SECURITY ISSUES IN THE TUPLE-SPACE COORDINATION MODEL

Mario Bravetti   Nadia Busi   Roberto Gorrieri
Roberto Lucchi   Gianluigi Zavattaro
Dipartimento di Scienze dell'Informazione, Università degli Studi di Bologna,
Mura Anteo Zamboni 7, I-40127 Bologna, Italy.
{bravetti,busi,gorrieri,lucchi,zavattar}@cs.unibo.it

Abstract We present some security issues that emerge when the tuple-space coordination model is used in open systems. Then we describe `SecSpaces`, a tuple-space based language, which supports secure coordination in untrusted environments. Finally, we will discuss some real examples of applications interacting via tuple spaces by showing how to support some of the main security features with `SecSpaces`.

1. Introduction

New networking technologies are moving to support applications for *open systems* (e.g., peer-to-peer, ad-hoc networks, Web services), in which the entities that will be involved in the application are unknown at design time. Further, the connectivity is exploding: a growing number of devices need to communicate with each other and the new challenge is how to design and to program the communication among devices.

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 this class of applications. The interaction is programmed by means of a coordination infrastructure that abstracts away from the exact name/location of the components and the underlying network.

One of the most prominent coordination languages is Linda [Computing Associates, 1995] in which a shared space, containing tuples of data, is used by agents to collaborate. Agents can insert new tuples into the space, consume or read tuples from the space thus implementing the so-called generative communication, in which tuples are independent of their producers.

We present some security issues that emerge when Linda is used in open systems, where any agent can access the tuple space. In this scenario, where the presence of malicious agents may compromise the behaviour of the system, designers have to deal with security. Unfortunately, Linda is not expressive enough to provide security solutions, because any agent can read, remove and reproduce any tuple available in the shared space.

We present the `SecSpaces` [Busi et al., 2002, Bravetti et al., 2003] coordination language, based on the tuple-space coordination model introduced by Linda, that supports security in untrusted environments by providing some access control mechanisms on tuples with a granularity at the level of single tuples. The proposed solution follows a data-driven approach: the access to a tuple is subordinated to the proof of knowledge of certain data stored into the tuple. In a few words, to access a tuple it is necessary to provide special data, that we call control fields, that must match the ones stored inside the tuple. We also describe how such control fields can be implemented and, finally, how to define the matching rule. The proposed solution has also been developed [Lucchi and Zavattaro, 2004] in the context of Web Services technology, which represents the emerging networking technology for programming Internet applications.

In order to show how the `SecSpaces` mechanisms can be exploited for supporting some security features (e.g., secrecy, entity authentication) we consider real examples of applications where the interaction is programmed via tuple spaces.

The paper is structured as follows. Section 2 describes the Linda coordination primitives and `SecSpaces` with particular care to the security mechanisms obtained by decorating tuples with control fields. Section 3 describes some real examples where the `SecSpaces` model is used to support some forms of secure interaction. Finally Section 4 reports the main related works and concludes the paper.

2. `SecSpaces`

The `SecSpaces` language is an extension of Linda supporting security. In order to introduce `SecSpaces` we first present the Linda primitives. The Linda language provides coordination primitives that allow processes to insert new tuples into the tuple space (TS for short) or to access the tuples already stored in the shared tuple space. More precisely, a tuple is a sequence of typed values [Computing Associates, 1995] and a TS is a multiset of tuples.

Processes can exchange tuples through introducing them into the TS. The primitive $out(e)$ permits to add a new occurrence of the tuple e to the TS.

The data-retrieval primitives permit processes, by specifying a template t, to access tuples available in the TS that match the template. More precisely, a template is a sequence of fields that can be either actual or formal: a field is actual if it specifies the type and a value, while it is formal if the type only is given. Two typed values match if they have the same value, while a typed value matches a formal fields if it has the type specified in the latter. A tuple e matches the template t if t and e have the same arity and each field of e matches the corresponding field of t.

The $in(t)$ is the blocking input primitive: when a tuple e matching the template t is available in the TS, an occurrence of e is removed from the TS and the primitive returns e. The $rd(t)$ primitive is the blocking read primitive: differently from the $in(t)$, when a tuple e matching the template t is in the TS, it returns e without removing it from the TS.

Linda also provides the non-blocking version of the data-retrieval primitives: the inp and the rdp are the non-blocking version of the in and the rd, respectively. If the tuple e is in the TS, their behavior is the same as for the blocking operations, otherwise they return a special value indicating the absence of e in the TS.

