Secure P2P programming on top of tuple spaces∗

F. Benigni1, A. Brogi1, J.L. Buchholz1, 2, J.M. Jacquet2, J. Lange1, 2, R. Popescu1
(1) Department of Computer Science, University of Pisa, Italy
(2) Department of Computer Science, University of Namur, Belgium

Abstract

A new programming model for secure (embedded) peer-to-peer systems has been recently proposed in the context of the European project SMEPP. In this paper we present the design and implementation of such a model on top of tuple spaces. More precisely, we show how the SMEPP service-oriented interaction primitives can be effectively implemented using SecureLime.

Keywords: Peer-to-peer systems, tuple spaces.

1 Introduction and motivations

The flexibility of the peer-to-peer model allows the design of scalable and robust applications in many situations where a client-server approach is not suited, especially in those situations provided by mobile ad hoc networks where devices dynamically join and leave networks whose topology is not known in advance. The drawback of such a flexibility is a harder management of device discovery and coordination. However, since many low-level issues occurring in developing peer-to-peer systems are recurrent, various middleware solutions (e.g., JXTA [9]) have been deployed in order to ease the development of peer to peer applications by abstracting from those issues. One such middleware, especially targeted at enabling secure peer to peer communication between embedded systems, is currently under development in the SMEPP (Secure Middleware for Embedded Peer-To-Peer Systems) European project [3].

The SMEPP middleware is based on a service-oriented model allowing a dynamic integration of functionalities as devices get connected to the network. It is focused on security, so that SMEPP services will be easily provided and used with security guarantees that would be hard and inconvenient to achieve at application level.

To experiment the effectiveness of the interaction model designed for the SMEPP middleware, we have developed a prototype which has been built on top of tuple spaces. The idea of experimenting whether tuple spaces can provide a suitable basis to implement the SMEPP middleware has two main motivations. On one hand, the abstract coordination model featured by tuple spaces has proven to notably ease the specification of complex distributed heterogeneous systems. On the other hand, the generative communication featured by tuple spaces can be seen as an enhancement of the basic coordination mechanism offered by standard data-centric storage techniques (e.g., distributed hash tables), which are indeed one of the key techniques employed in peer-to-peer systems.

The SMEPP features introduce new coordination challenges since we have to cope with service and group availability, random peer disconnection, etc. without a central entity. As we will show in the following, the result of our work shows how a coordination language such as SecureLime [7] provides several mechanisms which make the implementation of a realistic service model much easier.

Roughly speaking, the two main issues to be faced in order to implement the SMEPP model on top of tuple spaces are: (1) how to express SMEPP service-oriented aspects (e.g., groups, services, communication a.s.o.), and (2) how to suitably implement security aspects. The tuple space based language we chose to implement the system is SecureLime [7], an extension of the Lime coordination language [11], that adds security properties to tuples and (federated) tuple spaces. As we will discuss in Section 3, we exploited Lime features (such as generative communication and federated tuple spaces) to implement service-oriented aspects, and SecureLime specific features to implement security aspects.

In the following we present the key concepts of the SMEPP model and an example of its use (Section 2), the design and implementation of the tuple space based prototype (Section 3), and some concluding remarks (Section 4).

∗Work partly supported by the SMEPP project EU-FP6-IST 0333563.
The SMEPP model for secure P2P programming

We first introduce the key SMEPP concepts and the abstract SMEPP primitives followed by an example.

\[
\begin{array}{l}
// \text{Peer Management} \\
\text{pid newPeer(credentials)} \\
\text{pid getPeerInfo(pid)} \\
\text{pid[4] getPeers(pid)} \\
\text{// Group Management} \\
\text{gId createGroup(grDescr) gid[4] getGroups(grDescr)} \\
\text{grDescr getGroupDescription(gId)} \\
\text{void joinGroup(gId, creds)} \\
\text{void leaveGroup(gId)} \\
\text{// Service Management} \\
\text{<gId, gSId> publish(gId, contract)} \\
\text{void unpublish(gSId)} \\
\text{-gId, gSId, gFId>[] } \\
\text{-getServices(gId?, pId?, sContract?, maxRes?, creds)} \\
\text{-sContract getServiceContract(id)} \\
\text{-sId startSession(id)} \\
\text{// Message Handling} \\
\text{out? invoke(id, opName, in?)} \\
\text{<cId, in?> receiveMessage(gId?, opName)} \\
\text{void reply(id, opName, out?, fName?)} \\
\text{// Event Handling} \\
\text{void subscribe(evName?, gid?)} \\
\text{void unsubscribe(evName?, gid?)} \\
\text{void event(gId?, evName, in?)} \\
\text{<cId, in?> receiveEvent(gId?, evName)} \\
\end{array}
\]

Figure 1. SMEPP Primitives.

