Proceedings IEEE International Conference on Parallel and Distributed Systems

Implementation of MAP: A System for Mobile Assistant Programming

Stéphane Perret, Andrzej Duda
IMAG-LSR, Grenoble, France

Abstract

We have define a network programming model called *Mobile Assistant Programming* (MAP) for development and execution of communication applications in large scale networks of heterogeneous computers. *MAP assistants* are high-level interpreted programs that can move between nodes, create clones and report results. Their execution is asynchronous and persistent to take into account client disconnections and node failures. This paper presents the implementation of the MAP model using the World-Wide Web framework and the Scheme programming language. Our measures and the first experience with MAP applications show that significant performance improvement can be achieved by moving computation closer to data: both the elapsed time and network traffic are reduced.

1 Introduction

The advent of the World-Wide Web [4] has modified the traditional view of communication networks and distributed systems. Traditionally, communication applications were based on the familiar client-server paradigm and were deployed on local area networks. The emergence of information servers on the Internet and the explosion of the number of interconnected hosts have aroused interest in new network applications such as information access and mobile computing. Our goal is to provide a network programming model that supports new communication applications.

Large scale communication networks such as the Internet present several important characteristics that influence the design of support for communication applications:

- *large scale*: the Internet is comprised of over 30 millions of hosts and it is expanding very fast [8];
- *distributed information*: vast collections of data sources distributed over information servers like WAIS, Gopher, Archie, and FTP are available via the uniform framework of the World-Wide Web;
- *low bandwidth*: although communication links have increasing capacity, the growing volume of the network traffic limits the effective bandwidth that can be exploited by the users;
- *disconnected operation*: the majority of client machines are connected intermittently by modem connections;
- *server failures*: in a large scale network, the probability of a server failure or temporal disconnection is significant;
- *heterogeneity*: many different types of computers are connected to the Internet.

In such a large scale network, we want to be able to perform efficiently some complex tasks on remote nodes. Traditional programming models for network applications are based on message passing, remote procedure call (RPC) or remote object invocation. These communication paradigms are well suited for client-server environments on local area networks, but they do not fit well the characteristics of large scale networks presented above [11]. To address the characteristics, we need a model that:

- addresses the problem of scale by exploiting inherent parallelism;
- allows accessing information on distributed information servers;
- reduces network traffic by shipping function to data and allowing execution of complex remote tasks;
- provides asynchronous invocation and result reporting;
- guarantees continuous persistent execution in spite of server failures or disconnections;
- deals with heterogeneity by means of high-level interpreted programs.

We have defined a model called *Mobile Assistant Programming* (MAP) that addresses all these issues. *MAP assistants* are high-level interpreted programs that retain their state while moving between nodes and gathering results. This approach reduces network traffic and elapsed time by shipping function to data. MAP assistants can save their state to survive node failures. Unlike familiar network programming concepts such as RPC, MAP assistants do not require a client to be active during the execution of its requests by a server—the client node can activate a MAP assistant to perform some tasks on remote nodes and disconnect. The results can be obtained at some later time. The model also addresses the problem of large scale by exploiting inherent parallelism: MAP assistants can generate clones to perform complex tasks in parallel. We have implemented the Mobile Assistant Programming model using the World-Wide Web framework and the Scheme programming language.

In the remainder of this paper, we briefly summarize the Mobile Assistant Programming model (Section 2), present its implementation based on Scheme and WWW (Section 3), discuss related work (Section 4), and outline conclusions (Section 5).

2 Mobile Assistant Programming

To make the paper self-contained, we provide below an overview of the main concepts of the MAP model (for a more complete description see [9]). An application *activates* an assistant program on a remote node to accomplish a complex task. An interpreter interprets the assistant program and its execution state can be *saved* to stable storage (this action is called a *checkpoint*). In case of a node failure, assistants are *restored* from stable storage automatically. To accomplish its task, an assistant *moves* to a remote node, *clones* other assistants and *reports* results. The results are collected in a *result collector*, a persistent object located on any node in the system and unique for an activation. An application can *get results* from the collector. Figure 1 presents the principle of the MAP programming model.

