Active State Space: a Distributed Supporting Environment for Autonomous Mobile Agents

Minjie Zhang
Department of Computer and Software Engineering
The University of Newcastle, NSW 2308, Australia
minjie@cs.newcastle.edu.au

Wei Li
Department of Information Management
Capital University of Economics and Business
Beijing 100026, China
liw@cueb.edu.cn

Abstract

Agents have to be both autonomous and mobile in many WWW applications such as mobile computing. In this paper, we present *ASS* (*Active State Space*), a model of distributed supporting environments for autonomous mobile agents. We use *ASS* as a mediate media, which maps heterogeneous resources and knowledge representations into homogeneous virtual resources. In this manner, *ASS* is able to deal with heterogeneity and dynamicity of Internet. For supporting autonomies of mobile agents, *ASS* decouples the rigid and complex protocol, usually used in agent communications, into simple, fast and local accesses to *ASS* spaces by providing a unified interface for facilitating interactions of both *inter-agents* and *agent-hosting execution environments*. In the context of autonomy, asynchronous, anonymous and fully local interactions among mobile agents can be obtained by exchanging messages with *ASS*. We can use autonomous, lightweight and mobile agents and simple interaction protocols to construct WWW applications on unified *ASS* regardless of the heterogeneity and dynamicity of multiple platforms. A prototype of *ASS* has been implemented in Java as a component of a mobile agent system *MAT* (*Mobile Agent Template*). This paper also presents a case study of distributed information retrieval. The case tests interactions among mobile agents, which evaluates the effectiveness of *ASS* for supporting executions of autonomous workflow of mobile agents.

Keywords: distributed supporting environment, agent interaction, autonomous agents, and mobile agents

1 Introduction

It is necessary for a mobile agent to have autonomy in many WWW applications such as mobile computing [13, 17]. In a typical mobile agent application, usually, a user launches a mobile agent from a

laptop that is connected to the Internet, then the user disconnects the laptop from the Internet. The mobile agent travels in the Internet autonomously, retrieving and updating information locally on behalf of its owner. Later, the mobile agent will return to the user's laptop and report the results when the user's laptop is reconnected to the Internet. Mobile agents should have the "intelligence" for self-contented navigation, computation and communication that give them the adaptation powers to the dynamic and heterogeneous network, because in most cases mobile agents can not interact with their owners. Usually, an autonomous mobile agent should have three kinds of autonomies as defined in Definition 1 to Definition 3.

Definition1: Mobile *autonomy* is the capability of self-navigation of mobile agents through the underlying networks.

Definition 2: Computational *autonomy* is the capability of self-contention of mobile agents in computational functions for a distributed task accomplishment.

Definition 3: Communicational *autonomy* is the capability of self-containment of mobile agents in sending and receiving messages asynchronously, anonymously and indirectly.

However, no matter how much degree of autonomy they have, mobile agents need interactions of *inter-agents* for cooperation and interactions of *agent-hosting execution environments* for resource accesses. Many interaction models [1][2][12][14][19] rely on network communication and naming services, which decrease the autonomy of agents. Those models cannot provide asynchronous, anonymous and fully local interactions for mobile agents, so those models are not meaningful for agent communication in the context of autonomy. In order to overcome these limitations, we have designed *ASS* (*Active State Space*), a distributed supporting environment for autonomous mobile agents. *ASS* supports the autonomous workflow for mobile agents in three ways. Firstly, asynchronous, anonymous and fully local *inter-agent* interactions are obtained by exchanging messages with *ASS* and are not involved in any network communication and naming services. Secondly, abstract and homogeneous description of distributed resources simplifies complexities of agent design, and decreases coding loads carried by mobile agents to access heterogeneous resources. Finally, a unified interface facilitates both interactions of *inter-agents* and *agent-hosting execution environments* with security mechanisms embedded at the same time. Some case studies, such as distributed information retrieval, have demonstrated that *ASS* maximizes the autonomy of agents in interactions and minimizes network communication with low reliability in Internet.

This paper is organized as follows. In Section 2, we give our perspectives of the relationship between autonomy and interaction of mobile agents, and sum up preconditions that must be satisfied for mobile agent interactions in the context of autonomy. In Section 3, we present the architecture of *ASS* including design motivations, components and *Reflection* mechanisms of *ASS*. In Section 4, a case study of Internet information retrieval demonstrates the effectiveness of *ASS* in interactions among autonomous mobile agents. Finally, in Section 5, we present our conclusions and directions of future researches.

2 Problem Description of Autonomy vs. Interaction

Autonomous mobile agents have characteristics of desirable autonomies without interactions with their owners when the agents roam in Internet for many WWW applications such as mobile computing. However, no matter how much degree of the autonomy they have, autonomous mobile agents exist in a community composed of other agents and heterogeneous hosting environments. Besides the interactions with their owners at their home machines, autonomous mobile agents need at least two kinds of interactions during their lifecycles.

1. Interactions of *inter-agents* are needed when a WWW application consists of multiple mobile agents. Let's consider the following scenario. When a mobile agent realizes that its task can be concurrently executed, for example a mobile agent finds other n interesting links in a HTML page while retrieving contents of the page, the mobile agent will duplicate $n-1$ its instances with retrieval tasks. The agent and its $n-1$ duplicates will transport to n interesting sites for further information retrievals concurrently. Interactions of *inter-agents* occur when results from different agents are synthesized into a final result for submission to the agent's owner at the home machine. This kind of interaction is often necessary for cooperation among agents.