Recent distributed applications such as Web services, applications for Mobile Ad Hoc Networks (MANETs), Peer to Peer Applications (P2P) are inherently open to processes, agents, components that are not known at design time. When the Linda coordination model is exploited to program the coordination inside this class of applications (see e.g. Lime [Murphy et al., 2001] in the context of MANETs and PeerSpaces [Busi et al., 2003] for P2P applications) new critical aspects come into play such as the need to deal with a hostile environment which may comprise also untrusted components.

The main issues are related with the fact that, in such a context, any entity is allowed to perform insertion, read and removal of tuples to and from the tuple space. In particular this means that any process can maliciously insert an unbounded number of tuples; in such a way, since the manager of the space has to handle any out operation, a process can generate a denial of service attack.

Another denial of service attack is due to the fact that any process can maliciously read/remove any tuples from the space, thus compromising the applications interacting via tuple-space. Indeed any entity can, by using the wildcard, generate a template that matches with any tuple having th same arity. Therefore, for example, a template having two wildcard fields can be used to read or remove any tuple containing two data fields. Moreover, since any entity can read/remove/reproduce any tuple from and into the space in such a model, we cannot authenticate neither the producer, nor the receiver of tuples. The threat of such lacks, that `SecSpaces` aims to cover, will be highlighted in the following section, but it should be rather clear that such state is to be avoided in open systems, where the applications interact by using the same tuple space and the availability of the tuples they produce is necessary to guarantee their correct behavior.

SecSpaces introduces an access control to the tuple-space coordination model which follows the data-driven mechanism. More in detail, Linda tuples are decorated with two kind of control fields: the *partition key* and the *asymmetric key*. Each tuple contains, for each possible operation the process can perform on the tuple (i.e. read and removal), a pair of control fields composed of a partition and an asymmetric key. Control fields are evaluated in the matching rule which is responsible for controlling the access to the tuple: the access to a tuple is allowed only to the entities which provide control fields matching those of the tuple associated with the operation the entity is performing. Two partition keys match if they are equal, while two asymmetric keys match if one is the co-key of the other one.

Formally, let $Mess$, ranged over by m, n, ..., be an infinite set of messages, $Partition \subseteq Mess$, ranged over by c, c_t, ..., be the set of partition keys and $AKey \subseteq Mess$, ranged over by k, k', k_t, ..., be the set of asymmetric keys. We also assume that $Partition$ (resp. $AKey$) contains a special default value, say # (resp. ?), used to allow any entity to access the space. Let $\overline{\cdot} : AKey \rightarrow AKey$ be a function such that $\overline{?} = ?$ and if $\overline{k} = k'$ then $\overline{k'} = k$. Informally, such function maps asymmetric keys to the corresponding co-keys. Moreover, as in the public-key mechanism, we assume that given an asymmetric key it is not possible to guess its co-key. In the following, we use $\vec{d}$ to denote a finite sequence of data fields.

The tuple structure in SecSpaces is defined as follows:

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

where $\vec{d}$ is a finite sequence of data fields whose values range over $Mess$, $c, c' \in Partition$ and $k, k' \in AKey$. The sequence of data fields in $\vec{d}$ represents the content of standard Linda tuples, while c and k (resp. c' and k') are the control fields used when such tuple is accessed by a read (resp. removal) operation. In the following we use the function $key(e, op)$ (resp. $akey(e, op)$) as the one that given $op \in \{rd, in\}$ and a tuple e returns the partition key (resp. asymmetric key) of e associated to op.

Templates are decorated with one occurrence of control fields, that will be associated to the operation the process is performing:

$$t =< \vec{dt} >^{[c_t]}_{[k_t]}$$

where $\vec{dt}$ is a finite sequence of data fields, $c_t \in Partition$ is the partition key and $k_t \in AKey$ is the asymmetric key associated to t. Differently from tuples, data fields contained in $\vec{dt}$ can also be set to the wildcard value denoted with $null$: the wildcard is used to match with all field values.

DEFINITION 1 (MATCHING RULE) *Let* $e = < d_1; d_2; \ldots; d_n >^{[c]_{rd}[c']_{in}}_{[k]_{rd}[k']_{in}}$ *be a tuple,* $t = < dt_1; dt_2; \ldots; dt_m >^{[c_t]}_{[k_t]}$ *be a template and* $op \in \{rd, in\}$ *be an operation. We say that* e *matches* t *(denoted with* $e \triangleright_{op} t$*) if the following conditions hold:*

1 $m = n$

2 $dt_i = d_i$ *or* $dt_i = null$, $1 \leq i \leq n$

3 $key(e, op) = c_t$

4 $\overline{akey(e, op)} = k_t$.

