View metadata, citation and similar papers at core.ac.uk brought to you by CORE
provided by IMT Institutional Repository

A Java Middleware for Guaranteeing Privacy of Distributed Tuple Spaces[⋆]

Lorenzo Bettini Rocco De Nicola

Dipartimento di Sistemi e Informatica, Università di Firenze
Via Lombroso 6/17, 50134 Firenze, Italy
{bettini,denicola}@dsi.unifi.it

Abstract. The tuple space communication model, such as the one used in *Linda*, provides great flexibility for modeling concurrent, distributed and mobile processes. In a distributed setting with mobile agents, particular attention is needed for protecting sites and information. We have designed and developed a Java middleware, Klava, for implementing distributed tuple spaces and operations to support agent interaction and mobility. In this paper, we extend the Klava middleware with cryptographic primitives that enable encryption and decryption of tuple fields. We describe the actual implementation of the new primitives and provide a few examples. The proposed extension is general enough to be applied to similar Java frameworks using multiple distributed tuples spaces possibly dealing with mobility.

1 Introduction

A successful approach to concurrent programming is the one relying on the Linda coordination model [10]. Processes communicate by reading and writing *tuples* in a shared memory called *tuple space*. Control of accesses is guaranteed by requiring that tuples selection be *associative*, by means of pattern matching. The communication model is *asynchronous*, *anonymous*, and *generative*, i.e., tuple's life-time is independent of producer's life time.

The Linda model has been adopted in many communication frameworks such as, e.g., *JavaSpaces* [1] and *T Spaces* [9], and for adding the tuple space communication model to existing programming languages. More recently, distributed variants of tuple spaces have been proposed to exploit the Linda model for programming distributed applications over wide area networks [6, 2], possibly exploiting code mobility [7, 11]. As shown in [8], where several messaging models for mobile agents are examined, the *blackboard* approach, of which the tuple space model is a variant, is one of the most favorable and flexible.

Sharing data over a wide area network such as Internet, calls for very strong security mechanisms. Computers and data are exposed to eavesdropping and

⋆ This work has been partially supported by EU within the FET – Global Computing initiative project *MIKADO* IST-2001-32222, by MIUR project *NAPOLI* and by Microsoft project *NAPI*. The funding bodies are not responsible for any use that might be made of the results presented here.

manipulations. Dealing with these issues is even more important in the context of code mobility, where code or agents can be moved over the different sites of a net. Malicious agents could seriously damage hosts and compromise their integrity, and may tamper and brainwash other agents. On the other hand, malicious hosts may extract sensible data from agents, change their execution or modify their text [16, 12].

The flexibility of the shared tuple space model opens possible security holes; it basically provides no access protection to the shared data. Indeed there is no way to determine the issuer of an operation to the tuple space and there is no way to protect data: a process may (even not intentionally) retrieve/erase data that do not belong to it and shared data can be easily modified and corrupted. In spite of this, within the Linda based approaches, very little attention has been devoted to protection and access control.

In this paper we present a Java middleware for building distributed and mobile code applications interacting through tuple spaces, by means of cryptography. In this middleware, classical Linda operations are extended for handling encrypted data. Primitives are also supplied for encrypting and decrypting tuple contents. This finer granularity allows mobile agents (that are not supposed to carry private keys with them when migrating) to collect encrypted data, while executing on remote sites, and decrypt them safely when back at the home site.

The proposed extension, while targeted to our middleware for mobile agents interacting through distributed tuple spaces, Klava [3], is still general enough to be applied to similar Java frameworks using multiple distributed tuples spaces possibly dealing with mobility, such, e.g., [11, 1, 6]. Indeed, this extension represents a compromise between the flexibility and open nature of Linda and of mobile code, and the privacy of data in a distributed context.

