Comparative Efficiency and Implementation Issues of Itinerant Agent Language on Different Agent Platforms

Abdullah Almuhaideb[1,3] Kutila Gunasekera[1] Arkady Zaslavsky[1] Seng Wai Loke[2]
[1]Faculty of Information Technology
Monash University
Melbourne, Australia
[2]Department of Computer Science and Computer Engineering
La Trobe University
Melbourne, Australia

aalmuhaidob@kfu.edu.sa, {kutila.gunasekera, arkady.zaslavsky}@infotech.monash.edu.au, S.Loke@latrobe.edu.au

ABSTRACT

An itinerary scripting language provides a tool to accelerate the development of mobile agent applications. The main purpose of this paper is to discuss research challenges of itinerant mobile agents (ITAG) and describe the migration of ITAG to JADE, the popular FIPA-compliant agent platform. The migration process was based on two major steps: the migration of agent communications and agent tasks (behaviors). The ITAG Engine has been extracted to provide an easy migration to any new agent platform. In our experiments the results show that the current ITAGIII (using JADE) has better performance and stability compared to ITAGII (using Grasshopper). The paper discusses the experiments and comparisons in detail.

Categories and Subject Descriptors

I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence – *Multiagent systems*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems

General Terms

Algorithms, Measurement, Performance, Design, Languages, Experimentation, Theory

Keywords

Itinerary Language, Itinerary Agent (ITAG), JADE, Grasshopper, Agent Platforms, Scalability, Applications, Agent Communication, Agent, Messaging

1. INTRODUCTION

Mobile agents are defined as a future framework of distributed electronic services and are used in data mining, electronic commerce and network management [1]. All stages of a business transaction, such as negotiating and signing contracts can be carried out using mobile agents. A mobile agent can be described as a software entity which is capable of moving from one location to another and continuing its execution. ITAG (ITinerary AGents) is a scripting language and prototype system previously developed by us [2, 3, 4]. An itinerary scripting language (for mobile agents) aims to make developing agent applications easier. It allows the developer to script together mobile agents from existing components by specifying what the agent should do, at which location and when.

The purpose of this paper is to extend the Itinerary Language and port the current ITAG system which runs on Grasshopper agent platform to the popular FIPA (Foundation for Intelligent Physical Agents)-compliant agent platform JADE [5]. We also developed a visual tool which allows users to easily create and execute their own test cases and demonstration environments for ITAG.

There are a number of reasons behind choosing JADE. One of these reasons is that JADE is open-source, continuously maintained and well supported [6]. Since ITAG aims to support mobile devices, the existence of JADE-LEAP (Lightweight Extensible Agent Platform) [6], a lightweight release of JADE for mobile devices, was also a consideration.

The rest of this paper is organized as follows. In section 2, an overview of the ITAG system will be given. Section 3 will discuss the migration process from Grasshopper to JADE. In section 4, we will present new features of ITAGIII. In section 5, the experiments and result will be evaluated. Some related work will be discussed in section 6. Finally, conclusions and future work of this paper will be presented in section 7.

2. ITAG SYSTEM OVERVIEW

2.1 ITAG: The Itinerary Scripting Language

ITAG is an executable implementation of the itinerary algebra in the form of a scripting language described in [2, 4]. We first briefly outline the algebra below. We assume an object-oriented model of agents, where an agent is an instance of a class given roughly by:

mobile agent = state + action + mobility

We assume that agents have the capability of cloning, which is, creating copies of themselves with the same state and code. Also, agents can communicate to synchronize their movements, and the agent's code can execute at each location it visits.

[3]Abdullah Almuhaideb is a staff member at the Department of Network Computing, King Faisal University in AlAhsa, Saudi Arabia. This work was done while he was visiting Monash University.

Jung, Michel, Ricci & Petta (eds.): *AT2AI-6 Working Notes, From Agent Theory to Agent Implementation, 6th Int. Workshop,*May 13, 2008, AAMAS 2008, Estoril, Portugal, EU.
Not for citation

Let **A**, **O** and **P** be finite sets of agent, action and place symbols, respectively. Itineraries (denoted by *I*) are now formed as follows representing the null activity, atomic activity, parallel, sequential, nondeterministic, conditional nondeterministic behavior, and have the following syntax:

$$I :: 0 \mid A^a_p \mid (I \parallel_\oplus I) \mid (I \,.\, I) \mid (I \mid I) \mid (I :_\Pi I)$$

where $A \in \mathbf{A}$, $a \in \mathbf{O}$, $p \in \mathbf{P}$, $\oplus$ is an operator which, after a parallel operation causing cloning, recombines an agent with its clone to form one agent, and Π is an operator which returns a boolean value to model conditional behavior. We specify how ⊕ and Π are used but we assume that their definitions are application-specific.