Condition 1. and 2. rephrase the classical Linda matching rule, that is test if e *and* t *have the same arity and if each data field of* e *is equal to the corresponding field of* t *or if this latter one is set to wildcard. Condition 3. tests that the partition key of the tuple associated to the operation op is equal to that of the template. Condition 4. checks that the asymmetric key of the template corresponds to the co-key of the asymmetric key of the tuple associated to the operation op.*

Essentially partition keys are a special kind of data field that do not accept wildcard in the matching evaluation. In this way, such keys logically partition the space and the access to a partition is restricted to those processes that know the associated key. Indeed, in order to perform an operation on the partition containing all the tuples with a certain partition key, processes must know the key which identifies that partition.

Differently from partition keys, the asymmetric keys make it possible to discriminate the permission of write, read and remove of a tuple. For instance, to read a tuple with asymmetric key k the process must provide a template with asymmetric key set to $\overline{k}$. It is worth noting that by using such keys the knowledge used to produce a tuple (k) is different from the one used for retrieving that tuple ($\overline{k}$). Therefore, by properly distributing these values we can assign processes the permission to perform a subset of the possible operations on that tuple, thus discriminating among the processes that can produce, read or remove that tuple.

EXAMPLE 2 *Some matching example follow (*$e \not\triangleright t$ *means that* e *does not match with* t*):*

$< d >^{[c]_{rd}[c']_{in}}_{[k]_{rd}[k']_{in}} \triangleright_{rd} < null >^{[c]}_{\overline{[k]}}$

$< d >^{[c]_{rd}[c']_{in}}_{[k]_{rd}[k']_{in}} \not\triangleright_{rd} < null >^{[c]}_{\overline{[k']}}$

$< d >^{[c]_{rd}[c']_{in}}_{[k]_{rd}[k']_{in}} \not\triangleright_{in} < d' >^{[c']}_{\overline{[k']}}$

DEFINITION 3 (RETURN VALUE) *The rd (resp. in) primitive with template t terminates when a tuple e such that $e \rhd_{rd} t$ (resp.$e \rhd_{in} t$) is available in the tuple space and the return value is composed of the data fields contained in e, while control fields are not returned. For example, if the matching tuple is $< d >^{[c]_{rd}[c']_{in}}_{[k]_{rd}[k']_{in}}$ the return value is $< d >$.*

By such definition it follows that dynamic privileges acquisition can happen only when control fields values are stored inside the sequence of data fields.

`SecSpaces` implementation

The study of all the issues related with a secure implementation of the `SecSpaces` model has been investigated in [Lucchi and Zavattaro, 2004]. Here we just report the way we use to implement control fields.

The implementation of partition keys is rather easy and similar to symmetric cryptography; the only assumption is that a process should not be able to guess an unknown partition key used by other processes. Similarly to symmetric encryption keys (see e.g. [Schneier, 1996]), we need to implement the set $Partition$ so that to guess one of its values has low probability. Such feature can be realized, e.g., by encoding partition keys with data composed by a large number of bits (say 512 bits).

The main problem we have to tackle when we implement asymmetric keys is how to satisfy the function $\overline{\cdot}$. Such function must guarantee that: i)it is possible to check whether two keys k and k' match (i.e. to verify if $k' = \overline{k}$), and ii)it is not possible for a process to guess $\overline{k}$ starting from the knowledge of k. To implement such keys we exploit the public-key cryptographic mechanism. Formally, let $PlainText$, ranged over by p, p',..., be the set of plaintexts, Key, ranged over by $PrivK$, $PubK$, ..., be the set of encryption keys 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 by using $PrivK$ (resp. $PubK$). Let $Ciphertext$, ranged over by s, s_t,..., be the set of ciphertexts obtained by encrypting plaintexts with encryption keys (we denote with $\{p\}_k$ the encryption of p with key k).

We encode any asymmetric keys, except the default value $?$ encoded with $?'$, with a triple $(p, PubK, s)$. The following implementation of $\overline{\cdot}$ satisfies the requirements of asymmetric keys:

- given $?'$, we have that $\overline{?'} = ?'$;
- given the triple $(p, PubK, s)$, we have that $\overline{(p, PubK, s)} = (p', PubK', s')$ if $s = \{p'\}_{PrivK'}$ and $s' = \{p\}_{PrivK}$.