2 Distributed Private Generative Communications

The Linda communication model [10] is based on the notion of *tuple space* that is a multiset of *tuples*. These are just sequences of items, called *fields* that are of two kinds: *actual fields*, i.e., values and identifiers, and *formal fields*, i.e., variables. Syntactically, a formal field is denoted with $!ide$, where ide is an identifier. Tuples can be inserted in a tuple space with the operation **out** and retrieved from a tuple space with the operations **in** and **read** (**read** does not withdraw the tuple from the tuple space). If no matching tuple is found, both **in** and **read** block the process that execute them, until a matching tuple becomes available. *Pattern-matching* is used to select tuples from the tuple space; two tuples match if they have the same number of fields and corresponding fields do match: a formal field matches any value of the same type, and two actual fields match only if they are identical (but two formals never match). For instance, if *Val* is an integer variable, then tuples (“foo”, “bar”, $!Val$) and (“foo”, “bar”, 300) do match. After matching, the variable of a formal field gets the value of the matched field; in the previous example, after matching, *Val* will contain the integer value 300.

The middleware we are presenting is based on Klava [3], a Java framework implementing Klaim (*Kernel Language for Agent Interaction and Mobility*) [7] that provides features for programming distributed applications with mobile code and mobile agents, relying on communication via multiple distributed tuple spaces. KLAIM extends Linda by handling multiple distributed tuple spaces: tuple spaces are placed on *nodes* (or *sites*), which are part of a *net*. Each node contains a tuple space and a set of processes, and can be accessed through its *locality*. Thus, classical Linda operations are indexed with the locality of the node they have to be performed at. A reserved locality, `self`, can be used to access the current execution site. Moreover in KLAIM processes are first class data, in that they can be transmitted and exchanged among sites, so that mobile code and mobile agent applications can be easily programmed.

For guaranteing privacy of data stored in tuple spaces we have extended KLAVA with some cryptographic primitives. In our view, this extension is a good tradeoff between the open nature of Linda (and of mobile code) and data privacy. In particular we aim at having this extension as smooth as possible, so that the original model is not perverted.

The basic idea is that a tuple may contain both clear text fields and encrypted fields. All the encrypted fields of a specific tuple are encrypted with a single key. This choice simplifies the overall design and does not harm usability of the system; it would be unusual that different fields of the same tuple are encrypted with different keys. Encrypted fields completely hide the encrypted contents that they embody: they even hide the type of the contents. This strengthens the secrecy of data (it is not even possible to know the type of sensible information).

In line with the open nature of the Linda model, our main intention is not to prohibit processes to retrieve data belonging to other processes, but to guarantee that these data be read and modified only by entitled processes. A shared tuple space is basically a shared communication channel: in such a channel information can be freely read and modified.

At the same time one of our aims is avoiding that wrong data be retrieved by mistake. Clear text fields of a tuple can be used as identifiers for filtering tuples (as in the Linda philosophy), but if a matching tuple contains encrypted fields, which a process is not able to decrypt, it is also sensible that the tuple is put back in the tuple space if it was withdrawn with an **in**. Moreover, in such cases, a process may want to try to retrieve another matching tuple, possibly until the right one is retrieved (i.e., a tuple for which it has the appropriate decryption key), and to be blocked until one is available, in case no such tuple is found.

Within our framework it is possible to

- use tuple fields with encrypted data;
- encrypt tuple fields with specific keys;
- decrypt a tuple with encrypted fields;
- use variants of the operations **in** and **read** (**ink** and **readk**) to atomically retrieve a tuple and decrypt its contents.

The modified versions of the retrieving operations, **ink** and **readk**, are based on the following procedure:

1. look for and possibly retrieve a matching tuple,

2. attempt a decryption of the encrypted fields of the retrieved tuple
3. if the decryption fails:
    (a) if the operation was an **ink** then put the retrieved tuple back in the tuple space,
    (b) look for alternative matching tuples,
4. if all these attempts fail, then block until another matching tuple is available.

