Available online at www.sciencedirect.com

ScienceDirect

Science of Computer Programming 64 (2007) 205–222

Science of Computer Programming

www.elsevier.com/locate/scico

# Coordination with multicapabilities

Nur Izura Udzir[*,1], Alan M. Wood, Jeremy L. Jacob

*Department of Computer Science, University of York, York YO10 5DD, UK*

Received 15 September 2005; received in revised form 15 March 2006; accepted 15 June 2006
Available online 26 September 2006

**Abstract**

In the context of open distributed systems, the ability to coordinate agents coupled with the possibility to control the actions they perform is important. As open systems need to be scalable, capabilities may provide the best-fit solution to overcome the problems caused by the loosely controlled coordination of tuple-space systems. Acting as a 'ticket', capabilities can be given to the chosen agents, granting them different privileges over different kinds of data—thus providing the system with a finer control on objects' visibility to agents. One drawback of capabilities is that they can only refer to named objects—something that is not universally applicable since, unlike tuple-spaces, tuples are nameless. This paper demonstrates how the advantages of capabilities can be extended to tuples, with the introduction of *multicapabilities*, which generalise capabilities to collections of objects. We also present discussions on implementation and application examples to illustrate the use of capabilities and multicapabilities in tuple-space systems.

*Keywords:* Tuple-space coordination; Capabilities; Multicapabilities

## 1. Introduction

Coordination is essential in open systems, where agents and active objects are free to join and leave the system at any time, i.e. they need not be defined prior to starting the infrastructure. The discussion in this paper is based on the tuple-space, or LINDA model [10,5,9] as an open distributed system. LINDA promotes *generative* communication where agents interact by 'generating' data (an ordered collection of typed values called a *tuple*), using the primitive `out` into a shared data space known as a *tuple-space* (TS). A tuple can be retrieved, destructively (using `in`) or non-destructively (using `rd`), from the tuple-space by specifying a *template* whose pattern matches the tuple. Both `rd` and `in` block if no matching tuple is available. Non-blocking versions of these primitives, `inp` and `rdp` were originally introduced. However, the definition of these primitives was ill-formed, until the proposal of a principled semantics of `inp` [12], based on a deadlock-breaking mechanism, which the work reported here enhances. The associative matching retrieval is *non-deterministic*: a retrieving agent may get *any tuple* that matches its template; and a tuple may be given to *any agent* specifying a matching template.

* Corresponding author.
*E-mail addresses:* izura@fsktm.upm.edu.my (N.I. Udzir), wood@cs.york.ac.uk (A.M. Wood), Jeremy.Jacob@cs.york.ac.uk (J.L. Jacob).
[1] The author's other affiliation is the Department of Computer Science, Universiti Putra Malaysia.

0167-6423/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2006.06.005

A popular alternative to the conventional point-to-point communication approaches, LINDA is distinguished by its temporal and spatial separation properties, as well as its independence from any computation language or machine architecture—which are essential properties for coordination in open systems. Now a mature technology, research in tuple-space based coordination is providing general-purpose data spaces to create efficient large-scale implementations of open distributed multi-component systems.

LINDA's power for coordination in an open, heterogeneous environment is well known. However, in order to benefit from the advantages of open and flexible coordination mechanisms, a number of challenging practical problems need to be addressed. These have been noted by many authors in the past, and several solutions have been proposed, all imposing varying degrees of additional control by the system. Unfortunately, getting the optimum balance between flexibility and tighter control is difficult, and many of the proposed solutions lose the principal advantages that LINDA-like systems have over many other models.

It is useful, however, to provide a finer control over the agents' interactions and coordination, than is available in the 'classic' LINDA model. This is particularly challenging in a decentralised, and distributed environment, while at the same time maintaining the flexibility inherent in open systems. One aspect of having a finer control is to be able to restrict what methods an agent is allowed to invoke on an object. Earlier work on coordination using object attributes [29] demonstrated a simple solution to control agents' access on objects in the system without resorting to any complex cryptographic security approach. Together with access control lists (ACLs), this earlier paper also discussed the advantages of capability-based control [8,14] in distributed environments. A capability [19] is an unforgeable 'ticket' given to an agent that specifies which kind of actions on a certain object are permitted to the holder of the capability. We can define capabilities in a more general way: as 'visibility' filters to create a more refined control over agents' actions on objects in the system.

Capabilities are particularly suitable for open distributed systems as they themselves are *distributed* in the sense that the controlling attributes are held by the agents, rather than being attached to objects; and they are transferable, where any agent can pass a copy of a capability it holds to another. In addition, a capability mechanism also supports the *flexibility* inherent in distributed systems: it accommodates user-defined rights, not restricted to those pre-defined by the system, thus allowing them to be dynamically changed; and its 'domain flexibility' feature allows agents to join and leave the system simply by requesting (and possessing) appropriate capabilities to access the objects, as opposed to having to modify numerous lists attached to each object relevant to the agents' execution.

While various capability systems have been developed over the years, and is still an active research in some areas such as in object-based systems, it does not enjoy the same popularity in the tuple-space based systems.[2] Despite the works of Chung and McDonald [7] and Gorla and Pugliese [11], for instance, it seems that capability-based coordination has yet to demonstrate a significant impact to improve LINDA-like coordination in the open distributed environment.

Although capabilities have their drawbacks, mainly related to their management, it has also been established that capabilities offer more flexibility than access control lists [29,7,24], thus making them more attractive for open systems. However, unlike access control lists, capabilities must refer to single named objects. In the LINDA context, tuple-spaces are uniquely identifiable—therefore, they can be referenced by capabilities—whereas tuples are anonymous: they can only be referred to using associative matching. We propose a solution to this problem by using *multicapabilities*, as will be elaborated in the next section.

## 2. Multicapabilities

Multicapabilities extend capabilities for a collection of objects, rather than the traditional notion of a unicapability referring to a single *named* object. Whereas a unicapability allows its holder to operate in a certain way on the *object* it refers to, a permission in a multicapability allows the operation on any *element* within the group, not on the entire group referred to by the multicapability. In the LINDA context, multicapabilities extend capabilities to apply to *nameless* tuples without jeopardising the associative matching and the non-deterministic properties of the tuple-space model. This enables various operations to be performed on the tuples, since they can now be referenced.

Tuples are classified by their multicapabilities. Two tuples with the same pattern (possibly having the same, or different values) may be referred to by two different multicapabilities, while the same multicapability can refer to

[2] At least, none have been proposed as a 'pure' capability system, as capabilities are combined with other security techniques.

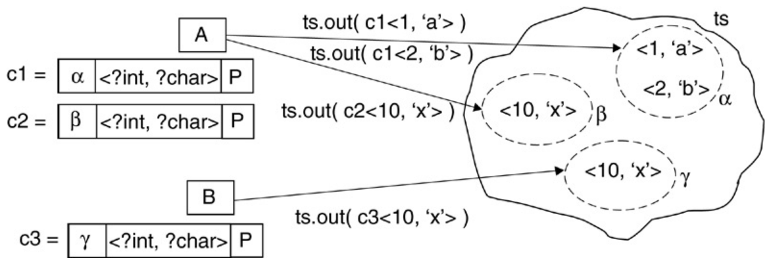

Fig. 1. TS operations using multicapabilities.

(a collection of) many different tuples (with any value). To illustrate this, let us consider a simple example where an agent, $A$, wishes to `out` a tuple of an integer and a character, `<?int,?char>`. Before referring to a class of tuples, either for output or retrieval, it must first request an appropriate multicapability from the kernel.[3] It will be given a multicapability[4] for the template with a unique multicapability name, e.g. $\alpha$:

$$c1 = [\alpha, \langle ?int, ?char\rangle, P]$$

where $P$ is the set of all permissions, granting the agent full rights to perform all operations on tuples which match the template. For the purpose of this discussion, let us assume that $P = \{i, r, o\}$, where $i$, $r$ and $o$ represent permissions to perform the operations `in`, `rd`, and `out`, respectively.

If $A$ makes another request for a multicapability, $c2$ for a similar template from the kernel, it will receive a *new* multicapability, different from $c1$, i.e. $[\beta, \langle ?int, ?char\rangle, P]$, where $\alpha \neq \beta$.

Now assume another agent, $B$, requests a multicapability for a similar template from the kernel, and receives a multicapability which it stores in $c3$: $[\gamma, \langle ?int, ?char\rangle, P]$, where $\gamma \neq \alpha \neq \beta$.

