SPRINGS: A Scalable Platform for Highly Mobile Agents in Distributed Computing Environments*

Sergio Ilarri, Raquel Trillo, Eduardo Mena
IIS Department, University of Zaragoza (Spain)
{silarri, raqueltl, emena}@unizar.es

Abstract

In the last decade, mobile agents have arisen as a promising paradigm to build distributed and mobile computing applications. However, mobile agents have not been massively adopted. One of the reasons could be that some issues have yet to be solved to increase the confidence of developers. Thus, scalability problems sometimes arise in applications with a high number of mobile agents when calls and trips happen very frequently.

In this paper we present SPRINGS, a novel Java-based mobile agent system which is scalable, flexible, and easy to use. Our work has been motivated by our experience with mobile agents in several research projects. We focus on scalability issues and efficient maintenance of location-independent agent references in dynamic scenarios where agents communicate and travel frequently among computers. We have obtained encouraging performance results through an extensive set of experiments. Moreover, our tests show that SPRINGS achieves a degree of concurrency that other well-known platforms cannot reach.

1 Introduction

Mobile agents [7] have stirred up a lot of interest and research efforts. Over the traditional client/server approach, they present a range of unique advantages [5] that make them suitable for mobile distributed computing [10]. Mobile agents have been around for more than a decade, and many mobile agent (MA) platforms have been developed during that time [6, 9]. So why do we propose yet another MA platform? Most existing platforms do not scale up well in distributed environments with highly mobile cooperative agents. According to our experience, this may be due to bugs that arise when there is high concurrency and overloading.

*Supported by the CICYT project TIN2004-07999-C02-02.

In this paper we present SPRINGS[1], a platform that tries to gather the nice features of existing MA platforms while avoiding some of their disadvantages. We focus on scalability issues in environments with a great number of highly mobile and cooperative agents. SPRINGS provides location transparency by dealing with *dynamic proxies*: a reference to an agent remains valid independently of the agent's migrations; thus, our system avoids searching an agent every time a remote call is performed.

The structure of this paper is as follows. In Section 2 we describe the architecture of our system. In Section 3 we explain how movements and calls are managed by SPRINGS using dynamic proxies. In Section 4 we show some performance tests that prove the suitability and scalability of our approach. In Section 5 we present some related works. Finally, the conclusions and future work appear in Section 6.

2 Architecture of SPRINGS

In this section, we show the basic architecture of SPRINGS, that allows the realization of scalable mobile agent based applications. As other MA platforms (e.g., [1]), our system is composed of agents, contexts and regions. Agents are created in contexts, which are the environments where agents execute. A context, called *place* in some other platforms, provides agents with services such as a call routing service irrespective of target agent locations and a transportation service to move to other contexts. A set of contexts can define a region, which provides underlying agents and contexts with a service that guarantees name uniqueness. In large information systems, several regions can be defined (the name service is distributed among regions).

The functionality of a *region* is provided through a remote object called *Region Name Server* (RNS), located on any computer in the network. An RNS has several functions, such as ensuring the uniqueness of agent/context names, mapping from context names to context addresses,

[1]Scalable Platform foR movING Software, which allows software agents to *spring* into computers (http://sid.cps.unizar.es/SPRINGS/).

and assigning tracking responsibilities to contexts. Regarding this last task, the RNS selects some contexts as *location servers* of a certain agent, based on their load. A location server of an agent a is a context that stores an updated reference to a, so it receives location updates when a moves; i.e., each location server of an agent a stores a dynamic proxy to a.

Concerning agent migration, we define a context c as an *observer* of a certain agent a when c is interested in knowing the current location of a. A context c can become an observer of a in two cases: 1) when a local agent wants to communicate with a, or 2) when it is designated by the RNS as a location server for a. When a context c is an observer of an agent a, then c stores a dynamic proxy to a (see Section 3.1.2). To avoid unnecessary updates, dynamic proxies are removed if they have not been used recently.

3 Agents: Movements and Calls

In this section we describe the two basic functionalities of SPRINGS agents: movements to other contexts and calls to other agents. Agents can move from context to context by just specifying the name of the target context. Similarly, an agent can communicate with another one by specifying the name of the target agent. As movement and communication exceptions could be frequent (due to network errors and agent movements, respectively), unsuccessful movements and calls are retried internally by SPRINGS instead of always force the programmer to handle them. In contrast to other MA platforms, we provide total location transparency: agents do not need to use network addresses, neither when moving to a new context nor when calling other agents.

