Analyzing non-functional properties of mobile agents

Pascal Fradet, Valérie Issarny, and Siegfried Rouvrais

IRISA/INRIA, Campus Universitaire de Beaulieu, 35042 Rennes, France
{fradet,issarny,rouvrais}@irisa.fr

Abstract. The mobile agent technology is emerging as a useful new way of building large distributed systems. The advantages of mobile agents over the traditional client-server model are mainly non-functional. We believe that one of the reasons preventing the wide-spread use of mobile agents is that non-functional properties are not easily grasped by system designers. Selecting the right interactions to implement complex services is therefore a tricky task. In this paper, we tackle this problem by considering efficiency and security criteria. We propose a language-based framework for the specification and implementation of complex services built from interactions with primitive services. Mobile agents, RPC, remote evaluation, or any combination of these forms of interaction can be expressed in this framework. We show how to analyze (i.e. assess and compare) complex service implementations with respect to efficiency and security properties. This analysis provides guidelines to service designers, enabling them to systematically select and combine different types of protocols for the effective realization of interactions with primitive services.

1 Introduction

Code mobility is gaining more acceptance as a useful and hopefully future technology [1]. In particular, the mobile agent technology is emerging as a new way of building large distributed systems. Here, we consider a mobile agent as an entity in which code, data and execution state can explicitly migrate from host to host in heterogeneous networks [3]. Other types of interaction exist; let us cite remote evaluation or the classical RPC-based client-server[1] (i.e. remote procedure calls). Functionally, all that can be implemented using mobile agents can also be achieved by using RPC communication protocols. The advantages/drawbacks of mobile agents compared to RPC are mainly non-functional [9]. The motto being to move the computation to the data rather than the data to the computation, mobile agents can improve performances by reducing the bandwidth usage. On the other hand, they pose new security problems, namely how to protect agents against malicious hosts (or *vice versa*).

[1] In our context, we consider them both as degenerate forms of mobile agents. Remote evaluations execute remotely the code on a single server. A RPC amounts to sending a request (not a code) to a primitive service.

In this paper, we are specifically concerned with the use of mobile agents for the construction of complex services. By complex services we mean client requests built from primitive services available over networks composed of various devices ranging from powerful workstations to PDAs. Clearly, there is not a single best interaction protocol to combine such a variety of primitive services. The goal of this paper is to guide the choice of the adequate protocols (mobile agent, remote evaluation, RPC or a mixture) depending on the efficiency and security properties that the complex service requires. There have been experiments comparing performance of those different forms of interaction, but to the best of our knowledge, the formal assessment of non-functional properties of mobile computation has not been addressed so far. Yet, the need for formalization is obvious when dealing with security properties. On the other hand, mobility complicates seriously the specification and analysis of non-functional properties. Thus, we decided to restrict ourselves to a simple language of services but to tackle the analysis and comparisons of interaction protocols using a formal approach.

We specify complex services as simple expressions where primitive services and treatments are represented as basic functions. These expressions are mainly used to specify dependencies. The specification also includes information about the place (host), and non-functional properties of basic services. The abstract expressions are refined into concrete expressions, which make implementation choices (number of agents and their route) syntactically explicit. Mobile agents, RPC, remote evaluation or any combination of these forms of interactions can be represented, analyzed, and compared with respect to performance and security properties. The notion of performance considered in this paper is the total size of data exchanged in the interactions, whilst the notion of security focused on is confidentiality. The proposed framework enables designers to analyze (i.e assess and compare) various implementations of a complex service and to select the one that suits best their overall design goals. In addition, our framework can conveniently be integrated in a design environment supporting the abstract description of service software architectures, which further eases the actual implementation of services.

This paper is structured as follows. Section 2 introduces the basic setting of our abstract model of interactions using two simple functional languages. It further shows how to automatically refine expressions from one language (the specification) to the other (the implementation). Section 3 addresses the analysis of the implementations of complex services with respect to efficiency and security. Section 4 briefly discusses the practical application of our model through its combination with the architectural description of service architectures. Section 5 reviews related work and concludes.

2 Functional models of interactions

We describe complex services (i.e. client requests) as functional expressions where primitive functions represent services (i.e. basic services offered by providers) or treatments (i.e. basic actions defined by clients). Such complex services can be

specified at two levels of abstraction. At the *abstract level*, a client request is a simple (abstraction-less, recursive-less) functional expression. This expression specifies the functional semantics of the request and makes the dependencies between basic services clear. At the *concrete level*, expressions can be seen as a collection of remote evaluation, mobile agent or RPC interactions. Compared to abstract expressions, implementations choices such as the number, the type, the path (or route) and the composition of interactions are now explicit. The benefit of concrete expressions is that they can be easily analyzed with respect to non-functional properties such as efficiency or security. A way to assess implementation choices is to compile the abstract client request into concrete expressions which can be analyzed and compared.

In this section, we present in turn the abstract language, the concrete language, and their relationship.

