Distributed Task Plan:
A Model for Designing Autonomous Mobile Agents

Wei Li
Department of Information Management
Capital University of Economics and Business
Beijing 100026, China

Minjie Zhang
School of Computer and Information Sciences
Edith Cowan University
WA 6027, Australia

Abstract *— In this paper, we present DTP (Distributed Task Plan), a model to depict distributed tasks for executions by mobile agents. DTP is composed of autonomous primitives and can generate the autonomous workflow when a mobile agent executes it. DTP provides mobile agents with navigational and computational autonomies when transporting through the underlying computational network. A mobile agent plans a DTP according to current user's objectives and the knowledge about current network environments when the agent is created. If necessary a mobile agent can also replan its DTP when exceptions occur during the execution of current DTP. The replaned DTP provides the mobile agent with an alternative workflow to accomplish its owner's objectives when current DTP fails. Concentrating DTP, we discuss the autonomies of mobile agents, the autonomous methodology and the implementation mechanism of autonomous mobile agents by DTP in this paper.*
Keywords: distributed task plan, autonomous agents, and mobile agents

1 Introduction

It is necessary for a mobile agent to have the autonomy in many distributed applications such as mobile computing [3, 7]. In the application of mobile computing, 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 "intelligence" of self-contented navigation and computation, which give mobile agents the adaptation powers to the dynamic and heterogeneous network, because in most cases mobile agents can not interact with their owners.

Little has been published on the autonomy of mobile agents although much work has been done concerning the mobile agent systems [4][5][6][9][11]. The details of how some autonomous mobile agent systems [1][5][10][11] work can not be found, and what is an autonomous mobile agent is open now.

In this paper we concentrate on the design of *DTP*, which is a model for generating the autonomous workflow of mobile agents in our mobile agent system *MAT* [8]. *DTP* embodies some degree of autonomy or "intelligence" into mobile agents. Using *DTP*, navigational and computational autonomies are carried by mobile agents as they transport through the underlying computational network. A mobile agent can freely transport and use many different computational resources in a heterogeneous network by executing autonomous primitives in *DTP* without interaction with its owner. In *MAT*, a mobile agent can also replan its *DTP* in any visiting network node when current *DTP* fails due to the dynamicity of the network.

This paper is organized as follows. In Section 2, we give our perspectives to the autonomy for clarifying the meaning of the autonomy of mobile agents. In Section 3, we present the autonomous methodology of *DTP*. The implementation mechanism of *DTP* including autonomous primitives and the planing (replaning) of *DTP* are given in Section 4. Finally, we present our conclusions and directions of future researches in Section 5.

2 Problem Description for the Autonomy of Mobile Agents

2.1 Mobile and computational autonomies

More and more distributed applications, such as mobile computing [3, 7], depend on desirable autonomy of mobile agents. In this paper, The autonomy of mobile agents is conceptualized into two integral parts, the mobile autonomy and the computational autonomy, which can be defined as follows:

Definition1: The mobile autonomy is the capability of self-navigation of mobile agents through the underlying network.

The mobile autonomy of mobile agents should be inherent. Mobile agents can be capable of making decisions about their destinations, navigation modes and where to dock when a destination is not be reachable. the navigation modes could be serial or in parallel. Normally, a mobile agent completes a distributed task by visiting a series of network nodes serially and processing quires and updating 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 concurrent executing subtasks. The solutions of subtasks from different mobile agents are combined into a result finally. A mobile agent should also know where to dock temporarily in order to avoid getting lost or exception to death when a network connection is broken or unavailable, such as the disconnection of the laptop computer from the network.

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

A mobile agent should make use of all kinds of computational resources. A mobile agent can invoke functions in different processes or load functions into its process for the execution when those functions are resident in its visiting network nodes. A mobile agent can also complete a special computational task by executing its own functions carried by the agent when visiting a network node with desirable resources but without necessary computational routines.

