MobiGrid[⋆]
Framework for Mobile Agents on Computer Grid Environments

Rodrigo M. Barbosa and Alfredo Goldman

Department of Computer Science
Institute of Mathematics and Statistics
University of São Paulo
{rodbar, gold}@ime.usp.br

Abstract. This paper presents a project which focuses on the implementation of a framework for mobile agents support within a grid environment project, namely InteGrade. Our goal is to present a framework where time consuming sequential tasks can be executed on a network of personal workstations. The mobile agents may be used to encapsulate long processing applications (tasks). These agents can migrate whenever the local machine is requested by its user, since they are provided with automatic migration capabilities. Our framework also provides to the user a manager that keeps track of the agents submitted by him.

1 Motivation: Mobile Agents on Computational Grids

In the past, high-performance computation was done only on supercomputers. These computers were parallel computers, composed of many processors with shared or distributed RAM, interconnected by a high-speed bus. Nevertheless, this kind of computer has a very expensive price and when it is not being used, there is a huge waste of resources, since plenty of computation time is lost.

Facing this problem, researchers looked for a new paradigm in order to build non-expensive high-performance computers: the *clusters*. A *cluster* is a set of many ordinary computers - usually PCs - interconnected by a high-speed network. Even though the price problem was faced by this solution, the waste problem still remains: when a *cluster* is not being used, plenty of resources are still being wasted.

So, a new paradigm was created: the *grid*. The *computational grid* idea was clearly inspired by *clusters*, in the sense that we have many computers interconnected by a network in order to provide together greater computational power. However, a *grid* does not rely on high speed networks and is more available; they can be composed of computers spread around the world, interconnected by Internet, for example. The idea is to provide computational resources similarly to the way we get power supply [1]: when you want power supply, you may

⋆ Project supported by the Research Support Foundation of the State of São Paulo - FAPESP. Process number: 02/11165-6

connect your device to the power grid; when you want computational resources, you may connect your device to the *computational grid*. The waste problem is addressed in a way that whenever a computer is idle, its computational power can be supplied to the *grid*.

On this context, the idea of mobile agents can be interesting. They can be used to encapsulate opportunistic applications, which can use small slices of the available computational time of personal workstations, migrating to another machine whenever the local user request his machine, always preserving the processing already done. On this way, mobile agents can be considered as a complementary tool to decrease even more the idle time of a *grid*.

This text proposes our solution to provide a mobile agents environment in a grid and is organized in the following way: Section 1 explains the motivations to provide mobile agents support in a *grid*, specifically InteGrade; Section 2 introduces the InteGrade project; Section 3 describes our project objectives; Section 4 introduces some mobile agents environments; Section 5 gives a general overview of the framework; Section 6 shows the class structure of our framework; Section 7 shows some tests we made for measuring the overhead; Section 8 provides ideas for future work and concludes this paper.

2 InteGrade

InteGrade project [2] is building a middleware infrastructure which enables idle time utilization of machines already owned by public or private institutions. One of InteGrade's goals is to use this idle time to solve many kinds of parallelizable problems, including strongly coupled applications.

InteGrade is being built using the most modern technologies of distributed objects systems, industry standards, and high-performance distributed computing protocols.

The main requirements that are considered in the InteGrade development are:

1. the system must know itself: this means the necessity of maintaining a database refreshed dynamically which contains information on the system, on hardware and software platform of each machine, on the links interconnecting the machines, besides the grid dynamic state, which means the use of resources like disk, processor, memory and bandwidth;
2. almost no overhead for the clients: the *middleware* must be able to use the available idle resources of the client machines with the least possible impact on the overall performance perceived by their users;
3. security guarantee: since it will be possible to dynamically load executable code at the clients machines, it is important to guarantee that this code will not harm the correct functioning of other applications being executed at the client machine. It is also important to guarantee that this code will not modify or gain access to personal information, possibly confidential, stored at the client machines.