2.1 Abstract language

The abstract language used to specify complex services is given in Figure 1. A service is either a tuple of independent expressions, a primitive function applied to an expression, or a data. *Service* and *Treatment* denote respectively the set of primitive services and the set of client actions. *Data* denotes the data provided by clients with their requests; it could be seen as the 0-ary treatment functions. The semantics of such expressions is straightforward and relative to an environment (e) associating a meaning to each primitive functions and data identifiers (see Figure 2).

$$E \ ::= \ (E, ..., E) \ \Big| \ fE \ \Big| \ d$$

where $f \in \mathit{Primitive} = \mathit{Service} \cup \mathit{Treatment}$ and $d \in \mathit{Data}$

Fig. 1. Abstract language

$$\begin{array}{lcl} \mathcal{E}_a : \mathit{Exp}_a \rightarrow \mathit{Env} \rightarrow \mathit{Value} & & \\ \mathcal{E}_a[\![(E_1, \ldots, E_n)]\!]\ e & = & (\mathcal{E}_a[\![E_1]\!]\ e, \ldots, \mathcal{E}_a[\![E_n]\!]\ e) \\ \mathcal{E}_a[\![f\ E]\!]\ e & = & (e\ f)\ (\mathcal{E}_a[\![E]\!]\ e) \\ \mathcal{E}_a[\![d]\!]\ e & = & e\ d \end{array}$$

Fig. 2. Semantics of abstract expressions

Examples. Let us take two examples of abstract expressions that will be considered throughout the paper. One of the simplest example is requesting a basic service s on a data d, and applying a treatment t on the result. This is specified as:

$$t(s\ d) \tag{1}$$

An instance of this kind of service is taken in [8] to illustrate the benefit of mobile agents compared to RPC interactions. In their weather forecast example, the services are requests to a picture database. The data d specifies a picture, the service s (actually the database) returns the picture matching the specification d, and t is an image processing treatment (e.g. a filter). If t decreases drastically the size of the image then it is clearly more bandwidth efficient to implement the service as a mobile agent. The treatment t is then executed on the server and only the much smaller image is sent back to the client.

As a more complex example, consider the following expression:

$$t(s_1(d_1, s_2\ d_2), t_3(s_3\ d_3)) \tag{2}$$

Such an expression may represent a service to book plane and train tickets depending on various criteria: for example, the sub-expression $t_3(s_3\ d_3)$ is a request (d_3) to a train company service (s_3) returning schedules that are then filtered (by t_3) to retain only daily trains going to the airport; independently, the service s_2 returns a list of possible destinations according to the criteria d_2 (e.g. hotel descriptions); the service s_1 takes a list of possible dates (d_1) and a list of destinations ($s_2\ d_2$) and returns a list of flights; then, the final treatment t matches selected flights and trains.

□

Abstract expressions specify the functionality of complex services as well as dependencies between base services and treatments. For the latter example, the base service s_2 must be accessed before s_1, whereas s_3 can be accessed independently. Even if abstract expressions are particularly simple, they are sufficient to model many realistic complex services and to pose interesting challenges.

2.2 Concrete language

The concrete language makes implementation choices explicit. Its syntax is described in Figure 3. A concrete expression is a collection of let-expressions, each one representing an interaction (i.e. a mobile agent, a remote evaluation or remote procedure call). We use a uniform representation for the interactions, in the sense that RPC and remote evaluation protocols are seen as particular agents in our concrete language. An interaction is a continuation expression A applied to a data argument D. Continuation-passing-style (CPS) [13] is the standard technique to encode a specific evaluation order in functional expressions. We use it here to express the sequencing of basic services and treatments. A function f now takes an additional argument, a continuation A, and applies it to the result

of its evaluation. The CPS version of a function f such that $fd = d'$ is a function $\overline{f}$ which takes an additional argument A (a continuation), and applies it to the result of its evaluation, that is $\overline{f}\ A\ d = A\ d'$. The continuation represents the sequence of primitive services or treatments that remain to be executed. The functions $go_{i,j}$ denote migrations from a place i to a place j[2]. Functionally, the *go* functions are just the identity. Once combined with continuation expressions, they suffice to express the agent's route precisely. The special continuation *end* terminates the evaluation of an interaction. The semantics of the concrete language is described in Figure 4.

$$
\begin{array}{rcl}
E & ::= & \textbf{let}\ (r_1, ..., r_n) = A\ D\ \textbf{in}\ E \mid D \\
A & ::= & f\ A \mid go_{id_1,id_2}\ A \mid end \\
D & ::= & (D_1, ..., D_n) \mid d \mid r
\end{array}
$$