2. Interactions of *agent-hosting execution environments* are needed when a mobile agent accesses resources and services in a local execution environment at its visiting network nodes. All the necessary information in a hosting execution environment must be structured for mobile agent accessing, and protocols for retrieving, updating and protecting information must be carefully defined.

There are new features in intersections among autonomous mobile agents or between autonomous mobile agents and their hosting execution environments. In the context of autonomy, interactions should be fully asynchronous, anonymous and local between communication partners. Firstly, asynchronism does not require a strict agreement on communication time and a place while synchronism implies that any interaction between communication partners depends on complex protocols, synchronization mechanisms and realities of networks. For direct interactions, mobile agents would have too many code loads for the protocol and the synchronous mechanism accomplishments. Secondly, if an autonomous mobile agent must know the communication partners' name for interactions, naming services would be involved in agent communications. The naming services result in extra network (not local) interactions between mobile agents and naming service agencies. Too many network interactions weaken the autonomy and reality for which we pursue in many WWW applications by mobile agents. Also, it is not necessary for an autonomous mobile agent to know other agents' names because it is enough for an agent to interact as long as the agent can identify that its communication partners and itself belong to the same application. Finally, any interaction should be local. An important advantage of mobile agents over other distributed technologies is the locality of distributed information processing because the locality leads to advantage of

mobile agents in terms of high efficiency, reliability and flexibility. So interactions between autonomous mobile agents and their communication partners should not introduce any extra network communication.

Minimization of network communications and localization of interactions are the pursuits in the context of autonomy. To achieve the above aims, we cannot use the extensively studied agent communication models such as the direct communication model and the rendezvous model [1][2][14], or the meeting-oriented model [12][19] because these models do not have the above features that are suitable for interactions of autonomous mobile agents. Instead, we design a new model: *ASS* (*Active State Space*), which is a third party to facilitate interactions between autonomous mobile agents and their communication partners for cooperation or resource accesses. Generally, three basic components, which are mobile agents, agent servers and hosting execution environments, comprise the architecture of a mobile agent system. Mobile agents have to interact with an agent server at each of their visiting network nodes for the transportation, the execution restoration or the resource access. Although mobile agents can directly access resources of a hosting execution environment, theoretically, it is not the case due to security and heterogeneity. So an agent server is already the third party for supporting mobile agent applications. By introducing *ASS* into agent server to facilitate interactions, we construct a unified distributed supporting environment without introducing any extra spatial and time complexity. Some ideas of *ASS* design come from Jada [5]. In Jada's space, objects are passive. *ASS* takes Jada's concepts one step further by allowing components of *ASS* to actively respond to interaction events and by allowing components to be installed, updated and deleted dynamically by the *ASS* itself or mobile agents.

3 Active State Space

3.1 Motivation of *ASS* Design

There are three motivations for the design of *ASS*. The first motivation is to provide a unified description of resources that are accessed by mobile agents. The typical application environments of mobile agents are heterogeneous and dynamic. Internet nodes are composed of different architectures, knowledge representations and processing technologies. It is impractical (although possible) for mobile agents to face heterogeneity by themselves because it makes mobile agents have complex and heavy code loads. In addition, agents must be redesigned to suit new environments. *ASS* abstracts the resource descriptions of different systems into the data integrity, so *ASS* takes charge of handling heterogeneity and dynamicity of the hosting execution environments. Both agents and information resources are rid of the burden of dealing with heterogeneity. In this aspect, *ASS* is quite similar to JVM (Java Virtual Machine) in coping with heterogeneous architectures for programming platforms, independently. Mobile agents use abstract resource descriptions and a unified interface for accessing the real resources. *ASS* is responsible for the mapping between abstract resources and real resources. *ASS* can be designed to take the speciality of local resources into account, so mobile agents can use integrated and simple protocols to access resources.

The second motivation is to provide communication facilities for autonomous mobile agents. Interactions among agents, by means of *ASS*, are not involved in any network communication and naming service. Interactions among agents are fully asynchronous, anonymous and local. That means a mobile agent can put messages (called *Record* objects) in *ASS*, which another agent can access later. In this respect, *ASS* is similar to Linda [11] where a tuple space provides shared memory for processors to communicate by storing and retrieving tuples. This kind of interaction makes mobile agents not face critical issues, such as heterogeneity, dynamicity and unreality of networks. While direct *inter-agent interactions* require strict agreements among mobile agents on whom and where communication partners are, and when to connect to each other, it would be too complex, ineffective and unreliable for mobile agents to communicate with each other directly.

The last motivation is to provide a security interface between mobile agents and system resources. Mobile agents use abstract resources exposed by *ASS* rather than real resources. It is natural to associate any requirement of resource access with security inspections in the mediate level *ASS*. It is necessary for *ASS* to perform security inspections on some sensitive operations such as deleting files because a malicious or bad-programmed agent could possibly contain dangerous codes. Resource abstractions and interaction modes with *ASS* are graphically illustrated in Fig. 1.

