Washington University in St. Louis
Washington University Open Scholarship

All Computer Science and Engineering Research | Computer Science and Engineering

Report Number: WUCSE-2002-31

2002-09-09

Secure Sharing of Tuple Spaces in Ad Hoc Settings**Please see WUCSE-03-26**

Gruia-Catalin Roman and Radu Handorean

Practical applications of coordination models demand appropriate security guarantees. In ad hoc settings this must be achieved without reliance on any central point of control. \lime is one of the few coordination models and middleware to provide support for ad hoc networking and mobility. This paper shows how security can be added to \lime by simple extensions to the original model. The extensions include password protected tuple spaces, per tuple access controls and encrypted communication between parts of application running on different hosts. Furthermore, these new capabilities are accommodated with minimum changes to the original design.
... Read complete abstract on page 2.

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research

Recommended Citation
Roman, Gruia-Catalin and Handorean, Radu, "Secure Sharing of Tuple Spaces in Ad Hoc Settings**Please see WUCSE-03-26**" Report Number: WUCSE-2002-31 (2002). *All Computer Science and Engineering Research.*
https://openscholarship.wustl.edu/cse_research/1148

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

This technical report is available at Washington University Open Scholarship: https://openscholarship.wustl.edu/cse_research/1148

Secure Sharing of Tuple Spaces in Ad Hoc Settings**Please see WUCSE-03-26**

Gruia-Catalin Roman and Radu Handorean

Complete Abstract:

Practical applications of coordination models demand appropriate security guarantees. In ad hoc settings this must be achieved without reliance on any central point of control. \lime is one of the few coordination models and middleware to provide support for ad hoc networking and mobility. This paper shows how security can be added to \lime by simple extensions to the original model. The extensions include password protected tuple spaces, per tuple access controls and encrypted communication between parts of application running on different hosts. Furthermore, these new capabilities are accommodated with minimum changes to the original design.

Department of Computer Science & Engineering

Washington University in St. Louis
School of Engineering & Applied Science

2002-31

Secure Sharing of Tuple Spaces in Ad Hoc Settings**PLEASE SEE WUCSE-03-26**

Authors: Roman G.C.; Handorean R.

September 9, 2002

Abstract: Practical applications of coordination models demand appropriate security guarantees. In ad hoc settings this must be achieved without reliance on any central point of control. \lime is one of the few coordination models and middleware to provide support for ad hoc networking and mobility. This paper shows how security can be added to \lime by simple extensions to the original model. The extensions include password protected tuple spaces, per tuple access controls and encrypted communication between parts of application running on different hosts. Furthermore, these new capabilities are accommodated with minimum changes to the original design.

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160

Secure Sharing of Tuple Spaces in Ad Hoc Settings

Gruia-Catalin Roman and Radu Handorean
Washington University in St. Louis
Campus Box 1045, One Brookings Drive
St. Louis, MO 63130-4899, USA
{roman, raduh}@cse.wustl.edu

Abstract

Practical applications of coordination models demand appropriate security guarantees. In ad hoc settings this must be achieved without reliance on any central point of control. LIME *is one of the few coordination models and middleware to provide support for ad hoc networking and mobility. This paper shows how security can be added to* LIME *by simple extensions to the original model. The extensions include password protected tuple spaces, per tuple access controls and encrypted communication between parts of application running on different hosts. Furthermore, these new capabilities are accommodated with minimum changes to the original design.*

1 Introduction

As distributed applications continue to grow, so do their needs for quality of service. Deadlines, availability and security are only some of the quality of service attributes expected from the various components of a distributed application. Since distributed applications inherently require communication between remote components, the security of interactions is a basic requirement. Given that various parts of a distributed application can be developed by different programmers, executed for different users, and can travel through the networks, accessing resources and interacting with each other, security mechanisms are needed in order to control and limit, when necessary, accessibility to resources and information.

In wired networks security issues are often resolved by appealing to specialized services on a central machine. Ad hoc networks cannot employ such solutions. In ad hoc settings the computational environment consists of a group of devices that communicate only when in proximity to each other. No central database is available to store names, passwords, access rights and so forth. Furthermore, frequent disconnections present us with additional challenges since communication cannot be guaranteed for a long period of time. Protection techniques must be adjusted to take into account these factors.

While the original Linda[6] model has numerous implementations and extensions, few of them address ad hoc networking. One such implementation is LIME [11], which offers strong support for coordination in ad hoc networks. The computational unit is called an agent. The model allows for multiple tuple spaces per agent, identified by names. Agents on different hosts within communication range can share tuple spaces with the same name. When connected, transient tuple space sharing implemented by LIME makes the content of tuple spaces with the same name available to all others as if local to each agent. In the initial release of LIME, agents on different hosts coordinate their activities via public tuple spaces. The information being communicated is available to any agent interested in a particular tuple space.

Security enforcement is needed to protect the easily accessible tuple space information content from tampering or unauthorized usage. In general, security can be achieved by implementing *authentication*, *cryptography*, and *access control* into the underlying middleware supporting the application.

