A Coordination Model for Agents based on Secure Spaces

Bryce, Ciaran; Oriol, Manuel; Vitek, Jan

A Coordination Model for Agents based on Secure Spaces

Ciarán Bryce
Manuel Oriol
Jan Vitek

Abstract

Shared space coordination models such as Linda are ill-suited for structuring applications composed of erroneous or insecure components. This paper presents the *Secure Object Space* model. In this model, a data element can be locked with a key and is only visible to a process that presents a matching key to unlock the element. We give a precise semantics for Secure Object Space operations and discuss an implementation in JAVA for a mobile agent system. An implementation of the semantics that employs encryption is also outlined for use in untrusted environments.

1 Introduction

Coordination languages based on shared data spaces have been around for over fifteen years. Researchers have often advocated their use for structuring distributed and concurrent systems because the mode of communication that they provide, sometimes called *generative communication*, is **associative** and **uncoupled**. Communication is associative in that processes do not explicitly *name* their communication partners, but rather specify the kinds of messages that they are interested in reading. Communication is uncoupled in that no links are established between communicating partners; a message placed in the space may be read at any time and by any process interested in it. These properties make it straightforward to code resource discovery protocols that match up clients with servers based on their respective offers [27], to program in an event-driven style [23] and to dynamically configure running systems [17].

Our goal is to use shared data spaces to coordinate applications made up of potentially **erroneous** and **malicious** components. In particular, we want to use shared data spaces as the main communication model in a mobile agent system called JAVASEAL [29]. *Mobile agents* are programs that carry out distributed computations by communicating locally with resources on a site, and migrating to a remote site to gain access to resources there [24, 12, 30, 5]. A key issue for agent programming is *how to structure and regulate communication between agents*. Our experience with JAVASEAL has shown that a standard message passing model is cumbersome. An agent that arrives in a foreign environment might not know the naming conventions of that environment; it is more convenient for that agent to specify the attributes of the resources that it needs rather than the actual names — hence the utility of associative communication. Further, since agents might migrate in the midst of a protocol, communication must be strongly

*To appear in Proceedings of the 3rd Int. Conference on Coordination Models and Languages (COORDINATION'99), Amsterdam, Netherlands, April 1999, Springer-Verlag, LNCS 1594.

uncoupled. We identified three design goals for agent communication in JAVASEAL which we use for the model presented in this paper:

- **Efficiency**: Frequent communication patterns require low overhead.
- **Controlled Communication**: A message exchanged between agents is subject to a security policy that verifies whether the message is allowed.
- **Privacy**: The secrecy of messages sent between agents must be preserved.

These requirements highlight shortcomings of the shared space model. The model works well when all processes with access to the shared space have been designed carefully and are well-behaved. Unfortunately, mistakes are easily made — the simplest example being reading an entry which is part of another protocol [27, 18] — and nothing prevents malicious processes from spying or even corrupting the data exchanged over the common space. Moreover, denial of service attacks can easily be mounted by flooding the space with requests.

In this paper we present a coordination model, called the Secure Object Space model (SECOS), that meets the above mentioned requirements. In a SECOS, entries in the shared space are locked using *keys*. These keys are used to query the data space and to hide the contents of the data space entries: that is, the contents of the locked object remain hidden until the correct key is presented. In Section 3, we give a precise semantics to SECOS operations in the context of a small programming language derived from the π calculus. Locking is enforced by dynamic typing. In Section 4, we give an overview of the implementation of SECOS in JAVA, and cite an example of its use within the JAVASEAL agent platform. Finally, since it is not possible to rely on typing to protect space entries in untrusted or open environments, we describe a cryptographic implementation of SECOS in Section 5.

2 Shared Spaces

Linda is probably the most widely known shared space coordination model [13]. The model is based on the concept of generative communication; two processes can communicate if one process generates a message tuple, posts it to a Linda tuple space, from where it can be read by the second process. The basic model consists of three primitive operations: out to write a tuple, in to perform a destructive read, and read to perform a non-destructive read.

The Linda input operations are associative since the reader process need only provide a partial description of the value to be retrieved. It can specify some of the tuple fields and leave others blank. Using our syntax, $\mathsf{in}(\langle ?, 3\rangle, \mathsf{x})$ is a Linda operation that retrieves a two element tuple with integer 3 in the second position, and any value in the first position (? represents the wild card). In Linda, a template specifies the number of fields a matching tuple should have as well as the value of each field when a wild card is not used.

While Linda is adequate for coordinating closed parallel systems, coordinating potentially erroneous or malicious components is more challenging. Consider the following two examples:

1. *Secure logical channels*: In a client-server protocol, the reply channel used to send the result of a query to the client must be protected from interference. In a Linda implementation there is no way to guarantee that a tuple containing a server's reply will not be accidentally or willfully read by another process.

2. *Secure garbage collection*: In a long-lived agent environment, migration is frequent and tuple spaces can be littered with "garbage" tuples. These are tuples that are no longer needed, or worse, tuples that have been output as part of a denial of service attack. Thus a garbage collector is needed. Since there can be no clear cut criteria for deciding which tuples are garbage, the garbage collector will rely on heuristics coded at the user-level. Privacy of data being one of the requirements for agent communication, a *secure* garbage collector is defined as a plain process that is trusted to extract tuples from the space according to some policy, but not to access the contents of the extracted tuples. Linda does not distinguish the removal of a tuple from the reading of its fields.

