Coordinating Processes with Secure Spaces

Jan Vitek[a], [1], Ciarán Bryce[b], Manuel Oriol[b]

[a] *CERIAS, Dept. of Computer Sciences, Purdue University, West Lafayette, IN.*
[b] *Object Systems Group, University of Geneva, Geneva, Switzerland.* [2]

Abstract

The Linda shared space model and its derivatives provide great flexibility for building parallel and distributed applications composed of independent processes. However, the shared space model does not provide protection against untrustworthy processes. Linda processes communicate by reading and writing messages in a globally visible data space, so a malicious process can launch any number of security attacks. This paper presents the design of a new coordination model which extends Linda with fine-grained access control. The semantics of the model is presented in the context of a process calculus. A prototype of our model, called SecOS, has been implemented in Java.

Key words: Coordination Languages, Linda, Security, Access Control.

1 Introduction

Coordination is the theory and practice of assembling software systems out of independently developed components. Coordination in open networks such as the Internet is particularly difficult since the processes to coordinate might not be trustworthy. Thus coordination infrastructures must provides mechanisms to protect applications, as well as the overall system, against attacks. This paper presents the design of a coordination infrastructure named SecOS, built on top of the Java programming language, as an extension of Gelernter's Linda [12] coordination language.

[1] Corresponding author.
E-mail: `jv@cs.purdue.edu` (J. Vitek), `bryce@cui.unige.ch` (C. Bryce) and `oriol@cui.unige.ch` (M. Oriol).
[2] This work was supported by the Swiss National Science Foundation, under grant FNRS 20-53399.98.

Linda [12,13] is an elegant coordination model for parallel and weakly distributed systems in which processes communicate by generating new message objects and placing these objects in a shared data space for other processes to retrieve. The space is commonly known as a tuple space, and the objects stored in the space are called tuples since they are ordered sequences of basic data types. This programming model, often referred to as generative communication, allows for interaction between processes separated in time: since the data space is persistent, a message can be retrieved anytime after it has been placed. Processes are also separated in space: communicating processes need not know each other's identity, nor have a dedicated connection established between them. Linda is therefore suitable for anonymous communication and resource discovery protocols [26], and to coordinate mobile components [21,7,20]. Several coordination infrastructures have implemented this model by embedding the basic tuple space operations in a host language [23,18,17,11].

The Linda model provides three operations[3], `out`, `rd` and `in`; informally their semantics is:

- **out** $\langle x, y, \ldots \rangle$: writes tuple $\langle x, y, \ldots \rangle$ to the data space without blocking.
- **in** $\langle x, y, \ldots \rangle\, z$: blocks until a tuple matches the template $\langle x, y, \ldots \rangle$; if several candidates are found, then one is nondeterministically removed from the space and bound to variable z.
- **rd** $\langle x, y, \ldots \rangle\, z$: behaves like **in** except that the matching tuple is not removed from the space.

The main distinguishing characteristic of Linda is the pattern matching of tuples in input requests. The simplest form of pattern matching is by equality comparison. Thus for example, a process may retrieve a tuple such as $\langle\, 1, 2, \text{“}xyz\text{”} \,\rangle$ by executing **in** $\langle\, 1, 2, \text{“}xyz\text{”} \,\rangle\, x$. The input operation will block if the tuple is not in the shared space. The space is thus the basic mechanism for process synchronization. Processes may also use partially defined templates, the special value "?" denotes fields that can take any value, to describe the data that they wish to retrieve. The tuple $\langle\, 1, 2, \text{“}xyz\text{”} \,\rangle$ can be matched by input requests such as **in** $\langle\, ?, 2, \text{“}xyz\text{”} \,\rangle\, x$, **in** $\langle\, 1, ?, \text{“}xyz\text{”} \,\rangle\, x$, **in** $\langle\, 1, 2, ? \,\rangle\, x$, and **in** $\langle\, ?, ?, ? \,\rangle\, x$. Partially defined templates allow processes to exchange information, and thus are the basic mechanism for process communication in Linda.

The main obstacle to the use of Linda for coordinating untrusted components is the lack of any protection mechanism in the basic model. Without a means to constrain the behavior of processes running in the shared data space, there

[3] For simplicity, we do not consider predicate forms (`inp` and `rdp`) which are nonblocking variants of some operations, although they are provided in SecOS. Furthermore Linda introduced another basic operation, `eval`, for starting new threads of computation, which is unnecessary when the host language is concurrent.

is simply no way to prevent a malicious or faulty process from wrecking havoc on an entire system. For instance, consider the simplest of processes, one which repeatedly removes an arbitrary tuple from the space, [4]

$$!\ \mathbf{in}\langle\ \rangle\, x\,.\,\mathbf{0}$$

Here $\langle\ \rangle$ denotes a template that can match any tuple; $\mathbf{in}\langle\ \rangle\, x\,.\,\mathbf{0}$ denotes a process that reads a tuple and then evolve to the inert process, and the exclamation mark denotes infinite repetition. The above process indiscriminately removes messages that are part of ongoing protocols, thus starving or at least disrupting the processes running them.

Another example of dangerous behavior is demonstrated by a process that eavesdrops on messages exchanged in the space,

$$!\ \mathbf{in}\langle\ \rangle\, x\,.\,\big(\mathbf{out}\, x \mid \ldots\big)$$

The process repeatedly inputs a random tuple and outputs it again retaining a copy bound to x. This does not interfere with other users of the data space, but lets the process peek at the data exchanged between unrelated processes.

These were examples of integrity and privacy attacks. Another kind of attack is denial of service. The following process repeatedly deposits tuples in the shared space,

$$!\ \mathbf{out}\langle\, 1\,\rangle$$

Without any limitation on tuple lifetimes or bound on the number of iterations, any implementation of a shared data space will eventually run out of memory.

The extreme simplicity of these three malicious processes underscores the lack of protection in coordination infrastructures. The goal of this research is to investigate how to extend a coordination model with support for fine-grained access control. The challenge we are faced with is to provide access control mechanisms without loosing the flexibility that makes generative communication attractive. With the exception of KLAIM [20] and JavaSpaces [11], we are not aware of any work in this direction. Our approach is to investigate language design issues and to provide a practical solution to the problem of coordinating untrusted processes. This paper presents the semantics of a new coordination model and discusses the implementation of a prototype system called SECOS embedded in the JAVA programming language.

[4] For the sake of brevity, examples are given using the syntax of the secure spaces calculus introduced in Section 3, rather than in the concrete syntax of the SECOS implementation.

2 Coordination with Secure Spaces

In this paper we present a coordination model, referred to as *Secure Spaces*, which extends Linda with fine-grained access control to the shared data space. The motivation for the design of secure spaces comes from the difficulty in engineering a comprehensive security architecture that enforces the security requirements of a variety of applications without being overly restrictive. Rather than enforcing a specific security policy in the model, we chose to define a set of simple mechanisms that can be used by application logic to efficiently implement a range of security policies.

The core idea of secure spaces is simple: we protect every field of a tuple with a lock. A lock prevents unauthorized processes from gaining access to the data held in the field. Instead of storing tuples made up of an ordered sequence of fields, a secure space stores objects consisting of locked fields, each of these fields being composed of a label and a value. The label can be thought of as specifying the key needed to unlock the value. The semantics of secure spaces have been designed to ensure that processes that do not have the key required to access a field may not gain any information about the field's contents. This also requires hiding such fields during pattern matching.

The remainder of this section informally introduces the concepts of the secure space coordination model. To specify the semantics of secure spaces without having to deal with the syntax of an actual programming language, for example JAVA the host language of SECOS, we introduce secure spaces in terms of a process calculus that gives a precise semantics to the secure space primitives. This calculus is presented in Section 3.

2.1 Objects and Locks

In secure spaces, an *object* is an unordered set of locked fields, or locks. A *secure space* is a multiset of objects. Locks are labeled values, the field's value can be an object and is the data part of the field. The label regulates access to the contents as it specifies which key is needed to unlock the value. We use labels, which have to be distinct, to select fields instead of indices.

A locked field can only be unlocked with a key matching the field's label. But unlocking a field does not grant access to other fields in an object, only to that field's value. To implement this privacy feature, the rules for extracting values from an object as well as the pattern matching rules used to retrieve objects from the shared space have been modified.

There are two kinds of primitive locks, *symmetric locks* (s-locks) and *asym-*

metric locks (a-locks), as well as a derived form called *object locks* (o-locks). The simplest locks are symmetric where the same key is used to lock and unlock fields. For example, the Linda tuple $\langle\, 3,\ \text{“}xyz\text{”} \,\rangle$ can be represented by the following secure spaces object,

$$\langle\, a_a : 3\ \ b_b : \text{“}xyz\text{”} \,\rangle$$

Labels a_a and b_b denote symmetric keys protecting values 3 and “xyz” respectively. These keys must be presented in order to access the value of the locks or to construct a template that will match this object. The pattern matching rules of s-locks are a kind of structural subtyping, where shorter objects match longer ones with the same labels.

To select a value from an object, a process must present the corresponding key. In the case of an s-locked field such as $a_a : b_b$ for example, the matching key is a_a itself. Thus the following expression evaluates to 3,

$$\langle\, a_a : 3\ \ b_b : \text{“}xyz\text{”} \,\rangle\, .\, a_a$$

Labels are first-class values in our model, and can be transmitted in objects. Furthermore, processes can generate fresh labels, written as (**new** a_b). So, the following program creates a new key and uses it in the s-lock guarding x.

$$(\textbf{new}\, a_a)\ \textbf{out}\, \langle\, a_a : x \,\rangle$$

Since labels are lexically scoped, we have effectively locked x and thrown away the key.

Asymmetric locks (a-locks) are pairs consisting of a label a_b and its inverse b_a such that if a_b locks a field, then only b_a can unlock it. An example of an a-locked object is:

$$\langle\, a_b : 1 \,\rangle$$

One obvious use of asymmetric locks is to publish one of the labels, a_b, as a public key and keep the other, b_a, as a private key. The pattern matching rules for a-locks require the use of the inverse key, in the above example b_a, to retrieve an object lock with an a-lock. Thus an object locked with a public key is pattern matched using the private key.