Using the multicapabilities they possess, the agents can perform operations (within those permitted by $P$), e.g. writing tuples into a tuple-space (Fig. 1).

Even though the templates are identical, they are distinguished by the multicapabilities used to create (`out`) them. The result of any input operation performed by the agents will be limited to only the tuple(s) within the specified multicapability group. For example, a `rd` operation for `<?int,?char>` performed by $A$ using $c1$ will retrieve tuple `<1,'a'>` or `<2,'b'>`, but not any of those in other groups. Likewise, if $B$ performs an input operation

```
ts.in( c3<?int,'a'> );
```

the operation will block—there is no matching tuple in the group identified by $\gamma$. The template `<?int,'a'>` of multicapability $c3$ cannot match tuple `<?int,'a'>` of multicapability $c1$, even though they are of the same pattern.

Therefore, multicapabilities provide a partitioning of a tuple-space, thus enabling certain operations to be performed on tuples of a specific group, but not on those of another group, even though both groups have the same template.

## 2.1. Basic structure

A multicapability refers to a group of objects of a certain template, or pattern. In the LINDA context, a template is defined as a sequence of types and/or scalars. We define a multicapability as a structure consisting of three parts: $u$, a unique tag or identifier which acts as a reference to a collection of objects; $t$, a template of the objects; and $p$ which denotes the set of actions permitted on elements of the collection. In object-oriented terms, this set corresponds to a sub-interface of the methods in the objects' class. Hence, a multicapability $c$ can be written as $[u, t, p]$. In the case of unicapabilities, the capability for a *single* object $o$ is also the 'handle' to the object, with permission $p$ attached, $[o, p]$. We shall use the term 'capability' to refer to capabilities in general, and the terms 'unicapability' or 'multicapability' accordingly when referring to a specific class.

3 The term 'kernel' in this paper refers to the underlying distributed coordination mechanism that controls all operations in the system. It represents the totality of the LINDA 'middleware'.

4 A more formal definition of the multicapability structure will be given in Section 2.1.

The appropriate capability must be presented to the kernel for verification before an action is allowed to be realized. Having the required permission would mean that the action is valid for the target object. If an agent attempts to perform an action, $a$ on object $o$, using capability $c = [o, p]$, it will be allowed to take place (*succeed*) if $a \in p$. Likewise, in the case of multicapability, $c = [u, t, p]$: an action $a$ with a value $v$ is allowed to take place if $a$ is within the permission set $p$ of $c$, and $v$ matches the template $t$ of the multicapability group identified by $u$. Therefore, we define a function *allowed* where

$$allowed([u, t, p]) = \{a.v | a \in p \wedge v \textbf{ match } \langle u, t\rangle\}.$$

It should be reiterated that we view the permissions in a capability as a set of names of methods that the holder is allowed to invoke, and so are not limited to access control, but represent a more general concept: *visibility*—an agent cannot see a method that is not listed in the capability it holds. As the term 'object' can refer to anything that is appropriate to the system, the set of rights may well include any appropriate, even user-defined, operations.

The structure of our model is based on the following rules:

(1) Every tuple-space and tuple operation requires a capability. Our model requires that each agent performing an operation on a tuple must obtain a unicapability to access the tuple-space where the tuple resides (or to be written to), as well as the multicapability to operate on the tuple. Therefore, prior to performing an operation, all agents must first hold:
   (a) a unicapability for the target tuple-space, and
   (b) a multicapability for a specified template.
(2) Every request for a new capability returns a new, unique capability (even for identical patterns), with full rights.
(3) All agents have the full capability (with full rights $\top$) for the universal tuple-space (UTS), i.e. a default space that exists throughout the life-span of the system, and that is (publicly) accessible by all agents in the system. This unicapability only represents the permission to perform operations on the contents of the universal tuple-space in general, but multicapabilities for tuples within the universal tuple-space are required in order to operate on these tuples (See Rule 1).
(4) Each agent has a default universal capability for capabilities, $cc[\langle ?cap\rangle]$ with 'the least' rights, where $cap$ is a capability type, to enable capabilities to be passed among the agents. For the purpose of the discussions in this paper, we assume $cc$ to be $[\aleph, \langle ?cap\rangle, \{r, o\}]$ which allows non-destructive read and write permissions on capability type tuples.

The capability data (unique identifier, reference/template, and permissions) are assumed to be securely encapsulated in the capability and only interpretable by the kernel when the capability is presented for verification. To avoid confusion, it is important *not* to see multicapabilities as 'tagging' tuples—this leads to an ACL-like view. Rather, it is better to view tuples being grouped into regions specified by the multicapabilities referring to them. Each region then exists (virtually) in every tuple-space.

## 2.2. Operations on multicapabilities

A multicapability may be copied to be given to other agents. Each copy would have the same unique tag as the original, thus referring to the same group of objects. However, the template and permissions in a (copy of a) multicapability can be reduced to allow agents more control over the visibility of the objects they created to other agents. The *permissions* $p2$ in the reduced copy of a multicapability should not exceed those in the original multicapability ($p1$), i.e. $p2 \subseteq p1$; whereas its *template* can be the exact copy of the original template, or specialised to a sub-type of the same pattern.

We define two operators for this purpose: $-$ and @, where $c - s$ is capability $c$ *without* the permissions in $s$, and $c@s$ is $c$ *with only* the permissions in $s$ that $c$ also possesses. Let $c = [u, t, p]$ be a multicapability, and $s$ a set of permissions where $s \subseteq p$, then

$$\begin{array}{ll} \_ - \_ \in Mulc \times \mathbb{P}(P) \longrightarrow Mulc & \_@\_ \in Mulc \times \mathbb{P}(P) \longrightarrow Mulc \\ [u, t, p] - s = [u, t, p \backslash s] & [u, t, p]@s = [u, t, p \cap s] \end{array}$$

Even though the two operators serve similar purposes, they are separately defined for convenience, for instance, it is easier to express a sub-multicapability with only two (out of ten) permissions granted using @, rather than using $-$.

It should be emphasised that there is *no way* of adding permissions to a multicapability, nor to expand the scope of the template in the multicapability. For instance, if the template in the original multicapability is `<?int,'x'>`, then a copy of it can never have a template with the second element set to `?char` as this will generalise the template, and would defeat the purpose of the use of capabilities. Therefore, a *derived-from* $\preceq$ relation can be defined as:

$$[u, t, p] \preceq [v, s, q] \equiv\ u = v \wedge\ t \leq s \wedge\ p \subseteq q$$

where $t \leq s$ means $\forall i \in 1..n \bullet t_i \in s_i \vee t_i = s_i$, and $i$ is the index of an element in a template; $n$ is the length of the template, i.e. the maximum number of fields in a template allowed by the system.[5]

Derivation contributes to finer control over the objects in the system as it provides a means to have different versions of a multicapability (with different restrictions) referring to tuples of the same template.

### 2.2.1. Example: Reader–Writer

Consider a simple example of Reader–Writer, assuming that there are only two agents in the system: one reader and one writer. Suppose that the *Writer* agent intends to `out` a tuple `<1,2>`. Requesting, and consequently receiving the appropriate multicapability from the kernel, the *Writer* may then make a modified copy of the multicapability if necessary, by 'reducing' the template and the list of rights, before writing the tuple (using the derived multicapability) into the universal tuple-space.

The *Reader* agent wants to read a tuple that matches the template `<?int,2>`. If it requests a new multicapability for the template, it will receive a unique multicapability, different from the one given to *Writer*. This means that it can never retrieve any tuple produced by *Writer* (or any other producer, for that matter) with its newly acquired multicapability. If *Reader* wishes to read the tuple `out`'ed by *Writer*, it must use the same multicapability (or a derivation) as the tuple's. One way of doing so is to obtain it from *Writer*: since every agent automatically gets the default *cc*, *Writer* can pass the tuple containing the said multicapability to *Reader* via the universal tuple-space (UTS).

The following code excerpts illustrate the agents' operations where *Writer* `out`s the tuple `<1,2>` and the restricted multicapability $[\alpha, \langle ?int, ?int\rangle, \{r, o\}]$; and *Reader* retrieves the necessary multicapability before reading a two-integer tuple using the multicapability.

