Electronic Notes in Theoretical Computer Science 68 No. 3 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume68.html 18 pages

SecSpaces: a Data-driven Coordination Model for Environments Open to Untrusted Agents ⋆

Nadia Busi, Roberto Gorrieri,
Roberto Lucchi and Gianluigi Zavattaro

Dipartimento di Scienze dell'Informazione, Università di Bologna,
Mura Anteo Zamboni 7, I-40127 Bologna, Italy.
E-mail: {busi, gorrieri, lucchi, zavattar}@cs.unibo.it

Abstract

In this paper we initiate an investigation about security problems which occur when exploiting a Linda-like data driven coordination model in an open environment. In this scenario, there is no guarantee that all the agents accessing the shared tuple space are trusted. Starting from the analysis of the few proposals already available in the literature, we present a novel coordination model which provides mechanisms to manage tuple access control. The first mechanism supports logical partitions of the shared repository: in this way we can restrict the access to tuples inside a partition, simply by limiting the access to the partition itself. The second mechanism consists of adding to the tuples some extra information which exploit asymmetric cryptography in order, e.g., to authenticate the producer of a tuple or to identify its reader/consumer. Finally, we support the possibility to define access control policies based on the kind of operations an agent performs on a tuple, thus discriminating between (destructive) input and (non-destructive) read operations.

1 Introduction

New networking technologies are calling for the definition of models and languages adequate for the design and management of new classes of applications. Innovations are moving towards two directions. On the one hand, the Internet is the candidate environment for supporting the so-called wide area applications. On the other hand, smaller networks of mobile and portable devices, such as mobile ad-hoc networks or peer-to-peer systems, support applications based on agents or components which interact according to a dynamically

⋆ Work partially supported by MEFISTO Progetto "Metodi Formali per la Sicurezza e il Tempo", Microsoft Research Europe and by NAPOLI Progetto "Network Aware Programming: Oggetti, Linguaggi, Implementazioni".

reconfigurable communication structure. In both cases, the challenge is to develop applications without knowing, at design time, the overall structure of the system as well as the entities that will be involved. Such systems are usually referred to as *open systems*.

Coordination models and languages, which advocate a distinct separation between the internal behaviour of the entities and their interaction, represent a promising approach for the development of applications for open systems. For instance, we assist to a renewed interest in data-driven coordination infrastructures originated by Linda [Gel85] as exemplified by recent commercial products, such as JavaSpaces [Sun02] and TSpaces [PSD98], which are two coordination middlewares for distributed Java [GJS96] programming proposed by Sun Microsystem and IBM, respectively.

Both proposals exploit the so-called *generative communication* [Gel85]: a sender communicates with a receiver through a shared tuple space (TS for short), where emitted tuples are collected; the receiver can consume the tuples from the TS; a tuple generated by an agent has an independent existence in the tuple space until it is explicitly withdrawn by a receiver; in fact, after its insertion in TS, a tuple becomes equally accessible to all agents, but it is bound to none. This form of communication is referred to as generative communication because when a datum is produced, it has an existence which is independent of its producer, and it is equally accessible to all agents.

In open systems, new critical aspects come into play such as the need to deal with a hostile environment which may comprise also components which are unknown at design time. Clearly, also untrusted agents may enter the system and, according to the data-driven approach, they can access the repository in order to read/remove data, as well as maliciously produce new data.

In this paper we propose an extension/modification of the tuple space coordination model which supports some form of control in the production, reading and consumption of data. To the best of our knowledge, only languages such as KLAIM [NFP98,NFP97] and SecOS [VBO02] already provide access control mechanisms in data-driven coordination languages.

KLAIM exploits classic access control policies to manage the access among agents and tuple spaces: types are used to indicate the access rights of the agents, i.e., the operations that each agent may perform on each of the available tuple space. In open systems, especially in those with a high level of dynamicity, the managing of these information may be a critical task, mainly because the system should support a rapid and sometimes uncontrolled evolution of the agents community. More precisely, new agents may frequently enter the system, as well as old agents may rapidly exit, in an uncontrolled manner. Moreover, in some applications it could be useful to have a finer grained control, e.g., at the level of tuples and not at the level of spaces. For example, we may want to ensure that an agent cannot read tuples with a private contents, but it can read all the other tuples.

SecOS follows a quite different approach. The access rights are not associated to the agents, but all control information are stored inside the data. More precisely, SecOS supports two forms of locks which are called *symmetric* and *asymmetric*. The former exploits the same key to protect and access the information, while the latter uses a pair of keys, one to protect and another one to access. This two locking techniques can be applied to protect either one single field inside a tuple or the whole tuple. In the first case the used locks are called *Field-locks*, while in the second one, they are called *Object-locks*.

In SecOS a read operation is defined as the juxtaposition of an input followed by an output operation, namely

$$rd\ e\ x.P \stackrel{def}{=} in\ e\ x.(out\ x \mid P)$$

This fact has the two following consequences:

- There is no discrimination between non-destructive and desctructive input access policies. More precisely, there is no way to discriminate between the readers and the consumers of a datum. This could be an undesired feature in many applications. Consider, e.g., an information system in which an information can be accessed by any agent, but it can be removed only by specialized garbage collectors.
- An agent able to access a datum is also able to reproduce that datum, possibly creating new instances of that datum. This means that an agent which is a reader of a datum, is implicitly also a potential producer for that datum. Also this feature could be undesired in many applications. Consider, e.g., a master/worker system in which we want to ensure that tasks can be produced only by masters and consumed only by workers. According to the SecOS approach, there is no direct way to avoid a malicious worker to reproduce new instances of the tasks it consumes.