2.2 Pattern Matching

Secure spaces have different pattern matching rules than Linda. In secure spaces, pattern matching does not rely on the order of occurrence of fields in an object, but rather on field labels. As fields can contain objects, pattern matching is defined recursively on the complete object structure. To preserve

privacy, fields that are not present in a template are considered hidden and are therefore not used in pattern matching. Thus as we mentioned earlier there is a certain similarity between pattern matching and structural subsumption as a "longer" object is matched by a "shorter" template. For instance, an output offer such as,

$$\mathbf{out}\,\langle\, a_b : b\ \ d_d : e \,\rangle$$

can be matched by the input request,

$$\mathbf{in}\,\langle\, d_d : e \,\rangle\, x$$

Other matching templates for the same output are:

$$\langle\, a_b : b \,\rangle\ \langle\, a_b : ? \,\rangle, \langle\, d_d : ? \,\rangle, \langle\, a_b : b\ \ d : ? \,\rangle, \langle\, a_b : b\ \ d_d : e \,\rangle, \langle\ \rangle$$

The empty object $\langle\ \rangle$ can be used as a template to match any other object. Pattern matching an a-locks requires presentation of the inverse key, *e.g.,* the object $\langle\, a_b : \langle\ \rangle \,\rangle$ is matched by $\langle\, b_a : \langle\ \rangle \,\rangle$.

It is important to recall that retrieving an object does not grant access to its fields. Without the appropriate keys, fields remain hidden, so even if an object is leaked to a malicious process, the information it contains remains protected.

A simple key exchange protocol demonstrates the use of pattern matching rules. Consider the following term in which two processes use a key pair (a_b, b_a) to exchange the shared key c_c,

$$(\mathbf{new}\, c_c)\Big(\,\mathbf{out}\,\langle\, a_b : c_c \,\rangle \mid P\,\Big) \quad | \quad \Big(\mathbf{in}\,\langle\, b_a : ? \,\rangle\, x \,.\, Q\Big)$$

The output term $\mathbf{out}\,\langle\, a_b : c_c \,\rangle$ can be matched by $\mathbf{in}\,\langle\, b_a : ? \,\rangle\, x$ because b_a is the inverse of a_b and the wild card ? matches any value. The term thus reduces in one step,

$$(\mathbf{new}\, c_c)\Big(\, P \ \mid\ Q\{{}^{\langle a_b\, :\, c_c \rangle}/_{x}\}\,\Big)$$

In the resulting term, the key's scope encloses P and Q, since both processes now share c_c. Furthermore, all occurrences of x in Q have been substituted with $\langle\, a_b : c_c \,\rangle$, *e.g.,* an expression $x.b_a$ in Q becomes $\langle\, a_b : b_c \,\rangle.b_a$ and yields c_c.

2.3 Extending objects

Another feature of secure space is that objects can be extended without revealing their contents. An extension operation, denoted by $\oplus$, will add a new locked field to an object if the field label is not already present, or, in the case a field with that label exists, will overwrite the value. The following is an example of an extension in which the a-lock b_c : "xyz" is added to object

$\langle\, a_a : 22 \,\rangle$,

$$\langle\, a_a : 22 \,\rangle \oplus b_c : \text{“}xyz\text{”} \qquad \text{yields} \qquad \langle\, a : 22 \;\; b_c : \text{“}xyz\text{”} \,\rangle$$

The process performing the extension need not know anything about the contents of the object it is extending and will not gain any information as a result of extension. Thus for instance, consider:

$$\langle\, a_b : 22 \,\rangle \oplus a_b : \text{“}xyz\text{”} \qquad \text{yields} \qquad \langle\, a_b : \text{“}xyz\text{”} \,\rangle$$

Without the key b_a, there is no way for the extending process to even know that the object already had an a_b field, not to mention its value.

Object extension is essential to transparently tag objects, *e.g.,* with lifetime annotations. The conjunction of hidden fields and object extension allows us to implement a user-level garbage collector that tags objects without knowledge of their internal structure. This tagging does not affect the behavior of applications that operate on the tagged objects.

2.4 Locking objects

Up to this point, we have controlled access to fields, but not access to objects in the shared space. For example it is often desirable to prevent processes from using the empty template $\langle\, \rangle$ to indiscriminately match and remove objects. We therefore introduce object locks (o-locks) to restrict the visibility of objects from the pattern matching process. A locked object is created with

$$\mathbf{out}\, \langle\, a_a : 12 \;\; b_b : 3 \,\rangle @ c_d$$

where key c_d is used to lock the object; a matching input could be

$$\mathbf{in}\, \langle\, b_b : ? \,\rangle @ d_c \;\; x \,.\, P$$

Notice the use of the inverse key d_c to retrieve the object. Asymmetric keys are particularly interesting as they allow the expression of write-only and read-only access rights to a secure space.

Object locks can be viewed as partitioning the shared memory. For some o-lock term, $\mathbf{out}\, \langle\, \vec{f} \,\rangle @ \ell$, the key ℓ creates a partition of the space, such that the only processes that may write to it are ones that know ℓ, and processes that want to read from the partition must have the inverse key $\overline{\ell}$. In this sense, o-locks may be viewed as giving the same expressive power as variants of Linda with multiple tuple spaces [6,13,16,25], but they are more flexible. With multiple tuple spaces, a process is granted wholesale access to the space whenever it gets the space's identifier. Object locks can be used to give very limited and

controlled access to a partition. For instance, it is possible to restrict a process to input only one kind of object. Consider a configuration in which processes P and Q are running in parallel and Q wants to grant P read-access to some partition c_d of the space. One solution would be to give the inverse key d_c to P. But this would permit P to retrieve any object in the partition and also to extract the value of fields locked under c_d. Assuming that P should only retrieve objects that contain the key a_a and should not select fields labeled with c_d, a better solution is to hand P a template, for example $\langle\, a_c : ?\,\rangle @ d_c$, rather than a key, that it can use for matching. Let P and Q be as follows,

$$P = \mathbf{in}\,\langle\, b_b : ?\,\rangle\, x\,\mathbf{.}\,\mathbf{in}\,(x.b_b)\, y\,\mathbf{.}\,\ldots$$

and

$$Q = (\mathbf{new}\, c_d)\Big(\mathbf{out}\,\langle\, b_b : \langle\, a_a : ?\,\rangle @ d_c\,\rangle \mid Q'\Big)$$

The term $P \mid Q$ reduces in one step to

$$(\mathbf{new}\, c_d)\,\Big(\;\; \mathbf{in}\,\langle\, a_a : ?\,\rangle @ c_d\;\, y\,\mathbf{.}\,\ldots \quad\mid\quad Q'\,\Big)$$

P can use the template to retrieve objects but has no means to get at the label d_c. In particular, it does not have access to either of the partition keys (c_d and d_c) so it can neither add new objects nor select fields protected by these keys.

In the secure spaces calculus o-locks are a derived concept, Section 3.3 gives a translation from terms containing o-locks to terms in the core calculus.

3 The Secure Spaces Calculus

The secure spaces calculus is based on the asynchronous π-calculus [15,3,2], because π provides a small and elegant concurrent programming language with simple semantics and thus allows for a compact formulation of secure spaces in computationally complete setting. The main departure from the π-calculus is the use of generative communication operations instead of channel-based primitives. The idea of embedding Linda in a process calculus has been explored in depth in previous work [5,10]. The emphasis of this paper is on language design issues rather than on expressiveness. Since type checking is not the focus of this paper, the secure spaces calculus is untyped and allows ill-formed processes to be written. Type errors cause processes to get stuck and prevent further reduction.

3.1 Syntax

The syntax of the core calculus is summarized in Table 1. We take an infinite set of names ranged over by meta-variables a, b, c, d. Labels are pairs of name, written a_b, and ranged over by meta-variable ℓ. If $a = b$, we call label a_b a symmetric key, otherwise it is an asymmetric key. The inverse of a key a_b is the key b_a, an auxiliary inverse function, written $\overline{\cdot}$, is defined as $\overline{a_b} = b_a$ and $\overline{\overline{a_b}} = a_b$. Basic values are ranged over by v, and consist of labels, objects, and ?. The symbol ? denotes the distinguished void element. Locked fields, ranged over by f, are written $a_b : v$. Objects are, possibly empty, vectors of locks, written $\langle \vec{f} \rangle$. The function keys($\vec{f}$) returns the set of labels of the vector $\vec{f}$.

$$
\begin{array}{lrcl}
e & ::= & x \mid v & \mid \langle \vec{e} : \vec{e} \rangle \mid \quad e.e \quad \mid e \oplus e : e \\
v & ::= & ? \mid \ell & \mid \langle \vec{f} \rangle \\
f & ::= & \ell : v & \\
P & ::= & \mathbf{0} \mid P \mid Q \mid & \mathbf{!}\, P \mid \mathbf{in}\, e\, x\, \mathbf{.}\, P \mid \mathbf{out}\, e \mid (\mathbf{new}\, a_b) P
\end{array}
$$

Table 1: Core Language Syntax.

The syntactic category of *expressions*, ranged over by e, includes basic values, objects, selection expressions and extension expressions. The syntactic category of *processes*, ranged over by P and Q, includes the empty process $\mathbf{0}$ which has no behavior, parallel composition of processes $P \mid Q$, replication of processes $\mathbf{!}\, P$, as well as two communication primitives. The first of these is the input operation $\mathbf{in}\, e\, x\, \mathbf{.}\, P$ which tries to match the template e against an output offer and bind the result to variable x. The operation is blocking; P cannot execute until the match succeeds. The second operation is the asynchronous output $\mathbf{out}\, e$ which deposits the object denoted by e in the data space. Finally, The restriction operator $(\mathbf{new}\, a_b)$ generates a fresh key pair a_b and b_a. The calculus is lexically scoped, so $(\mathbf{new}\, a_b)P$ means that a_b and b_a are visible only in process P.

3.2 Operation Semantics

The operational semantics of the secure spaces calculus is given in Table 2. The reduction relation $P \rightarrow P'$ defines when process P reduces in one step of internal computation to P'. We define two auxiliary notions: structural congruence and evaluation.