We assume that all agents in an itinerary have a starting place (which we call the agent's home) denoted by $h \in \mathbf{P}$. Given an itinerary *I*, we shall use *agents*(*I*) to refer to the agents mentioned in *I*.

Agent Movement (A^a_p). A^a_p means "move agent *A* to place *p* and perform action *a*". This expression is the smallest granularity mobility abstraction. It involves one agent, one move and one action at the destination.

Parallel Composition ("‖"). Two expressions composed by "‖" are executed in parallel. For instance, ($A^a_p \parallel B^b_q$) means that agents A and B are executed concurrently. Parallelism may imply cloning of agents. For instance, to execute the expression ($A^a_p \parallel A^b_q$), where $p \neq q$, cloning is needed since agent *A* has to perform actions at both *p* and *q* in parallel. When cloning has occurred, decloning is needed, i.e. clones are combined using an associated application specific operator (denoted by ⊕ as mentioned earlier).

Sequential Composition ("."). Two expressions composed by the operator "." are executed sequentially. For example, ($A^a_p \,.\, A^b_q$) means move agent *A* to place *p* to perform action *a* and then to place *q* to perform action *b*.

Independent Nondeterminism ("|"). An itinerary of the form (*I* | *J*) is used to express nondeterministic choice: "I don't care which but perform one of *I* or *J*". If $agents(I) \cap agents(J) \neq 0$, no clones are assumed, i.e. *I* and *J* are treated independently. It is an implementation decision whether to perform both actions concurrently terminating when either one succeeds (which might involve cloning but clones are destroyed once a result is obtained), or trying one at a time (in which case order may matter).

Conditional Nondeterminism (":"). Independent nondeterminism does not specify any dependencies between its alternatives. We introduce conditional nondeterminism which is similar to short-circuit evaluation of boolean expressions in programming languages such as C. An itinerary of the form $I :_\Pi J$ means first perform I , and then evaluate Π on the state of the agents. If Π evaluates to true, then the itinerary is completed. If Π evaluates to false, the itinerary J is performed (i.e., in effect, we perform *I* . *J*). The semantics of conditional nondeterminism depends on some given Π.

2.2 Itinerary Language Examples and Implementation

We give an example using agents to vote. An agent V, starting from home, carries a list of candidates from host to host visiting each voting party. Once each party has voted, the agent goes home to tabulate results (assuming that home provides the resources and details about how to tabulate), and then announces the results to all voters in parallel (and cloning itself as it does so). Assuming four voters (at places *p, q, r,* and *s*), vote is an action accepting a vote (e.g., by displaying a graphical user interface), tabulate is the action of tabulating results, and announce is the action of displaying results; the mobility behavior is as follows:

$$V^{vote}_p \cdot V^{vote}_q \cdot V^{vote}_r \cdot V^{vote}_s \cdot V^{tabulate}_h .(V^{announce}_p \parallel V^{announce}_q \parallel V^{announce}_r \parallel V^{announce}_s)$$

Implementation: To allow the programmer to type the itinerary expressions into the computer, we provide an ASCII syntax and a Controlled English (limited natural language) version. The translations are given in Table 1. When the operators are used without op, we assume a pre-specified system default one, i.e. using op is an optional clause. $A^a_p \,.\, A^b_q .\, A^c_r$ can be described as follows: "(move A to a do p) then (move A to b do q) then (move A to c do r)." Apart from the above basic elements of the language, we define the following five phrases that map down to more complex expressions:

1. A^a_h is translated as `return A do a`.
2. $A^a_p \,.\, A^a_q . A^a_r .\, A^a_s$ is translated as
`tour A to` *p, q, r, s* `in series do a`.
3. $A^a_p \parallel A^a_q \parallel A^a_r \parallel A^a_s$ is translated as
`tour` *A* `to` *p, q, r, s* `in parallel do` *a*.
4. $A^a_p \mid A^a_q \mid A^a_r \mid A^a_s$ is translated as
`tour A to one of` *p, q, r, s* `do a.`
5. $A^a_p : A^a_q : A^a_r : A^a_s$ is translated as
`tour A if needed to` *p, q, r, s* `do a.`