Cryptography is a mechanism used to protect stored and exchanged information from potential unauthorized readers. The information is made illegible by encryption so that, even if available to everyone, it is meaningless for all those who cannot decrypt it. Encryption and decryption involve the use of keys or passwords. The distribution of keys and passwords in ad hoc systems is a very complicated issue and for simplicity reasons remains outside the scope of this paper.

Access control checks if a certain type of interaction is permitted between the entities involved (i.e., if the entities are allowed to perform the action they intend

to). For example, in the UNIX file system, different files on the hard disk can have different access policies for different users. Users are separated in different categories and each category has specific access rights to each file (e.g., read only, read-write, and so forth). This helps prevent unauthorized information access (like the UNIX case) or protects the encrypted information from being destroyed by an attacker that cannot decrypt it. In an ad hoc setting, an application interacts with other applications controlled by users whose identities cannot be readily verified. A distributed implementation that relies only on local information for user identification is preferable.

Authentication is the mechanism used to verify that an entity is indeed who it claims to be when trying to access resources or information. Since this involves a third (neutral and trusted) computing base (authentication server, database, etc.), in our approach if an agent meets the criteria imposed by the first two mechanisms, we consider it entitled to access the information.

In distributed systems various applications may need different security constraints. Cryptography and access control can be used separately or in combination where necessary. They have been added to the LIME coordination model in order to secure the interaction between entities, particularly the contents of tuple spaces. To achieve this we designed and developed password protected tuple spaces, secure communication between hosts, and tuple level access policies.

The reminder of the paper is structured as follows: Section 2 reviews the LIME coordination model. Section 3 presents extensions to the original model for secure coordination. Section 4 contains implementation issues. In Section 5 we discuss the implications of our work. Related work on secure coordination is presented in Section 6. We draw conclusions in Section 7.

2 The Lime Coordination Model

The LIME (version 2.0) middleware supports the development of applications exhibiting physical mobility of hosts, logical mobility of agents, or both. An agent is any piece of software that accesses tuple spaces and performs computational tasks. Hosts are containers for these agents. Hosts can move in physical space, while agents can move from host to host. LIME implements a coordination model inspired by the original Linda model. The context for computation, represented in Linda by a globally accessible, persistent tuple space, is represented in LIME by transient sharing of tuple spaces associated with each individual mobile agents. LIME also extends Linda tuple spaces with a notion of location and with the ability to react to a given state.

Transparent Context Maintenance. The model underlying LIME accomplishes the shift from a fixed context to a dynamically changing one by distributing the global Linda tuple space across multiple tuple spaces, each local to a mobile agent, and by introducing rules for transient sharing of the individual tuple spaces based on naming and connectivity; LIME also allows for multiple tuple spaces local to an agent. These tuple spaces are differentiated by name and are shared with other agents that share local tuple spaces with the same name, forming a federated (global) tuple space. From a mobile agent's point of view, the only way to access a global tuple space is through an *interface tuple space* (ITS), permanently and exclusively attached to the agent itself. The ITS contains tuples the mobile agent is willing to share with other agents. This represents the only context accessible to the agent when it is alone. This tuple space is *transiently shared* with similarly named ITSs belonging to mobile agents that are connected. Hence, the content perceived through the ITS changes dynamically in response to changes in the set of mobile agents sharing a tuple space. Access to the ITS takes place using the Linda primitives (e.g., **in**, **rd**, **out**), whose semantics are basically unaffected. Also, LIME offers non-blocking versions of **in** and **rd** in the form of probe variants of the same operations (e.g., **inp**, **rdp**). LIME also includes several extensions of the traditional Linda model designed to handle groups of tuples (e.g., **ing**, **rdg** and **outg**) as well as their non-blocking variants (e.g., **ingp** and **rdgp**, where applicable). While the basic operations return a matching tuple (if available) or null otherwise (if nonblocking), the group operations return all matching tuples (or null if none available and the call is non-blocking).

Another extension to the traditional Linda model is a new pattern matching mechanism. Each field in a tuple has a name, a type and a value. Since each field is identified by name, the tuples are no longer an ordered structure but a set of named values. Templates use field names to refer to the content of a tuple. The values of the named fields can be filtered using predicates in templates. A match will be declared between a tuple and a template if the tuple contains fields to satisfy each predicate in the template (i.e., the tuple has a field for each field of the template with the same name, the type of this field is the same or a subtype of the field in the template, and the value of the field satisfies the constraint specified in the template field). Templates need to specify only a subset of the fields for a matching operation to succeed (e.g., a template that specifies 3 fields can match a tuple with 6 fields if 3 of the tuple's fields satisfy the requirements expressed

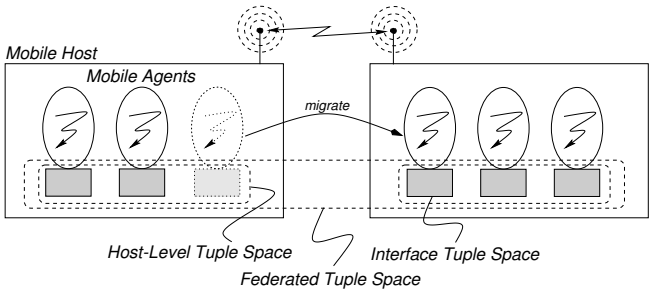

Figure 1. Transiently shared tuple spaces encompass physical and logical mobility.

in the template). LIME allows for polymorphic tuple matching, i.e., a field in the template provided for pattern matching will match the respective field of a tuple if the latter contains an object of the same type or of a subtype of the one specified in the template. This allows for wild cards in pattern matching (e.g., the Java Object class in a template field will match any field in a tuple) and for the use of interfaces to retrieve objects that implement them.

