Multiple Security Policies in Mob$_{adtl}$

G. Ferrari   C. Montangero   L. Semini   S. Semprini

Dipartimento di Informatica, Università di Pisa
{giangi,monta,semini,semprini}@di.unipi.it

1 Introduction

The growing use of mobile components (either code or autonomous agents) in networking applications has increased the need of having robust mechanisms to ensure that only legitimate mobile components can be granted access to specific resources or services. Research is underway to develop infrastructures, programming languages and programming methodologies for specification, implementation and certification (i.e., to state and prove properties) of secure networking applications.

We are developing Mob$_{adtl}$ [2], a temporal logic based model to specify and design networking applications. We contend that Mob$_{adtl}$ is particularly apt to support the specification and verification of applications where different security requirements coexist.

In the Mob$_{adtl}$ model, a system is based on an immutable net of elaboration nodes: the *neighborhoods*, bounded environments where several components live. The notion of neighborhood basically reflects the idea of an administrative domain where components run. A component's behaviours inside a neighborhood are constrained by the security requirements enforced by the neighborhood authority, namely the *guardian*. Guardians intercept messages and migrating components and decide which messages and components can enter or leave the neighborhood they control. The model itself does not embody any specific security policy. Effective security policies must be explicitly specified through suitable refinement steps.

The Mob$_{adtl}$ logic language has been designed to facilitate the specification and verification of applications by separating functionality from coordination (e.g.,, security and routing) policies. Mob$_{adtl}$ supports a development methodology where coordination policies are considered since early specification stages: both component functionalities and the security requirements can be specified independently at an abstract level. The complete system specification is then obtained by composing together several refinements corresponding to these different aspects of the system.

This has the main advantage that coordination issues are taken into account in the design stages and not considered later on top of existing infrastructures. So, Mob$_{adtl}$ fosters the natural evaluation of the impact on the overall architecture of specific policies. More in general, our goal is to provide a design methodology where developers have enough control on the overall architecture of the application and on the structure of its components to identify and deal early with issues like interoperability, flexibility and quality of service (e.g. performance, security).

In this paper we exploit the Mob$_{adtl}$ framework to specify security policies, which control the movements of agents among neighborhoods. In particular we describe a refinement step which gives guardians the ability to prevent agents from entering or exiting their neighborhoods according to their history.

2 A model for network–aware computing

We classify the models for network–aware computing along the following axis. A taxonomy of most models presented in the literature according to this perspective can be found in [2].

The nature of mobile units: They can be any combination of code, data and control. Models where only pieces of code can be moved are said to support a *weak* form of mobility, while models where the units of mobility are *processes* (code + control) or *agents* (code + control + data) support *strong* mobility.

Mobility extent: If not all the components can move, it is useful to distinguish between mobile and *stationary* components. Stationary components are considered as not transferable.

Location awareness: The units can be either *location aware* or not. Location awareness results in the ability of choosing between a set of possible next actions depending on the current location. Locations reflect the idea of administrative domains and computations at a certain location are under the control of a specific authority, or more in general have to satisfy specific security requirements.

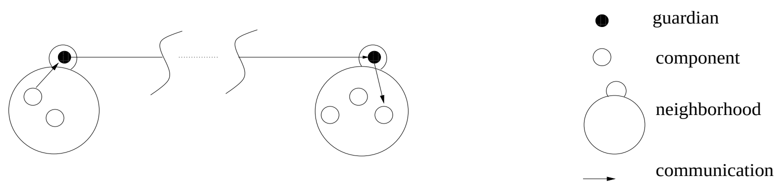
Location control: The mobile units can control their location (*proactive* or *subjective* mobility), or can be moved by other entities (*reactive* or *objective* mobility).

Communication models: transient shared memory, name passing over a named channel, asynchronous message passing via explicit addressing are the main choices here.

In our model, a system is based on an immutable net of elaboration nodes: the *neighborhoods*. The neighborhood is a bounded environment where several components (both stationary and mobile) live. Components have a unique name, e.g. determined by the initial neighborhood and a suffix to distinguish siblings.

