Chiron: a Framework for Mobile Agent System*

U. Ferreira

Abstract

In this paper, I introduce a complete and general solution for mobile agent systems on global environments such as the Internet, informally using a top-down approach without irrelevant details, and discussing all problems as well as presenting connected sub-solutions. The present solution is based on global computers and internets. In particular, I believe that without centralizing the security problem and closing the software from the public, one cannot guarantee a satisfactory level of security to the users in a feasible way. This may be viewed as an odd proposal as an academic work, but, in my opinion, this proposal is realistic.

1 Introduction

Any subset of an Internet-like network can be abstractly seen as a very large, inefficient and non-reliable computer, instead of a network. As an example, [5] is an updated bibliography for data mining research whose approach is clearly different from ours. Thus, one global computer[1] (GC) consists of an homogeneous set of interpreters (also called virtual machines) on a global network where application programs can run. I use here *Programming for a Global Computer* (PGC) to stress particular features of such a network, e.g. to be public. However, it is a good idea to keep in mind that a *general* purpose programming language ought to consider a global environment.

To date, there are not many mobile agent languages and systems, and information on them is easily found on WWW. Telescript[7, 8] by General Magic is pioneering and probably the best example of mobile agent technology. However, in addition to being suitable for different programming languages, the concept of identity in the present solution contrasts with the solution adopted by Telescript, where identity is simply a name. More recently, a few mobile agent technologies have been designed and implemented

*Thanks and regards to my friend S. Eglen, who read a previous version.

atop the Java Virtual Machine (JVM), such as Aglets Workbench[4] by IBM. However, because JVM is open, such systems are not secure enough for public applications on the Internet.

Section 2 briefly discusses programming for a global computer. Section 3 presents a general model for computation on a global environment by discussing problems and introducing solutions, and section 5 contains some concluding remarks.

2 Programming for a Global Computer

As global computers will never be so reliable nor efficient as local interpreters only, traditional programming might not be appropriate for global environments. Decisions must be made very often on WWW due to some delay, and this requires specific language constructs.

I recast here the characteristics of PGC from the programmers point of view: *long distances* is the first characteristics of global computers, hence *delays*. Programmers are *aware of locations* of resources. This not only allows the use of *resources* spread out on the global computer but also allows the control of distances between places, because the slowness of light is relevant here in performance. The programmer should be able to set a *time limit* under which his or her statement will wait for a remote operation to complete. If the operation was not completed by then, his or her program can try the same operation on a different site. *Fault* in connection is no longer regarded as exception but instead as a normal situation. Other requirements of PGC are *security* guaranteed by the underlying system, *code mobility*, *robustness* and *communication*, both synchronous and asynchronous. *Slowness* of the sequential computation is another characteristic. *Heterogeneity* of hardware and operating systems have been mentioned as typical but they are irrelevant in programming. *Parallel computation* is strongly suggested, in such a way that computing in a global environment may become even more *efficient* than local computation only.

I am regarding the above characteristics as being what defines PGC and, hence, any programming language or system for general purpose must provide at least all of those characteristics. Because of this, it turns out that no programming language or system to date is suitable for global environments, although there are languages on the top of mobile-code systems. Surprisingly, *concurrent programming* is not one of the requirements for PGC.

3 Problems

In this section, I analyze problems together with their solutions, because they are linked concepts and one solution to a problem may create other subproblems and so on. On the other hand, key words are emphasized for being searched.

I define *provider* a stationary agent in the present solution. Here, I define *Individual* as a mobile agent to stress that an Individual moves and are not split, although there are exceptional cases, as are explained later. To generalize, I refer to either *provider* and *Individual* as *agent* or *module*. The former is dynamic whereas the latter is static.

For philosophical reasons and to keep the solution uniform, both providers and Individuals contain themselves, that is, they cannot be broken into smaller modules but instead they are able to create other agents. The present solution always guarantees a unique identity for different agent although some attributes are shared after that creation. Moreover, providers do not create Individuals, and Individuals do not create providers. Symmetrically, providers and Individuals cannot be dynamically linked but instead they often *communicate*, providers with Individuals. In addition, providers can communicate with other processes designed by using different technologies. Thus, I refer to providers and Individuals in this solution as *self-contained modules* to distinguish them from *mobile agents* in some other solutions.