Reduction

$$\frac{P \to Q}{(\mathbf{new}\, a_b)P \to (\mathbf{new}\, a_b)Q} \qquad \frac{P \to Q}{P \mid R \to Q \mid R} \qquad \frac{P \equiv P' \quad P' \to Q}{P \to Q}$$

$$\frac{e \downarrow \langle\, \vec{f}\,\rangle \quad e' \downarrow \langle\, \vec{f'}\,\rangle \quad \langle\, \vec{f'}\,\rangle \leq \langle\, \vec{f}\,\rangle}{\mathbf{out}\, e \mid \mathbf{in}\, e'\, x \,.\, P \to P\{^{\langle \vec{f} \rangle}/_{x}\}}$$

Evaluation

$$v \downarrow v \qquad \langle\,\rangle \downarrow \langle\,\rangle \qquad \frac{e \downarrow \langle\, \vec{f}\,\rangle \quad \langle\, \vec{f}\,\rangle \equiv \langle\, \vec{f'}\,\rangle}{e \downarrow \langle\, \vec{f'}\,\rangle} \qquad \frac{e \downarrow \langle\, \ell : v\, \vec{f}\,\rangle \quad e' \downarrow \overline{\ell}}{e.e' \downarrow v}$$

$$\frac{e \downarrow \ell \quad e' \downarrow v \qquad \langle\, \vec{e} : \vec{e'}\,\rangle \downarrow \langle\, \vec{f}\,\rangle \qquad \ell \notin \mathrm{keys}(\vec{f})}{\langle\, e : e'\, \vec{e} : \vec{e'}\,\rangle \downarrow \langle\, \ell : v\, \vec{f}\,\rangle}$$

$$\frac{e \downarrow \langle\, \vec{f}\,\rangle \qquad e' \downarrow \ell \qquad e'' \downarrow v}{e \oplus e' : e'' \downarrow \langle\, \ell : v\, (\vec{f} \setminus \ell)\,\rangle}$$

Structural Congruence Rules

$$\langle\, \vec{f}\, \ell : v\, \vec{f'}\,\rangle \equiv \langle\, \ell : v\, \vec{f}\, \vec{f'}\,\rangle \qquad P \mid Q \equiv Q \mid P \qquad P \mid \mathbf{0} \equiv P \qquad \mathbf{!}\, P \equiv P \mid \mathbf{!}\, P$$

$$(P \mid Q) \mid R \equiv P \mid (Q \mid R) \qquad (\mathbf{new}\, a_b)(\mathbf{new}\, c_d)P \equiv (\mathbf{new}\, c_d)(\mathbf{new}\, a_b)P$$

$$(\mathbf{new}\, a_b)(P \mid Q) \equiv P \mid (\mathbf{new}\, a_b)Q \quad \textit{if}\ \ a_b, b_a \notin \mathit{fn}(P)$$

Pattern Matching Rules

$$\ell \leq \ell \qquad ? \leq v \qquad \langle\,\rangle \leq \langle\, \vec{f}\,\rangle$$

$$\frac{v \leq v' \qquad \langle\, \vec{f}\,\rangle \leq \langle\, \vec{f'}\,\rangle}{\langle\, \ell : v\, \vec{f}\,\rangle \leq \langle\, \overline{\ell} : v'\, \vec{f}\,\rangle}$$

Table 2: Operational Semantics.

Structural congruence $\equiv$ is the least congruence on processes satisfying the axioms and rules given in Table 2; it indicates when a process may replace another in a computation in such a way that the computation yields an equivalent result. The evaluation relation $\downarrow$ denotes the result of field selection and object extension expressions. The reduction relation $\rightarrow$ is the least relation on processes that satisfies the axioms and rules defined in Table 2.

The notation $\vec{e}$ denotes zero or more occurrences of e. The term $P\{^{e}/_{x}\}$ represents process P in which all free occurrences of x are replaced by e. Trailing inert processes are removed; thus $\mathbf{in}\, e\, x\, \mathbf{.}\, \mathbf{0}$ becomes $\mathbf{in}\, e\, x$. The free labels of a term are denoted by $fn(\_)$, and defined in Table 3.

$$fn(\mathbf{0}) = fn(x) = fn(?) = \{\}, \quad fn(a_a) = \{a_a\},$$

$$fn(\langle\, \vec{e} : \vec{e}\,' \,\rangle) = \bigcup fn(\vec{e}) \cup \bigcup fn(\vec{e}\,')$$

$$fn(\langle\, \vec{e} : \vec{e}\,' \,\rangle @ \ell) = \bigcup fn(\vec{e}) \cup \bigcup fn(\vec{e}\,') \cup \{\ell, \overline{\ell}\} \cup fn(e),$$

$$fn(P \mid Q) = fn(P) \cup fn(Q), \quad fn(!\, P) = fn(P),$$

$$fn(\mathbf{out}\, e) = fn(e), \quad fn(\mathbf{in}\, e\, x\, \mathbf{.}\, P) = fn(e) \cup fn(P),$$

$$fn((\mathbf{new}\, a_b)P) = fn(P) - \{a_b, b_a\}$$

Table 3: Free keys.

The main reduction rule determines when an an input request can consume an output offer. If the output term evaluates to an object $\langle\, \vec{f}\,\rangle$, the input to an object $\langle\, \vec{f}\,'\,\rangle$, and the objects match, then the output offer is consumed and the continuation P can execute.

$$\frac{e \downarrow \langle\, \vec{f}\,\rangle \quad e' \downarrow \langle\, \vec{f}\,'\,\rangle \quad \langle\, \vec{f}\,'\,\rangle \leq \langle\, \vec{f}\,\rangle}{\mathbf{out}\, e \mid \mathbf{in}\, e'\, x\, \mathbf{.}\, P \rightarrow P\{^{\langle \vec{f} \rangle}/_{x}\}}$$

The pattern matching relation $\leq$ is a relation on values with ? as minimum element. Objects are matched by pair-wise field and key comparison, intuitively a shorter object matches a longer if each field $\ell : v$ of the shorter object has a corresponding field $\overline{\ell} : v'$ and $v \leq v'$.

$$\frac{v' \leq v \qquad \langle\, \vec{f}\,\rangle \leq \langle\, \vec{f}\,\rangle}{\langle\, \ell : v\ \vec{f}\,\rangle \leq \langle\, \overline{\ell} : v'\ \vec{f}\,\rangle}$$

Pattern matching is recursive in the value of labeled fields, so to determine that

$$\langle\, a_a : \langle\, b_b : \langle\, \rangle\, \rangle\, \rangle \leq \langle\, a_a : \langle\, b_b : \langle\, \rangle\ c_c : \langle\, \rangle\, \rangle\, \rangle$$

it is necessary to check that

$$\langle\, b_b : \langle\,\rangle\,\rangle \leq \langle\, b_b : \langle\,\rangle\; c_c : \langle\,\rangle\,\rangle$$

On the surface the pattern matching relation appears to be a form of structural subtyping. But the presence of asymmetric keys ensures that the relation is neither reflexive nor transitive, consider for example that both $\langle\, a_b : ?\,\rangle \leq \langle\, b_a : ?\,\rangle$ and $\langle\, b_a : ?\,\rangle \leq \langle\, a_b : \langle\,\rangle\,\rangle$ hold while $\langle\, a_b : ?\,\rangle \leq \langle\, a_b : \langle\,\rangle\,\rangle$ does not.

The interesting cases of the evaluation relation are field selection and object extension. Selection, $e.e'$, extracts a value from an object if e evaluates to an object and e' to a label $\overline{\ell}$ such that the inverse key ℓ is present in the object.

$$\frac{e \downarrow \langle\, \ell : v\; \vec{f}\,\rangle \quad e' \downarrow \overline{\ell}}{e.e' \downarrow v}$$

An error occurs in case the key is not present and the execution gets stuck.

The object extension operation, $e \oplus e' : e''$, adds a field to an object, if a field with the same label is already present the old value is overridden. We write $\vec{f} \setminus \ell$ to denote the sequence of fields in which ℓ does not occur as a field label.

$$\frac{e \downarrow \langle\, \vec{f}\,\rangle \qquad e' \downarrow \ell \qquad e'' \downarrow v}{e \oplus e' : e'' \downarrow \langle\, \ell : v\; (\vec{f} \setminus \ell)\,\rangle}$$

While a more detailed study of equivalences is beyond the scope of this work, we expect that some simple secrecy properties hold in our calculus. For instance, the values of locked fields are protected, thus for any context $C[]$ if x does not occur free in $\vec{f}$, the following two expressions can not be distinguished,

$$C[(\mathbf{new}\, a_b)\mathbf{out}\, \langle\, a_b : y\; \vec{f}\,\rangle] \qquad \text{and} \qquad C[(\mathbf{new}\, a_b)\mathbf{out}\, \langle\, a_b : z\; \vec{f}\,\rangle]$$

The value of locked fields may not be observed without the matching key. On the other hand the following terms are not equivalent,

$$C[(\mathbf{new}\, a_b)\mathbf{out}\, \langle\, a_b : y\; \vec{f}\,\rangle] \quad \text{and} \quad C[\mathbf{out}\, \langle\, \vec{f}\,\rangle]$$

In the case all field labels in $\vec{f}$ are symmetric, the terms can be distinguished because an equality test can be encoded in the calculus. Term $e =_s e' \,.\, P$ reduces to P if e and e' evaluate to objects with the same (symmetric) field labels and $=_s$ values. The encoding of the test is as follows assuming x does not occur free in P:

$$e =_s e' \,.\, P \quad \stackrel{def}{=} \quad \mathbf{out}\, e \mid \mathbf{in}\, e'\, x \,.\, (\mathbf{out}\, x \mid \mathbf{in}\, e\, x \,.\, P)$$

The term reduces to P if and only if $e \downarrow v$, $e' \downarrow v'$, $v \leq v'$ and $v' \leq v$.

Object locks control access to objects in secure spaces. The syntax for emitting an object $\langle\, \vec{f}\,\rangle$ locked by ℓ is $\mathbf{out}\,\langle\, \vec{f}\,\rangle@\ell$, and the syntax for retrieving some object matching template $\langle\, \vec{f}\,\rangle$ locked under ℓ is $\mathbf{in}\,\langle\, \vec{f}\,\rangle@\ell\, x\,\mathbf{.}\, P$. The semantics can be expressed by the reduction rule,

$$\frac{e \downarrow \langle\, \vec{f}\,\rangle \quad e' \downarrow \langle\, \vec{f'}\,\rangle \quad \langle\, \vec{f'}\,\rangle \leq \langle\, \vec{f}\,\rangle}{\mathbf{out}\, e@\ell \mid \mathbf{in}\, e'@\overline{\ell}\, x\,\mathbf{.}\, P \rightarrow P\{^{\langle \vec{f}\rangle@\ell}/_{x}\}}$$

But the calculus need not be extended since o-locks can be expressed in the core language. Table 4 gives an inductive definition of an encoding from terms with o-locks to basic calculus terms. $[\![\, P \,]\!]$ denotes the translation of term P. The intuition is to extend all objects with one or two additional fields. Locked objects will be extended with a pair of fields $\mathtt{l}_1$ and $\mathtt{r}_\mathtt{r}$ with, respectively, the o-lock key and its inverse as values. For plain objects, an additional field labeled $\mathtt{l}_1$ and with value $\mathtt{l}_1$ will be inserted. We choose labels $\mathtt{l}_1$ and $\mathtt{r}_\mathtt{r}$ so that they do not occur free in P.

$$\begin{aligned}
[\![\, x \,]\!] &= x \\
[\![\, ? \,]\!] &= ? \\
[\![\, \langle\, \vec{e} : \vec{e'}\,\rangle \,]\!] &= \langle\, [\![\, \vec{e} \,]\!] : [\![\, \vec{e'} \,]\!]\,\rangle \oplus \mathtt{l}_1 : \mathtt{l}_1 \\
[\![\, \langle\, \vec{e} : \vec{e'}\,\rangle@\ell \,]\!] &= (\langle\, [\![\, \vec{e} \,]\!] : [\![\, \vec{e'} \,]\!]\,\rangle \oplus \mathtt{l}_1 : \ell) \oplus \mathtt{r}_\mathtt{r} : \overline{\ell} \\
[\![\, e \oplus e' : e'' \,]\!] &= [\![\, e \,]\!] \oplus e' : [\![\, e'' \,]\!] \\
[\![\, e.e' \,]\!] &= [\![\, e \,]\!].e' \\
[\![\, P \mid Q \,]\!] &= [\![\, P \,]\!] \mid [\![\, Q \,]\!] \\
[\![\, !\, P \,]\!] &= !\, [\![\, P \,]\!] \\
[\![\, \mathbf{0} \,]\!] &= \mathbf{0} \\
[\![\, \mathbf{out}\, e \,]\!] &= \mathbf{out}\, [\![\, e \,]\!] \\
[\![\, \mathbf{in}\, e\, x\,\mathbf{.}\, P \,]\!] &= \mathbf{in}\, [\![\, e \,]\!]\, x\,\mathbf{.}\, [\![\, P \,]\!] \\
[\![\, (\mathbf{new}\, a_b) P \,]\!] &= (\mathbf{new}\, a_b) [\![\, P \,]\!]
\end{aligned}$$

Table 4: Translation from a calculus with object locks to the core calculus.

