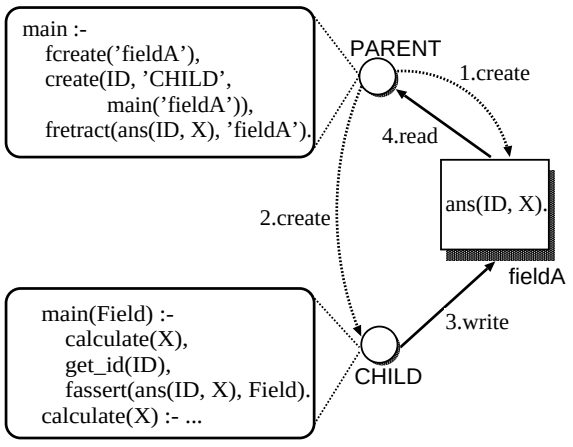

Figure 3: Agents can communicate synchronously through a field.

3.3 Agent Migration

Each agent server has globally unique identifier that is composed of the server IP address and defined name.

If the second argument of the predicates `in/2`, `fasserta/2`, `fassertz/2`, and `fretract/2` is specified in the form of `Field@ServerID`, the agent executing this predicate migrates to the host in which the agent server specified by `ServerID` runs, and enters `Field`. The agent returns to the host located before the migration automatically as it exits the field.

Figure 4 shows that the agent matches `f(X)` with clauses in two fields in hostA and hostB. As shown in Fig. 4, this attempt proceeds through performing the following steps and succeeds.

1. An agent enters fieldA in hostA and executes the goal `f(X)`. Consequently `X` is bound to 3, because `f(3)` is the first clause that matches with `f(X)`.
2. The agent migrates to hostB and enters fieldB.
3. The agent executes the goal `f(3)`. This attempt fails since there is no clause that matches with `f(3)`.
4. The agent returns to hostA and enters fieldA automatically.
5. The agent attempts to execute the next clause that matches with `f(X)`. `X` is therefore bound to 5.
6. The agent migrates to hostB and enters fieldB, again.
7. The agent executes the goal `f(5)`. This attempt succeeds this time since there is the clause `f(5)` in fieldB.

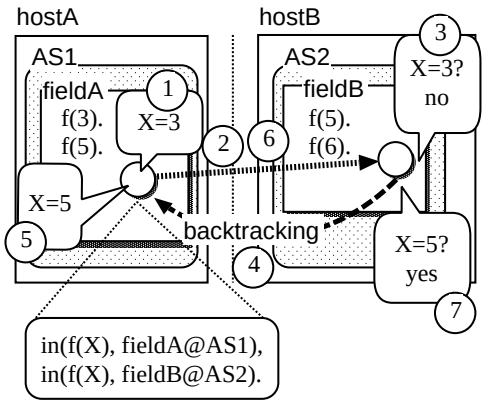

Figure 4: Backtracking and unification between two hosts.

4 IMPLEMENTATION

We have implemented Maglog on Java environment through extending PrologCafé(Banbara and Tamura, 1999) which is a Prolog-to-Java source-to-source translator system.

The program of an agent which is a set of Prolog clauses is translated into Java source code with our Maglog translator, and then it is compiled into Java classes with a Java compiler. As mentioned in Section 2.1, an agent can import Prolog clauses from a field at run time. These clauses are interpreted by the Prolog interpreter included in an agent instead of being compiled into Java classes. An agent runs as a thread in a process named agent server.

Agent servers have an XML-RPC(Winer, 1998) interface which is accessible from applications written in any other language with support for XML-RPC.

The following operations from other systems are available through XML-RPC.

1. create and kill agents,
2. create and delete fields,
3. assert clauses into fields and retract clauses from fields,
4. get a list of names of fields,
5. get a list of IDs of agents currently exist.

Implementation of Maglog features realized with the concept of the field is described in the rest of this section.

4.1 Predicate Library

As shown in Fig. 5, a field is implemented as a Java Hashtable, i.e. procedures in a field are put in a hashtable; a key is PredSpec of a procedure, and the value is a set of objects representing the procedure whose predicate indicator is PredSpec.

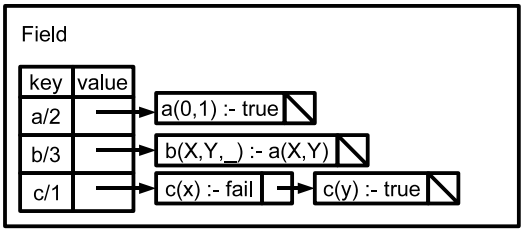

Figure 5: Structure of a field.

When an agent executes a predicate in `in/2`, it searches the predicate by specifying PredSpec from the hashtables of fields it currently enters and interprets the found value.

In order to improve execution rate, the concept of a static field is introduced into Maglog. It stores read-only procedures compiled into Java classes before the agent server to which the field belongs starts.

A static field is implemented as a Java Class Loader which receives PredSpec and loads the bytecodes of the class for the corresponding procedure.