2.2 The state of the art

The programming paradigms of many mobile agent systems have similarities, which can be depicted by using the following abstract object-oriented programming structure:

```
public class MobileAgent   {
        IPAddress Home, SiteA, SiteB;
        public void lunch()  {
                Home=GetHomeAddress();
                SiteA=GetIPAddress("SiteA");
                SiteB=GetIPAddress("SiteB");
                MoveTo(SiteA, "atSiteA");      }
        public void atSiteA()  {
                System.out.println("I am at SiteA");
                MoveTo(SiteB, "atSiteB");      }
        public void atSiteB()  {
                System.out.println("I am at SiteB");
                MoveTo(Home, "atHome");    }
        public void atHome()  {
                System.out.println("I come back");
                exit();  }}
```

The limitations of the above paradigm are:

1. Few procedures or primitives are provided for supporting agent autonomies in the mobility and the computation. It is difficult to program a mobile agent with desirable autonomies although it is possible.
2. The above programming paradigm is not for workflow [2] model so it provides no inherent supporting for designing an autonomous agent.
3. A mobile agent and its distributed task are programmed in the same program unit (or class), so both reusability and flexibility are lost. A mobile agent can execute only a distributed task without revising its codes.

In order to overcome the above limitations for designing autonomous mobile agents, we propose the new model, *Distributed Task Plan* (*DTP*).

3 The Autonomous Methodology of *DTP*

3.1 Autonomous primitives: the components of *DTP*

In order to obtain desirable autonomies for mobile agents, firstly, we provide enough programming components, autonomous primitives, which are used to construct *DTP* and further provide mobile agents with autonomies in the navigation and the computation. In *MAT*, four kinds of autonomous primitives are defined for *DTP*.

1. Mobile primitives define the mobility of an agent. A mobile agent can merely transport itself to the next destination from the current network node by calling the simple migration primitive, or clone and transport each of its duplicates to different destinations by calling the multiple migration primitive.

2. Computational primitives define invocations of computational resources. A computational primitive specifies where to find the current computational procedure, how to load it and how to execute it. By using the computational primitives a mobile agent realizes that (a) the current computational procedure is carried by the agent or resident at the visiting node; (b) the procedure should be started in a different process or loaded into its own process; and (c) how to run the computational procedure such as synchronously or asynchronously.

3. Solution synthesis primitives define the combination of multiple solutions from different mobile agents. The solution synthesis is needed when a task is divided into several subtasks and executed by different mobile agents. It is highly efficient to divide a task into several subtasks and assign these subtasks to different mobile agents for the executions when the task can be executed concurrently and multiple resources are available.

4. Control primitives define the execution flow of mobile primitives, computational primitives and solution synthesis primitives. Enough control structures of control primitives are needed to efficiently coordinate the executions of all the above three primitives.

3.2 Primitive references and reuses: the architecture of *DTP*

Having defined primitives, we provide reasonable model *DTP* for depicting distributed tasks of mobile agents by advantages of those pre-defined autonomous primitives. Normally, *DTP* is composed of all the four kinds of autonomous primitives.

Definition 3: The *Distributed Task Plan* (*DTP*) is static description of a distributed task for the execution by a mobile agent.

A *DTP* consists of primitives, which are arranged into two lists. A list, which we prefer to call *Control Queue* (*CQ*), only contains control primitives, and another list, which we prefer to call *Reusable Primitive List* (*RPL*), consists of any primitives except control primitives. The architecture of *DTP* is graphically illustrated in Fig. 1 (Concrete meanings of primitives of Fig. 1 are defined in Section 4).