3.1 Movements: Traveling to Contexts

We have implemented *weak mobility* (based on callback methods) in SPRINGS, as strong mobility is not straightforward to implement using a standard JVM [4] and weak mobility has proven to be sufficient in most scenarios [2]. An agent is transmitted to a new context using Java serialization through RMI. We borrow from Voyager [2] the idea of event-signaling methods that are automatically executed at different stages of a movement (preArrival, preDeparture, postArrival, and postDeparture).

3.1.1 When to Update the Proxies to An Agent

An agent proxy stores the (remote) reference to that agent: its name and current context. If the agent moves to another context, the information contained in the proxy becomes invalid. To make proxies dynamic, when an agent arrives at a new context, the proxies to that agent (held by other contexts) must be updated to reflect the new agent location.

[2]http://www.recursionsw.com/voyager_documentation.htm

A proxy should not be updated before the agent has safely arrived at its destination: an agent cannot be contacted while it travels, and its trip could fail. Once the agent has arrived and it is ready to resume its execution, two alternatives arise: 1) to update the proxies in parallel with the agent's callback, or 2) to update the proxies before the agent's callback, in serial. In Figure 1 we show the average delay for calls and movements with both strategies when we perform the test described in Section 4 with 500 agents. Notice that the parallel strategy needs more time to complete (4:00 minutes vs. 2:50 with the serial strategy), despite agents move faster with this strategy: an agent resumes its execution at destination immediately. The reason is that some agents move so quickly that someone trying to communicate with them needs an indefinite time for its call to get through [8], as quick agents travel to new contexts even before their proxies are updated. On the contrary, the serial strategy penalizes the agent's execution a little (the time needed to update its proxies) in favor of the performance of the whole network of cooperative agents. Therefore, we adopted the serial strategy.

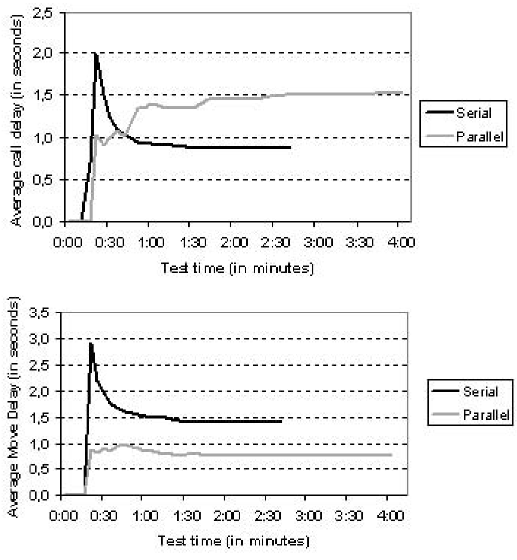

Figure 1. Strategies for updating proxies

Notice that the serial strategy cannot guarantee the absence of livelock by itself: an agent could leave its context *immediately* after its proxies have been updated, giving no chance to contact it. Thus, SPRINGS contexts detect too fast mobile agents (in our prototype, those which stay in a context less than 500 ms) and delay their departure (until 500 ms in our prototype), in order to avoid much bigger delays of calling agents. Our strategy yields good performance results, as shown in figure 2.

3.1.2 How to Update the Proxies to An Agent

In each context, a *Proxy Updater* thread is in charge of updating the *remote* (i.e., stored by other contexts) proxies to the incoming agents. To ease this task, each agent carries, as part of its status, the list of its observers (i.e., contexts interested in its location) and, on its arrival to a new context, the Proxy Updater updates its proxies at those contexts. As the Proxy Updater has to deal with simultaneous agent arrivals, it performs its task efficiently, in batches: it tries to update several proxies on the same context with just one call, independently of which agents those proxies reference. In case an update fails (e.g., a target context can be temporarily unavailable), it will be retried again in the next batch. We impose a threshold to limit the amount of time that the agent will wait until its proxies are updated (10 seconds in our prototype): this will prevent the agent from waiting indefinitely in case some of its proxies cannot be updated. Notice that all this process is performed transparently.