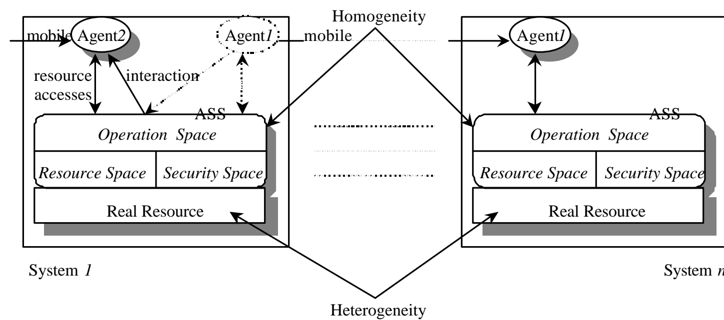

Fig.1 Resource abstraction of *ASS* to heterogeneous architectures

3.2 *Active Records*: Components of *ASS*

ASS is composed of *Resource Space*, *Security Space* and *Operation Space*. Abstract resources and messages for *inter-agent* communications are stored in the *Resource Space*. Any requirement to access *Resource Space* submitted to *ASS* by mobile agents will fire an *Operation* object in the *Operation Space*. For the acceptance or rejection of mobile agents' requirements, the *Operation* object needs to match the requirements with *Security Rules* in *Security Space*. This interaction procedure between mobile agents and hosting execution environments is called *Reflection of Operation* object in *ASS*. So every *Operation* object has a segment of *Reflection* codes. A successful reflection changes the states of *ASS*, and a failed reflection yields no influence on the *ASS*.

In an *ASS* any component such as an abstract resource, a security rule or an operation can be installed and deleted dynamically by the *ASS* itself or mobile agents. Components in above three spaces are integrated into abstract description called *Active Record* as defined from Definition 4 to Definition 6.

Definition 4: A Record in an *ASS* is composed of objects that are called *sub-records*, i.e. *<Record>::=<{subrec,}*>*, where *subrec* denotes any *sub-records* which compose the *Record*.

Definition 5: A *Record* is an *Atomic* Record **iff** its every *sub-record is* not a *Record* object.

Definition 6: A *Record* is a Coupled *Record* if at least one of its *sub-records* is a *Record* object.

A file in a local file system can be structured with an *Atomic Record* object with the type of *<String path, String Type, Date UpdateTime, Username owner>*. The *Atomic Record* object of *<'/user/wli/document/', 'html', 2/22/99, wli@cowan.edu.au>* represents system file resources, i.e. all the *html* files that are located in the directory of *user/wli/document/*, owned by user *wli@cowan.edu.au* and updated in *February 22, 1999*.

A message that a mobile agent puts in an *ASS* for communications with other agents can be structured with an *Atomic Record* with the type of *<String Results, Integer ApplicationID, Integer TaskID>*. Every application has a unique identification *ApplicationID*, and every distributed task of the same application has a unique identification *TaskID*. Any agent can get the results of a distributed task by providing the *ApplicationID* and *TaskID* to the resource matching mechanism of the *ASS*.

A security rule of *ASS* can be structured with a *Coupled Record* object with the type of *<Integer AgentID, String Operation, AtomicRecord Rec>*. The *Coupled Record* object of *<*, 'r',* Rec_1*>* represents that any agent can read *Record* Rec_1. Another *Coupled Record* object of *<44357676, 'rdu',* Rec_1*>* represents that mobile agent identified by *44357676* can read, delete and update *Record* Rec_1.

An *Operation* object of an *ASS* can be structured with an *Atomic Record* object with the type of *<String OperationName, Operation OpObj, Username Owner>*. Where the *OperationName* is the name of the *Operation*, *OpObj* is the instance of the *Operation* and the *Owner* is the entity (agent or local system) to install the *Operation*. Every *Operation* object has a *reflection* method. The *reflection* method is executed when the *Operation* object is called by *ASS*. An example of *reflection* method of *Operation* object *FileReader* is illustrated in Procedure 1 (see Fig.2 for the meaning of *NCR*).

```
public class FileReader implements Reflection;

  public Vector reflection(CoupledRecord NCR);
  {  Vector FoundFiles=new Vector();

     for (int i=0; i<NCR.VecRec.size(); i++)  // for each file that an agent wants to read
     { CoupledRecord SCR=new CoupledRecord(NCR.AgentID, NCR.VecRec.elementAt(i));
                                        //construct a Security-Checking Record
       if (securityInspection(SCR))  // if the agent has the right to read the file
             {File fileObject=newFileObject(NCR.VecRec.elementAt(i));
                                        //generate the file object of the file
              Foundfiles.addElements(fileObject); } }
     return (Foundfiles); }} //return all the file objects that the agent has the right to read
```

Procedure 1 the *reflection* codes of *Operation* object *FileReader*

A retrieval requirement from a mobile agent to a *hosting* execution environment can be structured as a *Coupled Record* object of the type *<Integer AgentID, String Operation, Boolean Synchronization, AtomicRecord Rec>*. The *Coupled Record* object *<44357676, 'r', true, <'/user/wli/document/', 'html', *, *>>* represents that the mobile agent wants to read the entire *html* files in the directory of */user/wli/document/* in a *synchronous* mode. If the resource matching and the security inspection succeed, the *ASS* will return a vector of references to the required *html* files.