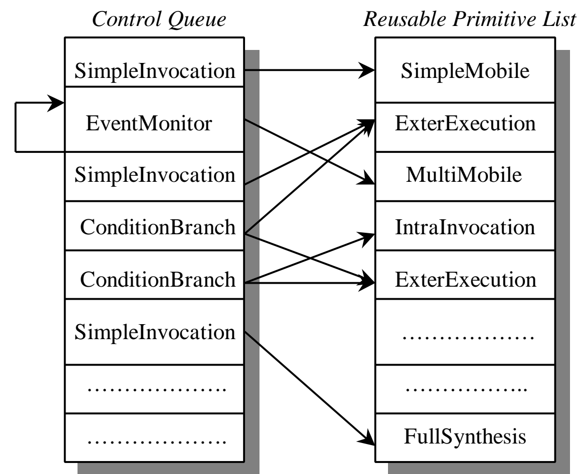

Fig.1 The architecture of *DTP*

3.3 Autonomous workflow: the execution of *DTP*

Definition 4: An execution of *DTP* by a mobile agent in a dynamic network environment is an autonomous workflow of the mobile agent.

The autonomous workflow of a mobile agent is generated when a mobile agent executes its *DTP*. The order of primitives in *CQ* is important. The control primitives in *CQ* are executed sequentially. A control primitive in *CQ* has one or more references to primitives in *RPL* corresponding to which type the control primitive is. A reference of a control primitive in *CQ* depicts a possible invocation to a primitive in *RPL*. The order of primitives in *RPL* is not important because the invocations to them are determined only by references of control primitives in *CQ*. The *RPL* is just a repository of autonomous primitives that a mobile agent maybe needs to execute when transporting in the underlying network. So a mobile agent only executes control primitives in *CQ* one by one, then further execute primitives in *RPL*. An execution of *DTP* is an autonomous workflow of a mobile agent. Constructing a complex workflow by *DTP* provides a mobile agent with autonomy, flexibility and reusability in distributed applications.

4 The Implementation Mechanism of *DTP*

In this section, the implementation mechanism of *DTP* is outlined. All the definitions and procedures are formalized with BNF or a Java-like language.

4.1 Definitions of autonomous primitives

4.1.1 Mobile primitives

The mobile primitives provide two modes of the mobility for mobile agents.

1. *SimpleMobile*

<SimpleMobile>::=<Destination, Action>, where *Destination* is an IP address of a network node and *<Action>::=MobileTo(Destination).*

When a mobile agent encounters a simple mobile primitive, it simply suspends its execution at current network node and transports itself to the next destination node.

2. *MultiMobile*

MultiMobile::=<Desinations, Action>, where Destinations is a set of IP addresses of network nodes. *Suppose Destinations* $=\{D_1, D_2, \ldots\ldots, D_{n-1}, D_n\}$, then Action complies with the following programming paradigm.

```
for each D in {D1, D2, ……, Dn-1}
        {  Agent   DMA=this.Duplicate();
           DMA.MobileTo(D);  }
           MobileTo(Dn)
```

When a mobile agent encounters a multiple mobile primitive, it knows that its distributed task will be executed concurrently on destinations $D_1, D_2, \ldots\ldots, D_{n-1}$, and D_n. The agent (called Parent) duplicates *n-1* its instances with respective subtasks, and each of them will transport to $D_1, D_2, \ldots\ldots$, and D_{n-1} respectively. Finally, the parent agent will transport to the destination D_n.

4.1.2 Computational primitives

The computational primitives provide three ways of the execution of pre-defined computational functions.

1. *ExterExecution*

The 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 absolute or relative path where 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* can be described as follows:

```
Runtime rt=Runtime.getRuntime();
Process proc=rt.exec(FunctionPath);
Results=proc.getInputStream().Read();
if  (Synchronization)  proc.WaitFor();
```

2. *IntraExecution*

The primitive *IntraExecution* permits a mobile agent to invoke a pre-defined computational function for the execution in the same process as that of the mobile agent. This primitive is often used when the invoked function is programmed in the same language as that of the mobile agent.

