MAP: Mobile Assistant Programming for Large Scale Communication Networks

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

Abstract

We have defined a new 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 allow client disconnections and survival of node failures. We have implemented the Mobile Assistant Programming model using the World-Wide Web framework and the Scheme programming language. Our initial experience shows that the MAP model is a useful paradigm for programming communication applications in large scale networks.

1 Introduction

Large scale communication networks such as the Internet provide access to various information resources and communication services. Despite some limitations such as low bandwidth, poor quality of service and insufficient network services, the Internet has many interesting features. Because of its large scale and variety of resources, it is in fact the first operational prototype for future Information Highways. The Internet presents several important characteristics that influence the design of support for communication applications:

- *large scale*: the Internet comprises over 30 millions of hosts and it is expanding very fast [11];
- *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 [4];
- *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 access resources efficiently, find useful information or perform user defined tasks on remote nodes. Traditional programming models for network applications based on message passing, remote procedure call (RPC) or remote object invocation 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 [17]. To address the characteristics, we need a model that:

- addresses the problem of scale by exploiting inherent parallelism;
- allows accessing information distributed over 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 and implemented a new model called *Mobile Assistant Programming* (MAP) that addresses all these issues. The implementation is based on the WWW framework and the Scheme programming language. We have tested and measured the first prototype with an example application that searches for relevant HTML documents on a set of WWW servers.

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

2 Mobile Assistant Programming

We consider a large scale network composed of nodes connected via communication links. Nodes are virtual processors with memory and secondary storage. As stated previously, we want to take into account the important characteristics of the networks like the Internet: large scale, distributed information, low bandwidth, disconnected operation, server failures, and heterogeneity. The goal of *Mobile Assistant Programming* is to provide flexible mechanisms for programming applications for accessing distributed data in such networks. The user view of MAP consists of *mobile assistants* that the user can *activate* to execute some operations on remote nodes and to *get results* of their work. Internally, MAP supports execution of assistants: it provides ability to interpret assistant programs, to migrate them to remote nodes, to create clones and to report results.

The characteristics of Internet-like networks greatly influenced our design:

- to allow disconnected operation, an assistant is an asynchronous process: the user *activates* an assistant and *gets results* at some later time;

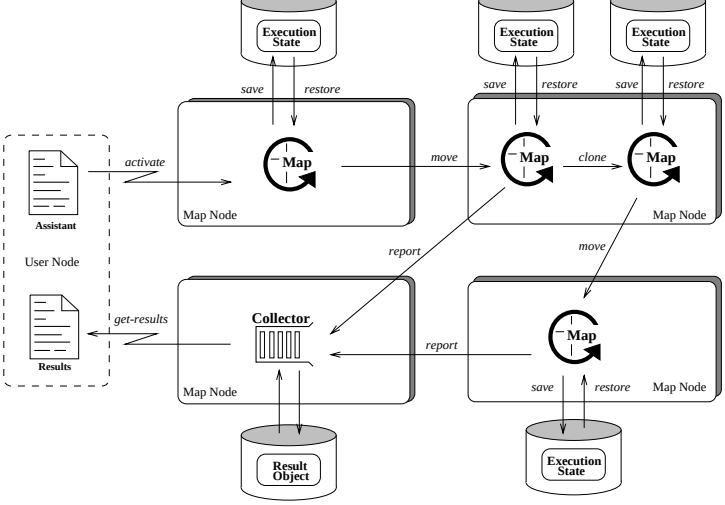

Figure 1: Principle of MAP

- to reduce network traffic, an assistant *moves* to a remote node and executes operations on data located on that node.
- to accomodate large scale, an assistant can *clone* other assistants that execute in parallel as autonomous and independent entities;
- to cope with node failures, an assistant is a persistent process: its execution state is *saved* and can be *restored* in the case of a node failure;
- to deal with heterogeneity, an assistant is a high-level interpreted program.

Figure 1 presents the principle of the MAP programming model. 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 persistent storage (this action is called a *checkpoint*). In case of a node failure, assistants are *restored* from persistent 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 2 presents the functional architecture of the model. It is composed of four layers: *application front-end* layer that provides user interface, activates assistants, and checks for results, *assistants* layer that performs work for the application and is programmed using MAP primitives, *MAP primitives* layer that provides basic MAP operations, and *system support* layer that takes care of checkpointing and communication.