Table 1. Translations

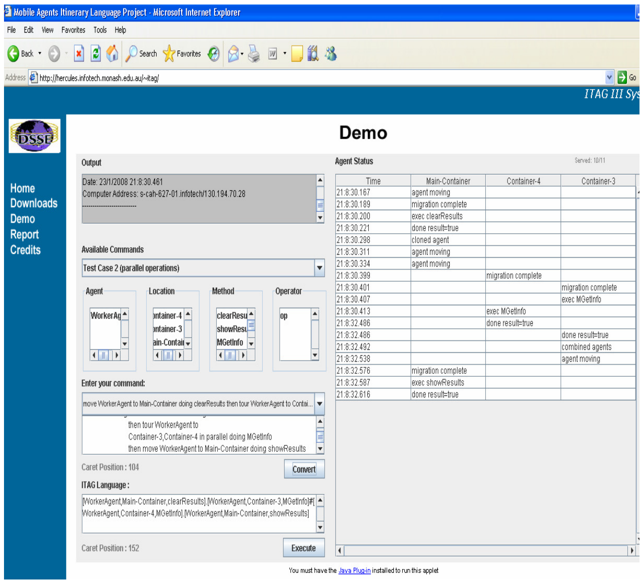
Symbol	ASCII	Controlled English
A^a_p	[A,p,a]	Move A to P do a
.	.	Then
$:_\Pi$	:{op}	Otherwise using op
|	|	Or
$\parallel_\oplus$	#{op}	In parallel with using op

Similarly, we also have $A^a_p :_\Pi A^a_q :_\Pi A^a_r :_\Pi A^a_s$ translated as `tour` *A* `if needed to` *p, q, r, s* `do` *a* `using` Π.

Using the phrases, the voting itinerary can be concisely described as follows:

```
(tour V to p,q,r,s in series do vote)
 then (return V do tabulate)
 then  (tour  V  to  p,q,r,s  in  parallel  do
announce)
```

ITAG implementation is in the Java programming language, previously built on top of Grasshopper (ITAG and ITAGII) and now JADE (ITAGIII) agent toolkits. In all implementations, the user first types in itinerary scripts into an applet (running in a Web browser). Then, the itinerary script is parsed into a binary tree representation and executed by an interpreter. Execution is as follows: the interpreter translates the actions specified in the script into commands which are then forwarded to agents which are initially at a place (the home). These agents on receiving the commands are then launched into the agent network to do their work. Figure 1 shows an example of the ITAG demo user interface. We explain the ITAG system architecture in the next section.


Mobile Agents Itinerary Language Project - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://hercules.infotech.monash.edu.au/~itag/
Go
ITAG III Sys
DSSE
Home
Downloads
Demo
Report
Credits
Convert
Execute

Demo

Output

```
Date: 23/1/2008 21:8:30.461
Computer Address: s-cah-627-01.infotech/130.194.70.28
-------------------------
```

Available Commands: Test Case 2 (parallel operations)

Agent: WorkerAg

Location: ontainer-4, ontainer-3, ain-Contai

Method: clearResu, showResu, MGetInfo

Operator: op

Enter your command:

```
move WorkerAgent to Main-Container doing clearResults then tour WorkerAgent to Contai...
                then tour WorkerAgent to
                Container-3,Container-4 in parallel doing MGetInfo
                then move WorkerAgent to Main-Container doing showResults
```

Caret Position : 104

ITAG Language :

```
[WorkerAgent,Main-Container,clearResults].[WorkerAgent,Container-3,MGetInfo]#[
WorkerAgent,Container-4,MGetInfo].[WorkerAgent,Main-Container,showResults]
```

Caret Position : 152

Agent Status — Served: 10/11

Time	Main-Container	Container-4	Container-3
21:8:30.167	agent moving		
21:8:30.189	migration complete		
21:8:30.200	exec clearResults		
21:8:30.221	done result=true		
21:8:30.298	cloned agent		
21:8:30.311	agent moving		
21:8:30.334	agent moving		
21:8:30.399		migration complete	
21:8:30.401			migration complete
21:8:30.407			exec MGetInfo
21:8:30.413		exec MGetInfo	
21:8:32.486		done result=true	
21:8:32.486			done result=true
21:8:32.492			combined agents
21:8:32.538			agent moving
21:8:32.576	migration complete		
21:8:32.587	exec showResults		
21:8:32.616	done result=true		

You must have the Java Plugin installed to run this applet

Figure 1. An example of ITAG demo user interface in action

2.3 ITAG System Architecture

The ITAG system is broken down into a number of distinguishable modules that handle different functionalities of the system. Figure 2 below illustrates these different modules and their interactions with each other.

Users of the ITAG system interact with it through the User Interface. The UI allows users to configure an itinerary using a limited natural language format. The UI also contains a separate area which shows the results of itinerary execution (Agent status and Output panes in Figure1). The UI itself does not understand the ITAG language and cannot execute an itinerary. A user can configure an itinerary in the limited natural language format in the User Interface and have it converted to an itinerary language statement. This conversion is done by the *ITAG Parser* module.

When the user requests the itinerary to be executed, the itinerary is passed on to the Controller Agent for execution.

ITAG Parser is part of the ITAG API and is independent of the underlying agent platform. It parses a user configured itinerary from its user-friendly limited natural language format to the itinerary language format (ASCII format).