Starting from these observations, we propose a novel coordination model called `SecSpaces` which aims at supporting the advantages of both the approaches. In particular, as in SecOS we introduce the access information inside the data, but we refine the access controls supported in SecOS by permitting to discriminate, e.g., the read from the in permissions.

More precisely, we propose to extend a typical Linda-like coordination model introducing three new ways for controlling the access to data. The first way is to support logical partitions of the shared tuple space. This can be used to restrict the access to the data inside a specific partition only to the agents holding an explicit handle to be used to address that partition. The second mechanism we consider is to add some cryptographic information: these can be exploited, e.g., to authenticate the producer of a datum as well as identifying the readers/consumers for that datum. Finally, we support the possibility to discriminate the access control policies between (destructive) input and (non-destructive) read operations.

The main contribution of `SecSpaces` is in the definition of an advanced matching rule, which exploits some extra information that we permit to enter in the tuples. The examples, that we report in the remainder of the paper, prove the generality and flexibility of this matching rule, that can be exploited to achieve several access control mechanisms. However, acting only at the level of matching rule, we do not support any control on the execution of the output operations. Due to the asynchrony of the communication supported by the tuple space, we consider the problem of controlling the ability to produce tuples, as a different and orthogonal task with respect to the analysis of advanced matching rules to be used to retrieve those tuples. We leave for future work the definition of policies for the control of output operations.

Another novelty of our approach is that we define only the coordination model, without considering the internal structure of the agents accessing the coordination medium. More precisely, we formally specify the actual state of the tuple space, how this is modified by the execution of the coordination primitives, and which is the return value of the primitives. This corresponds to the typical approach taken in the investigation of coordination models and languages, i.e., support a distinct separation between coordination and computation concerns. In our specific case, this approach is particularly promising mainly for two reasons.

(i) If we develop a coordination model without making any assumption on the nature of the agents that will exploit that model, we obtain a model which is clearly open to heterogeneous agents, components based on different technologies, as well as new forms of agents which are unknown beforehand.

(ii) A formal definition of the coordination model, independently of the agents, permits an analysis of the basic properties of the model itself. These properties of the model are then expected to hold in all its specific exploitations.

The definition of a complete framework, for the modeling and analysis of both the agents and the tuple space, as is done in KLAIM and SecOS, is left for future work, when we will move our interest to the analysis of instances of systems exploiting our coordination model.

The remainder of the paper is structured as follows. Section 2 formalizes our starting point, i.e., a Linda-like coordination model with some form of subtyping; this model is useful to clarify the problems that we see when exploiting this family of coordination models in environments open to untrusted agents. Sections 3, 4 and 5 introduce incrementally our novel coordination model `SecSpaces`. Section 6 discusses related work, future research and reports some concluding remarks.

2 Linda-like coordination model

Most of the currently exploited Linda based coordination infrastructures are developed for object-oriented programming languages (see, e.g., JavaSpaces and TSpaces). To be consistent with this family of models, we use a matching rule between entries and templates which considers some form of subtyping. More precisely, in our abstract model, a tuple is a finite sequence of fields taken from a generic set *Data*.[1] In this context, it is reasonable to map the notion of subtype with the notion of extension: given a tuple e, each new tuple e' obtained by adding some extra fields to e, is considered to be a subtype of e. This form of subtyping influences the matching rule as follows: if an entry e matches a template t, then also all the extensions e' of e matches template t. This matching rule is inspired by the way subtyping is exploited in JavaSpaces and in SecOS. Differently from our model, in JavaSpaces subtyping corresponds to class inheritance and fields are typed objects. On the other hand, SecOS uses a notion of extension similar to ours, but tuples are *unordered* collections of fields, each one with an associated field lock used as a selector to access its content.

We now formalize this Linda-like coordination model extended with subtyping. Let *Data*, ranged over by d, d_1, ..., be a denumerable set of data. Let *Entry*, ranged over by e, e', ..., be the set of finite sequences of data taken from *Data* and denoted by $< d_1; \ldots ; d_n >$, $n \geq 1$. Templates may use wildcards to let some field unspecified; syntactically, a wildcard is denoted by *null* (which we assume to be not in *Data*). In the following, dt, dt_1, ..., range over $Data \cup \{null\}$. Finally, let *Template*, ranged over by t, t', ..., be the set of finite sequences composed of data and wildcards denoted by $< dt_1; \ldots ; dt_n >$, $n \geq 1$. We define the matching rule as follows.

Definition 2.1 Matching rule - Let $e =< d_1; \ldots ; d_n >$ be an entry and $t =< dt_1; \ldots ; dt_m >$ be a template; we say that e *matches* t if the following conditions hold:

- $m \leq n$,
- $dt_i = d_i \vee dt_i = null, 1 \leq i \leq m$.

With $e \ \not{matches}\ t$ we denote that e and t does not satisfy matching rule.

The coordination primitives we consider are the classical Linda operations: $out(e)$ which introduces the entry e in the shared repository; $rd(t)$ which verifies the presence of a matching entry e currently available in the shared repository (i.e. e *matches* t); $in(t)$ which is the destructive counterpart of $rd(t)$; $rdp(t)$ which is the non-blocking version of $rd(t)$, i.e. it terminates even if no matching is currently avalaible in the repository; $inp(t)$ which is the non-blocking version of $in(t)$.