3.3 Reflection Mechanism of *ASS*

In this subsection, we introduce the reflection mechanisms of the *ASS* in detail. The *Resource Space* stores *Atomic Records*. These records abstractly describe system resources of the hosting execution environments or messages that mobile agents put for asynchronous interactions. The *Security Space* stores *Coupled Records*. These records describe security rules that determine which agents can do what kinds of operations on which records in the *Resource Space*. The *Operation Space* stores *Atomic Records*. These records describe *Operation* objects that are identified by names and are called by the *ASS* when mobile agents interact with the *ASS*. An *Operation object* can be installed by an *ASS* system or by mobile agents for some special reflection operations to interaction events.

Every agent server hosts an *Active Sate Space* object. When a mobile agent transports into a network node, the agent gets the reference to an *ASS* object from the agent server. An important feature of an *ASS* is that *Operation Records*, which are responding entities to interaction events, can be programmed, and can be installed dynamically by agents or an *ASS* itself. The whole procedure of an interaction event from the interaction requirement submission to the result return is shown in Fig. 2.

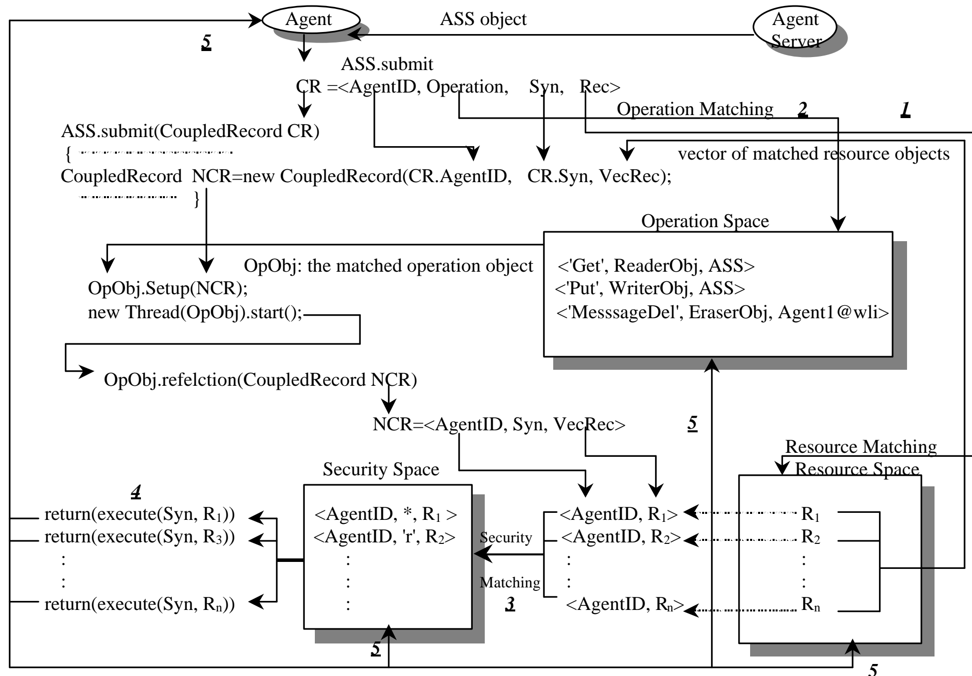

Fig.2 The reflection procedure of *ASS* object to an interaction requirement from mobile agents

Every interaction requirement can be represented by a *Coupled Record: <AgentID, Operation, Syn, Rec>*, which means that a mobile agent identified by *AgentID* hopes to execute an *Operation* on all the *Records* matching the template record *Rec* in a synchronous mode *Syn*. The agent provides the above requirement to an *ASS* object with the *submit* method of the *ASS* object to which the agent has a reference. The reflection procedure of an *ASS* to the interaction requirement goes through step1 to step 5, which corresponds to step 1 to step 5 in Fig. 2.

1. Searching the whole *Resource Space* to identify records that match *Rec*, and using these records to construct a vector *VecRec*.

2. Searching the whole *Operation Space* to identify a record that matches *Operation*, then generating a thread for the matched operation object *OpObj,* and starting the execution of the thread.

3. Constructing a record *SR*=<AgentID*, R*> for each record *R* in vector *VecRec* by *Operation* thread and searching *Security Space* to identify the presence of a record that matches *SR*.

4. Executing the required *Operation* on *R* by *OpObj* in the synchronous mode *Syn,* when a matched record is found in *Security Space*.

5. Returning the results of the *Operation* to the agent or storing them into *Resource Space*, *Security Space* or *Operation Space*.

4 A Case Study of Interactions among Autonomous Mobile Agents

4.1 Autonomous Workflow Model in *MAT* for Mobile Agents

One of the aims of our mobile agent system *MAT* (*Mobile Agent Template*) [18] is to generate autonomous workflow for mobile agents. In *MAT* we use *DTP* (*Distributed Task Plan*) [18] as a static template to describe distributed tasks, satisfying user's requirements. An execution of *DTP* by a mobile agent in a dynamic network environment is an autonomous workflow of the mobile agent when it roams the Internet.