The notion of neighborhood basically corresponds to that of location. Each neighborhood is associated with a stationary component, the *guardian*. The knowledge of their own guardian makes components (both stationary and mobile) location aware. A guardian acts as an interposition interface among components and neighborhoods: it specifies and implements communication and movement policies. In other words, guardians monitor the components and limit the resources they can use. More precisely, communications between components occur via the guardians. Guardians provide also routing facilities to forward messages and to handle migrating components.

The model supports strong *subjective* mobility. However component migration requests can be refused by guardians, for instance because of security reasons. Indeed, guardians intercept messages and components and can decide which messages and components can enter or leave the neighborhood they control. The following figure provides a pictorial representation of our model.



The notion of neighborhood permits to model several crucial issues of network–aware programming. For instance a neighborhood can be used to represent an administrative domain where components are under the control of a single authority (the guardian), a naming domain where components adopt a given naming policy, and also an address space. The current notion of neighborhood is not complete as important requirements are not covered. For instance the ability to define new neighborhoods and merge existing ones is missing.

Asynchronous communication permits to keep the model abstract from any specific communication protocol. The model itself does not embody any specific routing or security policy for the communications between guardians. Effective routing and security policies must be explicitly specified through suitable refinement steps. The development approach deals separately with functional, security, and mobility issues. For instance, we can fully specify the functionality of a component by giving only very abstract description of the security requirements. A complete system specification is therefore obtained by plugging together different refinements corresponding to different aspects of the system.

3 Axiomatization

The goal of the axiomatization is to capture the most general model which satisfies the properties of Sect. 2 and to provide the necessary hooks to specify applications as refinements of the axioms. The axioms fix an "interface" for agents and guardians. This permits to specify applications by instantiating the axioms and then adding sets of formulae which state the application specific functionalities and the communication and security policies in use. These sets can be defined independently of each other.

Axioms are defined using Oikos–*adtl*, a specification language for distributed systems based on asynchronous communications [6].

In the axioms, location awareness is encoded in the component state via predicate *guardedby(G)*, which identifies the current guardian, i.e. the current location. Stationary components are characterized by the invariance of this predicate. Dually, each guardian holds a representation of the neighborhood it controls via two mutually exclusive predicates, *guarding(A)* and *moving(A, T)*.

To move to another neighborhood, a mobile component (also called *agent*) A issues a mobility request to its guardian. The successful movement of an agent A to location T is represented by a suitable coordination of the events *guardedby(T)* in the agent and *guarding(A)* in the guardian T. Predicate *moving(A, T)* identifies those agents that asked to leave the neighborhood. Similarly, to send a message, a component sends a communication request to the guardian of the neighborhood it currently belongs to; delivery occurs via the receiver's guardian. Both mobility and communication requests can be vetoed (event *veto(_)*) by any guardian involved in the process.

The general model does not describe how messages or agents reach their target, or why and where a veto is raised: the axioms only make clear that any of the guardians involved in a communication or a movement, can veto it. Also, the guardians at both ends are necessarily involved. Which other guardians may be involved, is fixed by the routing and security policies at lower refinement levels. The formulae introduced there detail the guardians, and take care of what happens in between the sender and target guardians, who raises vetoes, and why.

3.1 Liveness properties of mobility

The complete axiomatization is presented elsewhere [2]. Here we present only the set of axioms modelling liveness properties of mobility. These are the axioms we refine to describe security policies.

When agent A decides to move to a target neighborhood T, it notifies a mobility request *to(A, T)* to its guardian O, which can either veto or accept the request. If the origin guardian O allows the agent to attempt to leave the neighborhood it controls, the mobility request may reach the destination neighborhood, or be vetoed along the route. If it reaches the destination, the target guardian can still reject the request instead of taking the agent under its own control.