The operations, except $out(e)$, have a return value: $rd(t)$ and $in(t)$ return a copy of the matching entry e (in the case of $in(t)$ this matching entry e is also removed from the repository); $rdp(t)$ and $inp(t)$ have the same return value

[1] In Linda, a tuple is a finite sequence of typed fields; for the sake of simplicity we consider a unique generic type *Data*.

(and effect) of $rd(t)$ and $in(t)$, respectively, except when no matching entry is currently avalaible. In such a case, the return value is $fail$ and denotes the absense of matching entries in the repository.

Let $Action = \{out(e), rd(t), in(t), rdp(t), inp(t)\}$ (ranged over by α) be the set of operations that an agent can perform, and $ReturnValue = Entry \cup \{fail, -\}$ (ranged over by r) be the set of return values of the operations. As described above, $fail$ denotes the failure of the non blocking read/input operations, whilst $-$ indicates that the operation does not have any return value, as it happens for the *out* operation.

In the following, TS, TS', ..., range over $\mathcal{M}(Entry)$, and are used to describe the states of the shared repository, i.e. the entries currently avalaible. To describe the effect of the operations on the shared repository, we use a labeled transition system, that is a quadruple ($\mathcal{M}(Entry)$,$Action$,$ReturnValue$, $\rightarrow$), where $\rightarrow\ \subseteq \mathcal{M}(Entry) \times Action$ x $ReturnValue \times \mathcal{M}(Entry)$. (TS,$\alpha$,$r$, TS')$\in\rightarrow$ (denoted also with TS $\overset{\alpha}{\underset{r}{\longrightarrow}}$ TS') has the following meaning: when the state is TS, and an action α is performed, then after the execution, the state of the repository is TS' and the return value is r. Table 1 defines the semantics of the primitives.

TS$\oplus\{e\}$ $\overset{in(t)}{\underset{e}{\longrightarrow}}$ TS, e *matches* t	TS$\oplus\{e\}$ $\overset{rd(t)}{\underset{e}{\longrightarrow}}$ TS$\oplus\{e\}$, e *matches* t
TS$\oplus\{e\}$ $\overset{inp(t)}{\underset{e}{\longrightarrow}}$ TS, e *matches* t	TS$\oplus\{e\}$ $\overset{rdp(t)}{\underset{e}{\longrightarrow}}$ TS$\oplus\{e\}$, e *matches* t
TS $\overset{inp(t)}{\underset{fail}{\longrightarrow}}$ TS, $\forall e \in$TS:e $\not{matches}$ t	TS $\overset{rdp(t)}{\underset{fail}{\longrightarrow}}$ TS, $\forall e \in$TS:e $\not{matches}$ t
TS $\overset{out(e)}{\underset{-}{\longrightarrow}}$ TS$\oplus\{e\}$	

Table 1
Semantics of the primitives.

As an example illustrating the use of wildcards to obtain previously unknown information, consider the entry $< d_1; d_2 >$ that can be read by any agent simply performing $rd(< d_1; null >)$ or even $rd(< null >)$. In particular, the operation $rd(< null >)$ exploits both a wildcard and subtyping to match with all possible entries in the space. This simple example shows that there is no way to bound the entry scope. Consequently, the first goal of this work is to extend the notion of entry, template and matching in order to introduce a manner to regulate the access to an entry.

3 Partitioning the space

In Linda-like systems each tuple in a shared repository may be potentially read or consumed by each agent having access to that repository. In many applications this could be an undesired feature; consider, e.g., applications in which the agents coordinated through the tuple space are divided into classes; it may be useful to distinguish the space of the entries in which the classes interact.

Starting from this observation, we show an approach to limit the visibility of entries. The proposal that we are going to introduce can be potentially used to separate (logically) the spaces in which groups of agents interact. Essentially, we extend the entry structure, with a special field, named *partition field*. The partition field is the only one in which the matching rule does not accept wildcards. Therefore, an agent executing a read operation on a subspace, is forced to deliver a proof of knowledge of the partition field value of the subspace that it wants to access.

Let C be a denumerable set of partition field values (ranged over by c, c_t, c_s, ...). We also assume that C contains a special default value $\#$; whose meaning will be described in the following. An entry e, and a template t with the partition field, are defined as follows ($n \geq 1$):

$$e =< d_1; \ldots; d_n >^{[c]}$$

$$t =< dt_1; \ldots; dt_n >^{[c_t]}$$

Let $Entry^c$ and $Template^c$ be the set of entries and templates with partition field, respectively. In the following, e, e', ..., range over $Entry^c$, and t, t', ..., range over $Template^c$. The tuples $< d_1; \ldots; d_n >$ and $< dt_1; \ldots; dt_n >$, $n \geq 1$, are also denoted in the compact form $< \vec{d} >$ and $< \vec{dt} >$, respectively. We also define $e|_d$ as the operator that, given an entry $e =< \vec{d} >^{[c]}$, returns the tuple of data $< \vec{d} >$.

Definition 3.1 Matching rule with partition fields - Let $e =< \vec{d} >^{[c]}$ be an entry, and $t =< \vec{dt} >^{[c_t]}$ be a template; we say that e *matches*^{c} t when the following conditions hold:

- $c_t = c$,
- $< \vec{d} >$ *matches* $< \vec{dt} >$.