Multiple data spaces have been used to address the access control problem [7, 13, 15, 26] by fulfilling the role of protection domains. In this approach, agents are granted access to a space if they are trusted to manipulate the data in that space and to interact with other agents that have access to the same space. The problem with this approach is that it supposes that each domain is independent, whereas some resources must be visible in several domains. Managing multiple spaces where agents may appear in several spaces is complex. Moreover, this approach does not protect the values stored in the data space, as is required for secure garbage collection. Our model uses multiple data spaces for convenience, but they do not represent an essential part of our proposal.

Another important issue in an agent environment is the treatment of distribution. We have not considered distributed shared spaces on the grounds of efficiency and scalability. Agent systems are meant to be distributed over the Internet. The size and connectivity involved in large scale networks render an efficient implementation of a shared space difficult, if not impossible. From the viewpoint of security, distributed spaces require a trusted implementation (on all nodes) which may be difficult. We therefore prefer to rely on mobile agents for distributed interaction and to keep shared spaces for local, fast and secure communication.

3 Secure Object Spaces

Secure Object Spaces (SECOS) extends Linda in three respects. Firstly, an SECOS system consists of multiple, disjoint, shared object spaces. Secondly, SECOS entries are sequences of *locked* values. Finally, matching is performed on locked entries. We will introduce these features by examples before giving their semantics.

A SECOS entry is called an *object*. It consists of an unordered sequence of *locked fields* – pairs of values and labels. Labels will be referred to as locks or keys. The following is an example of a process that outputs an objects with two fields; we use SECOS syntax.

$$\mathbf{out}(\langle a : \text{"Tom's number is"}\ ,\ b_c : \text{"233 349"}\rangle)@\text{main} \qquad (1)$$

Assuming that there is a SECOS space named main, the term (1) evaluates by depositing the object $\langle a$: "Tom's number is" , b_c : "233 349"$\rangle$ in main. We consider this to be an object composed of values "Tom's number is" and 233 349 locked under the keys a and b_c respectively. Keys serve two purposes in SECOS. First, they filter entries from a SECOS space. When performing an input operation, the template must specify the keys of fields on which associative matching is to be effected. A request for an object from the space main is written

$$\text{in}(\langle a : \text{"Tom's number is"} , \ c_b : \text{"233 349"}\rangle, x)@\text{main} \qquad (2)$$

The effect of evaluating this term is to retrieve an object matching template $\langle a$: "Tom's number is" , c_b : "233 349"$\rangle$, and to bind x to the result. The expression $x.a$ denotes the value "Tom's number is" and $x.c_b$ denotes 233 349.

The second purpose of keys is to provide security. A key for a locked object must be presented in order for a match on that field to succeed. There are two types of keys in SECOS: *symmetric* keys and *asymmetric* keys. A key is symmetric if it both locks and also unlocks an object; in the example above, the key a is symmetric. A key is asymmetric if it belongs to an asymmetric key pair, that is one of the keys of the pair is used to lock a field, and the other key is used to unlock it. The keys (b_c, c_b) form an asymmetric key pair: b_c is used to lock 233 349, and only c_b can unlock it.

Keys modify the notion of matching. Instead of the positional matching of Linda, SECOS relies on key directed matching. We say that a template object matches a target if for every field in the template, there is a matching field in the target. Two fields match if (1) they have been locked with compatible keys, and (2) if their values are equal or if the template's value is "?". Key compatibility is defined as follows. Two symmetric keys are compatible if they are equal (*e.g.*, a is compatible with a); two asymmetric keys are compatible if they belong to the same pair (a_b is compatible with b_a).

The definition of matching we have just given differs from Linda in that matching does not require the number of fields of the template and target to be equal. In SECOS, a shorter object can match a longer object. To illustrate this, the following table lists all of the templates that match the object of line (1).

out	$\langle a$: "Tom's number is" , b_c : "233 349"$\rangle$
in	$\langle a$: "Tom's number is" , c_b : "233 349"$\rangle$,
	$\langle c_b$: "233 349" , a : "Tom's number is"$\rangle$,
	$\langle a$: "Tom's number is" , c_b : ?$\rangle$, $\langle c_b$: "233 349" , a : ?$\rangle$,
	$\langle a$: ? , c_b : "233 349"$\rangle$, $\langle c_b$: ? , a : "Tom's number is"$\rangle$, $\langle a$: ? , c_b : ?$\rangle$,
	$\langle a$: "Tom's number is"$\rangle$, $\langle c_b$: "233 349"$\rangle$, $\langle a$: ?$\rangle$, $\langle c_b$: ?$\rangle$, $\langle\rangle$

An implication of this approach is that an empty template, $\langle\rangle$, matches everything. A simple-minded garbage collector whose goal is to match with any element can be written as follows:

$$!\ \text{in}(\langle\rangle, x)@\text{main} \qquad (3)$$

The repetition operator, !, indicates that the command will be executed in a loop. Line (3) will repeatedly use the empty template to retrieve an arbitrary object from the space main.

Nevertheless, an important property of the model is that *matching of objects happens without the identity of any entry being revealed.* The garbage collector must present a key to select a value locked with the corresponding key. This shows that it is possible to implement a user-level collector that cannot peek in other people's garbage; this was one of the requirements outlined in Section 2.

Another feature of the model is that *objects can be extended without revealing their contents.*

$$\langle a : \text{"22"} \rangle \oplus b : \text{"High"} \qquad \text{yields} \qquad \langle a : \text{"22"}, b : \text{"High"} \rangle$$

An implication of extension on a template, denoted by $\oplus$, is that the template becomes more specific, and can thus match with fewer objects. This technique can be used to implement a security policy on data. For instance, a security policy may designate a process as being low-level or high-level. All in and out requests from a process have the locked field (b:level) appended to the template, where level is "high" or "low". This means that a process can only match with objects of the same security group.