2.1 InteGrade Differentials, Compared to Other Grid Projects

1. reutilization of the installed computer base with low overhead to machine users: it is one of the major points on the InteGrade project. Reutilization can be observed in other projects, being not a very important concern in Globus [3] and Legion [4], and a major concern in Condor [5]. The InteGrade differential is in the fact that this concern was considered in the development of its architecture;
2. utilization of modern distributed objects technologies: exclusive feature of InteGrade, that will use mostly CORBA [6] on its implementation. With these characteristics we get two main advantages: we can reuse a lot of services already available in CORBA architecture; the integration of other services and applications on the Grid will be easier and faster. Even though our framework is not directly based on CORBA, it can easily be connected to InteGrade via CORBA.

3 Objectives

This project consists in the implementation of a mobile agents infrastructure for InteGrade. The idea of using mobile agents for grid computing is not new [7], but in our framework the main idea is to allow an efficient utilization of computational resources for large sequential or for embarrassing parallelizable applications.

Mobile agents migration ability meets two major InteGrade goals: the system must be transparent to the machine user, in other words, the machine user must have the highest priority compared with InteGrade applications; the idle resources must be used in the best possible way.

In the case where there is the necessity to free the machine resources for the local user, for example, if this is a machine running mobile code, this code can migrate to another machine without losing the partial results already computed. On this process, the InteGrade architecture provides information about the network and the other machines, allowing the mobile agent to choose a machine with more adequate available computational resources. In order to do that, utilization patterns of other machines resources can also be used.

It would also be very interesting to allow the migration of mobile agents to more powerful machines that may become available. But this migration would bring benefits only if this machine utilization pattern shows that it will be on this state for some time, since the migration process is costly.

On this way, mobile agents can be used in a complementary way to InteGrade applications, allowing a even better utilization of computational resources. Among the applications that could be executed using mobile agents, there are loosely coupled parallel applications like SETI@home [8] and BOINC [9], or sequential applications that demand long processing time.

4 Mobile Agents Environments

In an IBM's pioneer report *Mobile Agents: Are They a Good Idea?* [10], Chess et. al analyze the potential of mobile agents and introduce, among other ideas, the possibility of using mobile agents in order to use idle computational resources. That could be done by using mobile agents to encapsulate processes that would migrate through a network always looking for the resources they need. On this process, they take with them their execution state, creating a new paradigm that takes the programs to data instead of taking data to the program. Yet in this report, it is stated that the mobile agent support should be done over a interpreted language. This strategy faces the problem of saving the execution state besides the platform heterogeneity. Another report, *e-Gap Analysis* [11], which makes a study about the way scientists do science nowadays with technology support - what is named *e-Science* - points, among other gaps, the absence of mobile agents support in the existent grid infrastructures.

These articles lead us to the idea of creating a Java framework for mobile agents support on grids. The choice for Java was motivated by two reasons: the main reason is due to the fact that Java is a robust, popular and modern language; the second reason is due to the results of a comparison made among many mobile agents environments [12]. In this comparison, where lower grades represent better performance, the environments Grasshopper [13] and Aglets [14], both in Java, are placed among the best, with grades 9.25 and 10.15, respectively.

Grasshopper has excellent documentation and respects the OMG MASIF standard [6]. However, Grasshopper producer, IKV++ Technologies, restricts its utilization inside other projects without royalties payment, as well as to do comparisons with it. These factors were crucial and did not allow us to use this environment, leading us to choose Aglets.

Aglets is a Java environment for mobile agents development and implementation. Aglets Software Development Kit, ASDK, was prototyped and created by IBM [15]. It has become a open source initiative, ruled by IBM license for open source, which allows the code utilization and modification, as well as comparisons. The product documentation, although incomplete, is relatively organized, and the environment provides all the basic resources for the creation of our framework, besides more advanced resources. Resources for mobile agents creation, migration, clonation are provided in addition to advanced resources of security, synchronization and message exchange. Aglets provides a class structure whose main elements are [16]:

1. `Aglet`: it is the class that represents the mobile agent. It provides methods for migration, clonation, suspension among other features. An object of this class is named *aglet*;
2. `AgletProxy`: represents the *aglet* proxy. It serves as a interface between the *aglet* and the object that references it;
3. `AgletContext`: represents the *aglets* host. There can be many *aglets* in a *context* and many *contexts* in a server;

4. `AgletID`: corresponds to an unique global identifier of the *aglet*.