The definition of *matches*^{c} is a conservative extension of *matches* (see Definition 2.1).

Example 3.2 matches^c evaluation - A list of entry matching examples follows.

template	does it match with $< d_1; d_2 >^{[c]}$?
$< null >^{[c]}$	yes
$< null; null >^{[c]}$	yes
$< d_1; null >^{[c]}$	yes
$< d_1; d_2 >^{[c]}$	yes
$< d_1; d_2 >^{[c']}, c' \neq c$	no
$< null; null; null >^{[c']}$	no

The semantics of the primitives, using the entry (template) with partition field and the matching rule $matches^c$, is obtained simply by replacing in Table 1 $matches$ with $matches^c$, and the e used to describe the return value with $e|_d$ (in this context $ReturnValue = \{e|_d \; |e \in Entry^c\} \cup \{fail, -\}$). Observe that the return value does not contain the partition field, as we use this information only to control the access to the entries.

The partition field and the matching rule induce a partitioning of TS. More precisely, every $c \in C$ identifies a subspace, that is composed by all entries in TS with partition field set to the value c. To match with entries of the subspace characterized by c, agents must know c. However, the special default value # provides the agents with a manner to cooperate each other, even without knowing any specific partition field value for a working subspace. For the sake of simplicity, when an entry (template) has the partition field set to #, we omit to specify it.

Example 3.3 Secure group communication - We have previously observed that the partition field induces a partitioning of TS. Hence, this fact can be exploited in order to limit, to a group of agents, the access to a partition. To each group G_i, we associate $c_i \in C$, with $c_i \neq c_j$, if $i \neq j$. We assume that c_i is known only by all the agents that are members of G_i, and that they keep c_i secret. In order to write, and consume, a datum into the space dedicated to G_i, it is sufficient to execute $out(< \vec{d} >^{[c_i]})$ and $in(< \vec{dt} >^{[c_i]})$, respectively. The rd, rdp, inp operations into the space dedicated to G_i can be obtained similarly. Using this approach, an agent that can read (or write) an entry to the subspace identified by c_i, is a member of G_i. We motivate this assertion observing that: i) the execution of an out operation implies the knowledge of c_i; therefore, by hypothesis, only an agent that is a member of G_i can do it; ii) $matches^c$ implies that who reads an entry $e =< \vec{d} >^{[c_i]}$ knows c_i; therefore, by hypothesis, only agents that are members of G_i can read it.

In order to evaluate the expressiveness of the language, it is interesting to analyse which information an agent can take about the current state of TS, as well as to analyse the ability to force specific modifications of the state. Here we study two examples, the first one is the function EmptyTS? that tests if the space is empty, and the second one is the function RemoveAll that removes all the entries in TS.

Example 3.4 Implementing function EmptyTS? - In the model of Section 2, EmptyTS? can be implemented with an atomic operation: $rdp(< null >)$. If no matching entry is found, the space is empty, otherwise it is not. The problem we encounter when implementing EmptyTS?, using the enriched model of this section, is that $rdp(< null >)$ should be repeated for each $c \in C$. Hence, EmptyTS? cannot be performed in a single step; it answers correctly only if no other agent changes TS during its execution (thus, it requires to be executed in a transactional manner). As an example, if C contains more than one value, e.g. c_1 and c_2, when the function tests if the subspace identified by c_1 is empty, an agent can write an entry in the subspace identified by c_2, and vice versa. In these cases, the function may terminate with a wrong result.

Example 3.5 Implementing function RemoveAll - Using the language of Section 2, the function RemoveAll can be implemented as a program that repeatedly executes a non blocking input $inp(< null >)$, until no matching is found. When the program terminates, TS is empty. The following piece of code implements this program:

```
repeat { m = inp (<null>); } until (m == fail);
```

To do this using the enriched model of this section, it is necessary to repeat, for each value of partition field $c \in C$, a non blocking input operation, using the template $< null >$, until it finds no matching. If C contains more than one value, e.g. c_1 and c_2, when the program terminates the removal of all the entries from the subspace characterized by c_1, agents may have written in the subspace identified by c_2 in the meanwhile, and vice versa. Hence, the function may terminate incorrectly.

A consideration about performance is needed. Given a template $< \vec{dt} >^{[c_t]}$, if there exists a matching entry e in the space, then $e =< \vec{d} >^{[c]}$ and $c_t = c$. We can exploit c_t in order to reduce the research space to the subspace identified by c_t.

4 Adding cryptography to the entries

In Section 3 we have show how to implement a TS structured into logical subspaces. The idea that we have followed to control the access is that only the agents that know c_s can write, or read, the entries in the subspace identified by c_s. However, we cannot distinguish between the read and write permissions on entries. On the other hand, it is rather easy to find applications that need this feature. For example, a client-server service (such as a print service): clients send the requests to the server which is the only agent that collects them. Using TS to implement the model, the client submits the work writing an entry, and the server reads the requests. It is important to allow the permission to read the submitted jobs only to the server. Furthermore, several applications need to authenticate the sender or the receiver of the entry. For example, the distribution of software: a software house distributes its software products, upgrades and patches, writing them into the space. In order to avoid to download and install uncertified code, the receiver requires the authentication of the data producer. In order to support these class of applications, the following section introduces entries with a new kind of field, named *cryptographic field*.

