Tagged Sets: A Secure and Transparent Coordination Medium

Manuel Oriol and Michael Hicks

University of Maryland, College Park MD 20742, USA,
{oriol,mwh}@cs.umd.edu,
WWW home page: http://www.cs.umd.edu/~{oriol, mwh}

Abstract. A simple and effective way of coordinating distributed, mobile, and parallel applications is to use a virtual shared memory (VSM), such as a Linda tuple-space. In this paper, we propose a new kind of VSM, called a *tagged set*. Each element in the VSM is a value with an associated tag, and values are read or removed from the VSM by matching the tag. Tagged sets exhibit three properties useful for VSMs:

1. *Ease of use.* A tagged value naturally corresponds to the notion that data has certain attributes, expressed by the tag, which can be used for later retrieval.
2. *Flexibility.* Tags are implemented as propositional logic formulae, and selection as logical implication, so the resulting system is quite powerful. Tagged sets naturally support a variety of applications, such as shared data repositories (e.g., for media or e-mail), message passing, and publish/subscribe algorithms; they are powerful enough to encode existing VSMs, such as Linda spaces.
3. *Security.* Our notion of tags naturally corresponds to keys, or capabilities: a user may not select data in the set unless she presents a legal key or keys. Normal tags correspond to symmetric keys, and we introduce *asymmetric tags* that correspond to public and private key pairs. Treating tags as keys permits users to easily specify protection criteria for data at a fine granularity.

This paper motivates our approach, sketches its basic theory, and places it in the context of other data management strategies.

1 Introduction

Computer users require means to store, share, retrieve, and compute data to perform a myriad of tasks. Currently, these means are provided in different ways in different settings, ranging from relational databases to file systems to individual applications. To be useful, any data management approach must answer basic questions concerning organization and security:

1. Is the data organized so that relevant information can be easily found? Is the organizational model easy to use and understand?
2. Is the data protected from malicious tampering? Are the policies for doing so flexible and easy to use?

Relational databases are extremely flexible and optimized for concurrency, fault-tolerance, and throughput. However, they can be difficult to use, particularly in setting up and managing schemas. File systems are easy to understand, and support flexible and intuitive security policies, but have a limited organizational capacity. Linda-style tuple spaces [4], and more generally *virtual shared memories* (VSMs), have a simple and effective organizational strategy, but typical Linda spaces have limited support for security.

In this paper, we propose to manage data as *tagged values* forming part of a *tagged set*. Our approach is inspired by the simplicity and power of the many applications that use tagging as their organizational mechanism, including Google Mail (GMail)[1] and the iLife suite (iTunes, iPhoto, etc.).[2]

A tagged value is merely some data with attached meta-information specified as a tag. Tags are typically used to organize data. For example, iTunes uses the notion of a playlist to organize songs. A playlist is essentially a kind of tag, with each song tagged with the playlist (or playlists) it belongs to, and perhaps the order in which it should played for a given list. A photo album in iPhoto is a similar idea. A file system can also be viewed as tagging system by considering each directory name as a tag; a file "stored" in some directory d is tagged with d. The richer the language for tags, the more organizational traits one can express. In our approach, we encode tags as propositional logic formulae; selecting tagged data from a set is done by logical implication. This approach is powerful enough to easily construct the above examples, as well as to encode more structured repositories, like Linda-style tuple spaces.

In our system, tags serve not only to organize data, but also to protect it from unauthorized access. A tag corresponds naturally to the idea of a key (or capability). A user may not select data in a tagged set unless she presents the keys that protect it. Normal tags correspond to symmetric keys, and we introduce *asymmetric tags* that correspond to public and private key pairs. Treating tags as keys permits users to easily specify protection criteria for data at a fine granularity. Like a file system, individual tagged values can have widely differing access policies. We illustrate this idea by encoding a secure GMail, and extending the Linda-space encoding to incorporate security tags.

Treating tags as keys naturally lends itself to a distributed setting, which is important if tagged sets are to be used as a coordination medium between cooperating applications. By literally using tags as cryptographic keys, we can encrypt data to ensure it can only be read by the appropriate key holder. (As expected, with asymmetric tags we can do this without requiring the host of the tagged set know a user's private tag/key.)

The contributions of this paper are as follows:

– We present a simple formalism for tagged sets (Section 2). The key novelty of our approach is the use of propositional logic as the language of tags, and logical implication as the means to select tagged data.

[1] http://www.gmail.com
[2] http://www.apple.com/ilife/

– We show how tags can be treated as keys in order to protect data at a fine granularity (Section 3). We prove a confidentiality theorem that loosely states that one cannot select data protected by some tag t unless he is in possession of that tag. Tags can be used as cryptographic keys, both symmetric and asymmetric, for secure sharing over a distributed medium.
– We show how tagged sets compare to related approaches in terms of security, flexibility, and performance (Section 4).

2 Tagged Sets

At the most basic level, data stored within a repository can be *tagged* with attributes that describe the data. More formally, a repository is a multi-set of pairs $\langle \tau, v \rangle$, where each pair consists of a *tag* τ and a *value* v. We use the $\{\!| \cdot |\!\}$ notation to clarify our use of multi-sets (which can contain more than one copy of the same element).

As an example, consider an audio repository S containing 3 clips ($clip_1$, $clip_2$, and $clip_3$), each of which is tagged to indicate its genre, drawing from topics Jazz, $\mathsf{Classical}$, and Blues. If each clip falls squarely under one genre, we can tag them as such:

$$S_0 = \{\!|\langle \mathsf{Jazz}, clip_1 \rangle, \langle \mathsf{Classical}, clip_2 \rangle, \langle \mathsf{Blues}, clip_3 \rangle|\!\}$$

Naturally, a clip could be described by more than one genre. For example, if clip $clip_1$ is in genres Jazz and Blues, we could set up the repository as:

$$S = \{\!|\langle \mathsf{Jazz} \vee \mathsf{Blues}, clip_1 \rangle, \langle \mathsf{Classical}, clip_2 \rangle, \langle \mathsf{Blues}, clip_3 \rangle|\!\}$$

To select the clips belonging to a particular genre, we perform a *selection* operation (designated $\downarrow$) on the repository. For example, to select the clips in genre Blues, we would have:

$$S \downarrow \mathsf{Blues} = \{\!|\langle \mathsf{Jazz} \vee \mathsf{Blues}, clip_1 \rangle, \langle \mathsf{Blues}, clip_3 \rangle|\!\} \tag{1}$$

2.1 Selection as Logical Implication

We naturally think of selection as a kind of matching: to select with tag Jazz yields those elements whose tags contain Jazz. However, by considering the selection tag and the data tags as propositions, we can view selection more generally as a kind of implication: selecting tag t in set S yields those elements in S whose tags τ are *implied* by t.