Thus the programmer is relieved from the burden of executing all these internal tasks, and when a **readk** or an **ink** operation succeeds it is guaranteed that the retrieved tuple has been correctly decrypted. Basically the original Linda pattern matching mechanism is not modified: encrypted fields are seen as ordinary fields that have type `KCipher` (as shown in Section 3). It can be seen as an extended pattern matching mechanism that, after the structural matching, also attempts to decrypt encrypted fields.

In case mobile code is used, the above approach may be unsafe. Indeed, symmetric and asymmetric key encryption techniques rely on the secrecy of the key (in asymmetric encryption the private key must be kept secret). Thus, a fundamental requirement is that *mobile code and mobile agents must not carry private keys when migrating to a remote site* ("Software agents have no hopes of keeping cryptographic keys secret in a realistic, efficient setting" [16]). This implies that the above introduced operations **ink** and **readk** cannot be used by a mobile agent executing on a remote site, because they would require carrying over a key for decryption.

For mobile agents it is then necessary to supply a finer grain retrieval mechanism. For this reason we introduced also operations for the explicit decryption of tuples: a tuple, containing encrypted fields, will be retrieved by a mobile agent by means of standard **in** and **read** operations and no automatic decryption will be attempted. The actual decryption of the retrieved tuples can take place when the agent is executing at the home site, where the key for decryption is available and can be safely used. Typically a mobile agent system consists of stationary agents, that do not migrate, and mobile agents that visit other sites in the network, and, upon arrival at the home site, can communicate with the stationary agents.

Thus the basic idea is that mobile agents collect encrypted data at remote sites and communicate these data to the stationary agents, which can safely decrypt their contents. Obviously, if some data are retrieved by mistake, it is up to the agents to put it back on the site from where they were withdrawn. This restriction of the protocol for fetching tuples is necessary if one wants to avoid running the risk of leaking private keys. On the contrary, public keys can be safely transported and communicated. By using public keys mobile agents are able to encrypt the data collected along their itinerary.

Notice that there is no guarantee that a "wrong" tuple is put back: our framework addresses privacy, not security, i.e., even if data can be stolen, still it cannot be read. Should this be not acceptable, one should resort to a secure channel-based communication model, and give up the Linda shared tuple space model. Indeed the functionalities of our framework are similar to the one provided, e.g., by *PGP* [17] that does not avoid e-mails be eavesdropped and stolen, but their contents are still private since they are unreadable for those that do not own the right decryption key.

An alternative approach could be that of physically removing an encrypted tuple, retrieved with an **in**, only when the home site of the agent that performed the **in**, notifies that the decryption has taken place successfully. Such a tuple would be restored if the decryption is acknowledged to have failed or after a specific timeout expired. However, this approach makes a tuple's life time dependent on that of a mobile agent, which, by its own nature, is independent and autonomous: agents would be expected to accomplish their task within a specific amount of time. Moreover, inconsistencies could arise in case successful decryption acknowledgments arrive after the timeout has expired.

3 Implementation

Klava [3] is deployed as an extensible Java package, `Klava`, that defines the classes and the run-time system for developing distributed and mobile code applications according to the programming model of Klaim. In Klava processes are instances of subclasses of class `KlavaProcess` and can use methods for accessing a tuple space of a node: `out(t,l)`, for inserting the tuple `t` into the tuple space of the node at locality `l`, `read(t,l)` and `in(t,l)`, for, respectively, reading and withdrawing a tuple matching with `t` from the tuple space of the node at locality `l`. Moreover the method `eval(P,l)` can be used for spawning a `KlavaProcess P` for remote execution on site `l`. Some wrapper classes are supplied for tuple fields such as `KString`, `KInteger`, etc.

The extension of this package, CryptoKlava, provides the cryptography features described in the previous section. We have used the *Java Cryptography Extension* (*JCE*) [13], a set of packages that provide a framework and implementations for encryption, key generation and key agreement, and Message Authentication Code (MAC) algorithms. JCE defines a set of standard API, so that different cryptography algorithms can be plugged into a system or an application, without modifying the existing code. Keys and certificates can be safely stored in a *Keystore*, an encrypted archive.

