Bachelor Thesis:

„Security Aspects of Digital Actors“

By: Ayse Morali

Technische Universität Darmstadt
Fachbereich Rechts- und Wirtschaftswissenschaften
Institut für Betriebswirtschaftslehre
Fachgebiet Wirtschaftsinformatik I
(Entwicklung von Anwendungssystemen)
Prof. Dr. Erich Ortner

Advisor: Prof. Carlos Varela

Rensselaer Polytechnic Institute
Computer Science Department

29.06.2005, Troy/USA

ABSTRACT

Online computations are getting more and more embedded into the daily life of people, in particular, they have practical applications in electronic commerce. In the cyber world, mobile codes interact with each other and their environment on behalf of sellers and buyers of the real world. Mobile code technologies provide potential benefits to applications, but as the responsibilities and the complexity of mobile codes increase, the variety of security threats that imposed the applications increases as well. In order to protect these inherently distributed systems from attacks, their components have to consider security aspects, without giving up flexibility. Transparent information flow between the mobile codes and their environment can help increase efficiency for its participants and reduce user's cognitive load, yet add points of vulnerability to the system.

In this thesis we describe a middleware framework for online auctions, in order to illustrate security problems in electronic commerce applications and suggest potential solutions. This middleware framework is a model of a futuristic *electronic marketplace*, consisting of various management components and electronic stores. In the electronic stores, the mobile codes, embedded in software agents, representing sellers and buyers, come together, in order to fulfill their duties.

Furthermore, we describe a set of security requirements of the electronic marketplace. We then survey different models of mobile code in distributed computations, including actors, secure mobile actors, transactors, casts and directors, and mobile ambients. We then classify and compare these models according to their applicability to the security of the electronic marketplace scenario. We demonstrate how additional security instruments, such as proof-carrying code and cryptographic techniques, can be incorporated to these models in order to fulfill the security requirements of the marketplace electronic commerce applications.

In summary, we investigated the conflicting design goals of openness and security in electronic commerce applications. This research is a first step in achieving the goal of building open yet secure electronic commerce systems.

ACKNOWLEDGEMENT

I would especially like to thank to Prof. Carlos Varela for motivating and advising me throughout this thesis. I would like to thank Worldwide Computing group at RPI for the positive work atmosphere. Furthermore I would gratefully acknowledge the technical and informative supports and hospitality of Fikret Sivrikaya. Thanks also to Prof. Erich Ortner for giving me the opportunity to write my this thesis abroad. Last but not least I would like to thank my brother, Bülent Özgür Morali and my parents, Nurten and Vedat Morali, for their financial and moral supports.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

Actor Language	AL
Access Control List	ACL
Actor Topology Sensitive Random-Stealing	ARS
Central Processing Unit	CPU
Coordination Model Manager	CMM
Denial of Service	DoS
Global Positioning System	GPS
Information Technology	IT
Local Area Network	LAN
Message Authentication Code	MAC
Network Topology Sensitive Random-Stealing	NRS
Proof-Carrying Code	PCC
Partial Result Authentication Code	PRAC
Load-Sensitive Random-Stealing	RS
Seller Manager	SM
Secure Mobile Actor Language	SMAL
Transaction Manager	TM

1. Introduction

1.1 Motivation

As the hardware and software devices, that today's mobile users utilize, diversify strongly; the threats against the information security increase. This circumstance requires distributed systems that are adjusted with intelligent and secure components with an open dynamically reconfigurable design. These components have to fulfill furthermore security and dependability requirements like authentication, integrity, confidentiality, accountability, and availability.

This thesis describes a middleware platform for actor based e-commerce applications, like online auctions, which fulfills the features mentioned above. The seller and buyer *actors* [1] come together on this platform to interact with each other and their environment in order to fulfill their tasks. It is a flexible, dynamic middleware, which adapts itself to constantly changing distributed heterogeneous environments. Message traffic between the actors uses stateful and consistent protocols.

Furthermore, this middleware platform accomplishes a secure environment to model the behavior of the actors that migrate from one host (location) to the other. In order to realize security, the system protects a subset of actors, from malicious locations but also from malicious codes. Some of the actor models that can be utilized in such distributed, unreliable environments are the Secure Actor Model, Transactors Model[2], Hierarchical Coordination Model, and Mobile Ambients.

Considering the above mentioned aspects, we are going to start with defining the security requirements, actors and actor models with their computational semantics. Later we will return to the electronic marketplace model that we have developed for this thesis and discuss the applications of the actor models to security of the e-marketplace.

Information security is defined in the theory as achieved when security require-

[1] Actors are computational agents, which are distributed over time and space

[2] Transactors is a fault-tolerant programming model for composing loosely-coupled distributed components running in an unreliable environment such as the internet into systems that reliably maintain globally consistent distributed state [16] [17].

ments are fulfilled. These requirements are grouped as authentication, information confidentiality, data integrity, accountability, system availability, data anonymity, and privacy.

Digital actors [1] are mobile computational agents that encapsulating a state, a set of operations that manipulate the state and a thread of control. Each actor has furthermore a conceptual location, a unique mail address, associated with it. The Actor Language (AL) introduced by Agha, Mason, Smith and Talcott [2] as an extension to the call-by-value λ-calculus [27], provides a mechanism for specifying the creation and manipulation of actors. AL has been extended by Toll and Varela [41] with mobility features into the Mobile Actor Language (MAL) and with security features into the Secure Mobile Actor Language (SMAL).

Transactors proposed by Field and Varela [17] is a fault-tolerant programming model for composing loosely-coupled distributed components. The transactor model is formalized via the τ-calculus, which is an extended λ-calculus, that incorporates constructs to create globally-consistent distributed states. Furthermore, an operational semantics for the τ-calculus is used to formally prove safety and liveness properties.

The hierarchical model proposed by Varela and Agha [43] is an actor based model for coordination of concurrent activities. In this model, actors are grouped into *casts*. Unique actors called *directors* are responsible of the coordination in a cast and *messengers* are responsible of sending messages between casts. Hierarchical model furthermore, extends the computational semantics of AL in order to capture casts and directors.

Mobile Ambients [8] is a mobility paradigm, where computational ambients are hierarchical structures, which actors are confined to, and move under the control of actors. They scales up mobile computation into widely distributed, intermittently connected and well administered computational environments. Furthermore, Cardelli and Gordon introduce a calculus describing the movement of processes and devices, including movement through administrative domains.

Information systems, where mobile codes are migrating and communicating, consist of two main components: actors and locations. Both actors and locations have to be protected from malicious actors and locations. In the theory of information security there are numerous instruments that can be used in order to prevent or detect vulnerabilities of information systems. The most common ones are cryptographic techniques, Proof-carrying code (PCC) and sand boxing. These instruments can be combined with each other or with security specifications of actor models in order to achieve a secure actor environment.

The electronic marketplace model that we have developed consists of a middleware framework, where actors representing sellers and buyers come together in online auctions that take place in stores. We create various scenarios over this model in order to show how the actor models can overcome these vulnerabilities, since there is no single model that fulfills all of the security requirements.

1.2 Structure of Thesis

This thesis describes in Chapter 2, the basic security requirements of information systems. That is followed with the description of the actor model and various models that extend the actor model with features that have the potential to make the actors model more secure. Chapter 2 is concluded with instruments that have been suggested by other authors in order to make an actor based information system more secure.

In Chapter 3, an electronic marketplace scenario is introduced, its architecture is described, and its vulnerabilities are defined.

In Chapter 4 the actor models defined in Chapter 2 are classified according to how they compare with respect to security requirements of an information system compared. Later on in this chapter, we will show by example how far they can fulfill security for the marketplace scenario.

The thesis will be concluded with future directions and possible security and feature extensions of the electronic marketplace.

2. Background and Related Work

In this chapter, we are going to describe first the security requirements of information systems in general. Then, we will discuss different actor models with their computational semantics. And afterwards various security instruments that can be applied to actor environments are going to be described.

2.1 Security Requirements

Security refers to techniques for ensuring the ability of the software to prevent unauthorized access by mistake or deliberately to code or data.

The use of mobile codes generates a lot of security problems and achieving a 100% security in a distributed network system is not possible. The security problems or sort of attacks a specific computer system is open to have to be determined and a focused set of countermeasures against these identified threats has to be developed. Optimal security architecture could be constructed this way. Furthermore, the mobile agent systems must be seen as embedded in an application scenario, and the best way to evaluate an actor based approach is to determine whether the specific security problems for this application scenario are solved.

In order to avoid misunderstanding, in this section the general security requirements of distributed software systems will be roughly defined.

2.1.1 Authentication

Authentication between the mobile actors and the locations is a major concern. An actor must authenticate on each visited location so that the location can decide whether if the actor is trusted. On the other hand the location must authenticate with the actor so that the actor can be sure it is on the correct location. That is what we call *two-way authentication.*

If the two-way authentication fails then, a malicious location could, for example, masquerade as another location to deceive the actor about its real host. As a consequence, the actor could disclose sensitive information, because it assumes it is a trusted location. This can be realized for actors via unique actor addresses, similar to the IP-Addresses in the network systems. But the question is if they really are dependable, since they can be changed[3]. Some other cryptographic methods referring to authentication will be described later in this thesis.

2.1.2 Confidentiality

Confidentiality demands that information be protected against unauthorized access. Confidential information that an actor carries, like the trading algorithms, or the private data, like the account information of the actor's owner, must not be accessible for the other actors or locations. Furthermore confidentiality involves accomplishment of information security during an information flow.

Symmetrical and asymmetrical encryption techniques can be used in order to protect the information or code that an actor carries.

2.1.3 Integrity

Integrity is necessary to make sure that a particular piece of data has not been modified during some period. Integrity has to be fulfilled for example when an actor has a predefined itinerary, otherwise the actor could be sent to arbitrary hosts. Integrity of the mobile code is also a very common field of security, since it is hard to detect the locations and protect mobile codes during a remote computations.

Digital signatures, integrity/checkers or message authentication codes (MAC) are some of the security instruments that can be used to make sure that some specific data has not been changed during any time of an actor's life.

2.1.4 Accountability

Accountability means that each subject is responsible for any action it has taken and can not deny responsibility later on.

Using cryptographic techniques, such as digital signatures, is a way to provide accountability. Because actors are mobile, carrying secret information, like private

[3]In the Information-Technology-Security (IT-Security) there is a very common attack called IP-Spoofing, which is a technique in which a malicious host claims to own a specific IP address to intercept requests to that IP address and take the role of the other computer. See [13] for more details.

keys without compromising it is not trivial. How accountability could be managed in a system, where actors are mobile will be presented in the coming sections.

2.1.5 Availability

Availability aims to ensure that access to a service cannot be restrained in an unauthorized way and guarantees a reliable and prompt access to data and resources for authorized principals. Availability is broken usually because programming failure, like using insecure classes in object-oriented programming.

In case of mobile agents, a malicious host can, for example, refuse to execute an actor, which is Denial of Service (DoS) attack. Or malicious actor could consume all host resources, inflicting a DoS attack on other actors. Another attack against the availability of system components is called Distributed-DoS, where many malicious components attack one component at the same time, like bombing the destination with messages sent at the same time. Redundancy of computer systems, hardware and software could be realized through techniques like load balancing or fault tolerance. PCC can also be used to minimize such deficits. These techniques will be discussed later in detail.

2.1.6 Anonymity and Privacy

Anonymity is the state of not being identifiable, *privacy* on the other hand, is the state of being free from unwanted intrusion. Hiding one's identity may be by choice, for legitimate reasons such as privacy. In order to realize privacy, the private information saved in a system must be protected from unauthorized third parties, which requires trusting the system. In some cases anonymity entails privacy protection. By converting the private data into anonymous data, identity dependency of the data can be disestablished. Privacy refers to the concept of informational self-determination, which means that the owner of the private data is the only one who can decide about the usage of the data.

Anonymity is a major concern, especially in e-commerce. It must be for example possible for an actor to keep his owner as a secret, when purchasing some article. There are some projects like the JAP project at the University of Dresden[4], in which a software guarantees anonymity on the level of IP addresses when surfing the web. Referring to online actors, Callsen and Agha have proposed the visibility feature in ActorSpaces [7], as a way to ensure anonymity of message recipients.

2.2 Actors

Actors as Agha [1] describes in his approach, are computational agents, which are distributed over time and space. In comparison to passive objects, as can be seen in the Figure 2.1 [44], actors encapsulate a state and a thread of control. The communication between the agents are realized with point-to-point messages that are buffered in an input queue. The messages that are received by the interface of an actor are interpreted with public methods, which operate on the state of the actor. Furthermore, the actors are associated with a conceptual location and a unique name, like a mail address, which can be used as a reference by other actors [42].

The actors communicate with other actors or their surrounding in order to fulfill their duties. While processing a message there are three basic actions that an actor may perform:

1. changing the local state and becoming ready to process the next message in its mail queue,
2. creating a finite set of actors with some initialized behaviors, and
3. sending messages to peer actors.

The actor model does not necessarily use a specific programming language. But it is possible to extend any sequential language with appropriate constructs for creating new actors, sending asynchronous messages and accepting incoming messages. The local computation of an actor in response to the message received can be modeled by the usual sequential constructs.

[4]See http://anon.inf.tu-dresden.de/index_en.html for details.

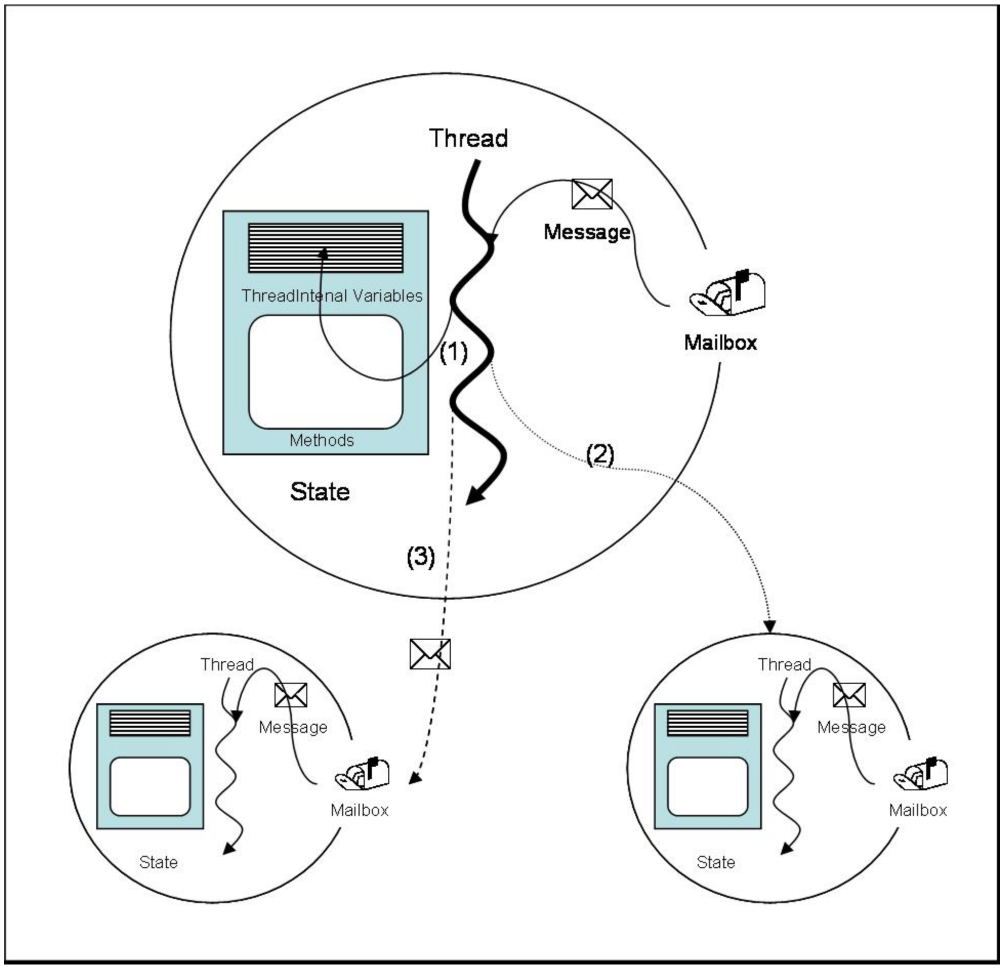


Figure 2.1: Architecture of an actor, and the actions it performs

2.2.1 Actor Language

Individual actors are the smallest coordination unit in the middleware model. The basic actor language (AL) proposed by Agha, Mason, Smith and Talcott [2] provides a mechanism for specifying the creation and manipulation of actors. Each actor is a unit of computation encapsulating data and behavior. The behavior defines how the actor reacts on receipt of a message [41]. In AL actor's behavior is described by a lambda abstraction, which embodies the code to be executed when a message is received. Respectively the actor primitives are:

- `new(b)` for creating an actor with behavior b. It also returns the new actor's name.

- `send`($\mathtt{v}_0, v_1$) where v_1 is the content and v_0 the destination actor, creates a new message and puts it in the message delivery system.