To illustrate the translation, consider the following process,

$$[\![\ \mathbf{out}\,\langle\, c_c : d_d \,\rangle @ a_b \ \mid\ \mathbf{in}\,\langle\ \rangle @ b_a\, y\,\mathbf{.0} \ \mid\ \mathbf{out}\,\langle\, c_c : d_d \,\rangle \ \mid\ \mathbf{in}\,\langle\ \rangle\, y\,\mathbf{.0}\ ]\!]$$

which yields,

$$\begin{aligned}&\mathbf{out}\,\langle\, \mathtt{r}_\mathtt{r} : b_a\ \mathtt{l}_\mathtt{1} : a_b\ c_c : d_d \,\rangle \ \mid\ \mathbf{in}\,\langle\, \mathtt{r}_\mathtt{r} : a_b\ \mathtt{l}_\mathtt{1} : b_a \,\rangle\ \mathbf{.0}\ \mid\\ &\mathbf{out}\,\langle\, \mathtt{l}_\mathtt{1} : \mathtt{l}_\mathtt{1}\ c_c : d_d \,\rangle \ \mid\ \mathbf{in}\,\langle\, \mathtt{l}_\mathtt{1} : \mathtt{l}_\mathtt{1} \,\rangle\ \mathbf{.0}\end{aligned}$$

In the translated term it is clear that objects locked with an o-lock can not be matched by unlocked objects, consider the translation,

$$[\![\,\langle\ \rangle @ b_a\,]\!] \;=\; \langle\, \mathtt{r}_\mathtt{r} : a_b\ \mathtt{l}_\mathtt{1} : b_a \,\rangle \;\leq\; \langle\, \mathtt{r}_\mathtt{r} : b_a\ \mathtt{l}_\mathtt{1} : a_b\, c_c : d_d \,\rangle \;=\; [\![\,\langle\, c_c : d_d \,\rangle @ a_b\,]\!]$$

According to the patter matching rules these o-locked objects match. Now consider the translation of the plain objects, they match as well.

$$[\![\,\langle\ \rangle\,]\!] \;=\; \langle\, \mathtt{l}_\mathtt{1} : \mathtt{l}_\mathtt{1} \,\rangle \;\leq\; \langle\, \mathtt{l}_\mathtt{1} : \mathtt{l}_\mathtt{1}\ c_c : d_d \,\rangle \;=\; [\![\,\langle\, c_c : d_d \,\rangle\,]\!]$$

But trying to match an o-locked object with an empty template will fail,

$$[\![\,\langle\ \rangle\,]\!] \;=\; \langle\, \mathtt{l}_\mathtt{1} : \mathtt{l}_\mathtt{1} \,\rangle \;\not\leq\; \langle\, \mathtt{r}_\mathtt{r} : b_a\ \mathtt{l}_\mathtt{1} : a_b\ c_c : d_d \,\rangle \;=\; [\![\,\langle\, c_c : d_d \,\rangle @ a_b\,]\!]$$

We conjecture that the terms $[\![\, P \mid (\mathbf{new}\, a_b)\mathbf{out}\, e @ a_b \,]\!]$ and $[\![\, P \,]\!]$ are equivalent for all P. That is, an o-locked object is protected from processes that do not have its key. This is a fundamental security property of the SecOS coordination infrastructure.

4 Examples

In this section we give some examples of the use of secure spaces.

4.1 Encoding Linda with Secure Spaces

It is legitimate to wonder whether any expressiveness has been lost going from Linda's positional notation to the secure space model with labeled fields. Furthermore, we lost the `rd` operator in the process. We show how to recover both through encodings into the core calculus.

The `rd` operator is encoded as an input followed by an output in parallel with

a continuation process:

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

To recover positional notation, assume a set of labels $\ell_{\mathbf{1}}, \ell_{\mathbf{2}}, \ell_{\mathbf{3}}, \ldots$ chosen so that they do not occur free in some term P written using positional notation, and assume also that for any sequence, $\vec{e} = e\, e'\, e'', \ldots$, the labeling function $lab(\vec{e})$ takes a sequence of expressions and creates a sequence of locked fields, $lab(\vec{e}) \stackrel{def}{=} \ell_{\mathbf{1}} : e\ \ell_{\mathbf{2}} : e'\ \ell_{\mathbf{3}} : e'', \ldots$. The idea behind the translation is to label all fields of objects in P with these keys; for instance $\mathbf{out}\,\langle\, e\ e'\,\rangle$ becomes $\mathbf{out}\,\langle\, \ell_{\mathbf{1}} : e\ \ell_{\mathbf{2}} : e'\,\rangle$. Inputs are labeled in a similar way. Pattern matching lines up fields in the right order. We nevertheless have to take care to prevent a template matching a longer object. This is done by encoding the length of the tuple in an additional field. The correct translation for a tuple output operation becomes $\mathbf{out}\,\langle\, \ell_{\mathbf{1}} : e\ \ell_{\mathbf{2}} : e'\ \ell_{\mathbf{len}} : \ell_{\mathbf{2}}\,\rangle$. With this translation, it is obvious that positional notation may be mixed with locked fields, giving a rich choice of programming styles.

4.2 Secure Communication Protocols

The process of securing a system begins with the task of identifying the resources or information to be protected, and the most likely sources of attacks. Only then can suitable protection mechanisms be designed. We apply this principle to a simple example to identify some of the weaknesses of Linda with respect to security.

The following streaming protocol (in pseudo-code) is typical of generative communication. Its goal is to transfer the contents of an array (`v`) from a sender process to some other receiver process. The sender process starts by output a header message that contains an identifier for this particular protocol run (here some value `j`), and the length of the array (`i`). Then each array element is output together with the run identifier. Furthermore, in order to enforce ordering each tuple also contains the sequence number of the element (the complete tuple is ⟨`j,i,v[i]`⟩).

ProcA	**ProcB**
out⟨`j,i`⟩	**in**⟨?,?⟩`,x`
	`i = x.2`
while`(i--)`	**while**`(i--)`
`  `**out**⟨`j,i,v[i]`⟩	`  `**in**⟨`x.1,i,`?⟩`, y[i]`

Fig. 1. Streaming protocol

The receiver uses the run identifier and the sequence number to query the data space for each array element.

There are four security properties relevant to this protocol. The first property is **authenticity**, the sender (resp. receiver) may require that its partner be a particular process. Thus both parties may have to be authenticated to one another. **Privacy** protects the data exchanged against disclosure, this means that each value `v[i]` must be hidden from processes other than the designated receiver. **Integrity** implies that no process should interfere with the protocol, *e.g.,* by outputting tuples that the designated receiver believes to have been placed by the designated sender. And finally, **availability** to ensure that no process other than the receiver may remove a tuple containing a `v[i]` from the space, as this would prevent the receiver process from continuing.

Linda-based coordination models can not provide such security guarantees. The very nature of generative communication allows a malicious process to mount attacks against every one of these properties. We proceed to show examples of secure protocols in our calculus.

4.2.1 Message Privacy

The simplest example is one where two processes exchange data that no other process should read. For this, secure spaces operations can be viewed as providing protection analogous to cryptography. In order for two process to be able to exchange private messages, they need to share a symmetric key. The following configuration is an example,

$$(\mathbf{new}\, a_a)\Big(\mathbf{out}\,\langle\, a_a : e\,\rangle \mid P \quad | \quad \mathbf{in}\,\langle\,\rangle\, x\,\mathbf{.}\, Q\Big)$$

If process Q holds the key a_a then it may retrieve the payload of the object $\langle\, a_a : e\,\rangle$. Of course, nothing prevents another process from matching the object with the empty template, thus disrupting the protocol. A malicious process may also copy the object and try to replay it later, but this can be prevented by traditional means such as adding a nonce to the data.

4.2.2 Message Authenticity

Guaranteeing authenticity of messages in open networks is often by digitally signing messages with the private key of the sender. The sender's public key may then be used to check that the message is authentic and that its contents are intact. We adapt this idea to secure spaces, using pattern matching. To sign an object $\langle\, \vec{f}\,\rangle$ with the key a_b, the sender process executes $\mathrm{sign}(\mathbf{out}\,\langle\, \vec{f}\,\rangle, a_b, b_a)$. The intuition is that a signed object contains an extra signature field which matches the signed object's payload. So, that in order

to authenticate the message, the receiver needs only extract the signature and match it with the message. The sign function is defined as,

$$\begin{aligned}&\mathrm{sign}(\mathbf{out}\,\langle\,\vec{f}\,\rangle, a_b, b_a) \quad \stackrel{def}{=} \\ &\quad (\mathbf{new}\,c'_{c'})(\mathbf{new}\,c_c)\;\mathbf{out}\,\mathrm{mark}(\langle\,\vec{f}\;\; a_b : c_c\;\; c_c : \langle\,\mathrm{inverse}(\vec{f})\;\; b_a : c_c\;\; c_c : c_c\,\rangle\,\rangle, c'_{c'}) \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad c_c, c'_{c'} \notin fn(\vec{f})\end{aligned}$$

The auxiliary function mark is defined inductively on values,

$$\begin{aligned}\mathrm{mark}(\langle\,\vec{f}\,\rangle, c'_{c'}) &= \langle\,\mathrm{mark}(\vec{f}, c'_{c'})\;\; c'_{c'} : c'_{c'}\,\rangle \\ \mathrm{mark}(\ell : v\;\vec{f}, c'_{c'}) &= \ell : \mathrm{mark}(v, c'_{c'})\;\mathrm{mark}(\vec{f}, c'_{c'}) \\ \mathrm{mark}(\ell, c'_{c'}) &= \ell \\ \mathrm{mark}(?, c'_{c'}) &= ?\end{aligned}$$

The function's role is to add an extra field to all objects in $\vec{f}$. The auxiliary function inverse is defined inductively on values as follows,

$$\begin{aligned}\mathrm{inverse}(\langle\,\vec{f}\,\rangle) &= \langle\,\mathrm{inverse}(\vec{f})\,\rangle \\ \mathrm{inverse}(\ell : v\;\vec{f}) &= \overline{\ell} : \mathrm{inverse}(v)\;\mathrm{inverse}(\vec{f}) \\ \mathrm{inverse}(\ell) &= \ell \\ \mathrm{inverse}(?) &= ?\end{aligned}$$

The inverse function creates a matching replica of the payload of the message by recursively inverting all field labels. In an implementation of secure spaces the inverse function would have to be built-in, and would not be made directly available to untrusted process as it could be used to construct templates with keys to which the process does not have access. The mark function is used to to tie the value to its signature so that the signature may only be used to match the value and vice versa. This prevents misuse of the signature.