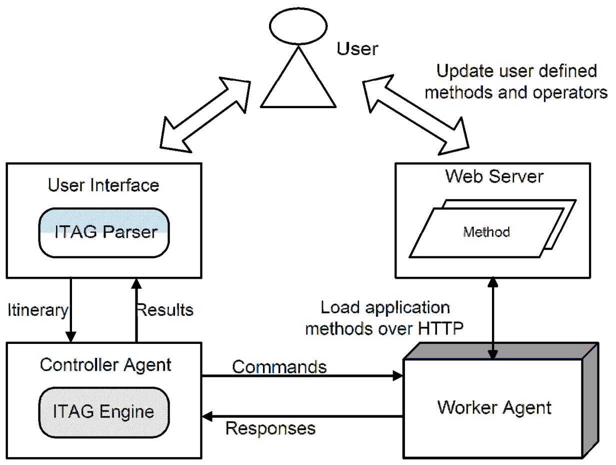


Figure 2. System Architecture

The *ITAG Engine* is the core of ITAG and contains the logic of the Itinerary Language. It also forms part of the ITAG API and is independent of the underlying agent platform. The engine provides the necessary components to execute an itinerary and the Controller Agent uses the *ITAG Engine* to understand and execute an itinerary. The Engine takes as input an itinerary. As output the engine makes method calls to carry out the tasks specified in the itinerary.

The ITAG system consists of two types of agents: *Controller Agents* and *Worker Agents*. Worker Agents are created and controlled by Controller Agents. During the execution of an itinerary Worker Agents maybe created and destroyed afterwards. A system may consist of multiple Controller Agents. These two types of agents are described below.

The *Controller Agent* gets as input an itinerary from the User Interface. Its main functionality is executing this itinerary by driving worker agents accordingly with the help of the ITAG Engine. The Controller Agent is the controller and executor of an itinerary.

The *Worker Agents* carry out application-specific useful functionality requested by users from the ITAG system. A worker is a simple agent controlled by another (controller) agent. It only responds to commands from the controller agent and is not aware of itineraries or the ITAG language. The worker agent is able to do the following tasks when requested by its controller:

- move or copy itself to different locations
- execute a method (by downloading its class files from a pre-defined location)
- store any results it gathers by executing various methods in its “pocket”
- combine “pockets” with other worker agents
- destroy itself

The web server provides a place to host the User Interface (e.g. applet) and the user defined method and operator classes. Methods are application specific code to be executed by ITAG agents (i.e. worker agents) which are represented as *actions* in the itinerary language. They contain useful functionality written by application developers and added to the system dynamically.

Operators (represented by ⊕ and Π in the itinerary language) can also be dynamically added to the web server for use by the ITAG system.

3. MIGRATION PROCESS FROM GRASSHOPPER TO JADE PLATFORM

The migration of ITAG from Grasshopper to JADE consists of two main stages. The first is porting the agent communication and the second, agent tasks (behaviors). The communication and behavior mechanisms of JADE lead to ease of development and better performance in comparison to Grasshopper.

3.1 Agent Communication

Agent communication which is a fundamental feature of an agent platform describes how two agents converse. In Grasshopper, the communication between agents is through their proxies. But in JADE, agents communicate through message passing as asynchronous agent messages [5]. Proxies do not exist in JADE; instead, an agent searches the current location of its target by querying the AMS (Agent Management Service) according to the FIPA specifications. The Agent Management Service gives JADE a better communication hub compared to Grasshopper. The region server in Grasshopper could become a bottleneck, as it must update every proxy just before being used [7]. Based on our implementation we found that JADE has an excellent control over cloned agents in terms of keeping references of these agents and destroying them at the end of their tasks whereas in Grasshopper some cloned agents could be left without being killed after task completion.

3.2 Agent Tasks (Behaviors)

In JADE, the agent is allowed to have just a single Java thread per-agent [5]. Inside this thread multiple behaviors can be added using a round-robin non-preemptive scheduling policy [8]. Grasshopper makes use of the more complicated alternative of implementing a multi-threaded system to handle, for example, socket connections and communication processes. The JADE behaviors improve agent performance as the switching between behaviors is far faster than switching between Java threads. Another advantage of the JADE behaviors against multi-threaded systems is its elimination of all synchronization issues between parallel behaviors accessing the same resources since all behaviors are executed by the same Java thread which result in a performance enhancement as well. [5, 8, 9]

4. ITAGIII NEW FEATURES

With ITAGIII, an improved GUI has been introduced to provide simplicity to the system. The first screen that a user encounters in the demo system lists the existing applications in the server that an ITAG agent is capable of executing. An application, in ITAG, is a collection of methods that collaborate together to provide a complete service to the user. Also an intermediate page has been introduced as an option in case an application may require further user input to run this application. For example, an intermediate page has been introduced in "make an appointment" application in order to collect an appointment preferred times of the user.