According to the experiments, an agent can execute a clause in a static field about 250 times faster than in an ordinary field.

4.2 Inter-agent Communication

As mentioned in Section 3.2, an agent which attempts to execute or to retract a non-existent clause in a field simply fails in the fail mode, while the agent in the block mode is blocked by calling the Java `wait` method.

When another agent adds one clause to a field, the blocked agents in the field are waked up by the Java `notifyAll` method and try to execute their goal. The agents whose target clause is added restart while the rest of the wake-up agents are blocked again.

4.3 Agent Migration

The migration of an agent is realized by using a Remote Procedure Call (RPC) as the following:

1. The source agent server encodes the agent as the argument of a RPC.
2. The source agent server gets serverID of the destination agent server from the second argument of the predicates `in/2`, `fasserta/2`, `fassertz/2`, and `fretract/2`.
3. The source agent server sends a RPC request to the destination agent for invocation of `receiveAgent` method.
4. The destination agent server decodes the argument of the RPC and restarts the decoded agent.

The mechanism for RPC is implemented using XML-RPC.

5 APPLICATIONS

In this section, one of the applications is demonstrated to confirm the effectiveness of Maglog.

5.1 Distributed e-Learning System

A distributed e-Learning system for asynchronous Web-based training has been built using Maglog. This system allows a student to study by himself in his

own time and schedule, without live interaction of a teacher.

Our distributed e-Learning system consists of exercise agents and user interface programs. Each exercise agent includes not only exercise data but teacher's functions to mark user's answers, to tell the correct answers and to show some extra information. Every computer of students receives some number of exercise agents from another computer when it joins the system and takes the responsibility of sending appropriate exercise agents to requesting computers.

The user interface program is developed as a plug-in program of Firefox web browser. XML-RPC is used for communication between the plug-in program and an agent server.

Figure 6 shows one part of key codes in this application. This procedure is a part of an exercise agent. This is the procedure to provide an exercise for a remote user. In executing this procedure, following steps are performed.

1. An agent retrieves a clause `request/2` which other agent added from `fieldA`. Here, `Host` and `Field` are host name and field name of student's computer.
2. The agent migrates to `Host` and enters `Field`, and it provides an exercise for student. When the student finishes the exercise, the agent returns to the host it belongs automatically.
3. The agent recursively executes this procedure.

```
    loop:-
1:      fretract(request(Field,Host),fieldA),
2:      in(provide_exercise,Field@Host),
3:      loop.
```

Figure 6: This is the procedure to provide an exercise for a remote user.

In this procedure, two types of field, fieldA and `Field` are used. `fieldA` in line 1 of Fig.6 is used as a medium of asynchronous communication between agents, and `Field` in line 2 is used as an abstraction of migration.

6 CONCLUSION

The new framework named Maglog for mobile agent systems was designed and developed on Java environment. In Maglog, a concept called "field" is introduced. By using this concept, migration, inter-agent communication and adaptation functions are realized.

The effectiveness of the proposed framework Maglog is confirmed through the demonstration of an application: the distributed e-Learning system.

Regarding the issue of error handling, Maglog currently handles only one type of error which occurs when an agent intends to migrate to a host. Handlings of errors after or during migration are remained as future works. Security issues are indispensable problems for distributed applications using mobile agents. In Maglog, programmable security functions are not provided sufficiently because security issues are vast. Those functions are due to be added in future works. In addition to make programs more practical, it is necessary to provide a program developing environment, such as a debugging tools and a testing tools.

REFERENCES

Banbara, M. and Tamura, N. (1999). Translating a linear logic programming language into Java. In M.Carro, I.Dutra, et al., editors, *Proceedings of the ICLP'99 Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages*, pages 19–39.

Carriero, N. and Gelernter, D. (1989). Linda in context. *Communications of the ACM*, 32(4):444–458.

de Carvalho, C. L., Pereira, E. C., and da Silva Julia, R. M. (1999). Netprolog: A logic programming system for the java virtual machine. In *Proceedings of the 1st International Conference on Enterprise Information Systems*, pages 591–598. Setubal, Portugal.

Kawamura, T., Hasegawa, T., Ohsuga, A., and Honiden, S. (2000). Bee-gent: Bonding and encapsulation enhancement agent framework for development of distributed systems. *Systems and Computers in Japan*, 31(13):42–56. John Wiley & Sons, Inc.

Lange, D. B. and Oshima, M. (1998). *Programming and Deploying Java Mobile Agents with Aglets*. Addison Wesley.

Satoh, I. (2000). Mobilespaces: A framework for building adaptive distributed applications using a hierarchical mobile agent system. In *Proceedings of IEEE International Conference on Distributed Computing Systems*, pages 161–168. IEEE Press.

Tarau, P. (1999). Inference and computation mobility with jinni. In Apt, K., Marek, V., and Truszczynski, M., editors, *The Logic Programming Paradigm: a 25 Year Perspective*, pages 33–48. Springer.

Winer, D. (1998). Xml-rpc specification. http://xmlrcp.com/spec.