2.1 MAP primitives

The basic primitives of the MAP model are as follows:

- The *activate* operation initializes the execution of an assistant. An application provides source code to interpret and specifies the node where a collector object is created. A capability for the collector is returned to the user for later retrieval of results.

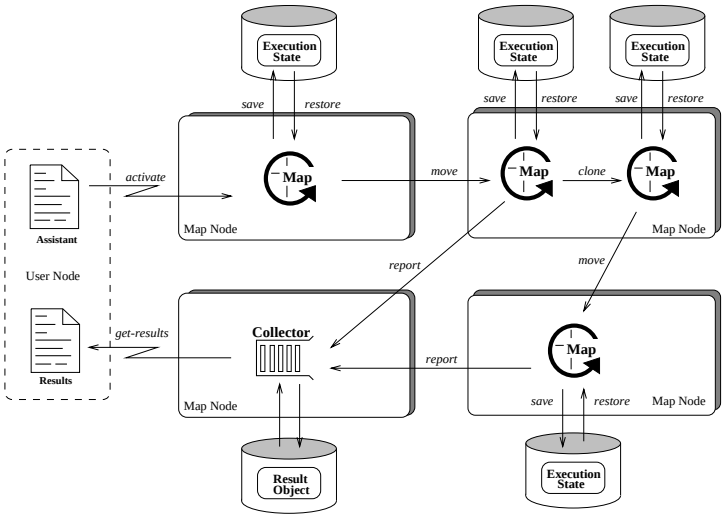

Figure 1: Principle of MAP

- The *get-results* operation uses the collector capability to retrieve the results of an execution. The operation detects the termination of all assistants associated with the given collector. If all the results are not yet available, the operation returns partial results and status *not-terminated*. If all the reports are available, the operation returns the results and status *terminated*.

- The *move* operation moves an assistant to a remote node. The execution state of the interpreter is saved to stable storage, moved to the node using the *post* system operation and restored at the target node. A checkpoint is created on the target node and the assistant resumes execution. After the operation, the current state of the assistant is confined to the target node.

- The *clone* operation creates a copy of an assistant with a given identifier. Both the creator and the clone assistants are checkpointed. The clone assistant begins its execution independently from its creator. The creator knows all its clones.

- The *node* operation returns the identifier of the node where an assistant executes.

- The *identity* operation returns the identifier of an assistant.

- The *print* operation puts the value of an object to the *result* buffer. The buffer, which is a part of the execution state of an assistant, accumulates partial results.

- The *report* operation reports the result buffer to the collector object using the *post* system operation and clears the buffer. A checkpoint is taken after reporting.

- The *exit* operation stops the execution of an assistant. The information about all clones created by the assistant is reported to the result collector object, so that the termination of all assistants can be detected.

2.2 System support operations

MAP primitives require some system support for checkpointing and communication. They are used by MAP primitives, but they are not accessible by the programmer. The system support operations are as follows:

- The *save* operation creates a checkpoint by saving the execution state of the interpreter to stable storage.

- The *restore* operation resumes execution by restoring the state of the interpreter from stable storage. This operation is also done automatically after a node failure for all active assistants - they are restored from the last checkpoint.

- The *post* operation transfers data to a remote node. The operation takes into account node failures or disconnections and it tries to deliver the data even in such cases. It implements an authentication scheme.

3 Implementation and performance

We have implemented the Mobile Assistant Programming model using the World-Wide Web framework and the Scheme programming language[1]. As we are interested in applications that access data distributed over the Internet, WWW was the obvious choice for the first prototype. Two features of WWW have enabled fast prototyping of MAP: the HTTP POST method and the WWW extension mechanism based on CGI scripts. The POST method provides a means for transferring information between WWW servers and CGI scripts provide support for activating MAP assistants.

Scheme has many advantages: its a fully fledged programming language with first-class procedures. The source code for its interpreter is widely available and it can run on many heterogeneous platforms [12].

Our implementation is based on three elements:

- *interpretation support*: to interpret MAP assistant programs, we provide a modified Scheme interpreter.

- *communication support*: to transfer data (the state of assistants, result reports) between nodes we use the standard WWW HTTP protocol.