where $f \in \overline{Primitive}$, $id_i \in Place$, and $d \in Data$

Fig. 3. Syntax of concrete language

$$
\begin{array}{lcl}
\mathcal{E}_c : Exp_c \rightarrow Env \rightarrow Value & & \\
\mathcal{E}_c[\![\textbf{let}\ (r_1, ..., r_n) = A\ D\ \textbf{in}\ E]\!]\ e & = & (\lambda(r_1, \ldots, r_n).\mathcal{E}_c[\![E]\!]\ e)(\mathcal{A}_c[\![A]\!]\ e\ (\mathcal{D}_c[\![D]\!]\ e)) \\
\mathcal{E}_c[\![D]\!]\ e & = & \mathcal{D}_c[\![D]\!]\ e \\
 & & \\
\mathcal{A}_c[\![f\ A]\!]\ e & = & e\ f\ (\mathcal{A}_c[\![A]\!]\ e) \\
\mathcal{A}_c[\![go_{id_1,id_2}\ A]\!]\ e & = & \mathcal{A}_c[\![A]\!]\ e \\
\mathcal{A}_c[\![end]\!]\ e & = & \lambda d.d \\
 & & \\
\mathcal{D}_c[\![(D_1, ..., D_n)]\!]\ e & = & (\mathcal{D}_c[\![D_1]\!]\ e, ..., \mathcal{D}_c[\![D_n]\!]\ e) \\
\mathcal{D}_c[\![d]\!]\ e & = & e\ d
\end{array}
$$

Fig. 4. Semantics of concrete expressions

Examples. The expression $t(s\ d)$ can be implemented in two different ways: either by RPC (i.e. processing the treatment at the client place) or by using a mobile agent (i.e. executing the treatment at the service place). Two concrete expressions correspond to these two options. Let us write $\overline{s}$ and $\overline{t}$ for the CPS

[2] Actually, even if some migrations could be deduced from the locations of base services, the *go* functions are needed to specify where treatments are to be executed.

versions of s and t, and write c and 1 to denote the client place and service place respectively. The first, RPC-based, implementation is represented by the following expression:

$$\begin{array}{ll}\textbf{let } r_1 = go_{c,1}(\overline{s}(go_{1,c}\ end))\ d & \textbf{in} \\ \textbf{let } r_2 = \overline{t}\ end\ r_1 & \textbf{in} \\ r_2 & \end{array}$$

The data d is transmitted to place 1 ($go_{c,1}$), the remote service s is called and its result is returned to the client ($go_{1,c}$). Then, the treatment is performed locally by the client. The second, agent-based, implementation is represented by the expression:

$$\begin{array}{l}\textbf{let } r_1 = go_{c,1}(\overline{s}(\overline{t}(go_{1,c}\ end)))\ d\ \textbf{in} \\ r_1\end{array}$$

The treatment is transmitted and executed at the service place (i.e. 1).

Let us describe now two possible implementations of the abstract complex service (2) presented earlier. We assume that the three base services s_1, s_2, and s_3 are located at different places, respectively 1, 2 and 3. The implementation based on two RPC protocols to interact with s_1 and s_2 and a mobile agent for s_3 with a remote treatment t_3 can be represented by the following concrete expression:

$$\begin{array}{ll}\textbf{let } r_1 = go_{c,3}(\overline{s_3}(\overline{t_3}(go_{3,c}\ end)))\ d_3 & \textbf{in} \\ \textbf{let } r_2 = go_{c,2}(\overline{s_2}(go_{2,c}\ end))\ d_2 & \textbf{in} \\ \textbf{let } r_3 = go_{c,1}(\overline{s_1}(go_{1,c}\ end))\ (d_1, r_2) & \textbf{in} \\ \textbf{let } r_4 = \overline{t}\ end\ (r_3, r_1) & \textbf{in} \\ r_4 & \end{array}$$

Figure 5-a gives the graphical representation of the interactions, where boxes represent the client and the service places, and arrows represent the data-flow between the components. Dashed-arrows indicate the place where the treatments are executed. The implementation based on a mobile agent protocol to interact with both s_1 and s_2 and on a RPC protocol to interact with s_3 can be represented by:

$$\begin{array}{ll}\textbf{let } r_1 = go_{c,3}(\overline{s_3}(go_{3,c}\ end))\ d_3 & \textbf{in} \\ \textbf{let } r_2 = \overline{t_3}\ end\ r_1 & \textbf{in} \\ \textbf{let } r_3 = go_{c,2}(\overline{s_2}(go_{2,1}(\overline{s_1}(go_{1,c}\ end))))\ (d_1, d_2) & \textbf{in} \\ \textbf{let } r_4 = \overline{t}\ end\ (r_3, r_2) & \textbf{in} \\ r_4 & \end{array}$$

The graphical representation of this complex-service is given in Figure 5-b.

□

Of course, different implementations of the same abstract expression are functionally equivalent. As we see in the next section, a concrete expression is functionally equivalent to the abstract expression it implements.

