Programming languages for mobile code*

Tommy Thorn
BRICS†
University of Århus, Denmark
thorn@brics.dk

December 1, 1999

1 Why mobile code

The expression ‘mobile code’ is used with various different meanings in the literature. Just to take three examples, let us cite:

- The term mobile code describes any program that can be shipped unchanged to a heterogeneous collection of processors and executed with identical semantics on each processor [1].

- ...mobile code, an approach where programs are considered as documents, and should therefore be accessible, transmitted and displayed {(i.e., evaluated)} as any other document [23].

- Mobile agents are code-containing objects that may be transmitted between communicating participants in a distributed system [13].

In this survey we will refer to mobile code as software that travels on a heterogeneous network, crossing administrative domains, and is automatically executed upon arrival at the destination. The administrative domain can be as big as a corporate network and as small as a personal hand held digital assistant. We believe that this characterization is general enough to encompass most usages and precise enough to exhibit the distinctive features of this technique. For example, it excludes the cases where code is loaded from a shared disk or downloaded (manually) from the Web.

Mobile code supports a flexible form of distributed systems where the desired non-local computations do not have to be known in advance at the execution site. The advantages of this model are many, including:

- In terms of efficiency: when repeated interactions with a remote site are needed, it can be more effective to send the computation to the remote site and to interact locally. This is especially the case when the latency of the network is high and the interactions consist of many small messages.

- In terms of simplicity and flexibility: the maintenance of a network can be much simpler when the applications are located on a server and clients themselves download them automatically on demand. Installing new or updated software becomes independent of the nature and number of clients. In some cases, it is even impossible to know in advance all the pieces of code that will be needed at a given site.

- In terms of storage: loading code on demand rather than having all programs duplicated on all sites can reduce significantly the total storage requirement.

We start with a short review of some of the most well-known examples of applications using mobile code.

- PostScript† is a page description language designed by Adobe Systems. PostScript is remarkable in that it is also a stack based programming language and it is associated with a large standard library suitable for page rendering. Printing on a PostScript printer consists

---

*This is a revised and reduced version of a paper published in the ACM Computing Surveys, 29(3):213–239, Sept., 1997.
†This work has been done in the context of Dyade (R&D joint venture between Bull and Inria).
‡PostScript is a registered trademark of Adobe Systems Incorporated.
of composing a program describing the pages to be printed and sending this program to the printer. The printer then executes this program and prints pages as a side effect. This example is a good illustration of some of the benefits that motivate mobile code: the algorithmic description of a complex image can be made very compact and general, independent of printer specifics, like printer resolution and number of available colours. Also, PostScript printers offload some of the work burden from the printing host. This compactness, expressiveness, and device independence has made PostScript become a de facto standard today.

- Database technology is another area where a form of mobile code has long been used advantageously. The size of a typical database makes it infeasible to transmit it in entirety to a client, thus any database operation must be communicated to and performed by the database server. Today, most commercial databases supports the ANSI standard query language SQL for database access. SQL offers a compact notation for expressing complex operations on multiple database relations.

- Documents with embedded executable contents transmitted on the network are another kind of mobile code. Multimedia documents in the Andrew System can include executable scripts written in an object oriented script language. These enriched documents can be sent as electronic mail or posted as news articles. The scripts are executed under user control and can present interactive dialogs and use graphical facilities. A similar extension for the Internet multimedia standard MIME has been proposed. This extension enables the use of embedded executable content written in Safe-Tcl [7], a restricted variant of the script language Tcl.

The usefulness of mobile code has also been realized for the world wide web. One of the problems with the web is that interactive pages are impractical in general due to network latencies. Even when latencies are not a problem, the content and appearance of web pages are constrained by what is expressible with the HTML language. Allowing embedded executable contents removes the constraints of HTML and network latency. Among the many existing proposals, the most well known is probably Java by Sun Microsystems.

- Finally, a fourth class of applications of mobile code addresses the problem of software distribution and installation, traditionally known to be costly and difficult. Lucent Technologies has developed Inferno [16], a mobile code enabled network operating system aimed mostly at media providers and telecommunication companies. Customer’s decoder-boxes or portable telephones running Inferno can be extended dynamically with software in response to their requests. In the same spirit, but in a different context, Sun has proposed the use of Java capable network computers to replace workstations in corporate networks. The benefit here is the low cost of maintaining an ‘Intranet’ of network computers, which download all applications on demand from an application server.