*Writer*:
```
    // Request new multicapability
    c1 = newcap( <?int,?int> );
    // c1 now is [α,<?int,?int>,{i,r,o}]
    UTS.out( c1<1,2> );   // write tuple into UTS
    c1_1 = c1 - {i};      // reduce permission
    // c1_1 holds [α,<?int,?int>, {r,o}]
    UTS.out( cc<c1_1> );  // write capability for the tuple
```

*Reader*:
```
    // Retrieve the multicapability
    cap1 = UTS.in( cc<?cap> );
    // Assuming cap1 now holds [α,<?int,?int>,{r,o}]
    data = UTS.rd( cap1<?int,2> );
```

Prior to performing an action, the kernel will verify that the intended action is valid based on the multicapability presented by the agent. To `out` a tuple, *Writer* has to present the multicapability $c1$ for verification, and since the `out` permission in its set of rights verifies that the action is valid, the action is allowed to proceed. Note that *Writer* can `out` a tuple of $c1$ or $c1\_1$: both multicapabilities refer to the same collection of tuples.

After obtaining $c1\_1$, *Reader* can specialise the values (in the template) to suit its need. As $c1\_1$ is a sub-capability of $c1$, it still refers to the same object(s) as $c1$, but with restricted permissions.

Note that we are assuming they are the only agents in the system. In real systems with a large number of agents, however—due to the non-deterministic property of LINDA—there is no guarantee which agent might read

[5] Note that a more general and expressive condition based on *sub-typing* is possible. However, for simplicity we only consider the relation based on templates here.

the capability tuple $c1\_1$, hence gaining access to the tuple produced by the *Writer*. Should the two agents wish to eliminate interference, they can establish private channels for secure conversation, as described in [27].

### 2.3. Combining multicapabilities

In this paper, we also present preliminary work on combining multicapabilities: operations which take one or two multicapabilities and produce a new multicapability. For example, three obvious operations that might be performed on two multicapabilities are the union, intersection, and relative negation. Naturally it is sensible to only allow the combination of those multicapabilities with a similar pattern that belong to the same agent. The multicapability produced as a result of these operations may be stored in another capability variable.

#### 2.3.1. Union

Assuming the function *allowed* given in Section 2.1, the *union* of two multicapabilities $c$ and $d$ can be defined by

$$allowed(c \cup d) = allowed(c) \cup allowed(d).$$

The expression produces a multicapability referring to the templates of either $c$ or $d$, and represents permissions if *either* $c$ or $d$ (or both) grants that permission. If both multicapabilities refer to the *same* collection of tuples, i.e. $c_u = d_u$,[6] then

$$c \cup d = [c_u, c_t, (c_p \cup d_p)].$$

To further elaborate on this operation, let us consider the following multicapabilities:

$$c1 = [\alpha, \langle ?int, ?char \rangle, \{i, r, o\}]$$
$$c2 = [\beta, \langle 3, ?char \rangle, \{r, o\}]$$

If $c1$ and $c2$ refer to the same group, i.e. if $\alpha = \beta$, writing a combined tuple of these multicapabilities, e.g.

```
ts.out( (c1 ∪ c2)<3,‘a’> );
```

produces the tuple into the group referred to by both multicapabilities. It should be emphasised that the semantics allow the tuple to be of either templates—which is reasonable as the agent *does* possess these multicapabilities. The union operation merely gives it an added advantage. The produced tuple can be retrieved by any agent possessing either $c1$ or $c2$, and not necessarily both multicapabilities. However, as the tuples are written using multicapabilities of different rights, then any agent who has $c1$ (or both multicapabilities) can `in` or `rd` the tuple, while those with $c2$ can only `rd` it.

Holding a capability resulting from a union of two capabilities—received as a ‘package’ from some agent—is different from holding each of the two capabilities separately: firstly, a retrieval using the union must search *both* groups; the holder cannot *choose* which component individually. Secondly, the holder cannot ‘extract’ one capability from the ‘union capability’ to give to another agent.

A read operation using the union of $c1$ and $c2$, would operate on the tuples that match the templates `<3,‘a’>` or `<?int,?char>` from the region of $c1$ or $c2$.

If the multicapabilities refer to *different* collections of objects, then the operation should be a *disjoint union*. Uniting (conjoining) the templates and the permission sets would be semantically incorrect, as the permission sets cannot be merged—even though they contain method(s) with the same name, these methods are of different signatures, e.g. method $m$ in (signature) $c$ is not the same as method $m$ in $d$: they are applicable to different collections of objects. Therefore, a disjoint union of multicapabilities referring to two different groups is defined as

$$c \cup d = [c_u, c_t, c_p] \uplus [d_u, d_t, d_p].$$

This means that the intended action will be performed on *either* group non-deterministically chosen by the kernel.

Performing a disjoint union on $c$ and $d$, where $c_u \neq d_u$, would mean that the system will (non-deterministically) choose one of the groups, before performing the action $a$ on an object in the said group. Assuming the group selected

[6] We shall use $c_u$, $c_t$, and $c_p$ hereafter to refer to the unique tag $u$, the template $t$, and the permission set $p$ of multicapability $c$.

is $c_u$, the action succeeds if $a \in c_p$, and fails otherwise. In LINDA, however, there is another (richer) possibility: if $a \notin c_p$, instead of failing, the action blocks (on $c_u$), but the system may allow the action to be performed on the other group, i.e. $d_u$. If the action succeeds this time around, upon completion, the previously blocked action will be broken.

This is useful when an agent is looking for a particular tuple, but does not know which group the tuple belongs to. Since the agent has multicapabilities to both groups, it can simply perform a read from (a disjoint union of) $c_u$ or $d_u$. Without the union operation, the agent needs to attempt to read from one group, before the other, and risks being blocked on the first attempt, before it has the chance to try reading from the second group. With union, the probability of being blocked can be reduced. Indeed, what is more, possible deadlocks can be avoided—a sequence of two input operations may deadlock, whereas the union (which is equivalent to a parallel combination of two input operations) will only block until a tuple becomes available in either group.

The union operation allows actions permitted by any of the two multicapabilities to be performed on either of the two templates for the same collection of tuples. However, if the union involves different collections of tuples, the action will only be permitted if it is allowed by the permission set of the group non-deterministically selected by the kernel.

**Example.** Consider a problem where an agent wishes to produce a tuple containing an arbitrary capability `<prodCap>` which is *removable* by the producer agent itself, but can only be *read* by others. We know that a capability tuple can be accessed using the universal *cc* which does not allow destructive read. For explicit garbage collection, the producer must request another capability (for a capability type), e.g. *cc*1, which grants full rights, including `in`. As sketched in the code excerpts below, it can then write the tuple `<prodCap>` using the union of the new capability and *cc* (assuming + is the ASCII representation for union). Since the other agents do not have a copy of *cc*1, they can only read `<prodCap>` using *cc*. The tuple can later be removed by the producer using *cc*1.

```
        Producer:                           Other(s):

cc1 = newcap( <?cap> );             ts.rd( cc<?cap> );
ts.out( (cc+cc1)<prodCap> );        // in is not allowed
...                                 ...
ts.in( cc1<?cap> );
```

### 2.3.2. Intersection

Generally, an *intersection* of two multicapabilities *c* and *d* is defined by

$$allowed(c \cap d) = allowed(c) \cap allowed(d)$$

which is a multicapability referring to the 'lesser' (i.e. the less generic) template of the two, and represents permissions only if *both* *c* and *d* grant that permission. If *c* and *d* refer to the *same* group of objects, $(c_u = d_u)$, then

$$c \cap d = [c_u, (c_t \cap d_t), (c_p \cap d_p)].$$

Otherwise, if *c* and *d* refer to *different* groups, then

$$c \cap d = [(c_u, d_u), (c_t \cap d_t), (c_p \cap d_p)].$$

As the permission sets are intersected, an action *a* will always be valid for both regions, except, of course, if $a \notin (c_p \cap d_p)$ or if $(c_p \cap d_p) = \emptyset$. $(c_u, d_u)$ implies that the action will be performed on *both* regions simultaneously.

Using the multicapabilities $c1$ and $c2$ defined in the previous section, an `out` operation using $(c1 \cap c2)$ will write a tuple which can only be of the template `<3,?char>` which is 'less' than `<?int,?char>`. The tuple is written into the region referred to by both multicapabilities, if $\alpha = \beta$; otherwise, if the multicapabilities refer to different regions, then it is written into the 'shared' (i.e. intersecting) section of the two multicapabilities, which is tagged with $(\alpha, \beta)$. This tuple can only be retrieved by an agent that possesses both multicapabilities. In the previous example of $c1$ and $c2$, the kernel will only allow the tuple to be `rd`—the only permission granted by both multicapabilities. The read operation using $(c1 \cap c2)$ will operate on tuples of the more restricted version of the two templates that exist in both groups, which can be regarded as the shared region of $\alpha$ and $\beta$.