<IntraExecution> ::= <FunctionURL, Synchronization, Results, Action>, where, *FunctionURL* is the URL where the pre-defined function can be found by the mobile agent. *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* can be described as follows:

```
InputStream is=FunctionURL.OpenStream();
byte bytes[]=is.read();
Class class=DefineClass(bytes);
If (Synchronization)
        Results=class.Instance().run()
else  Thread.new(class.Instance()).start();
```

3. *IntraInvocation*

The primitive *IntraInvocation* permits a mobile agent to invoke a pre-defined computational function for the execution in the same process as that of the mobile agent. This primitive is often used when the invoked function is programmed in the same language as that of the mobile agent and carried by the agent.

<IntraInvocation> ::= <Object, Synchronization, Results, Action>, where *Object* is an object whose method is the invoked function for the execution by the mobile agent. *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* can be described as follows:

```
if (Synchronization) Results=Object.run()
else Thread.new(Object).start();
```

4.1.3 Solution synthesis primitives

Results from different mobile agents are combined into a final result when a distributed task is divided into several subtasks and each of them is assigned to a mobile agent for the concurrent execution. A mobile agent that combines results from other agents into itself will be continuously alive. A mobile agent will exit its execution if its result is combined into another agent.

1. *FullSynthesis*

The *FullSynthesis* primitive combines all the results with the same task identification from different mobile agents with the same agent identification. The parent agent (refer to the *MultiMobile* primitive in Subsection 4.1.1) is responsible for the solution synthesis. The duplicated agents will exit their executions after being synthesized.

<FullStnthesis> ::= <AgentID, TaskID, Results, Action>, where *AgnetID* is the agent identification, *TaskID* is the task identification of the subtasks which solutions should be synthesized and *Action* can be described as follows:

```
if (isParent())
{while SynthesizedNumber()<SubtaskNumber()
 { Agent DMA=FindDMA(AgentID, TaskID)
   if (DMA!=null)
   {Results=MergeWith(DMA.getResults(TaskID));
    DMA.Shutdown();
    IncreaseSynthesizedNumber();}
   else Thread.sleep(time) ;}}
else WaitforBeingSynthesized();
```

2. *PriSynthesis*

When a mobile agent encounters a *PriSynthesis* primitive and the agent is the first one to accomplish this special subtask identified by task identification, the agent will be continuously alive, otherwise it will exit its execution.

<PriSynthesis> ::= <AgentID, TaskID, Action>, where *AgnetID* is the agent identification, *TaskID* is the task identification of the subtasks which solutions should be synthesized, and *Action* can be described as follows:

```
if (FindPriSynthesisSign(AgentID, TaskID)=null )
        CreatePriSynthesisSign(AgentID, TaskID)
else Shutdown();
```

4.1.4 Control primitives

The control primitives are used to construct the complex execution flow of all the above three primitives.

1. *SimpleInvocation*

The *SimpleInvocation* primitive only invokes another primitive to which it has a reference.

<SimpleInvocation> ::= <Primitive, Action>, where *Primitive* is a reference to another primitive object in *RPL*, and *Action* can be described as follows:

```
Primitive.run();
```

2. *EventMonitor*

The *EventMonitor* primitive 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* is any primitive object in *RPL*. *Action* can be described as follows:

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

3. *ConditionBranch*

The *ConditionBranch* primitive checks a condition so as to decide which primitive in *RPL* should be executed by next step.

<ConditionBranch> ::= <Condition, Primitive1, Primitive2, Action>, where *Condition* is the condition that the *ConditionBranch* checks, *Primitive1* and *Primitive2* are primitives in *RPL*, and Action can be described as follows:

```
if (Condition) Primitive1.run() else Primitive2.run()
```

4.2 Planing and replaning of *DTP*

A mobile agent plans its own *DTP* for the execution of a distributed task satisfying user's requirements when the agent is generated. The process of planning is graphically illustrated in Fig.2.