It should be noted that there are subtle differences in the usage of mobile code in the preceding four applications; in the PostScript and the database models, it is the sender of mobile code that takes the initiative of the communication, while in the Java and the Lucent models, the execution site takes the initiative to load the mobile code. We come back to this distinction when we present programming languages for mobile code.

In this survey we first identify the special concerns of mobile code and their impact on programming languages. In Section 3 we focus on the two most important issues: safety and security. With this background, we examine, in Section 4, three representative languages that have been proposed for mobile code. Section 5 draws the lessons of this study and compares the studied languages. We conclude the paper giving our perspective on the current state of mobile code programming languages and point out directions for future research.

2 Programming languages concerns

From the application classes of mobile code listed in the introduction, we can extract a number of common needs in terms of programming languages:

- The need for portability: inherent in the idea of mobile code is the notion of heterogeneous execution sites. It is not possible to have a specific version of the code for every possible architecture, thus the need for portability. The

---

1Intranet is a popular term for local networks using Internet technologies.
The need for safety: we use the word safety here in the sense that a bug in a safe application should not affect the execution of other independent parts of the environment. Limited assumptions can be made on code imported from an unknown source, which means that it cannot be trusted a priori to be safe, and special protection must be provided. Notice that this definition addresses the safety from an operating system point of view. Safety can also be imposed by appropriate restrictions at the programming language level. For example, free access to memory can be eliminated through a disciplined pointer model and systematic checking of the bounds for array accesses.

The need for security: as the mobile code is crossing administrative domains, special care must be taken in order to protect against security threats presented by mobile applications. The boundary between safety and security concerns is not always clearcut. Safety is mostly concerned with the behavior of systems in the presence of bugs, but as a lack of safety can be exploited for security breaches, safety becomes a necessary (but not sufficient) prerequisite for security.

We distinguish between four security properties [24]:

**Confidentiality**, also known as secrecy: it concerns the absence of leakage of private information (which often occurs through a covert channel, i.e., a channel that is not explicitly intended for communication).

**Integrity**, also known as accuracy: private data should not be modifiable by unauthorized parties.

**Availability**, the negation of which is known as denial of service: the attacker denies normal use of shared resources, for example by overloading them.

**Authenticity** guarantees that the identity of a communication partner can be trusted.

The need for efficiency: efficiency is almost always an issue for programming languages and their implementations. The special need here is for a minimal overhead for the measures taken to ensure portability, safety, and security.

Portability and efficiency are issues that have been studied in the programming language community for quite a long while. The safety and security issues are well known in the context of operating systems, but safety, and especially, security are issues that have not received enough attention so far in the area of programming languages. Security and safety problems take a new dimension in the context of mobile code. For this reason, we choose to focus on these issues in the next section.

3 Safety and security issues

Safety and security concern many aspects of a system. We will distinguish four levels at which to address these issues: the communication level, the operating system level, the abstract machine level, and the programming language level.

3.1 The communication level

In a top level view of a computer system, we consider a collection of computers connected with some networking technology. Safety concerns here require a robust protocol implementation that can withstand faulty or malicious communication partner.

For networks, like the Internet, where data can pass through untrusted intermediate hosts, the communication itself cannot be assumed to be secret or authentic. Therefore secure protocols based on cryptographic techniques are employed to guarantee confidentiality, integrity, and authentication, even on such networks. This is included in the new Internet protocol, IPv6, but there is also a large number of proposals that are layered on the top of existing protocols. Secure HTTP (SHTTP) and the Secure Socket Layer (SSL) are two examples. Availability is also an issue, but it is very difficult to deal with. This difficulty is illustrated by the many denial of service attacks on the Internet (see for example [11]).

3.2 The operating system level

Safety and security at the communication level is not sufficient in general. Handling safety and security is also a primary concern at the operating system level.

Safety is generally ensured through the use of hardware memory protection. This isolates a process from the rest of the system, leaving operating system calls as the only accessible interface. As no assumptions have to be made on the nature of the process,
this leaves a great degree of freedom to the implementation, e.g., to choose any programming language available. For mobile code, it has the problem of being very dependent on the operating system and the hardware, and is thus not portable. Even when using memory protection is possible, it may not always be desirable:

- Memory protection means that all communications have to cross a protection boundary, e.g., using a system call mechanism. This can be inefficient.
- Many smaller systems, e.g., personal digital assistants (PDAs), do not have the hardware needed.
- Requiring the use of memory protection makes embedding the mobile code environment into another application much more complicated and often impossible.