Also in ITAGIII, the problem of nondeterministic activity ("|") implementation has been fully solved and full control of the clone agents has been achieved. The implemented semantics carry out all activities in parallel and select the first one to finish (others are discarded). The complexity of killing other agents and threads was the reason for non-implementation in the previous version of ITAG under Grasshopper. (i.e. The original agent could be killed under nondeterministic activity but with the clone agents left alive).

Another feature of ITAGIII is the ITAG Engine which has been extracted from ITAGII version. The ITAG Engine provides for easy development of an ITAG system on any new mobile agent platform. The Controller Agent for the new agent platform has to implement an interface `itagIII.engine.AgentDealings` which contains methods representing the output commands of the ITAG Engine. We give the AgentDealings interface below followed by figure 3 which shows the ITAG Engine and its relation to other components.

```
public interface AgentDealings {
   public boolean go(String agentId,
      String tempLocation) throws Throwable;
   public boolean exec(String agentId,
      String method, String
      dynamicpath) throws Throwable;
   public String cloning(String agentId)
      throws Exception;
   public void combine(String source,
      String destination, int ativityNum)
      throws Exception;
   public Vector getAgentPocket(
      String source)throws Exception;
   public void showAgentPocket(
      String agentId)throws Exception;
   public void removeAgent(String agentName)
      throws Exception;
   public String yourLocation(
      String agentName) throws Exception;
   public boolean existAgent(
     String agentId);
   public boolean existLocation(
     String agentId);
}
```

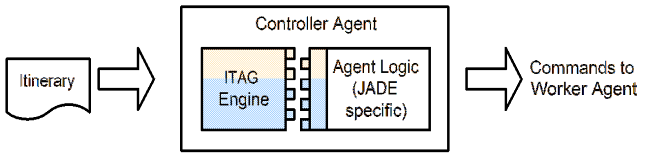


Figure 3. ITAG Engine and its relation to other ITAG system components

5. COMPARISON OF ITAGII AND ITAGIII PERFORMANCES

The aim of this experiment is to compare the performance of the two versions of ITAG system on JADE (ITAGIII) and Grasshopper (ITAGII) agent platforms.

5.1 Experiment Set-up

5.1.1 Test Scenarios

The basic test scenario is explained in this section. First, we give below an itinerary encapsulating all the test cases.

```
move WorkerAgent to Home doing clearResults
  then tour WorkerAgent to [one of]
  Location-1, Location-2, Location-3 ...
```

```
[in series|in parallel|if needed]
    doing getInfo
then move WorkerAgent to Home doing
    showResults
```

The controller agent communicates first with the worker agent to return home and run the clearResults method, which will clear the worker agent's pocket. Then the controller agent communicates with the worker agent to move to Location-1, Location-2 and Location-3 running the getInfo method. getInfo method will retrieve the date and time of the location of the agent as well as the machine name and IP address. Also getInfo method returns true or false randomly in order to test nondeterministic and conditional nondeterministic activities. Finally the controller agent communicates with the worker agent to return home running the showResults method. This method will display the agent's pocket in the user interface as well as writing the result to a log file.

Different test scenarios are realized through change of parameters, which are explained below.

- *Number of destinations* - The most likely scenario for an agent is to travel to a number of locations. Therefore, in our test we examine the worker agent in a different number of locations starting with two and increasing up to six locations.
- *Type of activities* – The four types of ITAG activities to be tested are: sequential, parallel, nondeterministic and conditional nondeterministic.

5.1.2 Test Environment and Measurement

The experiment is repeated for every activity, with the locations on the same host and on different hosts. Due to limitations with ITAGII (under Grasshopper), the system was not tested for nondeterministic activity, and also the system could not be tested with the different locations on different hosts. The testing environments for a number of locations located on different hosts were four PC's: one hosted the Controller Agent and the JADE main container and the other three hosted the six locations with two containers for every host. Table 2 shows hardware and software configuration of the test PC's. All PC's were connected to a 100Mbps network. To gain accurate results, each experiment was repeated ten times and the average taken. The performance was measured for every activity based on the number of locations and the average time taken to complete the test.

Table 2. Hardware and Software of the testing environment

Processor	Intel Pentium 4 CPU 3.00 GHz
Memory	1GB (SDRAM)
Operating System	Windows XP Professional Version 2002 Service Pack 2
Java version	Sun JDK 1.5.0_12
JADE version	3.5
Grasshopper version	2.2.4

5.2 Results

This section presents the results based on ITAG system activities. Each activity section will contain two figures. The first one will compare ITAGII and ITAGIII by plotting the average time versus number of locations in the itinerary. All locations for this test are on the same host. The second figure shows a similar graph but compares ITAGIII (JADE based agents) with the locations physically distributed on four different hosts and on the same host.