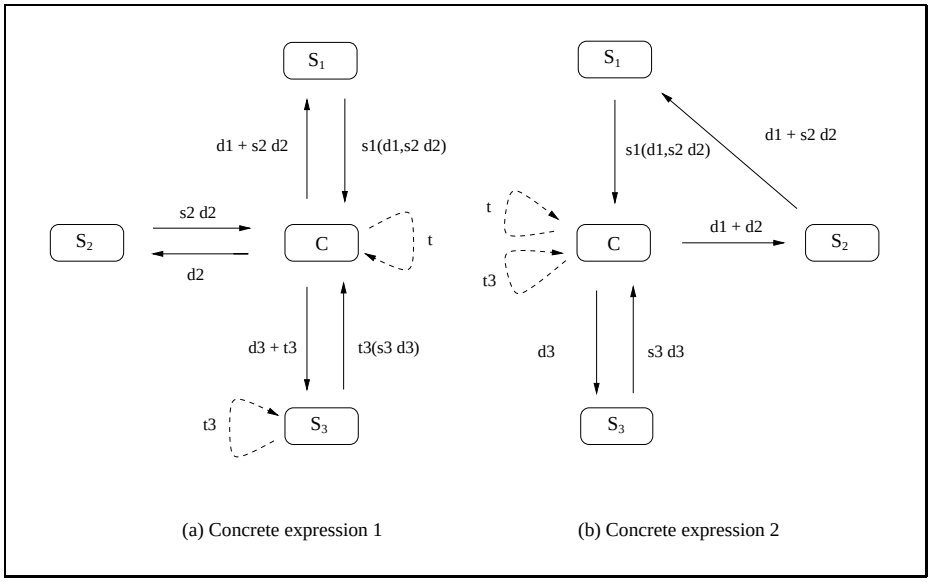

Fig. 5. Graphical representations of two implementations

2.3 From abstract to concrete expressions

We outline here how to compile an abstract expression into a concrete one. The compilation process depends on three choices: the number and constituents of agents, their route, and the place for the execution of each treatment. The third choice is represented by an environment (p) mapping any primitive function to its location (i.e. the server providing the service or the execution environment). The two first choices are represented as a list of lists of primitive functions. The outer list represents the agents whereas inner lists describe the route of each agent. For example, the choices corresponding to the concrete expression depicted in Figure 5-a are summarized in the following list of agents:

$$[[t_3; s_3]; [s_2]; [s_1]; [t]]$$

That is, there are four interactions; for example, the first one (i.e. sublist $[t_3; s_3]$) is a mobile agent that calls s_3 and then applies t_3. Of course, not all agent lists are valid. A valid list of lists must include all the different treatments and services of the abstract expression and must respect the dependencies of the abstract term.

Given a valid list of agents and an environment (p) mapping primitive functions to places, it is easy to translate an abstract expression into a concrete one. This compilation is described by the function *Abs2Conc* given in Figure 6. *Abs2Conc* takes a list of agents *ll* and produces a let-expression for each of them. It supposes that all functions have been renamed so that a function in the list denotes unambigously a call in the expression. For each sub-list l (i.e. an agent), *Abs2Conc* extracts (using the function *SubExp*) the sub-expressions $(E_1, \ldots, E_n)$

of the global expression E involving the primitives in l. For example:

$$SubExp\ [t_3; s_3]\ (t(s_1(d_1, s_2\ d_2), t_3(s_3\ d_3))) = t_3(s_3\ d_3)$$

and

$$SubExp\ [t_3; s_3; s_2]\ (t(s_1(d_1, s_2\ d_2), t_3(s_3\ d_3))) = (s_2\ d_2, t_3(s_3\ d_3))$$

$$Abs2Conc\ :\ Exp_a\ \rightarrow\ List(List(Primitive))\ \rightarrow\ (Exp_c, Env)$$

$$\begin{array}{l}
Abs2Conc\ E\ nil\ e\ =\ (E, e) \\
Abs2Conc\ E\ (l.ll)\ e\ = \\
\quad \underline{let}\ (E_1, ..., E_n) =\ SubExp(l,E)\ \underline{in} \\
\quad \underline{let}\ (X_1, e_1) =\ Agent[\![(E_1, \ldots, E_n)]\!]\ l\ client\ end\ e\ \underline{in} \\
\quad \underline{let}\ (X_2, e_2) =\ Abs2Conc\ E[r_i/E_i]\ ll\ e_1\ \underline{in} \\
\quad (\textbf{let}\ (r_1, \ldots, r_n) = X_1\ \textbf{in}\ X_2\ ,\ e_2)
\end{array}$$

$$Agent\ :\ Exp_a\ \rightarrow\ List(Primitive)\ \rightarrow\ Place\ \rightarrow\ Cont \rightarrow\ Env\ \rightarrow\ (Exp, Env)$$

$$\begin{array}{l}
Agent[\![E]\!]\ nil\ i\ k\ e\ =\ (go_{client,i}\ k\ E\ ,\ e) \\
Agent[\![(E_1, \ldots, fE_i, \ldots, E_n)]\!]\ (f.l)\ i\ k\ e\ = \\
\quad Agent[\![(E_1, \ldots, E_i, \ldots, E_n)]\!]\ l\ (p\ f)\ (\overline{f}(go_{(p\ f),i}k)) \\
\qquad\qquad\qquad\qquad e[\overline{f}\ \leftarrow\ \lambda c.\lambda(x_1, \ldots, x_n).\ c\ (x_1, \ldots, e\ f\ x_i, \ldots, x_n)]
\end{array}$$