Some mobile code technologies have been designed and implemented but nowadays there are few applications that offer *security* in such a way that the public feel secure. Extending from [6] to PGC, it becomes even more critical. Thus, we provide security to protect agents and their interactions as follows: communication between hosts and between CEs; the host from malicious visitors; visitors from malicious hosts; communication between Individuals; communication between an Individual and a provider; Individuals from being bothered by other agents; Memory space. CPU from time abuse. However, incoming agents do not need to be granted access rights, as will be explained in this section.

The interpreter controls accesses by Individuals and providers to operating system calls.

The easiest and safest way to guarantee security in the solution is to initially forbid all critical operations and then allow only those operations that are necessary and do not violate the security policy. Input/output operations are regarded as such. Thus, the system guarantees that *no Individuals, by definition, perform directly input/output operations, but instead, communicate with providers that* may *perform the operations for the Individuals.* Therefore, providers must be able to recognize Individuals, which in turn,

require the definition of the concept of *agent identity*, or *identity* for short.

Authentication of messages is easily solved by digital signature, but this technique is used only when the sender is not the receiver. In the present solution, at this level, we can safely regard the system as being both the sender and the receiver, which simplifies the solution, also from the user's point of view. Here, there is no need for users to have public and private keys to use the system, because the authenticity is already guaranteed by the mobile agent system (MAS) by keeping one secret key (or pair of keys) for the whole system. I call this key *system key*, from now on. This also keeps the solution consistent because the person responsible for a host is not necessarily responsible for the computations that are sent from his or her host, as Individuals can move from host to host more than once. Notice that an Individual is not a static document.

Researchers adopt the idea that incoming computations must be granted access rights[6], which may be consistent but produces practical problems: 'access' is a very broad concept when we talk about security in global computers. The concept of access should not be limited to the level of operating system but, instead, be extended to services at higher levels. To allow an agent to access files is too risky, for example. We want to state a permission such as "Abelardo is allowed to read *xyz.abc* file on Thursdays", for example. At such a level of detail and flexibility, there are typically many permissions in an application system and, because of this, I do not adopt the approach of giving permissions beforehand. Instead, this approach allows providers to respond to Individuals requests in a programmable way, by accessing directly their identities and checking authorization on demand. Thus, all levels of access and services are controlled in a uniform way. Because there are typically so many permissions, declarative knowledge bases, e.g. composed by logic programming clauses, are natural candidates for representing such permissions.

In the present solution, the system has to provide a way of distinguishing Individuals from providers. This can be easily solved statically: if a program contains some **flyto** statement, the statement that causes the Individual proactive move, the compiler stamps the status of Individual in the generated code. On the other hand, if there are critical operations in the program, the compiler stamps the status of provider instead. A program cannot be Individual and provider, and, in case of both kinds of operation in the same program, the compiler reports an error message.

Because of the **flyto** special nature, it has to be a statement in the language, neither a method nor a library function. The same is true in the interpretable byte code: the **flyto** statement has to be generated as a corresponding virtual machine **flyto** instruction. This not only permits the

compiler and interpreter to distinguish Individuals from providers, but also allows the interpreter to deliver Individuals to the local *airport* for departure. A library function call does not normally provide information at that level of detail. Therefore, *all* critical operations in providers are no longer traditional library functions or methods, no matter their syntaxes: the compiler and interpreter have to be able to recognize such operations, otherwise the system will probably not provide satisfactory security in the real world.

This creates another problem: how to protect the programming system from potentially malicious compilers that generate critical operations in Individuals. My solution is to hide the virtual machine architecture and the Individuals format. Although this solution may cause surprise for being proprietary, it closes the architecture to the world in the same way as purely interpretable languages do. Furthermore, although it prevents other compiler designers from implementing other programming languages directly on the runtime system, it allows compilers to translate a program from another language to the source language owner of the interpreter. Moreover, the present solution does not require checking whether an Individual contains critical operations.