5 Framework Overview

The main idea of our framework is to provide the programmer with a programming environment for long running applications, which we call *tasks*. We will mention all the components of our framework at a high level:

1. *task*: long running applications, encapsulated in a mobile agent. On the implementation of these *tasks*, the programmer must take care of the *task* state, since the standard Java environments for mobile agents provide weak migration, in other words, only the objects states and variables are preserved, not the state of execution stacks. This comes from the fact that Java Virtual Machine (JVM) forbids threads inspection by the user application, for security reasons. There are a several implementations of mobile agents which allow strong migration [17]. Nevertheless, these environments use modified versions of JVM, what is not desirable, since such JVMs usually become obsolete compared to new versions of Java 2, not following Sun Microsystems standardization;
2. *manager*: it is the component responsible for registering *tasks*. The user who submits *tasks* must have the *manager* active at his machine. The two most important functions of the *manager* are:
   (a) migration: when a *task* is submitted, the *manager* queries the InteGrade infrastructure searching for an idle machine which has a chance of remaining on this state for a given time. With such information, the *manager* dispatches the *task* to such a machine. A similar procedure is used when the *tasks* need to migrate;
   (b) *liveness*: the *manager* also creates a clone of the *task* and dispatches it to another machine. In the context of our work, a *task* that is being executed in more than one machine has *liveness*. When one of the clones dies - which can occur for many reasons, for example, a energy supply interruption at the machine where it is running - the *manager* makes a copy of the clone still alive and dispatches it to another machine. We call *twins* the two clones of a *task*. The choice for two clones is arbitrary, because we could choose a greater number. However, a greater number of clones would imply more waste, since we would be using many resources for running the same *task*;
3. *light server*: this is the server installed on each machine that provides resources to the framework. It provides a execution environment for the *tasks*. When the machine is requested by the local user, the *server* asks the evacuation of the *tasks* that are being hosted by it. These *tasks* query their *managers*, which communicate with InteGrade looking for idle machines. When the *managers* get such information, they take actions in order to evacuate the *tasks* to new machines. At this point, there is another important discussion: how to build *servers* light enough to not interfere with the local

user? We choose two solutions: the first is to implement a minimal server that does not use advanced resources of Aglets which are not necessary to our framework; the other solution is to use a small *daemon* written in C;
4. *daemon*: it verifies whether the machine is idle or not. When the machine is idle, it communicates that to InteGrade and turns on the *light server*. When the machine is requested by its local user, the *daemon* informs the *server*, which evacuates the *tasks* and terminates. This *daemon* can be inserted on InteGrade's LRM (Local Resource Manager) [2], which is responsible for monitoring the local resources. LRMs send this information periodically to the GRM, which uses it for scheduling within the grid. So, the *daemon* is responsible for informing InteGrade that the machine is ready to receive *tasks*;
5. *client*: component that provides the user with tools to submit *tasks* to the framework. Also provides a host environment for the *manager*.

In Figure 1, we have an overview of our framework. Each one of the *clients* hosts a *manager*, which communicates with InteGrade. Client 1's *manager* manages Task 1 and its clone. Client 2's *manager* manages Task 2 and its clone. Observe also that the *daemons* communicate with InteGrade in order to inform it when a local machine is idle, turning on the server then.

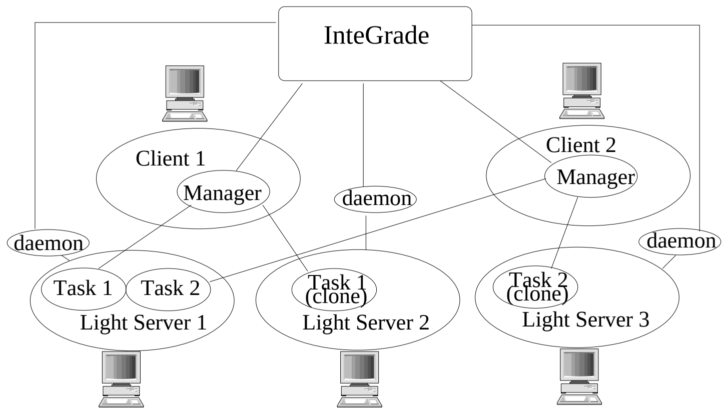

Fig. 1. General architecture of the framework

In the architecture creation, there were two main problems that should be addressed: (1) How to prevent the *tasks* from terminating suddenly? (2) How to use Java on the clients without making the local machine slow for its user?