- `ready(b')` for ending the current execution and making the actor ready to receive a new message using behavior b'.

`new(b)` creates a new actor and returns its address. It is also possible to create multiple actors with this schema. The parent actor knows the name of the newly created actors, therefore it can send messages to each of them with each others names.

2.2.2 Actor Configuration

An actor configuration is a global snapshot of a group of an actors. It consists of actor mapping, messages in transit, a set of receptionists, and a set of external actors.

Assuming that two sets AT (atoms) and X (variables) are given and V, E and M are defined as follows:

Definition 1: **V E M**

The set of values **V**, the set expressions **E** and the set of messages **M** are defined inductively as follows:

$\mathbf{V} = \mathbf{At} \cup \mathbf{X} \cup \lambda\mathbf{X}.\mathbf{E} \cup pr(\mathbf{V},\mathbf{V})$

$\mathbf{E} = \mathbf{V} \cup app(E,E) \cup \mathbf{F}_n(\mathbf{E}^n)$ where $\mathbf{F}_n(\mathbf{E}^n)$ is all arity-n primitives.

$\mathbf{M} = \langle \mathbf{V} \Leftarrow \mathbf{V} \rangle$

Actor names are denoted with variables. At any given point an actor can be either ready to receive a message, `ready(e)`, or currently busy with execution of some expression `e`. And $< v_0 \Leftarrow v_1 >$ denotes a message v_0 sent to an actor v_1.

An actor configuration with actor map, α, multi-set of messages, μ, receptionists, ρ, and external actors, χ, is written $\langle \alpha \mid \mu \rangle^{\rho}_{\chi}$, where[5] $\rho, \chi \in P_\omega[X]$, $\alpha \in X \rightarrow^f E$, $\mu \in M_\omega[M]$, and let $A = Dom(\alpha)$, then:

1. $\rho \subseteq A$, and $A \cap \chi = \emptyset$.

[5]Let $P_\omega[X]$ be the set of finite subsets of X, $M_\omega[M]$ be the set of multi-sets with elements in M, $X_0 \rightarrow^f X_1$ be the set of finite maps from X_0 to X_1, Dom(f) be the domain of f, and FV(e) be the set of free variables in e.

2. if $a \in A$, then $FV(\alpha(a)) \subseteq A \cup \chi$, and if $< v_0 \Leftarrow v_1 > \in \mu$, then $FV(v_i) \subseteq A \cup \chi$ for $i < 2$.

2.2.3 Operational Semantics

In this subsection, we are going to describe the operational semantics that have been developed in [2] for the actor model. The notation $R[e]$ represents a redex e in a reduction context R. For the formal definitions of reduction context, the reader is referred to [2].

- <**fun** : a >
  $e \rightarrow^{\lambda}_{Dom(\alpha)\cup\{a\}} e' \Rightarrow \langle \alpha\{[e]_a\} \mid \mu \rangle^{\rho}_{\chi} \quad \mapsto \quad \langle \alpha\{[e']_a\} \mid \mu \rangle^{\rho}_{\chi}$
- <**new**: a, a' >
  $\langle \alpha\{[R[new(e)]]_a\} \mid \mu \rangle^{\rho}_{\chi} \ \mapsto \ \langle \alpha\{[R[a']]_a, [e]_{a'}\} \mid \mu \rangle^{\rho}_{\chi} \qquad a'$ fresh
- <**send**: a, v_0, v_1 >
  $\langle \alpha\{[R[send(v_0,\ v_1)]]_a \mid \mu \rangle^{\rho}_{\chi} \ \mapsto \ \langle \alpha\{[R[nil]]_a\} \mid \mu \uplus < v_0 \Leftarrow v_1 > \rangle^{\rho}_{\chi}$
- <**receive**: v_0, v_1 >
  $\langle\ \alpha\{[ready(v)]_a\} \mid\ < a \Leftarrow v_0 > \uplus \mu \rangle^{\rho}_{\chi} \ \mapsto$
  $\langle\ \alpha\{[app(v,\ v_0)]_a\} \mid \mu \rangle^{\rho}_{\chi}$
- <**out**: v_0, v_1 >
  $\langle\ \alpha \mid \mu \uplus < a \Leftarrow v_0 > \rangle^{\rho}_{\chi} \ \mapsto \ \langle\ \alpha \mid \mu \rangle^{\rho'}_{\chi}$
  if $a \in \chi$ and $\rho' = \rho \cup (FV(v_0) \cap Dom(\alpha))$
- <**in**: v_0, v_1 >
  $\langle\ \alpha \mid \mu \rangle^{\rho}_{\chi} \ \mapsto \ \langle\ \alpha \mid \mu \uplus < a \Leftarrow v_0 > \rangle^{\rho}_{\chi \cup (FV(v_0) - Dom(\alpha))}$
  if $a \in \rho$, then $FV(v_0) \cap Dom(\alpha) \subseteq \rho$

In order to simplify actor programming and realize actor coordination more efficiently, some abstractions have been developed. In the next section, significant abstractions for the actor model will be described.

2.2.4 Communication Abstractions

There are some abstractions, which can be applied in order to simplify programming in scalable networks. These abstractions are as follows:

1. **Call/Return Communication:** Method execution may need in some cases information from other actors to complete. Such communication patterns, where a sender invokes a remote operation and uses the return to continue its computation, are called *call/return communication* [1] [7]. In the actor model, call/return communication makes explicit manipulation of the continuation and no synchronization is necessary because communication in actors is point-point and non-blocking. Although these characteristics are insufficient as programming abstractions, they provide an efficient execution model [24]. The computation dependent on the reply is separated into a *join continuation*, and these transformations preserve the maximal concurrency in the program [1].

2. **Pattern-Directed Communication** abstracts a group of actors, if it is need to communicate not only with one actor, but with a group of actors. The *ActorSpace* model [7] describes an abstract specification of a group of actors through associating an actor with specific attributes. An actor space can be associated with meta-level operations, which allows an actor space management that transparently schedules the requests, in order to ensure load balancing. Chien and Dally **??** proposes a similar abstraction called *Concurrent Aggregates.*

3. **Synchronization constraints** is a language construct proposed by Frölund [19], which is a part of an actor's interface, and allows programmers to write executable specifications of per-object message-ordering constraint. Synchronization constraints delay the dispatch of messages so that the dispatch order is consistent with the message-ordering constraint. A synchronizer [19], in comparison to synchronization constraints, is a distinct entity that specifies the message-ordering constraints for a group of objects. It transparently delays the dispatch of messages by a group of actors according to user-specified crite-

ria, including atomic group message display. A synchronizer can furthermore transparently observe message dispatch by objects in the group, and these observations can influence the way in which message dispatch is delayed in the future. Although both synchronizers and synchronization constraints express message-ordering constraints, there are significant differences between the two concepts. Synchronizers specify constraints in terms of interface of actors, whereas synchronization constraints specify constraints in terms of the implementation of actors.

Actors can be modeled in an open system through the concept of actor configurations, which is an instantaneous snapshot of an actor system. Each configuration has a set of receptionist, which may receive messages from actors outside the configuration and a set of external actors which may receive messages from actors within. The details of actor configuration will be discussed in the Section 2.5 with more details.

2.3 Secure Mobile Actors

In order to achieve mobility and security towards robust distributed computing systems, Toll and Varela [41] have extended the AL with primitives for mobility and security and then developed an operational semantics for these extensions. In this section, these extensions will be discussed.

2.3.1 Model Properties

Actors are inherently independent, concurrent and autonomous which enables efficiency in parallel execution and facilitates mobility. Each actor is a unit of computation encapsulating data and behavior. Communication between actors is purely asynchronous and guaranteed. That means, when a message is sent, the model guarantees that the destination actor will receive the message; however, it does not guarantee the order of message arrival or the order of processing.

In mobile computation, actors do not only interact with other actors, but also with their environments. Different from actor model, a mobile actor model considers this property. Because, when the problems associated with worldwide computing are considered, it becomes important to represent the actor's environment. Otherwise it is not possible to model the behavior of an actor.

The universal actors extend actors with locations, mobility, and the concept of universal names and universal locations. Locators represent references that enable communication with universal actors at a specific location. An actor's location abstracts over its position relative to other actors. Each location represents, like an actor's configuration, an actor's run-time environment and serves like an encapsulating unit for local resources. Ubiquitous resources have a generic representation. Resource-to-actor translations are stored in resource maps, which are maps between global names and the names of local actors who fill a resource's role. Actors keep references, which get updated upon migration to resources at new locations. They can also keep references to non-ubiquitous resources by using resource attachment and detachment operations.

Secure actors restrict communication and migration behaviors to actors within specific *Access Control Lists* (ACL). Using this method, no unprivileged actor can gain access to a resource. Every actor or location has an ACL, which contains a list of actors allowed to migrate into the location or send/receive messages. ACLs can only be changed by the location, the actor in consideration, or by a resident actor in case of passive locations.

2.3.2 Secure Mobile Actor Language

Secure Mobile Actor Language (SMAL) extends AL with semantics that fulfill some security and mobility features, as mentioned above. So it defines the following primitives besides the actor primitives *new(b)*, *send*(v_0, v_1) and *ready(b')*:
For mobility:

- `newloc(Y)` indicates the appearance or creation of a new location, with an initial resource map denoted by Y.
- `migrate(l')` moves an actor from its current location to one denoted by l'.
- `attach(v)` saves a resource denoted by v into the actor's resource map.
- `detach(v)` removes a resource denoted by v from the actor's resource map.

- `register(`$\mathtt{v}_0$`,`$\mathtt{v}_1$`,l)` adds the mapping of a resource name to an actor in a location.

- `deregister(`$\mathtt{v}_0$`,`$\mathtt{v}_1$`,l)` removes the mapping of a resource name to an actor in a location.

  And for security:

- `allow(v)` changes the actor's ACL to include actor v.

- `allowloc(v)` changes the actor location's ACL to include actor v.

- `disallow(v)` changes the actor's ACL to exclude actor v.

- `disallowloc(v)` changes the actor location's ACL to exclude actor v.

If there are no restrictions on messaging or migration of an actor, then as an argument *allow* or *allowloc* primitives receive a null ACL, which is represented by $\perp$.

SMAL is a language that has been expanded in layers. In the the first layer there is the call-by-value λ-calculus, which is extended with primitives for actor communication to AL. MAL extends AL with mobility primitives[6] and SMAL adds security primitives to MAL.

Assuming that two sets AT (atoms) and X (variables) are given and V, E and M are defined as follows:

Definition 1: **V E M**

The set of values **V**, the set expressions **E** and the set of messages **M** are defined inductively as follows:

$\mathbf{V} = \mathbf{At} \cup \mathbf{X} \cup \lambda\mathbf{X}.\mathbf{E} \cup pr(\mathbf{V},\mathbf{V})$

$\mathbf{E} = \mathbf{V} \cup app(\mathbf{E},\mathbf{E}) \cup \mathbf{F}_n(\mathbf{E}^n)$ where $\mathbf{F}_n(\mathbf{E}^n)$ is all arity-n primitives.

$\mathbf{M} = \langle \mathbf{V} \Leftarrow \mathbf{V} \rangle_X$

The set **X** of variables represent both actors and locations, where the set **L**, with $\mathbf{L} \subseteq \mathbf{X}$, is the set of locations, and the set **R**, with $\mathbf{R} \subseteq \mathbf{X}$, is the set of resource identities. The structure of M allows selection of valid senders via ACLs,

[6]Interested readers are refered to [41] for the operational semantics of MAL.

where list of ACLs is defined as: $ACL \in \mathbf{P}_\omega[Dom(\alpha) \cup \{\perp\}]$. α is defined as: $\alpha \in X \rightarrow^f (E \times L \times (R \rightarrow^f X) \times ACL)$[7].

The definition of actor configuration in AL is also extended with a mapping, π, from locations to resource maps ($\pi \in L \rightarrow^f (R \rightarrow^f X)$). According to these changes, a universal actor configuration is written as: $\langle \alpha \mid \mu \mid \pi \rangle^\rho_\chi$.

MAL and SMAL extends the actor configuration in AL with another rule to denote that actor names and locations are disjoint. Where $\downarrow_i$ indicates that the projection of a cross product onto its i^{th} coordinate, the actor configurations in MAL and SMAL are written as:

1. $\rho \subseteq A$, and $A \cap \chi = \emptyset$.
2. if $a \in A$, then $FV(\alpha(a)) \subseteq A \cup \chi$, and if $< v_0 \Leftarrow v_1 > \in \mu$ then $FV(v_i) \subseteq A \cup \chi$ for $i < 2$,
3. $Range(\alpha) \downarrow_i \cap A = \emptyset$.

2.3.3 Operational Semantics

In this section, we are going to present the operational semantics that Toll and Varela [41] have developed for SMAL.

- <**fun** : a >
  $e \rightarrow^\lambda_{Dom(\alpha) \cup \{a\}} e' \Rightarrow \langle \alpha\{[e,\ l,\ r,\ c]_a\} \mid \mu \mid \pi \rangle^\rho_\chi \ \mapsto\ \langle \alpha\{[e',\ l,\ r,\ c]_a\} \mid \mu \mid \pi \rangle^\rho_\chi$

- <**new**: $a,\ a'$ >
  $\langle \alpha\{[R[new(e)],\ l,\ r,\ c]_a\} \mid \mu \mid \pi \rangle^\rho_\chi \mapsto \langle \alpha\{[R[a'],\ l,\ r,\ c]_a, [e,\ l,\ r, \{a,\ a'\}]_{a'}\} \mid \mu \mid \pi \rangle^\rho_\chi$
  a' fresh

- <**send**: $a,\ v_0,\ v_1$ >
  $\langle \alpha\{[R[send(v_0,\ v_1)],\ l,\ r,\ c]_a\} \mid \mu \mid \pi \rangle^\rho_\chi \ \mapsto$
  $\langle \alpha\{[R[nil],\ l,\ r,\ c]_a\} \mid \mu \uplus < v_2 \Leftarrow v_1 >_a \ \mid \pi \rangle^\rho_\chi$
  if $r(v_0) = v_2$
  $\langle \alpha\{[R[nil],\ l,\ r,\ c]_a\} \mid \mu \uplus < v_3 \Leftarrow v_1 >_a \ \mid \pi \rangle^\rho_\chi$

[7]$\alpha(a) = e \times l \times r \times c$ implies that actor a has behavior e and is currently executed at location l with resource map r and ACL c.

if $\pi(l)(v_0) = v_3$

$\langle\alpha\{[R[nil],\ l,\ r,\ c]_a\} \mid \mu \uplus < v_0 \Leftarrow v_1 >_a\ \mid \pi\rangle^{\rho}_{\chi}$

if $v_0 \notin Dom(()r)$ and $v_0 \notin Dom(()\pi)$

- $<$**receive**: $v_0, v_1 >$

  $\langle\ \alpha\{[R[ready(v)],\ l,\ r,\ c]_{v_0}\} \mid\ < v_0 \Leftarrow v_1 >_a \uplus \mu \mid \pi\rangle^{\rho}_{\chi}\ \mapsto$

  $\langle\ \alpha\{[R[app(v,\ v_1)],\ l,\ r,\ c]_{v_0}\} \mid \mu \mid l\rangle^{\rho}_{\chi}$

  if $a \in c$ or $c = \bot$

- $<$**out**: $v_0, v_1 >$

  $\langle\ \alpha \mid \mu \uplus < a \Leftarrow v_0 >_a\ \mid \pi\rangle^{\rho}_{\chi}\ \mapsto\ \langle\ \alpha \mid \mu \mid \pi\rangle^{\rho'}_{\chi}$

  if $a \in \chi$, and $\rho' = \rho \cup (FV(v_0) \cap Dom(\alpha))$

- $<$**in**: $v_0, v_1 >$

  $\langle\ \alpha \mid \mu \mid \pi\rangle^{\rho}_{\chi}\ \mapsto\ \langle\ \alpha \mid \mu \uplus < a \Leftarrow v_0 >_{a'}\ \mid \pi\rangle^{\rho}_{\chi\cup(FV(v_0)-Dom(\alpha))}$

  if $a \in \rho$, then $FV(v_0) \cap Dom(\alpha) \subseteq \rho$

- $<$**migrate** : $l' >$

  $\langle\alpha\{[R[migrate(l')],\ l,\ r,\ c]_a\} \mid \mu \mid \pi \uplus [c',\ r']_l\rangle^{\rho}_{\chi}\ \mapsto$

  $\langle\alpha\{[R[nil],\ l',\ r,\ c]_a \mid \mu \mid \pi \uplus [c',\ r']_{l'}\rangle^{\rho}_{\chi}$

  if $a \in c'$ or $c' = \bot$

- $<$**newloc**: $Y >$

  $\langle\alpha\{[R[newloc(Y)],\ l,\ r,\ c]_a\} \mid \mu \mid \pi\rangle^{\rho}_{\chi}\ \mapsto$

  $\langle\alpha\{[R[l'],\ l,\ r,\ c]_a \mid \mu \mid \pi \uplus [\{a\},\ Y]_{l'}\rangle^{\rho}_{\chi}$ $\quad l'$ fresh