Encompassing Physical and Logical Mobility. In an ad hoc network, LIME mobile hosts are connected when the distance between them allows communication to take place. In LIME, activities associated with arrival or departure of a host are called engagement and disengagement, respectively. Mobile agents are connected when they are co-located (i.e., reside on the same host), or they reside on hosts that are connected. Creation and termination of mobile agents is a special case of connection and disconnection, respectively. Figure 1 depicts the LIME model. Mobile agents are the only active components; mobile hosts are roaming containers providing connectivity and execution support for agents. In other words, mobile agents are the only components that carry "concrete" tuple spaces with them. Multiple agents' tuple spaces that are transiently shared across hosts form a *federated tuple space*.

Controlling Context Awareness. LIME fosters a style of coordination that reduces the details of distribution and mobility to changes in what is perceived to be a local tuple space. This view is powerful as it relieves the designer from specifically addressing the changes in configuration, but some mobile applications need to explicitly address the distributed nature of the data for performance or optimization reasons. LIME provides such fine-grained control over the context perceived by the mobile unit by extending Linda operations with tuple location parameters that define projections of the transiently shared tuple space. LIME expresses tuple location parameters in terms of agent identifiers and host identifiers. These identifiers can be used both to place tuples at a particular agent location or to restrict queries to specific agents or hosts.

The read-only `LimeSystemTupleSpace` tuple space provides awareness of the system configuration. Its tuples contain information about the mobile agents present in the community, physical hosts they execute on, and tuple spaces created for coordination. Standard tuple space operations on the `LimeSystemTupleSpace` tuple space allow an agent to respond to the arrival and departure of other agents and hosts.

Reacting to Changes in Context. Mobility entails a highly dynamic environment, where reacting to changes constitutes a major fraction of the application design. Therefore, LIME extends the basic Linda tuple space with the notion of *reaction*. A reaction $\mathcal{R}(s, p)$ is defined by a code fragment s that specifies the actions to be executed when a tuple matching the pattern p is found in the tuple space. After each operation on the tuple space, LIME non-deterministically selects a reaction and compares the pattern p against the tuple space contents. If a matching tuple is found, s is executed, otherwise the reaction is a skip. This selection and execution proceeds until there are no reactions enabled, and normal processing resumes. Thus, reactions are executed as if they belonged to a separate reactive program which runs to fixed point after each non-reactive statement. Blocking operations are not allowed in s, as they could prevent the program from reaching fixed point.

Reactions in LIME come in two forms: *strong reactions* and *weak reactions*. Strong reactions execute atomically with the writing of the tuple that enables them. These reactions are not allowed over the entire federated tuple space; they must always be restricted to a host or agent. Otherwise, maintaining the requirements of atomicity and serialization imposed by strong reactive statements would require a distributed transaction encompassing multiple hosts for every tuple space operation. Location parameters have to be used to define the projection of the tuple space on which the reaction will be installed and executed. LIME also provides the notion of *weak reaction*. Processing of a weak reaction proceeds as in the case of strong reactions, except that the execution of s does not happen atomically with the detection of a tuple matching p; instead, it is guaranteed to take place eventually if connectivity is preserved. This eliminates the need for a distributed transaction and allows this type of reaction to be installed and to execute over the entire tuple space.

Maintaining Group Membership in Highly Dynamic Contexts. Environments characterized by frequent disconnections can seriously affect system performance and maintenance. For this reason, the initial version of LIME made the simplifying assumption of announced disconnection. This allowed operations to complete and communication to stop before the actual disconnection occurs. The current version of LIME protects applications from the complexity associated with sudden disconnection by using location information in the engagement/disengagement protocol.

The concept of safe distance [5] was introduced to help predict disconnections. Two hosts are considered to be at a safe distance if given the speed of the two hosts, any task in progress is guaranteed to complete before any disconnection can occur. The safe distance is usually a fraction of the range of the wireless transmitter. Once the safe distance is exceeded, an automatic disengagement protocol is triggered and the group is split. The safe distance ensures that no messages between group members are lost and that messages are sent and received in the same configuration. This helps prevent inconsistent states when the actual disconnection occurs. When a host approaches a group, it is allowed to engage the group only after it comes within safe distance of some member of the group. This ensures a clean and consistent group membership management, in case the host decides to move away from the group soon after engagement. (Since ad hoc routing is not included in the current software release, the groups are clusters of fully connected hosts. However, the use of ad hoc routing would not affect the LIME software in any way, only the data transport protocol. Work on including ad hoc routing in a future release is in progress.)