We have performed experiments, that we omit due to space limitations, that show that the cost of our proxy updating mechanism is linear with the number of agents; for example, updating the proxies of an agent in a test with 1000 agents traveling among 5 contexts/computers only spends 2.5 seconds (in average).

We would like to stress that proxies are kept by contexts, not by agents, which has two important advantages: 1) all the agents within a context *share* all the proxies stored locally, so the number of dynamic proxies in the system is reduced (as well as the number of updates due to agent movements); and 2) communicating an update to a context is easier than communicating it to an agent (because agents move and contexts do not!).

3.2 Calls: Communicating with Agents

Let us assume that an agent $a1$ in context $c1$ wants to communicate with another agent $a2$ in context $c2$. If any agent on $c1$ has recently communicated with $a2$, a dynamic proxy to $a2$ will be locally available ($c1$ is an observer of $a2$); in this case, the call will be directly routed through that proxy. Otherwise, $c1$ must find $a2$ in the following way: 1) $c1$ obtains from its RNS a location server for $a2$; 2) $c1$ obtains a proxy to $a2$ from that location server, which registers $c1$ as a new observer for $a2$; and 3) the retrieved proxy is used to route the call to $a2$, and it is stored by $c1$. Information about agents not located in the region will be requested by the local RNS to other RNSs.

4 Experimental Evaluation

In this section we present a set of experiments that show the feasibility and efficiency of our approach[3]. We consider a single region with 5 contexts residing on 5 different computers (Pentium IV 1.7 GHz with Linux RedHat 2.4.18 and 256 MB RAM). One of the computers also hosts the RNS.

We launch a network of cooperative agents that follow the next steps: 1) each agent is randomly assigned a peer[4] at the beginning; 2) it calls its peer; 3) it moves randomly to another context; and 4) steps 2 and 3 are repeated without delay until reaching 50 iterations. The number of agents range from 100 to 1500, depending on the specific test. Notice this agent behavior would push any MA platform to the limit, as many agents move continuously and call each other at the same time. Thus, this test is focused on evaluating: a) the efficiency of calls and moves (agents are continually calling or moving) and, therefore, the efficiency of proxy updates; b) how the system deals with livelock issues (agents move quickly among contexts); and c) the scalability of the system with an increasing number of agents.

4.1 Static vs. Dynamic Proxies

In Figure 2, where the end of the lines indicate the end of the test, we can see that a strategy with dynamic proxies outperforms one with static proxies (not automatically updated but built when they are needed).

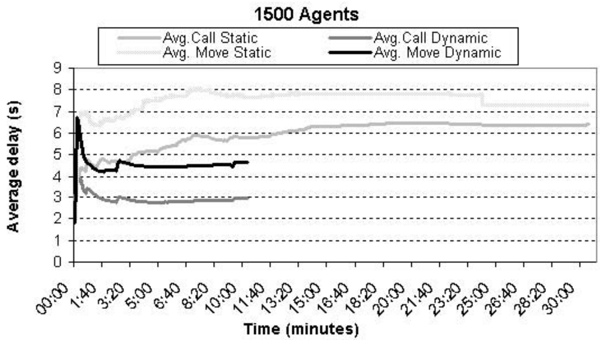

Figure 2. Static vs. Dynamic Proxies

This is because obtaining a proxy to an agent is an expensive operation that requires contacting the RNS and one of the location servers of the agent. With dynamic proxies, this additional effort happens only once per context (the first time each proxy is needed). While all the agents have finished in about 10 minutes with dynamic proxies, around 30 minutes are needed when they are static.

[3]Each test is repeated several times and average values are reported.

[4]If a new peer was chosen before each call, the management of dynamic proxies would not be tested.

4.2 Performance Evaluation

In Figure 3 we show the average move delays experienced by agents over time, depending on the number of agents in the system. The average move delay increases with the number of agents, as expected, but in a linear (with slope smaller than one) and constant proportion. Only at the beginning of the test the delays are higher, due to the overloading of the first calls (all the agents obtain the proxies to their peers at the same time). The total test time (ranging from 100 to 560 seconds) also increases with the number of agents. A similar trend is observed for the average call delay (with an average of about 2.5 seconds for 1500 agents), which is not shown here due to space limitations.