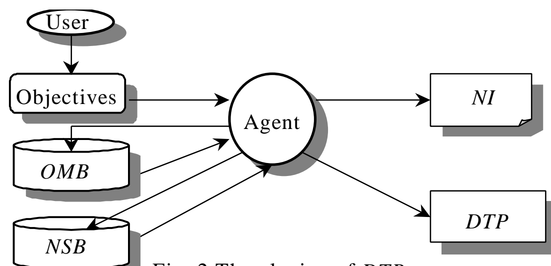

Fig. 2 The planing of *DTP*

The mobile agent matches the user objectives with entries in *Objective Matching Base* (*OMB*) to decide which primitives are used. The selected primitives are arranged into *CQ* or *RPL* according to their types. The references of primitives in *CQ* to primitives in *RPL* are set up at the same time. The developers of *MAT* applications define *OMB*. An *OMB* consists of reusable entries, and the matching method uses the algorithm that we prefer to call *Requirement Propagation* to match user objectives with pre-defined entries, which determine the *DTP* planning. The mobile agent also retrieves *Network State Base* (*NSB*) for *Network Information* (*NI*) such as IP addresses of home machine and docking machines. The retrieved *NI* will be carried by the mobile agent for the future use when traversing in the network. *NI* is necessary for a mobile agent to replan its *DTP* when it fails with its goals at a visiting network node, such as resources are unavailable or a destination is not reachable. The *NSB* is periodically updated by *MAT* information server in order to reflect the dynamic networks. A concrete example of *DTP* planning is graphically illustrated in Fig. 3. In this example a user wants to get a file *ProgramTable* from remote network node *butterfly* when the file is updated after *February 22, 1999*.

Each entry of *OMB* has four components, an entry name with parameters, *Operation*, *PreOperation* and *PostOperation*. *Operation* is executed after the execution of *PreOperation* and before the execution of *PostOperation*. The executions of *Operation*, *PreOperation* or *PostOperation* can possibly access other entries, so the objective matching is the *Requirement Propagation* by reusing entries in *OMB*. In the above example, the objective matching is in the order of 1-2-3-4-5-6 and the *DTP* planning is in the order of (a)-(b)-(c)-(d). The workflow generated by executing *DTP* will be:

Step 1: Transporting to *butterfly*,
Step 2: Monitoring file *ProgramTable* until it is updated after *February 22, 1999,*
Step 3: Reading the file, and
Step 4: Taking the file back the home machine.