To illustrate the signing of a message, consider the output term

$$\mathrm{sign}(\mathbf{out}\,\langle\,d_d : d'_{d''}\;\; d'_{d''} : \langle\,d_d : ?\,\rangle\,\rangle, a_b, b_a)$$

Here the signed message will be the object

$$(\mathbf{new}\, {c'}_{c'})(\mathbf{new}\, c_c)\ \mathbf{out}\ \langle\ d_d : {d'}_{d''}$$
$$\quad {d'}_{d''} : \langle\, d_d : ?\ \ {c'}_{c'} : {c'}_{c'}\,\rangle$$
$$\quad a_b : c_c$$
$$\quad c_c : \langle\, d_d : {d'}_{d''}\ {d''}_{d'} : \langle\, d_d : ?\ {c'}_{c'} : \langle\,\rangle\,\rangle\ \ b_a : c_c\ \ c_c : c_c\ \ {c'}_{c'} : {c'}_{c'}\,\rangle$$
$$\quad {c'}_{c'} : {c'}_{c'}\rangle$$

Notice that the payload is intact, but there are two extra fields, the first is locked under the private key a_b and contains a fresh symmetric key c_c, the second locked under c_c contains an almost exact replica of the object except that all asymmetric keys have been replaced by their inverse and that the field locked under c_c holds an empty object. The ${c'}_{c'}$ field has been added for technical reasons, without it a process could use the signature as a template.

The receiver process has in its possession the public key b_a. To authenticate a message, the receiver will use $\text{authenticate}(e, b_a)\,.\,P$ which blocks if e does not evaluate to a message signed by a_b. The definition of authenticate is

$$\begin{aligned}
&\text{authenticate}(e, b_a)\,.\,P \quad \stackrel{def}{=} \\
&\qquad \mathbf{out}\, e \oplus (e.b_a) : (e.b_a) \ \ \mid \\
&\qquad\quad \mathbf{in}\, e.(e.b_a)\, x\,.\,(\ \mathbf{out}\, x.(x.b_a)\ \ \mid\ \ \mathbf{in}\, (x \oplus (x.b_a) : (x.b_a)\,)y\,.\,P)\Big)
\end{aligned}$$

The variables x and y are chosen so that the do not occur free in P.

A message is considered authentic if an only if, it has exactly the same number of fields as when it was signed and all of the field values are authentic. Luckily, pattern matching performs this check. We have constructed two objects that should be identical, modulo asymmetric keys, and we will use each of them in turn as a template to match the other. If both matches succeed then the message is authentic and P can proceed. If we consider the example term given above, let $e = \langle d_d : {d'}_{d''}\ \ {d'}_{d''} : \langle\, d_d : ?\ {c'}_{c'} : {c'}_{c'}\,\rangle\ \ a_b : c_c\ \ c_c : \langle d_d : {d'}_{d''}\ {d''}_{d'} : \langle\, d_d : ?\ {c'}_{c'} : {c'}_{c'}\,\rangle\ \ b_a : c_c\ \ c_c : c_c\ \ {c'}_{c'} : {c'}_{c'}\rangle\ \ {c'}_{c'} : {c'}_{c'}\rangle$ the selection expression $e.(e.b_a)$ yields the signature $\langle\, d_d : {d'}_{d''}\ {d''}_{d'} : \langle\, d_d : ?\ {c'}_{c'} : {c'}_{c'}\,\rangle\ \ b_a : c_c\ \ c_c : c_c\ \ {c'}_{c'} : {c'}_{c'}\,\rangle$. As such these objects do not match because of their c_c fields. The object extension expression $e \oplus (e.b_a) : (e.b_a)$ overwrites the value of that field and makes the object match one another. Thus it is easy to check that

$$e \oplus (e.b_a) : (e.b_a) \quad \leq \quad e.(e.b_a)$$

and

$$e.(e.b_a) \quad \leq \quad e \oplus (e.b_a) : (e.b_a)$$

both hold.

The encoding is not entirely correct as a third party might disrupt the protocol by inputing one of these objects using an empty template. The solution to prevent accidental matches is to protect the objects with an o-lock. The correct encoding of authenticate is thus,

$$
\begin{aligned}
&\text{authenticate}(e, b_a)\,.\,P \quad \stackrel{def}{=} \\
&\quad (\mathbf{new}\, c_c)\Big(\ \mathbf{out}\,(\langle\, c_c : e \,\rangle @ c_c) \oplus (e.b_a) : (e.b_a) \ \ \mid \\
&\qquad\qquad \mathbf{in}\,\langle\, c_c : e.(e.b_a) \,\rangle @ c_c\, x\,.\,(\ \ \mathbf{out}\,\langle\, c_c : x.(x.b_a) \,\rangle @ c_c \ \ \mid \\
&\qquad\qquad\qquad\qquad \mathbf{in}\,(\langle\, c_c : x \,\rangle @ c_c) \oplus (x.b_a) : (x.b_a)\, y\,.\,P)\Big)
\end{aligned}
$$

This encoding ensures that only the object yielded by e is considered for authentication, and P will proceed if the object has been signed with key a_b.

4.2.3 Secure Channels

To ensure integrity and availability, an abstraction of secure channels should be provided. A secure channel is a communication abstraction between two processes that ensures no other process may read or write to that channel. We will demonstrate how to set up a secure channel between two arbitrary processes P and Q using the secure space primitives. Our only assumption is that one of the processes, for instance P which we call the initiator of the protocol, knows the other process' public key, *e.g.,* a_b. This process will set up a secure channel by first executing the establish(x, a_b, c_d, d_c) protocol, where x is the channel identifier, a_b is the interlocutor's public key, c_d is the initiator's public key and d_c the corresponding private key. Once a connection has been established, the processes can use send(x, e) to send an object e over secure channel x an recv(x, e) to receive an object from the secure channel.

The implementation relies on o-locks to protect the data being exchanged, so that for every send(x, e) there is an $\mathbf{out}\, e@\ell$ for some shared key ℓ. The crux of the protocol is to guarantee that ℓ is not divulged. The encoding of the send and receive operations are quite simple. If we assume that $\ell_{\texttt{chn}}$ is a symmetric key and that x will evaluate to an object with a $\ell_{\texttt{chn}}$ field containing the shared key used to for that particular channel, then the encoding of the operation is:

$$
\begin{aligned}
\text{send}(x, e) \quad &\stackrel{def}{=} \quad \mathbf{out}\, e@(x.\ell_{\texttt{chn}}) \\
\text{recv}(x, e, y)\,.\,P \quad &\stackrel{def}{=} \quad \mathbf{in}\, e@(x.\ell_{\texttt{chn}})\, y\,.\,P
\end{aligned}
$$

To establish a session the initiator will create a symmetric key that will be used in the o-lock and output an object $\langle\, a_b : \langle\, \ell_{\texttt{pub}} : c_d\ \ell_{\texttt{chn}} : d'_{d'} \,\rangle \,\rangle$ containing its own public key (locked under $\ell_{\texttt{pub}}$) and the channel key (locked under $\ell_{\texttt{chn}}$).

This information is itself locked with the public key of the other process (a_b). The whole object is signed with the initiator's private key (d_c). The initiator then waits for an acknowledgment message which it authenticates with the other party's public key. The acknowledgment is expected to contain a $\ell_{\texttt{chn}}$ field.

$$\begin{aligned}&\text{establish}(x, a_b, c_d, d_c)\,.\,P \quad \stackrel{def}{=} \\ &\qquad (\mathbf{new}\, d'_{d'})\,\Big(\,\text{sign}(\mathbf{out}\,\langle\, a_b : \langle\, \ell_{\texttt{pub}} : c_d\ \ell_{\texttt{chn}} : d'_{d'}\,\rangle\,\rangle, d_c, c_d)\ \ \mid \\ &\qquad\qquad \mathbf{in}\,\langle\,\rangle@d'_{d'}\,x\,.\,\text{authenticate}(x, a_b)\,.\,P\,\Big)\end{aligned}$$

A process willing to accept a secure connection will run the accept(x, b_a) protocol, where x will be used as the channel identifier and b_a is a private key. The protocol starts by reading an object that matches $\langle\, b_a : ?\,\rangle$, that is to say, an object with at least a field locked under the public key a_b. The process extracts the $\ell_{\texttt{pub}}$ field from that object and uses it to authenticate the message. The next step of the protocol is to extract the value of the a_b field and bind it to variable x. Finally, the protocol sends x as an acknowledgment signing it with it own private key.

$$\begin{aligned}&\text{accept}(x, b_a)\,.\,P \quad \stackrel{def}{=} \\ &\quad \mathbf{in}\,\langle\, b_a : ?\,\rangle\, y\,.\,\text{authenticate}(y, (y.b_a).\ell_{\texttt{pub}})\,. \\ &\qquad (\mathbf{new}\, c_c)\,\Big(\,\mathbf{out}\,(y.b_a)@c_c\ \mid\ \mathbf{in}\,\langle\,\rangle@c_c\,x\,.\,(\text{sign}(\text{send}(x, x), b_a, a_b)\ \mid\ P)\Big)\end{aligned}$$

4.3 Memory Management for Shared Spaces

Memory management is an important issue for shared data space implementations. This, partly because spaces are long lived data structures, so any accidental memory leak will persist, but also because of the danger of denial of service attacks. Generative communication precludes traditional garbage collection techniques since unlike pointer based data structures there is no clearcut concept of reachability in a shared space. One partial solution to this problem has been adopted by JavaSpaces [11], namely to associate a time-to-live (TTL) with each object deposited in a JavaSpace, once the TTL reaches 0, the object is removed from the space and its memory is reclaimed. While this policy may work well in some cases, it still does not prevent one or more processes to mount a denial of service attack. Furthermore the TTLs presuppose that it is possible to estimate beforehand how long a particular object will be useful. Such estimates are of course very difficult. In some cases, it may be more appropriate to be able to reclaim all the objects generated by a particular process. For example, if a process violates a security policy, all of

the objects it placed in the space might need to be reclaimed. For other applications, it may be desirable to clean up any object left over after a particular protocol run.