Key SMEPP Concepts and Primitives. The analysis of current state-of-the-art models in P2P systems (see [1, 2, 4, 8, 9, 10]) reveals the fact that existing frameworks for the development of P2P applications generally (i) do not provide a simple, high-level service (interaction) model that presents a suitable level of abstraction to ease the development of P2P applications, or (ii) do not model key concepts such as group-wise security, services offered both by peers and groups, message and event-based communication.

The SMEPP service-oriented model aims to tackle such limitations. It features a set of abstract primitives (see Figure 1), which can be used to develop P2P application specifications in a simple, high-level manner. We aim at deploying such primitives as different (language dependent) APIs, which will allow the deployment of SMEPP specifications as real (executable) applications. The SMEPP concepts are:

Peers. Roughly, peers are service containers. A peer executes a peer program built using the SMEPP primitives, and it may create or join groups, and offer or invoke services, as well as raise and receive events inside groups.

Groups. Groups are logical associations of peers, and they provide a secure communication environment, and a scope for published services. The SMEPP model offers security-aware primitives for group creation and joining. Furthermore, all communications among peers and services (see below) take place inside groups.

Services. Services have contracts and implementations. On the one hand, a service provides descriptive information on a service (e.g., what the service does). On the other hand, the implementation is the executable service (e.g., a Java service). Peers publish services in groups. Furthermore, service clients (viz., peers or other services) join groups and either directly invoke a particular service provider, or blindly invoke a group service. Furthermore, services could invoke other services or peers and raise or receive events.

Communication Abstractions. Peers and services communicate by exchanging (data or fault) messages, or events. Messages are used as input and output (possibly empty) for services operations. On the one hand peers and services raise events, on the other hand other peers and services can subscribe to events of their interest and wait to receive them.

For further details on the SMEPP model please, see [3].

Temperature Monitoring Example. We present here a simple example of using the primitives to model the behaviour of both peers and services. Roughly, the example aims to describe the message-based interaction between peers and services, and it shows (i) how to create peers and groups, publish services, join groups, and (ii) how to directly invoke peer services, and how invocations of request-response operations behave.

Suppose that a TempReaderPeer peer creates a TempReaderGroup in which it publishes a TempReaderService, TempReader-Service defines a getTemp request-response operation without input parameters, which (measures and) returns the ambient temperature. Another peer, InvokerPeer, joins the TempReaderGroup, and then invokes the getTemp operation of TempReaderService. Using the SMEPP primitives one could implement the above scenario as follows.

TempReaderPeer. The top of Figure 2 presents the behaviour of TempReaderPeer using a pseudocode-like notation. We use the opaque keyword to hide the value assigned to a variable. TempReaderPeer first registers itself as a new SMEPP peer, and then it creates the TempReaderGroup group. Following, it publishes TempReaderService in TempReaderGroup, and then it continues processing (e.g., it could loop forever). Note that the termination of the peer’s code implies the termination of TempReaderService, which is then unpublished by the middleware from TempReaderGroup.

TempReaderService. The middle of Figure 2 presents the behaviour of a state-less TempReaderService. TempReaderService first waits to receive an invocation of the getTemp operation. Then, it measures the temperature, and afterwards it replies to the invoker of getTemp. The execution of TempReaderService terminates after the reply.

InvokerPeer. The bottom of Figure 2 presents the behaviour of InvokerPeer. InvokerPeer first regis-
As the SMEPP project is by essence peer-to-peer and we had to choose a language amenable to distributed implementations capable of managing transient connection of peers. [R2: Available implementation] The purpose of our work was to implement the SMEPP service model by using a Linda-like language, not to make a new implementation of an existing language. Having an executable language gives us the opportunity to prove the service model is usable in a real environment. [R3: Security] SMEPP defines a configurable model of security which, by default, uses symmetric keys to manage access rights: one preshared symmetric key to access a SMEPP application (i.e., to become a SMEPP peer) and one preshared symmetric key for each group. In addition to access control, SMEPP defines group and service visibility restrictions. A peer can only discover groups and services of which it has the corresponding key. The use of preshared symmetric keys prevents the implementation to be able to expel maliciously behaving peers (in other words, authorised peers are assumed to be well behaved during its whole life). [R4: Java-integrability] Since the reference implementation of the SMEPP project will be Java-based, we decided to develop our implementation using Java, in order to have useful feedback from our proof-of-concept. So we needed a middleware providing Java integrability.

