Distributed Logic Programming using Mobile Agents

T. I. WANG
Department of Engineering Science
National Cheng Kung University
Taiwan
wti535@mail.ncku.edu.tw

K. L. Clark
Department of Computing
Imperial College
UK
klc@doc.ic.ac.uk

Abstract

This paper[1] *describes the use of mobile agent technologies in building a framework for supporting distributed logic programming and remote conditional querying. A mobile agent moves from server to server carrying its own knowledge. When it arrives at a server it is given read access to the some part of the server's knowledge base. It then answers its queries using the server's knowledge and its own, adding the results to its own knowledge. We can view the mobile agent's queries as conditional remote queries - what answers would the server give* if *the knowledge carried by the agent were added to the server.*

To implement a prototype framework of mobile agents and knowledge servers we have used a new multi-paradigm and multi-threaded programming language Go!. *This supports mobile agent's as knowledge objects and has powerful knowledge structuring features. The paper assumes some familiarity with Prolog.*

1 Introduction

Mobile agents are computations migrating between hosts. They locally communicate with other agents and processes running on each host and typically only need network communication when they move. Because of this, mobile agents consume fewer network resources and, therefore, the paradigm has been advocated as a network traffic reducing technology suitable for various distributed application domains. Use of mobile agents greatly benefits those distributed applications that compute with simple logic and large amounts of remote data. There have several implementations and prototype applications of mobile agents [5] [8] [12].

[1]The paper was written during the first author's visit to the Imperial College, UK. The visit was fully supported by the National Science Council, Taiwan

On the other hand, one of the major research themes in Distributed Artificial Intelligence (DAI) is knowledge base interaction. We have previously developed a distributed logic programming language, DK_Parlog$++$ to support this kind of application and shown how it can be used for enterprise modeling[2]. Each entity in the enterprise is represented as an active object or agent containing a Prolog knowledge base and a set of methods that enable this knowledge base to be queried and updated by other agents. Prolog rules within an agent can include remote queries of the form **Ag?Cond** to transparently access the knowledge of another agent. Each remote query is implemented as a method invocation using asynchronous messages.

Sometimes the evaluation of a query **Q** performed within an agent **C**, using its knowledge $\mathbf{K}_C$, will require several remote queries to another agent **S** to access its knowledge $\mathbf{K}_S$, resulting in frequent message exchange. In this situation, particularly if **C** is on a host with intermittent connection, or the part $\mathbf{K}_{C_Q}$ of C's knowledge accessed by **Q** is small, it is better to move $\mathbf{K}_{C_Q}$ and **Q** to the site of **S** and to have **Q** evaluated at **S** against a combined and $\mathbf{K}_S$. If we do this, we must ensure that the knowledge $\mathbf{K}_{C_Q}$ transferred to **S** does not interfere with the local knowledge $\mathbf{K}_S$ - it should not effect the evaluation of other queries that use $\mathbf{K}_S$. One way to do this, is to embed **Q** and $\mathbf{K}_{C_Q}$ inside a mobile agent that is given read access to $\mathbf{K}_S$.

This is our approach. A mobile agent, carrying knowledge as well as queries, migrates to different sites to evaluate its queries using a fusion of the knowledge it carries and that of the local knowledge server. As it moves from site to site, the knowledge it carries can be progressively augmented with the results of the queries, and this extra knowledge can be used in the evaluation of queries at subsequently visited sites.

A prototype mobile agent framework to explore this approach has been implemented using a new multi-paradigm, multi-threaded logic programming language Go! [4]. It has an imperative component of action rules and a declarative

component of function and relation rules. In addition, sets of rules can be wrapped as labeled, parameterised theories as in L&O[9]. These can be instantiated to produce theory objects, and query conditions can be evaluated relative to a theory object. Theory objects with action rule methods can also be transformed into active objects by spawning a call to such a method as a new thread.

We embed the knowledge carried by the mobile agent in a theory object **MbAgKn**. We then give it access to the knowledge of each server agent it visits as another theory object **SvrKn**. This keeps the server knowledge separate and allows the mobile agent and the local agent to have their own different definitions of some relation. The mobile agent evaluates its queries for **Svr** using calls of the form **SvrKn.Cond** instead of remote calls such as **Svr?Cond**, which involve network communication. When it moves, it carries forward its knowledge augmented with the results of its queries. A mobile agents is a theory object with an **activate** method that converts it into an active object.