- *distributed execution support*: to implement basic MAP functions such as *move* or *report*, we provide MapServer, a CGI script.

In our implementation, each WWW node that supports assistants runs the *httpd* daemon and provides MapServer, a CGI script with the following components: the modified Scheme interpreter, functions implementing MAP primitives and a local service interface.

Developing an application using the MAP platform consists of writing an application front-end and a program for an assistant. The application front-end activates the assistant on a chosen MAP node and specifies a result node that holds collector object for results. An application may run on a nomadic or a mobile node as well as a fixed-location node. The execution of assistants takes place on MAP nodes set up on WWW servers. Figure 2 present the structure of the implementation.

3.1 Scheme interpreter

To implement the MAP model in Scheme, we have modified the Scheme interpreter. We have removed some Scheme functions that are *unsafe*, such as interactive primitives (`reset`) and primitives for operating system access (`open, read, write`). We have added MAP primitives as well as a set of primitives for accessing and processing HTML documents.

Persistent execution of MAP assistants depends on the ability of the Scheme interpreter to save and restore its execution state. To provide this ability, we have added a module to the Scheme interpreter (written in C) that provides basic functions for implementing *save*, *restore*, *move*, *clone* and *report*. Figure 3 presents the execution state of the Scheme interpreter which is composed of the stack, the registers, the hash table, the heap and the code of primitives. The interpreter reads instructions from the source code one by one and compiles them to an internal representation composed of Scheme objects stored on the heap. The objects are referenced through the hash table. The

[1] activation of some example assistants can be made at `http://fidji.imag.fr/map.html`

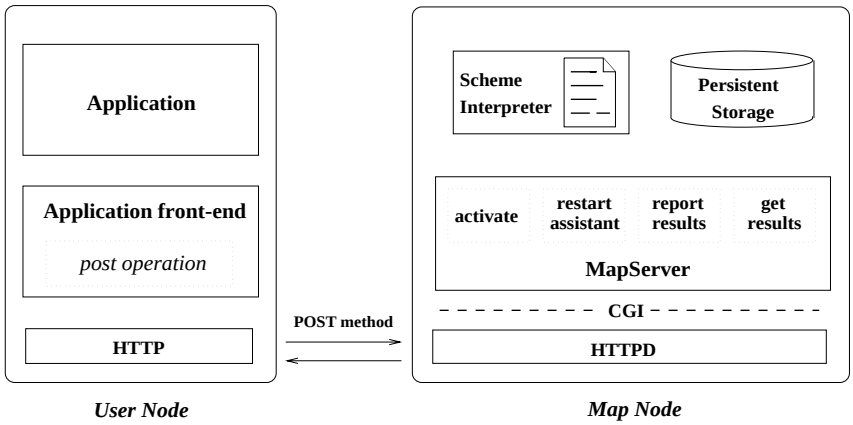

Figure 2: Implementation of MAP

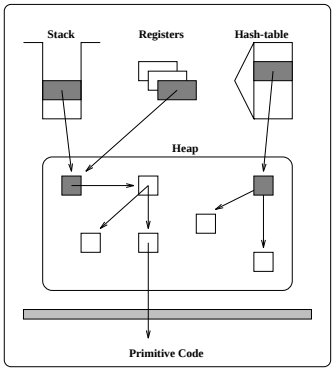

Figure 3: Scheme interpreter execution state

stack and the registers represent the current state in evaluation of instructions.

Saving the interpreter execution state to a checkpoint file is done as follows. First, we save the objects on the stack, the objects referenced by the registers and the hash table. Then, we run over the heap to save all the referenced objects in a way similar to garbage collection (we skip over all the objects that are not referenced). Finally, we save the code of the remaining portion of the program to be interpreted and the code of all source files loaded by the `load` primitive. Restoring the state consists of rebuilding all the objects of the interpreter as they were at the instant of saving. Objects are not necessarily restored at the same addresses. Restoring is done in two phases. First, we allocate memory for all the objects and then we restore the content of objects from the checkpoint file. Object references are swizzled to reflect current object addresses.