Confidentiality and integrity can, to some degree, be achieved by controlling processes’ access to information and communication channels. Complete control of covert channels is very hard and rarely attempted in non-classified systems. A form of availability can be attained by using limits on resources, such as disk space, number of processes and memory usage, and using preemptive scheduling and timeout in locks. Authentication is usually established through an initial identification of the user (for example, using a password scheme) and maintained by data structures of the operating system, protected from tampering from user level processes.

3.3 The abstract machine level

The safety guarantees obtained through the use of hardware memory protection can also be realized using an abstract machine. Using a language independent abstract machine retains all the language independence of the operating system solution, but does not have the portability problems. In the simplest setting, the protection boundaries are enforced by an interpreter, performing all the needed checks at runtime.

In the Omniware model [1] the overhead of interpretation is eliminated through software fault isolation (SFI). Code for the Omniware abstract machine is translated almost directly into native machine code, but all memory accesses are translated to code that checks for accesses outside a given boundary.

The self-certified code (SCC) [21] technique goes even further and eliminates the overhead of the protection as well. Self-certified code is a pair of machine object code and a machine checkable formal proof. The proof demonstrates that the object code respects the execution site’s published (low-level) safety policy. This policy comprises a set of proof-formation rules, along with a set of preconditions. The correctness proof can easily be verified automatically and ensures that the code respects the (low-level) safety and can therefore run without run-time checks.

3.4 The programming language level

Another way to obtain the required safety is to sacrifice the language independence and use programs written in a safe programming language. Most modern programming languages guarantee against low-level errors through mechanisms like typing, restricted pointers\(^1\), automatic memory management, and array bounds checking. It is possible to go even further and use the language scope and access rules to protect the interface of resources. The gain here is that the security implementation, such as resource management and control, can be written in the source language and used as a library.

As an optimization, the high-level program can be compiled and type checked before being shipped as mobile code. The question then arises on how to make sure that the object code is really a non-tampered output of a correct compiler. Three techniques have been proposed:

- Using cryptographic signatures to reduce the problem to one of trusting the author. As we already trust major software producers enough to run their applications, this can be seen as continuation of current practice.
- Using cryptographic signatures to trust compilers. The idea is to ship the source to one of a small number of trusted compilation sites for compilation and certification [16, 23].
- Compiling to an intermediate language which can be (type) checked to verify the same constraints that are imposed on the source language. The success of this approach is dependent on the intermediate language being suitable for efficient verification and permitting an efficient abstract machine.

\(^1\)Restricted pointers are like references known from, e.g., Standard ML. The only valid operations on a pointer variable are dereferencing and assignment.
These techniques are not exclusive. For example combining the first two seems easily feasible as they require much of the same technology and infrastructure. Likewise, the abstract machine can include operating system aspects, and can be more or less language dependent. For instance, depending on the context, we can consider the system libraries either as part of the abstract machine, or as part of the language. The languages we examine below all use combinations of these levels, although this is generally not made explicit.

4 Programming languages for mobile code

We have chosen to focus on a list of representative languages for mobile code here. Space considerations prevent us from presenting all the relevant languages in the paper. Among the other programming languages for mobile code let us mention Limbo [16], JavaScript, and VisualBasic. The interested reader is referred to the bibliography for a more extensive overview of the field.

The first two languages studied, Java and Obliq, are general purpose languages, intended for general application development. The last Telescript is a special purpose language. We expect Java to be the best known languages amongst these, and therefore give it a more detailed treatment and use it as the reference point for the other languages.

4.1 Java

Java is a class-based object-oriented language created by Sun Microsystems, with an emphasis on portability and security. As an example of how to use Java for mobile code, JavaSoft has created the ‘applet’ model. Applets are small applications which are automatically downloaded and executed upon visiting a web page containing them.

The language

For the sake of clarity we distinguish the language level and the abstract machine level.

Language level: The language is based on a simplified variant of C++ with all unsafe and most complicated language features removed. The features which have been removed include unsafe operations like pointer arithmetic, unrestricted casts, unions, and features leading to unmaintainable programs like the C preprocessor, unstructured gotos, operator overloading, and multiple inheritance. Automatic memory management has been added, guaranteeing against pointer errors due to manual memory management and making usage of dynamic memory much simpler. Array and string types are built-in with range check of all accesses. Exception handling has also been added, favouring the creation of robust programs. Finally, to enable concurrency, Java provides threads and serialized methods, using a mutex-locking on the corresponding object.