3.1 The SECOS programming language

The remainder of this section gives a semantics for the SECOS model. The SECOS language is a process calculus based on the asynchronous π-calculus [14, 4, 3] which is known to be very expressive, supporting many programming idioms and Turing-complete. To π we add a notion of objects and change the communication rules to use object spaces instead of channels.

Although we are interested in coordinating mobile agents, the language presented here does not allow the exchange of code in object spaces. A treatment of mobility can be found in [30, 6]. The calculi could be merged at the cost of some complexity, but we prefer to leave this as a topic of future investigation. The syntax of the calculus is now given.

Names and Labels We take an infinite set $\mathcal{N}$ of *names* ranged over by meta-variables $a, b, c, \ldots$ (except e, k, t, v). Object spaces and fields are named; names play the role of variables, as in the π-calculus. The infinite set $\mathcal{L}$ of *labels* consists of names (symmetric keys), and pairs of names (asymmetric keys) written a_b. Labels are ranged over by ℓ. We define the *co-label* function $\bar{\cdot}$ as $\overline{a}_b = b_a$ and $\overline{a} = a$.

Objects Basic values are taken from the set of labels extended with a distinguished void element, **?**, and are ranged over by v. The values communicated amongst processes are *objects*, ranged over by t. An object $\langle \ell_1 : v_1, \ldots \ell_i : v_i \rangle$ is a possibly empty sequence of fields where field labels are distinct.

Expressions The syntactic category of *expressions*, ranged over by e, includes basic values, objects, selection expressions and extension expressions.

e	::=	v	basic value
		t	object
		$e.e$	selection
		$e \oplus e : e$	extension

Processes The syntactic category of *processes*, ranged over by P, Q, follows the asynchronous π up to the communication primitives. The empty process $\mathbf{0}$ has no behavior; it is the inert process. Processes can be composed in parallel, $P \mid Q$, and replicated $!\,P$.

P	::=	$\mathbf{0}$	inactivity
		$P \mid Q$	composition
		$!\,P$	replication
		...	

There are two communication primitives: $\mathbf{in}(e, a)@e\,.\,P$ for input and $\mathbf{out}(e)@e$ for output. $\mathbf{in}(e_1, a)@e_2\,.\,P$ tries to extract an object matching the template e_1 from the object space e_2 and binds the result to variable a. The operation is blocking, P cannot execute until the match succeeds. $\mathbf{out}(e_1)@e_2$ outputs the object denoted by e_1 into object space e_2.

P	::=	...	
		$\mathbf{in}(e, a)@e\,.\,P$	input
		$\mathbf{out}(e)@e$	output
		...	

The restriction operator $(\mathbf{new}\ a)$ introduces a fresh name. In our case we also view it as key generation; $(\mathbf{new}\ a)P$ means that a is valid only in P.

P	::=	...	
		$(\mathbf{new}\ a)P$	new name creation

We use alpha conversions of bound names in expression evaluation. The free name function is denoted $fn(\_)$.

3.2 Matching, Keys and Dynamic Typing

We now turn to object types and matching. The notion of matching is somewhat similar to standard notions of subtyping. Asymmetric key pairs introduce an asymmetry that leads to the definition of the type-matching relation ($\trianglelefteq$) which is related to subtyping, but which has significantly different mathematical properties.

Definition 1 (SECOS object type) *An object* $t = \langle \ell_1 : v_1 \ldots, \ell_n : v_n \rangle$ *is of type* $\mathcal{T} = \{\ell_1, \ldots, \ell_n\}$, *written* $t \in \mathcal{T}$. *The co-type of* $\mathcal{T}$ *is* $\overline{\mathcal{T}} = \{\overline{\ell}_1, \ldots, \overline{\ell}_n\}$.

Intuitively, $\mathcal{T}$ type-matches $\mathcal{T}'$ if each field $\overline{\ell}$ of $\mathcal{T}'$ is compatible with a field ℓ of $\mathcal{T}$. Note that $\mathcal{T}'$ may have more fields than $\mathcal{T}$.

Definition 2 (type-matching) *The* match *relation* $\trianglelefteq$ *relates types,* $\mathcal{T} \trianglelefteq \mathcal{T}'$ *iff for all* $\ell \in \mathcal{T}$ *there exists* $\overline{\ell} \in \mathcal{T}'$.

We now define the matching procedure which takes a template object t and a candidate object t' and determines whether t' meets the specification of t.

Definition 3 (matching) *Let* $t \in \mathcal{T}$ *and* $t' \in \mathcal{T}'$, *we say that* t *matches* t', *written* $t \not\approx t'$, *iff (1)* $\mathcal{T} \trianglelefteq \mathcal{T}'$ *and (2) if* $\ell : v \in t$ *and* $\overline{\ell} : v' \in t'$ *implies either* $v = \mathbf{?}$ *or* $v = v'$.

Thus matching can be implemented by a combination of a dynamic type checks and a sequence of field value tests.

3.3 Reduction semantics

The semantic definition of the calculus is a *reduction semantics*, a one-step reduction relation $P \rightarrow P'$, indicating that P reduces in one step of internal computation to P'. We first define two auxiliary notions: *structural congruence* and *evaluation*. Structural congruence, $\equiv$, is the least congruence on processes satisfying the axioms and rules given in Figure 1. The evaluation relation ($\downarrow$) yields 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 Figure 1.