Clearly some flexibility is required. Secure spaces allow to implement all of these policies as user-level programs. In other words, there is no need to hardwire any particular policy. Instead different applications can run different memory reclamation algorithms concurrently. The key to a user-level implementation is twofold. Firstly, the extra information needed for reclamation, *e.g.,* TTLs or ownership, must be encoded in each object so that it is accessible to the collector but transparent from the application. Secondly, uncooperative applications should not be able to trick the collector, nor should the reclamation algorithm be able to gain information about the contents of the objects is collecting.

4.3.1 Tagged Object Collection

The simplest of memory reclamation schemes is for each application to voluntarily tag every object it outputs, with a time-to-live for instance, and every so often to run a collector process that locates all objects with a particular tag and removes them from the space. The idea is simple, every output term will be marked by executing $\mathrm{tag}(\mathbf{out}\, e, a_a)$, where a_a should be a fresh symmetric key that will play the role of a tag (*e.g.,* a TTL). Then to reclaim all objects tagged with a_a, the process need only execute $\mathrm{collect}(a_a)$.

The encoding of these operations is straightforward. Tagging implies the creation of a fresh symmetric key $\ell_{\mathtt{GC}}$ and extension of the output object with the s-lock $\ell_{\mathtt{GC}} : a_a$. Using a new label guarantees that the field is hidden from other processes and also prevent an attacker from trying to overwrite the field with a fake tag. In parallel with the output, a second process is started. This process performs the garbage collection. It first tries to input a trigger object o-locked with a_a. It then proceeds to reclaim one tagged object, and releases outputs the trigger to allow further collection. The encoding of the $\mathrm{collect}(a_a)$ is then simply to output a trigger object to start the collection. It is inter-

$$\mathrm{tag}(\mathbf{out}\, e, a_a) = (\mathbf{new}\, \ell_{\mathtt{GC}}) \Big(\mathbf{out}\, e \oplus \ell_{\mathtt{GC}} : a_a \;\; | \;\; \mathbf{in}\, \langle\, \rangle @ a_a\, x \,.\, (\mathbf{out}\, x \;\; | \;\; \mathbf{in}\, \langle\, \ell_{\mathtt{GC}} : a_a \,\rangle\;) \Big)$$

$$\mathrm{collect}(a_a) = \mathbf{out}\, \langle\, \rangle @ a_a$$

Table 5: Marking and collection.

esting to note that tagging can be performed without gaining any knowledge about the contents of the object being tagged. This approach is quite flexible,

multiple tags can be applied to the same object without risk of interference, and the applications that operate on the object need not be aware that tags are present. This means that different collection policies may be composed.

4.3.2 Object Revocation

Tagging is a basic mechanisms that can be used to implement collection policies. One example policy is revocation on exit, that is, when an application terminates all of the objects it generated that are still in the shared space must be removed. To reclaim objects in the shared space, it is necessary to be able to differentiate between objects that belong to distinct applications. We define a translation scheme that marks each application with a different key. Thus in the case we have two processes P and Q, the term $[\![P]\!]_{p_p} \mid [\![Q]\!]_{q_q}$ denotes the composition of the two processes where all outputs are properly tagged as defined in the table below. Revocation of a process simply entail invoking collect(p_p).

$$
\begin{aligned}
[\![\mathbf{out}\, e]\!]_{a_a} &= \mathrm{tag}(\mathbf{out}\, e, x) \\
[\![\mathbf{in}\, e\, y\, \mathbf{.}\, P]\!]_{a_a} &= \mathbf{in}\, e\, y.[\![P]\!]_{a_a} \\
[\![(\mathbf{new}\, b_c) P]\!]_{a_a} &= (\mathbf{new}\, b_c)[\![P]\!]_{a_a} \\
[\![P \mid Q]\!]_{a_a} &= [\![P]\!]_{a_a} \mid [\![Q]\!]_{a_a} \\
[\![\mathbf{!}\, P]\!]_{a_a} &= \mathbf{!}\, [\![P]\!]_{a_a} \\
[\![\mathbf{0}]\!]_{a_a} &= \mathbf{0}
\end{aligned}
$$

Table 6: Process marking.

4.3.3 Hygienic Protocols

Another reclamation policy is to enforce protocol hygiene, that is to say each protocol run must ensure that no outstanding message is left in the shared space after the protocol's end. A technique for ensuring hygiene is to rely on a post protocol clean up scheme. The idea requires that each participant in a protocol tags all of outputs that belong to a given run. And when the protocol completes successfully or aborts, a clean up procedure is invoked. A hygienic protocol is declared with **newprot** a_a, where a_a is a fresh protocol name. For each message of the protocol, the sender simply writes **out** e_{a_a}. The protocol can then be closed by **endprot** a_a. The encoding of these operations is simple. All outputs in the protocol are tagged and when the protocol terminates a collection is triggered.

$\mathbf{out}\, e_{a_a}$	$\stackrel{def}{=}$	$\mathrm{tag}(\mathbf{out}\, e, a_a)$
$(\mathbf{newprot}\, a_a)P$	$\stackrel{def}{=}$	$(\mathbf{new}\, a_a)P$
$(\mathbf{endprot}\, a_a)$	$\stackrel{def}{=}$	$collect(a_a)$

Table 7: Hygienic protocols.

One interesting point about these marking schemes and the associated collectors is that they are compositional. Several collection algorithms may run concurrently in the same secure space without application interference. The collectors are trusted to the point of being able to reclaim objects but no more; in particular, they cannot observe information contained in the objects that they collect.

4.4 Summary

We have presented Secure Spaces – a coordination model that extends generative communication with fine grained access control primitives. We have shown that these primitives are adequate for implementing more powerful security mechanisms, and that we have lost none of the expressiveness of Linda. We now turn to the question of practicality and argue that our extensions can be efficiently implemented and used in a mainstream programming language such as Java.

5 The SECOS Coordination Infrastructure

Secure spaces have been implemented in Java. The implementation, termed SecOS, was originally conceived for and used in the JavaSeal mobile agent platform [4] as an agent communication mechanism for untrusted agents running on the same platform. The implementation is a single machine implementation. We have experimented with extensions for a distributed setting, but several issues dealing with the secrecy of keys remain open. This section overviews the SecOS interface and then discusses implementation and efficiency considerations.

The public interface of SecOS has been kept simple and small. The three classes shown in Figure 2 are the only classes visible to users of the system. `SSpace` implements the shared data space, space objects are represented by the class `SObject` and finally labels are represented by the `SKey` class.

Data spaces are created with the factory method `makeSSpace` which returns a reference to a data space. In the current implementation, only one data space can be created in each virtual machine. To ensure this, the `SSpace` constructor is not public. Spaces have three public methods, `out` to output an object, `in` to perform a destructive read, and `read` to perform a non-destructive read. `inp` and `readp` are non-blocking variants of the above operations, they return `null` if the template could not be matched.

The `SObject` class is an immutable container. Once an object has been created it is guaranteed not to change. Immutability is an essential property for the implementation, as the shared space builds indices that rely on the fact that the keys and values in an object are not modified after it is inserted in the

```
public final class SSpace {
    public synchronized void out( SObject );
    public synchronized SObject in( SObject );
    public synchronized SObject read( SObject );
    public synchronized SObject inp( SObject );
    public synchronized SObject readp( SObject );
    public SSpace makeSSpace();
    SSpace();

}

public final class SObject {
    public SObject();
    public Serializable select( SKey );
    public SObject add( SKey, Serializable );
    public void olock( SKey );
}

public final class SKey implements Serializable {
    static public SKey makeSymKey();
    static public SKey[] makeAsymKeys();
    SKey();
}
```

Fig. 2. SecOS public classes.

space. An `SObject` contains an array of fields that are key-value pairs. The keys must be a `SKey` instance while the value are subtypes of `Serializable`. An `SObject` can be extended with additional fields by invoking the `add` method; this returns a copy of the original object. Values can be selected from an object by invoking `select` with a key as argument. If the object contains a field locked with that key, then the corresponding value will be extracted, otherwise `null` is returned. Finally, an object can be locked with the `olock` method. This method takes a key as argument and returns a new object o-locked under that key.

The `SKey` class stands for symmetric and asymmetric keys. There are no public operations on keys. Keys are created by two factory methods `makeSymKey` and `makeAsymKeys`, the latter returns a two key array containing both asymmetric keys.

Since a Java virtual machine allows untrusted code to inhabit the same address space as the implementation of SecOS, some security measures are necessary. All three classes in the interface are final to prevent attacks that use subtyping to inject malicious code into the kernel. Furthermore, the design of SecOS ensures that it is not possible for user code to modify the state of the SecOS system by other means than invoking the public methods in the interface. It also ensures that no user code will execute within the SecOS kernel. The first property is obtained by enforcing that values exchanged between the user code and the kernel is either immutable or copied. Thus, objects and keys are immutable, and field values are serialized. Serialization ensures that there is no sharing of mutable values between application and kernel. The use of serialization also means that object equality checks are performed without running the methods of these objects, instead the default bitwise comparison of serialized values is used.

Recursive pattern matching is not implemented in the current system, and the `SObject` class does not implement the `Serializable` interface.

5.1.1 A Streaming Protocol

A streaming protocol transfers an ordered sequence of messages between applications. This protocol is a representative example of generative communication. We give a straightforward implementation in SecOS consisting of two methods `streamOut` to output a vector of objects to the space, and `streamIn` to read the ordered vector from the space. An example of these methods is,

```
SKey sid = SKey.makeSymKey();
streamOut(space, sid, new SObject(), values);
Vectors results = streamIn( space, sid, new SObject());
// we have results.equals(values)
```

The `streamOut` method is shown in Figure 3. The method takes four arguments, a space, a key, a template object and the vector of values to transfer. The key `streamID` is the identifier for this particular run of the protocol. It should be distinct from any other instance of the same protocol. The method begins with the creation of new key, `valKey`, to be used as the label for value fields. The first object output is a header,

$$\langle \texttt{START\_STREAM : streamID, streamID : valKey} \rangle$$

The static variable `START_STREAM` is globally visible key. Every element of the `val` vector will then be output together with a sequence number in the format

$$\langle \texttt{streamID : i, valKey : val.elementAt(i)} \rangle$$

The stream is terminated with the object

$$\langle \texttt{streamID : i+1, END\_STREAM : END\_STREAM} \rangle$$

```
void streamOut(SSpace space, SKey streamID, SObject obj, Vector val) {
    SKey valKey = SKey.makeSymKey();

    space.out(obj.add(START_STREAM, streamID)
                 .add(streamID, valKey));

    for(int i=0; i<val.size(); i++)
        space.out(obj.add(valKey, val.elementAt(i))
                     .add(streamID, new Integer(i)));

    space.out(obj.add(streamID, new Integer(i+1))
                 .add(END_STREAM, END_STREAM));
}
```

Fig. 3. Procedure for sending an ordered sequence of objects.