Software Distribution. LIME is available under a GNU's LGPL open source license. Source code and development notes may be obtained from `lime.sourceforge.net`.

In the following sections, we show how the design of specialized coordination middleware takes advantage of the rich set of features offered by LIME.

3 Security Extensions

3.1 Password Protected Tuple Spaces

Why it is needed. Although LIME implements the Linda coordination model, it has some notable extensions. One important extension is the possibility to use multiple tuple spaces identified by different names, as opposed to the single, globally available, tuple space in Linda. LIME uses `LimeSystemTupleSpace` for internal management purposes. This tuple space contains tuples that represent agents, hosts and tuple spaces. However, the information in this tuple space is read-only available to an agent that wishes to look into it. By looking into this tuple space an agent can obtain the names of all tuple spaces created and shared by all other agents in the group. By simply knowing the name of a tuple space, any agent can create a local tuple space with the same name, share it with everybody else who has such a local tuple space shared and have access to all the information in the entire federated tuple space. The polymorphic pattern matching that LIME supports makes it even easier to tamper with the tuple space content.

How we provide it. The first extension towards secure coordination protects tuple spaces from unauthorized access. We consider an agent to be authorized to access the tuple space information if it has the name of the tuple space and the password that protects it. This makes the name of the tuple space the key to all the information in that tuple space. To protect the information means to protect the name of the tuple space. The `LimeSystemTupleSpace`, among other information, contains tuples that identify every tuple space (by name). Since the name is available in `LimeSystemTupleSpace`, the first step is to make the information obtained from `LimeSystemTupleSpace` unusable in its raw form. Changes are required to ensure that extracting the name of a tuple space from the `LimeSystemTupleSpace` will no longer provide enough information for an agent to create a tuple space with the same name and share it with other agents gaining access to information this way. To achieve this, some processing of the tuple space name will be done on the way from the constructor call, when creating the tuple space, to the internal storage of the name inside the system. The information available in `LimeSystemTupleSpace` will be the processed name of the tuple space. We make sure this information cannot be used in its form from `LimeSystemTupleSpace` and also that it cannot be generated incorrectly. For this reason tuple spaces are split in two categories: tuple spaces that we want to protect and tuple spaces that are freely accessible (i.e., unprotected). If the user creates a tuple spaces that is intended to be secure, the user will have to provide a password. If no password is provided the tuple space is assumed to be public (i.e., unprotected). For secure tuple spaces, the password is used to encrypt the name before marking it as a secure tuple space name and forwarding it to the previous implementation of LIME which will use it as if it were a regular string representing a name of a tuple space that will be used for sharing.

3.2 Tuple Level Access Control

Why it is needed. Currently, once a tuple space is shared, all tuples in that tuple space are available to all agents sharing a tuple space with the same name (these agents can be co-located or can run on different hosts within communication range). Any such agent can read and remove any tuple from the federated tuple space, no matter where that tuple is located, as long as the local tuple space that contains the tuple is shared. As we have seen in the previous section, we can protect tuple spaces by using passwords. This means that if an agent wants to remove a tuple, it must have already had access to that tuple space (using the right public name and password or just the name if the tuple space is not protected). Therefore, an attack that targets the removal of a tuple to provoke a denial of service from the agent that published that tuple is less likely to take place. However, the polymorphic tuple matching mechanism leaves plenty of room for programming mistakes if not used correctly. An agent executing an **in** with a template that contains two Object formals will remove any tuples with two fields from the federated tuple space, even though some of them may not be really needed by the agent that called the **in**. Since that agent is not likely to write those tuples back and even less likely to write them back to the local tuple spaces of the agents from where they were retrieved, they can be lost for the community of agents. Should this scenario happen, the original owner of the tuple(s) will not be notified and will not know that the tuple is no longer there. If that tuple represents a service that the agent wants to advertise or some other important information about the owner agent, its (accidental) removal may have negative impact on the interaction among agents and the application progress.

How we provide it. To prevent these accidents from happening, we designed and implemented an access mechanism based on access rights. The approach is somewhat similar to the file access policy in UNIX like operating systems. The community of agents is split in two: the owner and the others. We refer to the agent that writes the tuple in its own local (and shared) tuple space as the owner. This agent owns the tuples in its local tuple space. For the tuples in this local tuple space, all other agents are considered to be part of the "others" group.

The tuple space access represents writing, reading or removal of one or more tuples. Operations that handle groups of tuples and reactions reduce eventually to the three operations listed above. Given a federated tuple space writing a tuple to a contributing tuple space is always permitted and so is reading a tuple. If an agent needs to write tuples to a tuple space and does not want any other agent to read it, the agent can use a private tuple space for its own computation, not shared with anyone else. Since this is possible in LIME, no changes are needed. Only the removal of tuples needs to be controlled. We desire to allow the owner of a tuple to remove it while other agents may be restricted from removing it. This can be accomplished by marking the tuple read-only. Figure 2 shows the execution of an **in** and **inp** on the tuple space local to the issuer of the operation and on other tuple spaces shared by other agents and having the same name. In Figure 3 the execution of an **ing** operation is similarly depicted. Locally, all tuples are returned (assuming that they all match the template, regardless of access control policy). For an operation initiated by an agent different from the owner of the tuples, the result is filtered and only the non read-only tuples are returned.