Reduction does not proceed for nonsensical terms. While these terms can be avoided by the introduction of a type system, there is little benefit in the present discussion to add one. Trailing inert processes are elided; thus $\mathbf{in}(t, x)@e \,.\, \mathbf{0}$ becomes $\mathbf{in}(t, x)@e$. The notation for capture avoiding substitution is $P\{^{t}/_{x}\}$. We write $\ell : x \in \langle \ell : x, \ell_1 : x_1, \ldots, \ell_n : x_n \rangle$.

3.4 Examples of SECOS Programming

We now give some programming examples; we assume the presence of additional data types and meaningful space names.

Secure channels

One of the goals of SECOS is to support secure logical channels between processes. Consider an example in which Bob wants to speak with Alice and does not wish anyone else to listen in on their exchange, or even know that the exchange took place. We assume that Bob knows Alice's public key Alice_x. The protocol is fairly standard: Bob generates an object which contains his public key (Bob_y) locked under Alice's public key. When Alice retrieves the object, she may select Bob's key. In this way, Alice can send private messages to Bob and vice versa.

$$\begin{aligned} \mathsf{Bob} &= \mathbf{out}(\langle \mathsf{Alice}_x : \mathsf{Bob}_y, \mathsf{req} : \text{"let's talk"} \rangle)@\mathsf{main} \mid \mathbf{in}(\langle y_{\mathsf{Bob}} : \mathbf{?} \rangle, z)@\mathsf{main} \,.\, P \\ \mathsf{Alice} &= \mathbf{in}(\langle x_{\mathsf{Alice}} : \mathbf{?}, \mathsf{req} : \text{"let's talk"} \rangle, z)@\mathsf{main} \,.\, \mathbf{out}(\langle z.x_{\mathsf{Alice}} : \text{"ok"} \rangle)@\mathsf{main} \end{aligned}$$

When Bob and Alice are composed in parallel, the following reductions occur:

Reduction

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

$$\frac{e_2 \downarrow a \quad e_4 \downarrow a \quad e_1 \downarrow t \quad e_3 \downarrow t' \quad t' \not\lesssim t}{\mathbf{out}(e_1)@e_2 \mid \mathbf{in}(e_3, b)@e_4 \,.\, P \to P\{^t/_b\}}$$

Evaluation

$$\frac{}{v \downarrow v} \qquad \frac{e \downarrow t' \quad t \equiv t'}{e \downarrow t} \qquad \frac{t \equiv \langle e_1 : e_2, \ldots \rangle \quad e_1 \downarrow \ell \quad e_2 \downarrow v \quad e \downarrow \overline{\ell}}{t.e \downarrow v}$$

Structural congruence

$P \mid Q \equiv Q \mid P$
$P \mid \mathbf{0} \equiv P$
$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$
$!P \equiv P \mid !P$
$(\mathbf{new}\, a)(\mathbf{new}\, b)P \equiv (\mathbf{new}\, b)(\mathbf{new}\, a)P$
$a \notin fn(P) \;\Rightarrow\; (\mathbf{new}\, a)(P \mid Q) \equiv P \mid (\mathbf{new}\, a)Q$

$\langle \ell : v, \ell_1 : v_1, \ldots \ell_n : v_n \rangle \equiv \langle \ell_1 : v_1, \ldots \ell_n : v_n, \ell : x \rangle$
$\ell \notin \langle \ell_1 : v_1, \ldots \ell_n : v_n \rangle \;\Rightarrow$
$\langle \ell_1 : v_1, \ldots \ell_n : v_n \rangle \oplus \ell : v \equiv \langle \ell : v, \ell_1 : v_1, \ldots \ell_n : v_n \rangle$

Figure 1: SECOS reduction semantics.

$$\begin{aligned}
&\mathbf{out}(\langle \mathsf{Alice}_x : \mathsf{Bob}_y, \mathsf{req} : \text{"let's talk"} \rangle)@\mathsf{main} \mid \mathbf{in}(\langle y_{\mathsf{Bob}} : ? \rangle, z)@\mathsf{main} \,.\, P \mid \\
&\quad \mathbf{in}(\langle x_{\mathsf{Alice}} : ?, \mathsf{req} : \text{"let's talk"} \rangle, z)@\mathsf{main} \,.\, \mathbf{out}(\langle z.x_{\mathsf{Alice}} : \text{"ok"} \rangle)@\mathsf{main} \\
\\
\equiv\ &\mathbf{out}(\langle \mathsf{Alice}_x : \mathsf{Bob}_y, \mathsf{req} : \text{"let's talk"} \rangle)@\mathsf{main} \mid \\
&\quad \mathbf{in}(\langle x_{\mathsf{Alice}} : ?, \mathsf{req} : \text{"let's talk"} \rangle, z)@\mathsf{main} \,.\, \mathbf{out}(\langle z.x_{\mathsf{Alice}} : \text{"ok"} \rangle)@\mathsf{main} \mid \\
&\quad \mathbf{in}(\langle y_{\mathsf{Bob}} : ? \rangle, z)@\mathsf{main} \,.\, P \\
\to\ &\mathbf{out}(\langle \langle \mathsf{Alice}_x : \mathsf{Bob}_y, \mathsf{req} : \text{"let's talk"} \rangle.x_{\mathsf{Alice}} : \text{"ok"} \rangle)@\mathsf{main} \mid \\
&\quad \mathbf{in}(\langle y_{\mathsf{Bob}} : ? \rangle, z)@\mathsf{main} \,.\, P \\
\to\ &P\{^{\langle \mathsf{Bob}_y : \text{"ok"} \rangle}/_z\}
\end{aligned}$$

Private Store

An agent may store private information in a SECOS without having to be concerned about this data being read by another agent. For instance, an agent that regularly visits a site may leave partial results in a SECOS of the site; the matching semantics prevent the data from being matched, accidentally or on purpose by other agents.