Therefore, intersecting two multicapabilities allows actions limited to only those permitted by both multicapabilities to be performed on the less generic of the two templates.

**Example.** One way of modelling the 'shared' section is by attaching special 'shared' tags to virtual copies of the tuple. 'Virtual copies' implies that there exists only one tuple that should be accounted for: a removal of one of these copies would result in all the others being deleted. This would be useful in the case of data caching (see 4.3), where the producer can write the tuple into different partitions (represented by different multicapability regions) of the tuple-space using the intersection of the capabilities for these partitions. Whenever a tuple is `in`'ed, all virtual copies of the tuple (with the same 'shared' tags) can automatically be deleted.

### *2.3.3. Negation*

Another conceivable operation is the *relative negation* of multicapabilities, which is defined by

$$allowed(c - d) = allowed(c) - allowed(d).$$

The produced multicapability has a permission *only if* $c$ grants that permission, but $d$ does not, and operates on tuples of the template of $c$, except those of the $d$ template.

$$c - d = [c_u, (c_t \cap \neg d_t), (c_p \backslash d_p)].$$

For example, let $c3$ be another multicapability as an addition to the previously defined $c1$ and $c2$,

$$c3 = [\gamma, \langle 3, ?char \rangle, \{i\}].$$

An output operation using the negation expression ($c1 - c3$) will write a tuple of template `<?int,?char>`, excluding any tuple whose first element is `3`. This tuple will be written in the $c1$ region, and can only be retrieved by agents holding a copy of $c1$; it is not accessible with $c3$.

Similarly, an input operation using the expression ($c1 - c3$) will operate on tuples in the $c1$ region, while selectively disregarding any tuple that matches the template of $c3$. This provides further filtering mechanism for agents who wish to retrieve a certain template of data, except those with certain values.

The relative negation allows actions permitted by the multicapability on the left-hand side of the operator, except those in the right-side multicapability to be performed on any tuple that matches the left-side template, but not the other.

**Example.** Consider an administrative application agent which processes student records. To access all the student records, the agent has been given a capability $c_{\text{all}}$, for example. If the agent wants to retrieve records of all students except the first year students, it can produce a restricted copy of $c_{\text{all}}$, specialize the `Year` field to `'1'` (assuming `Year` is one of the fields in the tuple), and perform the negation operation ($c_{\text{all}} - c_{y1}$) in its `rd` operation.

The combinatorial operations of capability expressions offer richer possibilities for capabilities to be manipulated to provide a finer control on objects, visibility to agents in open systems. The above operations (union, intersection, and relative negation) are the most obviously applicable given the fundamental properties of a multicapability. There may be others that might usefully be defined—a possible future work—to identify a *sufficient* set of operations. This paper focuses on identifying *conceivably useful* operations which can be performed on multicapabilities (and capabilities), and work on the *feasible implementation* is ongoing.

## 3. Implementation overview

Every LINDA implementation assumes different characteristics. In this paper, our LINDA-like capability-based system Lindacap assumes an implementation having the following main characteristics:

- There is a public universal tuple-space (UTS) accessible by all.
- All agent communications are done solely in terms of tuple-space operations.
- Every tuple-space, with the exception of the universal tuple-space, is explicitly created by agents in the system.
- tuple-spaces in the kernel are flat structured. Every tuple-space is created at the same level, that is, tuple-spaces cannot be created inside others.
- Capabilities (handles/references) for tuple-spaces are first class objects.

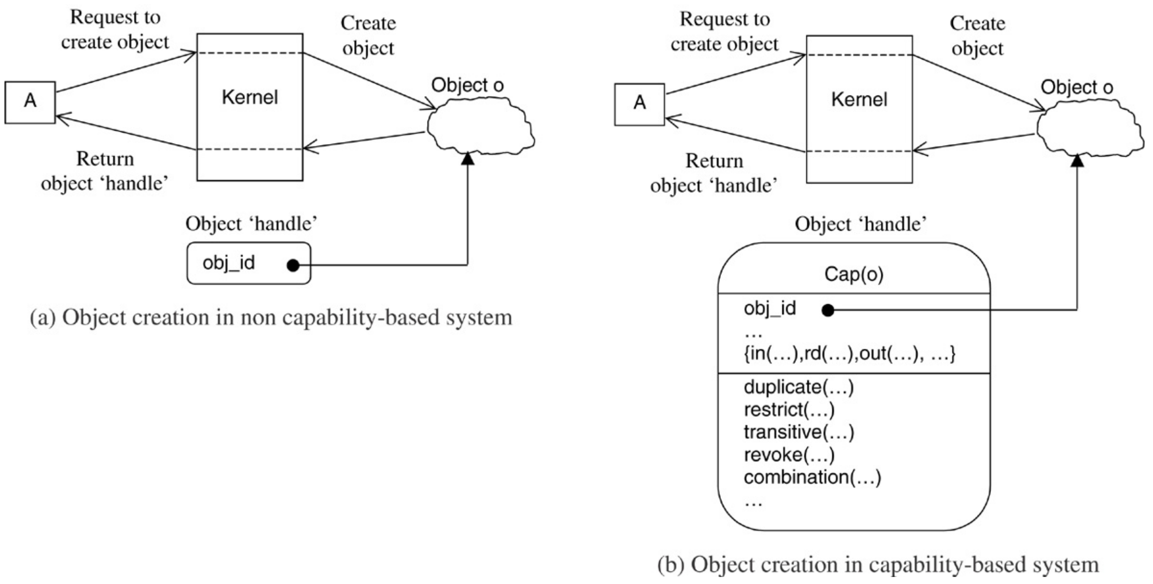

Fig. 2. Creating an object in capability system.

### 3.1. *Implementing unicapabilities*

Like many regular capability-based implementations, when an object is created, a capability for the object is returned to the creator [26,25,7]. Our view is that a capability itself is an object, just like other objects. This view differs from that of some authors (e.g. Peterson [20]) who believe that capabilities should be distinguished from other kinds of objects, and interpreted by an abstract machine on which higher-level programs run. Therefore, being an object, a capability has methods associated with it that define the operations allowed on the capability itself, as depicted in Fig. 2(b).

When a request to create a tuple-space is sent to the kernel, the kernel will call the necessary functions to create a new tuple-space and a unique capability, which will then be used as the tuple-space handle. The capability consists of a unique identifier, which is the identifier of the tuple-space itself, and the access rights. Naturally, by default, the capability grants full rights to access the tuple-space. For simplicity, we define three rights associated with a TS-capability: the right to `out` a tuple into the said tuple-space, and the rights to `in` and `rd` a tuple from the tuple-space. This capability is then returned to the requestor, who may later produce a modified, or an exact, copy of the capability to be given to another agent.

Tuple-spaces have unique handles: in a non-capability based system, a tuple-space handle contains a unique identifier pointing to the actual tuple-space. In a capability-based system, capabilities can be used as the handle to the tuple-space—containing the tuple-space identifier with some extra internal states and a set of rights that defines how the tuple-space will be 'viewed' by an agent holding the handle. These handles can be passed between agents via tuple-spaces in a tuple.

### 3.2. *Implementing multicapabilities*

In a non-capability system, each tuple-space can be seen as associated with a single `tuplecontainer`, a data structure used to store the tuples; in Lindacap, a tuple-space may have more than one `tuplecontainer`, each `tuplecontainer` represents a unique multicapability group. Multicapability groups are implemented as regions in any tuple-space, and one multicapability region is not restricted to one particular tuple-space. Just as tuple-spaces are distributed over multiple hosts, multicapability regions are distributed over tuple-spaces.