Java includes a novel notion of interface types. Interfaces define a collection of abstract methods and constants with their associated types. A class can be declared to implement an interface, in which case it must implement all the abstract methods of the interface. Anywhere a value of an interface type is expected, a value of a class implementing this interface can be used. Interfaces are useful for a number of purposes: they can be used to hide the implementation of a class and to group classes with a common functionality without forcing them into a class hierarchy.

Java also uses a notion of package. A package groups a number of class and interface definitions. Unlike most module systems, Java packages are open ended, and can be extended with definitions not envisioned by their original creator.

The default visibility of class and attribute definitions can be changed with a visibility modifier keyword. A class can be declared final, disallowing subclasses of itself to be derived, abstract, disallowing instances to be created, and private, limiting the scope of the class declaration to the containing package. Attributes have four (ordered) levels of visibility: private, default, protected, and public. Private attributes are only visible from within the object itself, i.e., not in objects of a subclass or other objects of the same class. The default visibility extends visibility of the attribute to the package in which it is defined. Protected attributes further extend the visibility to subclasses of the defining class, potentially defined in another package. Finally, public attributes are visible everywhere.

Abstract machine level: The Java Virtual Machine (JVM) is a language dependent abstract machine that is close enough to Java that its object code can be checked to respect the language semantics. In addition to these static (load-time) verifications, the JVM must implement dynamic checks to guarantee the safety of the language. These are bounds checking on array and string accesses, checking casts to a more specific type, invoking methods on null pointers, etc.
Security

The Java language, as described, is a modern ‘safe’ language, guaranteeing that type and access rules are always respected. This in turn enables a low-level security policy to be expressed within the language itself. The visibility rules for classes and attributes play a crucial role to this respect. Indeed, the interface to local resources is provided by libraries, protected by the scope and visibility rules. Most resources requiring dynamic access control, such as the file system or access to the network, are controlled by a centralized security monitor, called the SecurityManager. The SecurityManager has an abstract type, which cannot be instantiated by an applet. All security related methods are declared final, so that applications and applets are forced to use the appropriate code. Without this protection, malicious applets could redefine the method in a subclass, potentially circumventing the security invariants. A final class enjoys even stronger protection, in that the inability to create subclasses also implies the inability to define new methods with access to protected attributes.

The most recent version of Java generalises this model and introduces the notion of permissions as first class values, enabling arbitrary library methods to be guarded against misuse [12]. A method, checkPermission is provided to verify that a given permission is granted in the current context. In order for this method to succeed, the code of all the methods contributing to this call must have been granted the permission. Operationally this is formulated as an inspection of the current runtime call stack. As this is sometimes too restrictive, a mechanism is provided to signal that a call is privileged, that is, takes the responsibility for all its callers. The effect on the permission checking is to limit the stack inspected to the part that follows a privileged call (including that call).

Linking

The loading of classes over the network is done by an object of the class ‘ClassLoader’. This object is created during initialisation and cannot be replaced by applets afterwards (it is part of the SecurityManager state). As attributes with a default or protected visibility are fully accessible within the package of their definition, these two visibilities would be of little use if applets could introduce classes freely in any package and thus avoid the intended protection. To prevent this, the ClassLoader protects a fixed set of packages from being extended by applets. The exact set is not specified, but includes java and sun. The ClassLoader also maintains a unique name space for each network source, separate from the name space for classes coming from the local file system. Network sources are currently distinguished based only on their symbolic address.

Classes can be loaded from the local file system, if they are present in a directory specified in the CLASSPATH variable. This variable is part of the Java environment configuration, and can be changed by the user before launching the network browser or Java client, but cannot be accessed or modified by an applet.

As class files loaded through the network cannot be trusted to be untampered and the abstract machine runs with few type checks, the bytecode is passed through a bytecode verifier, which checks that the object code respects the Java semantics: it ensures that the bytecode is in a valid format, that pointers are not forged, that access rules are enforced, that the operand stack is used consistently with respect to the types, and the parameters passed have the expected types.

Reflections

Java is a promising language with a tremendous market acceptance. Much of this popularity stems from Java’s unique combination of characteristics: close to C++, safe, portable, and concurrent, as well as supplying a rich base library.