3 Implementing P2P systems with SecureLime

As discussed in Section 2, the SMEPP models relies on messages and events. Linda-like models, based on shared data, can easily incorporate these features. This is particularly the case for the SecureLime model we have chosen. To sustain this fact, Subsection 3.1 and Subsection 3.2 first state the requirements for the implementation and explain why SecureLime is an appropriate target language. Subsection 3.3 then provides an overview of our implementation.

3.1 Requirements

Following Section 2, the main requirements on the target language are as follows. [R1: Peer-to-peer orientation] As the SMEPP project is by essence peer-to-peer and embedded devices oriented, we had to choose a language

1TempServicePeerServiceId[0] denotes the value of tempServicePeerServiceId in <groupId, tempServiceGroupId, tempServicePeerServiceId, serviceContract>[0].

2An ITS is the agent’s local part of the federated tuple space.
in case of disconnection this tuple will become unavailable for others, so no peer can discover this service anymore. This feature also simplifies the group management in SMEPP as we will explain in the following subsection.

[R2] SecureLime provides an open-source implementation, complete and well-documented. This makes it a good candidate for our purposes since our aim is to avoid reinventing the wheel and implement a secure tuple space model from scratch. Moreover, one of the interesting features provided by the implementation is a fake GPS allowing to simulate physical disconnections of peers. As easily noted by the reader, these features meet requirement R2.

[R3] SecureLime offers two levels of access right control, one restricting access to tuple spaces and the other restricting access to tuples. The federation mechanism of Lime is modified in such a way that two tuple spaces are merged only if they were created using the same name and the same password. At the tuple level, SecureLime provides two special tuple fields, \( P_r \) and \( P_w \): one enabling a peer to read the tuple only if it knows the \( P_r \) field value and the other one enabling a peer to remove the tuple only if it knows the \( P_w \) field value. The former turned out to be useful for managing access to the SMEPP world and groups. The latter is used to manage visibility of groups and services. With regards to the SMEPP security requirement [R3], SecureLime is fitting really well because it offers password based encryption which can be mapped directly to the SMEPP symmetric keys. This kind of security made the authentication of peers into the SMEPP application really easy, as we will explain in details later.

[R4] Furthermore, the implementation is entirely Java-based, which meets the last requirement [R4]. Some other Linda-like models and middlewares have been considered as potential candidates. Some of them were not implemented (e.g. SecSpaces), and so, even if interesting got automatically discarded. Other implemented models were discarded because they failed to meet some requirements (e.g., both JavaSpaces and TSpace rely on a centralised architecture, provide limited security and are not open source).

3.3 Overview of the implementation

We will describe here the key concepts of our implementation. This will be done in three steps. The first one shows how the SMEPP basic concepts are modelled using Lime ones. This leads to the second step, which enriches this model with security by using SecureLime security primitives. Finally, we will present two typical use cases of the primitives with a scenario based on the example in Section 2.

High-level design. An important part of SMEPP is the discovery of services and groups. In our implementation, this is done by using two tuple spaces, playing the role of service (SD) and group (GD) directories. The tuples contained in these two tuple spaces constitute respectively a list of service descriptions and a list of group descriptions. So, to search for a group or a service we use a read operation on the right tuple space.

A peer is mapped to a Lime Agent\(^3\) which has two default tuple spaces\(^4\): the SD and the GD tuple spaces described above. When a peer wants to create or join a group \( G \), it has to: (i) create a group tuple space \( G \), (ii) put a tuple describing \( G \) in \( GD \), and (iii) put a tuple in \( G \) to update the group membership list. A group is a set of peers which have executed these three actions. The members of a same group \( G \) use the group tuple space to communicate, as it will be illustrated in the scenarios below. A service can be discovered through its description-tuple\(^5\) (providing its contract and IDs). As for the service implementation, a service is simply a Java-based process using our API. Basically, an event is modelled by the release of a tuple in a group tuple space \( G \). In order to receive an event (receiveEvent(.)) primitive, a peer creates a new Lime reaction [11] waiting for a tuple corresponding to the right event-tuple.

Security design. The security aspects of our implementation have been addressed by using the SecureLime’s extensions of Lime. According to the SMEPP guidelines: (i) every peer has an AppKey password granting access to a SMEPP application, (ii) every peer has a set of passwords (GKeys) granting access to groups.

In order to prevent illegal peers to get access to the SMEPP application, the directory tuple spaces (SD and GD) are protected using SecureLime secured tuple spaces with AppKey as password. This way, every data passing through these tuple spaces is encrypted with AppKey.