This notion is made precise in Figure 1, which gives the syntax and semantics of *tagged sets*. A tagged set S represents a multi-set of tagged values. Sets are defined by set literals $\{\!|\langle \tau, v \rangle|\!\}$, possibly modified by operators $\cup$, $\downarrow$, $\downarrow_n$, $-$, and $-_n$, discussed below. Values v in these sets are drawn from the countably-infinite set V; their exact makeup is not important for our purposes. Tags are constructed from tag literals t (drawn from the countably-infinite set T), $\top$ (representing "all tags"), and standard operators $\vee$ ("or") and $\wedge$ ("and"). (We do not have $\neg$ or $\bot$

Syntax:

tag literals	$\mathtt{t} \in$	T
values	$v \in$	V
tags	$\tau ::=$	$\mathtt{t} \mid \tau \vee \tau \mid \tau \wedge \tau \mid \top$
tagged set	$S ::=$	$\emptyset \mid \{\!\mid\langle \tau, v\rangle \mid\!\} \mid S \cup S \mid S \downarrow \tau \mid S - \tau \mid S \downarrow_n \tau \mid S -_n \tau$

Semantics:

$\mathcal{D}[\![\cdot]\!] : S \to \mathcal{P}(prop \times V)$

$\mathcal{D}[\![\emptyset]\!]$	$=$	$\emptyset$
$\mathcal{D}[\![\{\!\mid\langle \tau, v\rangle \mid\!\}]\!]$	$=$	$\{\!\mid\langle \tau, v\rangle \mid\!\}$
$\mathcal{D}[\![S_1 \cup S_2]\!]$	$=$	$\mathcal{D}[\![S_1]\!] \cup \mathcal{D}[\![S_2]\!]$
$\mathcal{D}[\![S \downarrow \tau]\!]$	$=$	$\{\!\mid\langle \tau', v\rangle \in \mathcal{D}[\![S]\!] \mid \tau \vdash \tau' \mid\!\}$
$\mathcal{D}[\![S - \tau]\!]$	$=$	$\mathcal{D}[\![S]\!] - \mathcal{D}[\![S \downarrow \tau]\!]$
$\mathcal{D}[\![S \downarrow_n \tau]\!]$	$=$	s iff $s \subseteq \mathcal{D}[\![S \downarrow \tau]\!]$ and $\lvert s\rvert = n$
$\mathcal{D}[\![S -_n \tau]\!]$	$=$	s iff $s \subseteq \mathcal{D}[\![S]\!]$ and $\lvert\mathcal{D}[\![S]\!] - s\rvert = n$

Fig. 1. Syntax and denotational semantics of Tagged Sets

as they would allow selections to violate a useful notion of confidentiality that we introduce in the next section.)

The semantics is given as a *semantic function* $\mathcal{D}[\![\cdot]\!]$ which maps the syntactic notion of tagged set S to a mathematical multi-set containing pairs of propositions and values. As described above, $S \downarrow \tau$ denotes the set whose elements are contained in S, but whose tags are implied by the selection tag τ, following the rules of propositional logic. For example, in (1) above we have

$$\begin{aligned} S \downarrow \mathsf{Blues} = \{\!\mid &\langle \mathsf{Jazz} \vee \mathsf{Blues}, clip_1\rangle, \langle \mathsf{Blues}, clip_3\rangle \mid\!\} \\ \text{since } &\mathsf{Blues} \vdash \mathsf{Jazz} \vee \mathsf{Blues}, \\ &\mathsf{Blues} \nvdash \mathsf{Classical} \text{ and} \\ &\mathsf{Blues} \vdash \mathsf{Blues} \end{aligned}$$

The inference rules for deriving judgments $\tau \vdash \tau'$ are standard; they are presented with one extension in Figure 4 in the next section.

The syntax $S_1 \cup S_2$ denotes the tagged set that results from the combination of tagged sets S_1 and S_2 (using multi-set union). $S - \tau$ denotes those tagged values not implied by the selection tag. Finally, one can limit the results of a selection or subtraction to n elements using the $\downarrow_n$ and $-_n$ operators, respectively. The actual contents of the defined set are non-deterministically chosen.

We can illustrate $\cup$ and $-$ with some additional examples. To define a refinement of S that covers genres Jazz *or* Classical could be done with two selections, and taking the union of the results:

$$(S \downarrow \mathsf{Jazz}) \cup (S \downarrow \mathsf{Classical}) = \{\!\mid\langle \mathsf{Jazz} \vee \mathsf{Blues}, clip_1\rangle, \langle \mathsf{Classical}, clip_2\rangle \mid\!\}$$

To select those documents that cover both genres Jazz *and* Blues, we can do one of two things:

$$\begin{aligned} (S \downarrow \mathsf{Jazz}) \downarrow \mathsf{Blues} &= \{\!\mid\langle \mathsf{Jazz} \vee \mathsf{Blues}, clip_1\rangle \mid\!\} \\ S \downarrow \mathsf{Jazz} \vee \mathsf{Blues} &= \{\!\mid\langle \mathsf{Jazz} \vee \mathsf{Blues}, clip_1\rangle \mid\!\} \end{aligned}$$

To select those documents that cover genre Blues but *not* topic Jazz, we perform a selection followed by a subtraction:

$$(S \downarrow \mathsf{Blues}) - \mathsf{Jazz} = \{\!|\langle \mathsf{Blues}, \mathit{clip}_3\rangle|\!\}$$

2.2 Playlists as Ordered Tuples

So far, we have not considered tags defined with $\wedge$. These are interesting because they effectively *restrict* selections: if a value has tag $\mathsf{t}_1 \wedge \mathsf{t}_2$, then it cannot be selected with either t_1 or t_2 alone: $\mathsf{t}_1 \not\models \mathsf{t}_1 \wedge \mathsf{t}_2$ and $\mathsf{t}_2 \not\models \mathsf{t}_1 \wedge \mathsf{t}_2$.