4.1 Cryptographic field

As commented above, we intend to use asymmetric cryptography to distinguish between writers and readers, and to provide a manner for authenticating the sender ("who has written the entry"), and the receiver ("who is executing the read/input"). In order to describe the entry, we firstly define:

- The set of plaintexts *PlainText*, ranged over by p, p',...;
- The set of encryption keys *Key*, containing private and public keys. In the following, when we refer to pairs of private and public keys $(PrivK, PubK)$, we assume that a plaintext encrypted with $PubK$ (resp. $PrivK$) can be decrypted only using $PrivK$ (resp. $PubK$);
- The set *Ciphertext*, ranged over by s, s_t,..., that contains the ciphertexts obtained encrypting $p \in PlainText$, with $k \in Key$, denoted with $\{p\}_k$.

In the following, r, r_t, ..., range over $(Key \times PlainText) \cup \{?\}$.

The idea is to exploit this new field in the matching rule, in order to capture a more sophisticated relation between entries and templates, that uses asymmetric cryptography. In particular, from one side a pair $r = (k, p)$ is given, and from the other one a ciphertext c; the test consists of comparing the text p with the text obtained by decrypting c (using an asymmetric encryption algorithm, e.g. [RSA78]) with key k. The special value $r =?$ denotes that the condition is to be ignored. An entry e, and a template t, with cryptographic (and partition) field, are defined as follows:

$$e =< \vec{d} >^{[c]}_{[r;s]}$$

$$t =< \vec{dt} >^{[c_t]}_{[r_t;s_t]}$$

Let $Entry^c_s$ and $Template^c_s$ be the set of entries and templates with cryptographic (and partition) fields, respectively. In the following, e, e', ..., range over $Entry^c_s$, and t, t', ..., range over $Template^c_s$. We define $e|_d$ as the operator that, given an entry $e =< \vec{d} >^{[c]}_{[r;s]}$, returns the tuple of data $< \vec{d} >$.

Definition 4.1 Matching rule with cryptographic fields - Let $e =< \vec{d} >^{[c]}_{[r;s]}$ be an entry, and

$t = < \vec{dt} >^{[c_t]}_{[r_t;s_t]}$ be a template; we say that e $matches^c_s$ t if the following conditions hold:

- if $r = (k, p)$ then $decrypt(s_t, k) = p$
- if $r_t = (k_t, p_t)$ then $decrypt(s, k_t) = p_t$
- $< \vec{d} >^{[c]}$ $matches^c$ $< \vec{dt} >^{[c_t]}$.

The definition of $matches^c_s$ is a conservative extension of $matches^c$ (see Definition 3.1).

Example 4.2 matches^c_s evaluation - Let $(PrivK_A, PubK_A)$ and $(PrivK_B, PubK_B)$ be pairs of private and public keys. A list of entry matching examples follows.

template	does it $matches^c_s$ with $< d >_{[(PubK_A,p),\{p_e\}_{PrivK_B}]}$?
$< null >_{[?,\{p\}_{PrivK_A}]}$	yes
$< null >_{[(PubK_B,p_e),\{p\}_{PrivK_A}]}$	yes
$< null >_{[?,\{p\}_{PrivK_E}]}$	no, if $PrivK_E \neq PrivK_A$

The semantics of the primitives, using the entry (template) with cryptographic field and the matching rule $matches^c_s$, is obtained simply by replacing in Table 1 *matches* with $matches^c_s$, and the e used to describe the return value with $e|_d$ (in this context $ReturnValue = \{e|_d \; | e \in Entry^c_s\} \cup \{fail, -\}$). Observe that also in this case the cryptographic fields are used only to control the access to the entries, then the return value does not include this field.

4.2 Examples

In order to prove the adequacy of cryptographic field to solve the security issues that we have tackled, this section presents several examples. In particular, we hint how to solve problems such as distinguishing the write and read permission, as well as the authentication of the sender and of the receiver.

4.2.1 Entry replication

Here we show that using cryptographic fields with private and public keys, it is possible to avoid the unauthorized replication of entries. This feature can be useful in several applications, e.g., to distinguish write and read permissions. On the contrary, if the readers can write the entries that they can read, then a reader is also a writer (read permission inherits write permission).

Let $(PrivK, PubK)$ be a pair of private and public keys of the agent P. If P publishes its public key $PubK$ and keeps $PrivK$ secret, then, even assuming p as a default value known by all the agents, only P can generate the entry $< \vec{d} >_{[?,\{p\}_{PrivK}]}$. By definition, the return value of a read operation does not include the cryptographic field. Hence, the reader does not obtain $\{p\}_{PrivK}$, and cannot generate exactly that entry because it does not know $PrivK$.

Therefore, except when the reader is the sender, the reader cannot reply the entry in the space.

Symmetrically, one may wonder if a writer is also a reader, that is if an agent can always match the entries that it produces. The answer is that an agent cannot always read the entries that it writes. In particular, it cannot when the entry that it writes has r set to $(PubK_i, p')$ with a generic plaintext p', and it does not know the respective private key $PrivK_i$. For instance, all agents can produce the entry $e = <\vec{d}>_{[(PubK_i,p'),s]}$ with a generic ciphertext s, because p' and $PubK_i$ are public. However, only the agent that holds $PrivK_i$ can read this entry, because to match with e, the reader must know, or must be able to generate, $\{p'\}_{PrivK_i}$ (s has no influence, because it may be disregarded setting in the template $r_t = ?$). Therefore, using the cryptographic field, we obtain the maximum flexibility to assign the write and read permission.