`Go!` will be briefly described in section 2, followed by the introduction to our mobile agent framework in section 3. Our prototypical implementation in `Go!` is sketched in section 4 together with an example use. We discuss some related work in section 5, before concluding.

2 The Programming Language `Go!`

`Go!` [4] is a multi-paradigm programming language that is oriented to the needs of programming secure, production quality, agent based applications. It is multi-threaded, strongly typed (with type inference support), and higher order (in the functional programming sense). It has been designed to allow fast development of intelligent agent based applications involving multi-threaded agents. The servers our mobile agents visit will be multi-threaded, with an interface thread for accepting new mobile agent objects and a thread for each activated mobile agent.

`Go!` has relation, function and action procedure definitions. Threads execute action procedures, calling functions and querying relations as need be. They can spawn new threads and communicate with other threads via messages or shared objects.

Each thread has its own message buffer of unread messages that have been sent to it by other threads. The inter-thread communication is asynchronous and is modeled on the inter-thread communication of Erlang[1] and April[10]. A `Go!` application can be distributed over a network of hosts with transparent communication between threads running on different hosts. Threads are identified by handles, which are terms of the form **hdl(*Th*,*Rt*)**. The second argument ***Rt*** is a symbol name of a family of threads, usually all the threads in a single `Go!` invocation. We will refer to this as the ***root*** name. The first argument ***Th*** is a symbol that identifies the thread within the family. When a `Go!` program is started we can set both the root and thread names of the initial thread with command line arguments. This gives the initial thread an assigned public handle such as **hdl('main','shop')**. When a new thread is spawned it inherits the root name of its parent's handle, but is given a new local thread name.

A thread can execute a non-blocking message send:

Exp >> *Rec*

where *Exp* is an expression denoting the message to be sent and *Rec* is a **handle**-valued expression. This transfers the value of *Exp* to the message buffer of thread *Rec*, which could be in another invocation of `Go!` on another host. Any `Go!` data value can be sent as a message, including higher order values, theory objects, and data terms containing variables. Any variables in a message are replaced by entirely new variables when it is received. Variables are not shared across threads.

A thread can search for and remove a message from its message buffer using a simple `<<` message receive action, or by a executing a choice message receive action. A `<<` action has the general form:

MsgPtn::*Test* << *Sender*

where *Test* is optional. **::** can be read as *such that*. This action will suspend if there is no current message matching *MsgPtn* for which *Test* is true until such a message is added to the buffer by a communication from another thread.

To allow search for one of several messages, with an appropriate action for each, a choice message receive can be used which is a disjunction of rules of the form:

($MsgPtn_1$::$Test_1$ << $Sender_1$ -> $ActionSeq_1$
| ....
|$MsgPtn_k$::$Test_k$ << $Sender_k$ -> $ActionSeq_k$)

The `|` is read as 'or'. A choice message receive operates by examining each of the messages in the message buffer; applying each of the message patterns and tests of its disjunction of message rules to each message in turn. As soon as a message is found that satisfies the preconditions of some rule *R*, that message is removed from the buffer and the action sequence of rule *R* is executed – none of the other actions in the choice message receive will be executed. Execution of the choice message receive will suspend if there is currently no message satisfying any of the alternative rules, until such a message arrives. Optionally a timeout can be specified, with a linked action.

For knowledge representation the language borrows features from McCabe's *L&O*[9] object oriented extension of `Prolog`. In particular, it borrows the ability to group a set of definitions into a lexical unit called a labeled theory. Inheritance rules allow new labeled theories to be constructed by extending and modifying existing theories[3].

A labeled theory has the form:

Label{ Def_1. Def_2. ..... Def_n}

where the definitions are for functions, relations, action procedures and attributes. **Label** may contain variables. If so, the variables may be used in and are global to any definition of the theory - they are parameters of the theory. A term **$Label'**, where **Label'** is an instance of **Label** in which these parameters are given values, denotes an instance of the theory. Logically it is a copy the theory in which each occurrence of a parameter in a definition is replaced by its given value. It reality it is a Go! higher order closure - a pointer to the theory paired with the tuple of parameter values. It can be held as the value of a variable, sent in a message, or passed as an argument. An instance of a theory can be viewed as a theory object, with methods the functions, relations, procedures and attributes of the theory.

