Mobile Agent Procedures: Metacomputing in Java*

Dimitrios Barelos
Computer Science Department
University of Ioannina
GR 45110, Greece

Evaggelia Pitoura
Computer Science Department
University of Ioannina
GR 45110, Greece
pitoura@cs.uoi.gr

George Samaras
Computer Science Department
University of Cyprus
CY 1678 Nicosia, Cyprus
csamara@turing.cs.ucy.ac.cy

Abstract

In this paper, we introduce Mobile Agent Procedures (MAPs) as an efficient, convenient and transparent means of utilizing available networked computational resources. MAPs are mobile remote procedures that are executed in the most appropriate site in a cluster of heterogeneous workstations. Procedure migration is performed by middleware software transparently from the application and without relying on operating system or compiler support. The implementation platform for MAPs are mobile agents in Java.

1. Introduction

A metacomputer is a network of remote and heterogeneous computational resources linked by software in such a way that they can be used seamlessly as a single computational unit. Over the past years, the Internet has grown rapidly connecting millions of mostly idle machines. This grow in conjunction with the development and widespread use of Java has set the scene for an efficient and transparent way of utilizing a large number of available networked machines towards metacomputing in the large.

In this paper, we introduce MAPs (Mobile Agent Procedures) as an efficient and convenient means of metacomputing. MAPs extend remote procedure calls (RPCs) by letting a procedure to be executed at the most appropriate site among a cluster of Internet or intranet connected workstations. Procedure migration is performed by middleware software transparently from the application and without relying on operating system or compiler support. Being implemented as simple annotations to a Java program, MAPs are easy to use.

The implementation platform for MAPs are mobile agents in Java. Since MAPs are solely implemented in Java, they provide the same level of security, portability and heterogeneity as Java. They can be executed at any machine independently of its platform or operating system.

An additional benefit is the asynchronous nature of a MAP execution. Being implemented as a mobile agent, a MAP may be launched to a site and return with the results when its assigned task is completed. Furthermore, a MAP can change its route and be executed at a different site, when, for instance, its initial destination site fails. Thus, MAPs work well in slow networks and with intermittent connectivity [14]. Moreover, MAPs can be used for programming using very light-weight clients, such as palmtops, to efficiently transfer computation to more resource-capable devices.

The underlying support for MAPs include a load monitoring system that is also implemented as middleware outside the operating system using the same Java mobile agent platform. The load monitoring system keeps track of the available resources and their load. The system configuration is dynamic: any networked machine can join or leave the MAP system to either contribute its resource or take advantage of the available resources. Using MAPs, the system automatically adapts to the current load by appropriately moving computation around.

The rest of this paper is structured as follows. Section 2 introduces the MAP system and the MAP client object model. Section 3 describes how the MAP system can be

*In Proceedings of the ICDCS Workshop on Middleware, May 1999. To appear.

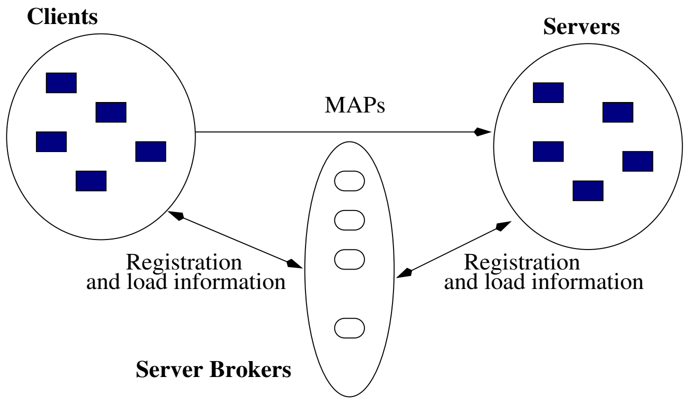

Figure 1. MAP's architecture

used for metacomputing. In Section 4, related work is summarized briefly. Finally, Section 5 concludes the paper by summarizing and presenting plans for future work.

2. The MAP System

The goal of the MAP system is to take advantage of the available networked workstations and distribute the load among them in the best possible way and in a manner transparent to the user. The workstations may be heterogenous running different operating systems or having different system architectures.