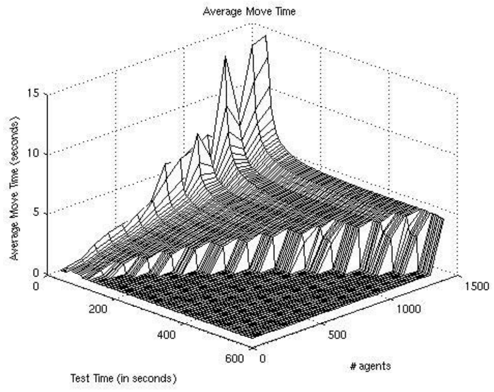

Figure 3. Average move time

In Figure 4 we show the percentage of agents that have a certain *lifetime* (expressed in minutes), which is defined as the time an agent needs to perform its tasks (in the test, 50 calls and 50 movements). For example, with 100 agents nearly all of them finish within 1 minute, while with 900 agents most of them need between 3 and 4 minutes. We can see how the lifetime increases with the number of agents linearly, showing the good scalability of the system.

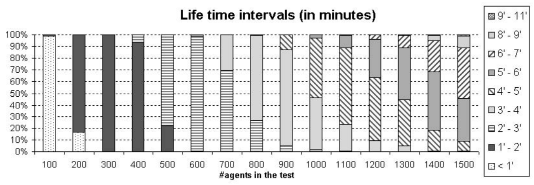

Figure 4. Occurrences with a certain lifetime

5 Related Work

There are many available MA platforms, some developed by research groups and others by private companies [6]; a reference model and evaluation can be found in [9]. In this section, we compare SPRINGS with some of the best mobile agent platform [6]: Aglets 2.0.2, Voyager 3.1.1, Grasshopper 2.2.5b, and Tryllian 2.0.

The test explained in Section 4 was developed in the different MA platforms. We use dynamic proxies whenever available in the evaluated MA platform, i.e., in Grasshopper and Voyager. Concerning Aglets, proxies to agents are not automatically updated when they point to roaming agents; therefore an aglet in our test searches for its peer in the existing contexts when a proxy becomes invalid due to an agent movement. Finally, Tryllian requires the specification of the target agent address when sending a message, so the only solution is to perform a brute force search of the target agent every time a call happens; we turned off persistency in Tryllian to avoid performance degradation.

5.1 Performance Comparison with 100 mobile agents

The greatest benefits of SPRINGS over the other evaluated MA platforms arise with a high number of agents. In Figure 5, we compare them using a test with 100 agents. We run the test for a maximum of 15 minutes (in the figures, we show only the first 10 minutes because the results were already stabilized by then).

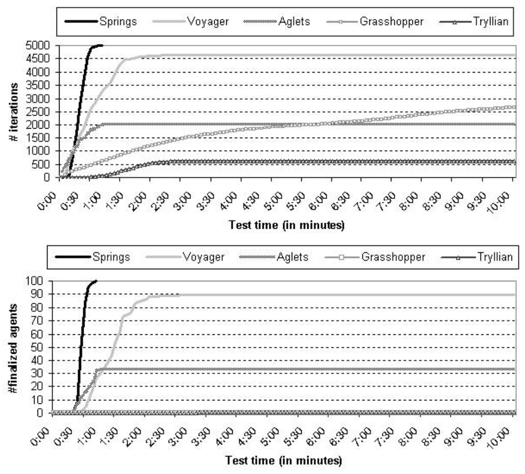

Figure 5. Comparison with 100 agents

We can observe several facts:

1. *SPRINGS exhibits the best performance and is the only platform that finishes the test.* The others just get stuck at some point and are unable to proceed due to failures of different nature. This problem grows in experiments with a higher number of agents, and so do the differences with our platform.

2. Voyager is the second best MA platform: 4600 calls are performed and 89 agents finish the test, in average.

After 2:30 minutes, the test is not able to progress (i.e., agents do not succeed in performing any more calls).

3. Next in performance is Aglets (about 2040 calls and 33 agents finished). After 1 minute, it cannot progress.
4. Grasshopper is the following MA platform in performance, with about 2600 calls. However, no agent is able to finish its tasks. This cannot be clearly observed because Tryllian achieves a similar performance.
5. Tryllian has a more variable behavior according to our experiments, so we used a greater number of samples to study it. On average, about 600 calls are performed, and only 1 agent finishes the test.