The following set of definitions constitute a mini-theory of a person:

```
person(Nm:string,A:number,Sx:symbol,Ls:list[string]){
  name=Nm.  age = A.  sex=Sx.
  lives(L) :- L in Ls }.
```

The label arguments **Nm**, **A**, **Sx**, **Ls** are parameters to the theory which, when known, make it a theory object representing a specific person. **in** is Go!'s list membership relation. The parameter types are declared as string, number, symbol and list of strings respectively. The class definition also implicilty introduces a new type **person**. We can use this for declaring that variables take a **person** object as their value.

We can create a **person** theory object, and query it, as follows:

```
P1=$person("Bill Jones",23,'male',["London","Cardiff"])
P1.name    returns name "Bill Jones" of P1
P1.age     returns age 23
P1.lives(Place)   gives solutions:
                  Place="London", Place="Cardiff"
```

A labeled theory can also contain procedures that execute actions. Below is a mini-theory/class definition of a simple mobile agent. Instances of this theory are **mobAg** agent objects. The thoery parameters are: a symbol **N** which is the agent name, a handle **H** which is the home location, a theory object **MbAgKn** of type **desires** which is the agent knowledge. **MbAgKn** has relations characterising a set of buying requirements and price constraints. A **mobAg** object can be sent as a message to a remote site and activated by spawning its **activate** procedure as a new local thread. This procedure is defined by a single -> action rule taking the name of the site as argument.

The **activate** procedure finds prices and catalogue descriptions of items using the **want**s relation of **MbAgKn**, the **entry** relation of theory object **Cat**, and the **acceptable** relation of **MbAgKn**. **Cat** is supplied by the server that launched it (its **creator**) in response to a **'catalogue'** request message. It is the local knowledge that the mobile agent needs to access. This could have been passed to the agent as an argument to the **initiate** procedure but requesting it by a message gives flexibility. Different local theory objects can be requested by different agents, and more than one may be requested.

```
mobAg(N:symbol,H:handle,MbAgKn:desires){
  name=N.
  __found = $dynamic[(symbol,list[symbol],number)]([]).
  found(Item,Descr,Pr):-
      __found.mem((Item,Descr,Pr)).
  activate('shop') ->   – proc. to activate agent
      'catalogue' >> creator; – request a catalogue obj
      Cat:catalogue << creator; – receive it
      (MbAgKn.want(Item),Cat.entry(Item,Descr,Pr),
          MbAgKn.acceptable(Item,Descr,Pr) *>
        __found.add((Item,Descr,Pr))); –save each answer
      this>> H. – send a clone home }
```

Each answer to the mobile agent's query, which is a 3-tuple comprising a symbol, a list of symbols and a number, is stored it in a private dynamic relation object **__found** - the __ prefix means it is a private. *> is Go!'s *forall* operator enabling an action to be iterated over all solutions to some query. In this case the action is to **add** each answer as a new tuple to the dynamic relation **__found**. (All Go! dynamic relations are unary, but can hold tuples of values, as here. Their argument type has to be declared.) The linked **found** relation is public, and can be used for accessing values stored in **__found**. **mem** is the accessing method for a dynamic relation object. When the *> query iteration terminates the agent sends a clone of itself (denoted by **this**), containing the updated **__found**, back to its home **H**. It could continue execution, but in this case the **activate** procedure immediately terminates.

```
desires:{
want('toaster').  want('kettle'). .....
acceptable('toaster',_,Pr):-Pr<15.
acceptable('toaster',Descr,Pr):-Pr<20, 'electronic' in Descr.
.....}.
shopping_launcher() ->
  $mobAg('MobAg1',self, $desires)
      >> hdl('main','shop'); – dispatch agent
  Ag:mobAg :: Ag.name='MobAg1' << _ ; – await return
  (Ag.found(Item,Descr,P) *> .....). – process results
```

An example use is given by the **shopping_launcher** definition: **self** will be the handle of the launcher thread - the home location of the agent. We assume that the

shop server has been launched with the assigned handle hdl('main','shop'). When the agent returns to its launcher thread each catalogue entry it has cached is accessed as Ag.found(Item,Descr,Pr).