Fig. 6. Translation algorithm

Each collection of sub-expressions is translated into an agent using the function *Agent*. The function *Agent* takes the sub-expressions to compile, the ordered list which indicates the sequentialization of services and treatments, the current place (initially the client place), the current continuation (initially the final continuation *end*) and the primitive function environment. Calls to services and treatments are sequentialized using CPS and $go_{i,j}$ are inserted according to the environment p. The environment e is updated to reflect the fact that new CPS versions of primitive functions are introduced. The correctness of the translation is expressed by the following property:

PROPERTY: $(E_c, e_c) = Abs2Conc\ E_a\ ll\ e_a\ \Rightarrow\ \mathcal{E}_a[\![E_a]\!]\ e_a = \mathcal{E}_c[\![E_c]\!]\ e_c$

In theory, the function *Abs2Conc* could be used to produce automatically all possible implementations of an abstract term, enabling their analysis and their comparison. It suffices to consider all possible valid agent lists, that is to say, all possible arrangements of primitive functions respecting the abstract dependencies, and for each of them, every admissible permutations. The algorithm to

verify that an agent list is valid is a simple check of the order of functions appearing in the list *w.r.t* the dependencies of the abstract term. In practice, this use of *Abs2Conc* might be considered only for small abstract expressions. In fact, there are up to 2^{n-1} agent arrangements (where n is the number of primitive functions), and up to $p!$ possible routes for an agent with p primitives.

We see in Section 4 more practical ways to use the compilation function *Abs2Conc*.

3 Analyzing performance and security properties

Concrete expressions make the analysis of performance and security properties easy. The number and paths of agents of our concrete expressions are explicit in the syntax. The only additional information needed is basic properties associated with primitive objects (data, functions, codes, places, ...). We consider that this information is given *via* environments mapping primitive objects to performance or security properties. The definitions of analyses are expressed similarly as the semantic functions of Figure 4; actually our analyses can be seen as abstract semantics/interpretations of concrete expressions. For performance properties, the measure we consider is the amount of traffic generated on the network. For security properties, we focus on confidentiality properties, without considering integrity for space reasons.

3.1 Performance

To estimate the cost (in terms of bandwidth) of the implementation of a complex-service, we consider two environments e_s and e_f. The environment e_s associates primitive functions to the size of their source code. For primitive services, this size is zero: services are tied to servers and do not travel on the network. The environment e_f associates each data to a size and each primitive function f to an abstract function s_f yielding the size of the result of f depending of the size of its argument (i.e. $s_f(size\ x) = size(f\ x)$). Sizes can be represented numerically or symbolically [3]. Of course, it is not always possible to know precisely in advance the sizes of results (e.g. number of hits for a query to an arbitrary database). In such cases, approximations such as the average or maximum sizes should be considered.

Using the two environments e_s and e_f, the amount of traffic involved by an implementation is evaluated by the function $Cost_E$ (see Figure 7). Compared to the semantics of Figure 4, the environment (e_f) now associates data and primitive functions to performance properties and the *raison d'etre* of $go_{i,j}$ functions becomes clear now. Note that the environment e_s is only read and used as a global constant. The cost of a let-expression is the cost of its sub-expressions. The cost of the body (E) is evaluated with its variables (r_i) associated with their

[3] Numerical values are easy to normalize and compare whereas symbolic values are more generic and represent more faithfully the reality. Both fit in our framework and we do not dwell on this issue any further in this paper.

size (d_i). The function $Cost_A$ evaluates the cost of an agent and takes as parameters the environment e_f, the size of the agent ($Source[\![A]\!]$, where α_{go} is the size of an instruction go), and the size of its data (d). The cost of an expression $go_{i,j}A$ is the cost of the continuation A plus the cost of transmitting the data and the agent source code ($i = j$ implies that there is no migration and therefore no transmission induced). More precisely, the transmission cost involved with a migration is:

$$\alpha_{i,j}(s \ + \ Sum \ d)$$

where $Sum\ d$ represents the total size of data (i.e. the summation of all the basic sizes in d), s denotes the size of the agent source code, and the coefficient $\alpha_{i,j}$ permits to take into account the bandwidth between places i and j (i.e. the quality of the connection).