Obviously, the correctness of such implementation is to be subordinate to a perfect implementation of cryptographic operations.

3. Examples

In this section we consider some real applications and we show how to manage the interaction by using the `SecSpaces` model. We show how `SecSpaces` makes it possible to guarantee some of the main security properties like secrecy, producer/receiver authentication and data availability. In particular we present two examples that we take from the use cases of [Gigaspaces]. For each of them, we proceed as follows: i)we describe how it works, ii)we describe the security lacks if the interaction is programmed with Linda, and iii)we describe how to support security by using `SecSpaces`.

Distributed Session Sharing

This example shows how to exploit a tuple-space repository for implementing a service for managing user sessions, e.g., the sessions used to control business activities.

The use case we consider consist of a customer that intends to reserve a car from one agency and a flight from another one. The car and flight reservation systems are located in separate servers. Both systems need to be able to share the user session, so that from the customer's point of view it is a single transaction. Since these services are distributed, we exploit the tuple space to implement a distributed session server that makes it possible to share the user session. The solution we are going to discuss is depicted in Fig. 1. The idea is that at any customer sessions the travel agency collects user data (shopping card, user id, etc.) and invokes (without a specific order) three services: i)the car service, ii)the flight service, and iii)the billing service, by passing as parameters the user session, shopping card and the user preferences. The information the travel agency receives from such services consists of, respectively: i)the ordered car, ii)the flight, and iii)the bill of the transaction. The car and flight services supply the corresponding request and then insert a tuple containing the fee of the supplied service and the user session id that is used as key field by the billing service that consumes both tuples and then returns the bill to the travel agency.

The main security problem is that the tuples inserted in the space are available to anyone, thus someone could maliciously use that id to perform, e.g., another business activity or to alterate the service fee. Indeed, let us suppose that car and flight services produce tuples with the following structure: $< userid, fee, preferences, serviceinfo >$, and that the billing service collects such tuples by performing two in with $< userid, null, null, null >$ as template. The threat is that such tuples can be removed/manipulated by, e.g., a malicious process, that can use the user id in a different context or can change the service fee, the user preferences or the identifier of the service ($serviceinfo$) supplying the requested task.

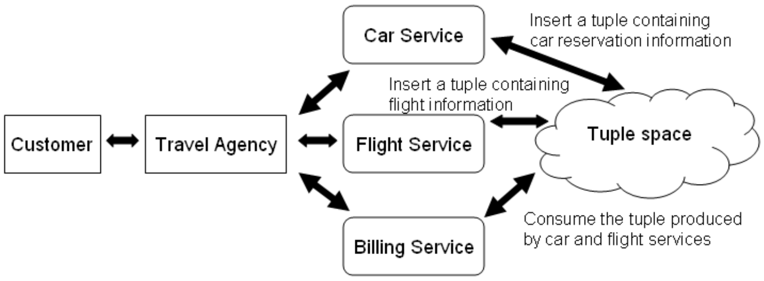

Figure 1. Distributed user sessions managed with a tuple space

The security issues explained above can be tackled by exploiting `SecSpaces`. In particular, the properties that should be guaranteed are: i)secrecy of exchanged data, and ii)receiver authentication. The former is needed to guarantee that the exchanged information are not used by unauthorized users while the latter is used to guarantee that the only process which is allowed to consume the tuples $< userid, fee, preferences, serviceinfo >$ is the billing service.

Let c and c' be two partition keys and k be an asymmetric key. To support such properties, we assume that:

- c is a private partition key shared by the car and the billing service (i.e. only those services know c),
- c' is a private partition key shared by the flight and the billing service (i.e. only those services know c'), and
- k is a private data of the billing service, while $\overline{k}$ is a public data.

In this paper we do not tackle the problem of how to distribute in a secure way such values. A possible solution is to exploit classic public-key infrastructure [Schneier, 1996].

The tuples produced (and then inserted into the space by an out) respectively by the car and by the flight service are now the following:

$$< userid, carfee, preferences, carserviceinfo >^{[c]_{rd}[c]_{in}}_{[\overline{k}]_{rd}[\overline{k}]_{in}}$$
$$< userid, flightfee, preferences, carserviceinfo >^{[c']_{rd}[c']_{in}}_{[\overline{k}]_{rd}[\overline{k}]_{in}}$$

To access such tuples it is necessary to provide a template having c (or c') as partition key and k as asymmetric key. In particular, the billing service perform $in(< userid, null, null, null >^{[c]}_{[k]})$ and $in(< userid, null, null, null >^{[c']}_{[k]})$ to read (and remove) the tuple containing information produced by the car and by the flight service, respectively. The secrecy of the data exchanged via the tuple space is guaranteed because only the two involved services (car-billing or flight-billing) know the partition key, while the authentication of the receiver directly follows by the fact that only the billing service is able to provide k as asymmetric key.