Note that if the peer does not provide the right password at peer creation (newPeer(.)) primitive, it will not get a fault message. Actually, it will create isolated directories (since the secured tuple spaces do not merge if they do not have the same password, thanks to the SecureLime federation mechanism). Furthermore, if an illegal peer creates a group, legal peers will not be able to see it since they will not share the same directories.

To ensure that every peer sees only groups and services matching its credentials, we had to prevent the tuples inside the directories from being visible to everyone. SecureLime made this task pretty easy. It suffices to use the password of the group as read-password, so a peer is only able to see a

\(^3\) A Lime Agent is a Java-based process using the Lime API [11].

\(^4\) Created by the execution of newPeer(.).

\(^5\) Which is placed after a publish(.) call in SD.
In this paper we described a running prototype implementation of the SMEPP primitives using SecureLime [7]. We consider the experiment was successful as the implementation complies with the SMEPP objectives and requirements [12]. Furthermore, this work showed how a real world service model can be implemented using a tuple space coordination language. We argue the paper features such as federated (password-protected) tuple spaces, event-based communication (one-to-many), and wise security, synchronous and asynchronous message patterns (one-to-one, direct or blind operation invocations), and event-based communication (one-to-many). Then, we have created a proof-of-concept prototype implementation of the SMEPP model that can be used to test/simulate interactions of peers and services. This implementation differs from the ones presented in [6] and [5], mainly because our service model features many more coordination concepts. In [6] and [5] only client, server and service are offered, while in the SMEPP service model groups, events and sessions are formed (one-to-one, direct or blind operation invocations), and event-based communication (one-to-many). Then, we have created a proof-of-concept prototype implementation of the SMEPP model that can be used to test/simulate interactions of peers and services. This implementation differs from the ones presented in [6] and [5], mainly because our service model features many more coordination concepts. In [6] and [5] only client, server and service are offered, while in the SMEPP service model groups, events and sessions are provided. However, service discovery is handled in a similar way, using a tuple space as repository.

We selected SecureLime for implementation based on the assessment of several tuple space-based coordination models with respect to SMEPP key requirements. SecureLime fulfills such requirements, and it also provides features such as federated (password-protected) tuple spaces, or read/remove tuple passwords. These features allowed us

---

6The last two fields of every tuple are respectively the read and remove-password. Here trpPwd (which has been securely generated by the peer) is used to ensure that only the tuple’s owner can remove it.

7The symbol # means that no password is used.
runs on top of SecureLime starting from a SMoL description, 

which firstly puts an "invocation-tuple" into TempReaderPeer’s local tempGroupId tuple space. This tuple contains the operation name (getTemp) and its parameters (empty here), the service ID (here the peer service ID contained in tempHeaderServicePeerServiceId[0]) and the ID of the caller.

2. Since invoke()’s execution must be blocked until the provider has done a receiveMessage(·). InvokerPeer will perform a (blocking) operation, waiting for the "reply-tuple" related to the getTemp operation.

3. When receiveMessage(groupId,"getTemp") is called by TempReaderService, it retrieves an "invocation-tuple" from the local tuple space of its container (TempReaderPeer) by doing a in operation on it. The template of this operation contains only the operation name (getTemp).

4. Here the service actually executes the operation.

5. TempReaderService calls reply(·) which puts a "reply-tuple" into the local tempGroupId tuple space of InvokerPeer (getTemp caller). This tuple contains the operation name, the caller ID, the operation result and a possible fault. This last action will unblock the invoker’s execution.

**Figure 4. Service invocation.**

to successfully model all SMEPP key concepts. However, a limitation of SecureLime is that it does not provide a way to change "on-the-fly" the tuple space passwords. Consequently, the developer has to deal with this issue at the application level.

We have also defined a SMEPP specification language (SMoL [3]), which allow one to orchestrate SMEPP primitives into complex behaviour (e.g., using sequential, parallel, choice, or event and fault handler operators). SMoL is meant to assist the SMEPP developer into (semi-automatically) generating peer or service code. Furthermore, such a language enables the formal analysis of the behaviour of peers and services, and of their interactions [3].

We have implemented a SMoL2Java translator (which generates Java code from a SMoL specification) and we have integrated it with our implementation based on SecureLime. The resulting prototype produces executable Java code that runs on top of SecureLime starting from a SMoL description of the behaviour of a peer or service. Unfortunately, space limitations do not allow us to describe the details of the translation here.

Beyond the above mentioned limitations due to the restrictions of the current SecureLime release, our prototype only features a basic mechanism for the discovery of service contracts based on the syntactic matching of tuples.

Our planned next step is to thoroughly experiment the prototype in order to engineer the implementation. We also intend to overcome the present limitations, and to experiment the implementation of an enhanced security mechanism based on session keys.

**References**