We can use $\wedge$ tags to extend our audio repository with *playlists*. A playlist is essentially a tuple whose first element designates the first clip to be played, whose second element designates the second clip, etc. Clips can belong to more than one playlist. With tagged sets, we can designate clips clip_1, clip_2, and clip_3 as tracks one, two, and three, respectively, of playlist Favorites by defining repository S_p as follows:

$$S_p = \{\!|\langle \mathsf{Favorites} \wedge 1, \mathit{clip}_1\rangle, \langle \mathsf{Favorites} \wedge 2, \mathit{clip}_2\rangle, \langle \mathsf{Favorites} \wedge 3, \mathit{clip}_3\rangle|\!\}$$

To play the first track of Favorites, we select it with Favorites $\wedge$ 1; to play the second we select with Favorites $\wedge$ 2, and so on:

$$\begin{array}{l} S_p \downarrow \mathsf{Favorites} \wedge 1 = \{\!|\langle \mathsf{Favorites} \wedge 1, \mathit{clip}_1\rangle|\!\} \\ S_p \downarrow \mathsf{Favorites} \wedge 2 = \{\!|\langle \mathsf{Favorites} \wedge 2, \mathit{clip}_2\rangle|\!\} \end{array} \tag{2}$$

To permit selecting *all* songs in a playlist, we can store the clips using a special tag Any:

$$\begin{array}{l} S'_p = \{\!|\ \langle \mathsf{Favorites} \wedge (1 \vee \mathsf{Any}), \mathit{clip}_1\rangle, \langle \mathsf{Favorites} \wedge (2 \vee \mathsf{Any}), \mathit{clip}_2\rangle, \\ \qquad \langle \mathsf{Favorites} \wedge (3 \vee \mathsf{Any}), \mathit{clip}_3\rangle|\!\} \end{array}$$

To select all of the songs in playlist Favorites, we simply do $S'_p \downarrow \mathsf{Favorites} \wedge \mathsf{Any}$. Of course, we can continue to organize songs by genre as well as by playlist:

$$\begin{array}{l} S'_p = \{\!|\ \langle (\mathsf{Favorites} \wedge (1 \vee \mathsf{Any})) \vee (\mathsf{Jazz} \vee \mathsf{Blues}), \mathit{clip}_1\rangle, \\ \qquad \langle (\mathsf{Favorites} \wedge (2 \vee \mathsf{Any})) \vee \mathsf{Classical}, \mathit{clip}_2\rangle, \\ \qquad \langle (\mathsf{Favorites} \wedge (3 \vee \mathsf{Any})) \vee \mathsf{Blues}, \mathit{clip}_3\rangle|\!\} \end{array}$$

2.3 Tuple Sets with Linda-style Matching

We can formalize this basic encoding of tuples to include matching as in Linda-style tuple spaces [4]. Consider the syntax of a simple language of *tuple sets* shown in Figure 2. The basic operations on tuple sets T are similar to those on tagged sets, but rather than performing selections based on a tag, the user provides a *pattern* p. This pattern consists of either a value v or a wildcard ? which matches any value. Subtraction with $T - p$ is as with tagged sets: it defines the set T' with all elements in T removed that match p.

Syntax:

tuple	$u ::= (v) \mid (v, v) \mid (v, v, v) \mid \ldots$
pattern var	$a ::= v \mid \ ?$
pattern	$p ::= (a) \mid (a, a) \mid (a, a, a) \mid \ldots$
tuple set	$T ::= \emptyset \mid \{\!\|u\|\!\} \mid T \cup T \mid T \downarrow p \mid T - p \mid T \downarrow_n p \mid T -_n p$

Semantics:

$$
\begin{array}{lcl}
\mathcal{L}[\![\emptyset]\!] & = & \emptyset \\
\mathcal{L}[\![\{\!|(v_1, \ldots, v_n)|\!\}]\!] & = & \{\!|\langle \underline{n} \wedge \bigvee_{1 \le i \le n} (i \wedge v_i), (v_1, ..., v_n)\rangle|\!\} \\
\mathcal{L}[\![T_1 \cup T_2]\!] & = & \mathcal{L}[\![T_1]\!] \cup \mathcal{L}[\![T_2]\!] \\
\mathcal{L}[\![T \downarrow p]\!] & = & \mathcal{L}[\![T]\!] \downarrow \mathcal{T}[\![p]\!] \\
\mathcal{L}[\![T - p]\!] & = & \mathcal{L}[\![T]\!] - \mathcal{T}[\![p]\!] \\
\mathcal{L}[\![T \downarrow_n p]\!] & = & \mathcal{L}[\![T]\!] \downarrow_n \mathcal{T}[\![p]\!] \\
\mathcal{L}[\![T -_n p]\!] & = & \mathcal{L}[\![T]\!] -_n \mathcal{T}[\![p]\!] \\
\\
\mathcal{T}[\![(a_1, \ldots, a_n)]\!] & = & \underline{n} \wedge \bigvee_{1 \le i \le n} (i \wedge a_i) \text{ where } a_i \neq \ ?
\end{array}
$$

Fig. 2. Syntax of Linda-style tuples, semantics via Tagged Sets

As an example, say we have the following tuple set which mentions the birthdays of Alice and Bob:

$$T = \{\!|(\text{“}birthday\text{”}, \text{“}alice\text{”}, 10, 29, 1991), (\text{“}birthday\text{”}, \text{“}bob\text{”}, 10, 4, 1993)|\!\}$$

If we wanted to select all birthday records, we could do:

$$T \downarrow (\text{“}birthday\text{”}, ?, ?, ?, ?) = \{\!| \ (\text{“}birthday\text{”}, \text{“}alice\text{”}, 10, 29, 1991), \ (\text{“}birthday\text{”}, \text{“}bob\text{”}, 10, 4, 1993)|\!\} \tag{3}$$

If we wanted only Alice's birthday, we could do

$$T \downarrow (\text{“}birthday\text{”}, \text{“}alice\text{”}, ?, ?, ?) = \{\!|(\text{“}birthday\text{”}, \text{“}alice\text{”}, 10, 29, 1991)|\!\} \tag{4}$$

Conversely, we could define the set with an arbitrary birthday element removed:

$$\begin{array}{rcl} T -_1 (\text{“}birthday\text{”}, ?, ?, ?, ?) & = & \{\!|(\text{“}birthday\text{”}, \text{“}bob\text{”}, 10, 4, 1993)|\!\} \\ & \text{or} & \{\!|(\text{“}birthday\text{”}, \text{“}alice\text{”}, 10, 29, 1991)|\!\} \end{array} \tag{5}$$

The selections in Examples (3) and (4) are similar to what is possible with the `read(-)` operator for Linda-spaces. Example (5) is like the Linda-space `in(-)`, which removes a single tuple from the space (as our language is declarative, we actually define a new tuple set which lacks an element present in the original). Section 3.1 presents communication commands which when combined with these operators can be used to build traditional Linda spaces.