The 'shop' server 'main' thread will be executing a procedure such as:

```
shop()->
  (Ag:mobAg << _  -> spawn {Ag.activate('shop')}
  |'catalogue' << S::S=hdl(_,'shop') -> $catalogue >> S
  );
  shop()}.
```

This is a recursive procedure that will be executed as an iteration. The body is a choice message receive containing two rules. The first accepts **mobAg** objects and spawns their activate procedure as a new thread transforming them into active objects. This new thread will terminate when the **activate** procedure terminates. The second accepts a request for a 'catalogue' theory object. Notice that checking that the sender of the request has a handle with root name 'shop' ensures the request comes from a mobile agent that the shop agent has spawned. As it is a local agent, it is just sent a pointer to the 'catalogue' theory, not a copy. Since the server program loops immediately after it has spawned a mobile agent there can be many mobile agents concurrently executing at the same time, each of which will have 'read' access to the local 'catalogue' theory.

3 The Mobile Agent Information Framework

The framework we propose is simple and is designed to have an open architecture. It allows logic servers to join and leave freely and easily. It comprises: *mobile agents*, *logic servers*, and *agent launchers*.

3.1 The Mobile Agents

The components that comprise a mobile agent are shown as in the right of figure 1. Apart from **Agent Knowledge**, agent **State**, there is another **Control Part** component. The **Control Part** of a mobile agent determines its itinerary and queries to be executed at each visited server. It is this part that makes a mobile agent active and autonomous.

The activities of a mobile agent after arriving at a logic server are illustrated in figure 1. It is launched by the server and passed one or more theory objects defined in the server, on request. The supplied knowledge, together with its own knowledge and current state is used to answer queries for that server. The results are cached in its state.

3.2 The Logic Server

A logic server provides an executing environment and theory objects for mobile agents. Logic servers may all be

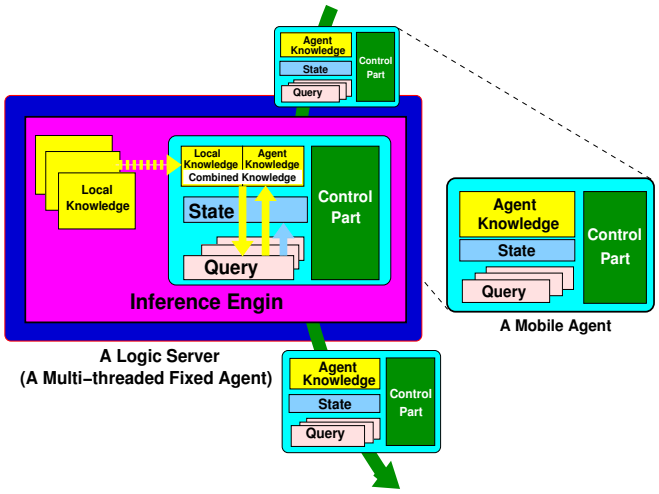

Figure 1. A Mobile Agent in Action

implemented in the same logic programming language, for example `Go!`. In this case, the knowledge transfer to a mobile agent is easier to implement. However, it is reasonable to allow other languages and data base systems to be used within a logic server. Then, an adapter, or bridge, has to be deployed. It translates knowledge represented in the other language or data base into the format used by our framework, or it gives indirect access to that information through the supplied knowledge.

3.3 The Agent Launcher

An agent launcher is a tool for creating mobile agents and sending them to logic servers. A programmer may use a graphic user interface (GUI) for entering queries and related information, such as the targeted logic servers of each query, the knowledge of the mobile agent etc. This data is then translated into an appropriate mobile agent theory an instance of which is launched to roam the network of logic servers.

4 A Prototype Implementation and Example

Assume there are three Logic Servers: Office, Headquarters and Warehouse, with knowledge as shown. Initially, there is no knowledge in the Office server.

Headquarter Logic Server

```
uprice('t00001',5000).  uprice('t00002',6400).  .....
product('t00001','tainan').
product('t00001','kaohsiung').
product('t00002','taipei').  .....
```

Warehouse Logic Server

```
stock('t00001',300,'tainan').  stock('t00002',200,'taipei').
stock('t00001',400,'kaohsiung').  .....
throughput('t00001',200). throughput('t00002',300).  .....
available(T,F,N,UP,TP,D) :-
        stock(T,D1,F), D1 >= N, To=UP*N, D=0.
available(T,F,N,UP,TP,D) :-
        stock(T,D1,F), D1 < N, throughput(T,O),
        To=UP*N, D=ceil(N-D1)/O.
```

The knowledge has to wrapped inside a process that implements each server. They all have a similar structure. The following code fragment is a simple *office* logic server.

```
Order = $dynamic[(string,symbol,symbol,number)]([]).
orders{order(Ord) :- Order.mem(Ord)}.
ordersOb=$orders.
office() ->
   ( Agent: salesAg << S ->
        spawn {Agent.activate('office') };
   | 'orders' << S :: S=hdl(_,'office') ->
        ordersOb >> S
   |  NewOrd << _ ->
        Order.add(NewOrd));
   office().
```

Order is a dynamic relation object which initially contains no entries. It enables the office server to accumulate knowledge over its operational life.