The approach taken by MAP is that the programmer specifies which procedures to move. There are three types of procedures: local, host-specific remote, and mobile remote. *Local procedures* are procedures that are executed locally at the client. *Host-specific remote procedures* are procedures executed remotely but at a specific network site. Finally, *mobile remote procedures* are procedures that are executed at any from a number of network sites. The site of execution of a remote mobile procedure is selected by the MAP-system transparently from the user based on the current load. Mobile remote procedures can be relocated to a new site transparently from the user if their destination site fails. In addition, remote procedures mask network disconnections from the user, since they do not abort when a network failure occurs; instead they poll their sending site repeatedly till the connection is re-established. The user can declare a procedure to be any of the three types.

The platform for implementing MAPs is a Java-based mobile agent framework (see [12] for a survey of such frameworks). Each remote procedure is transformed transparently from the user into a mobile agent. Mobile agents are processes that may be dispatched from a client computer and transported to a remote server computer for execution. A Java-based agent framework provides an agent server along with mobile agents that can migrate from server to server in some fashion carrying their state with them. Our platform is the IBM's Aglets Workbench [11]. The primary server provided by this workbench is called *Tahiti*, while the mobile agents are called *aglets*.

2.1. The MAP System Architecture

Sites act as clients or servers. *Clients* are the sites that send procedures for execution at other sites. *Servers* are the sites that accept procedures for execution. A site may be both a client and a server. In addition, *server brokers* act as intermediates (Figure 1). Each server broker:

- keeps track of the system configuration, that is of the participating clients and servers,
- monitors the servers' resources, and
- notifies the clients of any changes in the system load.

To monitor the servers' resources, the server broker creates a mobile agent called $LoadCounter$. The $LoadCounter$ agent visits the participating servers in turn. To compute the load of each server, the $LoadCounter$ takes into account:

- the number of mobile agents that are hosted in the given context,
- the available memory of the Java Virtual Machine,
- the response time of a simple benchmark that the $LoadCounter$ submits at each server it visits.

The server broker sends at each participating client information regarding the available servers and their current load along with an indication of the "best" destination site. The client decides where to send any mobile remote procedures using the following heuristic: with probability $1 - a$, it follows the broker's suggestion and with probability a

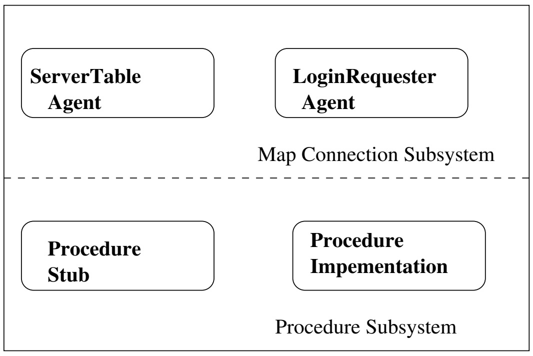

Figure 2. The components of a MAP client

randomly chooses as destination one of the available sites, where $0 \leq a \leq 1$. This is done to avoid overwhelming any particular site.

2.2. The MAP Client Object Model

Figure 2 depicts the constituent parts of a client; where each component is implemented as a mobile agent. There are two subsystems: the procedure subsystem and the MAP connection subsystem. Before a remote procedure is transfered, a $ProcedureStub$ agent is created. This agent is the local part of the procedure. The $ProcedureStub$ agent transforms the procedure to be executed into a mobile agent called $ProcedureImpl$ and provides it with the address of a destination site. The $ProcedureImpl$ moves to the specified address. After its execution, it returns to its $ProcedureStub$ with the results.

Besides scheduling its procedure for execution, each client must also register with the server brokers. This is taken care of by the MAP client connection subsystem. To enter the MAP system, the $ServerTableAgent$ creates the mobile agent $LoginRequester$. The $LoginRequester$ moves to the server broker to negotiate the entrance of the client into the system. In case of success, it returns with the table of available servers and their current load. It is the responsibility of the $ServerTableAgent$ to accept the tables that are subsequently sent by the server broker.

