Secure mobile agents for the Pocket Companion

(Draft position paper)

Gerard Smit & Paul Havinga
University of Twente, dept. of Computer Science
P.O. Box 217, 7500 AE Enschede, the Netherlands
{smit, havinga}@cs.utwente.nl

Abstract
A mobile agent as proposed in this paper, is a program that can migrate once from a server to a Pocket Companion and vice-versa. In this way a Pocket Companion can receive programs relevant to its environment. By migrating to the location of a resource, an agent can access the resource more efficiently. This is particularly attractive for mobile computing, even when the network conditions are poor and unreliable, because it does not require a permanent connectivity.

Security is a significant concern for mobile agents, as a Pocket Companion receiving a piece of code for execution may require strong assurances about the agent’s behaviour.

The security aspect can be handled basically in two ways: by preventing the agents doing harmful things, or by detecting that it does, or intents to do harmful or unpleasant things. In practice, the best way to achieve a high security is to use a combination of several methods.

In our view the agents consist of: a piece of code, a profile and a signature. The profile defines its access control list, the resources the agent needs or expects to use, and its behaviour.

1 Introduction

1.1 Environment

This paper is written as part of the Moby Dick project\(^1\) [MobyDick 95]. In this project we develop and define the architecture of a new generation of mobile hand-held computers, so-called Pocket Companions. These devices are resource-poor, i.e. small amount of memory, limited battery life, low processing power, and they are connected with the environment via a low bandwidth network with variable connectivity. The system is designed for daily use applications, such as: guided tours, electronic payment, railway ticket payment, information retrieval, e-mail, voice-mail, paging applications, etc.

1.2 Mobile agents

In general a mobile agent is an autonomous program that can migrate under its own control from machine to machine in a heterogeneous network [Harrison 95], [Hartvigsen 95]. In other words, an agent can suspend its execution at any point, transport its code and state to another machine, and resume execution on the new machine. By migrating to the location of a resource, the agent can access the resources efficiently even when the network conditions are poor or the resource has a low-level interface. This efficiency, combined with the fact that an agent does not require a permanent connection with its home site, makes an agent particularly

\(^1\) The Moby Dick project is a joint European project (Esprit Long Term Research 20422).
attractive for mobile computing since roving devices often have low-bandwidth and unreliable connection to the network. The idea of transmitting executable programs between clients and servers has been proposed by other researchers e.g. Telescript [White 95] and Tcl/Tk [Ousterhout 94].

Security is a significant concern, as a client receiving a piece of code for execution may not know anything about the agent’s intentions and behaviour. The main security risk is the amount of privileges that the agent can obtain when entering the client. The problem is to find a useful compromise between the desire to isolate the agent execution environment from the system, and the need to provide sufficient privileges in order to accomplish the agent’s task. Security can be divided into issues related to network communication and issues related to the execution of an agent [Hartvigsen 96]. The network security deals with mutual authentication, integrity of the communication and confidentiality. Crytography can be used to provide secure communication. We mainly focus on security issues related to the execution of an agent on a client.

1.3 Moby Dick agents

In our context we use a restricted view on mobile agents. A mobile agent, as proposed in this paper, is static: the program can migrate only once. When an agent migrates from a server to a Pocket Companion it can receive programs relevant to its environment. Furthermore, the Pocket Companion can send an agent to a server to perform some task. This means that we exclude several distributed applications, e.g. distributed calculations. We have made these restrictions for simplicity and security reasons.

The main emphasis in this paper is on downloading code from a server to the Pocket Companion. The primary reason for downloading code to a Pocket Companion is that the application has interaction with the user at fairly high rate.

In this paper we mainly investigate the security problems of downloading code on a Pocket Companion. In this context we make the following assumptions:
1. In our view the agents run under the control of the user, i.e. the user on which the machine the agent runs, may refuse an agent or kill an agent.
2. The user only trusts its own hardware, but not the software, nor the downloaded agent of the server.

1.4 Advantages of mobile agents in relation to Moby Dick

There are a good number of reasons for using mobile agents on a Pocket Companion:

1. Reduction of power consumption.

Agents can be used to save processing power, and hence battery power. If a Pocket Companion has a relatively high computational intensive task, it can send an agent to a server to perform this task on his behalf. Moreover, agents can be used to save bandwidth. First, no bandwidth is used for the user-interaction because the user-interface is executed local on the Pocket Companion. Another advantage lies in the ability of an agent to perform information retrieval, filtering and real-time user interaction, while returning to the client only the relevant information (just like the Alta Vista search engine).

2. Environment awareness.

The machine can run environment aware software for example run a guided tour program in a museum and a ticket agent in a railway station. The user can also download personalized agents, e.g. a Dutch version of the guided tours agent in a Italian museum.
3 **Asynchronous interaction.**

The Pocket Companion might only be connected to a network intermittently, hence it has only intermittent access to the server. A downloaded agent may continue with its task, even when it is disconnected. Furthermore, because the programs we foresee often have a considerable user interface, so a fast response time is achievable.