The *office* logic server loops accepting three types of messages. The first is mobile agent object of type **salesAg**; the second is a request from a launched mobile agent for read access to the 'orders' theory object; the third is new data from a sales person to add to the **Order** dynamic relation. [2] When a new agent object is received, a call to its **activate** method is spawned as a new thread. Orders are sent as messages such as

```
   ("CompShop",'harddisk','t00001', 100)
```

from a sales person giving the customer name, product name, type and quantity.

The sales person may later want to check out the details of all their orders, perhaps to find out the total price for each order and when it will be delivered. Using remote calls, the query they need to evaluate is given as the query:

```
customer(Cust), order(Cust,Product,Type,Qnty)@office,
product(Type,Factory)@hq,uprice(Type,UPr)@hq,
available(Type,Factory,Qnty,UPr,TPr,DTm)@wh.
```

Here **customer** is a local relation on the sales person's computer. If solutions to this query are found using the normal Prolog-style backtracking evaluation it will result in many remote calls. For example, if there are 50 customers, where each has on average two outstanding orders, there will be 50 remote calls to **office**, 200 remote calls to **hq** and 100 to **wh**, even though several of the calls to the last two sites may redundantly concern the same type of product.

```
salesAg(Nm:symbol,H:handle,MbAgKn:customers){
    name=Nm.
   __order=$dynamic[(string,symbol,symbol,number)]([]).
   __product_price=$dynamic[(symbol,symbol,number)]([]).
   __ans=$dynamic[(string,symbol,symbol,number,
                        symbol,number,number)]([]).
   ans(Cst,Prd,Tp,Qnty,Fct,TPr,Dlvry):-
        __ans.mem((Cst,Prd,Tp,Qnty,Fct,TPr,Dlvry)).
   activate('office') ->
        'orders' >> creator;
        Ords:orders << creator;
        (MbAgKn.customer(Cst),
         Ords.order(Cst, Prd, Tp,Qnty) *>
              __order.add((Cst, Prd, Tp,Qnty)))
        this >> hdl('main','hq').
   activate('hq') ->
        'products' >> creator;
        Prods:products << creator;
        (__order.mem((_, _, Tp,_)), Prods.product(Tp,Fct)
         Prods.uprice(Tp,UPr) *>
            __product_price.add((Tp,Fct,UPr));
        this >> hdl('main','logistic').
   activate('wh') ->
        'logistics' >> creator;
        Logs: logistics << creator;
        (__order.mem((Cst,Prd,Tp,Qnty)),
         __prod_price.mem((Tp,Fct,UPr)),
         Logs.avaliable(Tp,Fct,Qnty,UPr,TPri,Dlvry) *>
            __ans.add((Cst,Prd,Tp,Qnty,Fct,TPr,Dlvry)));
        this >> H}.

customers:{customer("CompShop"). .....}.
launcher(Nm) ->
   $salesAg(Nm,self,$customers)>> hdl('main','office');
   Ag:salesAg :: Ag.name=Nm << _;
   (Ag.ans(Cst,Prd,Tp,Qnty,Fct,TPr,Dlvry) *> .....).
```

Instead of executing the query with remote calls, a mobile agent is created and launched, which is an instance of the **salesAg** class. It visits the logic servers *office*, *hq*, and *wh* in turn to implement the distributed query. The knowledge it starts with is just the **customer** relation of the sales person wrapped as a knowledge object **MbAgKn**. At the **office** site it adds all the details of the orders for these customers and carries this forward to **hq**. Here it adds factory data and unit prices for all their ordered product types before continuing to **wh** to get a total price for each order and a delivery time from each factory at which its product type is manufactured. The **activate** procedure is defined by three rules.

[2] A more complex server would also allow removal of orders.