Identity in the present solution allows the public to even identify the programmer of an Individual, if necessary, as compilers could also have identities to be stamped in the generated code. The present solution does not discard the possibility of having different worlds, say global computers, that communicate, each world with its own programming language and implementation.

When a message arrives at a site, a program of the system called *airport* recognizes the format as an Individual, decrypts the incoming message by using the system key, verifies the Individual identity and verifies the integrity of the Individual. If everything is correct, then the airport verifies the Individuals passport, updates the passport and keeps a record about that arrival. If the message is not an Individual or the Individual integrity is not certified, the airport records the event and ignores the message.

When an Individual departs, the airport updates its passport, keeps a record about its departure, encrypts the Individual by using the system key, adds some header, and finally sends the package.

I divide *resources* in three classes: temporal, material and service. The first is CPU time plus some overhead due to instruction interpretation. The second may be persistent data while the third corresponds to responses to requests by agents. When an Individual arrives at a site, before running, the runtime system assigns a time interval for the Individual to leave the host, and this time is controlled by the system before interpreting every instruction. Thus, when an Individual leaves the host, its intervals of time in the host, including its effective time and total response time spent by providers, are

available. A similar solution is adopted for memory allocation, controlled by the interpreter.

Another way to prevent from time abuse in some specific applications is to allow or forbid, as a local policy, Individuals whose programs contain iterative statements, such as **while**. Another policy is to inhibit some interpretable operations, such as system library calls or method invocation. For example, square roots might not be calculated on a host that offers a simple and public commercial service. The *airport* can inspect for both policies at arrival time.

A kind of resource that can be regarded as both material and service is executable code, which also can be delivered by agents as any material resource. An Individual can carry and deliver such code in such a way that it can be executed remotely as long as the parties wish. This partially solves the problem of a few applications that require more efficiency than interpreters can provide.

A particular case of porting executable code is when a new version of a mobile agent system is released. Mobile agents can then visit sites to update all copies as long as this is part of some contract. In other words, unlike the real world, the approach does not prevent an Individual from carrying the whole mobile agent system, including the interpreter and the airport, installing it and even continuing running on it. In fact, an Individual can install any application.

Identity is partially generated by the compiler, some other fields are filled in when the computation starts, and passports are updated by airports when Individuals arrive and depart. Therefore, the format of object code is not the same as an Individual format.

Although some systems make use of passwords, for public applications, identity is not intended to be explicitly passed by the caller as parameter but instead its passing must be implicit and always guaranteed by the runtime system. I initially define identity as an abstract data type that contains the following fields:

Entity flag (whether Individual or provider); Program owner's identification; Home city; Internet Home Address (notice the country code); Postal Home Address (optional); File name of the Individual at home; Initial interpreter version; The initial interpreter Id; (Local) Date-Time when the program started running; The initial interpreter time zone (optional); Latitude and longitude of the first interpretation (optional); User's cryptographic key (optional); User's password for the application (optional); Password of the application (optional); The Individual passport (list of tuples about departure or arrival).

Identity is a class in the adopted language. Notice that this concept of identity is unchangeable and its presence is guaranteed by the present solu-

tion. This concept, however, is normally extended at the level of application, which will depend on the set requirements for security. Agents use *communication* to identify others at the level of application. Here, passwords can be used to identify Individuals.

An agent is not allowed to move an Individual but instead to request the latter to move, because every Individual is the responsible for its move. The exception is a situation when the MAS punishes an Individual because it violated some security policy. For *remote evaluation*, a programmer can write an Individual with enough code and data to accept the request and perform the operation remotely, while another program simply invokes the Individual. Therefore, there is no statement in the system for *shipping* an Individual. Accordingly, in the present solution, there is no concept of *downloading*.

An Individual is globally referred to by the ⟨Program owner's Id, Internet Home address, file name at home, sequence⟩ tuple which is unique.