$$
\begin{array}{ll}
Cost_E \ : \ Exp_c \ \rightarrow \ Env \ \rightarrow \ Size & \\
Cost_E[\![\textbf{let } (r_1, \ldots, r_n) = A \ D \textbf{ in } E]\!] \ e_f = & \underline{let} \ (d_1, \ldots, d_n) = \mathcal{A}_c[\![A]\!] \ e_f \ (\mathcal{D}_c[\![D]\!] \ e_f) \ \underline{in} \\
& \quad Cost_E[\![E]\!] \ (e_f[r_i \ \leftarrow \ d_i]) \\
& \quad + \ Cost_A[\![A]\!] \ e_f \ (Source[\![A]\!]) \ (\mathcal{D}_c[\![D]\!] \ e_f) \\
Cost_E[\![D]\!]e_f & = 0 \\
 & \\
Cost_A[\![f \ A]\!] \ e_f \ s \ d & = e_f \ f \ (Cost_A[\![A]\!] \ e_f \ s) \ d \\
Cost_A[\![go_{i,j} \ A]\!] \ e_f \ s \ d & = Cost_A[\![A]\!] \ e_f \ s \ d \ + \\
 & \quad \underline{if} \ i = j \ \underline{then} \ 0 \ \underline{else} \ \alpha_{i,j}(s \ + \ Sum \ d) \\
Cost_A[\![end]\!] \ e_f \ s \ d & = 0 \\
 & \\
Source[\![f \ A]\!] & = e_s \ f \ + \ Source[\![A]\!] \\
Source[\![go_{i,j} \ A]\!] & = \alpha_{go} + Source[\![A]\!] \\
Source[\![end]\!] & = 0
\end{array}
$$

Fig. 7. Cost evaluation

Example. Let us consider the simple expression $t(s\ d)$ again and suppose that the abstract environments are:

$$e_f = [d \ \leftarrow \ 10^4; s \ \leftarrow \ \lambda x.10^7; t \ \leftarrow \ \lambda x.xdiv10]$$
$$e_s = [s \ \leftarrow \ 0; t \ \leftarrow \ 10^5]$$

That is to say, the request is of $10Kb$, the database s returns $10Mb$ images, the treatment t divides image size by 10, the size of the source of t is $100Kb$, whereas, by convention, the size of the service s is null. The performance of an implementation choice ll for this expression is represented by:

$$\underline{let} \ \ (E, e'_f) = Abs2Conc \ [\![t(s \ d)]\!] \ ll \ e_f \ \ \underline{in} \ \ Cost_E \ [\![E \ ]\!] \ e'_f$$

Assuming that the cost of a go (α_{go}) is null and that $\alpha_{i,j} = 1$ for all places i and j, the cost of the associated RPC implementation (see Section 2.2) is $1.001\ 10^7$ (the data and the complete picture travel on the network, i.e. $10Kb + 10Mb$) and the cost of the mobile agent implementation (see Section 2.2) is $1.11\ 10^6$ (the data, the reduced size picture, and the code of the treatment travel on the network i.e. $10Kb + 1Mb + 100Kb$).

□

We have focused on the volume of transmission implied by an implementation. Other criteria, such as time efficiency could be considered as well. Some additional information such as the service response times or communication delays should be introduced in environments but the analysis would remain similar.

3.2 Security

Security is regarded by many designers as the most critical issue to address before promoting the use of mobile agents in large service-provisioning systems. There are now good and accepted techniques to protect hosts from malicious mobile agents. However, protecting mobile agent from malicious hosts is a much more difficult task [11]. The analysis of security properties can be of a great help to the system designer. In particular, this kind of information should permit to rule out implementations that make sensitive data or treatments travel through untrusted hosts.

In our framework, each object (data, code, place) is associated with a value taken from a lattice of security levels. As before, we consider two environments. The environment e_s associates server places to the security level they insure[4] and treatments to the security level their source requires to roam safely. The environment e_f associates each primitive function f to an abstract function s_f returning the security level of the result of f depending of the security level of its argument. The environment e_f also contains the security level associated to data. In the following, we consider confidentiality (non-divulgation) properties. Integrity (i.e. non-modification) properties are dual and can be analyzed likewise.

The analysis in Figure 8 checks if confidentiality requirements are met by a given concrete expression. The function $Conf_E$ is defined as a recursive scan of the expression and returns a boolean. The symbols $\sqsubseteq$, $\bot$, $\sqcup$ denote respectively the partial order relation, the smallest element, and the join of the lattice of security levels. The expression $\bigsqcup d$ denotes the least upper bound of all the confidentiality levels in d. The auxiliary function Lub evaluates the confidentiality level required by the source code of an agent; this is the least upper bound of the levels of all its treatments. Note that the environment e_s is only read and is used as a global constant. The function $Conf_A$ takes as parameters the expression to check, the environment e_f, the current place (initially *client*), the confidentiality level of the source code ($Lub[\![A]\!]$) and the confidentiality level of the data.

[4] When there is no knowledge on the security of a place, the lowest security level should be taken.