4 **Extended memory.**

Memory of a Pocket Companion will not likely be sufficient to hold all programs that you might need. With agents environment aware programs can dynamically be initiated.

5 **Detecting malicious programs.**

Because for security reasons a visiting agent is constantly checked, the code of an unknown user can be run relatively safe on your Pocket Companion. Quite remarkably, these security measures are normally, during the installation of a new program, not taken!

2 **Security aspects and solutions**

Why would someone even want to interact with a piece of code for which nothing can be absolutely guaranteed? The answer is, obviously, that the person hopes that the program will be useful to him and that it will not do any harm.

Basically the security involved in executing an agent can be handled in two ways: by preventing the agents doing harmful things, or by detecting that it does, or intents to do harmful or unpleasant things. In practice, the best way to achieve a high security is to use a combination of several methods.

2.1 **Prevention**

Prevention can be achieved in several ways:

1 **Signature of the code.**

Agent must act on behalf of some principal\(^1\) that can be held responsible for its actions. Digital signatures can be used to authenticate the principal and/or guarantee that the agent has not been tampered with [Microsoft], [Inferno 96]. The agents maliciousness, whether deliberate or due to bugs, cannot be decided by any level of cryptography.

2 **Trusted compiler or interpreter.**

The compiler or interpreter only allows access to safe resources and a restricted set of library functions. The language’s type system should be safe - preventing forged pointers and checking array bounds. In addition to that, the system should garbage-collect memory to prevent memory leakage, and carefully manage system calls that can access the environment outside the program, as well as allow agents to effect each other. Examples are: Java [Java 95], Safe-Tcl [Borenstein 93], Agent-Tcl [Kotz 96], Telescript [Whight 95], and Phantom [Phantom 96].

3 **Capability list.**

A capability list defines which resources the agent will use [Mullender 86]. The list can be inspected before the agent is allowed to execute. This gives the user the ability to refuse the agent on beforehand. Once the agent is accepted and running it needs to be checked whether

---

1. A principal can be a computer system, a user, or an organisation. This does not include a compiler since a compiler itself cannot be held responsible.
it behaves according to the access control list. This can be accomplished in software at the system call to the OS, or - for some resources - in hardware with the help of for instance an MMU. The user-interface might give a problem because classifying resources according to how private they are is hard and the secrecy of a piece of information depends on by whom it is accessed, e.g. MCF [Levitt 95].

4 Restricted view of the name space.

In this method only a restricted set of resources can be used by the agent by giving it a restricted view of the possible resources. For example, in Plan 9 [Pike 92] this is accomplished by giving an agent a mount point in a tree based file system. It can only access resources below the mount point. Resources that are available to applications all appear exclusively in the name space of the application. This implies to data, to communication resources, and to the executable modules that constitute the applications. Security-sensitive resources of the system are accessible only by calling the particular modules that provide them.

5 Restricted environment

Another approach is to run an untrusted agent in a ‘safe’ or restricted environment. This is used, for example, by the Safe-Tcl interpreter, which has removed the ‘dangerous’ primitives of Tcl [Borenstein 93] and is also used by the HotJava class loader [Java 95].

However, none of these methods prevent the agent from doing unpleasant things like abundant CPU-usage.

2.2 Detection

Another approach is to use detection of maliciously behaving agents. As long as the user has the possibility to supervise what is happening he might be prepared to take the chance and let it access his computer. This approach has been proposed in [Rasmusson 96]. An untrusted program is not denied access to resources unless it misbehaves in some way. Instead of making up rules for what agents are allowed to do and how they are allowed to interact with the rest of the system, the user can be given information about what the agent is doing. The user will decide whether what the agent does is illegal. If the user makes an estimate of what the agent is or what it might want to do, the agent can be granted access to only those resources usually needed to accomplish its tasks. The user will be notified when the agent deviates from its expected behaviour. If the user can classify agents by what kind of program they are, rather than by which resources they should be granted, the demands on what the users need to know about resource consumption etc. could decrease.

This approach draws a lot from the work done in intrusion detection in computer systems [Lund 93]. It is used to detect anomalies in user behaviour or misuse of a computer. Anomalies behaviour is behaviour that deviates from ‘normal use’ pattern, e.g. accessing a certain number of files per minute. Misuse means that the weaknesses or flaws in the system are deliberately used to get unauthorised access to system resources.

The objective of detection is to find deviations from an expected behaviour pattern. The detection measurements can for example be based on: time (e.g. time of day or amount of time used), numbers (e.g. number of files or bytes read), or intensity (e.g. detect bursts of behaviour).

There are several ways to implement this detection. [Rasmusson 96] uses generic or neural network algorithms for finding patterns in audit trails. [Crosbie 95] uses autonomous sensors which are taught to recognise intrusive behaviour. A similar way is used by [Ko 94] through
monitoring code execution using audit trails, where the monitoring is with respect to specifications of the security-relevant behaviour of the programs.