Since the first presentation of Java, a number of ‘safety’ bugs have been discovered [9]. It is of concern that many of the sources of the bugs can be attributed to the vague nature of the definition of Java. Although the core language is seemingly simple, many details are in fact quite subtle. For example, in Java the integrity of the security depends upon applets not being able to instantiate subclasses of critical classes, like ClassLoader. This condition is checked at runtime by the constructor\(^1\), which throws an exception in case of violation. If the applet can catch this exception within the constructor, it has succeeded the instantiation, though the object will only be partially initialized. The subtle restriction imposed on the constructor to avoid these situations were checked by the compiler, but not enforced by the bytecode verifier in an early version of Java [9].

Java’s current security implementation can only be seen as a first step, as it has a number of shortcomings. For example, as noted in [4], it currently does not scale beyond simple applets. Many of the

\(^1\)The constructor is a special method that is devoted to initialize the object upon construction.
prospective applications for Java, such as the ones mentioned below, require additional local libraries. Unfortunately, there is currently no way to protect user-defined libraries from redefinition and extensions from applets. Only the system defined fixed set of packages are protected. The fact that packages in Java are always extensible makes it impossible to guarantee the security of a package based on its source alone; the semantics of the ClassLoader must be taken into account as well. This seems against the spirit of Java’s language based security. Furthermore, this can be a serious problem considering that the ClassLoaders of the major Java applet environments do not necessarily have identical semantics [4].

We strongly believe that a formal approach to security in Java could help avoiding most of these weaknesses and would result in a much cleaner and coherent design. Work on the formalization Java is underway though, and progress has been made on many aspects.

4.2 Obliq

Obliq [8] of DEC System Research Center, is a lexically-scoped, dynamically typed, prototype based language, designed for distributed object-oriented computations. Computations in Obliq are network transparent, i.e., they depend neither on the allocation site or on the computation site, but the distribution is managed explicitly at the language level.

The language

To support the network transparency, Obliq extends the static scope to the network: free variables of transmitted computations can refer to objects from the origin site. The language has three main characteristics:

- Any value can be transmitted between hosts, including closures and object references. Objects themselves are local to a site and are not considered as values, but object migration can be programmed with a combination of closure transmission, aliasing, and object cloning (see below).

- Obliq belongs to a class of object oriented languages called ‘prototype based’. In prototype based languages there are no classes, and objects are created by copying (cloning) existing objects (the prototypes). Obliq uses a simple variant of prototyping, called ‘embedded’ prototyping, which avoids all the complications of delegation based prototyping [18]. In embedded prototyping, all the methods valid on an object are contained in the object itself, that is, they are not searched for in a list of super-classes.

- Obliq is dynamically typed. Type errors are caught cleanly and propagated to the origin site.

An object in Obliq is a collection of attributes (named values). A simple point object p can be written {x => 3, y => 4}. There are four basic operations on objects:

Selection/invocation: using the value of an attribute or invoking a method, for example, p.x and display.plot(p).

Updating/overriding: changing the value or the method bound to an attribute, for example, p.x <- 4 and display.plot <- lineto. Notice that it is legal to change a value into a method and a method into a value.

Cloning: Cloning an object creates a shallow copy: the immediate values of attributes are copied, but structured values introduce sharing. For example, array elements are shared between the clone and the original object. Cloning is generalized to support mixing several objects with disjoint names. Using the given examples, clone(p,display) produces an object with at least the attributes x, y, and plot.

Aliasing: Attributes can be redirected to attributes in other objects via the mechanism of aliases. All selections and updates on an aliased field are done on the redirection target. For redirected method invocation, the ‘self’ object is the object containing the redirected target, not the object containing the alias. An alias itself can redirect to another alias. Objects consisting of only aliases are called surrogates (also known as proxies in other languages). For examples of aliasing and redirection, consider {x => alias x of p1} and redirect p2 to p end. The latter makes all attributes of p2 aliases of the corresponding attribute of p.

Objects can be protected against modification, aliasing, and cloning from outside the object using the protected keyword. Safe interfaces to objects can be constructed through a combination of protection and surrogates.