5 Related Work

Jinni[11] is a lightweight, multi-threaded, Internet logic programming language. It is a tool for gluing together knowledge processing components and Java objects. It supports remote predicate calls without distributed unification and is multi-threaded. Jinni supports mobile agents through thread migration.

Milog[7] is a logic programming based infrastructure for information gathering mobile agents. On each site is a meta-agent which accepts mobile agents and sends agents to other meta-agents on request from the agent. The meta-agent encodes and decodes the agent programs and checks incoming code. In Go!, encoding and decoding of the agent objects is done automatically by the $>>$ and $<<$ primitives which automatically checks code safety in an external message before being transferring it to a message buffer.

The related April language[10] is used to implement a mobile agent framework for distributed information management in [6]

6 Conclusion

Our framework is suitable for applications requiring relatively coarse-grained distributed deductions. As the agent travels it incrementally constructs the solution to the distributed query that is its mission. We have illustrated this for an agent with just one query, but several queries can be handled, and at each site the different queries can make use of different local theory objects requested via messages. An agent can also make use of the knowledge it accesses at a server to determine what subsequent servers to visit, making its route dynamic.

We can also handle disjunctive queries. Suppose our distributed query has the form:

$\text{Cond}_1@\text{S}_1$,($\text{Cond}_2@\text{S}_2$ | $\text{Cond}_3@\text{S}_3$),$\text{Cond}_4@\text{S}_4$

We program the agent with four activate rules, one for each server. But the rule for S_2 finishes by sending a clone to both S_2 and S_3 using:

this $>>$ S_2; this $>>$ S_3

Each clone is then activated by the rule for its new location, each of which terminates with the action: this $>>$ $\mathbf{S}_4$. The launcher will now receive two returning agents, one with the query answers obtained from S_1,S_2,S_4, the other with the answers obtained from S_1,S_3,S_4, which it combines.

References

[1] J. Armstrong, R. Virding, and M. Williams, Concurrent Programming in Erlang, Prentice-Hall International, 1993.

[2] Keith Clark, Nikolaos Skarmeas, Tzone Wang, "Distributed Object Oriented Logic Programming as a tool for Enterprise Modelling", In Modelling and Methodolgies for Enterprise Integration, (ed Bernus & Nemes), Chapman and Hall, 1996.

[3] K. Clark and F. McCabe, "Ontology Representation and Inference in Go!", Technical Report, www.doc.ic.ac.uk/~klc/ontology.html, 2003.

[4] K. Clark and F. McCabe, "Go! – a logic programming language for implementing multi-threaded agents", Annals of Mathematics and Artificial Intelligence, Special Issue on Logic based Agent Implementation, 2004, to appear.

[5] S. Covaci, Zhang Tianning, I. Busse, "Java-based intelligent mobile agents for open system management," In Proceedings of Ninth IEEE International Conference on Tools with Artificial Intelligence, Page(s): 492 -501, 1997.

[6] J. Dale, D. C. DeRoure, A mobile agent architecture for distributed information management, Proceedings of the International Workshop on the Virtual Multicomputer 3, 1997.

[7] Naoki Fukuta, Takayuki Ito, and Toramatsu Shintani, "A Logic-based Framework for Mobile Intelligent Information Agents", In Proceedings of 10th International WWW Conference, http://www10.org/cdrom/start.htm, 2001.

[8] D. Kotz; R. Gray; S. Nog; D. Rus; S. Chawla; G. Cybenko, "AGENT TCL: targeting the needs of mobile computers," IEEE Internet Computing Volume: 1 4 , Page(s): 58 -67, 1997.

[9] F. McCabe. L&O: Logic and Objects. Prentice-Hall International, 1992.

[10] Frank McCabe and Keith Clark, "April: Agent Process Interaction Language," In Intelligent Agents, (ed Jennings & Wooldridge), LNCS, Vol 890, Springer-Verlag, 1995.

[11] Paul Tarau and Veronica Dahl, "A Logic Programming Infrastructure for Internet Programming", In M. J. Wooldridge and M. Veloso, editors, Artificial Intelligence Today - Recent Trends and Developments, pages 431-456. Springer, LNAI 1600, 1999.

[12] T. I. Wang, "A Mobile Agent Carrier Environment for Mobile Information Retrieval," 11-th International Conference on Database and Expert Systems Applications - DEXA 2000, 05-08/09, 2000.