Tuple sets and their operations can be encoded using tagged sets. Shown in Figure 2, the translation function $\mathcal{L}[\![\cdot]\!]$ maps tuple sets T to tagged sets S, employing auxiliary function $\mathcal{T}[\![\cdot]\!]$ to map patterns p to tags τ. In the tagged set, the tuple is stored as the value part of the tagged value (i.e., it is in V),

and the tag encodes its structure. The first part of the tag is the tuple length $\underline{n}$; to select a tuple of length $\underline{n}$ one must provide this length as a tag. The second part is a union of tags, one for each element in the tuple. As with playlists, these tags encode the position of the element i as tag i. In addition, we include the element v itself as tag, so that we can match literal values present in patterns. The resulting element tag is thus $i \wedge v$. Selection patterns are encoded similarly, except that when a ? appears in a pattern, it does not appear in the tag. This way, it has no bearing on the selection (thus encoding its meaning as a "wild card"). Note that we ignore the possible collision between the tags of indices (i), the tag for indicating size ($\underline{n}$), and integer values used as tags. Addressing this problem would be straightforward.

Here are some examples that illustrate the translation:

Tuple sets	Tagged sets
$\mathcal{L}[\![\{\!\|(v)\|\!\}]\!]$	$= \{\!\|\langle \underline{1} \wedge (1 \wedge v), v\rangle\|\!\}$
$\mathcal{L}[\![\{\!\|(v_1, v_2)\|\!\}]\!]$	$= \{\!\|\langle \underline{2} \wedge ((1 \wedge v_1) \vee (2 \wedge v_2)), (v_1, v_2)\rangle\|\!\}$
$\mathcal{L}[\![T - (v_1, v_2)]\!]$	$= \mathcal{L}[\![T]\!] - \underline{2} \wedge ((1 \wedge v_1) \vee (2 \wedge v_2))$
$\mathcal{L}[\![T \downarrow (?, v_2)]\!]$	$= \mathcal{L}[\![T]\!] \downarrow \underline{2} \wedge (2 \wedge v_2)$

Tagged sets are not rich enough to encode SQL-style or publish/subscribe service queries, mainly because tags can only be used to match set elements; it is not possible to, for example, treat a tag as an integer and then return all tagged values whose tag is "greater than 1." We are compare our approach more closely to these systems in Section 4.

3 Secure Tags

If a tagged set is to be used in a secure, multi-user setting, it should allow users to only reveal their data to others whom they trust. For example, a typical file system labels a file with an access control list, specifying an effective list of users and the operations they can perform on the file (e.g., read, write, delete).

A useful feature of tagged sets is that "user lists," or more properly operation system-style *capabilities*, can be encoded using tags. That is, a tag can be viewed as a key, which means that to select a value, one must produce the key with which it is locked. A value can be locked multiple times (using $\wedge$) requiring the selector produce multiple keys to unlock it. A value may have alternate access points (using $\vee$) which permits unlocking with different key sets. This is similar to Gifford's sealed objects [6].

In the remainder of this section, we show how tags can form the foundation of secure access control of shared data. We present a simple interface for shared tagged sets that ensures confidentiality. Then we show how *asymmetric tags* can be used to support public key-style encryption in tagged sets. Finally, we consider how to provide secure, distributed access to a shared tagged set.

3.1 A Shared Repository

Imagine we wish to define tagged sets that may be shared by processes within an operating system. We extend our presentation so far in two ways. First, we need a way to name a shared tagged set. Second, we must specify a list of commands for manipulating named tagged sets that respects the confidentiality policies of data stored in them; these policies are specified by tags. The syntax and semantics of these changes is shown in Figure 3.

Shared tagged sets STS are tagged sets S extended to include variable names x, whose semantics is simply to look up the tagged set named by x from a global store. We designate this in the semantics $\mathcal{D}[\![x]\!]$ as the function $\mathit{lookup}(x)$; elsewhere we use the function $\mathit{update}(x, s)$ to designate updating the name x in the global store to refer to multi-set s.

Commands cmd can be used by processes to manipulate shared tagged sets: $\mathrm{read}_\tau(STS)$ reads (selects) data from a tagged set STS; $\mathrm{add}_\tau(x, STS)$ adds the set STS to the named tagged set x (using multi-set union $\cup$), and $\mathrm{remove}_\tau(x, STS)$ removes data in set STS from the named tagged set x (using multi-set subtraction). Both add and remove operate by side-effect only, "returning" the empty set $\emptyset$. We assume the implementation of commands is atomic. Linda tuple spaces are essentially the combination of these commands with the tuple encoding presented in Section 2.3.

$$
\begin{array}{lll}
STS & ::= & x \mid \emptyset \mid \{\!|\langle \tau, v\rangle|\!\} \mid STS \cup STS \mid STS \downarrow \tau \mid STS - \tau \\
 & & \mid \; STS \downarrow_n \tau \mid STS -_n \tau \\
\mathit{cmd} & ::= & \mathrm{read}_\tau(STS) \mid \mathrm{add}_\tau(x, STS) \mid \mathrm{remove}_\tau(x, STS)
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{D}[\![\cdot]\!] : STS \to \mathcal{P}(\mathit{prop} \times V) & & \\
\mathcal{D}[\![x]\!] & = & \mathit{lookup}(x) \\
\mathcal{D}[\![STS]\!] & = & \text{as in Figure 1 otherwise}
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{C}[\![\cdot]\!] : \mathit{cmd} \to \mathcal{P}(\mathit{prop} \times V) & & \\
\mathcal{C}[\![\mathrm{read}_\tau(STS)]\!] & = & \mathcal{D}[\![STS \downarrow \tau]\!] \\
\mathcal{C}[\![\mathrm{add}_\tau(x, STS)]\!] & = & \text{let } s = \mathcal{D}[\![STS \downarrow \tau]\!] \text{ in} \\
 & & \quad \text{let } \_ = \mathit{update}(x, \mathit{lookup}(x) \cup s) \text{ in } \emptyset \\
\mathcal{C}[\![\mathrm{remove}_\tau(x, STS)]\!] & = & \text{let } s = \mathcal{D}[\![STS \downarrow \tau]\!] \text{ in} \\
 & & \quad \text{let } \_ = \mathit{update}(x, \mathit{lookup}(x) - s) \text{ in } \emptyset
\end{array}
$$