ML1: A: moveTo(T) WHEN guardedby(O) CAUSES O: to(A, T)
ML2: O: to(A, T) WHEN guarding(A) CAUSES_C O: moving(A, T)
ML3: O: to(A, _) WHEN moving(A, _) CAUSES A:double_req
ML4: O: moving(A, T)) CAUSES T: from(A, O) ∨ (O: guarding(A) ∧ A: veto(mob(A, O, T)))
ML5: T: from(A,O) CAUSES (T:guarding(A) ∧ A:guardedby(T)) ∨ (O:guarding(A) ∧ A:veto(mob(A,O,T)))
ML6: C: guardedby(G′) WHEN guardedby(G) ∧ G≠G′ CAUSES_C C: ¬ guardedby(G)

ML1: event *moveTo(T)* in A represents its commitment to move to T. The reaction to this event is a mobility request $(to(A,T))$ to the current guardian O. ML2: guardians record mobility requests from the agents they are guarding (*O: moving(A, T)*). Operator CAUSES_C, instead of CAUSES, require the reaction to be immediate. This "freezes" the requesting agents, and prevents their guardians from considering other mobility requests from the same agent, before the current request is handled. Indeed, ML3 states that a further request from a frozen agent causes a veto of the kind *double_req*. ML4: a mobility request may be forwarded, or may result in a veto. If the request is vetoed, the agent is unfrozen, and returns under the control of the origin guardian. ML5: if a request to enter neighborhood T is accepted, the guardian starts controlling the requesting agent (*T: guarding(A)*), which enters the neighborhood (*A: guardedby(T)*). ML6: when an agent changes neighborhood, the data recording its old location are immediately removed.

4 Examples

This section, by two simple examples, shows how Mob_{adtl} supports the specification of security policies as refinements of the axioms. The aim of the first policy is to give a guardian the ability to prevent an agent from leaving its neighborhood if the requested destination is not trusted. The second policy permits to prevent an agent from entering a neighborhood if the agent "history" is not trusted by the guardian.

The guardians in a system can exhibit none, one, or both this policies. A guardian embodying both policies can be obtained by composing together the corresponding refinements.

To specify the first policy, we introduce the predicate *trusted(T)*, which holds in a guardian state when the target T of a movement request fulfills the guardian security requirements. The refinement step strengthens axiom ML4 as follows, and leaves the other axioms unchanged.

P1a: O: moving(A, T) WHEN trusted(T) CAUSES T: from(A, O)
P1b: O: moving(A, T) WHEN ¬ trusted(T) CAUSES O: guarding(A) ∧ A: veto(mob(A, O, T))

To describe the second policy, we first introduce the predicate *hist(H)* to trace, in the agents state, of the visited neighborhoods. The new axiom P2a is introduced to state that initially an agent history is empty. Axiom ML6 is refined into P2b to show how histories are updated. Then, predicates *to, from*, and *moving* change their arity to record the current history of an agent in the movement requests, as in *to(A, T, H)*. Axiom ML1 is refined to P2c. Finally, the new predicate *trusted_hist(H)* is introduced: it holds in a guardian state if H fulfills the guardian security requirements. Axiom ML5 is refined into P2d and P2e.

P2a: A: INIT hist([ ])
P2b: C: guardedby(G′) WHEN guardedby(G) ∧ G≠G′ ∧ hist(H) CAUSES_C C: ¬ guardedby(G) ∧ hist(G′|H)
P2c: A: moveTo(T) WHEN guardedby(O) ∧ hist(H) CAUSES O: to(A, T, H)
P2d: T: from(A, O, H) WHEN trusted_hist(H) CAUSES T: guarding(A) ∧ A: guardedby(T)
P2e: T: from(A, O, H) WHEN ¬ trusted_hist(H) CAUSES O: guarding(A) ∧ A: veto(mob(A, O, T))

Note that the policies lead to stronger properties of the system: some behaviours which are possible in the general model, are no longer allowed when policies are in place.

5 Discussion

Mob$_{adtl}$ provides good foundations for expressing and evaluating security policies:

- It provides each guardian (authority) with the ability to define its own security policy. Each policy is specified by refinement steps which place constraints on the behaviour of the guardian. Hence, behaviours which are possible in the general model no longer manifest when the security policies are in place. The policies can cope also with guardians which do not trust each other.
- The model allows each authority to modify its own security policy by adding new restrictions. However, arbitrary modifications could disrupt the coordination among the guardians. The logic permits to assess how a security policy modification affects the overall system behaviour.