2.1 Assistant Primitives

The assistant layer provides the following primitives (we present the syntax of each primitive):

- `activate` *source-code activate-node result-node*
  activate primitive initializes the execution of an assistant on a given node. An application provides *source code* to interpret and specifies a *result node* where a collector object is created. A capability for the collector is returned to the user for later retrieval of results.
- `get-results` *collector-capability*
  get-results primitive uses the collector capability to retrieve the results of an execution. The primitive detects the termination of all assistants associated with the given collector. The termination detection is based on the graph of assistant cloning constructed from the information provided by each assistant when it exits: the identity of its creator and the number of clones it has created. If all the results are not yet available, the primitive returns partial results and status *not-terminated*. If all the reports are available, the primitive returns the results and status *terminated*.

2.2 MAP primitives

The basic MAP primitives are as follows:

- `map-move` *node*
  move primitive transfers an assistant to a remote node. The execution state of the interpreter is saved to persistent storage, moved to the node using the *post* system operation and restored at the target node. The operation uses an authentication scheme to verify if the assistant is allowed to execute on the target node. If for any reason the assistant cannot move to the target node (for example the access is denied or *post* fails), the primitive returns appropriate status. A checkpoint is created on the target node and the assistant resumes execution. After the primitive, the current state of the assistant is confined to the target node. The primitive executes as an atomic action.
- `map-clone` *id*
  clone primitive 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 keeps track of all its clones.
- `map-node`
  node primitive returns the identifier of the node where an assistant executes.
- `map-identity`
  identity primitive returns the identifier of an assistant.
- `map-print` *object*
  print primitive puts the value of an object to a *result* buffer. The buffer, which is a part of the execution state of an assistant, accumulates partial results.
- `map-report`
  report primitive reports the result buffer to the collector object using the *post* system operation and clears the buffer. A checkpoint is taken after reporting.
- `map-exit`
  exit primitive 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.

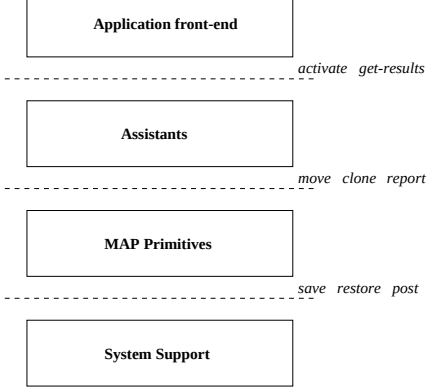

Figure 2: Functional architecture of MAP

2.3 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 persistent storage.
- The *restore* operation resumes execution by restoring the state of the interpreter from persistent 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 also implements an authentication scheme based on digital signatures.

2.4 Discussion

In the MAP model, assistants are asynchronous independent entities. They execute in parallel, independently of one another. There is no communication between assistants and each assistant reports results directly to the collector. Reporting is also asynchronous—each assistant can generate as many reports as it wishes at any time. A mobile host may access partial results or check for the termination of all assistants at will. These features of the model make execution of an application transparent to the underlying communication system. That means that an application can be executed on a nomadic or a mobile host as well as on a fixed-location node indifferently.

Persistent execution of assistants is based on *checkpointing*—the execution state of an assistant is saved to persistent storage. When an assistant executes a primitive that modifies its external environment such as *move, clone*, or *report*, a checkpoint is taken in the primitive. The primitive and the checkpoint are executed as one atomic action, so that an assistant can be restored after a node failure in a consistent state and continue its execution. After a failure, a node restores all the assistants that were active before a failure. This mechanism allow assistants to progress in computation in spite of node failures.