We conclude that SPRINGS shows the best performance from the point of view of reliability and scalability when running a highly dynamic distributed mobile agent network.

5.2 Performance Comparison with 1 mobile agents

To compare the platforms in a less challenging environment, we also performed a test there is a single mobile agent that performs 50 iterations, each one consisting of a call to a fixed static agent and a movement to a context chosen randomly. In Figure 6 we compare the MA platforms in terms of the total number of iterations ended along time. SPRINGS, Voyager, and Aglets (in decreasing efficiency order) perform similarly (below 5 seconds). However, Grasshopper and Tryllian need a much longer test time (about 50 and 110 seconds, respectively).

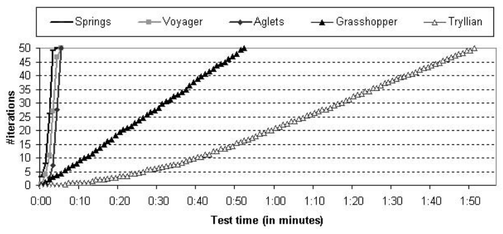

Figure 6. Comparison with 1 mobile agent

6 Conclusions and Future Work

In this paper we have presented SPRINGS, a mobile agent platform that manages location transparency efficiently, allowing scalable systems with many highly mobile and cooperative agents. Our proposal presents the following features:

- It outperforms and is more scalable than other well-known platforms in highly dynamic multiagent systems in distributed environments.
- It deals with dynamic proxies, which are automatically and efficiently updated when a mobile agent changes its location, relieving the programmer of such an effort.
- It minimizes the probability of livelocks by observing traveling agent behaviors, to avoid that calling agents cannot get their calls through for a long time.

We have successfully implemented LOQOMOTION [3], a highly dynamic multiagent application to process continuous location-dependent queries in mobile computing environments, using SPRINGS. Its use allowed us to increase the scalability of our query processing. As future work, we plan to offer a MASIF API to SPRINGS, release it as an open-source project, and develop some security mechanisms.

References

[1] C. Bäumer and T. Magedanz. The Grasshopper mobile agent platform enabling shortterm active broadband intelligent network implementation. In *First International Working Conference on Active Networks (IWAN'99)*, pages 109–116, London, UK, 1999. Springer-Verlag.

[2] L. Bettini and R. D. Nicola. Translating strong mobility into weak mobility. In *Mobile Agents*, pages 182–197, 2001.

[3] S. Ilarri, E. Mena, and A. Illarramendi. Monitoring continuous location queries using mobile agents. In *Sixth East-European Conference on Advances in Databases and Information Systems (ADBIS'2002), Bratislava (Slovakia)*, pages 92–105. Springer Verlag LNCS, ISBN 3-540-44138-7, September 2002.

[4] T. Illmann, T. Krueger, F. Kargl, and M. Weber. Migration in Java: Problems, classification and solutions. In *Multi-Agents and Mobile Agents in Virtual Organizations and E-Commerce (MAMA'00), Australia*, December 2000.

[5] D. B. Lange and M. Oshima. Seven good reasons for mobile agents. *Commununications of the ACM*, 42(3):88–89, 1999.

[6] K. Ludwig, A. Josef, W. E. Edgar, S. Wolfgang, and G. Franz. Using mobile agents in real world: A survey and evaluation of agent platforms. In *Second International Workshop on Infrastructure for Agents, MAS, and Scalable MAS*, May 2001.

[7] D. Milojicic, F. Douglis, and R. Wheeler. *Mobility: processes, computers, and agents*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1999.

[8] A. L. Murphy and G. P. Picco. Reliable communication for highly mobile agents. *Autonomous Agents and Multi-Agent Systems*, 5(1):81–100, 2002.

[9] A. R. Silva, A. Romäo, D. Deugo, and M. M. D. Silva. Towards a reference model for surveying mobile agent systems. *Autonomous Agents and Multi-Agent Systems*, 4(3):187–231, 2001.

[10] C. Spyrou, G. Samaras, E. Pitoura, and P. Evripidou. Mobile agents for wireless computing: the convergence of wireless computational models with mobile-agent technologies. *Mobile Networks and Applications*, 9(5):517–528, 2004.