- $<$**attach**: $a' >$

  $\langle\alpha\{[R[attach(a')],\ l,\ r,\ c]_a\} \mid \mu \mid \pi\rangle^{\rho}_{\chi}\ \mapsto$

  $\langle\alpha\{[R[nil],\ l,\ r \cup (a' \to a''),\ c]_a \mid \mu \mid \pi \uplus [c',\ r' \cup (a' \to a'')]_{l'}\rangle^{\rho}_{\chi}$

- $<$**detach**: $a' >$

  $\langle\alpha\{[R[detach(a')],\ l,\ r \cup (a' \to a''),\ c]_a\} \mid \mu \mid \pi\rangle^{\rho}_{\chi}\ \mapsto$

  $\langle\alpha\{[R[nil],\ l,\ r,\ c]_a \mid \mu \mid \pi\ \rangle^{\rho}_{\chi}$

- $<$**register**: $a', a'', l' >$

  $\langle\alpha\{[R[register(a',\ a'',\ l')],\ l,\ r]_a\} \mid \mu \mid \pi \uplus [c',\ r']_{l'}\rangle^{\rho}_{\chi}\ \mapsto$

$\langle \alpha\{[R[nil],\ l,\ r \cup (a' \rightarrow a'')]_a\} \mid \mu \mid \pi \uplus [c',\ r' \cup (a' \rightarrow a'')]_{l'}\rangle^{\rho}_{\chi}$

if $(a' \rightarrow a'') \in r$

- <**unregister**: a', a'', l' >

  $\langle \alpha\{[R[unregister(a',\ a'',\ l')],\ l,\ r]_a\} \mid \mu \mid \pi \uplus [c',\ r' \cup (a' \rightarrow a'')]_{l'}\rangle^{\rho}_{\chi} \ \mapsto$

  $\langle \alpha\{[R[nil],\ l,\ r]_a\} \mid \mu | \ \pi \ \rangle^{\rho}_{\chi}$

  if $(a' \rightarrow a'') \in r$

- <**allow**: a' >

  $\langle \alpha\{[R[allow(a')],\ l,\ r,\ c]_a\} \mid \mu \mid \pi\rangle^{\rho}_{\chi} \ \mapsto$

  $\langle \alpha\{[R[nil],\ l,\ r,\ c \cup \{a'\}]_a \mid \mu | \ \pi \ \rangle^{\rho}_{\chi}$

- <**allow**: $\perp$ >

  $\langle \alpha\{[R[allow(nil)],\ l,\ r,\ c]_a\} \mid \mu \mid \pi\rangle^{\rho}_{\chi} \ \mapsto$

  $\langle \alpha\{[R[nil],\ l,\ r, \perp]_a\} \mid \mu | \ \pi \ \rangle^{\rho}_{\chi}$

- <**allowloc**: a' >

  $\langle \alpha\{[R[allowloc(a')],\ l,\ r,\ c]_a\} \mid \mu \mid \pi \uplus [c',\ r']_l\rangle^{\rho}_{\chi} \ \mapsto$

  $\langle \alpha\{[R[nil],\ l,\ r,\ c]_a\} \mid \mu | \ \pi \uplus [c' \cup \{a'\},\ r']_l \ \rangle^{\rho}_{\chi}$

- <**allowloc**: $\perp$ >

  $\langle \alpha\{[R[allowloc(nil)],\ l,\ r,\ c]_a\} \mid \mu \mid \pi \uplus [c',\ r']_l\rangle^{\rho}_{\chi} \ \mapsto$

  $\langle \alpha\{[R[nil],\ l,\ r,\ c]_a \mid \mu | \ \pi \uplus [\perp,\ r']_l \ \rangle^{\rho}_{\chi}$

- <**disallow**: a' >

  $\langle \alpha\{[R[disallow(a')],\ l,\ r,\ c \cup \{a'\}]_a\} \mid \mu \mid \pi\rangle^{\rho}_{\chi} \ \mapsto$

  $\langle \alpha\{[R[nil],\ l,\ r,\ c]_a\} \mid \mu | \ \pi \ \rangle^{\rho}_{\chi}$

- <**disallow**: $\perp$ >

  $\langle \alpha\{[R[disallow(nil)],\ l,\ r,\ c]_a\} \mid \mu \mid \pi\rangle^{\rho}_{\chi} \ \mapsto$

  $\langle \alpha\{[R[nil],\ l,\ r, \emptyset]_a\} \mid \mu | \ \pi \ \rangle^{\rho}_{\chi}$

- <**disallowloc**: a' >

  $\langle \alpha\{[R[disallowloc(a')],\ l,\ r,\ c]_a\} \mid \mu \mid \pi \uplus [c' \cup \{a'\},\ r']_l\rangle^{\rho}_{\chi} \ \mapsto$

  $\langle \alpha\{[R[nil],\ l,\ r,\ c]_a\} \mid \mu | \ \pi \uplus [c',\ r']_l \ \rangle^{\rho}_{\chi}$

- <**disallowloc**: $\perp$ >
  $\langle \alpha\{[R[disallowloc(nil)],\ l,\ r,\ c]_a\} \mid \mu \mid \pi \uplus [c', r']_l\rangle^{\rho}_{\chi} \ \mapsto$
  $\langle \alpha\{[R[nil],\ l,\ r,\ c]_a\} \mid \mu |\ \pi \uplus [\emptyset,\ r']_l\ \rangle^{\rho}_{\chi}$

2.4 Transactors

The transactor model of Field and Varela [17] is a fault-tolerant programming model for composing loosely-coupled distributed components running in an unreliable environment such as internet into systems that reliably maintain globally consistent distributed state. This model allows the elements of traditional transaction processing to be composed modularly. This model extends the lambda-calculus based on the actor model and formalizes it with the τ-calculus.

2.4.1 Model Properties

Distributed state is a property that many distributed systems have. It denotes that the states of distributed components in a network-connected system are interdependent on one another. The transactor-model maintains these interrelated states in a wide-area network in a consistent way by exposing key semantic concepts related to distributed state in a common language.

Transactors extend the actor model by explicitly modeling node failures, network failures, persistent storage, and state immutability. Just like actors, a transactor encapsulates state and communicates with other transactors via asynchronous message passing. Besides these, in response to a message they may create new transactors, send messages to other transactors, or modify their internal state. In addition to these inherited actor operations, a transactor may stabilize, checkpoint, or rollback.

Stabilization is a transactors commitment not to modify its internal state until a subsequent checkpoint is performed or until another peer actor causes it to rollback due to semantic inconsistencies. After a transactor enters a stable state it can still process messages, it simply cannot change its own state. A checkpoint is not just a commitment to make the current transactor state persistent, but it is also a consistency guarantee. It creates a copy of the transactor's current state, which can be recovered in the event of temporary failures. In order to checkpoint only the globally consistent states, dependence information is carried along with messages. Checkpoint can be thought of as the second phase of a two-phase commitment protocol, where stabilization is the first phase. A rollback brings a transactor back to its previously check-pointed state or makes it disappear, if there is not any such state. Node failures are modeled as rollbacks.

2.4.2 Tau-Calculus

The extensional constructs of the τ-calculus can be divided into two categories: those that encode the traditional actor semantics with explicit state management and those that support distributed state maintenance.
Traditional actor constructs:

- `trans` $\mathtt{e}_1$ `init` $\mathtt{e}_2$ `snart` creates a new transactor with behavior $\mathtt{e}_1$, and initiate state $\mathtt{e}_2$
- `send v to t` sends a message with content `v` to the transactor `t`
- `ready` a transactor waiting to process the next incoming message
- `self` yield the transactor's own name
- `setstate(v)` updates a transactor's state to the value `v`
- `getstate` retrieves the value of the state

Distributed state maintenance constructs:

- `stabilize` current transactor becomes stable/immobile
- `checkpoint` creates checkpoint if stable and consistent
- `dependent?` checks whether the transactor is dependent on other transactors
- `rollback` reverts the transactor to its previous checkpoint

Because of page limitations of this thesis, we refer the interested reader to [17] for the operational semantics of the τ-calculus that Field and Varela developed.

2.4.3 Security and Liveness Properties

The operational semantics for the τ-calculus helps to prove important soundness and liveness properties by showing that globally consistent checkpoints have equivalent execution traces without any node failures or application-level failures. It furthermore shows the possibility to reach globally-consistent checkpoints provided there is some bounded failure-free interval during which checkpointing can occur.

Under certain reasonable preconditions, checkpointing in the transactor model is possible. *Soundness* under these conditions refers that a trace containing node failures and inconsistencies is equivalent to a normal trace (i.e., one containing no node failure) but with possible message losses. In order to show this, Field and Varela prove that arbitrary τ-calculus traces can be simulated by traces containing only the node failure free subset of the τ-calculus. This shows also how global reasoning about state inconsistencies can be reduced to local reasoning about the possibility of message loss.

Liveness, as the other τ-calculus property, refers for example to the fact that using the transactor model operational semantics global checkpoints can be reached. But not all transactor programs can reach global checkpoints. Therefore Field and Varela introduce a *Universal Checkpointing Protocol*, which assumes a set of preconditions that will entail global checkpointing for a set of transactors, and under those preconditions if the protocol terminates then a global checkpointing is reached.

2.5 Hierarchical Model for Coordination

Varela and Agha have grouped the actors that are active on a middleware platform, in their hierarchical model approach [43], into *casts*, so that their activities can be coordinated efficiently and the communication between these can be simplified. Furthermore, they have realized the coordination between the actors by actors designated as directors and also introduced migrating messenger actors to facilitate remote cast-to-cast communication. In this chapter, the properties of this model and its operational semantics will be described.

2.5.1 Model Properties

The hierarchical model uses abstractions that are themselves actors. The *hierarchical model of actor coordination* is based on communication, which requires the sender to explicitly name the target of a message. Therefore, it is kind of a constraint interaction between actors, which does not require shared spaces. It furthermore, explicitly represents locality in order to enable an explicit specification of the structure of the information flow.

The hierarchical model architecture (see Figure 2.2 [43]) consists of directors, messengers, coordinated and uncoordinated actors, and casts. An *actor* can only receive a message, when the coordination constraints for such message receipt are satisfied. The coordination constraints are checked for conformance at special actors named *directors*. A group of actors coordinated by a director is defined as a *cast*. Casts serve as abstraction units for naming, migrating, synchronization and load balancing, and in this architecture they symbolize a group the actors arranged to a singular director. The director-actor relationship forms a set of trees, which is called the coordination forest. *Messengers* are special migrating actors that carry a message from a local cast to a remote cast.

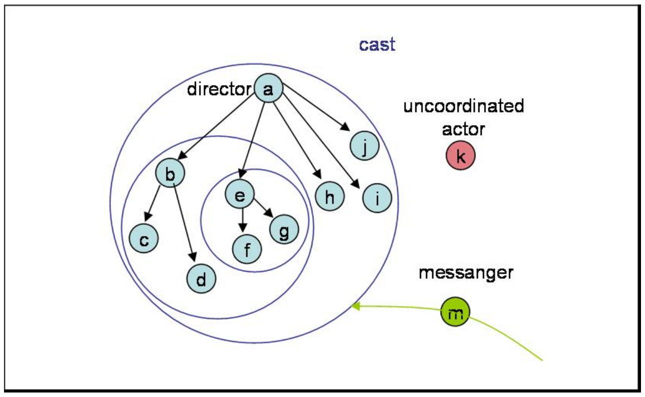


Figure 2.2: Coordinated activity with casts, directors and messengers.

Hierarchical model expands the operational semantics defined in [2] with the concepts of cast and directors. In this subsection, the hierarchical actor language and the reduction rules, which define valid transitions between actor configurations, will be introduced.

Compared to ActorSpaces, in a cast incoming messages to a group are managed by the director actor. A director actor is a computationally active component, which can explicitly support multiple group messaging paradigms. Furthermore, the hierarchical model requires an explicit description of how actors are grouped together. Policies, such as pattern matching according to particular actor attributes, are not directly supported in hierarchical model.

Compared to synchronizers, directors are not declarative. So the coordination of directors does not only allow for dynamic reconfigurability of coordination policies, but also for a hierarchic composition of constraints. Furthermore, the hierarchical model is more restrictive in that it requires actors to belong to one single cast at a given time. By contrast, the groups controlled by synchronizers may overlap arbitrarily. The former associates benefits of more predictable performance, because on the one side the path for hierarchical organization can be more rigid and on the other coordination is more determinate than in synchronizers.

Compared to reflection-based approach the fact that hierarchical model uses abstractions that are themselves actors causes some limitations, like the ability to customize communication by changing the meta-level operations for sending/receiving messages.

2.5.2 Operational Semantics

In this subsection, operational semantics of actor expressions will be discussed by defining a transition on coordinated open configurations.

Hierarchical coordination uses the following call-by-value lambda calculus primitives for creating and manipulating actors: `newactor(e)`, `newdirectactor(e)`, `send(`v_0`,`v_1`)` and `ready(v)`. Where as `newactor(e)` corresponds to `new(b)`; `ready(v)` to `ready(b')`; `send(`v_0,v_1`)` is just as the same as in AL, and `newdirectactor(e)` as a new primitive creates a new actor with behavior e, directed by the creator, and returns its name.

Assuming that two sets, **AT** (atoms) and **X** (variables), are given, the set of values **V**, the set expressions **E** and the set of messages **M** are defined as in AL, except they include the sender, $< ... > a$.

Let $\mathbf{P}_w$[X] be the set of finite subsets of X, $\mathbf{M}_w$[M] be the set of finite multi-sets with elements in M, X_0 X_1 be the set of finite maps from X_0 to X_1, Dom(f) be the domain of f and FV(e) be the set of free variables in e. We define actor configurations as follows.

Definition 2: Actor Configurations (K)

An actor configuration is written as $\langle \delta | \alpha | \mu \rangle^{\rho}_{\chi}$, whereas δ symbolizes a director map, α an actor map, μ a multi set of messages, ρ receptionists and χ external actors. In this configuration ρ, $\chi \in P_w[X]$, $\delta \in X \rightarrow^f X$, $\alpha \in X \rightarrow^f E$, $\mu \in M_w[M]$, and let A = Dom(α) and D = Dom(δ), then:

1. $\rho \subseteq A$ and $A \cap \chi = \emptyset$
2. if $a \in A$, then FV($\alpha(a)$) $\subseteq A \cup \chi$, and if $< v_0 \Leftarrow v_1 >_a \in \mu$ then FV(v_1) $\in A \cup \chi$ for i <2.
3. If $a \in D$, and $a_0 = \delta(a)$, $a_1 = \delta(a_0)$, $a_2 = \delta(a_1), \ldots, a_n = \delta(a_{n-1})$, such that $a_n \notin D$, then $\forall i \in 0..n : (a_i \in A$ and $a_i \neq a)$.

The last rule restricts actor configurations so that (1) all directors in the coordination forest path for an actor belong to the same configuration, and (2) no cycle are allowed in the actor-director relationship.

Definition 3: Coordination forest path (Δ) coordination forest path for an actor (a), $\Delta(a)$, in a configuration κ with director map δ, is defined as:

$$\Delta(a) = \begin{cases} \{a\} & if\ a \notin Dom(\delta) \\ \{a\} \cup \Delta(\delta(a)) & otherwise \end{cases}$$

R in the following definition ranges over the set of reduction contexts. The purely functional redexes inherit in the operational semantics from the purely functional fragment of the AL. The actor redexes are: newactor(e), newdirectedactor(e), send(v_0, v_1) and ready(v).