Brokered messaging

The use case we consider here is a messaging service where the interaction between the parties is mediated by a broker. Service producers (masters) produce messages about instructions or other information such as images, files, specifying also the receiver of such messages. A special service, the broker, is responsible for analysing submitted requests such as to determine which consumer service it should be sent to. It can possibly modify or insert additional data (e.g., a timestamp) into the message and then to deliver it to the relevant service consumer.

Such interaction can be programmed by exploiting a tuple space: i)masters insert a tuple containing all the information into the space with a certain structure that the broker knows, e.g., $< broker, msg, to >$ where $broker$ is the key used to identify tuples that the broker should take into account, msg is the information represented by the message and to specifies the receiver (we assume receivers can be unequivocally identified by an id), ii)the broker reads (and removes) submitted tuples by performing an in operation with $< broker, null, null >$ as template, it analyses the information about the message and the receiver and, after having performed some controls on submitted data, inserts a the tuple $< to, msg, timestamp >$ into the space where $timestamp$ indicates when the message has been receiver by the broker, and iii)any consumers whose id is $consid$ performs an in by using $< consid, null, null >$ as template; in this way it obtains the transmitted object and its timestamp. The interaction schema of such application is described in Fig. 2.

There are several kinds of aspects that a secure implementation should take into account like the secrecy of exchanged data (in order to guarantee the confidentiality), or the authentication of the message producers and consumers that can be managed by following an approach similar to the one used in the previous example. Here we just describe how to support another aspect: the fairness between producer and consumer. Indeed, any consumer (not only the proper consumer) can consume any tuples submitted by the producer thus preventing the broker to analyse such tuple. Essentially, the problem is that we cannot guarantee non-repudiation. In such a way, for example, the consumer can take an advantage w.r.t. the producer because it can also repudiate that it has received such message.

By assuming that the broker logs the exchanged messages we can exploit asymmetric keys of `SecSpaces` to cover this security lack by managing the

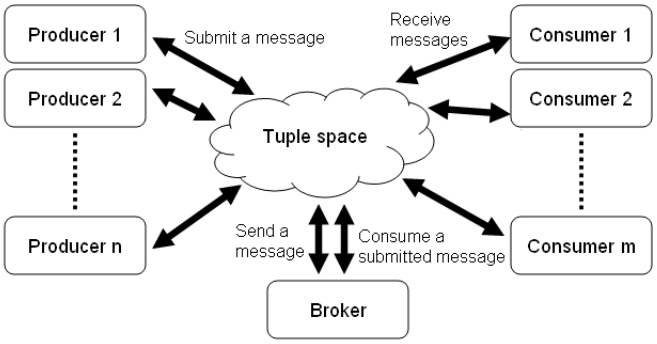

Figure 2. Brokered messaging managed with a tuple space

authentication of the receiver (the broker) for the tuples submitted by the producers. Moreover, we could manage the receiver authentication (the proper consumer) for the tuples inserted by the broker. In this way we guarantee that only the broker can consume (and then analyse) the tuples submitted by the producers and that only the receiver specified by the producer can consume the corresponding message. Technically, let k_b and k_i for $i = 1, \ldots, m$ be asymmetric keys; we make the following assumptions:

- k_b is a private information of the broker,
- k_i is a private information of the consumer i,
- $\overline{k_b}$, $\overline{k_i}$ for $i = 1, \ldots, m$ are public data.

The producers insert tuples with asymmetric keys (both for rd and in) set to $\overline{k_b}$, say $< broker, msg, to >_{[\overline{k_b}]_{rd}[\overline{k_b}]_{in}}$ that only the broker can access since it is the only one that knows k_b, which is needed to match the tuple. The broker consumes and analyses such tuples and by evaluating the field to selects the public key corresponding to the consumer, say k_i, and insert the tuple $< to, msg, timestamp >_{[\overline{k_i}]}$ that only the proper consumer can take since it is the only which knows k_i.