Checking an agent amounts to verifying at each step that the level required by the current data ($\bigsqcup d$) and by the source code (c) is less or equal than the one insured by the current place ($e_s\ p$). Furthermore, for each migration $go_{i,j}$, the confidentiality level required by the data and the code must be less or equal than the confidentiality level insured by the network connection from place i to j ($\alpha_{i,j}$).

$$
\begin{array}{ll}
Conf_E\ :\ Exp_c\ \rightarrow\ Env\ \rightarrow\ Bool & \\
Conf_E[\![\textbf{let}\ (r_1,\ldots,r_n) = A\ D\ \textbf{in}\ E]\!]\ e_f = & \underline{let}\ (d_1,\ldots,d_n) = \mathcal{E}_c[\![A]\!]\ e_f\ (\mathcal{D}_c[\![D]\!]\ e_f)\ \underline{in} \\
& \quad Conf_E[\![E]\!]\ (e_f[r_i\ \leftarrow\ d_i])\ \wedge \\
& \quad Conf_A[\![A]\!]\ e_f\ client\ (Lub\ [\![A]\!])\ (\mathcal{D}_c[\![D]\!]\ e_f) \\
Conf_E[\![D]\!]\ e_f & = true \\
\\
Conf_A[\![X]\!]\ e_f\ p\ c\ d & = (\bigsqcup d\ \sqcup\ c)\ \sqsubseteq\ e_s\ p\ \wedge \\
& \quad \underline{Case}\ \ X\ \ \underline{in} \\
& \quad f\ A\ \ \ \ : e_f\ f\ (Conf_A[\![A]\!]\ e_f\ p\ c)\ d \\
& \quad go_{i,j}\ A\ :\ (\bigsqcup d\ \sqcup\ c)\ \sqsubseteq\ \alpha_{i,j}\ \wedge \\
& \qquad\qquad Conf_A[\![A]\!]\ e_f\ j\ c\ d \\
& \quad end\ \ \ \ \ : true \\
\\
Lub\ [\![t\ A]\!] & = (e_s\ t) \sqcup Lub\ [\![A]\!] \\
Lub\ [\![s\ A]\!] & = Lub\ [\![A]\!] \\
Lub\ [\![go_{i,j}\ A]\!] & = Lub\ [\![A]\!] \\
Lub\ [\![end]\!] & = \bot
\end{array}
$$

Fig. 8. Confidentiality checking

In the context of mobile agents, integrity properties, that is to say the non-modification of data and treatments, is as important as confidentiality. Checking integrity requirements can be specified in much the same way as confidentiality. One difference is that once a treatment is executed on a host, it is not necessary to check its integrity until the end of the agent's route (whereas the confidentiality of treatment has to be checked for the complete journey).

4 Practical uses of the model

For our model to be of practical interest, it should be integrated within a design environment. At the implementation level, the environment should be based on a platform offering both RPC and mobile agent interactions (such as the Grasshopper platform [5] based on CORBA middleware). At an abstract level, an environment based on architecture description language is an ideal ground for our approach [10]. Software architecture lies at the heart of successful large design [4]. The main contribution of this promising research area is to abstract

away from the implementation details (macroscopic view of the system) and to mostly concentrate on the organization of the system to be specified, built and/or analyzed. This structural model is composed of components interconnected by connectors. The components are usually seen as processing units described by their behaviors and interfaces whereas the connectors are an abstract way of specifying the interaction protocols between those components.

In order to apply our approach, primitive services will be represented by components, interactions by connectors, and the location of base services (i.e. physical place) together with their associated attributes must be provided by the architecture description (i.e. functionality, efficiency and security levels). Several uses of such a framework can be considered:

- *Refinement of component.* Suppose that a component is specified as a complex service in our abstract language. The goal of the designer might be to refine this component into an architecture built from basic services. This refinement can be represented as a concrete expression. In other words, our approach can be used to choose the connectors. The naive generation of all the possible implementations is rapidly untractable if the complex service is not trivial. There are however admissible heuristics that permits to reduce this complexity. For example, if a treatment always increases the size of its inputs, then a simple mobile agent interaction will generate more bandwidth consumption than a RPC. Also asking for certain security properties permits to filter out many unsafe implementations.
- *Verification.* Given an architecture, the designer may want to verify that the protocols used are valid with respect to non-functional properties such as security. The architecture can be represented as a concrete expression and the analysis of Section 3.2 used to check the property.
- *Adaptation.* Given an architecture, the designer may want to change some parameters (such as service locations, bandwidth of connections, etc.) in order to meet non-functional requirements. The use of analyses in this context is much less problematic than in the context of refinement. Most parameters are already fixed and the best choice for the remaining ones can be found without combinatory explosion. For example, one may focus on optimal routes for fixed mobile agent in order to minimize bandwidth usage. Similarly, the security analysis may guide the designer in the use of securized connections, encrypted programs [14], or tamper-proof hardware, in order to ensure a security property.

5 Conclusion

Little work has been done towards evaluating non-functional properties of mobile agents. Some comparisons of RPC based *vs* mobile agent based applications on a given network have been done (e.g. see [6]). These approaches to performance evaluation are purely experimental. Furthermore, they do not consider hybrid interactions mixing mobile agents, remote evaluations, and remote procedure

calls. To the best of our knowledge, Carzaniga, Picco, and Vigna [2] were the first to provide some basic hints (relative to bandwidth consumption) to select the adequate interactions. Their simple performance model was extended for network load and execution time by Straßer and Schwehm [16] for a sequence of mixed interactions of RPCs and agent migrations. In our framework, these approaches boil down to analyzing a concrete expression where the size of all requests and replies is fixed.