For many applications, a provider normally sleeps and, when awaken by another program, performs some operations and sleeps again. An Individual, in its turn, moves to a host, requests some resources, perhaps it blocks while services are being provided, pays for services, moves on to another host and so on. Besides the communication is local to a host, I adopt a mechanism somewhat similar to method invocation. But its granularity is wider, i.e. the communication is between different agents, possibly written by different companies, with no assumption about static type correctness, nor even the existence of particular methods. Thus, compilers and linkers cannot see the whole picture, and this helps make global MAS feasible. The lack of a static global view is compensated by the ability to deal with partial information at the programming level, which is very important in any case.

In this solution, the concept of identity also allows a flexible scheme for porting material resources. As one example, an Individual I_1 might migrate from host H_1 to host H_2 without resources and, since at the destination, requests H_1 resources *from a provider* at H_2 which, in its turn, sends an Individual I_2 to H_1 that locally requests the resources from H_1 and take them back to H_2. The provider finally delivers the resources to I_1.

To date, almost all mobile-code languages adopt one or two fixed strategies to manage resources. Because I am looking for generality, in the present solution, whether the strategy is replication or sharing, or whether the replication is static or dynamic, or whether the latter is by copy or by move is entirely up to the programmer. For example, the Individual I_1 might migrate from a host H_1 to H_2 taking the resources eagerly.

Still regarding *communication* between programs, it can be *synchronous* or *asynchronous*. When the former is applied, it is possible for an Individual to migrate as part of its response to some message, while the calling process,

the running interpreter, is waiting. If the Individual comes back before the time has expired, it might give the response. Otherwise, the calling agent receives the special signal indicating that the value is unknown. Global MAS should take this mechanism into account. Thus, the solution guarantees safety and robustness on communication and migration.

From the point of view of a provider, a simple comparison between two references is enough to recognize an Individual, but some pattern matching between the identity fields can also be done. Identity cannot be changed by application programs at all, but it is public, which means that any program can access the caller's identity fields directly.

The Individual identity uniqueness is extremely important for both philosophical and security reasons. By no means, the language allows assignment to identity fields. Individuals never share identities and this is guaranteed by the system clock along with other fields.

Each interpreter has its unique Id, which is generated when it is delivered, and used by the system to authenticate Individuals *flights*. The same for compilers, thus allowing programmer's identification.

From the designer's point of view, a dynamic search for symbolic names in a program to identify the called object requires that the compiler writes the identifiers of the interface in the agent. Although Individuals get fat with so many names, it is a good idea for mobility and persistence to generate all identifiers, to be used by the virtual machine when saving and restoring contexts. Generating symbolically all identifiers also provides flexibility to the language.

By accessing methods and fields declared in the dynamic interface, providers and Individuals usually establish local communication according to the application. This requires the concept of *sender* and its key identifier. In the present language, **Sender** is used by the called program to refer to the calling one in the same way, and the presence of this key identifier in the calling program refers to the called one, i.e. the rôles are often reversed during communication, thus establishing a synchronous protocol. Therefore, every Individual or provider contains internally a stack of computations. In the present solution, a response may modify values that are used later by other responses.

Thus, the program interface (public objects, methods and fields) describes the objects of the program that can be accessed by another agent, local to the same host. This allows invocation of an external method that does not exist, for example. The same for accessing an external variable. Thus, there might be type mismatch between programs while they are running, and this condition must be dynamically checked and reported to the calling program. However, in the present solution, dynamic type mismatch is not regarded as

an error, although the corresponding operation is not performed.

The concept of *home* in Individual identity permits any agent to recognize the person who is responsible for that Individual by his or her Internet address. This person is also responsible for *all* messages sent by his or her Individuals.