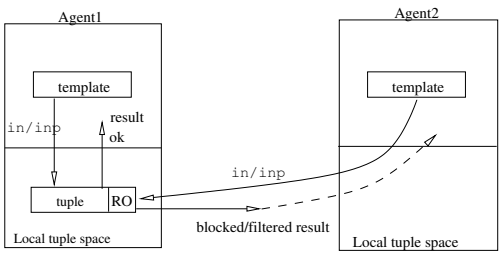

Figure 2. The execution of an **in** or **inp** on a read-only tuple local to the issuer of the call of in another agent's tuple space.

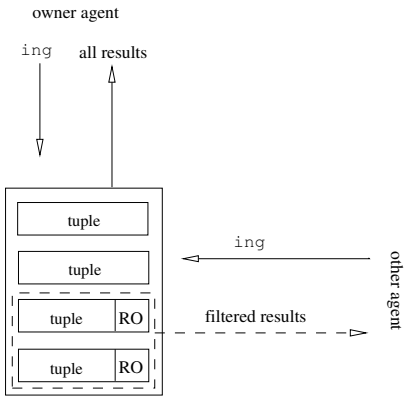

Figure 3. The execution of an **ing** on a group of read-only and fully accessible tuples.

In LIME, tuples are augmented with location information. When a tuple is written to a tuple space, it has a current location, and a destination location (which is used to migrate tuples to different agents or hosts). The current location contains information which identifies the agent in whose local tuple space the tuple currently resides. This agent is also the owner of the tuple. Upon migration, which takes place upon engagement or when the tuple is created, the location information changes and so do the access rights. The new owner of the tuple obtains full access to this tuple while the former owner becomes one of the "others". If the new agent wants to change the access policy for this particular tuple, it can remove it and reinsert it with the new access policy.

Since the location information changes upon migration, we added information about the creator of a tuple in the class that implements the tuple. The creator information is very important for templates. This will identify the agent that called the **in** operation (or any of its variations). The tuples marked read-only will not be returned as results for calls originating from other agents than the owner. This means that an **in** will remain blocked if the only matching tuples are read only, an **inp** will return null if no other tuple is found and an **ing** will filter out the read-only tuples if their scope is a tuple space owner by an agent different from the agent that issued the calls.

3.3 Secure Remote Access

Why it is needed. In LIME, tuple spaces with the same name are shared forming a federated tuple space that can span multiple hosts. Consequently, an operation can involve local and/or remote (federated) tuple space access. If the call involves remote execution, the parameters will be sent across the network and the results will be sent back over insecure lines.

How we provide it. To protect the communication we encrypted the parameters and the results with the password used when the tuple space was created (if any). The remote party is supposed to have the password since it shares a tuple space with the same (encrypted) name and it received the call. If the password is correct, the remote party will be able to decrypt the parameters and the command will be executed correctly. The involved parties can decide on a password over a secure channel (over the phone or, as seen in section 5, using public key encryption and read only tuples). Backward compatibility is preserved by allowing the use of unprotected tuple spaces. However, secure tuple spaces are protected from calls originating on hosts that use older (insecure) versions of LIME. Older APIs will send the requests unencrypted and and will not be served if the tuple spaces they refer to are password protected (an exception will be thrown in this case). The return of results from the tuple space will be handled in the same manner. Figure 4 shows how interceptors secure the communication between two hosts.

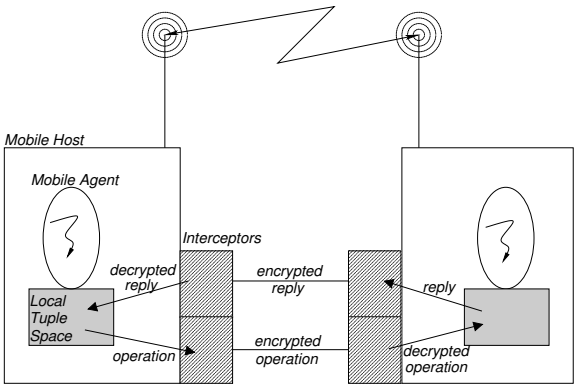

Figure 4. Interceptors catch messages and encrypt them before sending and then decrypt them upon receipt.

4 Implementation

4.1 Password Protected Tuple Spaces Implementation

The interface the programmer uses to create secure tuple spaces is very similar to the interface offered by the previous version of LIME. The difference is that tuple spaces (secure or not) are created using the SecureLimeTupleSpace class. While the constructors still exist in their previous form, a new one was created, with an extra parameter: the password (Figure 5). If no password is provided, a simple, unprotected tuple space will be created, like in the previous version of LIME. An important moment in the life of a tuple space is its creation. This is the only moment when the agent explicitly uses the password. Once the agent has the handle to the tuple space, it does not need the password anymore. The tuple space handle will enable the agent to access the tuple space for as long as the agent has it without having to provide the password. All operations will be called as before and will use the password transparently to the agent if needed (see Secure Remote Access Implementation Section). A tuple space operation can only be called by a LIME agent. Moreover, another important observation here is that the handle of a tuple space can only be used by the

agent that created it. When an operation is called on a tuple space, LIME verifies that it was called by the thread representing the agent that created it. Even if the handle of a tuple space is obtained correctly by an agent, it cannot be transferred and used by another agent. This is why it is not necessary to ask for the password when a tuple space operation is called.