Definition 4: The single-step transition relation ($\mapsto$)

On the actor configurations is the least relation satisfying the following rules (where a' stands for fresh a):

- <**fun** : a >
  e $\rightarrow^{\lambda}_{Dom(\alpha)\cup\{\alpha\}} e' \Rightarrow \langle \delta \mid \alpha\{a \rightarrow e\} \mid \mu\rangle^{\rho}_{\chi} \quad \mapsto \quad \langle \delta \mid \alpha\{a \rightarrow e'\} \mid \mu\rangle^{\rho}_{\chi}$

- <**newactor**: a, a'>
  $\langle \delta \mid \alpha\{a \rightarrow R[newactor(e)]\} \mid \mu\rangle^{\rho}_{\chi} \quad \mapsto \quad \langle \delta \mid \alpha\{a \rightarrow R[a'],\ a' \rightarrow e\} \mid \mu\rangle^{\rho}_{\chi}$

- <**newdirectedactor**: a, a'>
  $\langle \delta \mid \alpha\{a \rightarrow R[newdirectedactor(e)]\} \mid \mu\rangle^{\rho}_{\chi} \mapsto$
  $\langle \delta \mid \{a' \rightarrow a\} \mid \alpha\{a \rightarrow R[a'],\ a' \rightarrow e\} \mid \mu\rangle^{\rho}_{\chi}$

- <**send**: a, v_0, v_1 >
  $\langle \delta \mid \alpha\{a \rightarrow R[send(v_0, v_1)]\} \mid \mu\rangle^{\rho}_{\chi} \mapsto \langle \delta \mid \alpha\{a \rightarrow R[nil]\} \mid \mu,\ < v_0 \Leftarrow v_1 >_a\rangle^{\rho}_{\chi}$

- <**redirect**: a, v_0, v_1 >
  $\langle \delta \mid \alpha \mid \mu,\ < v_0 \Leftarrow v_1 >_a\rangle^{\rho}_{\chi} \mapsto$
  $\langle \delta \mid \alpha \mid \mu,\ < \delta(v_0) \Leftarrow msg(v_0,\ v_1) >_a\rangle^{\rho}_{\chi}$
  $if\ v_0 \in Dom(\delta),\ \delta(v_0) \neq a,\ and\ v_0 \notin \Delta(a)$

- <**receive**: v_0, v_1 >
  $\langle \delta \mid \alpha\{v_0 \rightarrow ready(v)\} \mid\ < v_0 \Leftarrow v_1 >_a,\ \mu\rangle^{\rho}_{\chi} \mapsto$
  $\langle \delta \mid \alpha\{v_0 \rightarrow app(v, v_1)\} \mid \mu\rangle^{\rho}_{\chi}$
  if $v_0 \notin Dom(\delta),\ \delta(v_0) = a,\ or\ v_0 \in \Delta(a)$

- <**out**: v_0, v_1 >
  $\langle \delta \mid \alpha \mid \mu,\ < v_0 \Leftarrow v_1 >_a\rangle^{\rho}_{\chi} \mapsto \langle \delta \mid \alpha \mid \mu\rangle^{\rho'}_{\chi}$
  if $v_0 \in \chi, \quad and\ \rho' = \rho \cup (FV(v_1) \cap Dom(\alpha))$

- <**in**: v_0, v_1 >
  $\langle \delta \mid \alpha \mid \mu\rangle^{\rho}_{\chi} \mapsto \langle \delta \mid \alpha \mid \mu,\ < v_0 \Leftarrow v_1 >_{a'}\rangle^{\rho}_{\chi\cup\rho\cup(FV(v_1)-Dom(\alpha))}$
  if $v_0 \in \rho$, $FV(v_1) \cap Dom(\alpha) \subseteq \rho$

The *redirect* and *receive* transitions rule ensure that all directors up to the common ancestor between the sender and the target actors get notified and controlled when to actually send a message to a target actor. As mentioned before, information flows here locally. That means, if two actors belong to the same cast, outside actors and directors do not need to be notified.

The last two transactions show the openness of actor configurations. The *out* transition represents a message delivery to an external actor, and the *in* transition represents a message coming from an outside system to one of the configuration's receptionists.

2.6 Mobile Ambients

Mobility involves the authorization to enter or exit certain domains. In mobile computation, it is not realistic to imagine that an agent can migrate from any point A to any point B on the internet. In order to do this it first has to obtain permission to exit its administrative domain, then obtain permission to enter someone else's administrative domain, and then obtain permission to enter a protected area of some machine, where it is allowed to run. Access to information is controlled usually by multiple levels of authorization, like local computers, local area networks and the internet. Mobile programs must be equipped to navigate this hierarchy of administrative domains, at every step obtaining authorization to move further. Therefore, Cardelli and Gordon [8] propose Mobile Ambients that captures notions of locations, mobility, and authorization to move at the most fundamental level. Mobile ambients is a paradigm of mobility, where computational ambients are hierarchically structured, agents are confined to ambients, and ambients move under the control of agents. To realize these, mobile ambients allow the movement of self-contained nested environments that include data and live computation, instead of moving single actors. In this subsection, mobile ambients with their computational semantics will be described.

2.6.1 Model Properties

Ambients as an environment, where computation happens, have a boundary around them. A boundary determines what is in and what is outside an ambient. Examples of ambients and boundaries, in this sense, can be a web page bounded by a file, or a single data object bounded by "itself". A boundary implies some flexible addressing scheme that can denote entities across the boundary, like symbolic links, and Uniform Resource Locators. Mobility is facilitated by flexible addressing, but at the same time it is a cause of problems when the addressing links are broken.

An ambient can be nested within other ambients, therefore the administrative domains are often organized hierarchically. Each ambient has unique names, which is used to control access.

Ambients can furthermore, be moved as a whole. If an agent is moved from one computer to another, its local data should move accordingly and automatically.

Each ambient has local actors, which are the computations that run directly within the ambient and control the ambient, for example if the ambient has to move.

Ambient model involves besides ambient *boundaries*. The existence of separate locations is represented by a topology of boundaries. This topology induces an abstract notion of distance between locations. Locations are not uniformly accessible, and are not identified by globally unique names. Process mobility is represented as crossing of boundaries, and security is represented as the ability or inability to cross boundaries. Interaction between processes can only happen with consideration of boundaries and their topology.

Cardelli and Gordon [8] use two calculi by describing Mobile Ambients. One is based on mobility primitives and the other one extending it with local communications.

When `P` and `Q` symbolize processes and `n` the name of an ambient, `P` $\rightarrow$ `Q` describes the evolution of a process `P` into a process `Q`; mobility primitives are as follows:

- <`P,Q ::= (`ν`n)P`>
  restriction: creates a new name within a scope `P` and is transparent with respect to reduction $(P \rightarrow Q \Rightarrow (\nu n)P \rightarrow (\nu n)Q)$

- <`P,Q ::= 0`>
  inaction: shows inactivity, and does not reduce

- <`P,Q ::= P | Q`>
  composition (parallel): parallel execution is commutative and associative $(P \rightarrow Q \Rightarrow P \mid R \rightarrow Q \mid R)$

- `<P,Q ::= !P>`
  replication: represents iteration and recursion, unbounded replication of process `P` (`!P` can process as many replicas of `P` as needed, and is equivalent to $P\,|!P$.)

- `<P,Q ::= n[P]>`
  ambient: `P` is the process running inside the ambient ($P \to Q \Rightarrow n[P] \to n[Q]$)
  In general, an ambient is denoted as a tree structure by the nesting of ambient brackets. Each node may contain in addition to subambients a collection of non-ambient processes running in parallel. ($n[P_1 \mid \dots \mid P_n \mid m_1 \mid [\dots] \mid \dots \mid m_q[\dots]]$) where $P_i \quad \neq n_1[\dots]$

- `<P,Q ::= M.P>`
  action and capabilities: executes an action regulated by the capability `M`, and then continues as process `P`

For each capability `M` there is a rule for reducing `M.P`, with the following primitives:

- `<M ::= in n>`
  can enter **n** *(entry capability)*: `in n` can be used in the action `in m.P`, and the reduction rule is: $n[in\ m.P \mid Q] \mid m[R] \to m[n[P \mid Q] \mid R]$
  If successful, this reduction transforms a sibling `n` of an ambient `m` into a child of `m`. After the execution, the processes `in m.P` continues with `P`, and both `P` and `Q` find themselves in a lower hierarchical level. If no sibling m can be found, the operation blocks until a time when such a sibling exists. If more then one `m` siblings exist, any one of them can be chosen.

- `<M ::= out n>`
  can exit **n** *(exit capability)*: `out m` can be used in the action `out m.P`, and the reduction rule is: $m[n[out\ m.P \mid Q] \mid R] \to n[P \mid Q] \mid m[R]]$
  If successful, this reduction transforms a child `n` of an ambient `m` into a sibling of `m`. If the parent is not named `m`, then the operation blocks until a parent `m` exists. After the execution, the process in `m.P` continues with `P`, and both `P` and `Q` find themselves at a higher hierarchical level.

- `<M ::= open n>`
  can open ***n*** *(open capability)*: `open m` can be used as `open m.P`, and the reduction rule is: *open* $m.P \mid m[Q] \rightarrow P \mid Q$
  If no ambient `m` can be found then the operation blocks itself until such an ambient is found. If there is more then one ambient m, then it chooses one.

Communication primitives extend the mobility primitives with the following primitives:

- `<P,Q ::= (x).P>`
  ambient input: captures a capability from the local ether and binds it to a variable within a scope

- `<P.Q ::=` $\langle M \rangle$`>`
  ambient output: releases a capability into the local ether of the surrounding ambient

The reduction for the ambient I/O, where $P\{x \Downarrow M\}$ symbolizes the substitution of the capability `M` for each tree occurrence of the variable `x` in the process `P`, is : $(x).P \mid \langle M \rangle \rightarrow P\{x \Downarrow M\}$

Communications also enrich the capabilities in order to include paths. For the purpose of communication, names and variables are added to the collection of capabilities. Names and capabilities are the only entities that can be communicated. Multiple capabilities can be combined into paths, especially when one or more of them are represented by input variables. The path-forming operation on capabilities is `M.M'`, for example, `(in n.in m). P` is interpreted as `in n.in m.P`.

2.6.2 Operational Semantics

In this subsection, we are going to give operational semantics suggested by Cardelli and Gordon [8] of the calculus described in the previous subsection, based on a structural congruence between processes, $\equiv$, and a reduction relation, $\rightarrow$.

Structural Congruence

$P \equiv P$	(StructRefl)
$P \equiv Q \Rightarrow Q \equiv P$	(Struct Symm)
$P \equiv Q, Q \equiv R \Rightarrow Q \equiv R$	(Struct Trans)
$P \equiv Q \Rightarrow (\nu n)P \equiv (\nu n)Q$	(Struct Res)
$P \equiv Q \Rightarrow P\|R \equiv Q\|R$	(Struct Par)
$P \equiv Q \Rightarrow !P \equiv !Q$	(Struct Repl)
$P \equiv Q \Rightarrow n[P] \equiv n[Q]$	(Struct Amb)
$P \equiv Q \Rightarrow M.P \equiv M.Q$	(Struct Action)
$P\|Q \equiv Q\|P$	(Struct Par Comm)
$(P\|Q)\|R \equiv P\|(Q\|R)$	(Struct Par Assoc)
$!P \equiv P\|!P$	(Struct Repl Par)
$(\nu n)(\nu m)P \equiv (\nu m)(\nu n)P$	(Struct Res Res)
$(\nu n)(P\|Q) \equiv P\|(\nu n)Q$ if $n \notin fn(P)$	(Struct Res Par)
$(\nu n)(m[P]) \equiv m[(\nu n)P]$ if $n \neq m$	(Struct Res Amb)
$P\|0 \equiv P$	(Struct Zero Par)
$(\nu n)0 \equiv 0$	(Struct Zero Res)
$!0 \equiv 0$	(Struct Zero Repl)

Furthermore, Cardelli and Gordon identify processes up to renaming of bound names, by which they mean that these processes are understood to be identical, as opposed to structurally equivalent: $(\nu n)P = (\nu m)P\{n \leftarrow m\}$ if $m \notin fn(P)$

Reduction

$n[inm.P\|Q]\|m[R] \rightarrow m[n[P\|Q]\|R]$	(Red In)
$m[n[outm.P\|Q]\|R] \rightarrow n[P\|Q]\|m[R]$	(Red Out)
$openn.P\|n[Q] \rightarrow P\|Q$	(Red Open)
$P \rightarrow Q \Rightarrow (\nu n)P \rightarrow (\nu n)Q$	(Red Res)
$P \rightarrow Q \Rightarrow n[P] \rightarrow n[Q]$	(Red Amb)
$P \rightarrow Q \Rightarrow \quad P\|R \rightarrow Q\|R$	(Red Par)

$P' \equiv P, P \rightarrow Q, Q \equiv Q' \Rightarrow P' \rightarrow Q'$	(Red $\equiv$)
$\rightarrow^*$	reflexive and transitive closure of $\rightarrow$

Reduction relations give the behavior of processes. The first three rules are the one-step reductions for *in*, *out* and *open*. The next three rules propagate reductions across scopes, ambient nesting and parallel composition. The final rule allows the use of equivalence during reduction. Finally, $\rightarrow^*$ is the chaining of multiple reduction steps.

Two processes are contextually equivalent if and only if whenever they are inserted inside an arbitrary enclosing process, they admit the same elementary observations. In the setting of Mobile Ambients contextual equivalence is formulated in terms of observing the presence of top-level ambients. A process P *exhibits an ambient named n*, $P \downarrow n$, only if `P` is a process containing a top-level ambient named `n`, and a process P *eventually exhibits an ambient named n*, P$\Downarrow$n, only if after some number of reductions, `P` exhibits an ambient named `n`. Defining formally:
P$\downarrow n \doteq P \equiv (\nu m_1 ... m_i)$ (n[P'] $|P'')$ where $n \in \{m_1 ... m_i\}$
P$\Downarrow n \doteq P \rightarrow^*$ Q and $Q \downarrow n$
Contextual equivalence in terms of the predicate $P \downarrow n$ is defined as follows: Let a context `C()` be a process containing zero or more holes, and for any process `P`, let `C(P)` be the process obtained by filling each hole in `C` with a copy of `P`. Let contextual equivalence be the relation $P \simeq Q$ defined by: $P \simeq Q$ $\forall n$ and `C()`, $C(P) \Downarrow n \Leftrightarrow C(Q) \Downarrow n$. And if there exists an `R` such that $P \rightarrow^* R$ and $R \simeq Q$, it is written as P$\rightarrow^* \simeq Q$.

In computational semantics of the communication property of Mobile Ambients `P` and `Q` range over large classes. So the following semantics are changed and respectively added:

Structural Congruent

$P \equiv Q \Rightarrow M[P] \equiv M[Q]$	(Struct Amb)
$P \equiv Q \Rightarrow (x).P \equiv (x).Q$	(Struct Input)

$\varepsilon.P \equiv P$ (Struct ε)

$(M.M').P \equiv M.M'.P$ (Struct .)

Identification of the processes up to renaming of bound variables:
$(x).P = (y).P\{x \leftarrow y\}$ if $y \notin f\nu(P)$

Reduction

$(x).P| \quad \langle M \rangle \quad \rightarrow P\{x \leftarrow M\}$ (Red Comm)

Now that processes may contain input-bound variables, the definition of contextual equivalence can be modified as follows: let $P \simeq Q$ if and only if for all C() such that $f\nu(C(P)) = f\nu(C(Q)) = \emptyset$, $C(P) \Downarrow n \Leftrightarrow C(Q) \Downarrow n$.

2.7 Vulnerabilities and Security Instruments

In the previous sections, we have described the actor model and its several variants. In this section, we are going to analyze the main security threats of the actor model.

The actor model has two main components: actors and locations. These components are all vulnerable by each other:

- A location may attack an actor's data
- A location may attack an actor's code and control flow
- An actor may attack another actor's data
- An actor may attack a location

The main problem concerning data that has been carried by an actor is that, mobile actors must disclose its information about code and data if it wants to be executed. Chess et al.'s [9] argument is that it is impossible to prevent actor tampering unless trusted hardware is available in locations, where actors come together. Without such hardware, a malicious location can always manipulate actors code, control flow or data. Possible attacks include simple spying on the actor's data or code and modifying the actor's data, code or control flow, terminating the actor or changing its state before it migrates from the location. These types of attacks will be distinguished in the next subsection, regarding the type of information that is targeted.

2.7.1 Types of data that can be manipulated

An actor is a piece of mobile code, so this code and control flow is the data that makes an actor. Besides these, actors carry some static and some dynamic data.

Actor's data:

This is the data, that an actor carries, which is usually transparent to the actor itself. Actor's data contains two types of data, *static* (nonmutable) and *dynamic* (mutable).

Static data is not modifiable during actor transmission. This data can be *public*, like the name of the items the actor wants to buy, or *private*, like the maximum amount to pay for an item. As security of public static data only has to fulfill integrity requirements, private static data has to fulfill both integrity and confidentiality requirements.

Dynamic data is the data that is modified during actor transmission. An example for dynamic data is the list of visited locations with the associated itinerary-price information. This type of data can also be public or private. In some applications, an actor sends some up-to-date information to the owner, for example when the owner is getting close to a physical shop that sells the item. A secure system must keep this private data confidential and protect its integrity. Otherwise malicious locations may read it and react dependently or even manipulate it for their own benefit.

Encryption is a method to protect private data. A very common attack to private encrypted data, that an actor carries, is called *cut-and-paste-attack*. This attack is characterized in Figure 2.3. In cut-and-paste-attack the malicious location `p` wants to sniff[8] the plain text `m`, which is saved in actor `a`, encrypted with the public key of the location `q`, which makes it only readable by `q`. When an actor is executed in a malicious location `p`, if the location `p` knows the data is encrypted with the public key of the location `q`, `p` may copy it to another actor `b` (1), while the actor `a` is located in `p`. Then, the location `p` sends actor `b`, to the location `q` (2). Location `q` thinks that the cypher text is from `p` and the decrypts it using his private key. Actor `q` furthermore encrypts it with the public key of the location `p` (3) and sends it back with actor `b` (4). Location `p` decrypts the cipher text using his private key and reaches the private data.