Two immediate solutions were studied for the first problem: check-pointing and redundance. Check-pointing would demand state saving on disk from time to time and redundance could be considered as having two or more copies of a *task* running on different machines. However, check-pointing cannot be considered a good solution by itself, since a machine could suddenly be turned off and remain on this state for a long time. From this point of view, check-pointing could not work without redundance, since a *task* would need a clone to prevent this undesirable situation. So, we choose to address first this problem by using only redundance. We always have two copies of a *task* running independently, in case of a sudden termination, we could clone the remaining *task*. This solution would fail on the improbable situation of a sudden death of both *tasks* before the *manager* realizes it, but this probability can be reduced by putting them to run on different networks. By using only the redundace solution, we prompt the user for a new submission in case of sudden termination of both *tasks*.

As said before, we addressed the second problem using a *daemon*.

6 Class Structure of the Framework

We will now give a more detailed view of the framework, specifying it by classes:

1. `AgletTask`: this class represents the *tasks* already cited. To define a *task*, the programmer must extend this class and redefine the `defineState()` method, which must return an `AgletState`, that encapsulates the *task* implementation, as well as its state. This class is associated with the `AgletState` class.
2. `AgletState`: class that represents the *task* state, as well as its implementation. The programmer must extend it and implement the following methods:
   (a) `run()`: method that defines literally the *task* implementation, in other words, it defines the long running application that the user wants to submit to the framework. The programmer must have in mind the fact that he is responsible for saving the present state of the application. For this purpose, a feature is being implemented: the method `checkPoint()`. Any Java object implementing `java.io.Serializable` which is referenced inside `AgletState` object will be automatically preserved when the *task* migrates. This `checkPoint()` method will be used to inform the framework that the *thread* has reached a consistent state. The programmer must test the value returned by it: if it returns `true` the *thread* must be stopped in a logical way; if it returns `false`, nothing is done. The reason for that is to prevent the *thread* from entering a inconsistent state. With this feature, the programmer would just take care of calling this method whenever the application reaches a consistent point (a point where the invariants are preserved). The programmer must know also that, at the end of the *task* execution, a call to the `finish()` method must be made.
   (b) `printResults()`: prints the results of the *task* processing. It is called by the framework when the *task* finishes to be executed and returns to the client.
3. `Server`: the class that represents the *light server* already cited. This `Server` hosts the `AgletTasks`. The server is provided with Aglets environment for hosting *aglets*. Also it receives communication of the *daemon* in order to know when it is time to evacuate the `AgletTasks`. We have a simple *light server* already implemented.

4. `AgletManager`: Represents the *manager*. We have implemented the register and evacuation of the `AgletTasks`, as well as the control of their *liveness*. `AgletManager` is nothing more but a special kind of `Aglet` which never migrates. We implemented it this way in order to use the Aglets tools for message exchanging among *aglets*. The communication between `AgletManager` and InteGrade is not implemented yet, since it is easier to test it alone.
5. `Client`: Represents the *client*, which means that this class is responsible for `AgletTasks` submission to the framework. It is also used to host the `AgletManager`. For now, we are testing our code by using Tahiti - Aglets visualization tool, which provides an environment for hosting the `AgletManager` and tools to submit *tasks* to our framework.
6. `AgletProxy`: Component of the Aglets environment. It is the class that mediates the communication between two *aglets*.
7. `AgletListener`: Class that executes pre-migration ant post-arriving operations: respectively, `onDispatching()` and `onArriving()`. This class is transparent to the user.