Several formalisms have been proposed to model mobility. Variants of the π-calculus (e.g. the Ambiant-calculus [1]) are very powerful models of computation. They contain an explicit notion of space composed of components (code or physical device) that move through locations. They describe precisely the distribution, mobility, or inter-communication of a group of agents and can provide some security checks to control the channels of communication. Their final goal is to propose a language for modeling Internet applications. The calculus of Sekiguchi and Yonezawa [15] is an extension of the lambda-calculus closer to our proposal. Their goal was to describe and compare different mobile agents movement mechanisms (like Obliq, Telescript, etc.).

The goal of our work is to analyze non-functional properties in order to guide designers in building large distributed systems. As a first step, we found it more reasonable to consider a simple language. On the other hand, our approach is formal and takes into account any kind of hybrid interactions. There are many avenues for further research. One is to investigate the integration of other service interaction protocols in our model. In [12], Picco identifies three paradigms that can be used to design distributed applications exploiting code mobility (i.e. remote evaluation, mobile agent, and code on demand). In code on demand interactions the client sends a request and gets a function to be processed locally. Modeling these applet-like interactions implies a higher-order version of our language (i.e. services may return treatments). Other important language extensions are non deterministic choice and iteration. In addition, more complex inter-service dependencies could be specified by adding variables and the let construction in the abstract language. Another plan for future work is to consider other non-functional properties like fault tolerance and to complete the integration of our model within a software architecture design environment. We are currently looking at the ASTER environment [7] associated with a platform providing both mobile agent and RPC interactions. Although some extensions should be considered, we believe however that our approach paves the way towards the systematic selection of interactions in service architectures.

Acknowledgments

This work has been partially supported by grants from the C3DS project[5].

[5] Control and Coordination of Complex Distributed Services. ESPRIT Long Term Research Project Number 24962 - www.newcastle.research.ec.org/c3ds

References

1. L. Cardelli. Abstractions for mobile computations. In Jan Vitek and Christian D. Jensen, editors, *Secure Internet Programming, Security Issues for Mobile and Distributed Objects*, volume 1603 of *LNCS*, pages 51–94. Springer, 1999.
2. Antonio Carzaniga, Gian Pietro Picco, and Giovanni Vigna. Designing distributed applications with mobile code paradigms. In R. Taylor, editor, *Proceedings of the 19th International Conference on Software Engineering (ICSE'97)*, pages 22–32. ACM Press, may 1997.
3. David M. Chess, Colin G. Harrison, and Aaron Kershebaum. Mobile agents: Are they a good idea? Research report RC 19887, IBM Research Division, T. J. Watson Research Center, Yorktown Heights, NY 10598, february 1994.
4. David Garlan and Mary Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Series on Software Engineering and Knowledge Engineering, Vol 2*, Worlds Scientific Publishing Company, pages 1–39. 1993.
5. IKV++ GmbH. Grasshopper - `www.ikv.de/products/grasshopper/index.html`.
6. L. Ismail and D. Hagimont. A performance evaluation of the mobile agent paradigm. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 306–313, 1999.
7. Valérie Issarny and Titos Saridakis. Defining open software architectures for customized remote execution of web agents. *Autonomous Agents and Multi-Agent Systems Journal. Special Issue on Coordination Mechanisms and Patterns for Web Agents*, 2(3):237–249, september 1999.
8. Dag Johansen. Mobile agent applicability. In *2nd International Workshop on Mobile Agents, MA'98, Stuttgart, Germany*, volume 1477 of *Lecture Notes in Computer Science*, pages 80–98. Springer, september 1998.
9. Danny B. Lange and Mitsuru Oshima. Seven good reasons for mobile agents. *Communications of the ACM, Multiagent Systems on the Net and Agents in E-commerce*, 42(3):88–89, march 1999.
10. Nenad Medvidovic and Richard N. Taylor. A framework for classifying and comparing architecture description languages. *SIGSOFT Software Engineering Notes*, 22(6):60–76, november 1997.
11. Jonathan T. Moore. Mobile Code Security Techniques. Technical Report MS-CIS-98-28, University of Pennsylvania, Department of Computer and Information Science, may 1998.
12. Gian Pietro Picco. *Understanding, Evaluating, Formalizing, and Exploiting Code Mobility*. Ph.d. thesis, Politecnico di Torino, Italy, february 1998.
13. John C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6(3/4):233–248, november 1993.
14. T. Sander and C. Tschudin. Towards mobile cryptography. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, Research in Security and Privacy, Oakland, CA, may 1998. IEEE Computer Society Press.
15. T. Sekiguchi and A. Yonezawa. A calculus with code mobility. In Chapman and Hall, editors, *Proc. 2nd IFIP Workshop on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pages 21–36, London, 1997.
16. Markus Straßer and Markus Schwehm. A Performance Model for Mobile Agent Systems. In H. R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications PDPTA'97*, pages 1132–1140, Las Vegas, 1997.