Fig. 3. Commands for manipulating shared Tagged Sets

The security of these commands is based on implication: to read, add, or remove data having tag $\mathtt{t}$ from a shared set, the user must hold a "credential" τ that implies $\mathtt{t}$; the credential τ is presented as a subscript on each operation.[3] Without requiring a credential, a user could use subtraction to extract elements from a shared set for which she did not hold the tags. For example, say that x

[3] While τ can be any tag, it is most useful as a *capability list*, having the form $\mathtt{t_1} \wedge \ldots \wedge \mathtt{t_n}$.

is bound to $\{\!|\langle \mathsf{Jazz} \vee \mathsf{Blues}, clip_1\rangle, \langle \mathsf{Classical}, clip_2\rangle, \langle \mathsf{Blues}, clip_3\rangle|\!\}$, and the user knows about the tag Classical. The user should thus only be allowed to read the tagged value $\langle \mathsf{Classical}, clip_2\rangle$, since she does not know the names of the tags on the other values. Indeed, $\mathrm{read}_{\mathsf{Classical}}(x - \mathsf{Classical})$ is $\emptyset$, whereas not requiring a credential and just permitting the subtraction would have yielded $\{\!|\langle \mathsf{Jazz} \vee \mathsf{Blues}, clip_1\rangle, \langle \mathsf{Blues}, clip_3\rangle|\!\}$, violating confidentiality. Tag implication has been defined so that only when a tag t appears in credential τ can t be selected. Thus a user must "know about" a tag, and place it in her credential, to be able to unlock values locked with that tag. We formally state and prove this notion of confidentiality in the next subsection.

In this scheme, if the tag t is implied by the credential τ, then any data $\langle \mathsf{t}, v\rangle$ can be added, removed, or read from the tagged set. We could encode richer access rights for better control over these operations. For example, to provide separate read and removal permissions, we could define special tags Read and Remove; a value's tag would be $(\mathsf{Read} \wedge \tau_{read}) \vee (\mathsf{Remove} \wedge \tau_{remove})$. The semantics of $\mathrm{read}_\tau(STS)$ would become $\mathcal{D}[\![STS \downarrow \tau \wedge \mathsf{Read}]\!]$, so as to verify against the τ_{read} part of the tag, and removal would be similar. We would also have to ensure that these special tags neither appear in credentials τ nor clash with normal tags.

3.2 Asymmetric Tags

Tags defined so far essentially correspond to *symmetric keys*: if the user can produce the key, he can acquire the value. We can also easily extend our notion of tag to model *asymmetric keys*, as are provided in public key cryptography.

Asymmetric tags are defined in pairs $(\mathsf{k}, \overline{\mathsf{k}})$. As shown in the top right of Figure 4, asymmetric tags are drawn from the countably-infinite set *Keys* and extend our notion of tags τ. Tagging a value using an asymmetric tag k is equivalent to locking it with an asymmetric key. To select this value requires producing the opposite tag $\overline{\mathsf{k}}$. We do not consider how asymmetric tags are generated; we only assume that if a tagged set contains data locked by some tag k, then it has some way of knowing when the complement tag $\overline{\mathsf{k}}$ is provided during selection. We consider how to do this securely in a distributed setting in the next subsection. By convention, we say $\overline{\mathsf{k}}$ is the private tag and k is the public tag.

The left side of Figure 4 shows the (slightly modified) inference rules for the fragment of propositional logic that we use in our tagging system. As usual, Γ is simply an ordered list of assumptions $\tau_1, \ldots, \tau_n$. The rules employ an additional operator $[\cdot]$ that is the identity on symmetric tags, but the complement for asymmetric ones. The operator is used in the assumptions of the ($\vee$ELIM) and (HYP) rules to enforce that k can only be implied by its complement $\overline{\mathsf{k}}$ and vice versa. This relationship is not transitive: having proven k, there is no way to include it in the assumptions to prove $\overline{\mathsf{k}}$. If there were, it would allow the holder of a public tag k to access data tagged with that tag, rather than only allowing the holder of $\overline{\mathsf{k}}$ to access it.

$$\frac{\Gamma \vdash \tau_1 \qquad \Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 \wedge \tau_2}\ \wedge\text{INTRO} \qquad \frac{\Gamma \vdash \tau_1 \wedge \tau_2}{\Gamma \vdash \tau_1}\ \wedge\text{ELIM}_1 \qquad \frac{\Gamma \vdash \tau_1 \wedge \tau_2}{\Gamma \vdash \tau_2}\ \wedge\text{ELIM}_2$$

$$\frac{\Gamma \vdash \tau_1 \vee \tau_2 \qquad \Gamma, [\tau_1] \vdash \tau_3 \qquad \Gamma, [\tau_2] \vdash \tau_3}{\Gamma \vdash \tau_3}\ \vee\text{ELIM} \qquad \frac{\Gamma \vdash \tau_1}{\Gamma \vdash \tau_1 \vee \tau_2}\ \vee\text{INTRO}_1 \qquad \frac{\Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 \vee \tau_2}\ \vee\text{INTRO}_2$$

$$\frac{}{\Gamma \vdash \top}\ \top\text{ELIM} \qquad \frac{}{\Gamma, [\tau], \Gamma' \vdash \tau}\ \text{HYP}$$

$$\begin{array}{lll} \mathsf{k}, \overline{\mathsf{k}} & \in & \mathit{Keys} \\ \tau & ::= & \ldots \mid \mathsf{k} \mid \overline{\mathsf{k}} \end{array}$$

$$\begin{array}{lcl} [\mathsf{t}] & = & \mathsf{t} \\ [\mathsf{k}] & = & \overline{\mathsf{k}} \\ [\overline{\mathsf{k}}] & = & \mathsf{k} \\ [\tau_1 \vee \tau_2] & = & [\tau_1] \vee [\tau_2] \\ [\tau_1 \wedge \tau_2] & = & [\tau_1] \wedge [\tau_2] \\ [\top] & = & \top \end{array}$$

Fig. 4. Proof system extended with asymmetric tags

We can now make our notion of confidentiality precise. We wish to ensure that *cmd* operations do not provide or revoke access to data tagged with keys not contained in the credential τ. This is ensured by the following theorem:

Theorem 1 (Confidentiality). *If* $\tau \vdash \mathsf{t}$, *then* $\mathsf{t} \in [\tau]$.