3.2 Communication

The *post* system operation transfers data to a remote node. It is implemented using the standard HTTP POST. The data may be either the source code, the encoded execution state of an assistant or the result report. The data are signed digitally and sent via the HTTP POST to a remote MapServer CGI script. The *httpd* daemon on the remote node starts the script and passes the data to the MapServer in an environment variable. The *post* operation copes with communication failures and node disconnections.

Data transfer makes use of the authentication scheme based on the PGP digital signatures [2]. First, a 128 bits MD5 Message Digest [1] is calculated from the data to transfer. Then, a digital signature is formed by encrypting the digest and an electronic timestamp using a secret key. The digital signature is sent along with the data (which remain not encrypted) and decrypted using the public key of the sender. Finally, we compute the digest from the received data state and compare it with the decrypted one. In this way, we assure that the data have not been modified and that they come from the node authenticated by its secret key. The authentication scheme requires adequate key management. A MAP node maintains a list of pairs: public key, identity for each trusted user and MAP node.

3.3 Distributed execution support

We present below the implementation of main MAP primitives.

The *activate* primitive takes the source code of an assistant and transfers it to the MapServer of *Activation Node* using the *post* operation. The activate function of the MapServer initializes the assistant by

starting the Scheme interpreter and initializes the result collector object on *Result Node* implemented as a file accessible by the *get-results* primitive.

The *get-results* primitive retrieves results by transferring a collector capability to the MapServer of *Result Node* using the *post* operation. The get-results function of the MapServer accesses the collector result object, checks for the termination of all assistants and returns formatted results.

The *move* operation saves the execution state of an assistant to a checkpoint file and invokes the *post* operation to transfer the state to a remote MapServer. The *restart-assistant* function of the MapServer takes the execution state of an assistant, saves it to persistent storage, rebuilds the execution state of the Scheme interpreter and resumes the execution. Finally, it returns a status indicating a successful *move* operation and the checkpoint file at the origin node is removed.

The *clone* operation is entirely implemented as a Scheme primitive. It copies the current state of the execution and modifies the assistant identifier in the copy state. Then, it takes a checkpoint of the copy state as well as of its own state. Finally, it starts the Scheme interpreter to restore the state of the clone and to resume execution.

The *report* primitive transfers the result report via the *post* operation to the MapServer of *Result Node*. The report-results function of the MapServer saves the result report to the collector result object and the assistant is checkpointed to persistent storage at the current execution node. When an assistant *exits*, the information about all clones created by the assistant is passed to the MapServer of *Result Node*. Only one collector object is used to collect result reports for an activation, because it is common to all assistants of the activation.

3.4 Current status and performance

We have implemented all the MAP primitives and we are currently working on the authentication and resource control scheme.

We measured the performance of some MAP primitives during the execution of an example program. The size of the execution state for the program is 19.9Kbytes. The measurements were done on three machines: `socorro.imag.fr` (Sun IPC) and `fidji.imag.fr` (Sun IPX) connected to the same 10Mbits/s Ethernet, and `paris.lcs.mit.edu` (Sun SS10) separated from the two previous machines by 22 IP routers. First, we have measured local primitives such as *save*, *restore*, and *clone*. Their performance on different machines is given in Table 1.

Operation	`socorro`	`fidji`	`paris`
save	$292ms$	$134ms$	$85ms$
restore	$367ms$	$198ms$	$125ms$
clone	$390ms$	$218ms$	$140ms$

Table 1: Performance of local primitives

The *move* primitive was measured in three configurations: locally on one machine (`fidji`), between two nodes connected to an Ethernet (`fidji` and `socorro`), and between two distant nodes (`fidji` and `paris`). We have decomposed the elapsed time of the *move* primitive into two components: the elapsed time of *save* (measured above) and the elapsed time of transferring the execution state to a remote node and resuming execution (we call this time *transfer-resume*). In this way, we are able to compare the performance of the primitive to the HTTP POST method. Table 2 presents the *transfer-resume* time compared to the elapsed time of invoking an empty CGI script by means of the HTTP POST method.