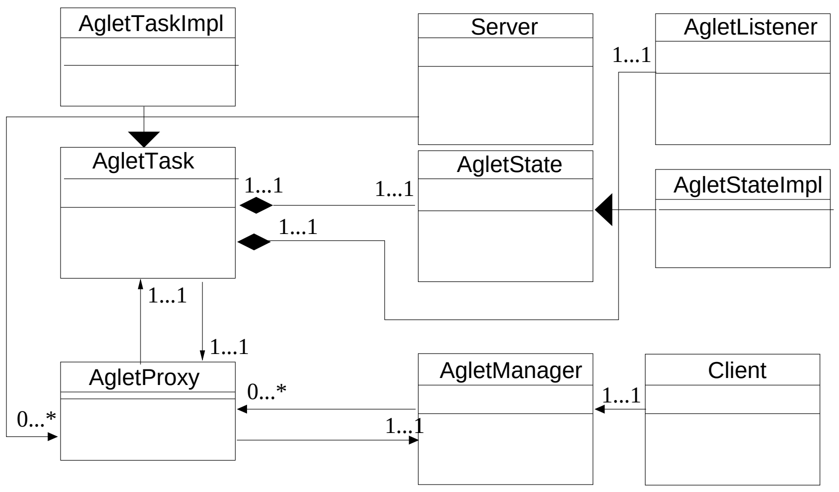

Fig. 2. General UML of the framework

In Figure 2, `AgletStateImpl` and `AgletTaskImpl` are user implementations for, respectively, `AgletState` and `AgletTask`. Notice that each `AgletTask` has one reference to the `AgletState` and `AgletListener`. The `Server` has a list of `AgletProxies` which represents the `AgletTasks` which it hosts. The `AgletTask` has one reference to the `AgletProxy` that represents its `AgletManager`. The `AgletManager` has a list of `AgletProxies` which represents the `AgletTasks` that it manages.

6.1 An Example

Next, we show, as an example, the implementations of a *Hello World task*. This implementation is merely an example of how to use our framework, since it does not have any practical utility. Notice that, whenever the *task* migrates, when it arrives at its new host, the `run()` method is executed. The *task* know how many times it has already said "hello" on other machines, preserving already done processing. When it migrates, all the instance variables of `AgletStateImpl` are preserved: `howMany` and `canSay`.

```
public class AgletTaskImpl extends AgletTask {
   public AgletState defineState () {
      return new AgletStateImpl ();
  }
}
public class AgletStateImpl extends AgletState {
   int howMany = 0;
   public void run () { /* go saying hello until migration_time */
      System.out.println (``I have said hello '' +  howMany +
                         ``times'');
      for (;howMany < 1000000; ++howMany} {
         System.out.println (``Hello'');
      }
      if (howMany >= 1000000) finish ();
   }
   public void printResults () { /* processing results */
      System.out.println (``I am back! I have said hello `` +
                         ``1000000 times'');
   }
}
```

7 Tests

We have made some tests in order to determine the migration and state saving overhead. The migration overhead is mostly related to the network speed, while the state saving overhead is related to the serialization of the objects associated with the `AgletTask`, which is a costly process. We submitted *tasks* that solve a *makespan problem*: To determine a distribution of n independent tasks[1] with execution times of $t_1, t_2, ..., t_n \in \mathbb{Z}^+$ in m machines minimizing the time the last task ends. This is a NP-hard problem and we implemented a exponential solution which tests all the possible combinations. Two kinds of implementations were made: a simple implementation that does not take care of state saving nor migration; an `AgletTask` implementation which saves, for each iteration, the last combination analyzed.

[1] Note that here the word *task* has its usual meaning, not denoting the meaning we attributed to it.

Table 1. Tests for a makespan problem

n, m	Simple, *host 1*	Simple, *host 2*	`AgletTask`	`AgletTask`, 2	`AgletTask`, 4
2, 26	22s	13s	50s	30s	25s
2, 28	1min 30s	50s	3min 17s	3min 08s	3min 04s
2, 30	6min 36s	3min 28s	15min 45s	15min 09s	14min 55s

In this table, we show tests for applications with parameters n, m in a simple application running on *host 1* and *host 2*, in a `AgletTask` that does not migrate running on *host 1* and in a `AgletTask` migrating 2 and 4 times, between *host 1* and *host 2*. *Host 1* is a Athlon XP 2500, 512 Mb of RAM and *host 2* is a Pentium IV 2.8 GHz HT, 1 Gb of RAM.

We can see that the state saving overhead is significative, what has motivated our study on this area leading to the `checkPoint()` method. Also, we verify that the migrating overhead may not be very significative, being absorbed when the *task* migrates to a faster machine. That happens because when the *task* migrates to a faster machine, its execution speed is increased, compensating the migration time loss. Even though the `AgletTasks` stayed most of the time on *host 1*, it was worth to migrate to *host 2*, which is a faster machine.