In order to obtain desirable autonomies for mobile agents, firstly, we provide enough programming components called autonomous primitives. These primitives are used to construct *DTP* and further provide mobile agents with autonomies in navigation and computation when agents roam in the Internet. Four kinds of autonomous primitives are defined in *MAT* for *DTP*, and they are:

1. Mobile primitives define the mobility of mobile agents for their roaming in networks. A concrete example of mobile primitives can be found in subsection 4.4.

2. Computational primitives define invocations of computational resources that are distributed in networks. For example, the computational primitive *ExterExecution* permits a mobile agent to invoke a pre-defined computational function for the execution in a different process from that of the mobile agent. This primitive is often used when the invoked function is programmed in a heterogeneous language from that of the mobile agent. *<ExterExecution>::= <FunctionPath, Synchronization, Results, Action>*, where *FunctionPath* is an absolute or relative path to the location of the pre-defined function can be found by the

operating system. *Synchronization* can be true or false corresponding to the synchronous or asynchronous execution of the function. Results are objects to store returns from the invoked function. *Action* is illustrated in Procedure 2.

```
Runtime rt=Runtime.getRuntime(); //get run time object of the system
Process proc=rt.exec(FunctionPath); //run the function in a process
Results=proc.getInputStream().Read(); //get the results
if  (Synchronization)  proc.WaitFor(); //wait the process to finish
```

Procedure 2 The *Action* of computational primitives *ExterExecution*

3. Solution synthesis primitives define the combination of multiple solutions from different mobile agents of the same WWW application. Concrete examples of solution synthesis primitives can be found in subsection 4.4.

4. Control primitives define the execution flow of mobile primitives, computational primitives and solution synthesis primitives. For example, the control primitive *EventMonitor* continuously monitors an event until the event occurs. *<EventMonitor>::=* *<EventCondition, Primitive, Action>*, where *EventCondition* is the occurrence condition of an event that *EventMonitor* checks. *Primitive* can be a mobile primitive, a computational primitive or a solution synthesis object. *Action* is illustrated in Procedure 3.

```
while (!EventCondition) {Thread.sleep(time)};
if  (Primitive!=null)  Primitive.run();
```

Procedure 3 The *Action* of control primitive *EventMonitor*

A *DTP* consists of primitives, which are arranged into two lists. A list, called *Control Queue* (*CQ*), only contains control primitives, and another list, called *Reusable Primitive List* (*RPL*), consists of any primitives except control primitives. The architecture of *DTP* is graphically illustrated in Fig. 3 (concrete meanings of primitives of Fig. 3 are defined in [20]).

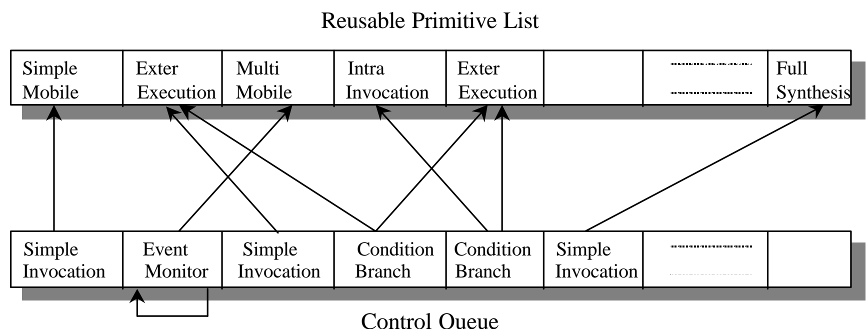

Fig. 3 The architecture of *DPT*

4.2 Planning, Running and Replanning of *DTP*

According to a user's objectives, a *DTP* object should be planned either by a mobile agent or a static agent. In order to make mobile agents lightweight in code loads and to support complex planning strategies, we move the planning algorithm of *DTP* [20] from the mobile agent in a previous version of *MAT* [18] into the static agent (called *Planner*) in current version. When a mobile agent is generated, it gets a task from its owner. The task is described by the following *Atomic Record OAR*.

OAR::=<String TaskLable, String Objectives*, Integer ApplicationID, Integer TaskID>*

Using procedure 4, the agent puts its OAR into a current *ASS*, and later gets its *DTP* object from the current *ASS*. The received *DTP* object is matched to the user's objectives described by *OAR*. Once a mobile agent gets its *DTP* object, the agent executes it.

```
ASS.submit(new CoupledRecord(this.AgentID, 'Put', true, OAR));
                    //submit a task description to the Planner in ASS
AtomicRecord   DTPID=new(*, OAR.Objectives, OAR.ApplicationID, OAR.TsakID);
AtomicRecord   DTPAR;
while ((DTPAR= ASS.submit(new CoupledRecord(this.AgentID, 'Get', true, DTPID)))==null)
    { Thread.sleep(time); }; //get a DTP that matches the submitted task description
DTP   DTPObject=DTPAR.DTPObject;
DTPObject.run(); // run the gotten DTP
```

Procedure 4 An agent interacts with *ASS* to submit a task description and get a *DTP* later

When the agent fails to execute its current *DTP* object due to the dynamicity of networks, it also uses the above codes to get a new *DTP* object, which is replanned by a *Planner* at its visiting network nodes, from the current *ASS*. The new *DTP* is an alternative workflow for the mobile agent to accomplish its owner's objectives in dynamic networks.