Concurrency is inherent in Obliq; processes can execute independently on distinct servers and processes can spawn new threads locally. To handle concurrent accesses, Obliq supports serializing objects.
An object is serialized if at most one thread can access an object or run one of its methods at any given time. This is realized using a mutex on the object, which is acquired when one of its methods are invoked, and released when the method returns. To avoid trivial deadlocks, operations and method calls from within the object itself are not subject to locking. For details, see [8].

Lexical scoping hides named values from outside a given block and run-time typing ensures that these scope rules are enforced. Extending the lexical scope to the network enables the use of scope rules to address security issues, like information hiding; a procedure executing on a foreign server has only access to its own parameters and free variables. Communications between two independent servers is mediated by a shared global name server, which allows servers to import and export local values. To have a procedure executed on a distant server, the name server is asked for the 'engine' object accepting procedures. For example, remote invocation can be programmed as follows (on the client side):

```plaintext
let mydisp=net_import("display", Namer);
mydisp.plot(p);
```

Here, the name server, Namer, is inquired for the value registered with the name display. After this call, mydisp is a reference to the display object, either locally or on a remote server, and it is treated like any other object. For example we can invoke the method plot with a variable p. This example assumes that some process has exported display with

```plaintext
net_export("display", display)
```

Object migration can be programmed using a combination of closure transmission, cloning, and surrogates. The following example is taken from [8]:

```plaintext
let migrateProc =
   proc(obj,name)
   let e = net_importEngine(name,Namer);
   let remObj = e(proc(arg) clone(obj) end);
   redirect obj to remObj end;
   remObj;
end;
```

To migrate an object obj to an engine, we first inquire a name server for a reference to the engine. Next, we remotely execute on this engine (e(.....)) a cloning operation of obj, resulting in a remote object. Finally, all attributes of obj are made aliases for the corresponding attributes of the remote object (redirect obj to remObj end).

The distributed object model is closely based on (and implemented with) the Modula-3 network objects [5].

**Security**

Besides the basic use of scope to control what is exported, no special provision for security in Obliq is provided at the time of writing [8].

**Linking**

Transmitted closures can use functions from the basic library, but do not otherwise gain access to names from the receiving site. Names are explicitly exported by passing them as parameters to the received closure.

**Reflections**

Using network-wide scope for distributed applications leads to an elegant and powerful model of programming. As object migration can be expressed within the language, it is possible to program autonomous travelling agents in Obliq. This is not possible in the model employed by Java. Without further experimental results however, it is difficult to evaluate the advantages and drawbacks. It seems that this model could be inefficient, leading to many small messages to be transmitted: one for each access to a remote object. Also Obliq seems to require a much tighter coupling of hosts to support a distributed garbage collection.

**4.3 Telescript**

Telescript [17] of General Magic is an object-oriented class-based language designed for network programming. Telescript is not intended as a general purpose language: it is intended as a specialized language for communication in the same way as PostScript is a language for printing. The system is based on a number of metaphors from the real world. The central concept in Telescript is the agent, which autonomously travels on the Telesphere (a Telescript network of engines) doing business on the behalf of its owner. The engine is a Telescript interpreter, with a collection of built-in classes and an engine place. Engines provide persistence of objects, even in the presence of a system crash. Places are stationary processes that can accept incoming travelling agents. Users can create their own places nested within other

---

1 Notice, that the engine specific information supplied as parameter arg to the migrated closure is not used here.
existing places. Resource usage can be tracked and charged to the responsible user.

**The language**

The Telescript language itself is class based and includes run-time typing. Classes can inherit from a single superclass and any number of mix-ins, abstract classes which cannot be instantiated. Mix-ins can themselves inherit from other mix-ins.

Use of classes can be restricted in two ways: a **sealed** class cannot be specialized, and an **abstract** class cannot be instantiated. Attributes of objects can only be accessed from the class itself and its subclasses, while public attributes are unrestricted. The operator **protect**, a novelty of Telescript, turns object references into protected references. A protected reference cannot be used to modify an object.

Agents are **processes** with a number of properties:

- The **telename** is a pair of an **authority** and a **process identity** which together name a process. The authority identifies the (usually human) Telescript user.

- The **owner** is the process that will own future objects created (except processes, which own themselves). This is usually the current process, but it can be temporarily changed. Objects not owned by any process are garbage collected.

- The **sponsor** is the process whose authority will be attached to and charged for new objects created.

- The **client** is the object whose code requests the current operation.