A multicapability object is created via a *newcap* call to the kernel, which returns a unique multicapability object with full rights (Fig. 3(a)). Whereas a unicapability object (e.g. a TS-capability) is returned for the newly created object (following a request), as shown in Fig. 2(b), requesting for a multicapability would return a new multicapability object for the specified pattern of objects (say a group of tuples), but the referred region is not yet created. The region

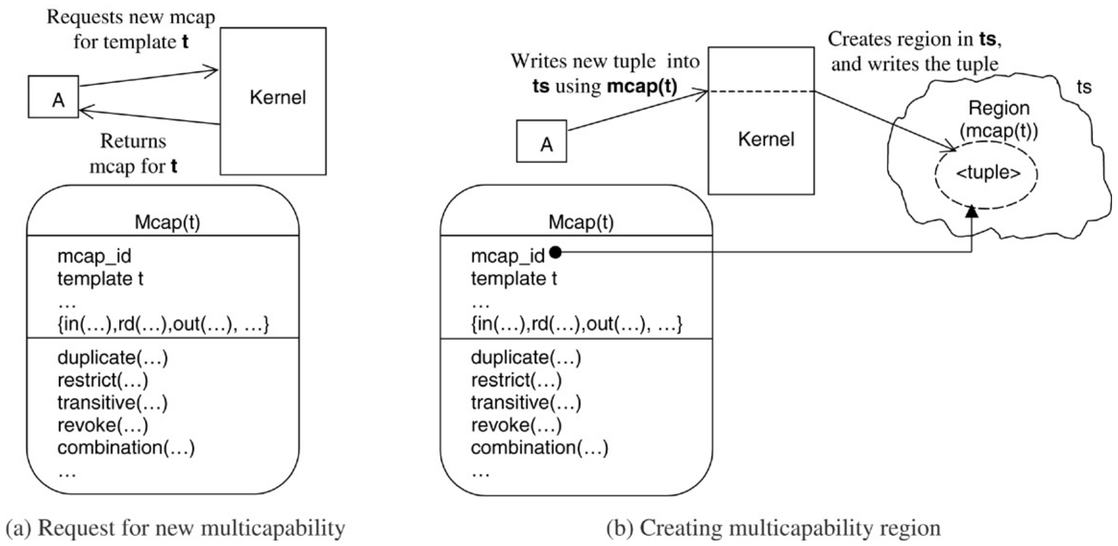
Requests new mcap for template **t**
A
Kernel
Returns mcap for **t**
Mcap(t)
mcap_id
template t
...
{in(...),rd(...),out(...), ...}
duplicate(...)
restrict(...)
transitive(...)
revoke(...)
combination(...)
...
(a) Request for new multicapability
Writes new tuple into **ts** using **mcap(t)**
Creates region in **ts**, and writes the tuple
ts
Region (mcap(t))
<tuple>
(b) Creating multicapability region

Fig. 3. Requesting a multicapability and creating the region.

will only be created when the first tuple is `outed` using the multicapability (Fig. 3(b)). The reason for this is that a multicapability region is not associated with any tuple-space, therefore, its location cannot be determined prior to an `out`, which specifies the destination tuple-space.

### 3.3. Passing capabilities

In order for information to be shared among agents, capabilities may be passed between them, via tuple-spaces, copies of capabilities being restricted if needed before being disseminated to other agents. Capabilities, like any first class objects, may be elements of tuples and thus stored within tuple-spaces. With the universal primordial capability, $cc$, these tuples may be retrieved by another agent, which will then be able to access the object referred to by the capability.

Our implementation also incorporates *tuple monitoring* [15] for the kernel to maintain some information on the capabilities being passed as tuple elements, which will be essential for garbage collection and deadlock detection (see 4).

### 3.4. Descriptions of the primitives

Capabilities in Lindacap are created via calls to the kernel. The methods in capabilities, such as restriction,[7] are invoked in a similar way. The primitives to create and restrict capabilities in Lindacap are:

`TScap` *`TupleSpaceC`*`()`: This call creates a tuple-space in the server, and a unique capability object, `TScap` with full rights, acting as the handle to the tuple-space, is returned. Capabilities are first class objects.

`TScap` *`restrictTScap`*`(TScap` *`ts`*`, Bool` *`p_out`*`, Bool` *`p_rd`*`, Bool` *`p_in`*`)`: This method creates and returns a new instance of a capability object for *`ts`*, with the set of rights as specified in the parameter list.

`Multicap` *`newcap`*`(Template` *`tmp`*`)`: This call to the kernel creates a new instance of a multicapability object for template *`tmp`*, and returns the multicapability with unique identifier and a full set of rights to the caller.

`Multicap` *`restrictMcap`*`(Multicap` *`mcap`*`, Template` *`tmp`*`, Bool` *`p_out`*`, Bool` *`p_rd`*`, Bool` *`p_in`*`)`: This call creates and returns a new instance of a multicapability object for *`mcap`*, with the same identifier, but the template and the set of rights of this new `Multicap` object may be overridden, as specified in the parameter list.

[7] We do not discuss the other methods of capabilities as they are not relevant to the discussion in this paper, but they include duplication, restriction, transitivity, revocation, and combination.

The basic input/output primitives in Lindacap extends the basic primitives of LINDA with multicapabilities as additional parameters. The primitives are:

`void out(TScap ts, Multicap mcap, Tuple tup)`: Given that the `TScap` and `Multicap` grants the *out* permission, this method stores a `Tuple` in the region specified by `Multicap`, creating a new region if none exists for that multicapability.

`Tuple rd(TScap ts, Multicap mcap, Template tmp)`: If both the `TScap` and `Multicap` grants the *rd* permission, this method searches for a `Tuple` matching the template `tmp` within the region specified by `Multicap` in the tuple-space `ts`. If a matching tuple is found, it is returned; otherwise, the method blocks until one becomes available.

`Tuple in(TScap ts, Multicap mcap, Template tmp)`: This method executes in a similar way to `rd`, except that the matching tuple is removed from the `mcap` region and returned.

Lindacap also provides predicated primitives `inp` and `rdp` [12,22,23], which are similar to the descriptions of `in` and `rd` above, except for their unblocking properties.

## 3.5. Implementation issues

As mentioned earlier, capabilities represent the validity of actions attempted by agents on objects, without which the action will not be allowed to take place. In LINDA, there are three possibilities if the agent does not possess a valid right for an action: the action either blocks, throws an exception, or fails. We now discuss the benefits and problems of each choice. For the purpose of the discussion, we assume a system where it is not possible to statically check the capabilities due to the openness of the tuple-space paradigm.

*Blocked* input/output operations affect the system differently. As discussed in [29], superficially, the agent cannot detect whether an input operation is blocked because of either (1) the unavailability of a matching tuple, or (2) the permission being denied due to an invalid capability. Thus, from the user's point of view, there is no difference in the behaviour of the system. A *blocked output* operation, however, would be noticeable as a fundamental alteration in the semantics of `out` has been forced—`out` never blocks in the standard LINDA model. In both input and output operations, if the operation blocks due to an invalid capability—which essentially means that it "blocks until the agent possesses the required capability", and it is obvious that it cannot 'possess' the capability simply by waiting there—it will block forever. In order to obtain the capability, the operation must first return to the agent, who will send a request (and wait) for the appropriate capability before retrying the operation.

If the implementation has an *exception* handling mechanism, then an exception is raised and the agent can resume its execution. Unfortunately, the implication of this is that the orthogonality of the coordination and computational language is broken [6]. Nevertheless, it would be sensible and practical to implement the system in such a way for the operation to simply throw an exception, to enable the agent to continue—possibly proceeding to obtain the necessary capability, or perhaps to look for a tuple (matching the template it requires for its execution) elsewhere. This is where the combination expressions can become useful: an agent (a lazy one at that) can simply present all capabilities it holds, and let the kernel choose which capability to use to suit the agent's attempted action—one that has the permission to perform the action on the specified template.

However, all these possibilities result in semantics for `out` which differ from classical LINDA.

## 4. Application examples

This section discusses three examples of applications in open distributed systems which now become possible, or are improved, using multicapabilities: extending the garbage collection mechanism to handle unusable tuples, providing a finer control for deadlock breaking mechanism, and replication of data. Garbage collecting tuples has not been possible before in the tuple-space model: only tuple-spaces can be garbage collected using Ligia [16]. As for deadlock detection and data replication, multicapabilities provide means to improve these applications.

### 4.1. Garbage collection on tuples

Resource management is vital in distributed systems that involve ubiquitous and persistent computing. One resource that needs to be managed is memory, which is limited and can be reclaimed through garbage collection. Garbage collection (Ligia [16]) has already been proposed for standard LINDA with multiple tuple-spaces to avoid memory exhaustion. The implementation, however, was restricted to garbage collection of tuple-spaces, but not tuples: the main problem in introducing garbage collection for tuples is the lack of sufficient information about their 'usage'. This information can be maintained if we can reference a particular tuple, or a group of tuples—something that is not possible in LINDA. While tuple-spaces have unique identities, tuples (and templates) do not: they are referred to by values instead of names. Thus it is difficult to employ garbage collection on tuples without modifying the model: giving unique names to tuples will certainly break one of the fundamental characteristics of LINDA: associative retrieval. However, this can be avoided with multicapabilities as the kernel can now reference a collection of *nameless* tuples to perform garbage collection on them, by garbage collecting the multicapability regions themselves.