The name of the secure tuple space is obtained from the provided name and password. This encrypted name appears in the `LimeSystemTupleSpace`. The tuple space name (encrypted name when a password is provided or the plain clear name if the tuple space is not meant to be protected) will be prefixed by a differentiator: letter "U" for unencrypted or "S" for secure tuple space. The tuple space "blue" is different form the tuple space "blue" + "pwd", where "pwd" is a password used to protect the second tuple space. They can coexist but no sharing takes place. The prefixes ensure that a tuple space cannot be created incorrectly. Since they are internally added, they cannot be manipulated by agents. Reading the name of a (secure) tuple space from `LimeSystemTupleSpace` will not be enough to create an insecure tuple space with the same name. The prefixes also address the case when the result of encrypting the clear name of a tuple space coincides with the name of an unencrypted tuple space (before adding prefixes).

Internally, the LIME server has a **SecurityTable** that stores entries of the form [encrypted name, password]. An entry is added to this table every time a new secure tuple space is created. When an operation is executed on the tuple space, if it runs on the local host of the issuer (identifiable by location parameters that define the projection of the tuple space) no further verification is needed. For executions of tuple space operations that span beyond the limits of issuer's host, the table will be used for more verifications. See Section 4.3 for details.

4.2 Tuple Level Protection Implementation

Even though tuple space access can now be protected by passwords, in some cases even finer grain control may be needed. To implement read-only tuples, several changes were needed to the previous version of LIME and to Lights, the tuple space implementation that LIME uses. First, the class Tuple was enhanced with a new constructor (Figure 6). This constructor specifies what access policy should be applied for this tuple, specifically if the tuple will be marked read-only or not. To enforce the access policy, the Tuple class was also enhanced with a new member called "owner". This is of type AgentLocation and identifies the agent that creates a tuple or a template. This information cannot be handled by the programmer and is filled in by the LIME system when a tuple is created and written to the tuple space. When any of the **in**, **inp** or **ing** is called, the creator information of the template (which always identifies the agent that issued the operation) is verified against the current location information of the tuple being checked for matching. If the call originates from an agent different from the owner of the tuple, and if the tuple is masked as read-only, then the tuple will not be declared a match for the template. The tuple can, however, be read using any of the **rd**, **rdp** or **rdg** operations and a matching template.

By enforcing different matching policies, different levels of tuple accessibility can be achieved. A tuple can be password protected if it contains a field that stores a password and if the matching policy requires that the template provides the exact password. To do so, the polymorphic matching has to be disabled for the field that stores the password and exact value match has to be enforced. Differently from the read-only tuples, this mechanism can allow or block the reading of tuples as well as their removal.

Implementation wise, a tuple is an ordered array of fields (which have a type and a value). The various field matching policies demanded changes in the Field class. The class was enhanced with a new member that specifies what kind of matching applies for each particular field. When fields are added to a tuple, the type of matching can be specified for each of them. Figure 7 shows how fields are added to tuples and how to specify the matching policy for each of them. Constants.EV, Constants.ET and Constants.PT are predefined integer constants that identify the **E**xact **V**alue, **E**xact **T**ype, and **P**olymorphic **T**ype matching policies.

For **E**xact **V**alue matching both the type and the value have to match exactly. **E**xact **T**ype policy allows for formals in the patters. The type, however, has to be exactly the type of the respective field in the tuple. **P**olymorphic **T**ype is the most flexible policy. This allows for wild cards in field matching (formals of type Object). Figure 8 shows what different templates match the tuple $\langle Integer(5) \rangle$, depending on the type of matching requested when the tuple was created.

There is an important difference between the implementation of read-only tuple access and the field matching policies. The field matching policies apply the same for both the owner of a tuple as well as for any other agent,i.e., it does not matter what type of call triggered the matching mechanism. For read-only tuples, both the type of operation and the location information influence the matching policy (results of read operations will not be verified while results of opera-

SecureLimeTupleSpace(java.lang.String name, java.lang.String password)
— creates a new secure tuple space using the public tuple space name and the password. This call places an entry in the **SecurityTable** mapping the mangled name to the password.

Figure 5. The Call that Creates a Secure Tuple Space

Tuple(int accessControl)
— creates a new tuple with the specified access policy. The acessControl parameter defines the visibility of the tuple for agents other than the one that owns it with respect to removal.

Figure 6. The Call that Creates a Tuple with a Specified Access Policy

Tuple t = new Tuple();
t.addActual(new Integer(1975), Constants.EV).addActual(new String("WashU"));
— adds fields to a tuple.
To match this tuple, a template will need to have an **E**xact **V**alue on its first field (that is an actual of type Integer and value 5). Since the second field doesn't have any matching policy specified, the **P**olymorphic **T**ype is assumed. That is any formal of type String (or a supertype) would match the tuple.