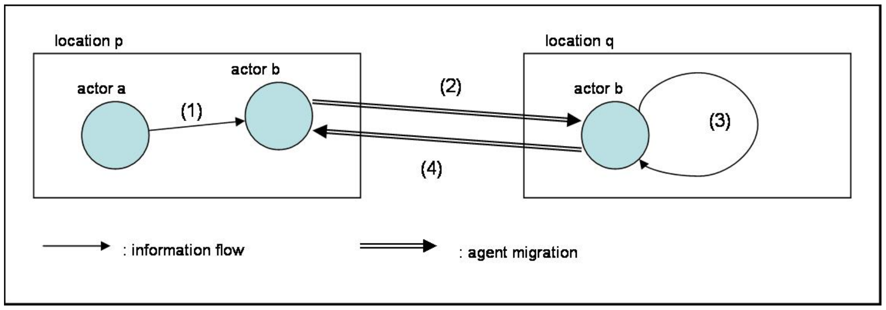


Figure 2.3: Cut-and-paste attack

A possible solution to cut-and-paste attacks is proposed by Roth and Conan [36]. They suggest using a Message Authentication Code (MAC) that binds the actor owner's public key with a symmetric key, and let this MAC be transmitted with the static part of the actor.

Actor's code and control flow:

General requirements for actor's code are integrity and confidentiality. However, because the actor must disclose at least some part of its code, so the complete code cannot be confidential for locations. The main attacks to the confidentiality of the actor's code are *black-box-attacks* and *sabotage*. In black-box-attack, the actor

[8]In the area of information security this type of attacks, where a not authorized third party gains information, are called sniffing-attacks.

is executed with different input parameters and different service-calls several times. The code itself is a black-box, and the malicious location tries the get information that it is not authorized for by comparing the input-output pairs. In sabotage the malicious actor alters code at random points to corrupt the actor.

Since the code cannot be protected as a whole, disclosing only some lines of it may perform an understandable overall semantics. Still if large parts of the actor are executed, a location can learn the overall semantics. It can spy out the code to analyze the actor's intended behavior. If a location successfully analyzes the code it may even understand the goal of the actor in a negotiation. Spying out code becomes even easier if the actor is built from reusable software components or uses code libraries that are already available at each location. One solution might be actors carrying their own libraries.

As the next step malicious location can manipulate the code, which is known as *spoofing attack* in the area of information security. This way the location may insert a component that makes it possible to remotely control the actor after it has migrated to another location.

2.7.2 Organizational solutions

Organizational solutions confine the openness of the mobile agent systems to achieve some level of security. Since there are two main elements in actor based systems, we are going to concentrate on possible solutions to the problem of data and information security, using trusted actors and location-reputation.

Trusted actors

The approach of *trusted actors* refers to the idea that trusted actors migrate only to and from trusted locations. In order to achieve this, either the actor must have a predefined itinerary that includes only trusted locations or in the case of dynamic routing of actors, we can assume that the location network consists of trusted locations only and no location will allow a mobile actor to migrate to a not trusted one. From the point of view of the location, it must be ensured that the only mobile actors accepted are those that have only visited trusted locations before.

This is also a solution for the problem that an actor carrying secure sensitive information, such as a secret key of its owner. This information can be encrypted with the public key (K^{-}) of the trusted location, making it possible for only the location to read it. Therefore, an actor can even carry the (K^{+}) of its owner, if the location is trusted.

In order to create a network of trusted locations, rules that are positioned outside the mobile systems must be established. However there is the problem of notifying the actors and the locations after a location becomes malicious.

Location-Reputation

Location-Reputation [5] can be seen as an approach to allow mobile actors to decide which locations are trusted in an open network environment. This approach is built over a *social control*[9] mechanism, according to which actors can complain about any location with certain registration locations. This way malicious locations lose reputation.

There are several problems with this approach that have to be solved. The actors can only complain about the locations but not refuse a bad reputation. Another problem is, since a malicious actor can always complain about a good location, it may decrease the reputation of this location by keeping the reputation of his home-location higher. A further problem is that behaving properly for a long time does not necessarily mean that an actor is not malicious. A location might work properly over a long time, accumulating a high reputation, and then begin attacking actors promptly.

Besides these two organizational solutions that circumvent the security problems, there are several instruments to solve these. These instruments are going to be discussed in Subsections 2.7.3 and 2.7.4.

2.7.3 Protecting Actors

The techniques of protecting actors from attacks can be grouped in two: techniques that prevent attacks on actor and techniques that detect attacks on actors.

[9]Social Control is defined as a type of behavior, which enforces all members of a social group to behave according to rules that were defined within this group.

Both of these can be used in combination in order to achieve an heir level of security.

2.7.3.1 Preventing Attacks on Actors

Once an actor has arrived to a location, little can be done to stop the location from treating the actor as it likes. There is no universal solution to the malicious location problem, but some partial solutions have been proposed. In this subsection, we are going to describe different approaches that can be used in order to protect actors from malicious actors and locations.

Encryption Functions

Sander and Tschudin [37] propose *encrypting functions* to encrypt an actor while still allowing it to execute at remote locations. The goal of this approach is to make it difficult to analyze an actor. The general idea is to let a location execute the actor carrying an encrypted function without knowing the original function.

The general protocol is as follows:

1. Alice encrypts the function f, resulting in $E(f)$.
2. Alice send the actor $A(E(f))$ that executes $E(f)$ to Bob.
3. Bob executes $A(E(f))$ using input x.
4. Bob sends $A(E(f))(x)$ back to Alice.
5. Alice decrypts $A(E(f))(x)$ and obtains x.

This protocol works for polynomials with smooth integers[10], which allows a malicious location to apply black box[11] attacks on the actor in order to get information about the function.

This technique can so far unfortunately be used only to protect basic mathematical functions. However, with the development of enhancement to this approach, in the future, it might work with actors in general.

[10]Smooth integers are those that consists only of small prime numbers.

[11]An actor is a black box if at any time code and data of the actor cannot be read and modified.

Time-Limited Black Boxes

The encrypted functions approach results in an encrypted actor that will be completely protected against spying out code and data for the lifetime of an actor. Hohl [20] proposes a technique that will protect the actor for a limited time by relaxing the requirement of an actor being a black box for its entire life and introduces a new property that he calls *time-limited black boxes.*

Hohl's approach depends on the difference between understanding every line of the code and understanding the semantics of an actor. Therefore the author proposes to mess up code so that an automated program analyzer will need a large amount of time to understand the code. Without being able to understand the code and data, a malicious location cannot modify the actor in its own favor. The only possible attack is then the sabotage of the actor.

Hohl proposes three mess-up techniques, each of which is for actor's code, data and control flow. Furthermore, in order to protect data items Hohl proposes *variable recomposition.* The technique works on a set of variables and cuts each variable into pieces, and creates new variables that are composed of segments from different original variables. Access to these variables must be rearranged accordingly. Finally, Hohl suggests converting control flow elements, that is, evocative of finite state machines, where in each state some atomic expression is calculated.

Another important issue of this approach is the need to take precautions for the time after the protection interval. All data stored within the actor's state or data package and needed to interact with other parties must be attributed with an expiration date, which is encapsulated with each data item in an unforgeable way. This will make the other locations to verify it and reject the actor and actor's data if necessary.

The most important drawback of this approach is that only the existence of an exact expiration date makes it impossible to use long-living actors and contradicts one of the principles of software actors in general. The longer the interval is, the higher is the probability that the actor will be attacked; the shorter the interval is, the shorter the itinerary can be [5]. Leasing may help here to overcome the problem. In leasing the expiration date can be extended (renewal of a lease) depending on the need and current level of security.

Environmental Key Generation

The goal of the *environmental key generation* [34] is to have *clueless actors*, so that their private data can be protected against the sniffing attacks from the malicious locations. This is realized by not allowing actors to know what their behavior will be because portions of their code or data is encrypted with a secret key. The scenario is as follows: The actor has a cipher text message and a method to search the environment for the data needed to generate the secret key for decryption. When the information is found, the actor can generate the key and decrypt the message. Without that key, the actor has no idea about the content and the semantic of the encrypted message; that is, the actor is clueless.

The general idea is to let the actor carry the hash value of information and let it compare this with the computed hash values at the remote location. If they match, the hash value is used to decrypt additional information or code that should be processed.

By using environmental key generation the intention of the actor is protected as not giving it full knowledge about its task and its code is protected against actions by using encryption.

The hosting malicious location can still attack the actor after it has decrypted the message, but to do this the actor must be executed. A priori analysis by dictionary attacks[12] is also very expensive.

Riordan and Schneider propose in addition more protocols considering the time when keys are generated. These protocols can be used to ensure that a particular data item can be encrypted only before or after, and in combination during a time interval. All of these relay on the existence of a minimally trusted third party that is used for key generation but that does not need to understand the semantics of the

[12]A method used to break security systems, specifically password-based security systems, in which the attacker systematically tests all possible passwords beginning with words that have a higher possibility of being used, such as names and places. The word "dictionary" refers to the attacker exhausting all of the words in a dictionary in an attempt to discover the password. Dictionary attacks are typically done with software instead of an individual manually trying each password.

information. The interested reader may read more about these protocols in [34].

The major concern of this approach is that it protects data and code but does not protect the behavior of the agent. For example if we consider an actor performing a patent search. It searches through the patent data store, computes hash values of keywords, and compares them with the hash value that it carries. The location in such a case will not be able to analyze which type of patents the actor is searching for. Another problem is that decrypting pieces of code at runtime implies that it must be allowed to create code dynamically, which might be prohibited by the hosting location.

2.7.3.2 Detecting Attacks on Actors

In the theory of mobile actors, there are lots of mechanisms for detecting attacks on mobile actors. In this subsection, these mechanisms are going to be discussed with their strengths and weaknesses.

Replication of Locations

The general idea here is to replicate the actor so that not only a single, but many actors of the same type roam the network with the same task [5]. Replicating and comparison can be used to mask the effects of executing an actor on a faulty processor [38]. Because, according to Schneider's argument a malicious location can corrupt a few copies but there will be enough replicas that the encounter can be avoided and the task successfully completed.

In order to tolerate faulty locations, the behavior of the actor has to be deterministic at every location. Therefore every stage except the source and the destination is being replicated. Every location in stage i, takes as its input the majority of the inputs that are received from the replicas comparing the stage $i-1$ and sends its output to all of the locations that it determines comprise stage $i+1$.

Assuming that an actor executes in a sequence *stage actions*, S_i, where $0 \leq i \leq n$, actors process an itinerary that consists of several locations, L_i, $0 \leq i \leq n$. Let L_0 be the home location and L_n be the destination, which might be equal to the home location. Schneider assumes that there are not only one location for each stage but many of them providing the same set of services and behaving the same, $S_i = \{L_{0,i}, L_{1,i}, \ldots\}$.

The actors home location replicates the actor and sends copies to all locations $L_{i,1} \in S_i$. Each replicated actor processes the same action in different locations. At a stage i, each location $L_{k,i}$ sends a copy of the actor to all locations of the stage i+1. So each location at stage S_{i+1} receive many copies of the same actor, probably with different data packages, since they might have been attacked. It compares all actors with each other and chooses the actor with the most frequent stage to execute. By this decision, it assumes that not more then half of all locations in stage S_i were malicious.

Without the knowledge of penultimate location, which has replicated the actor, a sufficient large number of processors could behave as though there are in the penultimate stage and impose a majority of malicious actors on the destination. Schneider suggests that, if actors carry a privilege, malicious actors can be detected and ignored by the destination. For implementing such privileges, he furthermore suggests two protocols, one of which is based on *proactive secret sharing* [22] and the other on authentication chains.

The main problem with this approach is that it is unrealistic in common application domain to assume that locations can be replicated [5]. More then one location means, for example in an e-commerce scenario for online auctions, more then one auction for one item at the same time, which is not possible without extreme synchronization overhead. In addition, in Schneider's approach, actors must be replicated as well, and it depends on the actor's task whether this is possible or not. For example in the above mentioned auction scenario, if the actors are replicated this may end up with actors from the same buyer biding against each other in the same auction.

Replication of Actors

Yee [47] proposed another solution based on actor replication. Whereas two actors are created for one item, each of which are processing a predefined itinerary in reverse orders. And if there is a single malicious location in the itinerary, that attacks the actor's data integrity, this approach will make it possible to detect any tampering with the actor by comparing the results of the two actors.

This approach is built up on the idea that, if one of the replicate actors reaches the malicious location after reaching the location providing the best price, then malicious location changes the actor's data. But since, the other replicate actor will reach the malicious location before reaching the location providing the best price, the malicious location will not be able to modify the data about the lowest price.

Although this approach seems to be more applicable then assuming the locations can be replicated, it requires the actors itinerary to be predefined. Furthermore it does not work if more than a single malicious location is present.

Actors as Black-Boxes

Actors, which are converted to a black-box, use basic cryptographic mechanisms like authentication, encryption and the establishment of secure communication channels in an almost unmodified manner [21].

The malicious locations are no longer able to determine the semantics of an actor, if the actor is converted into black-box, but it might still be able to run black-box attacks. By re-executing an actor, a malicious location can gain information about the actor's behavior; that is it can determine how it reacts on the given input parameters [5].

Hohl and Rothermel [21] propose a technique to make the actors capable of detecting such attacks. The main idea of preventing black-box attacks is to let the actor verify that the location delivers the same result for equal system call and any inquiries to the location. In order to ensure this a trusted component that controls the mechanism is needed and, this component has to be placed on a trusted location, a registry. For each input, the actor sends a message to the trusted registry. The message contains a unique statement identification number associated with the actor's source or by code and a hash value of the data response. The registry contains information about all results of inquiries of the actor. Every time the registry receives such a message, it compares the statement identification number with the hash value. If they match, then the registry acknowledges the message, otherwise it sends an error message. In case of error message the actor knows that the location wants to re-execute it and reacts accordingly.

In order to make sure that an actor resides at the expected place, a *two-way authentication* has to be performed when the actor moves to a new node [21]. Due to the black-box property of actors, the actor can run this protocol even when it already resides on the new node. For authentication, a standard protocol can be applied. The only modification consists in the specification that an actor must not communicate before the successful authentication of both partners. The reason for that specification lies in the possibility that a malicious location may mask itself and send out an actor to another location. This actor then could simply act as a relay for the malicious host, forwarding the messages from the other actor and sending back the answers from the correct host.

Cryptographic Techniques

Cryptographic techniques are going to be discussed here, which are concerned with:

- Protecting an actor's read-only data,
- Revealing an actor's data at specific locations and,
- Protecting an actor's dynamic data

In order to protect the integrity of the read-only data of an actor, which is immutable during its the life, techniques based on *asymmetric cryptography* can be used [5]. At the actor's home location, A_0, the read-only data is signed. This is done by encrypting the hash value[13] , which is extracted from the read-only data, with the private key of the data owner. The signature becomes a part of the actor, and the private key remains in the home location. When the actor migrates to new location, the location verifies the signature using the public key of the home location, usually in form of a trusted certificate.

Cryptographic techniques can be furthermore used to protect the data item in such a way that they can be read only at certain locations. This is necessary

[13]Here a one-way hash function, like MD5, is used.

when data items are defined at the home location but will be read only at other locations or when the data items defined at different locations will be read at the home location. This problem may be solved by encrypting the data item with the public key of the destination location. In that case an additional signature may be used to ensure that the data item has not been modified. Karnik [23] proposed an approach very similar to this. The disadvantage of this solution is that the data item has to be encrypted `n` times if it has to be readable at `n` locations [5]. Roth and Conan [36] proposed a solution to this problem. They suggest to use a hybrid encryption technique, whereas the data item is encrypted using symmetric encryption and only the key is encrypted `n` times using the public key of all target locations.

An actor's dynamic data consists of data that the actor wants to carry to other locations on its itinerary or that it wants to send to its owner [5]. The dynamic data items, which the actor carries, can be protected against sniffing attacks using encryption and against spoofing attacks by using digital signature. The main difference between protecting the dynamic and static data is that the dynamic data changes as the actor migrates from one location to the next. There for general idea Braun and Rossnak suggest is to let the location to digitally sign each data item it has transferred to the actor and encrypt some data item with the public key of the destination location. There are two attacks against this proposal: deleting the data item without understanding, and modifying the encrypted data, signing it with the private key of the malicious location and encrypting them with the public key of the owner. Another problem with this approach is that the size of the actor increases as it migrates from one location to the next and only a small portion is real information.

Young and Yung [48] propose the *sliding encryption* as a solution to this problem, in which an actor can gather small amount of data from several locations and encrypt them using a public key without spending too much storage space. Sliding encryption uses a large key, while at the same time, taking into account the limited storage of an actor, therefore it aims conserving space rather than time, which might be of importance for actors that collect small amounts of data on many different hosts.

Karnik [23] proposes *append-only logs* as a part of an actor's state to protect a data item from modifications. Whereas the new data that an actor obtains from the location it currently visits is inserted in an *AppendOnlyContainer* and signed by the current location. In addition, a checksum is carried by the actor, which is initiated at the actor's home location with a nonce that is encrypted with the actor owner's public key and kept secret at the home location. The checksum is updated after a new data item has been added. According to this method, the data item is still readable at later locations. If the confidentiality also has to be protected, then Karnic suggests encrypting the data by using the public key of the owner. When the actor returns to its home location, the owner can verify the integrity by unrolling the encrypted checksums. The drawback of this approach is that it is vulnerable to a cut and paste attack if a malicious location that knows a checksum as computed by a location visited earlier [36].