4.2.2 Secure channel

Several applications need to communicate using secure channels, e.g., the exchange of confidential data between two agents, or electronic commerce. Languages whose aim is to provide a way to guarantee a secure communication should accomplish secure channels implementation. Hence, we perform a comparison among SecOS, KLAIM and `SecSpaces` discussing the way a secure channels between two agents, say A and B, could be modeled in the three coordination platforms.

An encoding of a secure channel in SecOS [VBO02] could be based on asymmetric object locks. More precisely, a pair of keys is associated to the channel and one of the two keys is given to A and the other one to B. In this way, data availability and integrity, as well as authentication of sender and receiver are supported. A secure channel can be encoded in KLAIM creating a new location and allowing permissions of write/read on that location only to A and B, respectively. Both the proposals require a phase of construction/definition of the secure channel. In the case of SecOS, the encoding needs a trusted entity or a procedure that creates a pair of keys and distributes them to A and B. The phase of secure channel construction in KLAIM consists in the creation of the new location. This phase represents an overhead for the system and may be also an entry point for an attack by an enemy agent. Therefore, the elimination of such this phase could make the implementation more robust.

An encoding of a secure channel in `SecSpaces` follows. In particular, we do not exploit any preliminary phase, but we simply make the standard assumption that A knows the public key of B, and vice versa. Let $(PrivK_A, PubK_A)$, $(PrivK_B, PubK_B)$ be two pairs of private and public keys of A and B, respectively. We assume that $PrivK_A$ (resp. $PrivK_B$) is known only by A (resp. B), and that they keep it secret. We also assume that $PubK_A$, $PubK_B$, and a generic plaintext p are public pieces of information. The functions *ssend* and *sread*, used in order to send and to read a data from A to B on a secure channel, are encoded as follows:

$$ssend(<\vec{d}>) := out(<\vec{d}>_{[(PubK_B,p),\{p\}_{PrivK_A}]})$$

$$sread(<\vec{dt}>) := in(<\vec{dt}>_{[(PubK_A,p),\{p\}_{PrivK_B}]})$$

Using this encoding, we observe that: i) only A can generate $\{p\}_{PrivK_A}$, hence only A can write the entry encoded by the function *ssend*; ii) only B can generate $\{p\}_{PrivK_B}$, hence only B can (using *sread* function) read the data written by A; iii) B cannot reply the entry written by A on the secure channel (except for the case in which B is A), as observed in section 4.2.1. Hence, by i) and ii), the communication on the secure channel guarantees the mutual authentication of A and B. This property provides a way for guaranteing non repudiation. For example, suppose that A performs an electronic payment to B using the secure channel. A sends the payment to B using *ssend*, then A cannot repudiate the written entry (because it contains a data that is signed with its own private key), i.e. to deny the payment, that, by ii), can be collected only by B. Moreover, by iii), B cannot reply the entry in the space, hence it cannot recycle the payment.

Finally, a consideration on p. At least A and B must know p, then the solution assumes p to be a public data. However, in order to avoid to sign always the same p, during the interaction this value can be refreshed, e.g., transmitting new plaintexts p' within the tuple of data. In this way, p' is known only by A and B.

Example 4.3 Many-to-many communication - In this example we show how to encode the many-to-many communication model, distinguishing the permission of writing and reading the entries.

Let D_W and D_R be the set of agents constituting the group of writers and readers, respectively. Let $(PrivK_W, PubK_W)$ and $(PrivK_R, PubK_R)$ be pairs of private and public keys, and p_W, p_R be two generic plaintexts. We assume that: i) $PrivK_W$ (resp. $PrivK_R$) is known only by all the agents in D_W (resp. D_R), and that they keep it secret; ii) $PubK_W$, $PubK_R$, p_W and p_R are public pieces of information. The functions *outtogroup* and *infromgroup*, used in order to write and to read a data, can be encoded as follows:

$$outtogroup(<\vec{d}>) := out(<\vec{d}>_{[(PubK_R,p_R),\{p_W\}_{PrivK_W}]})$$

$$infromgroup(<\vec{dt}>) := in(<\vec{dt}>_{[(PubK_W,p_W),\{p_R\}_{PrivK_R}]})$$

The encoding ignores the partition because it is not necessary. Hence, the encoding that we provide can be applied to each partition. Using this encoding, we observe that:

- Only agents in D_W know $PrivK_W$, hence the ciphertext $\{p_W\}_{PrivK_W}$ can be generated only by an agent in D_W;

- In the reader template, $(PubK_W, p_W)$ imposes to capture only entry signed with $\{p_W\}_{PrivK_W}$, therefore written by an agent in D_W;
- Only agents in D_W know $PrivK_W$, hence a reader that is not also a writer (i.e. it is not in D_W) cannot replicate the data written by the agents in D_W;
- Only agents in D_R know $PrivK_R$, hence the ciphertext $\{p_R\}_{PrivK_R}$ can be generated only by an agent in D_R;
- In the writer entry, $(PubK_R, p_R)$ imposes to match only with template signed with $\{p_R\}_{PrivK_R}$, therefore generated by agents in D_R.

The approach that we have followed is similar to the encoding of the secure channel. However, the assumption that the private keys are known by more than one agent does not allow to authenticate the entities that write or read the entries.