The system can provide security against tempered interpreters and airports, although total security cannot be guaranteed. I adopt the following solution: one who develops the system has his or her own *police*. He or she often sends an Individual to each host that has an interpreter and then performs a privileged operation in the same programming language that permits access to the interpreter byte code, the same for the airport. Such Individuals can run an algorithm that checks whether the interpreter (or the airport) was tempered and then returns to the police host with the result along with some other pieces of information, such as the interpreter identity. The algorithm to check interpreters and airports can vary along the time. The checking result together with date and time is also of general interest because it *certificates* that the former host was not tempered at that time. Because many Individuals might wish to consult the police before critical transactions, the police itself should be organized in hierarchy, that is, the developer's police should be spread out over the network in such a way that every interpreter host belongs to some geographical region which is under its local police. Indeed, the police can provide a sophisticated and efficient distributed mobile system for the whole network.

In order to protect Individuals from malicious hosts, there can be two alternative solutions in this framework. In the first solution, the runtime system keeps a unique file of contexts for all blocked agents. While saving and restoring agents, the MAS always encrypts and decrypts the file by using the system key. The only kind of attack that can be done is to delete the file (therefore all saved agents at once), but the airport records information on both arrivals and departures of every Individual. The airport file is also encrypted and decrypted by using the system key, although the information is available by queries, both locally and remotely. The second solution for the problem of protecting Individuals from malicious hosts consists in assigning each Individual to one running interpreter, both forming one process of the operating system. Thus, two Individuals run in two processes, and so on. When an Individual sleeps, the whole process sleeps and the security of the Individual is shifted to the operating system level. It is desirable that new versions of operating systems will provide the concept of *public process* and forbid the normal user to kill it. Only privileged users ought to be able to kill such processes. I tend to adopt the latter solution.

In the security sub-model, I suggest that virtual machines and Individ-

uals format should be hidden, at least from normal users. In this way, the system protects Individuals against malicious hosts and malicious implementations. If operating systems provide and manage *public process* and forbid normal users to kill such processes running locally, then Individuals are pretty protected from malicious hosts, although not totally protected from malicious interpreters. The latter protection lies on that interpreters are general programs whose object code can be acquired from a public Network and that their developers are publicly known. Thus, this solution centralizes the responsibility of security towards MAS, while propose diversity. While in different models[2] encryption is seem as almost pointless if it is intended to protect the agent from the interpreter, here there is only one key for the whole system, to be used when it encrypts and decrypts agents, besides application keys. In security, I make use of the concept of centralization which considerably simplifies the solution because all users assume that the mobile-code system itself is reliable. In this way, some applications that require security do not normally require an extra agent to act as a *trusted third party* (TTP).

To protect Individuals while they are flying, airports encrypt them before departure key and decrypt them after arrival.

Privacy is another issue in this setting. Solutions that have been adopted by other programming systems can be used, for example, Java type modifiers are adopted as part of the present solution. In this scenario, privacy is also guaranteed at runtime. Dynamically, each variable (method) is public or otherwise.

For applications among known partners, programmers can write passwords in the **flyto** statement that is checked at arrival. This solution is similar to a login session on FTP, which can be anonymous or not, depending on the Internet account. Depending on the system implementation, this password may be encrypted and sent in some message before sending the whole Individual, as part of the system protocol.

Another problem that arises from programming for a global computer is *naming*, that is, to use services, the user who writes an Individual must know beforehand names in the interface of the providers that he or she will use in his or her program. If the Individual visits many hosts it must keep different names for the same service. On the other hand, a provider that wants to control accesses of resources by different kinds of Individuals, it has to keep a large database (or knowledge base) of identities and authorizations. This is a concern on programming languages design or AI techniques[3].

I consider *parallel computation* as one of the requirements for a general programming system for global environments and the reason is straightforward. Because communication to some Individual can be asynchronous, a provider may trigger more than one Individual that fly over the Internet to

run in parallel. It makes *strong mobility* one of the requirements.

One of the criticisms to the mobile agent paradigm is the fact that agents do not maintain connections upon migration. I do not think that this is a problem: first, connection is a matter of abstraction, that is, a connection that was interrupted at a lower level might be rescued in such a way that, for an upper level of abstraction, the interruption did not even occur. Therefore, an Individual can keep states of a connection at a lower level of programming in order to maintain the corresponding connection at a higher level. Although connections are not implicitly maintained by agents during migration at the language level, conceptually, what is more important is *communication*. I think that mobile agents is the most general paradigm, also when communication is the concern. Some remote communication can be indirect by local communication with providers that might perform the requested communication. This scheme improves security. Because Individuals can move more than once, in some cases, another alternative for keeping connections is to move the Individual to the host in order to communicate locally where the resource is.