5.2.1 Sequential Activity

The result of this experiment shows that ITAGIII has a better performance than ITAGII. As figure 4 illustrates, an ITAGIII agent was consistently able to finish its itinerary task faster than ITAGII. The performance of ITAGII degrades rapidly with increasing number of locations compared to ITAGIII. For example, with four locations we see ITAGIII is seven times faster and with six locations it is six times faster.

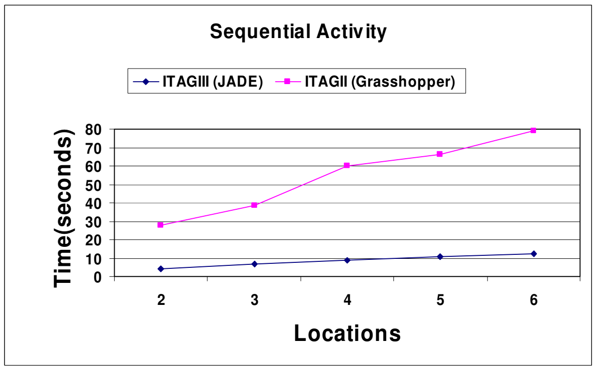


Figure 4. Sequential activity, comparing ITAG system with JADE and Grasshopper based agents on the same host

When comparing ITAGIII under JADE based agents with different number of locations on the same host and on different hosts (Figure 5) we see no significant variation in the time taken. Since the experiments were conducted on a high-speed LAN with low traffic, communication overheads were negligible. This indicates that ITAGIII does not incur an extra overhead when the agents communicate and move between multiple machines. As future work we intend to conduct experiments in heterogeneous wide-area network environments using agents with larger workloads.

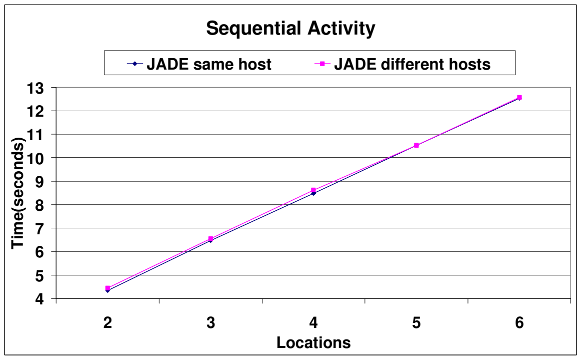


Figure 5. Sequential activity, comparing ITAG system with JADE based agents on the same host and on different hosts

5.2.2 Parallel Activity

The parallel activity scenario test in figure 6 shows that, as in the sequential activity, that ITAGIII has a better performance than ITAGII (six times faster on 3 locations). However, on ITAGII the agent was unable to continue its tasks with more than 3 locations to be visited in parallel. This is due to ITAGII's issues with managing multiple clones which was previously explained. Also this indicates that ITAGIII is more stable, scalable and efficient than ITAGII.

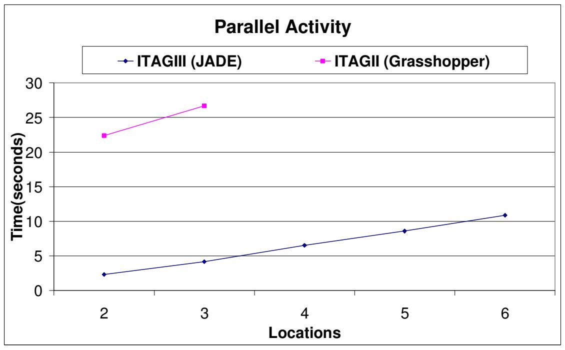


Figure 6. Parallel activity, comparing ITAG system with JADE and Grasshopper based agents on the same host

Figure 7 shows that ITAGIII has the same performance with the locations on the same host and on different hosts, which support the same conclusion from the sequential activity.

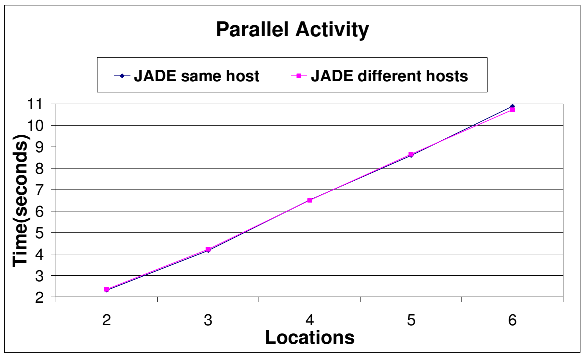


Figure 7. Parallel activity, comparing ITAG system with JADE based agents on the same host and on different hosts

5.2.3 Nondeterministic Activity

In this test, the worker agent clones itself based on the number of locations and sends them to the different locations to execute the tasks. When one of the agents finishes its task that agent's result is taken and all the clones destroyed. We can see from figure 8 that ITAGIII on different machines has a slightly better and stable performance than on the same machine. We believe this behaviour could be due to the distribution of the computation load between multiple CPUs with a fast network connecting them. A similar argument is put forward in [8].