Security is the major issue in a programming model such as MAP, in which external programs execute on remote nodes. There are two problems that must be addressed: access control and resource control. Concerning access control, we assume that assistants execute in a standard Internet environment, so we need an authentication mechanism to verify if an assistant has rights to execute on a MAP node. We use the digital signature scheme provided by PGP [1] that allows a MAP node to verify if data transmitted over the network come from an authorized user or another authorized MAP node. We assume that we do not need systematic encryption for transmitting the code or the state of assistants. However, we guarantee that it is not modified and it comes from an authenticated sender. If an assistant needs to handle sensible information that should be protected for privacy, it can use a public key encryption scheme provided as a separate Scheme primitive. We also assume that MAP nodes are protected from external threats and they can store private keys needed for authentication in a secure manner.

Resource control has three aspects: data access, system functions, and resource consumption. The use of a safe, high level interpreted language can address the first two aspects—assistants do not have access to data nor to system functions. We have removed some Scheme functions that are *unsafe*, such as interactive primitives (`reset`) and primitives to access operating system facilities (`open, read, write`). Resource consumption control is enforced on a per user basis. According to the identity of the user that have initiated an assistant, we limit resource consumption at different levels. For example, we restrict execution time, memory usage, disk space and network traffic volume of assistants. User groups of different resource consumption levels are defined in a MAP node configuration file and are used to control resource usage.

To illustrate the MAP model, we present below an example of a MAP program defining an assistant that goes over a list of nodes and creates clones on each of them. Each clone is identified by a small integer and the first assistant has the predefined identifier `main`. The clones prints a message containing their identity and the execution node. Figure 3 presents the time diagram of the assistant activation on `node-1`.

```
(define (message msg)
        (begin (map-print (map-identity))
               (map-print " on ")
               (map-print (map-node))
               (map-print " : ")
               (map-print msg)
               (map-report)))
(define (name id)
        (number->string id))
(define (go-over id node-list)
        (if (null? node-list)
            (map-exit)
            (if (not (map-move (car node-list)))
                (begin (message "move failed")
                       (go-over id (cdr node-list)))
                (if (not (map-clone (name id)))
                    (begin (message "clone failed")
                           (go-over id (cdr node-list)))
                    (if (equal? (map-identity) "main")
                        (go-over (+ id 1) (cdr node-list))
                        (message "work"))))))
(go-over 1 '( "node-1"
```

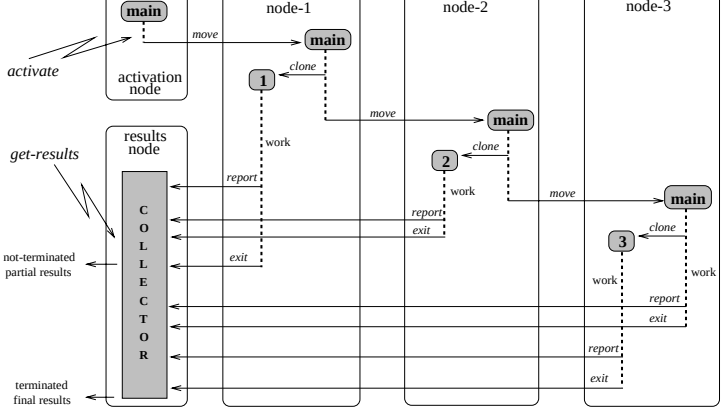

Figure 3: Execution of the example

```
"node-2"
"node-3" ))
```

Upon successful completion of the program, the collector contains all reports of the assistants, including those generated on *exit*. The application front-end will see the following results:

```
1 on node-1 : work
2 on node-2 : work
3 on node-3 : work
```

3 Implementation and application

We have implemented the Mobile Assistant Programming model using the WWW framework and the Scheme programming language. The implementation and measured performance of MAP primitives are described elsewhere [13].

We have tested and measured our implementation with an example application that searches for HTML documents on a set of WWW servers [14]. The application activates a MAP assistant that clones on each WWW server chosen by the user in a given domain. A clone accesses all HTML documents that can be found in the domain starting at a given root URL. Then, the clone parses the documents and matches them against the user query. All relevant documents are included in a report. The application collects the reports and puts the documents into a temporary local space of WWW documents. After the search is completed, the user can efficiently browse all relevant documents, because they are stored in the local storage.