The solution relies on creating a fresh symmetric key to lock an object. The process shown below creates a new name a and deposits an object locked by a into space main.

$$(\mathbf{new}\ a)\big(\mathbf{out}(\langle a : \langle \mathsf{name} : \text{"joe"}, \mathsf{cookie} : 19990212184523\rangle\rangle)@\mathsf{main} \mid P\big) \mid Q$$

Since the name a is restricted, composition with an arbitrary Q is possible, and we know that Q may not match on a. Thus if $P = \mathbf{in}(\langle a : ?\rangle, x)@\mathsf{main}\,.\,P'$ then a reduction can take place:

$$\begin{aligned}&(\mathbf{new}\ a)\big(\mathbf{out}(\langle a : \langle \mathsf{name} : \text{"joe"}, \mathsf{cookie} : 19990212184523\rangle\rangle)@\mathsf{main} \mid \\ &\ \mathbf{in}(\langle a : ?\rangle, x)@\mathsf{main}\,.\,P'\big) \mid Q \\ \rightarrow\ &(\mathbf{new}\ a)\big(P\{^{\langle a\,:\,\langle \mathsf{name}\,:\,\text{"joe"},\mathsf{cookie}\,:\,19990212184523\rangle\rangle}/_{x}\}\big) \mid Q\end{aligned}$$

Another way to obtain a similar result is to use a private SECOS, that is, a space with a restricted name.

Care must be taken when matching with the empty object, $\langle\rangle$, as it may retrieve any object indiscriminately from a shared space.

Interposition

Untrusted agents should not be granted direct access to a shared space. The technique of interposition allows us to filter SECOS requests emitted by an agent without revealing the contents of the objects being manipulated.

Interposition relies on mediating access to the main object space by filter processes that implement an access control policy. We show one implementation of this idea below. A filter is a process that reads requests deposited in a space called from-space and reroutes them to a space called to-space. A client process deposits an object $\langle i : t\rangle$ in from-space to request that the filter perform an input in to-space using t as a template. The result x of the input is then deposited back into from-space as an object $\langle r : x\rangle$. An output can be requested by depositing an object $\langle o : t\rangle$ into from-space. The filter process, parameterized over the two space names, is defined as

$$\begin{aligned}&\mathit{Filter}(\mathsf{from\text{-}space}, \mathsf{to\text{-}space}) = \\ &\quad !\,\mathbf{in}(\langle i : ?\rangle, x)@\mathsf{from\text{-}space}\,.\,\mathbf{in}(x.i, y)@\mathsf{to\text{-}space}\,.\,\mathbf{out}(\langle r : y\rangle)@\mathsf{from\text{-}space} \\ &\quad \mid !\,\mathbf{in}(\langle o : ?\rangle, x)@\mathsf{from\text{-}space}\,.\,\mathbf{out}(x.o)@\mathsf{to\text{-}space}\end{aligned}$$

Thus, the following configuration will perform an output of object $\langle a : 1\rangle$ in space main, and then an input of that object. Space buf is used as the from-space.

$$\begin{aligned}&\mathbf{out}(\langle o : \langle a : 1\rangle\rangle)@\mathsf{buf} \mid \mathbf{out}(\langle i : \langle a : ?\rangle\rangle)@\mathsf{buf} \mid \mathbf{in}(\langle r : ?\rangle, x)@\mathsf{buf}\,.\,P \\ &\mid (\mathbf{new}\ \mathsf{main})(\mathit{Filter}(\mathsf{buf}, \mathsf{main}))\end{aligned}$$

This system will reduce to $P\{^{\langle r\,:\,\langle a\,:\,1\rangle\rangle}/_{x}\} \mid (\mathbf{new}\ \mathsf{main})(\mathit{Filter}(\mathsf{buf}, \mathsf{main}))$.

Garbage collection

We can implement a time-based garbage collector agent (GCA) as a process that mediates access to an object space by interposition. Every output request is extended by the GCA with some GC specific information before inserting the object into the space. For a time-based

garbage collector this information may simply be a time-to-live field. This is coded in the calculus as

$$\mathbf{in}(\langle\rangle, x)@a \,.\, \mathbf{out}(x \oplus \mathsf{TTL} : y)@b$$

assuming that a denotes a space in which insertion requests are placed and b denotes the actual SECOS used for communication. Further, y is some time stamp. The evaluation rule decrees that the extension expression $x \oplus \mathsf{TTL} : y$ returns an object with a TTL field, here TTL is a shared key which we assume to be known only to the GCA.

When a time stamp ceases to be valid, the following process collects the garbage

$$!\,\mathbf{in}(\langle \mathsf{TTL} : y\rangle, x)@b$$

using the rule that only objects stamped at time y are extracted.

This examples reiterates the main properties of SECOS. The GCA is a user level agent which does not gain any knowledge about the content of the objects it manipulates. Secondly, we may have several different GCAs with different policies mediating the same space. Thirdly, GC information is not visible in normal communication; processes may retrieve time stamped objects as before but will not see the GC information.