The example can be furtherly extended. Let us suppose the case where the producer can also specify that the message can be read but not removed because, e.g., it is necessary to track the execution of a complex activity or protocol. In this case the broker can set to k_i the asymmetric key of the tuple associated to the rd and to set to another value that consumers cannot match (e.g., $\overline{k_b}$) the one associated to the in. In this way consumers can only generate templates that match such tuple when it is accessed with read operations.

4. Conclusion

We have described the main security issues that emerge when Linda is used in open systems and described `SecSpaces` that make it possible to support some of the main security properties (e.g., secrecy, producer/receiver authentication). The adequacy of such proposal has been proved by considering some real examples of usage of tuple spaces in the interaction.

Other proposals supporting security are available in literature. The most interesting ones that deserve to be mentioned are Klaim [De Nicola et al., 1998] and SecOS [Vitek et al., 2003]. The former exploits a classic access control mechanism in which permissions describe, for each entity, which are the operations it is allowed to perform (insertion, read and removal of tuples), while the latter is based on access keys stored on tuples, which has inspired the `SecSpaces` language. Another approach is presented in [Handorean and Roman, 2003] where a password-based system on tuple spaces and tuples permits the access only to the authorized entities, that is those that know the password. In particular, password-based access permissions on tuples can be associated to the read and to the removal operations. Differently from `SecSpaces`, if an entity is allowed to remove a tuple (i.e. it knows the password associated to the removal operations), it has also the permission of reading that tuple.

We consider that the data-driven approach followed by SecOS and subsequently by `SecSpaces` is more suitable for open systems w.r.t. to classic one used in Klaim. In a few words, we identify the problem in the fact that to know all the possible entities that may enter in the system is a difficult task. Since in Klaim access permissions refer to entities, such task is necessary. On the other hand, the data-driven mechanism makes it possible to avoid such task since the access permissions are simply based on the proof of knowledge the entities provide when they perform coordination primitives.

The main contribution of `SecSpaces` is a refinement of the SecOS access permissions on tuples that make it possible to discriminate between the permissions of producing, reading and consuming a tuple. For example, in SecOS a process that can consume a tuple is also able to reproduce that tuple, thus in permission inherits out permissions. If we consider the brokered messaging example, we cannot guarantee non-repudiation of the messages received by the consumers.

References

[Bravetti et al., 2003] Bravetti, M., Gorrieri, R., and Lucchi, R. (2003). A formal approach for checking security properties in SecSpaces. In *1st International Workshop on on Security Issues in Coordination Models, Languages and Systems*, volume 85.3 of *ENTCS*.

[Busi et al., 2002] Busi, N., Gorrieri, R., Lucchi, R., and Zavattaro, G. (2002). Secspaces: a data-driven coordination model for environments open to untrusted entities. In *1st International Workshop on Foundations of Coordination Languages and Software Architectures*, volume 68.3 of *ENTCS*.

[Busi et al., 2003] Busi, N., Manfredini, C., Montresor, A., and Zavattaro, G. (2003). PeerSpaces: Data-driven Coordination in Peer-to-Peer Networks. In *Proc. of ACM Symposium on Applied Computing (SAC'03)*, pages 380–386. ACM Press.

[Computing Associates, 1995] Computing Associates, S. (1995). *Linda: User's guide and reference manual*. Scientific Computing Associates.

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

[Gigaspaces] Gigaspaces. Use cases. http://www.gigaspaces.com/usecases.htm/.

[Handorean and Roman, 2003] Radu Handorean and Gruia-Catalin Roman (2003). Secure Sharing of Tuple Spaces in Ad Hoc Settings. In *1st International Workshop on on Security Issues in Coordination Models, Languages and Systems*, volume 85.3 of *ENTCS*.

[Lucchi and Zavattaro, 2004] Lucchi, R. and Zavattaro, G. (2004). WSSecSpaces: a Secure Data-Driven Coordination Service for Web Services Applications. In *Proc. of ACM Symposium on Applied Computing (SAC'04)*, pages 487–491. ACM Press.

[Murphy et al., 2001] Murphy, A., Picco, G., and Roman, G.-C. (2001). A middleware for physical and logical mobility. In *21st International Conference on Distributed Computing Systems*, pages 524–533.

[Schneier, 1996] Schneier, B. (1996). *Applied Cryptography*. Wiley.

[Vitek et al., 2003] Vitek, J., Bryce, C., and Oriol, M. (2003). Coordinating Processes with Secure Spaces. *Science of Computer Programming*, 46:163–193.