4.2.3 Symmetric and asymmetric cryptography

Security protocols based on communication channels, typically use symmetric and asymmetric cryptography, in order to guarantee security properties (e.g. secrecy, authentication). The section compares symmetric and asymmetric cryptography with fields we have. In particular, we show similarities between symmetric cryptography and partition fields, as well as between asymmetric cryptography and cryptographic fields.

If agents A and B share a symmetric key K_{AB}, that is secret, then they are the unique agents that can encrypt the data with K_{AB}, and decrypt the obtained ciphertext. If A and B share a partition field p_{AB}, that is secret, then only A and B can write, and read, the entry $e = < \vec{d} >^{[p]}$ with $p = p_{AB}$. Hence, either solutions can be used to guarantee the secrecy of the data exchanged between A and B. Moreover, in either solutions there is no way to authenticate who writes, or who reads, the data.

Using asymmetric cryptography, it is possible: i) to sign a data (e.g. encrypting the data with its own private key) in order to allow the receiver to authenticate the sender; ii) to send an encrypted message, and to be sure that only one can decrypt it (simply by encrypting the message with the public key of the receiver), i.e. only the legitimate receiver can read the data. The same results can be obtained using cryptographic fields, as shown in Section 4.2.2.

Finally, we can encode the partition fields using cryptographic fields. Let $(PrivPF, PubPF)$ be a pair of private and public keys known by all the agents. We can encode the partition field set to c simply by compiling the cryptographic fields, in the entry and in the template, as follows: $[(PubPF, c), \{c\}_{PrivPF}]$. In order to write this entry, the writer must be able to generate $\{c\}_{PrivPF}$, hence it must know c. Analogously, to satisfy the matching rule the reader must be able to generate $\{c\}_{PrivPF}$, hence it must know c.

However, we continue to support both kinds of fields, for the following reasons: i) let $e = < \vec{d} >^{[c]}_{[r;s]}$, and $t = < \vec{d} >^{[c_t]}_{[r_t;s_t]}$, if $r = r_t = ?$ then $e\ matches^c_s\ t$ if and only if $< \vec{d} >^{[c]}\ matches^c < \vec{d} >^{[c_t]}$; ii) the $matches^c$ evaluation is faster than the one for $matches^c_s$, because it does not involve cryptography; iii) asymmetric cryptography is usually used to estabilish a symmetric session key between agents, because symmetric encryption is faster than asymmetric encryption. Analogously, we can use cryptographic field to estabilish a "session" partition field between agents. For example, suppose that A has a new partition field p_{AB}, that is secret. In order to estabilish the session partition with B, A can use a secure channel (see Section 4.2.2) to communicate p_{AB} (e.g., using $ssend(< p_{AB} >)$) to B.

5 Discriminating the in and rd permissions

There are examples of applications in which several (or all) agents can read some data, but only a subset of them can delete such data. For example, every person can publish some data of interest to the community (e.g. name, phone, e-mail, ..), writing an entry in the space. It is reasonable to think that this information should not be removed by the agents, except by the owner (for example, to update the published data). To accomplish this goal, cryptographic field values are not sufficient to describe the whole set of permissions that are necessary. In this section we present how to extend the entry in order to describe the permission of the data retrieval operations. In particular, we introduce two distinct permissions, the first one for non-destructive (*rd* and *rdp*) operations, and the second one for destructive (*in* and *inp*) operations. To obtain this goal, we simply associate a partition field and a cryptographic one, to each permission. Let $Entry^c_s(rd, in)$ be the set, ranged over by e, e', ..., of entries with read and input permissions, defined as follows:

$$e = < \vec{d} >^{[c]_{rd}[c']_{in}}_{[r;s]_{rd}[r';s']_{in}}$$

We define $e|_d$ as the operator that, given an entry $e = < \vec{d} >^{[c]_{rd}[c']_{in}}_{[r;s]_{rd}[r';s']_{in}}$, returns the tuple of data $< \vec{d} >$. Hence, the matching rule is a function on the operation currently executed. The template continues to be defined as in the previous section (that is $t \in Template^c_s$), because the reference to the current operation is inherited by the operation itself.

Definition 5.1 Matching rule with permissions - Let $op \in \{rd, rdp, in, inp\}$ be the primitive on which the matching must be evaluated, t be a template and $e = < \vec{d} >^{[c]_{rd}[c']_{in}}_{[r;s]_{rd}[r';s']_{in}}$ be an entry; we define $e\ matches^c_s(op)\ t$ as follows:

- $< \vec{d} >^{[c]}_{[r;s]}\ matches^c_s\ t$, if $op \in \{rd, rdp\}$.
- $< \vec{d} >^{[c']}_{[r';s']}\ matches^c_s\ t$, if $op \in \{in, inp\}$.

In other words, $matches^c_s(op)$ is reduced to $matches^c_s$ using only the partition and the cryptographic field of e corresponding to op.

It is trivial to prove that $matches^c_s(op)$ is well defined; in both cases $matches^c_s$ is applied to entry $e \in Entry^c_s$ and template $t \in Template^c_s$.

The semantics of the primitives, using the entry with permission on the operations and the matching rule $matches^c_s(op)$, is obtained simply by replacing in Table 1 *matches* with $matches^c_s(op)$, and the e used to describe the return value with $e|_d$ (in this context $ReturnValue = \{e|_d \, |e \in Entry^c_s(rd, in)\} \cup \{fail, -\}$).

We call `SecSpaces` the coordination model characterized by the primitives and the semantics of this section.