In Menezes's Ligia, the universal tuple-space (including its contents) can never be garbage collected, therefore any tuple put into it will persist in the system forever until *explicitly* removed, or the system terminates [16]. The number of tuples in the universal tuple-space may grow, thus consuming valuable memory space. With multicapabilities it becomes possible to garbage collect some of these tuples as we can specify region(s) in the universal tuple-space to be garbage collected, without having to remove the whole universal tuple-space—thus providing a finer control over the system.

A simple and naive strategy for implementing garbage collection is to run garbage collection when and every time the reference to an object has been decreased to zero/nil; and one reason for a reference to be deleted is when the agent holding the reference dies. However, it is not practical to run a garbage collection every time an agent dies, as an essential part of the tuple-space communication is temporal separation—data produced by an agent can be consumed by another, long after the agent dies. We also know that performing garbage collection can be an expensive operation (in terms of kernel load). Therefore, it is more efficient to garbage collect only when needed, i.e. when there is insufficient memory space available. Even though this strategy involves a larger amount of work to be carried out at one time compared to the former, garbage collection is likely to be performed less often.

Experiments have been carried out to demonstrate our claim. These experiments compared two capability systems for memory exhaustion: one incorporates the garbage collection mechanism for *tuples*, whereas the other does not. The characteristics for the systems are:

(1) Both systems have garbage collection for *tuple-spaces*, based on Ligia.
(2) All interactions are via the universal tuple-space, which is not being garbage collected—the TS garbage collection mechanism [16] cannot be performed on the universal tuple-space. Therefore, we can be certain that these experiments only concern garbage collection on tuples, and not on tuple-spaces.
(3) Each agent requests their own multicapability with no capabilities being passed among the agents, which implies that all the agents used different multicapability regions. Thus, these regions cannot be referenced by any other agents, and are considered garbage when the agents creating them die.

The experiments involved running a group of agents in limited memory space, where each agent requests a new multicapability, `out`s a number of tuples into the universal tuple-space using the newly acquired multicapability, and then dies after explicitly deleting the multicapability. The agents' code snippet is given below.

```
cap = newcap( <int,int> );
for (t = 0; t < 10000; t++)
   UTS.out( cap<1,t> );
del cap;
```

As expected, the server with no tuple-garbage collection eventually ran out of memory, whereas the server with garbage collection did not encounter the same problem, in fact it was able run indefinitely.

For the purpose of these experiments, we used the following strategy: perform garbage collection every time a reference to a multicapability region is removed (i.e. when a multicapability is deleted or revoked), or when an agent dies, in which case all the capabilities (representing references) it holds is removed.

Although this is, again, a costly strategy, it demonstrates how these additions can improve tuple-space based coordination. Even though the garbage collection mechanism has not been fully implemented, we have shown that it works successfully—with capabilities, we can now garbage collect groups of tuples within tuple-spaces, without garbage collecting the whole tuple-space. A better, more complex, strategy is presented in the work of Menezes [16].

#### *4.1.1. Keeping track of capabilities*

We know that for garbage collection the kernel must maintain some kind of information regarding the references—which agent knows about which object. There are three ways for an agent to acquire a capability (either for a tuple-space or a group of tuples), and how the kernel may keep the information it needs:

(1) The agent creates a tuple-space (therefore obtaining the TS-capability in return), or requests a new multicapability for some template of tuples. The kernel can simply record the agent's identity against the newly created capability.
(2) The agent has retrieved a tuple containing a capability for the object, as discussed in Section 3.3. To obtain this information, it is necessary for the kernel to implement tuple monitoring [15]—which we have also extended to monitor tuples containing multicapabilities—which enables the kernel to keep track of capabilities being passed as tuple elements.
(3) The agent has been spawned by a parent agent, and the capability has been passed from the parent. As discussed in [12], this is a rather complicated case, and the solution relies on *process registration* and 'deregistration' [15], which have been incorporated in our implementation—all newly spawned agents must register themselves and the capabilities they hold, and must 'unregister' with the kernel before terminating.

Termination ordering [16] should also be observed to avoid race conditions. Agent termination is an operation that can generate garbage, as capabilities may be deleted, which would result in the loss of references to some objects.

### *4.2. Deadlock breaking*

Managing deadlocks is also a part of resource management. A deadlock is a permanent blocking of a set of agents competing for the system's resources, such as processors and memory space, as part of their communication with each other. A deadlock breaking mechanism was proposed in earlier work related to `inp` [12]. The idea is that when the kernel detects that a group of agents are deadlocked, then it can return 'false' to one or more of the agent's `inp`s, since the existence of deadlock 'proves' that the `inp` can never return a tuple.

A multicapability is a triple consisting of a template, a set of rights and a unique tag. We have demonstrated that the reference part is useful for garbage collecting tuples of that template. The permission part of a multicapability can be used to refine deadlock detection. For instance, if an agent has a multicapability to perform an input operation on a tuple which does not (yet) exist, the operation would consequently block. If the kernel knows that the agent is the only one having the multicapability; and no other copies of the multicapability lying around in the system for some agent to retrieve it later; and no other agent in the system holds the right (multicapability) to `out` such a tuple (which means that the operation will block indefinitely), the kernel can immediately employ the deadlock breaking mechanism.

Based on the deadlock breaking mechanism using `inp` discussed in [12], we have also implemented the deadlock breaking on multicapability groups, and therefore demonstrated that multicapabilities provide the means to perform deadlock breakings to a more fine-grained level: that is, to break deadlocks within a group of tuples of the same pattern.

Deadlocks do not normally 'announce' themselves: the system needs to be reactive and perform deadlock detection routines from time to time, sensibly at times when deadlocks are likely to occur. The common ways (though not necessarily a good design) to check for deadlocks are when either of these situations arises:

- every time an operation blocks (when no matching tuple is available), or
- every time a TS-capability or a multicapability is removed.

In Lindacap, a deadlock occurs when the following conditions hold:

(1) When a group of agents are blocked on a multicapability region, on a tuple-space, and
(2) There is no tuple containing the capability anywhere in the tuple-space, and there is no other agent outside the blocked clique (i.e. the group of agents) can `out` the tuple, or

(3) There is one such tuple containing the multicapability in the tuple-space, but no other agent outside the blocked clique (i.e. the group of agents) can get the tuple, or
(4) There is one such tuple (containing the multicapability) but it is in a different tuple-space which no agent outside the clique has the (tuple-space) capability for.

The universal tuple-space and the region for the primordial capability, *cc* cannot be deadlocked, as all agents have the capabilities/references to them. When a deadlock is detected, the kernel would go through the list of blocked agents, and, as proposed in [12], any predicated operation (`inp`/`rdp`) in the list is randomly unblocked.

Checking for deadlocks every time an operation blocks is the least efficient strategy and can be costly. A better strategy is to detect one when it occurs, or better still, to be able to predict one. Capabilities provide the information needed to achieve this. For garbage collection, the kernel need only maintain information on which agent knows about which object; whereas for deadlock detection purposes, it also need to know which agent has *what rights* for which object. This information can be obtained as described in Section 4.1.1.

Consider two agents possessing references to a template, and only one of them has the permission to `out` a tuple. Suppose that this agent with `out` permission is blocked on an input operation, while the other (without `out` permission) is still running. No deadlock transpires at this point—and in the non-capability system, the kernel has no way of predicting one—until the running agent performs an input operation and blocks, too. With capabilities, the kernel would know in advance that there is an impending deadlock—as the running agent can only perform input operations—and can proceed to unblock the blocked agent without having to wait until the deadlock actually occurs.

Without capabilities, there is no way of knowing in advance what operation an agent might perform. Therefore, any deadlock can only be detected when one occurs. With capabilities, in some circumstances we have a better strategy: detect a deadlock *before* it happens. The kernel could know that an agent cannot perform an operation. Consequently a deadlock could be *predicted* and broken as soon as the `inp` (or `rdp`) is executed. Moreover, without capabilities, the kernel can break the deadlock by choosing a process doing an `inp` to be unblocked; whereas with capabilities, if none of the blocked processes is blocked on an `inp`, there is another choice: to unblock a process which has an `out` permission.