CryptoKlava is implemented as a subpackage of the package `Klava`, namely `Klava.crypto`, so that it is self-contained and does not affect the main package. In the rest of this section we will describe the main classes of the package `Klava.crypto`, implementing cryptographic features.

The class `KCipher` is introduced in order to handle formal and actual fields containing encrypted data (it follows the Klava convention that wrapper classes for tuple items start with a `K`). Basically it can be seen as a wrapper for standard Klava tuple fields. This class includes the following fields:

```
protected byte[] encItem; // encrypted data
protected Object ref; // reference to the real tuple item
protected String alg; // enc−dec algorithm type
```

The reference `ref` will be `null` when the field is a formal field, or the field has not yet been decrypted. After retrieving a matching tuple, `encItem` will contain the encrypted data (that is always stored and manipulated as an array of bytes). After the decryption, `ref` will refer to the decrypted data. Conversely, upon creation of an actual field, `ref` will contain the data to be encrypted; after encryption, `encItem` will contain the encrypted data, while `ref` will be set to `null` (so that the garbage collector can eventually erase such clear data also from the memory). `alg` stores information about the algorithm used for encryption and decryption.

An actual encrypted tuple field can be created by firstly creating a standard KLAVA tuple field (in the example a string) and then by passing such field to an instance of class `KCipher`:

```
KString s = new KString("foo");
KCipher ks = new KCipher(s);
```

Similarly the following code creates an encrypted string formal tuple field (In KLAVA a formal field is created by instantiating an object from a KLAVA class for tuple fields – such as `KString`, `KInteger`, etc. – through the default constructor):

```
KString s = new KString();
KCipher ks = new KCipher(s);
```

`KCipher` supplies methods `enc` and `dec` for respectively encrypting and decrypting data represented by the tuple field. These methods receive, as parameter, the `Key` that has to be used for encryption and decryption, and `enc` also accepts the specification of the algorithm. These methods can be invoked only by the classes of the package.

The class `Tuplex` extends the standard KLAVA class `Tuple`, in order to contain fields of class `KCipher`, besides standard tuple fields; apart from providing methods for cryptographic primitives, it also serves as a first filter during matching: it will avoid that ordinary tuples (containing only clear text data) be matched with encrypted tuples. Once tuple fields are inserted into a `Tuplex` object, the `KCipher` fields can be encrypted by means of the method `encode`. For instance, the following code

```
KString ps = new KString("clear");
KCipher ks = new KCipher(new KString("secret"));
Tuplex t = new Tuplex();
t.add(ps); t.add(ks);
t.encode();
```

creates a tuple where the first field is a clear text string, and the second is a field to be encrypted, and then actually encrypts the `KCipher` field by calling `encode`. Also `encode` can receive parameters specifying the key and the algorithm for the encryption; otherwise the default values are used. `encode` basically calls the previously described method `enc` on every `KCipher` tuple field, thus ensuring that all encrypted fields within a tuple rely on the same key and algorithm.

As for the retrieval operation, this can be performed either with the new introduced operations, **ink** and **readk**, if they are executed on the local site

```
KString s = new KString();
KString sec = new KString();
KCipher ks = new KCipher(sec);
Tuplex t = new Tuplex();
t.add(s); t.add(ks);
ink(t, l);
Print("encrypted data is: " + sec);
```

or by first retrieving the tuple and then manually decoding encrypted fields:

```
... // as above
in(t, l);
...
t.decode();
Print("encrypted data is: " + sec);
```

Notice that in both cases references contained in an encrypted field (such as `sec`) are automatically updated during the decryption. The **ink** in the former example is performed at a remote site but this does not mean that the key travels in the net: as explained in the previous section, the matching mechanism is implicitly split into a retrieve phase (which takes place remotely) and a decryption phase (which takes place locally).