Linda

It is straightforward to model Linda in SECOS. Linda uses positional notation for matching and does not have any equivalent to our locking primitives. A Linda tuple $\langle x, y, z\rangle$ is represented as an object $\langle \mathbf{1} : x, \mathbf{2} : y, \mathbf{3} : z\rangle$ where **1**, **2** and **3** are globally scoped names that stand for positions in the obvious way. A Linda output operation $\mathbf{out}(\langle x, y\rangle)@a$ is translated in SECOS to $\mathbf{out}(\langle \mathbf{1} : x, \mathbf{2} : y\rangle)@a$. A Linda $\mathbf{in}(\langle ?v, y\rangle, a)@\,.\,P$ is translated to $\mathbf{in}(\langle \mathbf{1} : ?, \mathbf{2} : y\rangle, u)@a \,.\, P'$ where all occurrences of v in P are replaced by $u.\mathbf{1}$.

4 SECOS in JAVA

We implemented SECOS over JAVA for the JAVASEAL agent system [29]. As was the case with the programming language proposal, typing is used to implement tuple locking. Encryption of objects is not necessary for protection so long as the objects remain with the confines of the JAVA virtual machine.

Implementing SECOS within an object-oriented language required careful treatment of two points: inheritance and aliasing. Regarding **inheritance**, SECOS matching requires that the value fields be compared. In an object-oriented language we may either use a default hard-wired comparison operation or allow objects to customize this code by inheritance. The advantage of inheritance is that the meaning of *match* can be adapted to the application. The problem is that extensions introduce security risks. An attacker could program a match method of an object that loops, thereby blocking the thread of the SECOS and subjecting the system to a denial of service attack.

In Jada [9] the match can be extended. In JavaSpaces [23] an object written to the shared space is first transformed to a serialized java.rmi.MarshalledObject; the equals method of MarshalledObject is used for the match, which the user has no possibility of overwriting. In SECOS we also chose a fixed match.

Aliasing occurs when an object can be reached by following different sequences of references. Aliasing is the key to dynamic information sharing, and is pretty much unavoidable as JAVA uses references for parameter passing. The problem with aliasing is that it undermines the role of the space as the sole communication mechanism between communicating agents. Objects placed within an SECOS must not introduce aliases between agents. We use JAVA serialization - the JAVA deep object copy mechanism - in the implementation of the in and out operations to satisfy this constraint.

4.1 Classes

We briefly look at the main classes needed to implement SECOS; helper classes are not described. We model SECOS labels with three classes: an abstract Key class, and two final subclasses AsymKey and SymKey which represent the asymmetric and symmetric keys respectively.

The Key class defines the common attributes of keys. The match method compares two key instances.

The class SymKey represents shared keys. Its match method expects its argument to be this, *i.e.*, the same key is used in actual encryption and decryption.

The class ASymKey represents asymmetric keys. This class contains the dual key that is needed to match with it. Its constructor is protected since an asymmetric key is created indirectly. A user first creates a KeyPair helper class, which creates the two asymmetric keys. In this way, one is sure that each asymmetric key has a dual.

The SecOSObject class implements the object space entries. Recall that the notion of subtyping and type matching defined relied on the notion of structural subtyping. This differs from the JAVA type system since structural subtyping does not require an explicit type hierarchy (such as the hierarchy in JAVA explicitly defined using extends in declarations), and structural subtyping naturally allows for multiple supertypes for any given type. The implementation of matching is thus done dynamically in the match() method which is a final method. For the purpose of matching, null (the "empty" value for pointers) plays the role of ?. A SecOSObject instance is created with a set of key-value pairs. Keys are checked to be either of type SymKey or AsymKey; a user-defined key type is automatically rejected. The select operation has a method that takes a key as parameter and returns the object if the key is the correct key, or null if no matching object is found in the SecOSObject. The extend method returns a new object obtained by extending the target with a key-value pair. The operation fails if the pair is already present in the SecOSObject. The match method returns true if the argument SecOSObject matches. This requires that the keys of both SecOSObjects match, and that the values match.

The SecureObjectSpace class represents the SECOS itself. It provides associative addressing and the methods in, out and read. The latter is a non-destructive version of in (mod-

eled as the atomic execution of an in followed by an out). Both of these are blocking. The out operation is synchronized to guarantee mutual exclusion. The SECOS represents an object in serialized form within the space; thus out serializes the value component of each SecOSObject, in and read deserialize it.

4.2 Coordinating Agents

Our implementation platform for SECOS is the JAVASEAL agent platform [29, 28]. The main characteristic of JAVASEAL is that each agent executes within a *protection domain*, meaning that the agent cannot directly view or modify the data of another agent; communication between agents must use the system provided *channels*. Protection domains are structured in a hierarchy; the root of the hierarchy is the JVM. An agent protection domain can only send a message to its parent domain or to its direct children. Hence, a message sent between two domains must be routed through the common ancestor of the domains.

HyperNews [19] is an agent-based newspaper application that runs over JAVASEAL. Articles are downloaded in the form of agents to client sites; these agents contain code that verify receipt of payment and which only decrypts articles when payment is made. A client site contains an *area* allocated for each newspaper provider; article agents belonging to that provider reside in the provider's area. A basic security requirement is that a provider may not interfere with the articles of another provider. In the JAVASEAL implementation of HyperNews, the root seal of the hierarchy contains the core of the application. A child protection domain is allocated for each provider *proxy*; within each proxy, a child domain is allocated to each article of that provider.