### 4.2.1. *Example: Stable marriages*

The goal of the stable marriages problem [13] in its simplest form, is to pair up *n* men and *n* women such that all marriages are stable. Each person rates their prospective partners in strict order of preference. A marriage is stable if all the women that a man ranks higher than his wife prefer their husbands to him, and all the men that a woman ranks higher than her husband prefer their wives to her. In the simplest LINDA implementation of the algorithm, a 'stable' marriage assignment is reached when a deadlock occurs. A much simpler implementation of the algorithm has demonstrated in [12] using the deadlock-breaking mechanism (using `inp`), with the participants, rather than an additional 'broker' agent, distributively decide when a stable assignment has been reached.

The solution in [12] was implemented in a non-capability system. Adapting the algorithm, we have implemented the solution in the capability-based system. To enable data sharing among the participants, the capabilities for the working tuple-space and the multicapability for the relevant data need to be passed among them. In Lindacap, these references need only to be written once, as the tuple containing the capability for the tuple-space will not be in any danger of being removed by any agent—the primordial capability *cc* does not allow removal.

We have also run two completely separate programs—each solving a separate instance of the stable marriages problem—within the same tuple-space on the server, and they do not interfere with one another. Assuming the working tuple-space, `wts`, has been created, and the capability for it has been written into the universal tuple-space (UTS) using *cc*:

```
wts = TupleSpaceC();
UTS.out( cc<wts> );
```

Each program begins by requesting their own multicapability (region) to work in, and writes the multicapability into the UTS:

```
c1 = newcap( <?str,?str,?str> );
UTS.out( cc<c1> );
```

All participants obtain the TS-capability and the multicapability they would be working in:

```
wts = UTS.rd( cc<?ts> )[0];
c1 = UTS.rd( cc<?cap> )[0];
```

The algorithms for agents modelling ‘men’ and ‘women’ are sketched below:
$men, m_i$:

```
fiancee = FirstChoice();
while(true){
    wts.out( c1<'propose', myname, fiancee> );
    if !(wts.inp( c1<'reject', myname, ?fiancee> ))
        break;
    fiancee = NextChoice();
}
```

$women, w_i$:

```
fiance = null;
while(true) {
    if !(wts.inp( c1<'propose', ?suitor, myname> ))
        break;
    fiance = BestOf(fiance, suitor);
    reject = WorstOf(fiance, suitor);
    wts.out( c1<'reject', reject, myname> );
}
```

Since the list of participants, names are exactly the same for both programs, a man $m_i$ in both programs would be looking for the same tuple, i.e. $\langle$*‘reject′*, $m_i$, *?fiancee*$\rangle$, whereas a woman $w_i$ would be waiting for $\langle$*‘propose′*, $w_i$, *?suitor*$\rangle$. As each program uses different multicapability regions, the tuples and queries in both programs are independent of each other, as their capabilities do not match. They are isolated programs, effectively partitioned into different multicapability regions. We can even have several stable marriages programs running separately within the same tuple-space, oblivious to each other’s existence.

## *4.3. Replication and caching*

A final example of an application that is aided by information derived from multicapabilities is replication and caching: if the kernel knows all (or at least a large number of) agents have the right to only read a tuple, it can replicate the tuple and put the copies into different partitions of the distributed tuple-space to be accessed by the agents. In a distributed system, a tuple-space can be distributed on a number of processors. Replicating data into these physically distributed partitions of the tuple-space could reduce the amount of communication, for example, involved in reading the data, as compared to reading it from its original location which could be on a remote machine. Naturally, these replicas will have to be removed if and when any copy of the tuple is `in`’ed later. Based on the information provided by multicapabilities, the kernel can decide whether to replicate the tuple after weighing the benefits of caching the data against the task of removing the copies later.

Consider a publisher/subscriber example: the scenario is that one agent publishes material to be accessed by many subscribers. Knowing that the subscribers can only read the material, the kernel can replicate the material to be accessed by the subscribers. When the publisher decides to remove (or update) the material, all the (old) copies must be removed. The issue here is “when”—one might argue that the removal is temporally incorrect—but the question of “when” is ‘flexible’ in LINDA since there is no notion of global time in the model.

The system can be implemented in such a way that when an `in` operation is performed on the data, a ‘tidying up’ message is sent to the kernel, indicating that the copies should be removed. This removal is not necessarily immediate: nevertheless, to preserve the logical state of the system, the copies should be removed before the next `out` operation of the publisher, for example, or before the effect of the next operation is propagated to other parts of the tuple-space.

Otherwise, an agent executing the next `in`/`rd` operation may accidentally get (a copy of) the supposedly removed tuple.

## 5. Related work

Although LINDA has become a well known coordination model, as an alternative to the conventional communication paradigms, it is sometimes considered as 'too open and too flexible', leaving it vulnerable to accidental, or even malicious, manipulation. Much work has been carried out to impose more control, at the expense of its most attractive feature—flexibility. Capabilities, on the other hand, provide the mechanism to finely control a system without losing the flexibility essential in an open environment. Examples of early attempts at improving LINDA using a capability-like mechanism is Pinakis's Joyce-LINDA [21], which used public-key encryption to support capabilities, and Law-Governed LINDA (LGL) [17]. Unfortunately, the capability values in Joyce-LINDA cannot be matched by formals in templates, thus preventing them being passed via tuple-space. Introducing an additional special data type to enable the matching process only complicates things. The notion of capability-based control in LGL is somewhat restricted: capabilities are required in order to send a message to another agent, but none is necessary to receive one. This means that a message can only be sent if the sender has the capability for the target agent. Therefore, there is a possibility of indefinite waiting, since these agents might not be aware of the sender's need to acquire the necessary capability.

SECOS [28,3] generalizes the tuple-space model so that tuples can play the role of capabilities. It is flexible enough to do (almost) everything that Lindacap can do, but there are distinct differences, and neither subsumes the other. As in Lindacap, SECOS provides a facility to define 'views', using a two-level locking scheme: objects (tuples) are locked with a key, and each field must be locked with a different key. Like our multicapabilities, their object locks provide a partitioning facility for the tuple-space. Unfortunately, the problem with using a key-related control is that it is not possible to discriminate the rights for certain operations: if an agent possesses a key to a tuple, there is no way of restricting the permission, e.g. for a read-only, but not remove, operation. Indeed, their can-match-anything 'empty' template, which can be useful for garbage collection, for example, may be exploited by any agent to remove any tuple. Although SECOS does allow restriction of a capability, it does not support the other capability operations of Lindacap. Unlike the decentralized capability distribution in our model where any agent may distribute any capability in its possession, the distribution of keys in SECOS is the responsibility of the tuple's originator, which, though more controlled, can become an onerous task, especially in a large open system. Another system that uses a capability-based security policy with keys is Lana [4]. The messages (i.e. tuples) on their (tuple-space based) associative 'message boards' are locked with keys, not capabilities—capabilities are used for remote method invocations. Locked with a key, the reference to (the capability for) the object being called is placed on the message board.

An interesting example of a more recent work is VLOS [7], a distributed operating system based on tuple-spaces. Capabilities are required to create new tuple-spaces, new field types, as well as type signatures, i.e. unique groupings of field types which are unique to a particular tuple-space, to make up a tuple type (i.e. template). In VLOS, a tuple-space is created with nothing associated with it, except some basic types, and capabilities. Whenever an agent wishes to perform a standard input/output operation, such as `out`ing a tuple, the system requires it to register the tuple type before the operation can be carried out. A capability for an object in VLOS consists of a unique identifier and the type of the object (i.e. whether it is a tuple-space, a field type, a type signature, or other objects), the name of the tuple-space (of which the object is a member), a set of rights, and a cryptographic hash function to minimise forgery. Like ours, a capability in VLOS acts as a 'handle' to the object it is associated with. Our multicapabilities, however, yield more flexibility in the sense that they are not associated with a tuple-space—they can be used with any tuple-space as long as the user possesses the capability for the target tuple-space. Furthermore, the combination calculus of capabilities gives an added advantage for our model to maintain control while being flexible.

A $\mu$KLAIM [11] capability is a pair that represents the operation allowed on a pattern of the matching (target) tuple, whereas our multicapability is a triple that provides extra features of partitioning via tags, to limit and control agents' accesses, as well as the combination calculus to further enrich the model. Based on the KLAIM language [18], $\mu$KLAIM uses its type system to enforce access control. Unlike our capability model that uses dynamic checking, $\mu$KLAIM relies on both static and dynamic checking.