5.1 Examples

Entry access control is based on the capacity of the reader to provide a proof of possessing the permission (i.e., to satisfy $matches^c_s(rd)$). Consequently, to assign the read-only permission to an entry corresponds to the problem of finding a setting of partition and cryptographic fields, relative to the *in* operation, that no agent can be able to match. Hence, the problem is to find a pair (k, p) such that nobody can guess the s that, when decrypted with key k, gives p. The same conclusion follows when we want to assign only the permission of consuming the data (*in*,*inp*). A possible solution is to include, within Key, a special value, say \$, for denoting that the matching can never be satisfied.

The model "many-to-many" encoded in Example 4.3, can be extended in this context for refining the way that agents can use to interact. For example, the agents interacting can be collected into three groups: D_W be the group of writers; D_R be the group of agents that can match the entry written by agents of D_W, only using rd and rdp; D_I be the group of agents that can match the entry written by agents of D_W, only using in and inp. It is obtained simply by introducing a new pair of private and public key $(PrivK_I, PubK_I)$, and p_I. We assume that $PrivK_I$ is known only by all the agents in D_I, that they keep $PrivK_I$ secret, and that $PubK_I$ and p_I are public pieces of information. Using also the assumptions of Example 4.3, the encoding of the write, read and input group operations is as follows:

$$outtogroup(<\vec{d}>) :=$$

$$out(<\vec{d}>_{[(PubK_R,p_R),\{p_W\}_{PrivK_W}]rd[(PubK_I,p_I),\{p_W\}_{PrivK_W}]in})$$

$$rdfromgroup(<\vec{dt}>) := rd(<\vec{dt}>_{[(PubK_W,p_W),\{p_R\}_{PrivK_R}]})$$

$$infromgroup(<\vec{dt}>) := in(<\vec{dt}>_{[(PubK_W,p_W),\{p_I\}_{PrivK_I}]})$$

Example 5.2 The *in* permission does not inherit *rd* permission - It is legitimate to wonder if an agent, with only the *in* permission on the entry e, can simulate the read removing e with an *in*, and then writing it in the space. In section 4.2.1 we have shown that it is possible to avoid the unauthorized replication of data in TS. Hence, *in* permission does not inherit *rd* permission.

6 Conclusion and related work

In this paper we have proposed `SecSpaces`, a Linda-like coordination model that permits to control the access to the entries stored in the shared data space, and to authenticate/identify the producer of an entry or its reader/consumer. `SecSpaces` substantially introduces a new advanced matching rule, which exploits two new kinds of fields, namely partition and cryptographic fields.

In the Introduction we have already discussed two proposals, KLAIM and SecOS, that aim to solve the problem of adding access control policies to the primitives of Linda. A comparison is shown in Section 4.2.2, where the encoding of secure channels in SecOS, KLAIM and `SecSpaces` are discussed. Here we want to add a further observation, the symmetric and asymmetric field locks of SecOS are used to select specific fields inside entries, hence they provide also a way to assign the permission for accessing the fields. Differently, `SecSpaces` uses the position of fields inside the tuples to access them, and an agent may use standard cryptographic approaches for protecting specific fields.

Here we report a list of other significant proposal present in the literature. The proposal in [MMU01] is based on the concept of law-governed interactions. In particular, permission policies (law) are described using Prolog rules; the agents have the corrects access rights if they satisfy the rules. This check is executed by policy-independent trusted controllers. The proposal of [Pin92] is to provide Linda with directed communication, i.e. private channels. The solution exploits asymmetric cryptography, at each entry corresponds a special data, the ticket. The matching is avalaible only if the tickets of entry and template match (i.e. one of them is the public ticket and the other one is the corresponding private ticket). Finally, [Woo99] presents a survey of the possibilities provided adding attributes in the tuple-space systems. In particular, several manners to introduce the attributes are discussed, more precisely how to insert them to the level of fields, of entries and of tuple-spaces.

References

[Gel85] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[MMU01] N. Minsky, Y. Minsky, and V. Ungureanu. Safe Tuplespace-Based Coordination in Multi Agent Systems. *Journal of Applied Artificial Intelligence*, 15(1), 2001.

[NFP97] R. De Nicola, G. Ferrari, and R. Pugliese. Coordinating Mobile Agents via Blackboards and Access Rights. *In Proc. of the Second International Conference on Coordination Models and Languages, Lectures Notes in Computer Science 1282, Springer*, pages 220–237, 1997.

[NFP98] R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: A Kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, May 1998. Special Issue: Mobility and Network Aware Computing.

[Pin92] J. Pinakis. Providing directed communication in Linda. *In Proceedings of the 15th Australian Computer Science Conference*, pages 731–743, 1992.

[PSD98] P.Wyckoff, S.W.McLaughry, and D.A.Ford. TSpaces. *IBM System Journal*, August 1998.

[RSA78] R.L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communication of the ACM*, 21(2):120–126, February 1978.

[Sun02] Sun Microsystems, Inc. *JavaSpaces^{TM} Service Specification*, 2002. URL: http://www.sun.com/jini/specs/.

[VBO02] Jan Vitek, Ciarán Bryce, and Manuel Oriol. Coordinating Processes with Secure Spaces. *To appear in Science of Computer Programming*, 2002.

[Woo99] Alan Wood. Coordination with Attributes. *In Proceedings of the Third International Conference COORDINATION '99, Springer LNCS-1594*, pages 21–36, 1999.