Yee [47] suggest partial result authentication code (PRAC) as a cryptographic technique to protect actor's dynamic data. Encryption of partial results is a faster method, since it relies on secret key cryptography. By using this method, the data achieved before an actor migrates to a malicious location remains unaffected. When an actor starts, it has a key for the first location, and before leaving the current location dynamically generates a new key, by using an m-bit to m-bit one-way hash function for the next location. Each key can only be used once. Before leaving a location the actor puts the partial results it got from this location into a message. To prevent integrity, a MAC is computed using the key, associated with this location, on the message. The PRAC consists of the message and the MAC. Afterwards the message can be sent to the owner of carried along to the following locations. A similar functionality can be achieved using asymmetric cryptography by letting the host produce a signature on the information instead [6].

Execution Tracing

Execution tracing, proposed by [45] and evolved by Braun and Rossak [5], allows detecting code and control flow manipulations, by makes it possible to trace an actor, which is to save the history of execution at a single specific location in an unforgeable way, such that it can be proved that some malicious location has tampered with the location. The general idea is that each visited location sends a message to the actor's home location, or to some other trusted location, a message after the actor migrates to the location, before it starts to execute the actor and before it the actor migrates to the next location.

The notification of the forthcoming migration message is sent to the home location of the actor, and consists of the name of the sender location, the name of the next location, a hash value of the current state and the executed trace, and a unique identifier. The message, besides the name of the sender, is signed with the private key of the sender location. The second message is a notification of the migration. This message consists of six parts as follows: the name of the sender; the name of the destination location; and the name of the home location; actor's code and current state encrypted with the public key of the destination; the hash value of the actor's current state and the name of the destination location encrypted with the private key of the sender location; and the information signed by the home location, which consists of hash value of the actor's code, a time stamp to guarantee freshness, and a unique identifier to prevent replay attacks. After performing the authentication and verifying the integrity the destination location sends an acknowledgement message to the actor's home location in order to confirm the state it has received the message. This acknowledgment message consists of the name of the current location, and the last two parts of the migration message signed with the private key of this location. And the actor's owner can prove the correct execution in execution tracing by retrieving the execution trace of the suspicious location and simulating actor execution at the home location.

The drawback of this approach is the size of the trace and that it has to be transmitted from each location to the trusted server and its respective home location, which slows the overall performance of the actor [5]. To reduce the size of the trace Vigna [45] does not trace the entire program but only selected statements where the control flow changes. Another drawback is the necessary management of created logs.

Techniques for Detecting Itinerary Manipulations

The goal of the techniques to detect the itinerary manipulation is to prevent a location from stopping an actor to migrate to other locations or to send it to undesirable locations. The general idea of the technique proposed by Roth [35] is to allow two actors migrate independently within a mobile actor system and exchange information about the last location visited and the next location to visit. But this technique has lots of drawbacks since the costs of remote messaging is too high and the actors can be killed by malicious locations.

2.7.4 Protecting Locations

Malicious actors can attack both other actors and locations they migrate to. In the previous subsection, the main mechanisms have been described that protect actors from attacks. In this subsection, the main techniques are going to be described that protect locations.

This problem is almost solved in the large parts of the literature, e.g., Java as a programming language and execution environment already provides several techniques that can be used to protect the underlying location from several types of attacks carried out by malicious actors [5].

2.7.4.1 Security through Programming Language

If Java is compared to other commonly used programming languages like C or C++, it is strictly typed and has a pointer model that does not allow illegal type casting or pointer arithmetics [5]. For example the programmer does not have opportunity at Java to use an insecure class method like textttstrcpy, which allows DoS-Attacks through overwriting stack segment of the code.[14]

SALSA [44] as an actor oriented programming language, uses the object-oriented concepts of Java, like encapsulation, inheritance, and polymorphism, SALSA programs may be executed in heterogeneous distributed environments. During the compilation process SALSA code is compiled to Java, which allows SALSA programs to employ components of the Java class library [12]. Therefore all of the

[14]Interested reader can read more about Java security in [32].

security aspects of Java can be easily applied to SALSA. At the same time SALSA introduces a discipline of concurrent programming using actors over the Internet. In this subsection, we are going to describe the major security properties of Java.

Code Signing

Code signing is a technique used to verify the integrity of mobile code up to a level. The programmer or the owner of the actor signs a hash value of the actor with his private key and sends the signature with the actor to the destination location. The destination location can then verify the integrity of the code by decrypting the signature. The drawback of this method is that if the actor's owner signs the code, then he cannot know if the code is really malicious or not. PCC, which is going to be described in the following section, proposes a solution to this problem.

Byte-Code Verification

When a Java class is loaded to a virtual machine, it is verified on the level of byte code [25]. Among other things, *byte code verification* includes ensuring that only valid instruction codes are used, no final method is overwritten, local variables are not accessed until they have been defined with the appropriate value, and control flow instructions target the beginning of an instruction [5].

The drawback of byte code verification is being very time consuming. Therefore, Amme et al. [3] suggest testing type safety through looking at the structure of the code representation.[15]

Sand-Boxing

The actor authorization process defines which permissions an actor should have on a specific location. *Sand boxing* is a further security check that can be performed after a code has passed he verification and during runtime. This technique includes the following elements:

- Each class is loaded from a specific code source, and if the class has a signature then the code includes that, too.

[15]Interested reader can read more about Java bytecode and the verification in [25].

- A permission is a specific action that a code is allowed to perform. In Java permissions have a type (a class name), a name and an action. For example in java, to access all files in a directory `/foo/bar`, the code must have the `java.io.FilePermission` with the directory as name and read as action string.
- A protection domain is an association of source code and a set of permissions.
- Policy files are used to define protection domains. They can be plain text files in which it is defined which permissions a code loaded from some URL will have.
- Key-stores contain certificates that can be used to verify signed code.

The drawback of sand boxing concept is that assigning permissions to protection domains is done statically per default: that is, it is not possible to withdraw a permission for an actor once it has been given.

Integrity Checkers

Buffer overflows are the leading cause of software vulnerability. The common way for the attacker to overwrite values stored on the stack is to use a buffer overflow, where large inputs are used to cause more data to be written to an area of memory than space has been allocated. Cowan et al. [10] propose StackGuard against stack smash attacks resulting from buffer overflows. To detect corrupted control information in procedure activation records, StackGuard adds a location that it calls a *canary*[16] to the stack layout to hold a special guard value.

The goal of StackGuard is to do integrity checking on activation records, with sufficient precision and timeliness that a program will never reference corrupted control information in an activation record, which is written to once on entry to a function, and read from once on exit from a function.

[16]Before the advent of chemical analyzers, miners used canaries to warn them of the presence of dangerous gases: when the canary stopped chirping (or, more likely, dropped dead), it was time to leave the mine in a hurry.

A malicious actor has the capability to overwrite control information in some frame on the stack via a sequential write operation starting from somewhere lower in memory. Assuming that the attacker does not need to inject code, but can use executable code already in the address space. This is a growing technique in practice, and permits us to focus on the most important part of the attack: overwriting control information, particularly pointers to code, such as return addresses. The attack works if it can rewrite the control information between the time it was written with correct values to be saved and the time it was later read assuming it contained correct values of things to be restored.

By inserting a canary immediately before the control information in each frame on the stack. Any sequential write through memory, such as by a buffer overflow, that tries to rewrite the control information will be forced to also rewrite the canary location. Then the remaining problem is to make the value of the canary something that's hard to spoof. The canary is checked immediately before the control values are restored. The control region is protected by virtue of the fact that the canary is checked before each use of the protected information.

Integrity checkers has not been adopted to work in mobile agents and the main drawback of this approach is that it causes delays in the execution of the actor.

2.7.4.2 Actor Authentication and Authorization

One of the main concerns of protecting locations is how the underlying operating system and hardware can be protected against unauthorized access of actors. In this subsection, we are going to discuss techniques that can be used to authenticate and authorize an actor that has migrated to a location.

Authentication of an actor consists of verification of the actor's identity and the identity of the actors owner and sender. The result of actor authentication is the assignment of privileges to the actor according to its identity.

Proof-Carrying Code

Proof-Carrying code is a mechanism proposed by Necula and Lee [29] by which a host system can determine that it is safe to execute program from a not trusted source. Hence, the authors refer to problems in the field of operating systems; they illustrate PCC in [31] by using an example of a mobile actor that visits several online stores.

PCC is a technique by which the host has a *safety policy* that guaranties safe behavior of the code, and the *code producer* creates a formal safety proof for the not trusted code, that acknowledge to the code's adherence to the safety rules. And when the host receives the mobile code, it uses a simple and fast proof validator to check, that the proof is valid and hence the foreign code is safe to execute [30].

By using PCC the responsibility of ensuring security is shifted to the code producer. Furthermore, PCC programs are *tamperproof* in the sense that any modification will either result in a proof that is no longer valid or one that does not correspond to the enclosed program. In both cases the program will be rejected. And in comparison to cryptography, no trusted third parties are required because PCC is checking intrinsic properties of the code and not its origin [30].

Code consumer receives the PCC, validates the proof that is a part of the PCC, and loads the code. This check must be done only once, even if the code is going to be executed several times. Any attempt to change the code or the proof can be detected and the code can be rejected.

Until now, PCC has not been adopted to work in Java-based environments and mobile agents [5]. The main drawback of PCC is the size and complexity of the proof. Experiments showed that it can become even larger than the code that it has to prove [5]. Besides that it is very difficult to generate such formal proofs in an automated and efficient way [6].

Path History

Considering the problem of deciding on the level of trust of an actor, two main criteria play a big role: actor's identity and which other locations it has visited until now. *Path history* [33] supplements an actor about the locations it has visited, so that when it migrates to a new location, the list of the locations it has visited before can be verified. In addition the location will ask further questions to the actor in order to decide on the level of trust.

Ordille [33] suggests two methods:

- Each location adds itself to the path history of the actor and signs the complete path. The destination location verifies the signature, and then determines whether it can trust every location in the list, whether they have forwarded the actor properly and whether it has authenticated its immediate predecessor.
- Each location signs a forward, showing to which location the actor is going to migrate. In order to prevent tempering the forwarding location signs not only its forward but also the previous ones, and the destination location authenticates each location on the path.

The drawback of this method is the increasing size of the path history and in he same manner, the verification time.

State Appraisal

When an actor is traveling, it visits several locations and it may be attacked by a single location or several locations. *State appraisal* [15] can be used to assure actor owners and the locations that an actor's state has not been tampered, such that both actor's owner and location are protected. It is based on the idea that the illicit modifications of an actor can be predicted, described, and later verified. Therefore, several state appraisal functions are defined before an actor migrates from its home location, and after a location receives the actor, it verifies the state appraisal function. This results in giving execution privileges to the actor or rejecting it.

State appraisal is stated after actor authentication and before actor authorization. There are two types of state application functions: one developed by the actor's owner and sent with the owner's signature as a part of the actor's code; and one developed by the sender. As the first function determines the set of permissions that the actor's owner would like to see granted to the actor, the second function consists of the set of permits, which is a subset of the first one.

The drawbacks of state appraisal is that a malicious location can also see the appraisal function and modify the actor in such a way that it will not be detected in the following locations it visits.

History-Based Access Control

The goal of *history-based access control* is to maintain a selective history of access requested by a mobile code and to use this information to decide between trusted and not trusted codes. This approach, developed by Edjlali et al. [14], helps the location determine the privileges of a mobile code according to its behavior in its runtime.

With static privileges, as in state appraisal or path history, it is possible to define that the code has either both of the privileges or neither of them. The authors propose that it is worthwhile to define mutual exclusive privileges so that the code is allowed to first open a file for reading but it is allowed to open a network connection later, where the file might be sent to a remote host. The concept is based on unique identifiers for programs, which are computed using hash functions over the entire code. This prevents the applicability of the approach for programs that use dynamic code loading during runtime.

History-based access control is suggested for Java applet and used to be extended for actors [5]. In contrast to Java applet, which can open network connection only to remote hosts in case they want to steal data, actors simply carry the file as part of their own code. Therefore, some kind of firewall is needed to examine the actor before it leaves the current location.

2.7.4.3 Actors Execution

After authentication and authorization an actor is executed. Each actor should be executed in a separate environment, where each access to host resources is verified against the actor's permissions [5].

The sand boxing concept, described before, also includes that each actor gets a separate class loader and thread group [5]. These are necessary to distinguish between the actors. Class loader creates an individual name space for each actor and helps the location to make sure that this actor's classes are removed after it has left the location. Each actor and the actors it creates belong to the same thread group. Because if all the actors in a location belongs to one group then a malicious actor can enlist all of the actors of this thread group and lock them, which would be a DoS-Attack against the actors.

In order to stop malicious actors from unauthorized resource consumption Czajkowski and von Eicken [11] proposed a technique that relies on native code implementation, code rewriting to track memory usage, and CPU and network bandwidth consumption. Furthermore Villazon and Binder [46] proposed another solution that relies completely on Java byte code rewriting, making it suitable for mobile agent toolkits. It creates a meta-actor for each actor that is currently residing in the location. Braun [5] suggests applying this to CPU, memory, and network control. Reification of network bandwidth consists of redirecting calls to the components that provide network services to the meta-actor, which itself then calls the network service. Reification of memory on the other hand, consists of object creations and disposal. Finally, ratification of CPU usage is done by analyzing the actor's code and inserting accounting instruction at selected points in the control flow. The drawback of this approach is because of byte code rewriting process and additional inspection, increased execution times.

Load Balancing

Internet is constantly growing as a ubiquitous platform for high-performance distributed computing. For actors as mobile codes using heterogeneous resources, between these resources a load balancing must be realized, so that not only several resources are overloaded, but the total consumption is equally distributed. An overload of resources may end up with delays in the system or in the worst case with DoS. Actor satisfaction is a measure of an actor's ability to process and send messages, which will be used in describing the success of different load balancing techniques.

Distributed Join-Calculus, proposed by Fournet et al. [18] realizes load balancing in an environment of mobile actors. This calculus allows expressing actors moving between physical sites. Individually a location resides on a physical site, and contains a group of processes. Therefore a location can be automatically moved to another site. Actors may have sub-actors, which move together with the actor. For this reason Fournet et al. organized the actors in a tree.

Desell et al. [12] propose a framework that wraps computation into *autonomous* actors, which freely roam over the network to find their target execution environment. This framework consists besides other features a middleware infrastructure for autonomous reconfiguration and load balancing, which is completely transparent to application programmers. This IOS middleware triggers actor migration based on profiling resources in a completely decentralized manner. Furthermore this infrastructure allows continuously balancing the load, while the resources change as nodes are dynamically added and removed.

In order to balance computational load, Desell et al. suggest three types of random work stealing, which vary by the amount of profiling done and the complexity of the decision agents. The simplest decision agent decides according to the load of the individual theaters and actors, while more complex agents consider additional factors such as network and actor topology. A theater joins autonomous network by registering with a peer server and receiving addresses of other peers in the network.

- Load-Sensitive Random Stealing (RS): based on randomly propagating a random steal packet over the network
- Actor Topology Sensitive Random Stealing (ARS): extends RS with additional profiling information
- Network Topology Sensitive Random Stealing (NRS): in addition to resource availability of ARS, NRS takes the topology of the network into consideration and classifies its neighbors into groups (local, regional, national, and international).

Soft-Fault Tolerance

The concept of soft-fault tolerance of software contains aspects like: separate control and forwarding; modular processes that can be restarted independently; processes protected in own memory space; and individual process watch dogs.

Fournet et al. [18] suggest *Distributed Join-Calculus*, which expresses mobile agents moving between physical sites. It expresses remote execution and dynamic loading of remote resources, and with locality and static scooping rules, is an asynchronous variant of Miller's π-Calculus [26]. But is still has the same expressive power as the π-Calculus. Distributed Join-Calculus furthermore, provides a simple model of failures. According to this model, crash of a physical site causes a permanent failures for all actors that are executed in that site, so the failure of an actor can be detected from any other running actor, allowing error recovery.

Desell [12] suggests using a *round-robin strategy* to disperse actors to all their available neighbors in the case of soft termination of a location. The goal of this to complete the termination process as soon as possible before the node failure.

Stoica et al., suggest in *Chord Model* [40], which is a decentralized lookup service that stores key/value pairs for distributed networks. Each Chord node is identified by an `m`-bit identifier and each node stores the key identifiers in the system closest to the nodes identifier. Each node maintains a `m`-entry routing table that allows it to look up keys efficiently. The lookup service functions despite network partitions and node failures. Stoica et al., provides a "best effort" availability grantee based on access of at least one of the reachable replica nodes. Chord is furthermore, incrementally scalable, with insertion and lookup costs scaling logarithmically with the number of Chord nodes. Although Chord protocol and system has not been used on actors until now, the experiments show that it is a valuable component for many decentralized, larga-scale distributed applications.

The drawback of these methods is that they are application dependent, so that they do not tolerate hard node failures at the middleware level. Besides that it uses replication or information propagation in order to fulfill its function, and because of that the amount of information that has to be protected increases.