Operations **ink** and **readk** are provided as methods in the class `KlavaProcessx`, which extends the class `KlavaProcess` for standard processes. `KlavaProcessx` also keeps information about the `KeyStore` of the process and the default keys to be used for encryption and decryption. Obviously these fields are `transient` so that they are not delivered together with the process, should it migrate to a remote site. All these extended classes make the extension of KLAVA completely modular: no modification was made to the original KLAVA classes.

Finally, let us observe that, thanks to abstractions provided by the JCE, all the introduced operations are independent of the specific cryptography mechanism, so both symmetric and asymmetric encryption schemes can be employed.

4 An Encrypted Chat System

The chat system we present in this section is simplified, but it implements the basic features that are common to several real chat systems. The system consists of a `ChatServer` and many `ChatClients` and it is a variant of the one presented in [3] with the new cryptographic primitives. When a client sends a message, the server has to deliver the message to all connected clients. If a message is "private", it will be delivered only to the clients specified in the list sent along with the message.

Messages are normally delivered through the network as clear text, so they can be read by everyone:

– an eavesdropper can intercept the messages and read their contents;
– a misbehaving chat server can examine clients' messages.

Moreover, the messages might also be modified so that a client believes he is receiving messages from another client, while it would be reading messages forged by a "man in the middle".

While this is normally acceptable, due to the open nature of a chat system, nonetheless there could be situations when the privacy and integrity of messages is a major concern; for instance if two clients want to engage a private communication. This is a typical scenario where cryptography can solve the problem of privacy (through encryption).

In this example we implement a chat server and a chat client, capable of handling private encrypted messages:

- when the client wants to send a private message to a specific receiver, it encrypts the body of the message with a key;
- the server receives the message and simply forwards it to the receiver;
- the receiver will receive the message with the encrypted body and it can decrypt it with the appropriate key.

Notice that clients that want to communicate privately must have agreed about the specific key to be used during the private message exchange; this is definitely the case with symmetric keys. As for public and private key encryption the receiver can simply use its private key, to decrypt a message encrypted with its own public key.

A private message is represented by a tuple with the following format:

("PERSONAL", <*body*>, <*recipient*>, <*sender*>)

where <*recipient*> and <*sender*> are, respectively, the locality of the client the message is destined to and the locality of the issuer of the message. Basically, when a client wants to send a message with an encrypted body, it will have to perform the following steps:

```
Tuplex t = new Tuplex() ;
KCipher cryptMessage = new KCipher( message ) ;
t.add( new KString( "PERSONAL" ) );
t.add( cryptMessage ) ;
t.add( selectedUser ) ;
t.add( self ) ;
t.encode();
out( t, server ) ;
```

where `message` is the actual message body.

The server handles encrypted messages by retrieving them through the following actions (it will deliver the tuple without the field <*recipient*>, which is useless at this time):

```
KString message = new KString() ;
KCipher cryptMessage = new KCipher( message ) ;
Locality to = new PhysicalLocality() ;
Locality from = new PhysicalLocality() ;
```

```
Tuplex t = new Tuplex() ;
t.add( new KString( "PERSONAL" ) );
t.add( cryptMessage ) ;
t.add( to ) ;
t.add( from ) ;
in( t, self ) ;
```

and it delivers the message to the recipient as follows:

```
out( new Tuplex(new KString ("PERSONAL"), cryptMessage, from), to );
```

On the other hand, the receiver, which is always waiting for incoming messages, will read and decrypt a message (in one atomic step), by means of the operation **ink**:

```
KString message = new KString() ;
KCipher cryptMessage = new KCipher( message ) ;
KString from = new KString() ;
Tuplex t = new Tuplex() ;
t.add( new KString( "PERSONAL" ) ) ;
t.add( cryptMessage ) ;
t.add( from ) ;
ink( t, self ) ;
Print("Received message: " + message);
```

Both the server and the clients execute these operations within the loop for handling incoming messages.