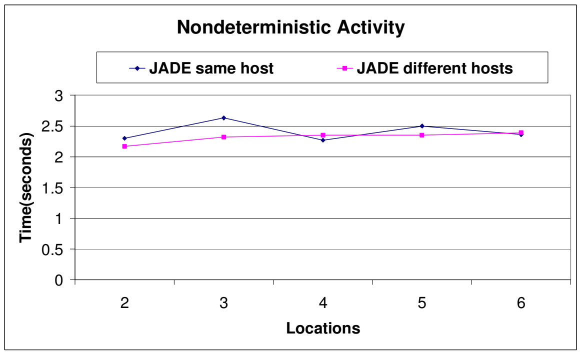


Figure 8. Nondeterministic activity, comparing ITAG system with JADE based agents on the same host and on different hosts

5.2.4 Conditional Nondeterministic Activity

When running this test multiple times the number of locations the worker agent travels to (in sequence) fluctuates based on the result returned by the getInfo method. For example, the agent travels from one location to another as long as the getInfo method returns false, otherwise the agent will skip the rest of locations and return home to show the result. In the graphs below we show the average times taken to complete the test. Figure 9 supports the previous test results and shows that ITAGIII has better performance than ITAGII.

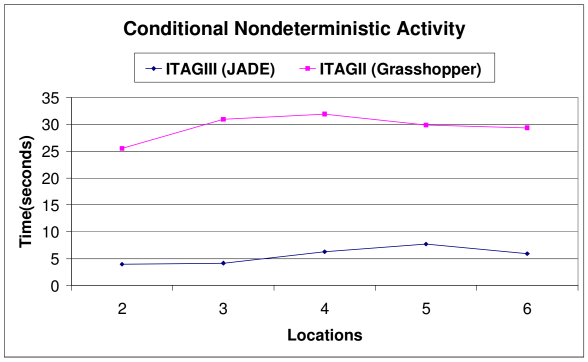


Figure 9. Conditional Nondeterministic activity, comparing ITAG system with JADE and Grasshopper based agents on the same host

In previous tests we saw that the performance of JADE on same host versus multiple hosts is similar, but in figure 10 we do not see this relationship because of the random behavior explained above.

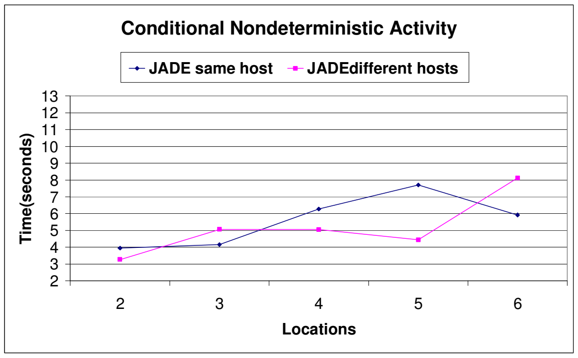


Figure 10. Conditional Nondeterministic activity, comparing ITAG system with JADE based agents on the same host and on different hosts

6. RELATED WORK

In this section we first briefly look at several itinerary languages found in literature. Lu and Xu defined a mobile agent itinerary language (MAIL) [10] which has been implemented as a feature in the Naplet [11] mobile agent system. Their work however is limited to a single mobile agent platform (Naplet) while ITAG can be easily implemented on any mobile agent platform by using the ITAG Engine. Performance figures for MAIL based agents were also not available for comparison with ITAG.

Rech et al. in [12, 13, 14] describe a flexible itinerary that can be adjusted at run-time to ensure greater fault tolerance. In ITAG with the activity $A^{a}{}_{l}$ the agents have some flexibility by specifying the place of execution as a result of some function "*l*". Also, a branch of ITAG [15] with the concept of "Goals" introduces a degree of flexibility to the itinerary.

Finally, we consider the Itinerary Graph [16] system. In their work the migration strategies which are used to perform the mobile agent's actions are *sequential*, *parallel* and *selective*. The *selective* strategy is similar to ITAG's *Conditional Non-determinism*. The lack of a strategy equivalent to *Independent Non-determinism* is a limitation in Itinerary Graphs.