3. Electronic Marketplace Model

In Chapter 2, some major mechanisms are discussed that can be used for the security of actors. In this chapter, a marketplace model for actors will be described, which we will use in order to make the actor security scenarios in Chapter 4 more concrete.

In Section 3.1, only the basic elements of the model will be described. It will be followed, in Section 3.2, by the description of the online marketplace architecture, which separates application level and middleware levels. We will conclude this chapter with the vulnerabilities of our marketplace model.

3.1 Elements of the Marketplace Model

The marketplace consists of two types of actors; *manager actors* that are responsible of management of different features of the marketplace, and *seller* and *buyer actors* that represent the product owners and the customers of the real world, respectively. (See Figure 3.1)

Buyer/seller actors migrate from one location to the next, take part in the auctions, and try to buy/sell the items in their item list with the best price. Auctions take place in online stores. In each store different auctions take place and after the auction is over the buyer actors are sent to other locations. Actors join the Marketplace every time by connecting the Coordination Model Manager (CMM) and the Seller Manager (SM). The seller and buyer actors come together in auctions. After every auction the results are reported to the Transaction Manager (TM). TM is the storing unit of the marketplace. All of the transaction results flow to its database.

Each seller actor contains a list of items on sale and for each item their properties like product description, available quantity, an initial minimum bidding price, and a minimum bid increment. Similarly, each buyer actor contains a list of items to buy, descriptions, desired quality of the items and maximum price to pay for each item, and a common budget.

In every store (see Figure 3.2) there is a Store Manager responsible for match-

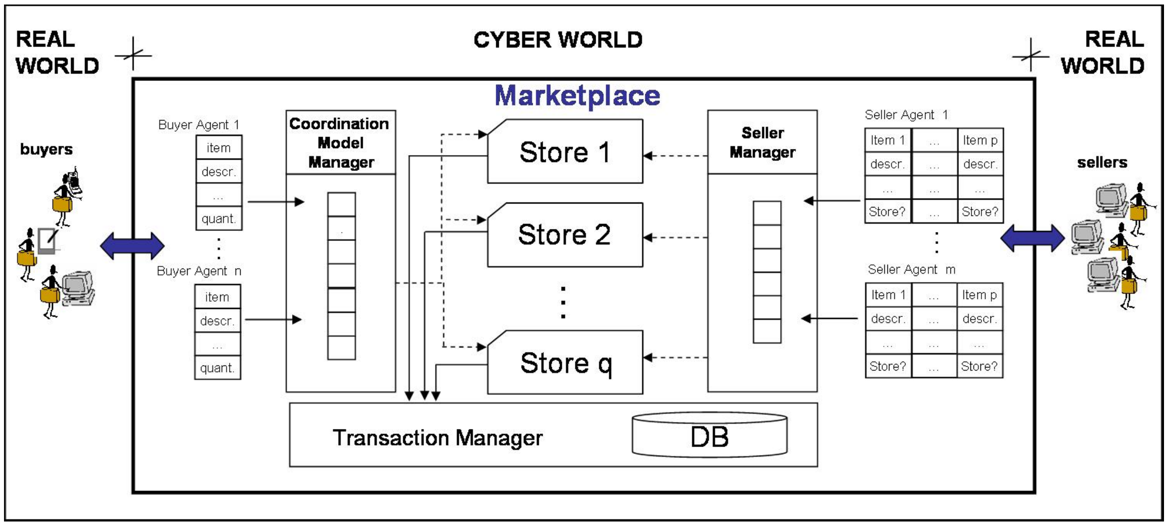


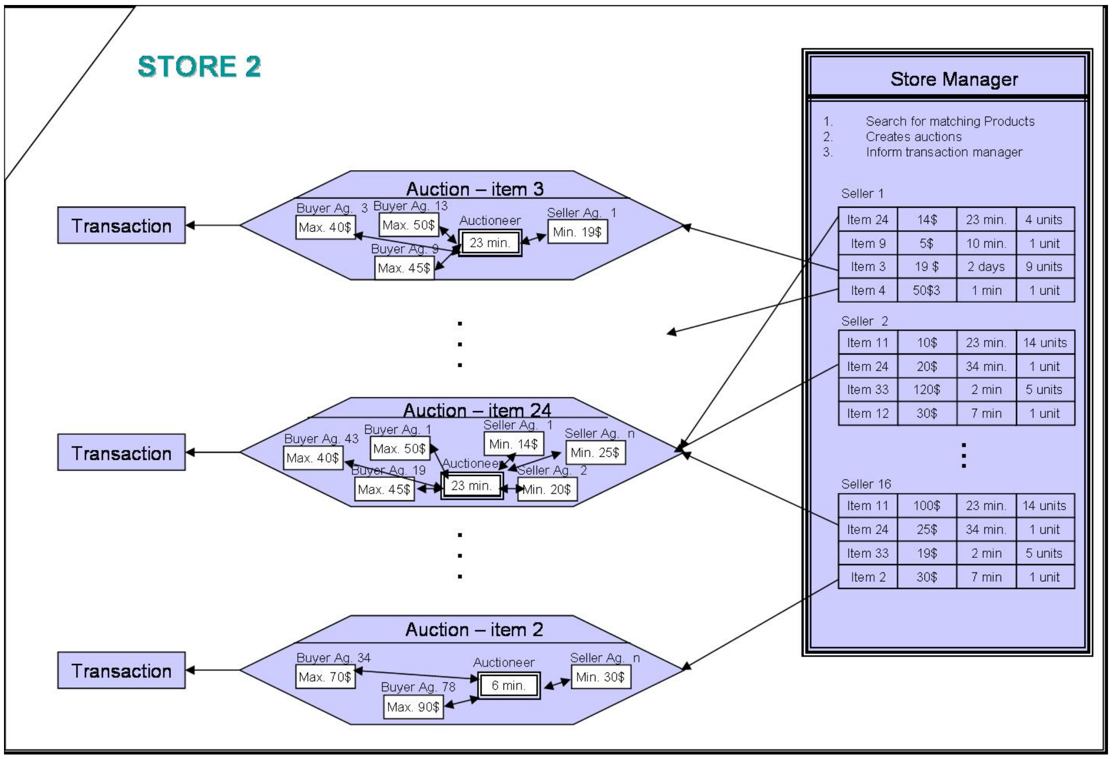
Figure 3.1: Architecture of the Marketplace

ing the seller and the buyer actors and creation of auctions. One product of a buyer is only represented in one store in order to prevent inconsistencies. At the end of an auction the information in the store manager is updated and the agents are sent to another location, which can be another auction or the home location of the actor depending on whether the actor has fulfilled its duties or not.

The electronic marketplace is a system that enables dynamically choosing the number of actors, which impacts the number of actor migrations, and the number of messages sent, for a given buyer. The goal is to increase the possibility of the actors to get all the products desired by the buyer within the given budget, and to accomplish this task in the given time period. Furthermore, from the perspective of the sellers and auctioneers, the system enables to dynamically choose the most suitable auction type, with the goal of optimizing their profitability.

3.2 Customizable Middleware for Actor Communication

middleware refers to software technology that enables the modular connection of distributed software [49]. The role of middleware is to abstract over the low level protocols required to implement the policies, which represent the roles for component interactions. The system itself is responsible for handling low-level details such as sending messages over the network, providing for synchronization, and guaranties



1. Search for matching Products
2. Creates auctions
3. Inform transaction manager

Seller 1

Item 24	14$	23 min.	4 units
Item 9	5$	10 min.	1 unit
Item 3	19 $	2 days	9 units
Item 4	50$3	1 min	1 unit

Seller 2

Item 11	10$	23 min.	14 units
Item 24	20$	34 min.	1 unit
Item 33	120$	2 min	5 units
Item 12	30$	7 min	1 unit

Seller 16

Item 11	100$	23 min.	14 units
Item 24	25$	34 min.	1 unit
Item 33	19$	2 min	5 units
Item 2	30$	7 min	1 unit

Figure 3.2: Architecture of a store in the marketplace.

reliability. And through middleware application interactions appear to be local. The applications need not be explicitly aware of the composition and distribution of hardware.

In distributed systems different autonomous computing elements interact over a shared network. The asynchronous nature of the actor model makes the application development difficult. Actor model requires policies expressing basic requirements for service, like for the entrance of a seller actor to an auction, realizing the payment or product delivery. Factors such as the presence of faulty hardware or insecure networks, which affect the security of a system very strongly, are most of the time orthogonal to each other. These factors are orthogonal in the sense that they only affect the implementation of the protocol and do not alter the basic implementation of the service.

A clear separation of protocols from application code provides an abstraction limit in the spirit of abstract data types, which increases the flexibility, maintainability, and portability of distributed code. Hence, policies, which define the rules by which components interact, used to be specified separated from protocols, which define the mechanisms by which policies are implemented.

Orthogonal design constraints, furthermore, force components to stick on to multiple policies. A framework for protocol composition preserves the modularity of such policies so that their implementations may be safely composed. For example, an existing encryption protocol may be safely composed with an existing primary backup protocol in order to satisfy both secrecy and fault-tolerance policies [4].

Furthermore the middleware may be extended with automatic load balancing component, which lets auction stores migrate autonomously by profiling resource usage on sites.

The marketplace model will use a middleware in order to realize the features mentioned above. Since this thesis concentrates on the security aspect of actors that are used in e-commerce applications, especially online negotiations, the middleware concept will not be further discussed here.

3.3 Actor Scenario

In this section, we will define a possible scenario, where a mobile user, Bob, is willing to buy a notebook. He knows the main features of the laptop he wants to buy and maximal amount of money he wants to pay for it. He goes to the Marketplace and logs in by using his login and password. If the server verifies these, then it opens a site where Bob can enter the specifications of the notebook he wants to buy. The CMM generates an actor and sends it to a store where notebooks with the matching specifications are sold. The store manager looks at the ongoing auctions and depending on if there is a matching auction or not, whether lets Bob's actor join the auction or creates a new auction. The actor bids on behalf of Bob during the auction. If it wins the auction then returns the information to the TM, which updates the database and realizes the money transaction. At the same time generates a message and sends it to Bob. The message involves the information about the seller, bidding result and the exact specifications of the notebook. If the actor does not win the auction, then it is sent to the next auction.

This scenario may be used also for the *reverse auctions* [28], where multiple seller actors try to make the lowest bid, in order to sell the notebook to a single buyer actor.

When an actor is active it has a lot of weaknesses, which makes its code and the data it carries open to attacks. The marketplace has also a lot of sensitive private information about the owners of the actors. These vulnerabilities of the actor model will be discussed in the next section.

3.4 Vulnerabilities of the Marketplace Model

If we regard the marketplace model components roughly, we can group them as actors and locations. Both actor and locations can have a malicious character.

Malicious actors may attack a location for example by using its resources like memory, CPU cycles or network bandwidth, so the location is not able to provide its usual service to the other actors. These kinds of attacks are known as DoS-attacks in the area of information security. A malicious actor can also attack other actors that are currently resident at the same location. Some of the possible attacks result from the programming language, other are possible because of the hosting location, communication structure and services provided. A malicious actor may, for example, masquerade himself as some other actor by changing its address. This way it may gain sensitive information stored in the databases of the middleware or use services on behalf of others.

Malicious locations may try to attack the actors that are currently resident on it or other actors by attacking the communication links. The general problem, what makes an actor vulnerable is, that an actor must disclose its information about code and data if it wants to be executed.

The goal of our actor model is to have mechanism that realize the entrance-security and mechanisms that protects the code and the data of the actor during its transport between auctions and locations.

4. Model Applicability to Secure e-Marketplace

In this chapter, we will give some examples of how the mechanisms described in Chapter 2 can be used in order to achieve a secure actor based e-commerce application.

Figure 4.1 summarizes which security requirements of information systems can be fulfilled by using the methods we have described in Chapter 2. The properties of the different actor models are going to be discussed in Chapter 4 with more details and some examples are going to be given.

Actor Models / Sec. Requirements	Secure Mobile Actors [Toll 2003]	Transactors [Field 2004]	Hierarchical Coordination [Varela 1999]	Mobile Ambients [Cardelli 1998]
Authentication	- universal names - ACL		- director agent	-in n, out n -unique names
Confidentiality	- ACL		-coordination constraints - messenger agent	- open n
Integrity	-ACL	-stabilize, checkpoint - persistent storage		- open n
Accountability		- global consistency ditr. state - Rollback - persistent storage - state immutability		
Availability			- load balancing	
Anonymity & Privacy				

Table 4.1: Actor models and security requirements

4.1 Security Aspects of Secure Mobile Actors

The secure mobile actor model extends the actor model with locations, mobility, universal names for actors and locations, and resources acccess control.

The universal names, as unique identifiers of actors and locations, allow actors and locations to identify themselves when they migrate or communicate. The names of the trusted locations and actors are saved in form of ACLs, in every actor and location. Secure actors also restrict communication and migration using ACLs, so that no unprivileged actor can gain access to a not authorized resource and respectively no location can indirectly query an actor without being in its ACL.

Suppose that a seller actor wants to migrate to the marketplace in order to sell the products in its item list. The SM looks at its ACL, and dependently sends the actor to a store or rejects the actor. Rejection does not mean that this actor will not be allowed to enter the marketplace any more, but that the actor has to prove, using mechanisms like PCC, who it is and that it will not harm locations or other actors in the system. Since the actor names are universal, as far as a malicious actor does not masquerade itself as an other actor, who is already in the ACL of the SM, they can be used for authentication.

Since no unauthorized actor can enter the system, the information that actors in the system carries or locations have, stays confidential for the not authorized access. Furthermore, since not authorized actors/locations cannot access the data they also cannot manipulate it.

The secure actor model furthermore has some reliability property; since it guaranties that the destination actor will receive the messages sent. But it does not guarantee that the message will not be changed on the way, so this property can be extended with some other integrity mechanisms, like digital signatures or MACs.

Suppose that a buyer actor bidding for a product calculates the MAC of the bid it is going to send to the auctioneer actor and sends both of them in one message. The auctioneer actor can then verify the MAC in order to be sure that it has not been modified, and then accepts it. So that we can make sure that both the message has not been changed and reaches the auctioneer.

4.2 Security Aspects of Transactors

Transactors extend the actor model by explicitly modeling node failures, network failures, persistent storage, and state immutability. It reliably maintains globally constant distributed states using stabilize, checkpoint and rollback operations.

Stabilization of a transactor is a commitment not to modify its internal state until subsequent checkpoint is performed or until another peer actor causes it to rollback due to a semantic inconsistency. Checkpoint, as the next commitment that stores globally consistent states of a transactor persistently, is at the same time a consistency guarantee. Both stabilization and checkpointing fulfill data integrity requirements, since in case of a not authorized manipulation the not manipulated states are available in the system.

Suppose that the bidding process of an auction is over and seller actor *s* is going to sell an item to buyer actor *b* for a decided price. The auction is not over till the money transfer is over. So states of all of the actors that took place in the auction are stabilized but not yet committed. Auctioneer actor notifies the TM from the result of the auction. TM sends a message to the bank server of the buyer actor, asking it to transfer to the price of the product to seller's account. If the bank server refuses to the transfer the amount or does not answer in a definite time period then TM notifies the auctioneer actor, actor `s` and actor `b` from the result. All the actors do a rollback and the auction starts from last committed state, without the buyer `s`. If the bank server sends a message saying that the transfer has succeeded, then TM notifies the auctioneer actor, actor `s` and actor `b`, so that they commit their states. At the same time the auctioneer actor sends a broadcast to all of the actors about the success of the transaction, and they commit their states.

Furthermore transactors allow fault-tolerance through globally consistent distributed states. In case of a node or network failure, a rollback operation brings a transactor back to its previously checkpointed, immutable state. Through these, transactors may not allow all the time the authorized actors to reach the system resources but stops the actor data from getting completely lost because of unauthorized manipulation of the system. So we may say that transactors partially fulfill the accountability requirement.

4.3 Security Aspects of Hierarchical Model

Hierarchical model extend actor model with concepts of casts and directors. Actors are mapped to the director actors, which coordinate the communication in that cast. In this model messengers coordinate remote cast-to-cast communication. Casts serve as abstraction units for naming, migrating, synchronization and load balancing.

Since the director is responsible of the communication in a cast, we can talk about a net of trust. Where the actors do not need to authenticate themselves against other actors in the same cast. In case an uncoordinated actor wants to send a message to one of the actors in the cast, it uses a messenger that has to authenticate itself only to the director, which simplifies the authentication process.

Suppose a new actor wants to join an auction. All of the seller and buyer actors that are bidding in the auction belong to one cast. The auctioneer agent plays in that case the role of a director actor. It authenticates itself using some authentication mechanism and after that it is authorized to send and receive messages directly from other actors that are taking part in the auction. Besides that all of the actors in a store can be grouped into a cast. So that if an actor wants to leave auction `p` and want to migrate to auction `q` in the same store, then the director of `p` communicates with `q` through the director of the store cast, without the need of authentication, and notifies it from the situation. In the next step, director actor q accepts the actor's migration without an authentication.