We have compared the performance of our MAP application to the performance of a WWW robot that executes the same task. The robot retrieves every document from remote WWW servers and analyzes it locally. Table 1 presents this performance comparison. In this experiment, 121 HTML documents (of total size 1300 KBytes) have been accessed and 14 documents relevant to query `keyword:network` (of total size 150 KBytes) have been found. Clearly, moving document processing closer to data stored at remote WWW servers and transferring only relevant documents results in better performance.

MAP application	WWW robot
$540s$	$2277s$

Table 1: Performance of the application MAP vs. WWW robot

Our initial experience with MAP and the network application developed on top of it shows that significant performance improvement can be achieved by moving computation closer to the data. Our approach also allows nomadic computers to take advantage of the information resources on the WWW efficiently. A user can for example run several queries on interesting topics, disconnect and access the results of the search at some later time.

4 Related Work

Previous work can be broken down into two broad categories: *distributed programming* and *programming agents*.

Distributed programming has addressed the problem of exploiting remote resources since the advent of the first networks. Familiar paradigms include *message passing*, *remote procedure call*, *object invocation*, and *remote evaluation*.

Message passing allows development of a communication application as a set of communicating processes exchanging messages. However, because the programmer must take care of all problems at the network level, application development is a tedious task [3]. A remote procedure call (RPC) makes development easier, because it extends the traditional concept of a procedure call to execute procedure body on a remote node [5]. Despite many advantages, a remote procedure call fails to solve some semantics problems related to disjointed execution spaces and node failures [18]. Object-oriented programming tries to solve some of the problems by encapsulating code and data in an object as a system known entity that can be identified, invoked and migrated [7]. Both paradigms (RPC and object invocation) are well suited to distributed applications that use fine-grain operations on remote nodes in local area networks. However, they fail to address performance problems related to limited bandwidth—frequent invocations of fine-grain operations induce excessive communication traffic.

Remote evaluation (REV) goes a step further in addressing the problem of excessive network traffic [17, 16]. It is a technique that allows evaluation of a program expression at a remote computer. A client sends the server the body of a procedure and the arguments of the call, whereas in the case of RPC, only the name of a procedure is sent to the server. REV can improve performance by reducing communication between nodes: instead of a sequence of several RPC, REV can evaluate an expression composed of the sequence of procedure calls without any communication with the client. A similar concept was considered by Falcone [8].

Several proposals for *mobile code* have recently appeared. Java [2] is an interpreted portable object-oriented language that allows a client to dynamically download classes and execute them within its address space. Omniware [10] 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 and do not provide any support for computation migration.

The presented distributed programming models do not address disconnected operation, node failures and large scale. In all these models, the client application stays operational when some execution takes place on a remote node, because invocation is synchronous. This makes it difficult to support disconnected operation of clients. On the contrary, the MAP model provides asynchronous invocation that allows disconnected operation of a client.

Another important issue concerns node failures. In traditional synchronous programming models based on message passing or RPC, determining a global consistent checkpoint for a distributed computation is difficult because of dependency between interacting processes: the caller remains on a client node whereas the execution request is transferred to a server node. As a MAP assistant is an autonomous, asynchronous entity independent of any created clones, its state is only defined by the internal state of the interpreter and it is confined to a single node. Moreover, its state is independent of the state of other assistants that compose a distributed computation. So, unlike the synchronous models, a global consistent checkpoint in MAP can easily be determined as the union of the checkpoints of all the assistants. In practice, we do not even need the notion of the global checkpoint—each assistant can individually rollback to the last checkpoint and resume its execution.

The last issue concerns large scale. We need to address the problem of scale by exploiting inherent parallelism. Traditional synchronous models do not favor large use of parallelism. In MAP, the *clone* operation allows exploitation of possible parallelism in a large scale network.

MAP assistants are similar to *worm* programs [15]. Worm programs were designed to travel from machine to machine and do useful work in a distributed environment. A worm was composed of multiple segments, each running on a different machine. Maintenance mechanisms were responsible for finding free machines when needed and replicating the program for each additional segment. These techniques were successfully used to support several real applications such as a real-time animation system. The experience with the worm programs was more geared towards exploiting idle machines to run distributed and parallel computations, than providing a framework for new applications for large scale networks.