In terms of platform comparisons, we find that previous work such as [7, 8, 9] give general discussions of the performance and efficiency of JADE and Grasshopper. However, in this paper we compare them in a more specific manner through the implementation of ITAG and its four behaviours, namely *sequential, parallel, Independent Non-determinism* and *Conditional Non-determinism.*

7. CONCLUSIONS AND FUTUREWORK

This paper describes an implementation of ITAG (ITinerary AGent) system based on the theory of itinerary scripting language which aims to minimize the effort in mobile agent applications development. A description of the itinerary language has been given with examples as well as descriptions of the implementation of ITAG on JADE. The ITAG Engine is the fundamental component of the ITAG implementation which can be ported to any mobile agent platform. The experimental results demonstrate an evidently better performance of the new ITAGIII under JADE platform compared with ITAGII under Grasshopper. Under the tested situations, there were no noticeable differences in performance between JADE based agents on the same host and on different hosts. In our future work, we aim to extend the itinerary language with more behaviors as well as enhancing the ITAG demo system with more applications.

8. ACKNOWLEDGMENTS

The work reported in this paper has been funded in part by the Australian Research Council's Research Network on Enterprise Information Infrastructure through the Taskforce on Context-Aware Computing (EII-CAC). We also acknowledge the travel assistance by King Faisal University, AlAhsa, Saudi Arabia.

9. REFERENCES

[1] Esparza, O., Fernandez, M. and Soriano, M. 2003. Protecting mobile agents by using traceability techniques. In Proceedings of the International Conference on Information Technology: Research and Education 2003, 264-268.
[2] Loke, S.W., Schmidt, H. and Zaslavsky, A. 1999. Programming the Mobility Behaviour of Agents by Composing Itineraries. In The 5th Asian Computer Science Conference (ASIAN'99), (Phuket, Thailand), Springer-Verlag, 214–226.
[3] Loke, S.W., Zaslavsky, A., Yap, B. and Fonseka, J.R. 2001. An Itinerary Scripting Language for Mobile Agents in Enterprise Applications. In Proceedings of the 2nd Asia-Pacific Conference on Intelligent Agent Technology (IAT 2001), (Maebashi, Japan), 124-128.
[4] Yap, B. and Fonseka, J.R. 2001. ITAG: Itinerary Agent, DSTC, Monash University, 29.
[5] Bellifemine, F.L., Caire, G. and Greenwood, D. 2007. Developing multi-agent systems with JADE. John Wiley, Chichester, England ; Hoboken, NJ.
[6] Leszczyna, R. 2004. Evaluation of agent platforms, Technical report, European Commission, Joint Research Centre,Institute for the Protection and security of the Citizen, Ispra, Italy
[7] Trillo, R., Ilarri, S. and Mena, E. 2007. Comparison and Performance Evaluation of Mobile Agent Platforms. In Third International Conference on Autonomic and Autonomous Systems (ICAS'07), IEEE Computer Society, 41.
[8] Burbeck, K., Garpe, D. and Nadjm-Tehrani, S. 2004. Scale-up and performance studies of three agent platforms. In IEEE International Conference on Performance, Computing, and Communications, 857-863.
[9] Kusek, K.J.G.J.M. 2006. A Performance Analysis of Multi-Agent Systems. International Transactions on Systems Science and Applications (ITSSA), *1, No. 4*. 335 – 341.
[10] Lu, S. and Xu, C. 2005. A formal framework for agent itinerary specification, security reasoning and logic analysis. In 25th IEEE International Conference on Distributed Computing Systems Workshops. 580-586.
[11] Cheng-Zhong, X. 2002. Naplet: a flexible mobile agent framework for network-centric applications. In Proceedings International, IPDPS 2002, Parallel and Distributed Processing Symposium, 219-226.
[12] Rech, L., Montez, C. and de Oliveira, R. 2006. A New Model for the Itinerary Definition of Real-Time Imprecise Mobile Agents. In 2006 IEEE International Conference on Information Reuse and Integration, 123-126.

[13] Rech, L., Montez, C. and de Oliveira, R. 2006. A Clone-Pair Approach for the Determination of the Itinerary of Imprecise Mobile Agents with Firm Deadlines. In ETFA '06. IEEE Conference on Emerging Technologies and Factory Automation, 9-15.

[14] Rech, L., de Oliveira, R.S. and Montez, C. 2005. Dynamic determination of the itinerary of mobile agents with timing constraints. In IEEE/WIC/ACM International Conference on Intelligent Agent Technology, 45-50.

[15] Toan, P., Loke, S.W. and Harland, J. 2003. Adding flexibility using structured goals: the case of itinerant mobile agents. In IEEE/WIC International Conference on Intelligent Agent Technology. IAT 2003, 562-565.

[16] Bo, Y., Da-You, L., Kun, Y. and Wang, S.-S. 2003. Strategically migrating agents in itinerary graph. In International Conference on Machine Learning and Cybernetics, 1871-1876.