The class hierarchy of a MAP client is depicted in Figure 3. A program, called for instance $aProgram$, to be run as a MAP client, is defined by MAP to be a subclass of the MAP-system defined class MAP_client. The MAP_client class includes the following variables:

- $context : AgletContext$: This is the context in which all procedures that are going to be executed at remote sites are created as well as the context in which all procedures returning from a remote site are placed.

- $daemon : Daemon$: implements the Agent Transfer Protocol (ATP). It listens to a specified port for incoming aglets and places them in the right context.

- $servers : HashTable$: a table of all servers and their current ATP address.

- $server\_broker : String$: The server broker's addresses.

The $aProgram$ class includes all procedures to be executed. A local procedure is just a method of the $aProgram$ class; the code of the method is the code of the local procedure. For a host-specific or mobile remote procedure, two agents of class $ProcedureStub$ and $ProcedureImpl$ are created. Besides being an aglet, the $ProcedureImplBase$ has the following characteristics:

- when the connection to the sending client is down, it periodically polls the sending client, and

- if the destination site is down, it notifies the sending client and asks for an alternative route.

The $ProcedureStub$ passes the parameters of a remote procedure in a hash table. When $ProcedureImpl$ returns, it sends a message of type $result$ to the $ProcedureStub$ agent. In turn, $ProcedureStub$ passes any results to the main program. A special parameter called $ResultNotifier$ acts as a semaphore to synchronize the execution of a remote procedure. When a remote procedure is sent for execution, the execution of the main program is suspended. Execution of the main program resumes, when the procedure returns with the result. Remote procedures can be executed concurrently with each other, if the user defines them using a special MAP instruction called $parallel$. In this case, the main program continues when all such defined procedures return.

The $ServerTableAgent$ and the $LoginRequester$ classes implement the MAP client connection subsystem. A client gets the address of the broker(s) through a pair of programs called $geturl$ and $urlsrv$. The $geturl$ program broadcasts a message in the local network asking for the address of the broker. The $urlsrv$ program is running at some of the nodes of the network. It replies to the message sent by the $geturl$ program with the address of the broker.

3. Programming with MAPs

To use the MAP system, the user just needs to define the type (local, host-specific remote, or mobile remote) of each procedure. This is done using a very simple declaration language, called MAPDL (Mobile Agent Procedure Definition Language), the syntax of which is given in the Appendix.

The MAPGEN precompiler accepts as input the definitions of the procedures in MAPDL and generates the code

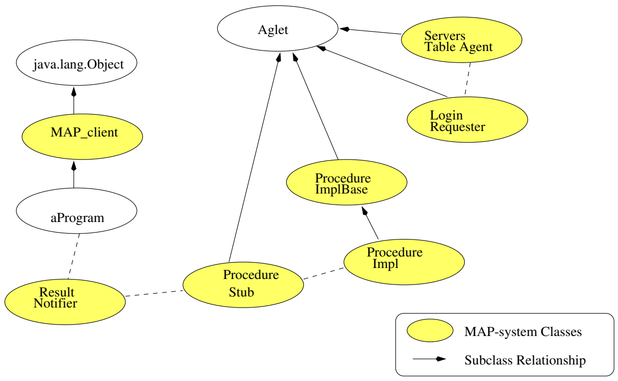

Figure 3. The client object model

necessary for the program to run as a MAP client. The user just provides the code for implementing the procedures and the main procedure that invokes them. MAPGEN works as follows. It:

1. reads the MAPDL input file to determine the type of each procedure (local, remote, or remote mobile),
2. checks whether the necessary source files for each procedure exist,
3. produces the program code (.java files), and
4. creates a Makefile to be used in the final compilation (.class files).

The steps for creating a MAP client are illustrated in Figure 4.

An Example. Assume the following application: meteorological data are selected from various remote stations and used to make a prediction. There are three main procedures:

- *Hostinfo get_data(URL host)*: that selects the data at each host,
- *Forecast compute_forecast(MeteoData data)*: that makes the prediction using the data selected (this is a resource-demanding task),
- *void print_forecast(Forecast)*: that prints the results.