Figure 7. Adding Fields and Matching Policy to a Tuple

	Exact value	Exact type	Poly type
$\langle Integer(5) \rangle$	yes	yes	yes
$\langle Integer \rangle$	no	yes	yes
$\langle Object \rangle$	no	no	yes

Figure 8. Various answers of the matching mechanism to different tuples under different matching policies

tions that involve the removal of tuple(s) may need to be filtered.

4.3 Communication Level Protection Implementation

Remote operations are executed on some projection of the federated tuple space defined by location parameters, which include hosts different than the one on which the requesting agent runs. The messages that carry operation requests are of two types: messages that carry reactions and messages that carry regular tuple space operations. When an agent executes a call that spans beyond the limits of the current host, an interceptor catches it, analyzes the tuple space that the message refers to and takes the appropriate action (the use of the interceptor pattern [17] is natural for this case, when we add security to a system that in its initial design did not address this issue). The interceptor verifies if the name of the tuple space is present in the **SecurityTable**. If the message refers to an unprotected tuple space (it is not in the table), the interceptor lets it to pass through unchanged. If the tuple space is a secure one, the interceptor will extract from the table the password that corresponds to that tuple space and will use it to encrypt the message. The interceptor creates a packet that contains the encrypted message and the encrypted name of the tuple space the message refers to and forwards this packet to the other involved hosts. On the recipient's side, actions happen symmetrically. Another interceptor will catch the incoming the message will lookup the name of the tuple space in the local **SecurityTable** and if found, will use the corresponding password to decrypt the message. The message is then forwarded to the `LimeServer`. If the target tuple space is not a secure one, the name will not be found in the **SecurityTable** and will be forwarded unchanged to the `LimeServer`. The return of results is handled the same way.

5 Discussion

There is an essential conflict between the flexibility and ease of access to information and the expres-

sive power of an access control model. By designing and implementing this extension to the original LIME model, we sought to achieve a balance between the two and to offer the programmer the possibility to choose the level of access control. The locking and matching mechanisms we provide make possible many constructions useful in building a safe distributed application.

For example, with the mechanisms we provide the use of public keys has become possible. When using public keys, a major security concern is to be able to tie the key to its real owner. Using the facilities offered by our implementation, the man-in-the-middle attack can be avoided. If an agent publishes its public key in a read-only tuple in its local tuple space, any agent that wants to use this public key to send a secret message to the first agent can easily verify that the agent is indeed the creator of the tuple or not. An attacker might write a tuple and have it migrate to the first agent's local tuple space (specifying the destination location of the tuple) but it will not be able to alter the information about the creator of this tuple. The second agent will be able to detect that the creator of the tuple is not the owner of that tuple and thus realize that this tuple is a trap. Once the advertising of public keys is secure, agents can exchange private keys to create safe communication channels. Even though they can communicate via password protected tuple spaces, this mechanism allows for session keys or changing the password used to create the secure tuple space the next time.

Another important aspect of this work is the possibility to protect applications against denial of service attacks. An application may offer services over the network to interested parties. To do so, it can advertise the service in a tuple written in the local tuple space. The tuple contains an object that represents a proxy of the service along with a profile that describes the service. Interested agents search the network for services that advertise a profile that matches their needs. This mechanism was introduced in Jini [4], [18], [12] and was adapted to ad hoc networks in [9]. In such a scenario, a malicious agent may consistently remove the advertisement for the service and the advertiser wouldn't even notice. Read-only tuples make this attack impossible.

The various field matching policies can also be used in service provision scenarios to ensure specific types of access to a proxy object. For example a server may deny service to all clients that do not ask specifically for its help (e.g., only "laser printer" may be too vague for a certain server which expects a more precise description like "color laser printer with 600 dpi").

The secure remote access protects the communication between hosts. The quality of protection depends on the algorithm used. In our implementation and experiments we used triple DES.

Read-only tuple spaces are currently under investigation. This adds another level of security by allowing an agent to protect itself against being flooded with tuples from a malicious agent. Furthermore, once read-only tuple spaces become available, more scenarios can be developed under the new circumstances. The guarantee that the content of a tuple space was generated by its owner can have multiple beneficial implications. Public keys and services can be published in such a tuple space without worrying about the man-in-the-middle attack and eliminating the necessity for other agents to verify the creator information of a tuple.

We place our work at a level above the physical level so we do not address issues related to radio communication interference or any other low level attacks.

6 Related Work

In an open environment such as a computer network and especially in presence of mobile code roaming across hosts, security is an important issue. Other projects also address this issue, trying to add different levels of protection to mobile agent systems and tuple space coordination of mobile agents. KLAIM (A Kernel Language for Agents Interaction and Mobility) [13] addresses the protection of data through the use of a capability based system combined with type hierarchy based system for access control. In Secure Spaces [19] the authors employ a finer grained approach to tuple matching mechanisms than the original Linda model. They go down to field level to address security. They can protect each field individually by locking it with a password. This is somehow similar to using exact value matching for specific fields in the matching mechanisms described in this paper. Agents can be stopped from learning from tuples by requesting them to provide exact information in the templates for tuple matching.