In JAVASEAL, the SECOS is implemented within the root domain of the hierarchy. An in or out message generated by an agent is sent to the root seal via channels. A message can be treated at each level of the hierarchy. For instance, an article can publish its contents by sending an out message to the SECOS. To protect articles from other providers, the root domain which intercepts all messages enforces a security policy. It does this by appending a *password* field to each out and in request. The password varies for each proxy; the extension of each request is transparent to the agent that generates the message and it implements the $\oplus$ operator of Section 3. In JAVASEAL, this is implemented with the following code segment.

```
SymKey AgentKey = new SymKey() ;
String password = new String( "KeepMeSecretTimes" ); // Password for Times
...
public SecOSObject in( SecOSObject request ) { // Request on Times Channel
    SecOSObject extRequest = request.extend( AgentKey , password );
    return sos.in( extRequest );
}
public SecOSObject out( SecOSObject request ) {// Request on Times Channel
    SecOSObject extRequest = request.extend( AgentKey , password );
    return sos.out( extRequest );
}
```

5 A Cryptographic Implementation

It should be obvious that keys and locks in SECOS are analogous to cryptographic keys. The crucial difference is that cryptography is based on the transformation of data from clear-text to cipher-text, and involves heavy-weight mathematical procedures. SECOS locking on the other hand is enforced by dynamic typing, thus making protected communication cheap. However, the protection afforded by types breaks down as soon as values leave the trusted runtime of the SECOS system. This can happen whenever an object is stored on disk, transferred on the network, or used on malicious platforms.

In this section, we consider how to retain the semantics of SECOS in a hostile environment. We begin with a look at some relevant facts about cryptography, and then explain how the locking primitives can be implemented.

5.1 Cryptography

There are two fundamental cryptography schemes. *Symmetric schemes* use shared keys: the same key encrypts and decrypts data from clear-text to cipher-text and back [20]. *Asymmetric schemes*, or public-key cryptography, use key pairs [22]; one key is used to encrypt data, the other to decrypt. The syntax for cryptographic operations is summarized in the following table:

Scheme	Encryption	Decryption
symmetric	$K_a(v)$	$K_a^{-1}(K_a(v))$
asymmetric	$PK_{a_b}(v)$	$PK_{b_a}^{-1}(PK_{a_b}(v))$

In the remainder we make some standard assumptions about cryptography (*c.f.*, [2]): (1) The only way to decrypt a value is to know the corresponding key. (2) An encrypted packet does not reveal the key under which it was encrypted. (3) There is sufficient redundancy in messages so that the decryption algorithm can detect whether the cipher-text was encrypted with a given key.

5.2 Implementing Locking in Open Environments

An implementation of locking in open environments is subject to the following requirements: (R1) Locked values should remain protected while in transit, on secondary media, or while in a third-party shared space. By protection we mean that only a process possessing a matching key may access fields. (R2) Matching should not reveal actual values or keys. (R3) It should not be possible to falsify entries.

In other words, (R1) implies that values should be, and remain, encrypted until fields are selected with the appropriate keys. (R2) implies that matching is performed on encrypted values. (R3) implies that if malicious site sees a request, it should not be possible for it to fabricate an object that matches it. This requirement is the hardest to satisfy.

We outline the proposed solution. Each symmetric lock key ℓ is represented by a pair (a, r) in which a is a symmetric cryptographic key and r is a random number. A compatible key $\overline{\ell}$ is the pair (a, r). A field $\ell : v$ locked under ℓ, is represented as the pair $K_a(v), r$. Each asymmetric lock key ℓ is represented by a triple (a_b, c_d, e) in which a_b and c_d are asymmetric cryptographic keys and e is a symmetric cryptographic key. The matching key $\overline{\ell}$ is (b_a, d_c, e). A field $\ell : v$ locked under asymmetric key ℓ is represented as the triple $(c_d, PK_{c_d}(a_b), PK_{a_b}(K_e(v)))$. Conversely, the value locked by $\overline{\ell}$ is $(d_c, PK_{d_c}(b_a), PK_{b_a}(K_e(v)))$. We require that $K_e(?) = null$ so that the server can recognize a request containing **?**, but $PK_{b_a}(null) \neq null$ so that we can hide the fact that a request contains **?** and thus prevent a replay attack by someone who sees the request and tries to reuse the key to fake further requests.

Fields are only decrypted during field selection, when a matching key is presented, *e.g.*, $o.\overline{\ell}$. Otherwise objects are cryptographically protected.

Matching is performed on ciphertext values, the matching rules differ only on the determination of compatible fields. Here we say that fields are compatible if (1) in the case of symmetric locks: (1a) the encrypted fields are equal, (1b) the template is (null, r) and the target is $(K_a(v), r)$. (2) For asymmetric keys, (2a) if the template is $(c_d, PK_{c_d}(a_b), PK_{a_b}(K_e(v)))$ and the target is $(d_c, PK_{d_c}(b_a), PK_{b_a}(K_e(v)))$ or (2b) if the template is $(c_d, PK_{c_d}(a_b), null)$ and the target is $(d_c, PK_{d_c}(b_a), PK_{b_a}(K_e(v)))$.

Clearly (R1) and (R2) are respected, but (R3) is only partially met. For a symmetric key, any process that sees a request may $K_a(v), r$ may create a false request $null, r$ that matches everything. For asymmetric keys we fare a little better, as seeing $(c_d, PK_{c_d}(a_b), PK_{a_b}(K_e(v)))$ does not suffice for an attacker to guess $(d_c, PK_{d_c}b_a, PK_{b_a}(K_e(v)))$, $(d_c, PK_{d_c}(b_a), PK_{b_a}(null))$ or $(c_d, PK_{c_d}(a_b), PK_{a_b}(null))$. But, seeing two different requests $(c_d, PK_{c_d}(a_b), PK_{a_b}(K_e(v_1)))$ and $(d_c, PK_{d_c}(b_a), PK_{b_a}(K_e(v_2)))$ is sufficient to be able to generate $(d_c, PK_{d_c}(b_a), PK_{b_a}(null))$ which represents an entry with a **?** argument.