There is one *Planner* running on every distributed *ASS*. A *Planner* continuously gets users' objectives (described by *OAR*) from the current *ASS*, generates *DTP* objects and puts these objects also into the *ASS* using Procedure 5.

```
AtomicRecord   OARTemplate=new('task', *, *, *);
while (true) {
  AtomicRecord   OARObject=ASS.submit(new CoupledRecord(this.AgentID, 'Get', true, OARTemplate));
                 //
  if  (OARObject!=null) {
    DTP   DTPObject=planning(OARObject.Objectives); //plan DTP
    AtomicRecord   DTPAR=new AtomicRecord(DTPObject,OAR.Objectives, OAR.ApplicationID, OAR.TaskID);
    ASS.submit(new CoupledRecord(this.AgentID, 'Put', true, DTPAR)); }
                 //put the planned DTPs also into the ASS
  else Thread.sleep(time); }
```

Procedure 5 The Planner plans *DTP*s for agents

The procedure that a mobile agent interacts with a Planner to get its *DTP* is graphically illustrated in Fig. 4.

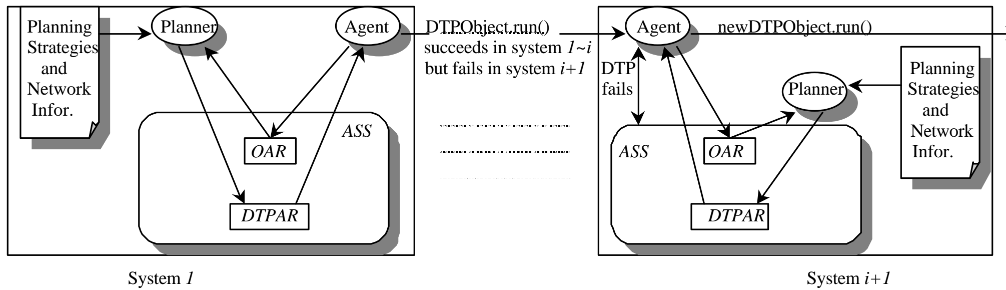

Fig.4 Planning, running and replanning of *DTP*

4.3 A Case of WWW Application for Distributed Information Retrieval

We have constructed a WWW application to evaluate the effectiveness of *ASS* in communicational autonomies. The scenario is Distributed Information Retrieval in Internet using mobile agents. A mobile agent roams into Internet from a mobile computer to retrieve references about a subject. Normally, a mobile agent completes a distributed task by visiting a series of network nodes serially and processing queries and updating information locally. However, a mobile agent can clone itself and assign a subtask to each of it's duplicates for executing a distributed task concurrently when multiple resources are available and a distributed task can be divided into concurrently executing subtasks. For example, if a mobile agent finds other *n* interesting links in an HTML page while retrieving contents of the page, the mobile agent will duplicate *n-1* instances with respective retrieval tasks. The agent and its *n-1* duplicates will transport to *n* interesting sites for further information retrievals. The solutions of subtasks from different mobile agents are combined into a result finally. In our system, we construct the task of distributed information retrieval by generating and assigning a *DTP* to a mobile agent. The agent uses the communication facilities provided by *ASS* to complete its task.

4.4 Asynchronous Interactions among Autonomous Mobile Agents by *ASS*

Now let's consider a *DTP* that consists of primitives of *MultiMobile*, *FullSynthesis* and *PriSynthesis*. An agent that executes that *DTP* can autonomously retrieve information distributed in Internet. That *DTP* is typical because it generates cooperation between autonomous mobile agents with the aid of *ASS*.

When a mobile agent *PA* encounters a *MultiMobile* primitive, it knows that its distributed task *DT* will be executed concurrently on destinations $D_1, D_2, \ldots\ldots, D_{n-1}$, and D_n. The agent *PA* (called Parent) duplicates *n-1* its instances $SA_1, SA_2, \ldots\ldots, SA_{n-1}$ with respective subtasks $ST_1, ST_2, \ldots\ldots, ST_{n-1}$. Each of the duplicates will transport to $D_1, D_2, \ldots\ldots, D_{n-1}$ respectively, and finally the parent agent *PA* will transport to the destination D_n with a subtask ST_n. The semantics of *MultiMobile* is illustrated in Procedure 6.

```
for each D in {D1, D2, ..., Dn-1}
        { Agent   SA=this.Duplicate(); //the agent clones n-1  instances
          SA.MobileTo(D);  };
MobileTo(Dn);
     //the agent and its clones will execute their tasks on different network nodes concurrently
```

Procedure 6 The semantics of primitive *MultiMobile*

Subtasks ST_1, ST_2, …, ST_{n-1} and ST_n can be same or different. If our aim is to exploit distributed resources for executing a distributed task for a fast response, subtasks should be same, such as getting the same program table from different TV stations. If our aim is to *exploit* distributed resources for retrieving different information, subtasks should be different, such as getting different price tables of the same travel from different travel agencies. Because SA_1, SA_2, ……, SA_{n-1}, and *PA* concurrently execute subtasks ST_1, ST_2, ……, ST_{n-1}, and ST_n divided from *DT*, results from different mobile agents are combined into a final result at a network node *Site* on which agents agree. Results of subtasks ST_1, ST_2, ……, ST_{n-1}, and ST_n from different agents can be described by the following *Atomic Record SAR* of an *Active State Space*.