The hierarchical architecture of the model and directories checking the coordination constrains provide confidentiality in the casts. According to that an actor can only receive a message, when the coordination constraint for such message receipt are satisfied. For the remote messaging, since the messenger does not have to know about the information it is carrying, confidentiality of this information can be gained by using cryptographic mechanisms, like encrypting the information with the public key of the destination actor. This property of the hierarchical model furthermore, can be used to keep actor replicas consistent, in order to cope with malicious location attacks.

Supposing that an auctioneer actor wants to send the results of an auction to the TM. It encrypts the information about the results using the public key of TM, and sends it through the messenger actor to TM. Since the remote messenger is just a cipher text, malicious actors or location can not know from whom it is, and although they may try to sniff it, they cannot access the information. Only the real destination, TM, may decrypt the message using its private key.

Hierarchical model also satisfies availability up to a level because of its load balancing property.

Let a new buyer actor joins the Marketplace, with some specific items to sell. Instead of sending a broadcast to all buyer actors in the marketplace, SM sends a message to all of the store-directors and store directors then send only to the buyer actors that are interested in buying these items. So that the network will not be overloaded and the availability of the resources for authorized actors will not be diminished.

4.4 Security Aspects of Mobile Ambients

In the mobile ambients model, an actor has to go through a three-phase authentication-authorization process based on unique actor and location names. They first have to get a permission to exit their current location, and then they have to get permission to enter the destination location. Here we can talk of a multiple level of authentication depending on the route to take. After entering the destination location, the actor furthermore has to get authorization for entering the protected information of some server. Therefore only authorized actors can access the confidential data and change it.

Besides mobile ambients allowing an authentication through names, they simplify the authentication process through a hierarchical structure as in the hierarchical model. In our Marketplace scenario, a buyer/seller can be represented with more than one actor depending on the coordination model [28]. Extending the cast concept of the hierarchical coordination model, in Mobile Ambients, actors may belong to more then one ambient at the same time. Therefore, besides grouping actors depending on the auctions, at the same time each actor representing a single buyer/seller can be grouped to an ambient. This property allows them to communicate with each other without authenticating themselves every time, as they are bidding in different auctions.

4.5 Security Integration of Model Properties

Combining the above-mentioned properties of an actor model is a difficult issue, and even if we manage it, the results are not going to fulfill all of the security requirements. Therefore, in order to fill out the security holes that remain after combining the actor models, we combine them with the security instruments described in Section 2.7. Table 4.2 shows a comparison of these instruments according to the level of fulfilling the security requirements. In the table *weak* refers to a property that is easy to come over or hard to realize at an actor environment; and strong refers to a property that can be efficiently used on actors or locations.

Concerning the results of Table 4.2, costs of each instrument, and the requirements of our marketplace model; evolving the actor models with the security properties of PCC will allow us to reach a optimal level of security at the state of the art in actor systems in electronic auctions. Bacause the actor models in combination already fulfill most of the security requirements, and PCC can back them by fulfilling the security requirements: availability, anonymity and privacy.

Assuming that an actor wants to enter the Marketplace, if it is a new actor, Marketplace will send a message to the actors owner and programmer, asking for a proof, which will ensure that the execution of the actor will not affect the availability of the marketplace in a negative way. This proof should demonstrate that the actor is not aimed to make any attacks that may stop or even slow down the access of the authorized actor from consuming the host resources. Next time a new actor wants to migrate to the marketplace it will bring the proof with. The Marketplace will check the proof and only after its acceptance allow the actor to enter the marketplace.

Since the proof can demonstrate anything about the execution of the actor, and the behavior of a location, a proof that the marketplace will demonstrate to the actor owners may include arguments about the anonymity and privacy policy of the marketplace.

				Security requirements					
				authentication	integrity	confidentiality	accountability	availability	anonymity & privacy
security instruments	protecting actors	preventing attacks	encryption function		weak	weak			
			time-limited black-boxes	strong	strong	strong			
			environmental key generation	strong	strong	weak			
		detecting attacks	replication of location		weak				
			replication of actors		weak				
			actors as black-boxes	strong	strong	strong			
			cryptographic techniques	strong	strong	strong			
			execution tracing	weak	strong		strong		
			detecting itinararay manipulation		weak		weak		
	protecting locations	programming language	code signing		weak				
			byte-code verification		weak				
			sand-boxing	weak	weak	weak		weak	weak
			integrity checkers		weak			weak	
		actor authentication and authorization	PCC	strong	strong		strong	strong	strong
			path history	weak			strong		
			state appraisal	weak	weak				
			history-based assecc	strong					
		actor execution	load balancing					strong	
			soft fault-tolerance					weak	

Table 4.2: Security requirements that the security instruments fulfill.

5. Conclusion and Future Directions

5.1 Contribution of this Thesis

The goal of this thesis was to analyze different actor models and their extensions from the point of view of information security in order to show the security potentials of these models. To achieve its goal this thesis first described different actor models and then in order to show how they can be used in secure electronic commerce evaluated on a marketplace model.

In Section 3, we have described a marketplace model, which is a middleware framework connecting distributed resources for the online auctions. The model consists of three control mechanisms: CMM, SM and TM. CMM and SM components are responsible of the managing the access of seller and buyer actors to the marketplace sending them to the convenient stores. TM is the database component that stores the transaction data for the system consistency. Besides these main control mechanisms of the marketplace, in each store there are store managers in charge of creating auctions and sending the seller and buyer agents to auctions where they can bid for the items they want to buy.

Mobile agent technologies provides potential benefits to applications, but they also pose security threats to those applications. These threats not only come as malicious agents, but also in the form of malicious hosts.

Using some application scenarios on our marketplace model, we have discussed in Chapter 4 which actor model satisfies which information security requirements. According to these secure actor model, hierarchical coordination and mobile ambients are in charge of fulfilling authentication and information confidentiality; transactor model consists components that realize data integrity and accountability; and because of its load balancing property hierarchical model can prevent some attacks against system availability. But none of these actor models prevents attacks against anonymity and privacy. Therefore, in Section 4.5, we have proposed a security scenario for the marketplace model that combines actor models with the security instruments, described in Section 2.7, and fulfills all of the security requirements.

5.2 Future Directions

The daily lifes of digital device users are getting more and more mobile every day. So a possible implicational extension of the secure marketplace is integrating modules that use different specifications of different end-devices, like Global Positioning System (GPS) united mobile Telephones or PDAs. Through such features mobile device users will be able to set an actor every moment of their life. And an actor that is in charge of finding the best bargain, for example for a laptop, can discover that his owner is getting close to a physical store, which is also represented in the cyber world and has a good offer. The actor then informs his owner about an actual bargain result list and where the shop is. So the buyer can go to the store and take a look at the notebook, and if the store has a better offer then get the notebook directly from the store.

But such an extension brings also other security problems with itself. Because the notification message that the actor sends will be transported over unsecured information channels. And although the message is transferred encrypted until the base station, today's security technology is not able to protect the message transfer from the base station to the mobile device. So it can be attacked during the last segment.

Since the platform involves grid computing properties many actors from many different countries will take part in auctions. And every country has its own trade but also information and data security legislations. So another possible extension may concentrate on legislative aspects and involving a module that deals with different legislations in different countries.

Besides that only some of the security models and instruments are backed with computational semantics, and although it is probably not possible to evolve a computational semantic that consists all of them, semantics that covers most of these instruments can be developed.

5.3 Conclusion

Security and agent-enabled electronic marketplaces are two research fields that are gaining more interest every day. Security and intelligence in processing transparent information flow in an electronic marketplace can help increase the efficacy for its participants and reduce user's cognitive load [39].

Although in the thesis a security strategy for the marketplace is defined, and according to many scenarios the secure actor communication with each other and their environment has been shown, in the praxis it is very challenging to combine all of the aspects. Further studies on this field are critical to the future of secure electronic commerce.

REFERENCES

[1] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[2] Gul Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, 1997.

[3] Wolfram Amme, Niall Dalton, Jeffery von Ronne, and Michael Franz. Safetsa: A type safe and referentially secure mobile-code representation based on static single assignment form. In *SIGPLAN '01 Conference on Programming Language Design and Implementation*, pages 137–147, 2001.

[4] Mark Astley, Daniel C. Sturman, and Gul Agha:. Customizable middleware for modular distributed software. *Commun. ACM*, 44(5):99–107, 2001.

[5] Peter Barun and Wilhelm Rossak. *Mobile Agents: Basic Concepts, Mobile Models and the Tracy Toolkit*. Morgan Kaufman, dpunkt.verlag, Amsterdam, Holland, 2005.

[6] Niklas Borselius. Mobile agent security. *Electronics Communication Engineering Journal*, 14(5):211–218, October 2002.

[7] Christian J. Callsen and Gul Agha. Open heterogeneous computing in ActorSpace. *Journal of Parallel and Distributed Computing*, 21(3):289–300, 1994.

[8] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In *FoSSaCS '98: Proceedings of the First International Conference on Foundations of Software Science and Computation Structure*, pages 140–155, London, UK, 1998. Springer-Verlag.

[9] David M. Chess, Colin G. Harrison, and Aaron Kershenbaum. Mobile agents: Are they a good idea? In *Mobile Object Systems*, pages 25–45, 1996.

[10] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, January 1998.

[11] Grzegorz Czajkowski and Thorsten von Eicken. Jres: a resource accounting interface for java. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 21–35, New York, NY, USA, 1998. ACM Press.

[12] Travis Desell, Kaoutar El Maghraoui, and Carlos Varela. Load balancing of autonomous actors over dynamic networks. In *Proceedings of the Hawaii International Conference on System Sciences, HICSS-37 Software Technology Track*, pages 1–10, January 2004.

[13] Klaudia Eckert. *IT- Sicherheit: Konzepte, Verfahren, Protokolle* . Oldenburg Verlag, Amsterdam, Holland, 2004.

[14] Guy Edjlali, Anurag Acharya, and Vipin Chaudhary. History-based access control for mobile code. In *CCS '98: Proceedings of the 5th ACM Conference on Computer and Communications Security*, pages 38–48, New York, NY, USA, 1998. ACM Press.

[15] William M. Farmer, Joshua D. Guttman, and Vipin Swarup. Security for mobile agents: Authentication and state appraisal. In *Proceedings of the Fourth European Symposium on Research in Computer Security*, pages 118–130, Rome, Italy, 1996.

[16] John Field and Carlos Varela. Toward a programming model for building reliable systems with distributed state. In *Proceedings of the 1st International Workshop on Foundations of Coordination Languages and Software Architectures (affiliated with CONCUR)*, Brno, Czech Republic, August 2002. http://www.cs.rpi.edu/˜cvarela/foclasa2002.pdf.

[17] John Field and Carlos A. Varela. Transactors: a programming model for maintaining globally consistent distributed state in unreliable environments. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 195–208, New York, NY, USA, 2005. ACM Press.

[18] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *CONCUR '96: Proceedings of the 7th International Conference on Concurrency Theory*, pages 406–421, London, UK, 1996. Springer-Verlag.

[19] Svend Frölund. *Coordinating Distributed Object: An Actor-Based Approach to Synchronization.* MIT-Press, 1996.

[20] Fritz Hohl. Time limited blackbox security: Protecting mobile agents from malicious hosts. In *Mobile Agents and Security*, pages 92–113, London, UK, 1998. Springer-Verlag.

[21] Fritz Hohl and Kurt Rothermel. A protocol preventing blackbox tests of mobile agents. In *Kommunikation in Verteilten Systemen*, pages 170–181. Springer-Verlag, Berlin Germany, 1999.

[22] Stanislaw Jarecki. Proactive secret sharing and public key cryptosystems. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA; Advisor: Professor Ronald L. Rivest, September 1993.

[23] Neeran Mohan Karnik. *Security in mobile agent systems.* PhD thesis, University of Minnesota; Adviser-Anand R. Tripathi, 1998.

[24] Wooyoung Kim. Thal: An actor system for efficient and scalable concurrent computing. Technical report, University of Illinois at Urbana-Champaign, University of Illinois at Urbana-Champaign, IL, USA, 1997.

[25] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison Wesley, 1997.

[26] Robin Milner. The polyadic pi-calculus: a tutorial. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*, pages 203–246. Springer-Verlag, 1993.

[27] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings 4th Annual IEEE Symp. on Logic in Computer Science, LICS'89, Pacific Grove, CA, USA, 5–8 June 1989*, pages 14–23. IEEE Computer Society Press, Washington, DC, 1989.

[28] Ayse Morali, Leonardo Varela, and Carlos Varela. An electronic marketplace: Agent-based coordination models for online auctions. In *Proceedings of the Thirty-First Latin American Computing Conference (CLEI'05)*, Cali, Colombia, October 2005. Submitted for publication.

[29] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, Paris, France, January 1997.

[30] George C. Necula and Peter Lee. Research on proof-carrying code for untrusted-code security. In *DARPA Workshop on Foundations of Mobile Code Security*, pages 204–204, 1997.

[31] George C. Necula and Peter Lee. Safe, untrusted agents using proof-carrying code. *Lecture Notes in Computer Science*, 1419:61–91, 1998.

[32] Scott Oaks. *Java security.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1998.

[33] Joann J. Ordille. When agents roam, who can you trust? In *First Conference on Emerging Technologies and Applications in Communications (etaCOM)*, Portland, OR, 1996.

[34] James Riordan and Bruce Schneier. Environmental key generation towards clueless agents. In *Mobile Agents and Security*, pages 15–24, London, UK, 1998. Springer-Verlag.

[35] Volker Roth. Secure recording of itineraries through co-operating agents. In *ECOOP '98: Workshop ion on Object-Oriented Technology*, pages 297–298, London, UK, 1998. Springer-Verlag.

[36] Volker Roth and Vania Conan. Encrypting Java archives and its application to mobile agent security. *Lecture Notes in Computer Science*, 1991:229–143, 2001.

[37] Tomas Sander and Christian F. Tschudin. Protecting mobile agents against malicious hosts. *Lecture Notes in Computer Science*, 1419:44–60, 1998.

[38] Fred B. Schneider. Towards fault-tolerant and secure agentry. In Marios Mavronicolas, editor, *Distributed algorithms*, volume 1320 of *Lecture Notes in Computer Science*, pages 1–14, Sept 1997.

[39] Rahul Singh, A. F. Salam, and Lakshmi Iyer. Agents in e-supply chains: Realizing the potential of intelligent infomediary-based e-marketplaces. *Communications of the ACM*, 48:109–115, 2005.

[40] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

[41] Robin D. Toll and Carlos Varela. Mobility and security in worldwide computing. In *Proceedings of the 9th ECOOP Workshop on Mobile Object Systems*, Darmstadt, Germany, 2003. http:// www.cs.rpi.edu/research/ groups/wwc/papers/ecoopws2003Full.pdf.

[42] Carlos Varela. *Worldwide Computing with Universal Actors: Linguistic Abstractions for Naming, Migration, and Coordination.* PhD thesis, University of Illinois at Urbana-Champaign, May 2001. Advisor: Professor Gul A. Agha.

[43] Carlos Varela and Gul Agha. A Hierarchical Model for Coordination of Concurrent Activities. In P. Ciancarini and A. Wolf, editors, *Third International Conference on Coordination Languages and Models*

(COORDINATION '99), LNCS 1594, pages 166–182, Berlin, April 1999. Springer-Verlag. http://osl.cs.uiuc.edu/Papers/Coordination99.ps.

[44] Carlos Varela and Gul Agha. Programming Dynamically Reconfigurable Open Systems with SALSA. In *SIGPLAN Notices. ACM Object Oriented Programming Languages, Systems and Applications (OOPSLA '2001) Intriguing Technology Track Proceedings.*, pages 20–34, December 2001. http://www.cs.rpi.edu/ cvarela/oopsla2001.pdf.

[45] Giovanni Vigna. *Mobile Code Technologies, Paradigms, and Applications.* PhD thesis, Politecnico di Milano, Milano, Italy, February 1998.

[46] Alex Villazón and Walter Binder. Portable resource reification in java-based mobile agent systems. In *MA '01: Proceedings of the 5th International Conference on Mobile Agents*, pages 213–228, London, UK, 2002. Springer-Verlag.

[47] Bennet S. Yee. A sanctuary for mobile agents. In *Secure Internet Programming*, pages 261–273, 1997.

[48] Adam Young and Moti Yung. Sliding encryption: A cryptographic tool for mobile agents. In *FSE '97: Proceedings of the 4th International Workshop on Fast Software Encryption*, pages 230–241, London, UK, 1997. Springer-Verlag.

[49] Apostolos Zarras and Valérie Issarny. A framework for systematic synthesis of transactional middleware. In *IFIP: Proceedings of Middleware'98*, pages 257–272. Chapman-Hall, September, 1998.

EHRENWÖRTLICHE ERKLÄRUNG

Hiermit erkläre ich an Eides statt, da ich die vorliegende Arbeit selbständig nur unter Benutzung der in der Arbeit angegebenen Literatur angefertigt habe.

Ayse Morali
Troy USA, den 29.06.2005