The solution means that we can give a cryptographic interpretation SECOS primitive, but their semantics is slightly weaker than in the typed interpretation. Of course, it is costly, since for every match at least one encryption and one decryption can take place. However, optimizations are possible. One does not program an internet application in the same way that one programs a local system. The programmer is likely to impose conventions on his use of remote spaces. For instance, he may decide to only send objects to sites that are trusted. Alternatively, he might only send requests when the destination space is statically known; in this situation he encrypts each request with the public key of the destination host for that request [31]; this makes the need for the second pair of asymmetric keys (c_d, d_c) redundant since one can no longer identify requests containing **?**. Fundamentally though, preventing replay attacks and thus avoiding the need for the extra encryption means employing mechanisms for protecting agent data from malicious hosts; this is still an open research area.

6 Related Work

The work of Gordon and Abadi has inspired some aspects of the theoretical treatment of SECOS. In the spi calculus, they extend the π-calculus to employ cryptography for messages

sent over named channels [2]. Abadi also equates the security of a protocol to its type correctness [1]. While this is clearly applicable to protocols that consists of a predefined set of message exchanges, we do not see a direct relation to dynamic communication environments such as SECOS.

The Klaim language takes another approach to security. There, a process requires access rights (read or write) to use a tuple space [11]. Access rights are represented as types, and a *static* type analysis is sufficient to determine if a process attempts to read or write a tuple space without possessing the right. This approach complements our proposal. While dynamic checking is necessary in a mobile agent context, it is also important to investigate the class of security properties that can be verified in SECOS using a static approach such as Klaim's.

The role of typing 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 [27]. 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 [25]. Dami also investigates type inference for generative communication [10].

As regards implementing the shared object paradigm in JAVA, we have already cited JavaSpaces [23] and Jada [9]. Jada is one example of the shared space paradigm being used to coordinate mobile agents: it is employed in the PageSpace agent architecture. More generally, 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.

Apart from JAVA, the shared space paradigm, usually in its Linda variant, has been integrated into several languages, C, Prolog [8], C++ [21], SmallTalk [18] and Eiffel [16]. Interestingly, the papers that treat object-oriented languages have not considered the security issues posed by inheritance and user defined matching.

7 Conclusions

In this paper, we have considered coordination support for independent and mistrusting software components that cooperate by means of shared object spaces. To this end, we presented a model that integrates the notion of keys into the shared space model. Security is enforced through typing. Further, the model can be implemented in JAVA provided that some care is taken with respect to aliasing and inheritance. Finally, we discussed the implementation of SECOS semantics using encryption for objects that leave a trusted SECOS environment, for instance, when they are transferred over a network or stored on disk.

References

[1] M. Abadi. Secrecy by Typing in Security Protocols. *Theoretical aspects of Computer Software*, September 1998.

[2] 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.

[3] 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.

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

[5] L. Cardelli. Abstractions for mobile computations. Manuscript, Microsoft Research, 1998.

[6] L. Cardelli and A. D. Gordon. Mobile ambients. In *Proc. of Foundations of Software Science and Computation Structures (FoSSaCS), ETAPS'98, LNCS 1378*, Mar. 1998.

[7] 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.

[8] P. Ciancarini. Distribued Programming with Logic Tuple Spaces. Technical Report UBLCS-93-7, The Technical University of Berlin, April 1993.

[9] 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.

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

[11] R. DeNicola, G. Ferrari, and R. Pugliese. Coordinating Mobile Agents via Blackboards and Access Rights. In *Proc. 2nd Int. Conf. on Coordination Models and Languages*, volume 1282 of *LNCS*, September 1997.

[12] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *CONCUR96*, 1996.

[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] 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.

[15] 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.

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

[17] G. Matos and J. Purtilo. Reconfiguration of hierarchical tuple spaces: Experiments with Linda-polylith. Technical report, University of Maryland.

[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] J.-H. Morin and D. Konstantas. Hypernews: A MEDIA application for the commercialization of an electronic newspaper. In *Proceesings of SAC '98 - The 1998 ACM Symposium on Applied Computing*, Marriott Marquis, Atlanta, Georgia, U.S.A, Feb. 27 - Mar. 1 1998.

[20] N. B. of Standards. The Data Encryption Standard. Technical Report Publication 46, Federal Information Processing Standards, January 1977.

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

[22] R. Rivest, A. Shamir, and L. Aldeman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *CACM*, 21(2), 1978.

[23] Sun MicroSystems. JavaSpaces Specification. Technical report, Sun MicroSystems Inc., July 1998.

[24] D. Tennenhouse. Active networks. In USENIX, editor, *2nd Symposium on Operating Systems Design and Implementation (OSDI '96), October 28–31, 1996. Seattle, WA*, 1996.

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

[26] 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.

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

[28] J. Vitek and C. Bryce. Secure Mobile Code: The JavaSeal experiment. In *submitted for publication*, 1999.

[29] J. Vitek, C. Bryce, and W. Binder. Designing JavaSeal: or How to make Java safe for agents. In D. Tsichritzis, editor, *Electronic Commerce Objects*. University of Geneva, 1998.

[30] J. Vitek and G. Castagna. A calculus of secure mobile computations. In *Proceedings of the IEEE Workshop on Internet Programming Languages, (WIPL)*. Chicago, Ill., 1998.

[31] B. S. Yee. A sanctuary for mobile agents. Technical Report CS97-537, UC San Diego, Department of Computer Science and Engineering, Apr. 1997.