- The **permit** specifies the capabilities of the current process. A permit has a number of process parameters:
  - the **age** is the maximum life in seconds,
  - the **extent** is the maximum size of memory allowed to the process,
  - the **priority** is used to determine when to schedule the process for execution,
  - the boolean parameters **canCreate**, **canGo**, **canGrant**, and **canDeny** specify whether the process can create new processes, can travel, can raise the permission level of other processes, and can lower the permission level of other processes, respectively.

Agents are sent by invoking their **go** operation with a ticket, specifying the destination place and possibly the route to this address. If the destination accepts the agent’s authority and permits, the agent is sent together with its objects to the place and resumes execution within the new place. The effective capabilities of a process are computed as the intersection (minimum) of the four permits process, local, regional, and temporary. The local permit is imposed by the entered place, the regional permit is imposed by the engine, and the temporary permit can be imposed by the language construction **restrict**. Following we give an example of how to execute a method from an untrusted object, using a temporary permit:

```plaintext
paranoid := Permit();
paranoid.canCreate = false;
paranoid.canDeny = false;
paranoid.canGo = false;
paranoid.canGrant = false;
paranoid.age = *.age + 2;
paranoid.extent = *.size + 1000;
try {
    restrict paranoid {
        yourObject.yourSuspectCall();
    };
    catch failed: PermitViolated { ... };
    catch ...
}
```

First we create an new empty permit, named **paranoid**, which we initialize with very restrictive permissions. We set the maximal age to current plus two seconds and the allow it to allocate 1000 bytes of storage. The suspicious call, `yourObject.yourSuspectCall()`, is then executed in a **try** block using the **paranoid** permit. The **try** block enable us to catch violations, such as code running too long or using too much space.

Four built-in mix-ins are available for further protections on classes:

- Unmoved: objects of this class cannot be taken along with a travelling agent.
- Uncopied: objects of this class cannot be copied.
- Copyrighted: objects of this class can only be instantiated if authorized by a Copyright Enforcer object.
- Protected: objects of this class cannot be modified.

---

1This example is taken directly from [17].
Security

As is apparent from the above, security is an overall consideration, that affects most of the Telescript language. The permission model is elaborate and applies to resource consumption as well (the model can thus address denial of service issues).

Linking

Mobile processes in Telescript are run in a separate domain and can only interact directly with the engine in which they run. All interprocess access is mediated by the engine.

Reflections

The Telescript system includes a number of features to restrict the actions of agents, but they seem to suffer from a lack of a systematic design. It is not clear how to be convinced of the consistency of the implemented security restrictions.

An aspect unique to Telescript is that it tries to deal with denial of service attacks. Telescript agents have their own initiative to travel and are thus more powerful than Java applets, but in a sense, also more dangerous. It can be hard or impossible to control an agent once launched. An interesting aspect of Telescript is that the user does not have to be connected to the network while his agent is acting. The agent can finish its business and return to the user once he reconnects to the network.

5 Review and comparison

The main features of the languages presented in Section 4 are summarized in Table 1. Let us now review them in turn and use them as basis for a comparative study.

- Object orientation: in the context of mobile code, objects form a convenient entity in which to encapsulate data and programs to be sent on the network. They also serve as entities for grouping information with the same access restrictions.

- Concurrency: in a distributed context, the notion of simultaneous and independent computations is a natural one. For Java, support for concurrency is rudimentary; multiple threads of control and corresponding serialization is supported. Telescript add a rich support for network communication. Concurrency is also inherent in Obliq.

- Mobility: we can distinguish two different models of mobility:
  - We call code fetching (noted by ‘Fetch’ in Table 1) the model used by Java, in which the user downloads the code to be executed. The initiative is with the receiver of the code.
  - Mobile agents (Obliq and Telescript) are processes that can be programmed to migrate themselves, so the initiative is with the mobile code itself. We denote this model ‘Agent’ in Table 1.

- Safety: all of the studied languages are ‘safe’ in the sense that access boundaries expressed in the language are enforced. This is an essential property for a language if security issues are to be expressed using language constructs. Enforcing safety involves ruling out pointer arithmetics, checking array bounds, using automatic memory management, disallowing arbitrary casts, and dynamically checking casts that promote objects to more specialized classes.

- Trust in the object code: the safety of object code is based either on trust or (in the case of Java) on verification. Telescript agents are trusted based on their origin as the network is ‘secure’ and sender addresses can be trusted to be correct.