SAR::=<String Results, *Integer* ApplicationID, Integer TaskID>

If ST_1, ST_2, ……, ST_{n-1} and ST_n are same, agents SA_1, SA_2, ……, SA_{n-1}, and *PA* will execute the *PriSynthesis* primitive at the same network node *Site*. When a mobile agent encounters a *PriSynthesis* primitive and the agent is the first one to accomplish this special subtask identified by task identification *TaskID*, the agent will continue to be alive, otherwise the agent will exit its execution. We can use *Atomic Record LAR* as the message in an *ASS* to identify whether an agent, just transporting to the *Site*, is the first one to accomplish the special task.

LAR::=<String State, Integer ApplicationID, Integer TaskID>

If a mobile agent cannot find the *LAR* in the current *ASS*, the agent is the first one to accomplish the special task identified by ApplicationID and *TaskID*, so it will put a *LAR* into the *ASS*. If a mobile agent can find the *LAR* in the current *ASS*, the agent is not the first one and will exit its execution. The semantics of *PriSynthesis* is illustrated in Procedure 7.

```
AtomicRecord  FinishLable=new AtomaticRecord('success', this.ApplicationID, this.TaskID);
AtomicRecord  LAR=ASS.submit(new CoupledRecord(this.AgentID, 'Get', true, FinishLable));
                                      //search LAR label in current ASS
if (LAR==null) || (!LAR.State.equal('success'))
              ASS.submit(new CoupledRecord(this.AgentID, 'Put', true, FinishLable));
     //The first agent that finishes a special task successfully will put a LAR label in current ASS
else this.shutdown(); //other agents will exit their executions
```

Procedure 7 The semantics of primitive *PriSynthesis*

If ST_1, ST_2, ……, ST_{n-1} and ST_n are different, agents SA_1, SA_2, ……, SA_{n-1}, and *PA* will execute the *FullSynthesis* primitive. The *FullSynthesis* primitive combines all the results of subtasks ST_1, ST_2, ……, ST_{n-1}, and ST_n from different agents SA_1, SA_2, ……, SA_{n-1}, and *PA*. The parent agent *PA* is responsible for the solution synthesis. The duplicated agents SA_1, SA_2, ……, SA_{n-1} will exit their respective executions after putting their results into the current *ASS* at *Site*. The semantics of *FullSynthesis* is shown in Procedure 8.

The above case study demonstrates the effectiveness of *ASS* to facilitate asynchronous, anonymous and local interactions among mobile agents in the context *of* the autonomy.

```
if (isParent()) //the parent agent is responsible for the solution synthesis
   { AtomicRecord  ResultRecord=new AtomaticRecord(*, this.ApplicationID, this.TaskID);
     while (SynthesizedNumber()<SubtasksNumber())
                         //all the results from clone agents will be combined into a finial result
       { AtomicRecord  SAR=ASS.submit(new CoupledRecord(this.AgentID, 'Get', true, ResultRecord));
         if (SAR!=null)
           {  this.Results=MergeWith(SAR.Results);    IncreaseSynthesizedNumber();
              ASS.submit(new CoupledRecord(this.AgentID, 'MessageDel', true, SAR) ); }
         else Thread.sleep(time); }}
else { AtomicRecord  ResultRecord=new AtomaticRecord(this.Results, this.ApplicationID, this.TaskID);
       ASS.submit(new CoupledRecord(this.AgentID, 'Put', true, ResultRecord));
                         //the clone agents put results into current ASS
       this.shudown(); }
```

Procedure 8 The semantics of primitive *FullSynthesis*

5 Conclusion

Many WWW applications, *such* as mobile computing, depend upon the autonomy of mobile agents. But autonomous agents need to interact with other agents or *hosting execution environments* for cooperation or resource accesses. In the context of autonomy, interactions should be asynchronous, anonymous and fully localized.

In this paper, we introduced a model; called *Active Sate Spaces (ASS)*, to support distributed environments for autonomous mobile agents. *ASS* abstracts resource descriptions of *hosting execution environments* for mobile agent accesses so as to deal with heterogeneity and dynamicity of distributed information resources. Mobile agents use abstract and unified resources rather than real resources varying from machine to machine. This *technique* decreases code loads carried by mobile agents and makes agents much more compact and easy to design. *ASS* is the third part to facilitate interactions for mobile agents. Mobile agents only need to communicate with local *ASS*, then they can interact with other agents asynchronously and anonymously. So the *ASS* is suitable for interactions between mobile agents and their communication partners in the context of autonomy. Any interaction requirement submitted to an *ASS* will fire a corresponding *Operation* object. The *Operation* objects provide the interaction and the resource access services to mobile agents with security inspections at the same time. We prefer to call this procedure a *Reflection*. All the interaction components such as messages, security rules or reflection operations are integrated into the uniform type of *Record* (either atomic or coupled). Desirable flexibility is obtained by permitting installation, updating and deletion of *Records* in an *ASS* by mobile agents or an *ASS* itself for special purposes. In our mobile agent system *MAT*, the prototype of *ASS* has been implemented and more and more WWW applications have been constructed on *ASS*. Some simulation tests, such as distributed information retrieval in Internet, demonstrated that using ASS can maximize the autonomy of agents in interactions and minimize network communication in low reliability.