In the systems mentioned above, capabilities are mainly discussed as an access control mechanism for security purposes, some with the assistance of cryptography, as in [21,28,4]. $\mu$KLAIM for instance, emphasises security policies: an agent may have 'knowledge' of a location name even though it does not have a capability to it. We are concentrating on the functional properties of capabilities as 'visibility' filters—agents can only know about objects for which they hold a capability, and the capability makes visible a sub-set of the operations available for that object's type. This scheme enables a refined control over agents' actions (not limited to access control only) on objects in the system, and facilitates certain aspects of coordination, such as those mentioned in Section 4.

## 6. Conclusions and future work

Capabilities represent the information on 'who knows about what operation on a certain object'. The more the kernel knows of the system's potential behaviour, the better, more optimised coordination can be achieved, thus increasing the system's efficiency. The extra information, supplied by the capabilities given to the agents, can provide the facility to create a finer level of control in distributed systems.

As capabilities can only be applied to uniquely identifiable objects, such as tuple-spaces, we have proposed the new concept of multicapabilities which extends capabilities to apply to a group of un-named objects to accommodate tuples. A multicapability consists of a unique tag to differentiate between different capabilities for the same template (and which in addition aids in its unforgeability), a reference to a group of tuples, and a set of rights to control the actions permitted to be performed on an object in the said group. The set of rights need not be limited to input and output operations, but may include any sensible, even user-defined, operations that are appropriate to the system. Some capability combination operations have been introduced—namely the set-like union, intersection, and difference operations—to provide further mechanisms towards achieving a finely controlled system. We are still investigating the sensible 'combining' operations, and currently developing proper semantics.

It is known that one of the disadvantages of capabilities is that they are difficult to revoke. VLOS [7] provides single-use capabilities, which are valid to be used only once, to overcome this problem. An alternative solution is to incorporate indirect capability objects: a capability held by an agent does not directly point to an object, but instead refers to an indirection object, which in turn points to the object (as used in EROS [25]). Deleting an indirection object enables a capability to be permanently revoked. As an option to deletion, we are studying the idea of selective temporary revocation using indirection objects in which the corresponding sub-interfaces can be turned off and on to provide a finer control in the system. When an indirection object is turned off, any access (including `out`) to the object will block until it is switched back on.

Indirection objects can become a filter for a group of multicapabilities and their derivations: a derived multicapability with identical or restricted rights will point to the same indirection object as its super-multicapability; whereas creating a new multicapability (even of the same template) will automatically create a different indirection object. Deleting the former will revoke the said multicapability along with its sub-multicapabilities.

There are a number of ways in which the current work can be extended. Remaining in the coordination domain, we are considering how multicapabilities would fit into frameworks such as Gamma [2] or Reo [1]. More generally, we will pursue the view of a (multi)capability as a sub-interface to re-evaluate the use of capabilities in mainstream object-oriented programming.

On a final note, we view a capability as not merely access control, but in more general terms as *visibility* control. Visibility can represent security. Although this paper does not address the issue of security, it is indeed a crucial problem when dealing with agents with intelligence and autonomy, particularly those involved in some sort of confidential and sensitive business transactions or other critical applications.

## Acknowledgements

The authors are grateful to the anonymous reviewers for the helpful comments on the previous version of this paper. Nur Izura Udzir is funded by the Public Services Department of Malaysia.

## References

[1] F. Arbab, Reo: A channel-based coordination model for component composition, Mathematical Structures in Computer Science 14 (3) (2004) 329–366.

[2] J.-P. Banatre, D. Le Metayer, Programming by multiset transformation, Communications of the ACM 36 (1) (1993) 98–111.

[3] C. Bryce, M. Oriol, J. Vitek, A coordination model for agents based on secure spaces, in: Proc. 3rd International Conference on Coordination Models and Languages, Coordination'99, in: LNCS, vol. 1594, Springer-Verlag, Berlin, Heidelberg, 1999, pp. 4–20.
[4] C. Bryce, C. Razafimahefa, M. Pawlak, Lana: An approach to programming autonomous systems, in: ECOOP 2002, in: LNCS, vol. 2374, Springer-Verlag, Berlin, Heidelberg, 2002, pp. 281–308.
[5] N. Carriero, D. Gelernter, How to write a parallel program: A guide to the perplexed, ACM Computing Surveys 21 (3) (1989) 323–357.
[6] N. Carriero, D. Gelernter, Coordination languages and their significance, Communication of the ACM 35 (2) (1992) 97–107.
[7] V.-L. Chung, C.S. McDonald, The development of a distributed capability system for VLOS, Australian Computer Science Communications 24 (3) (2002) 57–64.
[8] J.B. Dennis, E.C. van Horn, Programming semantics for multiprogrammed computations, Communication of the ACM 9 (3) (1966) 143–154.
[9] E. Freeman, S. Hupfer, K. Arnold, JavaSpaces: Principles, Patterns, and Practice, in: The Jini Technology Series, Addison-Wesley, 1999.
[10] D. Gelernter, Generative communication in Linda, ACM Transactions on Programming Languages and Systems 7 (1) (1985) 80–112.
[11] D. Gorla, R. Pugliese, Enforcing security policies via types, in: Proc. 1st Int. Conf. on Security in Pervasive Computing, SPC'03, in: LNCS, vol. 2802, Springer-Verlag, Berlin, Heidelberg, 2003, pp. 88–103.
[12] J.L. Jacob, A. Wood, A principled semantics for `inp`, in: Coordination Models and Languages, in: LNCS, vol. 1906, Springer-Verlag, Berlin, Heidelberg, 2000, pp. 51–66.
[13] D.E. Knuth, Stable Marriage and Its Relation to Other Combinatorial Problems: An Introduction to the Mathematical Analysis of Algorithms, American Mathematical Society, 1997.
[14] H.M. Levy, Capability-Based Computer Systems, Digital Press, 1984.
[15] R. Menezes, A. Wood, Using tuple monitoring and process registration on the implementation of garbage collection in open linda-like systems, in: Proc. 10th IASTED International Conference on Parallel and Distributed Computing and Systems, ACTA Press, 1998, pp. 490–495.
[16] R. Menezes, Resource management in open tuple space systems, Ph.D. Thesis, Uni. of York, UK, 2000.
[17] N.H. Minsky, J. Leichter, Law governed Linda as a coordination model, in: Object-Based Models and Languages for Concurrent Systems, in: LNCS, vol. 924, Springer-Verlag, Berlin, Heidelberg, 1995, pp. 125–146.
[18] R. de Nicola, G.L. Ferrari, R. Pugliese, KLAIM: A kernel language for agents interaction and mobility, IEEE Transactions on Software Engineering 24 (5) (1998) 315–330.
[19] G. Nutt, Operating System: A Modern Perspective (2nd), Addison-Wesley, 2002.
[20] J.L. Peterson, A. Silberschatz, Operating System Concepts, 2nd edition, Addison-Wesley, 1985.
[21] J. Pinakis, Providing Directed Communication in LINDA, in: Proc. 15th Australian Computer Science Conference, 1995, pp. 731–743.
[22] A. Rowstron, A run-time systems for coordination, in: Coordination of Internet Agents: Models, Technologies, and Applications, Springer-Verlag, 2001, pp. 61–82.
[23] A. Rowstron, A. Wood, BONITA: A set of tuple space primitives for distributed coordination, in: Proc. 30th Hawaii International Conference on System Sciences HICSS-30, vol. 1, IEEE Computer Society Press, 1997, pp. 379–388.
[24] J.S. Shapiro, EROS: A capability system, Ph.D. Thesis, Uni. of Pennsylvania, USA, 1999.
[25] J.S. Shapiro, J.M. Smith, D.J. Farber, EROS: A fast capability system, in: Symposium on Operating Systems Principles, 1999, pp. 170–185.
[26] A.S. Tanenbaum, S.J. Mullender, R. van-Renesse, Using sparse capabilities in a distributed operating system, in: Proc. 6th International Conference on Distributed Computing Systems, ICDCS, IEEE Computer Society, 1986, pp. 558–563.
[27] N.I. Udzir, A. Wood, Establishing private communications in open systems using multicapabilities, in: Proc. 2nd IEEE International Conference on Information and Communication Technologies: From Theory to Applications, ICTTA'06, 2006.
[28] J. Vitek, C. Bryce, M. Oriol, Coordinating processes with secure spaces, Science of Computer Programming 46 (1–2) (1999) 163–193.
[29] A. Wood, Coordination with attributes, in: Coordination Languages and Models, in: LNCS, vol. 1594, Springer-Verlag, Berlin, Heidelberg, 1999, pp. 21–36.