On the receiving end, the `streamIn` method shown in Figure 4 takes three arguments, a space, a stream identifier and a template object. It starts by reading the header object and extracting `valKey`. The method matches objects using the `streamID` and sequence number. The end of stream is detected when an object has a non-null `END_STREAM` field.

While this protocol is effective in transferring an ordered sequence of objects between cooperating processes, it is still susceptible to interferences. An unrelated process may input an element of the stream using an empty template; then the protocol gets stuck on a missing value.

The solution to prevent interferences is to use o-locks. In fact, the template argument of the two methods allows us to reuse the code to implement a

```
Vector streamIn(SSpace space, SKey streamID, SObject template) {

    SObject first = space.in(template.add(START_STREAM, streamID));
    SKey valKey = first.select(streamID);
    Vector values = new Vector();

    int i = 0;
    while(true) {
        SObject msg = space.in(template.add(streamID, new Integer(i++)));
        if (msg.select(END_STREAM)== null)
            values.add(msg.select(valKey));
        else break;
    }

    return values;
}
```

Fig. 4. Procedure for reading an ordered sequence of objects.

secure stream. All that is needed is to call the method with matching o-locked templates. Consider the following invocation,

```
SKey[] ks = SKey.makeAsymKeys();
streamOut(space, streamID, new SObject().olock(ks[1]), val);
streamIn(space, streamID, new SObject().olock(ks[2]));
```

Every value output by `streamOut` will be a copy of the o-locked object `new SObject().olock(ks[1])`, where `ks[1]` is an asymmetric key. Every value input by `streamIn` will use a template that extends `new SObject().olock(ks[2])`, where `ks[2]` is the inverse of the key used for output.

5.2 SecOS *Implementation*

The two basic space operations, `out` and particularly `in`, are bottlenecks for the performance of secure spaces. We describe our approach to an efficient implementation and then give some performance results.

The main efficiency problem for any of the shared data space infrastructures is the matching of tuples that occurs during the `in` operation. In SecOS, matching is even more difficult than in Linda due to the extended pattern matching rules: order irrelevant and shorter templates may match a longer object. Linda implementations can at least disregard tuples of different lengths during matching.

The first step to making the secure space operations efficient is to have a fast inequality test. Since for any given query we expect most objects not to match the template, it is essential to be able to prune the search space efficiently. To achieve this, each `SObject` has two associated fingerprints. Fingerprints are bit strings with the property that if two objects match, $o \leq o'$, then $fp(o)$ is a subset of $fp(o')$. We use one 64 bit fingerprint to summarize an object's keys and another for its values. The fingerprints are computed by compacting every key and value down to a single bit. The implementation uses the `hashCode()` function. Both fingerprints are thus computed as follows,

```
for (int j = 0; j < keys.length; j++) {
    keysFP = keysFP | (1 << (keys[j].hashCode() %64));
    if ( val[j] != null)
        valsFP = valsFP | (1 << (val[j].hashCode() %64));
}
```

An `SObject` contains two arrays, one with keys and the other with values. Values can be null if the field was set to the wild card (? in the calculus).

The fast inequality test for an object `sobj` and a template `templ` is thus simply,

```
( templ.keysFP == ( templ.keysFP & sobj.keysFP ) ) &&
( templ.valsFP == ( templ.valsFP & sobj.valsFP ) )
```

If the test is negative then we know for sure that template `templ` cannot match with object `sobj`. A positive answer only indicates that there might be a match.

The second objective for efficiency is to avoid having to compare a template against all objects in the space. The SecOS implementation uses a binary search tree to prune the search space. Each object is associated with a 16 bit summary, computed as the union of the keys and values fingerprints,

```
short summary(long vfp, long kfp) {
    short sum = 0;
    for (int i = 0, j = 0; i < 64; i += 4, j++)
        sum = sum | (((7<<i)& vfp) << j) |
                    (((7<<i)& kfp) << j);
    return sum;
}
```

This 16 bit value is used to choose a leaf in the search tree where to store the object (it prescribes a path in the binary tree). The search tree is built lazily and empty branches are removed when detected in queries.

For any given query, with a template object `templ`, the search algorithm first computes the summary of `templ`. This summary will be used to prune the

search space. If bit n of the summary is 1, then we need only visit one branch at level n of the tree, if it is 0 both branches must be taken.

For any given query, we compute a 16 bit summary for the template object and use that summary to drive the search procedure. We need to visit only leaves that may match the template. The search space is further pruned by keeping, at each non-leaf node of the tree, the union of all fingerprints that have traversed that node. An inequality test is performed comparing this union with template's fingerprints. If they fail to match then the search need not proceed further down this path. When a leaf node is reached the list of objects at that node is searched reverse order of insertion.

The current implementation is tuned for moderately large spaces, on the order of 20K objects. We have constructed synthetic benchmarks to evaluate performance and scalability of the basic operations. The benchmark results are summarized in Figures 5-7. The tests were run on a dual 800Mhz Intel PIII machine with 512MB main memory using the IBM 1.3.0 virtual machine.

The first benchmark measures performance of the `in` operation for data spaces of different sizes. The secure spaces are populated with randomly generated objects and the measure shows the average time for the `in` of the last inserted object. For each space size we measured average times of 1000 different queries with different object and template sizes and characteristics. Performance is shown in operations per second. The results are relatively constant with respect to space size with rates above 90Kops/sec in all cases.

The second benchmark assess performance of the `read` operation. It measures the average cost of locating an arbitrary object in the data space. The performance drops significantly as the space size grows which is surprising as one would expect `read` to be faster than `in` (it does not have to update the internal data structures). Analysis of the data reveals that the median cost `read` is indeed slightly less than `in`, but the average is pulled down by the extreme values. The main difference between the two benchmarks is that we are not retrieving the last inserted object, instead we query for random objects in the data space. So, in the worst case we have to traverse and check the complete list of `SObject`s stored at a leaf node. For spaces of smaller than 10K objects we can nevertheless expect more than 30Kops/sec.

The last benchmark measures the performance of the streaming protocol presented in Section 5.1.1. The measure indicates the average time for constructing an object, writing it to the space, then constructing a template and retrieving a matching object. The data shows that for spaces up to 30K objects, it is possible to exchange more than 3K objects per second.

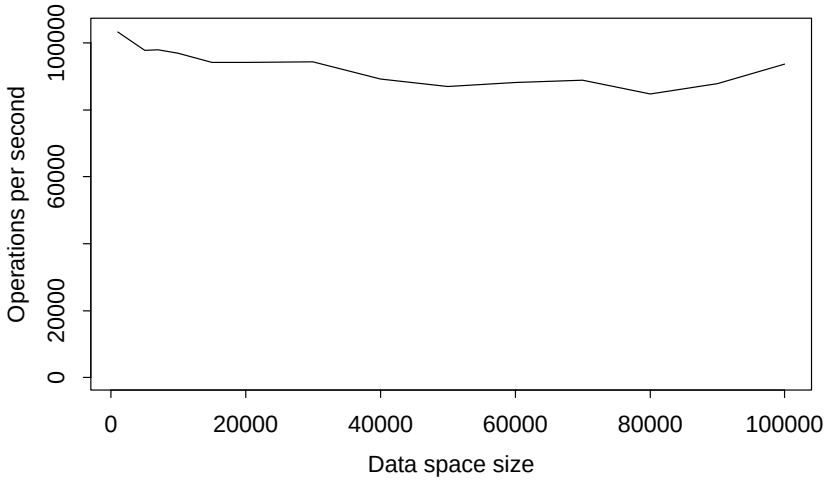

Fig. 5. Performance of the SecOS `in` operation.

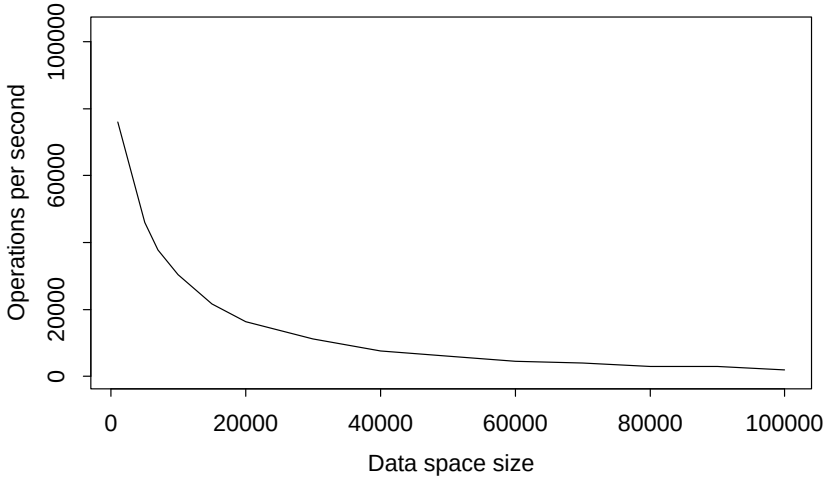

Fig. 6. Performance of the SecOS `read` operation.

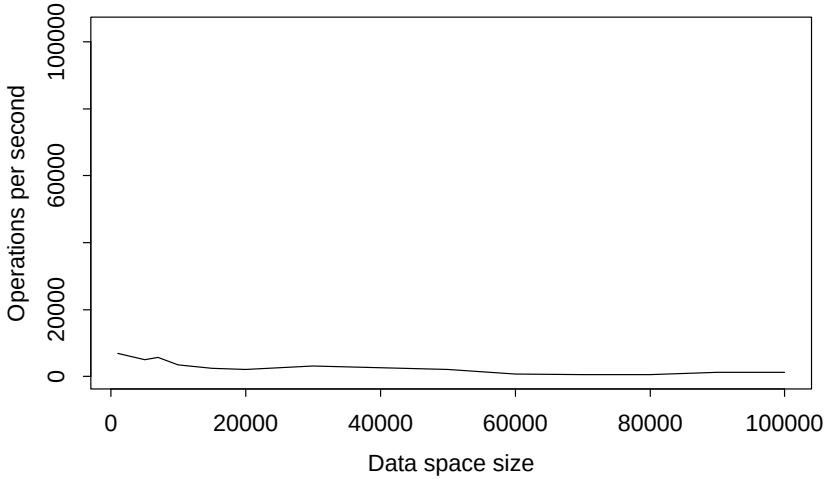

Fig. 7. Streaming protocol.