8 Future Work and Conclusion

For now, we have implemented a working part of the framework. We have `AgletTask` and `AgletState` classes implemented, so the user can implement *tasks* which will be submitted to the framework. We also have a `AgletManager` that takes care of *tasks* migration and *liveness*, as well as a `Server`, which already implements the evacuation function.

Even though the present code is functional, there are many details to be implemented, as well as features which deserve close attention like: *daemon* implementation; *client* implementation; better support to present state saving (already being implemented); communication between the `AgletManager` and InteGrade.

The idea of mobile agents in InteGrade is very instigating and interesting. Our framework, in the future, can be used to implement many kinds of applications from embarrassing parallel applications, like SETI@home, to applications that needs many kind of resources not available in a single InteGrade node. Given the opportunistic characteristic of mobile agents, InteGrade can reach near zero idleness on its nodes, what is impossible nowadays. Another interesting point is our concern on building a framework that is transparent for the machine user, so he will not face a performance loss. A strong feature of our framework is its portability, since it is written almost all in Java.

In the future, our framework can be extended to serve a grid infrastructure that manages different kinds of resources available on several machines. This would be a generalization for resources utilization, since for now our framework is focused only in computational resources.

References

1. Ian Foster and Karl Kesselman. *The Grid: Blueprint for a new Computing Structure*, chapter 2,3,5,11. Morgan Kaufmann Publishers, 1999.
2. Andrei Goldchleger, Fabio Kon, Alfredo Goldman, Marcelo Finger, and Germano Capistrano Bezerra. InteGrade: Object-Oriented Grid Middleware Leveraging Idle Computing Power of Desktop Machines. *Concurrency and Computation: Practice and Experience*, 16:449–459, March 2004.
3. Globus. Site of the Globus project, 2004. `http://www.globus.org`. Last visit on February, 2004.
4. Legion. Site of the Legion project, 2004. `http://www.cs.virginia.edu/~legion/`. Last visit on February, 2004.
5. Condor. Site of the Condor project, 2004. `http://www.cs.wisc.edu/condor/`. Last visit on February, 2004.
6. OMG. Site of the Object Management Group, 2004. `http://http://www.omg.org`. Last visit on February, 2004.
7. Rocco Aversa, Beniamino Di Martino, Nicola Mazzocca, and Salvatore Venticinque. Mobile agents for distributed and dynamically balanced optimization applications. In *Proceedings of the 9th International Conference on High-Performance Computing and Networking*, pages 161–172. Springer-Verlag, 2001.
8. SETI@home. Site of the SETI@home project, 2004. `http://setiathome.ssl.berkeley.edu/`. Last visit on February, 2004.
9. BOINC. Site of the BOINC project, 2004. `http://boinc.berkeley.edu`. Last visit on February, 2004.
10. David Chess, Colin Harrison, and Aaron Kershenbaum. Mobile Agents: Are They a Good Idea? Technical report, T.J. Watson Research Center, 1995.
11. Geoffrey Fox and David Walker. e-Science Gap Analysis. Technical report, Indiana Univesity and Cardiff University, june 2003.
12. Josef Altmann, Franz Gruber, Ludwig Klug, Wolfgang Stockner, and Edgar Weppl. Using Mobile Agents in Real World: A Survey and Evaluation of Agents Platforms. 2000. `www.umcs.maine.edu/~wagner/workshop/05_altmann_et_al.pdf`.
13. Grasshopper. Site of the Grasshopper project, 2004. `http://http://www.grasshopper.de`. Last visit on February, 2004.
14. Aglets. Site of the Aglets project, 2004. `http://aglets.sourceforge.net`. Last visit on February, 2004.
15. IBM. Site of Aglets on IBM, 2004. `http://www.research.ibm.com/trl/aglets`. Last visit on February, 2004.
16. Mitsuro Oshima and Guenter Karjoth. Aglets Specification 1.0. Technical report, IBM, may 1997. `http://www.research.ibm.com/trl/aglets/spec10.htm`.
17. Sara Bouchenak. Making Java Applications Mobile or Persistent. In *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems*, january 2001. http://citeseer.ist.psu.edu/bouchenak01making.html. Last visit on May, 1994.