The basic framework of the *ASS* for supporting autonomous mobile agents is described in this paper. Our future work will focus on investigating the suitability of *ASS* in WWW applications such as Internet information retrieval, electronic commerce and CSCW (Computer Support Cooperative Work).

Acknowledgments

Authors would like to thank Associate Professor A. S. M. Sajeev for his valuable comments and suggestions, which have been very helpful in improving the quality of this paper.

References

[1] A. Archarya, M. Ranganathan, and J. Saltz, "Sumatra: a Language for Resource Aware Mobile Programs", *Mobile Object System, Lecture Notes in Computer Science*, **No. 1222**, Springer-Verlag, pp. 111-130, February 1997.

[2] J. Baumann, F. Hohl, and N. Radouniklis, "Communication Concepts for Mobile agents", In *Proceedings of the First International Workshop on Mobile Agents*, Lecture Notes in Computer Science, **No. 1219**, Springer-Verlag, pp.123-135, Berlin, April 1997.

[3] L. F. Bic, M. *Fukuda*, and M. B. Dillencourt, "Distributed computing using autonomous objects", *IEEE Computer*, August 1996.

[4] T. Cai, P. Gloor, and S. Nog, "Dartflow: A workflow management system on the Web using transportable agents", Technical *Report TR96-283*, Department of Computer Science, Dartmouth College, Hanover, N.H., 1996.

[5] P. Ciancarini, and V. Rossi, "Jada: Coordination and Communication for Java Agents", *Mobile Object System, Lecture Notes in Computer Science*, **No. 1222**, Springer-Verlag, pp. 213-226, February 1997.

[6] P. Ciancarini, R. Tolksdorf, and F. Vitali, "Coordinating multiagent applications on the WWW: A reference architecture", *IEEE Transactions on Software Engineering*, **Vol. 24**, **No. 5**, pp.362-375, May 1998.

[7] Crystaliz, General Magic, GMD FOCUS, and IBM, "Joint Submission: Mobile Agent Facility Specification", June 1997

[8] E. Denti, and A. Natali, A. Omicini, "On the expressive power of a language for programming coordination media", In *Proceeding of the 1998 ACM Symposium on Applied Computing*, Marriott Marquis, Atlanta, Georgia, U.S.A., February 1998.

[9] P. Domel, A. Lingnau, and O. Drobnik, "Mobile agent interaction in heterogeneous environment", In *Proceedings of* the *1st International Workshop on Mobile Agents*, pp. 136-148, Berlin, Germany, April 1997.

[10] A. Fuggetta, G. P. Picco, and G. Vigna, "Understanding Code Mobility", *IEEE Transactions on Software Engineering*, **Vol. 24**, No. **5**, pp. 342-361, May 1998.

[11] D. Gelernter, "Generative Communication in Linda", ACM Transactions on Programming Languages, **Vol. 7, No. 1**, pp80-112, Jan 1985.

[12] General Magic, "Introduction to the Odyssey API", available at http://www.generalmagic.com /agents/ odysseyIntro.pdf, 1997-1998.

[13] R. Gray, D. Kotz, S. Nog, Daniela Rus, and George Cybenko, "Mobile agents for mobile computing", In *Proceedings of the* Second *Aizu International Symposium on Parallel Algorithms/Architectures Synthesis*, Fukushima, Japan, March 1997.

[14] R.Gray, "Agent Tcl: A flexible and Secure mobile-agent system", In *Proceedings of Fourth Annual Tcl/Tk Workshop*, *Monterey*, California, July 1996.

[15] L. Hurst, P. Cunningham, and F. Sommers, "Mobile agents --- smart messages", In *Proceedings of the First International Workshop on Mobile Agents*, Lecture Notes in Computer Science, **No**. **1219**, Springer-Verlag, pp.111-122, 1997.

[16] N. M. Karnik, and A. R. Tripathi, "Design Issues in Mobile-Agent Programming Systems", *IEEE Concurrency*, pp. 52-61, July-September 1998.

[17] D. Kotz, R. Gray, S. Nog, D. Rus, S. Chawla, and G. Cybenko, "Agent TCL: Targeting the needs of mobile computers", *IEEE Internet Computing*, **Vol**. **1**, **No**. **4**, pp. 58-67, July/August 1997.

[18] W. Li, and M. Zhang, "Distributed Task Plan: A Model for Designing Autonomous Mobile Agents", In *Proceedings of 1999 International Conference on Artificial Intelligence*, Las Vegas, pp. 336-342 1999.

[19] H. Peine, and T. Stolpmann, "The Architecture of the Ara Platform for Mobile Agents", In *Proceedings of the First International Workshop on Mobile Agents*, *Lecture Notes in Computer Science,* **No**. **1219**, Springer-Verlag, pp. 50-61, 1997.

[20] M. Zhang and W. Li, "Persisting Autonomous Workflow for Mobile Agents Using a Mobile Thread Programming Model", *Approaches to Intelligent Agents, Lecture Notes in Artificial Intelligence,* **LNAI Vole 1733**, Springer Verlag Publishers, pp. 84-95, 1999.