6 Related Work

Pinakis proposed a solution for secure directed communication in a Linda distributed environment [22]. His approach is based on a new data type called a *ticket*. Tickets are similar to capabilities of the Amoeba operating system, a message containing a public ticket can only be matched by the appropriate private ticket. Our work can be viewed as a generalization of this approach.

Minsky *et al.* present a law-governed approach to controlling interactions in tuplespaces in which Prolog rules specify constraints on the behavior of distributed agents interacting through a shared tuplespace [19]. Security of the overall system is enforced by controller running on secure co-processors at each client site. The main difference with our work is that the rules governing a system must be agreed upon by mutual concensus and are static, whereas in our approach different parts of a shared space can implement different access control policies and modify these policies dynamically.

KLAIM is an effort to extend the shared space model with protection [20]. The approach taken prioritizes static verification of security. Processes manipulate tuples at abstract locations in a network. Access control policies specify what a process can do at each site, *e.g.,* whether it can read or write tuples. At each site a type checker analyses the code of process to deduce its intentions – the set of access rights that it actually needs at the location – and only accepts a process on a site if it does not require more rights than permitted by the security policy for that site. Static security analysis is useful in many circumstances since it avoids the need for mechanisms at runtime. Nevertheless, static verification is hardly sufficient in the Internet context. Firstly, there are no guarantees that a process loaded is not subsequently modified by an attacker to exceed the privileges accorded to it. Secondly, there is no centralized allocator process that all clients can trust. KLAIM is better suited to a local area networked system with a single trusted administrator and where machines can be protected from tampering from users. In addition, access rights in KLAIM are used to control a process reading or writing tuples at a site; they do not control access to individual tuples. This restricts the flexibility of the mechanism since protocols such as secure channels or memory management cannot easily be programmed.

Our model has many similarities to the sealed object proposal by Gifford where objects are protected by sealing them with keys [14]. Gifford's proposal aims to guarantee secrecy and authentication for objects in a distributed system. Secrecy is enforced by the property that an object can only be unsealed, to be read or modified, by furnishing the correct key; authentication means that keys cannot be fabricated so once an object is sealed, only the correct matching key can unseal it. Our model also implements these properties. Gifford's proposal

is aimed at distributed systems made up of mistrusting nodes, where data must be encrypted as soon as it leaves a site. Gifford's basic key model is richer than that of Secure Spaces. As well as symmetric and asymmetric keys, objects can be sealed with a key quorum or indirect key. A key quorum is a set of keys used to seal an object, where a specified number of keys from this set is sufficient to unseal the object. An indirect key is generated from another key and can unseal an object sealed with the base key. The base key associated with an indirect key can be changed, which simplifies key management in the system. A key quorum and indirect keys can be used to model capabilities and access control lists. The main difference in approaches is that Gifford's proposal is implemented with cryptography, while in SecOS the checks are performed by the kernel.

Another system with similarities to ours is the spi calculus by Abadi and Gordon [1]. This is an extension of the π-calculus with primitives that encrypt and decrypt messages sent over channels. Though scope in π is powerful enough to express access control over channel names, encryption is awkward to express. Adding encryption primitives allows one to reason about the secrecy and authentication of security protocols. In secure spaces the inclusion of locking primitives is a matter of necessity rather than choice, because of the visibility of spaces to processes. Further, since the shared data space model is still a non-standard communication means, we chose to address more attention to language design issues, rather than to studying proof systems for verification of security properties.

The role of subsumption and types in the matching process has received much attention recently. The Laura system, for instance, is a WAN service architecture based on the shared space model [26]. One reason why the shared space paradigm is exploited is that it allows services to join and leave the system dynamically. Services place offers in the space which are matched with requests. An offer or request is an *interface* form that matches if the type of the service is a subtype of the requestor's. Alice is the type system employed for matching these interfaces [24]. Dami also investigates type inference for generative communication [9].

As regards implementing the shared object paradigm in Java, we can mention JavaSpaces [11] and Jada [8]. Jada is one example of the shared space paradigm being used to coordinate mobile agents: it is employed in the PageSpace agent architecture [7]. Neither Jada nor JavaSpaces were designed with security issues in mind. Though keys can be employed to protect items in the tuple from agents, this can only be done using encryption algorithms, even for agents executing within the same JVM, which is too inefficient for generalized use.

Sun's JavaSpaces [11] has support for memory management. In this system,

a process associates a *lease* with a tuple that it deposits in the tuple space; the lease specifies the life span of that tuple. This policy is hardwired into the implementation; the question of how reasonable limits for objects are determined is not addressed. It is quite likely that different applications will require different leases; leases can vary for some several seconds to several weeks, months or even longer. In JavaSpaces, the garbage collector is executed within the trusted computing base of the tuple space, so it cannot be refined to implement application specific policies.

There are many Linda variants in the literature. Multiple tuple spaces are the most relevant to our work [6,13,16,25]. Multiple tuple spaces models permit dynamic creation of new tuple spaces and exchange of tuple spaces as values between processes. While multiple tuple spaces can be used to provide secure communication channels, there is no clear solution to issues such as partial protection, key distribution or memory management.

7 Conclusions

The goal of the work presented in this paper is to exploit the advantages of the Linda programming model in a setting where the components being coordinated can not be fully trusted. Our model allows for security – by controlling access to the objects stored in shared space – and for space management – by allowing objects to be safely removed by a garbage collector. Our solution in the SecOS coordination infrastructure is to use fine grained access control based on locking. An object field that is locked with a key can only be read with the matching key. Keys are also essential in the pattern matching process as they can be used to hide certain objects.

We developed a core language for secure spaces and presented its semantics. Spaces are the sole process communication mechanism in this language, and object locking is enforced by the semantics. We have also implemented the model in Java, and argued that the implementation is efficient. The SecOS implementation is being used in the context of the JavaSeal mobile agent system [4]. Each agent platform on the network possesses a space which is used by agents that arrive at the site to communicate with services and static agents. Our current work is aimed at improving efficiency and at a distributed implementation with cryptographic protection for objects exchanged over the Internet.

References

[1] M. Abadi and A. D. Gordon. A Calculus for Cryptographic Protocols: The Spi Calculus. In *Proceedings of the Fourth ACM Conference on Computer and Communications Security, Zürich, April 1997*, 1997.

[2] R. M. Amadio, I. Castellani, and D. Sangiorgi. On Bisimulations for the Asynchronous π-Calculus. In U. Montanari and V. Sassone, editors, *CONCUR '96*, volume 1119 of *LNCS*, pages 147–162. Springer-Verlag, Berlin, 1996.

[3] G. Boudol. Asynchrony and the π-calculus (Note). Rapport de Recherche 1702, INRIA Sofia-Antipolis, May 1992.

[4] C. Bryce and J. Vitek. The JavaSeal Mobile Agent Kernel. In D. Milojevic, editor, *Proceedings of the 1st International Symposium on Agent Systems and Applications, Third International Symposium on Mobile Agents (ASAMA'99)*, pages 176–189, Palm Springs, May 9–13, 1999. ACM Press.

[5] N. Busi, R. Gorrieri, and G. Zavattaro. A process algebraic view of Linda coordination primitives. *Theoretical Computer Science*, 192(2):167–199, Feb. 1998.

[6] N. Carriero, D. Gelernter, and L. Zuck. Bauhaus Linda. In P. Ciancarini, O. Nierstrasz, and A. Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems*, volume 924 of *LNCS*, pages 66–76. Springer-Verlag, Berlin, 1995.

[7] P. Ciancarini. Agent Coordination in PageSpace. In G. C. Roman and C. Ghezzi, editors, *Proc. ESEC Workshop on Mobility and Network Aware Computing*, Zurich, CH, 1997.

[8] P. Ciancarini and D. Rossi. Jada: Coordination and Communication for Java Agents. In J. Vitek and C. Tschudin, editors, *Mobile Agent Systems: Towards the Programmable Internet*, volume 1222 of *LNCS*, 1997.

[9] L. Dami. Type Inference and Subtyping in Higher-Order Generative Communication. In D. Tsichritzis, editor, *Object Applications*. University of Geneva, 1996.

[10] R. DeNicola and R. Pugliese. A Process Algebra based on Linda. In P. Ciancarini and C. Hankin, editors, *Proc. 1st Int. Conf. on Coordination Models and Languages*, volume 1061 of *Lecture Notes in Computer Science*, pages 160–178. Springer-Verlag, Berlin, 1996.

[11] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces principles, patterns, and practice*. Addison-Wesley, Reading, MA, USA, 1999.

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

[13] D. Gelernter. Multiple Tuple Spaces in Linda. In E. Odijk, M. Rem, and J. Syre, editors, *Proc. Conf. on Parallel Architectures and Languages Europe (PARLE 89)*, volume 365 of *LNCS*. Springer-Verlag, Berlin, 1989.

[14] D. K. Gifford. Cryptographic Sealing for Information Secrecy and Authentication. *Communications of the ACM*, 25(4):274–286, Apr. 1982.

[15] K. Honda and M. Tokoro. On Asynchronous Communication Semantics. In M. Tokoro, O. Nierstrasz, and P. Wegner, editors, *Object-Based Concurrent Computing. LNCS 612*, pages 21–51, 1992.

[16] S. Hupfer. Melinda: Linda with Multiple Tuple Spaces. Technical Report RR YALEU/DCS/R-766, Dept. of Computer Science, Yale University, New Haven, CT, 1990.

[17] R. Jellinghaus. Eiffel Linda: an Object Oriented Linda Dialect. *ACM Sigplan Notices*, 25(12), December 1990.

[18] S. Matsouka and S. Kawai. Using Tuple Space Communication in Distributed Object Oriented Languages. In *Proc. ACM Object Oriented Programming, Systems, Languages and Applications (OOPSLA 88)*, 1988.

[19] N. Minsky, Y. Minsky, and V. Ungureanu. Safe tuplespace-based coordination in multi agent systems. *Journal of Applied Artificial Intelligence*, 15(1), 2001.

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

[21] G. P. Picco, A. L. Murphy, and G.-C. Roman. LIME: Linda Meets Mobility. In D. Garlan, editor, *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, pages 368–377, Los Angeles, CA, USA, May 1999. ACM Press. Also available as Technical Report WUCS-98-21, July 1998, Washington University in St. Louis, MO, USA.

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

[23] A. Polze. The Object Space Approach: Decoupled Communication in C++. In *Proc. Technology of Object-Oriented Languages and Systems (TOOLS 93)*, 1993.

[24] R. Tolksdorf. Alice - Basic Model and Subtyping Agents. Technical Report 1993/7, The Technical University of Berlin, 1993.

[25] R. Tolksdorf. Coordinating Java Agents with Multiple Coordination Languages on the Berlinda Platform. In *IEEE Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, 1997.

[26] R. Tolksdorf. Laura: A Service-Based Coordination Language. *Science of Computer Programming*, 31, 1998.