We can see that the time to restore the execution state and resume execution (without communication) is around $2s$. Worse performance in the third column can be explained by the difference in the speed between `socorro` and `fidji`. The figures for the distant case show that the communication cost is the major component of the *transfer-resume* time. However, interpretation of the measurements is difficult because of their significant variability: for *transfer-resume* time, the minimum and maximum values are $11.78s$ and $23.79s$; for *empty HTTP POST* time they are $0.89s$ and $18.41s$. In conclusion, we can observe that the performance of *move* mainly depends on the underlying communication facility. The HTTP POST mechanism is perhaps not the most efficient one, however, we have chosen it for its widespread and easy prototyping.

4 Related Work

The idea of programs that move in a network has appeared several times in the literature. Worm programs [10] were designed to travel from machine to machine and do useful work in a distributed environment. *Safe-Tcl*, a subset of TCL commands, permits the delivery of active e-mail messages that interact with their recipients and take differential actions based on the recipients' responses [5]. Telescript is a software platform for *remote programming* in an *electronic marketplace* [13]. Its execution model, unlike

Operation	local	`socorro` ⟶ `fidji`	`fidji` ⟶ `socorro`	`paris` ⟶ `fidji`
transfer-resume	2.22s	2.68s	3.79s	14.35s
empty HTTP POST	0.44s	0.62s	0.74s	6.83s

Table 2: Performance of *move*

MAP, is based on *rendez-vous* between agents. An infrastructure for mobile agents similar to MAP has been recently proposed by Lingnau *et al.* [6]: agents written in TCL and Perl can execute on nodes that run an agent server. Several proposals for *mobile code* have also recently appeared. Java [3] is an interpreted portable object-oriented language that allows a client to dynamically download classes and execute them within its address space. Omniware [7] is similar to Java and provides a variant of C++ with safety protection based on software fault isolation. Both proposals are based on the *remote evaluation* model [11] and do not provide any support for computation migration.

5 Conclusions

We have defined a model for the development and execution of communication applications in large scale networks of heterogeneous computers. The goal of the model is to enhance the ability of communication applications to perform complex actions in a large scale network by moving computation closer to data scattered over servers. The model takes into account all the important aspects of large scale networks like the Internet: large scale, distributed information, low bandwidth, disconnected operation, server failures, and heterogeneity. As a result, MAP assistants are mobile, persistent processes that work in parallel to accomplish a useful task.

We have implemented the Mobile Assistant Programming model using the World-Wide Web framework and the Scheme programming language. We have tested and measured the performance of MAP primitives. Our measures and the first experience with applications that use MAP show that significant improvement can be achieved by moving computation closer to data: both the elapsed time and network traffic are reduced.

References

[1] MD5, URL: `ftp://ftp.funet.fi/pub/crypt/-hash/mds/md5`.

[2] PGP, URL: `ftp://ftp.funet.fi/pub/crypt/-cryptography/pgp`.

[3] The Java language: A white paper, URL: `http://java.sun.com/` , 1994.

[4] T. Berners-Lee et al. World-Wide Web: The information universe. *Electronic Networking*, 2(1):52–58, 1992.

[5] N. S. Borenstein. EMail with a mind of its own: The Safe-Tcl language for enabled mail. Internet Draft, 1993.

[6] A. Lingnau et al. An HTTP-based infrastructure for mobile agents. In *Proc. Fourth International World-Wide Web Conference*, Boston, December 1995.

[7] S. Lucco et al. Omniware: A universal substrate for web programming. In *Proc. Fourth International World-Wide Web Conference*, Boston, December 1995.

[8] Special Issue on Internet Technology. *Communications of the ACM.*

[9] S. Perret and A. Duda. MAP: Mobile assistant programming for large scale communication networks. In *IEEE International Communications Conference 96*, Dallas, 1996.

[10] J. F. Shoch and J. Hupp. The worm programs–early experience with a distributed computation. *Communication of the ACM*, 25(3):172–180, 1982.

[11] J. W. Stamos and D. K. Gifford. Remote evaluation. *ACM Trans. on Programming Languages and Systems*, 12(4):537–565, 1990.

[12] W. Clinger and J. Rees. Revised report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 21(12):37–79, 1986.

[13] J. E. White. Telescript technology: The foundation for the electronic marketplace. Technical white paper, General Magic Inc., 1994.