5 Conclusions and Related Work

Since tuple space operations can be used both by local processes and by mobile agents, the extended operations, presented in this paper, address both the privacy of hosts and of mobile agents. We did not deal with key distribution explicitly that can be seen as an orthogonal problem. Digital signatures can be smoothly integrated in our framework and the pattern matching extended accordingly.

The work that is closer to ours is [4], which introduces the *Secure Object Space* (SecOS) model. This model is intended to extend Linda with fine-grained access control semantics. In SecOS all tuple fields are locked with a key, and each field must be locked with a different key. The basic idea is that a process, upon retrieving a tuple, can see only the fields for which he owns the corresponding key. The structure of a tuple does not influence pattern matching: due to an introduced *subsumption* rule, a template can match also a bigger tuple, and fields can be reordered during the matching. [5] proposes a similar, but richer framework, `SecSpaces`, where also resource access control and tuple space partitioning facilities are provided (orthogonal and complementary to our approach).

All these features tend to alter the original Linda model, while our principal aim is to provide an extension of the Linda communication model that can be smoothly integrated into the existing features, without significantly changing the original model. Moreover, neither SecOS nor SecSpaces handle code mobility, which is one of our main concerns.

Mobility imposes additional restrictions on the underlying model, e.g., requiring that agents do not carry private keys during migrations, and calls for alternatives such as explicit encryption and decryption mechanisms and a two-stage pattern matching. Indeed the problem of protecting an agent against a malicious host is even more complicated than that of protecting a host from a malicious agent (we refer to the papers in [14, 15]).

References

1. K. Arnold, E. Freeman, and S. Hupfer. *JavaSpaces Principles, Patterns and Practice.* Addison-Wesley, 1999.
2. K. Arnold, B. O'Sullivan, R. Scheifler, J. Waldo, and A. Wollrath. *The Jini Specification.* Addison-Wesley, 1999.
3. L. Bettini, R. De Nicola, and R. Pugliese. Klava: a Java Framework for Distributed and Mobile Applications. *Software – Practice and Experience*, 2002. To appear.
4. C. Bryce, M. Oriol, and J. Vitek. A Coordination Model for Agents Based on Secure Spaces. In P. Ciancarini and A. Wolf, editors, *Proc. 3rd Int. Conf. on Coordination Models and Languages*, number 1594 in LNCS, pages 4–20. Springer-Verlag, 1999.
5. N. Busi, R. Gorrieri, R. Lucchi, and G. Zavattaro. SecSpaces: a Data-driven Coordination Model for Environments Open to Untrusted Agents. In *Proc. of FOCLASA'02*, ENTCS. Elsevier, 2002.
6. P. Ciancarini and D. Rossi. Jada - Coordination and Communication for Java Agents. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems - Towards the Programmable Internet*, number 1222 in LNCS, pages 213–228. Springer, 1997.
7. 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, 1998.
8. D. Deugo. Choosing a Mobile Agent Messaging Model. In *Proc. of ISADS 2001*, pages 278–286. IEEE, 2001.
9. D. Ford, T. Lehman, S. McLaughry, and P. Wyckoff. T Spaces. *IBM Systems Journal*, pages 454–474, August 1998.
10. D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.
11. G. Picco, A. Murphy, and G.-C. Roman. Lime: Linda Meets Mobility. In D. Garlan, editor, *Proc. ICSE'99*, pages 368–377. ACM Press, 1999.
12. T. Sander and C. Tschudin. Protecting Mobile Agents Against Malicious Hosts. In Vigna [14].
13. Sun Microsystems. *Java Cryptography Extension (JCE), Refence Guide*, 2001.
14. G. Vigna, editor. *Mobile Agents and Security.* Number 1419 in LNCS. Springer, 1998.
15. J. Vitek and C. Jensen, editors. *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, number 1603 in LNCS. Springer-Verlag, 1999.
16. B. Yee. A Sanctuary For Mobile Agents. In Vitek and Jensen [15], pages 261–273.
17. P. Zimmermann. *The Official PGP User's Guide.* MIT Press, 1995.