The *get_data* procedure is a host-specific remote procedure. The *compute_forecast* procedure that is CPU-consuming is implemented as a mobile remote procedure. Finally, the *print_forecast* procedure is a local one. To code the application, the user creates the MAPDL file called $meto.map$ (Figure 5) that includes the definition of the types of the procedures. This file is given as input to the MAPGEN generator. The user also specifies three files namely the *get_data.impl*, *compute_forecast.impl* and *print_forecast.impl* files that include the implementation of the three procedures as well as the *main.impl* file that invokes them (Figure 5). The rest is left to the MAPGEN compiler.

4. Related Work

MAPs provide a simple annotation to a Java program towards utilizing the available networked resources. The idea of process migration and load balancing in workstation clusters is not new. Recent approaches include Condor [1], Globus [2] and the NOW projects [4]. Most of these approaches require users to have login access to the machines as well as maintaining binaries for all architectures participating in the computation.

There are many recent research proposals, complementary to ours, that utilize Java towards using available networked resources in a seamless manner. MILAN [3] is a research project aiming to provide services for transparent management and utilization of networked resources. In the MILAN project, Charlotte [8] provides a Java-based infrastructure for metacomputing on the web. It provides a distributed shared memory abstraction at the programming language level, load balancing and fault masking, through eager scheduling where a task can be submitted to several servers. The Java Market [6] follows a web-centric approach: users can contribute their machine's computational resources by just pointing a Java-capable browser to the Java Market web page, while similarly, they can launch jobs to the system by posting them on the Web.

Other projects include Javelin [10], ATLAS [7], SuperWeb [5] and ParaWeb [9]. Javelin [10] and SuperWeb [5] allow machines connected to the Internet to make their idle resources available to remote clients through Java-enabled Web browsers. ATLAS [7] combines Java and the Cilk programming model to allow the execution of parallel multithreaded programs on networked computing resources. Paraweb [9] provides extensions to the Java programming

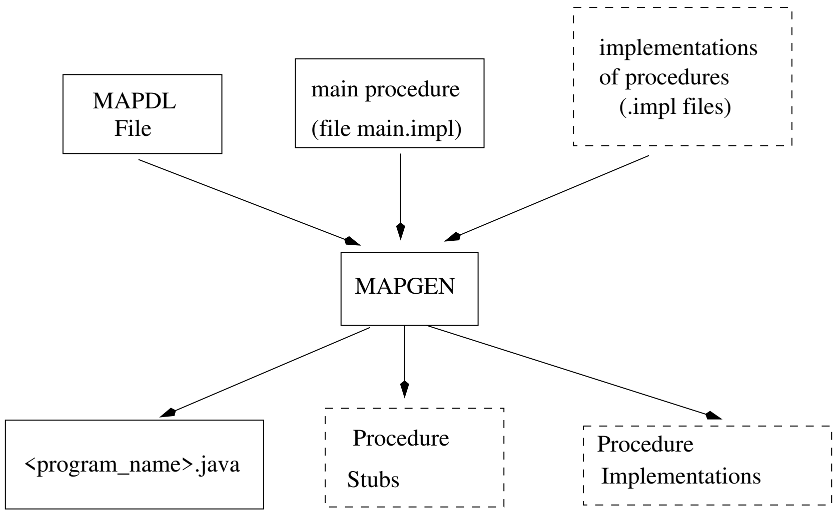

Figure 4. The steps for creating a MAP client

environment and the Java runtime system that allow programmers to develop new Java applications that exploit parallelism.

5. Summary and Future Work

We have designed and implemented the MAP system: a system that allows procedures to be executed at the most appropriate site among networked connected machines. The main characteristics of the MAP system can be summarized as follows:

- The MAP system is easy to use: the user just defines the type of each procedure.
- The MAP system is platform and operating system independent.
- MAPs are asynchronous thus they work well with slow networks or intermittent connectivity.
- MAPs are appropriate for light-weight clients.
- Load balancing is attained.

We are currently looking into various ways of improving our system. The issues we are looking into include: the frequency of collecting load information and of transferring this information to clients, more sophisticated ways of calculating the system load, and means of accounting and billing for the server resources used by clients. We are also investigating the integration of MAPs with our mobile agent prototype for web database access [13].