In a logical sense, this theorem simply states that to prove t, we have to know about t in the first place; intuitively it "appears" in our assumption τ. In the simplest case, if τ has the form $\mathsf{t}_1 \wedge \ldots \wedge \mathsf{t}_n$, then exactly $\mathsf{t}_1, \ldots, \mathsf{t}_n$ appear in τ; only these tags can be proved by τ. Since all operations on shared tagged sets must ultimately be filtered against the credential τ, we ensure that only data whose tags appear in the credential can be manipulated. This theorem is proven by structural induction on derivations $\tau \vdash \mathsf{t}$, aided by some simple lemmas on the $\in$ relation. A full definition of "appears in" ($\in$) is presented in Figure 5.

$$\begin{array}{llll} \mathsf{t} \in \mathsf{t} & \mathsf{t} \in \tau_1 \vee \tau_2 & \text{iff } \mathsf{t} \in \tau_1 \text{ and } \mathsf{t} \in \tau_2 \\ \mathsf{k} \in \mathsf{k} & \mathsf{t} \in \tau_1 \wedge \tau_2 & \text{iff } \mathsf{t} \in \tau_1 \text{ or } \mathsf{t} \in \tau_2 \\ \overline{\mathsf{k}} \in \overline{\mathsf{k}} & \tau_1 \vee \tau_2 \in t & \text{iff } \tau_1 \in t \text{ or } \tau_2 \in t \\ \top \in t & \tau_1 \wedge \tau_2 \in t & \text{iff } \tau_1 \in t \text{ and } \tau_2 \in t \end{array}$$

Fig. 5. Definition of $t \in t'$

A Secure Mail System As a simple example of the use of asymmetric tags, consider a secure e-mail system that supports tagging for categorization, as in GMail. When a mail is received for a particular user, it is tagged with her public key. For example, if Daniel sends a private message to Bridget, it is locked with her public key $\mathsf{Bridget}$:

$$S = \{|..., \langle \mathsf{Bridget}, \text{"}You\ appear\ to\ have\ forgotten\ ...\text{"} \rangle, ...|\}$$

To retrieve her messages, Bridget selects using her private key:

$$S \downarrow \overline{\mathsf{Bridget}} = \{\!|\langle \mathsf{Bridget}, "You\ appear\ to\ have\ forgotten\ ..."\rangle|\!\}$$

A message addressed to many users is simply tagged with their public keys:

$$S = \{\!|..., \langle \mathsf{Bridget} \vee \mathsf{Mark}, "Come\ home,\ I\ have\ jumpers\ for\ you\ two."\rangle, ...|\!\}$$

Either $\overline{\mathsf{Mark}}$ or $\overline{\mathsf{Bridget}}$ can be used to select the message.

Users can use the normal tagging system to organize their mail. For example, Bridget could annotate the message from Daniel as a love letter:

$$S' = \{\!|..., \langle \mathsf{Bridget} \wedge \mathsf{Loveletter}, "You\ appear\ to\ have\ forgotten..."\rangle, ...|\!\}$$

Multiple categories are of course possible, by annotating with tags of the form (e.g.) $\mathsf{Bridget} \wedge (\mathsf{t}_1 \vee \mathsf{t}_2 \vee ... \vee \mathsf{t}_n)$.

Encoding encrypted tuples. With asymmetric tags, we can support a simple extension to our tuple language: tagging each tuple with a public key. We can modify the syntax of tuples u and patterns p in Figure 2 to be as follows:

$$\begin{array}{ll} \text{tuple} & u ::= \mathsf{k} : (v_1, v_2, \ldots, v_n) \\ \text{pattern} & p ::= \overline{\mathsf{k}} : (a_1, a_2, \ldots, a_n) \end{array}$$

Tuples now include the public key, and patterns must now specify the private key to select the desired tuple. We modify our semantic functions to take these changes into account:

$$\begin{array}{l} \mathcal{L}[\![\{\!|(\mathsf{k} : (v_1, v_2, \ldots, v_n))|\!\}]\!] = \{\!|\langle \mathsf{k} \wedge (\bigvee_{1 \le i \le n}(v_i)), (v_1, ..., v_n)\rangle|\!\} \\ \mathcal{T}[\![\overline{\mathsf{k}} : (a_1, a_2, \ldots, a_n)]\!] = \quad (\overline{\mathsf{k}} \wedge (\bigvee_{1 \le i \le n, a_i \neq ?} a_i)) \end{array}$$

While this is a simple extension of the tuple space encoding, we can encode more complex systems, such as SecOS [15] or CryptoKlava [2].

3.3 Distributed Tagged Sets

To use shared tagged sets as a distributed coordination mechanism, we have to protect the repository and the data it contains from malicious tampering. We have two goals. First, tags/keys and the data they protect should not be transmitted in clear text, to prevent snooping. Second, users should not have to present their private tag/key to the database to retrieve data stored with the public key. Here we briefly sketch how these goals might be achieved.

A client can communicate with a server hosting a shared tagged set using a secure channel, in the style of the Secure Socket Layer (SSL)[4]. The client begins by presenting the server with a credential τ of the form $\mathsf{t_1} \wedge \ldots \wedge \mathsf{t_n}$, but rather than sending the actual tags $\mathsf{t_i}$, the client sends a *public name* for each tag. The

[4] http://wp.netscape.com/eng/ssl3/

mapping between the public name and the key must be known by both the client and server; for example, the tag Red could have the name "Red." This credential is sent along with a random integer n_u, both encrypted with the server's public key (to ensure that commands are not redirected to the wrong server). The server maps the names to tags, and then challenges the user to prove he actually holds tags $\mathsf{t_1} \ldots \mathsf{t_n}$ by sending a message containing integers encrypted with each tag, along with a nonce (to prevent replay attacks). The user decrypts these values, and from them derives the value n_s. The user sends the decrypted values back to the server, which verifies they are correct.

At this point, both sides have a shared secret (n_u,n_s), and the server is satisfied that the user holds the keys in the claimed credential. Users send commands encrypted with the shared secret, and the server evaluates the commands using the verified credential τ.

Because we are treating tags as keys, the server should be careful about the tags it returns with values selected by a read command. For example, if a shared tagged set contains the value $\langle \mathsf{Bridget} \vee \mathsf{Mark}, \text{"}You\ appear\ to\ have\ forgotten\ ...\text{"} \rangle$, then Bridget can read this value using her public key $\overline{\mathsf{Bridget}}$. However, if the server includes the tag $\mathsf{Bridget} \vee \mathsf{Mark}$ in the result, then Bridget can see Mark's private key! The simplest solution is to simply strip the tags at the server, returning only a set of values (encypted by the shared secret), or else to use the public names of the tags, rather than the tags themselves.