3 Proposed Moby Dick agents

We believe that in practice, the best way to achieve a high security is to use a combination of several methods mentioned above. It seems attractive to combine several methods of both prevention and detection.

3.1 Security model

An agent must act on behalf of some principal that can be held responsible for its actions. In our view the agents consist of: a piece of code, a profile and the signature of the principal (see figure 1). The agent can be secured with cryptography when needed, for instance to allow only specific clients to execute the agent. The agent’s code can be either a machine language executable or an interpreted language. The signature is used to authenticate the agent’s principal and to guarantee that the agent’s code and profile have not been tampered with. The profile defines the resources the agent needs or expects to use. The profile is used for two purposes. First the Pocket Companion will use it - in interaction with the user - to decide whether the agent is allowed to use the requested resources of the machine. During execution of the agent, the profile is used to detect malicious behaviour. Although at first sight, the profile seems to have similar properties as Quality of Service parameters, the profile extends this view. The profile is not only related to the ‘common’ resources such as disk, network, display, but also power consumption, CPU usage, etc. A profile might even include the access rights of a particular file, e.g. reading a password file1.

When an agent is instantiated the profile will be presented to the Pocket Companion. Once accepted the agent gains access to the resources it needs.

3.2 Method

We propose to use several complementary methods to obtain sufficient security. We expect that intrusions due to holes in one security method will be detected by other methods. The user decides whether to accept or refuse an agent. The credibility of an agent depends on its profile and the identity (trustworthiness) of the origin of the agent. When an agent arrives at a Pocket Companion the following actions have to be taken:

---

1. Access rights to a file can also be gained with delegation certificates, which will authorise access to files [Helme96]. The certificates can be part of the profile.
1  **Signature verification**
   The user checks the signature to authenticate the agent and the origin of the agent and to
detect whether the agent has been tampered with.

2  **Profile acceptance**
   The profile is inspected and compared with a list of privileges of known agents, servers and
principals. This gives the user the ability to accept or refuse the agent. Agents that are not
known to the system can be allowed only restricted access according to the kind of agent
and its principal.

3  **Profile verification**
   Once the user has accepted the agent and is running it, it needs to check whether it behaves
according the profile. This can be achieved in a number of ways. The first method is to use a
secure interpreter or compiler. The agents are written in a language (Tcl, Java, Limbo, ..) that is interpreted or compiled on the Pocket Companion. The language only allows certain
restricted safe functions and libraries. Furthermore, to detect malicious behaviour the sys-
tem will support a number of detection mechanisms. It detects excessive resource consump-
tion of an agent beyond the accepted profile (e.g. power, bandwidth, memory or CPU
usage).

3.3  **Further study**
A main problem of the security policies described above will be the user-interface. The user
has to do a classification and security assessments of his resources, which also depends on the
origin of the agent. This can be a rather complex and annoying matter, and so it is hard make
the system secure. To simplify this system we can define several profile categories such as:
super use, payment application, normal use, file read only use, display only, etc. Research has
to be done to distinguish relevant categories. We further need to investigate the consequences
of our approach for the efficiency.
Some of the requirements mentioned above are already implemented in existing (prototypes of)
operating systems like Inferno [Inferno 96], Amoeba [Mullender 86], and Nemesis [Neme-
sis]. None of these can fulfil all of our requirements, or are nor suitable for a Pocket Compan-
ion. It should however be possible to implement a prototype of a useful agent system on top of
an existing system.

4  **References and related literature**

[Borenstein 93]  Borenstein N.S.: “E-mail with a Mind of its Own: the Safe-Tcl Language for Enabled Mail”

[Crosbie 95]  Crodbie M., Spafford E.: “Defending a Computer System using Autonomous Agents”, Proceed-

www.cs.dartmouth.edu/~agent/.

[Harrison 95]  Harrison C.G., Chess D.M., Kershenbaum A.: “Mobile agents: Are they a good idea?”, IBM
research report, 1995

[Hartvigsen 96]  Hartvigsen et al.: “The Virtual Secretary Architecture for Secure Software Agents”, PAAM96 -
The First International Conference and Exhibition of Intelligent Agents and Multi-Agents, Lon-
don, April 22-24, 1996 (see also: http://www.pegasus.esprit.ec.org/people/arne/publications/
paam96-f.ps)

Trust Management in Distributed Systems, Rutgers University, October, 1996. (see also: http://
www.pegasus.esprit.ec.org/people/arne/publications/dimacs96.ps)

[Inferno96] Information available on http://cruel.com/vanni/


[Limbo 96] Information available on http://cruel.com/vanni/


[Nemesis] Information is available on http://www.cl.cam.ac.uk:80/Research/SRG/pegasus/nemesis.html