Another characteristic of global computers is the existence of *failures and delays*. Developers want to program for such environment considering that failure or delay may be the result from remote operations. A similar problem that can be solved by this special value is due to special conditions during the communication between agents, for example, type mismatch or absence of symbolic identifier in an external program. The solution for these problems consists of adding a special value for each data type. *uu* in this context means that a value in the problem domain is not available at moment for some specific reason. The reason can be available as a MAS variable. Thus, unlike many distributed systems, I shift failures and delays to the programming level, and hence I provide language constructs for that.

In order to program with delays, *timeouts* are part of all language constructs that perform external operations.

4 Communication Between Individuals

As I discuss problems of dealing with resources here, communication may also be viewed as a kind of resource. In fact, communication is one of the keywords with respect to agents at the level of application, such as AI agent systems. Here I introduce a general solution for systems and, therefore, communication at such a level of application is outside the scope of the present paper. For my proposal, at the language level, to consider communication a resource suffices, while without going into those details.

Mobile agent communication is one of the issues where AI has much to contribute, and also because of this reason, now I leave the problem almost untouched.

However, briefly speaking, communication can be represented by representing *curiosity* properly for AI systems. Curiosity generates communication. It is a relatively complex task which I dare not comment here in this paper

5 Conclusion

I define a global computer in terms of its requirements. Taking them, it turns out that no programming language or system to date has been suitable for this kind of distributed computer.

In particular, I think that,

- by centralizing the sub-system for security,
- by making the agent format be private to the developers,
- and by closing the architecture of the global computer,

developers may provide *the only feasible way* to guarantee a satisfactory level of security on an open environment. In particular, it is desirable that the organization who propose their "club" also act as the unique trusted third party.

The present solution is based on a metaphor and describes typical situations with which travelers are used to dealing. Airports, arrivals, departures, passports, security and such natural concepts are in the present solution. Because the metaphor is based on modern life, it is expected that its implementation will be easy to use. Except for communication and language concepts and constructs, almost all problems were addressed here with a solution. In general, systems suggest or impose programming language concepts and constructs, and the present solution is no exception. However, I believe that I have explained the solution at such a level of detail that implementors can write other systems based on this original description, in particular, providing at least the same level of security as described.

References

[1] L. Cardelli. Global computation. *ACM Computing Surveys*, 28A(4), 1996.

[2] W. M. Farmer, J. D. Guttman, and V. Swarup. Security for mobile agents: Issues and requirements. In *Proceedings of the 19th National Information Systems Security Conference*, pages 591–597, Baltimore, Md., Oct. 1996.

[3] U. Ferreira. Intelligent agents for the internet. In *Proceedings of XIX Congress of SBC, ENIA'99-SBC*. Sociedade Brasileira de Computação, July 1999.

[4] D. B. Lange and M. Ishima. *Program and Deploying Java Mobile Agents with Aglets*. Addison-Wesley, 1998.

[5] J. F. Roddick, K. Hornsby, and M. Spiliopoulou. An updated bibliography of temporal, spatial, and spatio-temporal data mining research. In *Temporal, Spatial and Spatio-Temporal Data Mining*, volume LNAI 2007 of *Lecture Notes in Artificial Intelligence*, pages 147–163. Springer, September 2000.

[6] J. Vitek, M. Serrano, and D. Thanos. Security and communication in mobile object systems. In *Mobile Object Systems: Towards the Programmable Internet*, pages 177–200. Springer-Verlag, Apr. 1997. Lecture Notes in Computer Science No. 1222.

[7] J. White. *Telescript Technology: the Foundation for the Electronic Marketplace*. General Magic, Inc., 1994.

[8] J. E. White. Telescript technology: Mobile agents, 1996. Also available as General Magic White Paper.