We can avoid the need for a secure channel for read commands by having the server encrypt returned values using their tags. For each tagged value $\langle \tau, v \rangle$ in the returned set, we return the value as the tuple $(clear(\tau), E[\langle \tau, v \rangle])$, where $clear(\tau)$ is a translation of τ with the keys replaced by their names, and where $E[.]$ performs encryption as defined below (similar to Gifford's schema [6]):

$$\begin{array}{ll} E[\langle \top, v \rangle] & \rightarrow v \\ E[\langle \mathsf{t}, v \rangle] & \rightarrow \{v\}_{\mathsf{t}} \\ E[\langle \mathsf{k}, v \rangle] & \rightarrow \{v\}_{\mathsf{k}} \\ E[\langle \tau_1 \wedge \tau_2, v \rangle] & \rightarrow (E[\langle \tau_1, E[\langle \tau_2, v \rangle] \rangle]) \\ E[\langle \tau_1 \vee \tau_2, v \rangle] & \rightarrow (E[\langle \tau_1, v \rangle], E[\langle \tau_2, v \rangle]) \end{array}$$

The notation $\{v\}_{\mathsf{t}}$ denotes encrypting v with key t. The user then maps backwards from the provided tag names to the actual tags to decrypt the returned values.

4 Discussion

Tagged sets aim to organize and share data simply and securely. They bear resemblance to coordination spaces [4], relational databases [13, 7], semi-structured documents [3], and publish/subscribe services [5, 14, 1], but differ chiefly in how data is organized, selected, and secured.

Data Organization and Selection. Coordination spaces (e.g., Linda-spaces) organize data as tuples, which are essentially a fixed-length, ordered list of simple val-

Table 1. Comparison of data selection and locking mechanisms

System	k	Data d ::=	Queries q ::=
Linda	v	$k \mid d \vee d$	$k \mid q \vee q$
Siena/Gryphon	t *op* v	$k \mid d \vee d$	$k \mid q \vee q$
Elvin	t *op* v	$k \mid d \vee d$	$k \mid q \vee q \mid q \wedge q \mid \neg q$
Tagged Sets	t, k	$k \mid d \vee d \mid d \wedge d$	$k \mid q \vee q \mid q \wedge q$
Sealed Objects	k	$k \mid d \vee d \mid d \wedge d$	$k \mid q \wedge q$

ues. Relational databases and publish/subscribe (hereafter *pub/sub*) systems organize data as (unordered) labeled records; an example might be (*what*=`alarm`) $\vee$ (*level*=5). For databases, records are stored in tables, while for pub/sub they are periodically published as *events*. Compared to these, tagged sets are relatively *semi-structured*: while events and records typically adhere to a schema, we place no restrictions on the form of tags.

As with tagged sets, these systems permit certain records/events/tuples to be selected based on user queries. Linda-based queries were described in subsection 2.3. For pub/sub systems, queries are in the form of *subscriptions*, defined as patterns meant to match subsequently published events. Subscriptions in Siena [5] and Gryphon [1] mimic the structure of events where = can be replaced by a more general operator. For example, the subscription (*what*=`alarm`) $\vee$ (*level* < 7) matches the event presented earlier. Elvin [14] extends these sorts of queries with other logical operators, approximating SQL-style queries used in databases. Compared with tagged sets, coordination spaces are strictly less powerful. Pub/sub systems have both more and less powerful selections: they are more powerful in that they can select based on *data*, whereas tagged sets only select on tags, but Siena and Gryphon are less powerful in that subscriptions can only be specified as a list of attributes, like a record, whereas tagged sets permit more general logical formulae.[5] Elvin and typical databases allow subscriptions/queries to include the $\neg$ (not) operator, which we have avoided for security reasons, but can model using subtraction.

These results are summarized in Table 1 (along with a row for Sealed Objects, described below). Selectable data d and queries over that data q are defined as a combination of elements k. The systems are ordered by expressiveness: Linda-spaces are the most simple, and Elvin (also representing databases) is the most expressive. We have made notation uniform so that in all cases selection is defined by logical entailment: query q selects data d iff $q \vdash d$ (adjusted to take data and relational operators into account for pub/sub systems).[6]

[5] To select on data in a tagged set, we could duplicate the data as part of the tag.

[6] This may be unintuitive at first: a query that says "select those records having both the t1 and t2 tags" would be expressed as $\mathsf{t1} \vee \mathsf{t2}$ since $\mathsf{t1} \vee \mathsf{t2} \vdash \mathsf{t1} \vee \mathsf{t2}$ but $\mathsf{t1} \vee \mathsf{t2} \nvdash \mathsf{t1}$; conversely, if we expressed the query as $\mathsf{t1} \wedge \mathsf{t2}$, then we would have the undesirable result $\mathsf{t1} \wedge \mathsf{t2} \vdash \mathsf{t1}$. See Section 2.1.

Generally speaking, the more expressive the system, the more expensive the selection algorithm. For example, given a record and a subscription each with m attributes, selection in Siena takes time $O(m)$ (assuming set membership can be implemented in $O(1)$ time); this approach can be further optimized [1, 14]. By contrast, selection in Linda-spaces is cheaper, and selection in Elvin is more expensive. The performance of tagged sets in part depends on how data is encoded. For example, a simple approach (implemented in our current Java prototype) is to store each tagged value in a collection, with each tag τ stored in Conjunctive Normal Form (CNF):[7]

$$\tau_{CNF} ::= c_1 \wedge c_2 \wedge \ldots \wedge c_n \qquad c ::= l_1 \vee l_2 \vee \ldots \vee l_m \qquad l ::= \mathsf{t} \mid \mathsf{k}$$

Given selection tag q and a data tag d, both having the form τ_{CNF}, the implication $q \vdash d$ holds whenever each literal l_i in each of the n conjuncts of d appears in all of the n' conjuncts of q. This can be decided in time $O(n'nm)$, where m is the maximum number of literals in each conjunct of d (again assume $O(1)$ set memberships). Pub/sub events and Siena-style subscriptions are degenerate cases in which n and n' are 1, yielding identical performance. A DNF-oriented encoding would be more expensive in this degenerate case (essentially $O(m^2)$), but would perform better on a complementary set of queries (ones that use $\wedge$ more heavily). A hybrid or adaptive encoding would be useful.

Security. Our notion of tags as keys for symmetric and asymmetric cryptography is similar to Gifford's Sealed Objects [6]. Sealed Objects are values encrypted/locked using a set of keys, which can be either combined with *KeyAnd* and *KeyOr*, analogous to our $\wedge$ and $\vee$. Like us, he supports asymmetric keys (not shown in the Table). Unsealing an encrypted value requires presenting the necessary held keys, specified as a conjunction.