A *programming agent* is a concept that appears frequently in the context of artificial intelligence [9]. It denotes an active entity with a well-defined goal that communicates with its peers by exchanging messages in an expressive agent communication language. An agent communication language can be either declarative or procedural. The declarative approach frequently used in AI is based on the idea that communication can be best modeled as the exchange of declarative statements. Declarative programming has its advantages. However, as we do not follow this approach, we will not enter into detailed discussion of if here.

In the procedural approach, communication is seen as the exchange of procedural directives. Scripting languages such as *TCL* [12], *Apple Events*, and *Telescript* are based on this approach. For example, a subset of TCL commands - *Safe-Tcl* permits the delivery of active e-mail messages that interact with their recipients and take differential actions based on the recipients' responses [6]. Telescript is a software platform for *remote programming* in an *electronic marketplace* [19]. The basic entities are *agents* and *places*. An agent is an active object that carries data and procedures. Agents occupy places representing real world entities such as a shop or a box office. Agents can change places using a *go* instruction and communicate with other agents using a *meet* instruction. In the Telescript execution model, a client and a server are represented by their respective agents. When an agent wants some service to be executed on a server, it goes to the server place and meets the server agent to request the service. After accomplishing its mission, the agent returns to the client place and delivers results. With commercial applications as a goal, Telescript lays great stress on security: agents and places are identified by cryptographically authenticated *telenames* and resource consumption of an agent is controlled by means of *teleticks*.

The MAP programming model is similar to Telescript. However, in Telescript, the execution model is based on *rendez-vous* between agents. The details of the *meet* instruction are not given in the white paper [19] and it is not clear how a client agent requests a service from a server agent (some kind of a communication protocol must be defined between these two entities). In MAP, there is no server agent that represents a service. Instead, a MAP assistant migrates to and executes its code on a server node, so that MAP assistants can be thought of as mobile persistent processes programmed with a high-level interpreted language. Compared to Telescript, MAP offers assistant cloning to exploit inherent parallelism in a large scale network. Moreover, their execution is persistent to allow client disconnections and survival of node and data link failures. Returning results is also different in MAP: at each stage of the execution an assistant can report results.

5 Conclusions

We have defined a new 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 WWW framework and the Scheme programming language. Our initial experience with the first network application developed using MAP shows that MAP is a use-

ful paradigm for programming communication applications in large scale networks. We continue the work on the important issues such as wide deployment of execution support for MAP assistants, controlling execution resources, and security.

References

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

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

[3] G. Bernard, A. Duda, Y. Haddad, and G. Harrus. Primitives for distributed computing in a heterogenous local area network environement. *IEEE Transactions on Software Engineering*, 15(12):1567–1578, 1989.

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

[5] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Trans. on Computer Systems*, 2(1):39–59, 1984.

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

[7] D. Decouchant and A. Duda. Remote execution and communication in Guide - an object-oriented distributed system. In *Proc. 2nd IEEE Workshop on Experimental Distributed Systems*, pages 49–53, October 1990.

[8] J. R. Falcone. A programmable interface language for heterogeneous distributed systems. *ACM Transactions on Computer Systems*, 5(4):330–351, November 1987.

[9] M.R. Genesereth and S.P. Ketchpel. Software agents. *Communications of the ACM*, 37(7).

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

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

[12] J.K. Ousterhout. An X11 toolkit based on the TCL language. In *Proc. USENIX Association 1991 Winter Conference*, pages 105–115.

[13] S. Perret and A. Duda. Implementation of MAP: A system for mobile assistant programming. In *IEEE International Conference on Parallel and Distributed Systems*, Tokyo, 1996.

[14] S. Perret and A. Duda. Mobile assistant programming for efficient information access on the WWW. In *Proc. Fifth International World-Wide Web Conference*, Paris, 1996.

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

[16] J. W. Stamos and D. K. Gifford. Implementing remote evaluation. *IEEE Transactions on Software Engineering*, 16(7):710–722, July 1990.

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

[18] A. Tanenbaum. *Computer Networks.* 2nd edition, Prentice-Hall, 1989.

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