As the security policy for the object oriented languages is implemented using objects, it is interesting to compare the possibilities for restricting access to part of the objects in the different languages. Table 2 summarizes the visibility rules for the four object oriented languages studied in Section 4.

6 Perspectives

The informal treatment of both language and security aspects is a major problem with all of the studied languages. Mobile code is executed within a complete environment (the run-time environment of the language, the web browser, the operating system, the network, etc. . . .), so arguing about security enforcement is meaningless without a clear specification of the separation of the responsibilities among the various entities of the environment (what entity is assumed to ensure what property?). A number of the flaws discussed in [9] can be seen as a consequence of the lack of such a clear separation. For example,
<table>
<thead>
<tr>
<th>Language</th>
<th>OO</th>
<th>Concurrency</th>
<th>Mobility</th>
<th>Safety</th>
<th>Trust in the object code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>✓</td>
<td>✓</td>
<td>Fetch</td>
<td>✓</td>
<td>Verified object code</td>
</tr>
<tr>
<td>Obliq</td>
<td>✓</td>
<td>✓</td>
<td>Agent</td>
<td>✓</td>
<td>No provision</td>
</tr>
<tr>
<td>Telescript</td>
<td>✓</td>
<td>✓</td>
<td>Agent</td>
<td>✓</td>
<td>Secure network</td>
</tr>
</tbody>
</table>

Table 1: Programming language features

<table>
<thead>
<tr>
<th>Protection offered</th>
<th>Java</th>
<th>Obliq</th>
<th>Telescript</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Class protection</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No subclasses</td>
<td>final</td>
<td>protected</td>
<td>sealed</td>
</tr>
<tr>
<td>No instances</td>
<td>abstract</td>
<td>protected</td>
<td>abstract</td>
</tr>
<tr>
<td>Invisible outside its package</td>
<td>private</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>No outside updates</td>
<td></td>
<td>protected</td>
<td></td>
</tr>
<tr>
<td>No aliases</td>
<td>NA</td>
<td>protected</td>
<td>NA</td>
</tr>
<tr>
<td>Mutual exclusion</td>
<td></td>
<td>serialize</td>
<td></td>
</tr>
<tr>
<td><strong>Attribute protection</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No restriction</td>
<td>public</td>
<td>default</td>
<td>public</td>
</tr>
<tr>
<td>Invisible outside its package or subclasses</td>
<td>protected</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Visible in subclasses</td>
<td>default</td>
<td>NA</td>
<td>private</td>
</tr>
<tr>
<td>Visible only in the defining class</td>
<td>private</td>
<td>NA</td>
<td></td>
</tr>
<tr>
<td>Runtime protected reference</td>
<td></td>
<td>protect</td>
<td></td>
</tr>
<tr>
<td>Mutual exclusion</td>
<td></td>
<td>synchronized</td>
<td></td>
</tr>
</tbody>
</table>

*a*Object used as prototypes play the role of classes in Obliq. The restrictions only apply to operations from outside the object.

*b*Not applicable

Table 2: Class and attribute protection and the keyword used

In Java, classes loaded from the local file system are more trusted than classes loaded through the network, and thus the former have access to more dangerous operations. Here, the integrity of the system depends on both the local operating system and on the Java system. One attack exploited a flaw that made it possible to load classes from anywhere in the file system [9]. For this attack to succeed, it must be possible for the attacker to upload a file somewhere on the victims file system. This can often be done in a variety of ways, depending on the local operating system. Another attack allowed applets to connect to arbitrary hosts. The attack succeeded due to a weakness in the Java library, where an external name service was implicitly assumed to be trustworthy, which in fact it is not.

Current work on mobile code does not take enough account of the research done in programming language semantics [22, 25], formal methods in software engineering, like VDM [6] and B [15], formal models of security [3, 14, 19], or research on static program analysis [2, 10, 20, 26]. As a starting point, a semantic definition of the language would provide an important insight and emphasize the weak parts of its definition with respect to security. Such a definition would also enable formal statements for the security claims made by the proponents of the language. Having a semantic for the language would not be enough, though. Security is a global property, so a security model must take into account all aspects of the system supporting the execution of the code. This includes in particular the hardware, the operating system, the abstract machine, the module libraries, the security manager, and the browser. A security weakness in just one of these endangers the security of the whole system.

**Acknowledgements**

I would like to thank Daniel Le Métayer for encouraging this work and thoroughly reading and commenting on drafts of this paper.
References