References

[1] The Condor Project. www.cs.wisc.edu/condor.

[2] The Globus Project. www.globus.org.

[3] The Milan Project. www.cs.nyu.edu/milan/milan/index.html.

[4] The NOW Project. now.cs.berkeley.edu/.

[5] A. D. Alexandrov, M. Ibel, K. E. Schauser, and C. J. Sheiman. SuperWeb: Research Issues in Java-Based Global Computing. *Concurrency: Practice and Experience*, June 1997.

[6] Y. Amir, B. Awerbuch, and R. S. Borgstrom. The Java Market: Transforming the Internet into a Metacomputer. Technical Report CNDS-98-1, Center for Networking and Distributed Systems, John Hopkins Univ., 1998. see also: www.cnds.jhu.edu/projects/metacomputing/java-market/.

[7] J. E. Baldeschwieler, R. D. Blumofe, and E. A. Brewer. ATLAS: An Infrastucture for Global Computing. In *Proceedings of the 7th ACM SIGOPS European Workshop on System Support for Woldwide Applications*, 1996.

[8] A. Baratloo, M. Karaul, Z. Kedem, and P. Wyckoff. Charlotte: Metacomputing on the Web. *International Journal of Future Generation Computer Systems*, 1998. To appear.

[9] T. Brecht, H. Sandhu, M. Shan, and J. Talbot. ParaWeb: Towards World-Wide Supercomputing. In *Proceedings of the 7th ACM SIGOPS European Workshop on System Support for Woldwide Applications*, 1996.

[10] P. Cappello, B. Christiansen, M. F. Ionescu, M. O. Neary, K. E. Schauser, and D. Wu. Javelin: Internet-Based Parallel Computing Using Java. *Concurrency: Practice and Experience*, November 1997.

[11] IBM. Aglet Workbench. www.trl.ibm.com.jp/aglets.

[12] J. Kiniry and D. Zimmerman. A Hands-on Look at Java Mobile Agents. *IEEE Internet Computing*, pages 21–30, July 1997.

[13] S. Papastavrou, G. Samaras, and E. Pitoura. Mobile Agents for WWW Distributed Database Access. In *Proceedings of the 15th International Conference on Data Engineering*, 1999.

main.impl

```
Vector meteoData = newVector();
Vector hosts = new Vector();
//
// add all hosts

while (Enumeration e = host.elements(); e.hasMoreElements(); )
{
   HostInfo = get_data( (URL)e.nextElement() );
   MeteoData.addElement(info);
}
// ready to forecast
Forecast fc = new Forecast();
Fc = compute_forecast(meteoData);
print_forecast();
// end of main.impl
```

meteo.map

```
import meteo.*


program Forecast
{
   HostInfo get_data(URL host):host_specific;
   Forecast compute_forecast(vector MeteoData): remote_mobile;
   void print_forecast(Forecast forecast);
}
```

Figure 5. MAP example

[14] E. Pitoura and G. Samaras. *Data Management for Mobile Computing*. Kluwer Academic Publishers, 1998.

Appendix: The Syntax of MAPDL

<definition_file> → <import_definition>
<const_definition> <program_definition>
<import_definition> → ϵ |
<import_definition> "import" <import_file_id> ";"
<const_definition> → ϵ | <const_definition> "const"
<const_type> <const_id> "=" value ";"
<program_definition> → "program" <program_name_id>
"{"<proc_decl_list> "}"
<program_decl_list> → ϵ | <proc_decl_list>
<return_type_id> <procedure_name_id> "("
param_list ")" <modifier_decl> ";"
<param_list> → ϵ | <param_decl> |
<param_list> "," <param_decl>
<param_decl> → <param_type_id> <param_name_id>
<modifier_decl> → ϵ | ":" "local" | ":" "remote_mobile" |
":" "host_specific"

Notes
− The <import_definition> and <const_definition> declarations follow the exact Java syntax.
− If all procedures are defined local, the code produced does not invoke the aglets framework.
− The default type for a procedure is local.