Several systems address the issue of protecting hosts from malicious agents. The D'Agents system [8] uses public key cryptography to authenticate incoming agents and thus increasing the security of hosts. The more difficult problem of protecting the agent form curious hosts led to the approach of computing with encrypted functions [15], [14]. The key idea here is that mobile agents are able to decrypt code and data only if certain conditions are met by the computing environment or at a specific moment.

In [16] the author proves that strong typing is an essential concept for achieving strong security properties. The access rights are stored in a typed access rights matrix inspired by the HRU model[10]. A capability based system adapted to distributed computing

is described in [7]. In Yalta [2] clients are logically grouped in dynamic coalitions. Yalta relies on certificates and certification authorities for emission, revocation and validation of certificates which leads to an architecture with several centralized hot points (certification authority and certification revocation service).

A distributed approach to trust management is described in [1]. Here, a trust relationship defined between exactly two entities is asymmetrical (unidirectional) and conditionally transitive. A direct trust relationship exists between two entities for collaboration reasons and a recommendation trust relationship is needed to spread trust information throughout the client. One type of relationship does not imply another (i.e., if Alice trusts Bob to recommend Cindy it does not imply that Alice trusts Bob to work with him).

Administrative domains [3], [20] restrict the execution environment by logically dividing it into nested levels. The scope of a user's operations can be limited to his/her domain and the movement of running code restricted to well determined areas.

7 Conclusions

In this paper we presented a way to add security capabilities to the LIME coordination model, to better control *who* can do *what* and *how* with *which* tuples. We have showed that simple changes can transform a coordination model in a secure platform suited for the development of secure applications. The mechanisms are general and can solve real issues in terms of secure coordination in ad hoc networks.

Acknowledgements

This research was supported in part by the National Science Foundation under Grant No. CCR-9970939. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] A. Abdul-Rahman and S. Hailes. A distributed trust model. In *New Security Paragidms Workshop*, pages 48–60. ACM Press, 1998.

[2] G. Byrd, F. Gong, C. Sargor, and T. Smith. Yalta: A secure collaborative space for dynamic coalitions. In *IEEE Workshop on Information Assurance and Security*, 1989.

[3] L. Cardelli and A. D. Gordon. Mobile ambients. In *Proceedings of Foundations of Software Science and Computation Structures (FoSSaCS)*, volume 1378, pages 140–155. Springer-Verlag, 1998.

[4] K. Edwards. *Core JINI*. Prentice Hall, 1999.

[5] R. G.-C., H. Q., and H. A. Consistent group membership in ad hoc networks. In *Proceedings of the 23rd International Conference in Software Engineering (ISCE)*, 2001.

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

[7] L. Gong. A secure identity-based capability system. In *IEEE Symposium on Security and Privacy*, pages 56–65, 1989.

[8] R. Gray, D. Kotz, G. Cybenko, and D. Rus. D'Agents: Security in a multiple-language, mobile-agent system. In *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 154–187. Springer-Verlag, 1998.

[9] R. Handorean and G. C. Roman. Service provision in ad hoc networks. In *Coordination Models and Languages*, volume 2315 of *Lecture Notes in Computer Science*, pages 207–219. Springer-Verlag, 2002.

[10] M. Harrison, W. Ruzzo, and J. Ullman. Protection in operating systems. *Communication of the ACM*, 19(8):461–471, August 1976.

[11] A. Murphy, G. Picco, and G.-C. Roman. LIME: A middleware for physical and logical mobility. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, pages 524–533, April 2001.

[12] J. Newmarch. *Guide to Jini Technologies*. http://jan.netcomp.monash.edu.au/java/jini/tutorial/Jini.xml, 2001.

[13] R. De Nicola, G. L. Ferrari, and R. Pugliese. KLAIM: A kernel language for agents interaction and mobility. *Software Engineering*, 24(5):315–330, 1998.

[14] J. Riordan and B.Schneier. Environmental key generation towards clueless agents. In *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 15–24. Springer-Verlag, 1998.

[15] T. Sander and C. F. Tschudin. Protecting mobile agents against malicious hosts. In *Mobile Agent Security*, Lecture Notes in Computer Science, pages 44–60. Springer-Verlag, 1998.

[16] R. S. Sandhu. The typed access matrix model. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 122–136, 1992.

[17] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern Oriented Software Architecture*, volume 2. John Wiley & Sons, Ltd., 1999.

[18] Sun Microsystems, Inc. Jini(tm) technology core platform specification. Technical report, Sun Microsystems, Inc., 2000.

[19] J. Vitek, C. Bryce, and M. Oriol. Coordinating agents with secure spaces. In *Proceedings of Coordination '99*, Lecture Notes on Computer Science. Springer Verlag, May 1999.

[20] J. Vitek and G. Castagna. Seal: A framework for secure mobile computations. In *ICCL Workshop: Internet Programming Languages*, pages 47–77, 1998.