In the original Linda model, security is implied by knowledge of the tuple structure: only a correctly-specified pattern (in particular knowing the arity of the tuple) can select the data [4]. A number of researchers have aimed to provide stronger security guarantees, e.g., to prevent untrusted mobile applications from illegally removing data from a Linda-space [10, 9, 17, 11]. These systems use a variety of access control policies (applying to the entire repository or the actions that might be performed) and verification strategies.

Several Linda-based approaches have similarly proposed to provide both security and easy selection. SecOS [15] provides the ability to match encrypted tuples. SecOS values can only be encrypted by one key, whereas our use of logical formulas provides many possible combinations of keys per value, without inhibiting selection. CryptoKlava [2] encrypts tuple elements. Selection requires an agent to select the tuple to decrypt it, and relies on the "good behavior" of the agents to put back the tuples that they cannot decrypt. The key itself does not play any role in the initial selection. It is similar to the example of encrypted tuples we showed in section 3. Finally, `SecSpaces` [8] extends the

[7] As tags are simply propositional formulae, an arbitrary tag can always be converted to CNF before it is stored in a tagged set.

operations allowed on Linda spaces to include partitioning the tuple space based on keys (possibly asymmetric) attached to tuples. These keys are treated as with our $\vee$. After the partitioning, no entry is still encrypted when read, similarly to proposal for distributed selection via secure channels.

Databases often provide fine-grained access control, to the granularity of individual table elements (e.g. Oracle 10g [13]). Tagged values need not adhere to a schema, so users can specify policies per object, rather than per relation. Treating tags as keys also naturally supports storing or communicating data in encrypted form [7].

In general, security in pub/sub systems is challenging and largely unexplored. Our approach tackles one aspect of the problem: *publication security* [16]. In particular, interpreting tags as cryptographic keys permits fine-grained control over what parts of an event should be encrypted, without requiring point-to-point communication. However, it reduces some "content queries" (e.g., is this event's $alarm < 7$) to "existence queries" (e.g., does this event have an *alarm* attribute), placing more work on subscribers to perform filtering. Opyrchal and Prakash [12] describe a means to implement dynamic access control lists within a pub/sub system. Our approach embeds policy with the data, treating tags as capabilities. Compared to access control lists, capabilities support more decentralized policies, but make revocation more challenging.

5 Conclusions

This article presented tagged sets: a data management approach that relies on tags based on propositional logic to lock and select values. The model is flexible, intuitive, and supports fine-grained access control for individual data, as we have shown with many examples. We believe it is a promising approach to organizing and securing data, and for supporting distributed coordination.

For future work, we plan to explore data encodings and selection strategies, drawing on results from pub/sub systems [1] and database systems. We are interested further clarifying the relationship between tagged sets and databases and pub/sub systems, with the hope of finding a useful and efficient encoding in both directions, to support various applications. We are interested in understanding how tagged sets with cryptographic operations could be implemented by peer-to-peer networks.

Acknowledgements Oriol is funded by the Swiss National Science Foundation under grant PBGE2-104794, and Hicks under National Science Foundation grant #0346989. The authors thank Jeff Foster and the anonymous referees for helpful comments on drafts of this paper.

References

1. M. K. Aguilera, R. E. Strom, S. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *Proc. of ACM Symposium on Principles of Distributed Computing (PODC)*, pages 53–62, May 1999.

2. L. Bettini and R. D. Nicola. A Java middleware for guaranteeing privacy of distributed tuple spaces. In *Proc. International Workshop on Scientific Engineering for Distributed Java Applications*, pages 175–184, 2003.
3. P. Buneman, A. Deutsch, and W. C. Tan. A deterministic model for semi-structured data. In *Proc. of the 1999 Intl. Workshop on Query Processing for Semi-Structured Data and Non-Standard Data Formats*, Jan. 1999.
4. N. Carriero and D. Gelernter. Applications experience with Linda. *ACM SIGPLAN Notices*, 23(9):173–187, Sept. 1988.
5. A. Carzaniga, D. S. Rosenblum, and W. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In *Proc. of ACM Symposium on Principles of Distributed Computing (PODC)*, pages 219–227, July 2000.
6. D. K. Gifford and A. K. Jones. Cryptographic sealing for information security and authentication. *Communications of the ACM*, 25(4):274–286, Apr. 1982.
7. J. He and M. Wang. Cryptography and relational database management systems. In *International Database Engineering and Application Symposium*, pages 273–284, 2001.
8. R. Lucchi and G. Zavattaro. WSSecSpaces: a secure data-driven coordination service for web services applications. In *Proc. of ACM symposium on Applied Computing (SAC)*, pages 487–491, 2004.
9. N. H. Minsky, Y. M. Minksy, and U. Ungureanu. Making tuple space safe for heterogeneous distributed systems. In *Proceedings of SAC '2000*, pages 218–226, 2000.
10. R. D. Nicola, G. Ferrari, and P. Pugliese. Programming access control: The KLAIM Experience. *Lecture Notes in Computer Science*, 1877:48–??, 2000.
11. A. Omicini and F. Zambonelli. Tuple centres for the coordination of Internet agents. In *Proceedings of the 1999 ACM Symposium on Applied Computing (SAC'99)*, pages 183–190, San Antonio (TX), Feb. 28 - Mar. 2 1999. ACM Press. Track on Coordination Models, Languages and Applications.
12. L. Opyrchal and A. Prakash. Secure distribution of events in content-based publish subscribe systems. In *Proc. of USENIX Security Symposium*, 2001.
13. Oracle. Oracle database 10g security and identity management. Technical report, Dec. 2003.
14. B. Segall and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Proceedings of AUUG97*, Brisbane, Australia, September 1997.
15. J. Vitek, C. Bryce, and M. Oriol. Coordinating processes with secure spaces. *Science of Computer Programming*, 46:163–193, January-February 2003.
16. C. Wang, A. Carzaniga, D. Evans, and A. Wolf. Security issues and requirements for internet-scale publish-subscribe systems. In *HICSS '02: Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS'02)-Volume 9*, page 303. IEEE Computer Society, 2002.
17. A. Wood. Coordination with Attributes. In P. Ciancarini and A. Wolf, editors, *Proc. 3rd Int. Conf. on Coordination Models and Languages*, volume 1594 of *Lecture Notes in Computer Science*, pages 21–36, Amsterdam, Netherland, Apr. 1999. Springer-Verlag, Berlin.