The security policy at the system level is the result of coordinating the security policies at the guardians level. Notice that we work top–down during the refinement process to verify by construction that an application satisfies global security properties. Complementary approaches may work bottom–up.

The approach has several advantages. First, it supports interoperability: different computing environment may need different security requirements. Moreover, the security policies of the guardians can be enforced and implemented by different mechanisms. These may include:

- *Secure type programming languages.* Capability based type systems have been recently introduced to specify and enforce access control policies of mobile agents [1, 5, 4, 10, 8].
  The specification of the security policy can be transformed into a secure typed interface and implemented in a secure type language: the soundness of the type system ensures that program does not violate the security requirements implied by the type interface.
- *Cryptography.* The security policy has the role of establishing *trust* among guardians. Standard implementations exploit cryptography to enforce authority and identity of components. Cryptographic mechanisms are included in modern programming languages, e.g. the Java APIs [3].

– *Proof-Carrying Code.* Necula [7] has introduced an approach to ensure correctness of mobile code with respect to a fixed safety policy: code producers provide the code with a proof of correctness that code consumers check before allowing the code to execute.
  The security policy can be transformed into a safety policy and a Proof-Carrying Code mechanism can be used to determine whether the component supplied by a different guardian satisfies the safety policy.

We are currently exploring the use of the theorem prover Isabelle [9] to automatize reasoning on Mob_{adtl} properties and policies. As a first step, we have represented in terms of Isabelle theories the axioms of Mob_{adtl}. Moreover, we have specified a CLIENT/SERVER system and proved several properties of the system by composing local properties of the components.

Acknowledgments

This work was partly supported by the ESPRIT W.G. 24512 COORDINA, and by the italian national projects SALADIN and TOSCA.

References

1. L. Cardelli, G. Ghelli, and A. Gordon. Mobility Types for Mobile Ambients. In J. Wiederman, P. van Emde Boas, and M. Nielsen, editors, *26th Colloquium on Automata, Languages and Programming (ICALP) (Prague, Czech Republic)*, volume 1644 of *Lecture Notes in Computer Science*, pages 230–239. Springer-Verlag, July 1999.
2. G. Ferrari, C. Montangero, L. Semini, and S. Semprini. Mobile Agents Coordination in Mob_{adtl}. Available at www.di.unipi.it/~ semini, To appear in *Proc. 4th Int. Conf. on Coordination Models and Languages, Lecture Notes in Computer Science*, 2000.
3. L. Gong. *Inside Java 2 Platform Security.* Addison-WESLEY, 1999.
4. M. Hennessy and J. Riely. Resource Access Control in Systems of Mobile Agents. In U. Nestmann and B. Pierce, editors, *Proc. HLCL '98: High-Level Concurrent Languages*, volume 16.3 of *Electronic Notes on Theoretical Computer Science*, pages 3–17, Nice, France, Sep. 1998. Elsevier Science Publishers.
5. F. Levi and D. Sangiorgi. Controlling Interference in Ambients. In *Proc. 27^{th} Annual ACM Symp. on Principles of Programming Languages*, pages 352–264, Boston, Jan 00. ACM Press.
6. C. Montangero and L. Semini. Composing Specifications for Coordination. In P. Ciancarini and A. Wolf, editors, *Proc. 3nd Int. Conf. on Coordination Models and Languages*, volume 1594 of *Lecture Notes in Computer Science*, pages 118–133, Amsterdam, Apr. 1999. Springer-Verlag.
7. G. Necula. Proof–carrying code. In *Proc. of the ACM Symposium on Principles of Programming Languages.* ACM, 1997.
8. R. D. Nicola, G. Ferrari, R. Pugliese, and B. Venneri. Types for Access Control. *Theoretical Computer Science*, to appear, 2000. Available at `http://rap.dsi.unifi.it/papers.html`.
9. L. Paulson and T. Nipkow. Isabelle. www.cl.cam.ac.uk/Research/HVG/Isabelle/.
10. P. Sewell and J. Vitek. Secure Composition of Insecure Components. In *Proc. Computer Security Foundations Workshop 12, CSFW-12.* IEEE, 1999.