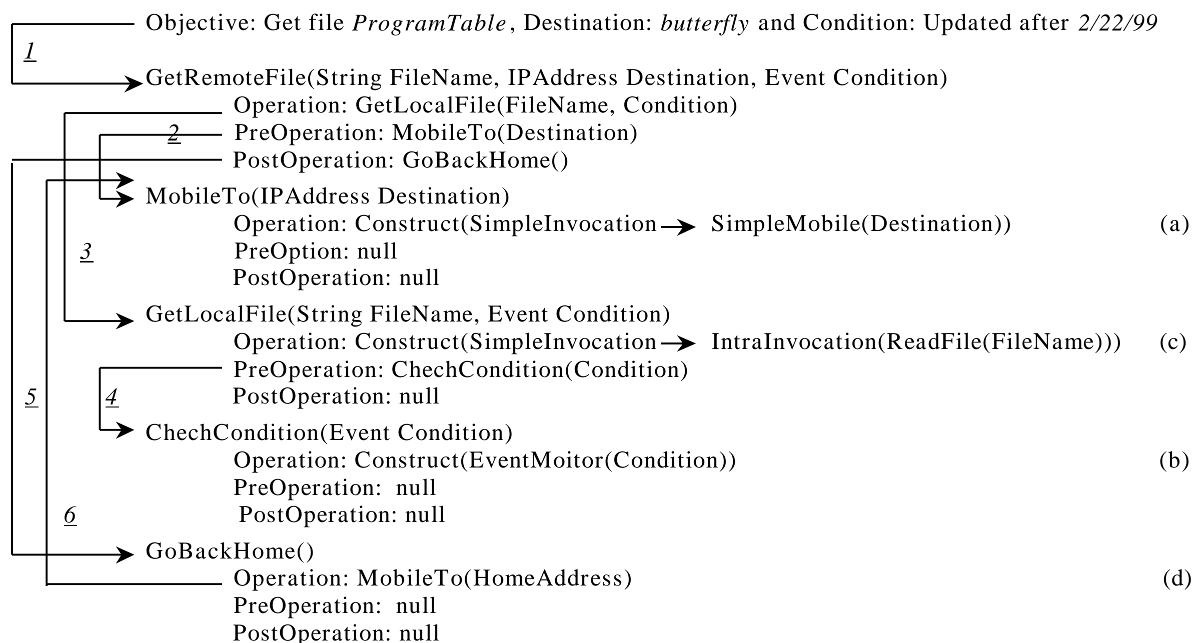

Fig. 3 DTP planing: objectives matching with reusable entries in *OMB*

The replaning of *DTP* is the same as the planing of *DTP* but the agent uses the *OMB* and the *NSB* of the visiting nodes rather than those of the home machine. The replaning of *DTP* gives a mobile agent an alternative workflow to accomplish its owner's objectives when current *DTP* fails.

5 Conclusion

In mobile agent system *MAT*, the *DTP* of mobile agents carries some amount of behavioral information of the navigation and the computation, so the mobile agents can be considered as having a certain degree of autonomies through heterogeneous networks. *DTP* has the advantages for programming a mobile agent with many transportable behaviors by a few mobile primitives, programming a mobile agent with many computational powers by a few computational primitives, getting a complex workflow of distributed processing with a simple programming paradigm, and obtaining reusability, flexibility and expandability at the same time. The basic framework for designing an autonomous mobile agent is described in this paper. Our future work will focus on investigating the suitability of *DTP* in WWW applications such as Internet information retrieval, electronic commerce and CSCW (Computer Support Cooperative Work). From feedback of future investigations, we may find problems in *DTP* and make improvements to both the model and its implementation.

Acknowledgements

I would like to thank Prof. Zhongzhi Shi for his valuable supports and comments to this paper.

References

[1] Lubomir F. Bic, Munehiro Fukuda, and Michael B. Dillencourt, Distributed computing using autonomous objects, *IEEE Computer*, August 1996.

[2] Ting Cai, Peter Gloor, and Saurab 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.

[3] Robert S. Gray, David Kotz, Saurab 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.

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

[5] Leon Hurst, Pádraig Cunningham, and Fergal Sommers, Mobile agents --- smart messages, In *Proceedings of the 1st International Workshop on Mobile Agents*, Berlin, Germany, April 1997.

[6] Jeremy Hylton, Ken Manheimer, Fred L. Drake, Jr., Barry Warsaw, Roger Masse, and Guido van Rossum, Knowbot programming: System support for mobile agents, In *Proceedings of the Fifth International Workshop on Object Orientation in Operating Systems*, pp. 8-13, Seattle, Wash., October 1996.

[7] David Kotz, Robert Gray, Saurab Nog, Daniela Rus, Sumit Chawla, and George Cybenko, Agent TCL: Targeting the needs of mobile computers, *IEEE Internet Computing*, 1(4):58-67, July/August 1997.

[8] Wei Li, The study and applications of mobile agents, *PhD thesis*, Institute of Computing Technology, Chinese Academy of Sciences, July 1998.

[9] Object Space, Voyager Core Technology 2.0 User Guide, available at http://www.jectspace.com/developers/voyager/white/voyager20.pdf, 1998.

[10] Daniela Rus, Robert S. Gray, and David Kotz, Transportable information agents, In *Proceedings of International Conference on Autonomous Agents*, pp. 228-236, February 1997.

[11] Markus Straßer, Joachim Baumann, and Fritz Hohl, Mole -- a Java based mobile agent system, In *2nd ECOOP Workshop on Mobile Object Systems*, pp. 28-35, Linz, Austria, July 1996.