Coordination using Object Attributes

Alan Wood
Department of Computer Science
University of York
wood@cs.york.ac.uk

Abstract

The development of the idea of *coordination languages*, as defined by Carriero and Gelernter [8], has enabled work in concurrency to abstract away from the internals of processes or agents and hence to focus on their external interactions. A variety of *practical* parallel and/or distributed systems have resulted from the application of the coordination language concept — the most widely-known of these being LINDA [7] and its variants.

All coordination systems involve a number of logically shared objects, which may themselves be structures containing further objects. In the context of LINDA, these objects are: *tuple spaces*, which are bags of *tuples* each consisting of ordered sequences of atomic *elements* (in some variants, elements can be non-atomic objects). Tuple spaces are shared between concurrent processes which then coordinate (asynchronously) by depositing tuples in, or associatively retrieving tuples from the tuple spaces. However, in the 'classical' LINDA model, there are no access properties or similar attributes associated with the objects (tuple spaces, tuples, and tuple elements).

This paper addresses the opportunities for, and effects of, incorporating object attributes — primarily those concerned with access-control, but including examples of some less familiar attributes — in LINDA-like tuple-space systems. The focus is on exploring how the coordination properties are affected by this enhancement, and investigating in what ways the classical LINDA model needs to be modified. Particular emphasis is placed on the potential *practical* gains to be expected in both performance, and expressibility, with consideration being given to methods for implementing attribute frameworks in open distributed systems.

1 Introduction

The concept of coordination could not be claimed to be 'new' — even in the narrow domain of computing, and its sub-domain of concurrency, the need to coordinate things has been around a long time. However, it is only relatively recently that coordination has been identified as a topic for independent study, and which promises to unify several apparently disparate fields in computing.

In their 1992 paper [8], Carriero and Gelernter crystalised this idea by identifying the *two* functions of a "full" programming language as *computation* and *coordination*, and suggesting that these two aspects of a language are *orthogonal*. Consequently, the study of coordination may be undertaken *independently of* any host computation technology.

Once the idea of coordination being distinct from computation is accepted, the question arises as to what the coordination *structures* are. These would be analogous to the control and data structures in the *computation* half of a language. Some coordination structures have been in common use for many years: semaphores, monitors, signals (in the UNIX sense). Others have been emerging from the increased *use* of parallel and distributed architectures ... examples might be light-weight threads, message-passing, or barrier-synchronisation.

In recent years, the focus in system design has been changing from single, monolithic programs, to concurrently executing, physically distributed ensembles of processes. This change in focus has inevitably led to the need for new conceptual tools for controlling the complexity of such designs, and the coordination abstraction is proving to be of central importance in this area. Consequently, coordination is intimately related to the move to parallel and distributed systems.

This paper addresses coordination problems in *open distributed systems*, which are to contrasted with *closed* systems. The distinction lies in the fact that in an open system agents (processes) can 'join' or 'leave' during the lifetime of the system with no prior knowledge of their (potential) existence necessarily being available to the other agents. From the point of view of a programmer, or more importantly a *compiler*, full information about the behaviour of the system while his/her/its process is executing cannot be determined. In a closed system, however, such information is in principle available, and can be used by a compiler, say, to optimise the process ensemble. The open system model is that of the internet; the closed system is that of a subset of processors in a box, with only a single parallel program running.

Finally, the term 'object' is used throughout the paper with its generic meaning — it is not restricted to objects in an object-oriented system (although there will be occasions when it could take this meaning.

1.1 Object Attributes

Objects have a number of properties associated with them — indeed, it is the set of properties that *define* an object. In a computational context, perhaps the most fundamental properties are *value* and *type*. However, there are other properties which are normally associated with objects including their *scope* (or visibility), updatability, or other accessibility features.

The names of these properties will be referred to as *attributes* — each attribute will name the value of some property. Although there are some obvious concrete examples of attributes (as mentioned above), they should be seen as a set of abstract properties associated with objects. This view admits the possibility of having attributes which may be user-defined (and, consequently, would need to be user-maintained), rather than fixed by the system.

1.2 LINDA

Although the need for coordination has been around a long time, it is only recently that *languages* specifically directed at expressing coordination structures have emerged. Since this aspect of the field of coordination technology is still in its infancy, there is considerable diversity in the features of those languages (whether executable or formal), as is demonstrated by looking at the characteristics of just those few languages mentioned in section 5.1.

Since this paper addresses some *general* aspects of coordination, while suggesting practical ways of implementing them, trying to relate the ideas to the diversity of current coordination languages would almost certainly obscure the issues. Therefore, only one coordination system will be used as a focus for the ideas presented — LINDA. There are two main reasons for chosing LINDA. Firstly, it is (probably) the earliest explicitly 'coordinative' language, and (certainly) one of the best-known of the current coordination languages. Secondly, it is a very flexible model which subsumes most others (although, as shall be seen later, it still has some deficiences in its coordinative features).

However, many of the issues raised here are not LINDA-specific — indeed, much of the point of the paper is in identifying 'purely' coordinative issues which exist independently of any specific coordination model.

The classical LINDA model [7] consists of a number of logically shared data repositories called *tuple-spaces*, which contain *tuples* — ordered sequences of typed values. Together with tuple-spaces, the model defines four principal access operations:

`out( ts, tuple )` which deposits `tuple` into tuple-space `ts`.

`in( ts, template )` which searches `ts` for a tuple that matches `template` (see later). When a match is found, the tuple is *removed* from `ts` and returned to the calling process. The operation *blocks* until a suitable tuple is returned.

`rd( ts, template )` acts in the same way as `in` except that a *copy* of the matching tuple is returned leaving the original in `ts`.

`eval( ts, active-tuple )` An `active-tuple` is one whose elements are expressions or function calls. The action of `eval` is to spawn a process to evaluate each element of the active-tuple concurrently. When each element has completed evaluation, the results returned from the processes ('ground' values) are formed into a normal, 'passive', tuple and placed in tuple-space.

Although it is possible to view the active tuple as 'existing' in `ts` while its elements are being evaluated, in the classical LINDA model it cannot be matched by any template, and hence is 'invisible' until it becomes passive.

In the following, tuples (and templates) will be shown delimited by angle brackets — for example, ⟨ `1, "hello", 10.3` ⟩ is a three-element tuple consisting of an integer, a string, and a real number respectively.

Templates are tuple "patterns". Their elements can be typed ground values in the same way as tuples (called *actuals*), but may also be typed 'holes', or *formals*. A template matches a tuple if it is the same length, and all its elements match. Two elements match if they are *actuals* with identical types and values, or an actual and a formal of the same type.

In the examples shown later, tuples are themselves values in the host computational language, and may be indexed in order to select particular elements.

1.3 Motivation

In LINDA-like systems, agents (processes) coordinate by means of *generative communication* [10]. This term is intended to suggest that the 'carrier particles' of coordination are created by agents, but then have existence independent of their creators.[1] Using this medium, LINDA is able to provide (nearly) all the coordination mechanisms normally used in parallel processing situations. There are other ways of achieving coordination which do not fall neatly into the category of data communications, as will be seen later. However, why should they be needed? There are two reasons:

[1]This is in contrast to conventional synchronous message-passing communication models in which the message only has a fleeting existence at the instant of synchronisation. It therefore cannot exist independently of the sender and receiver.

1. There are coordination tasks which are *not* supported by the model.

2. There are also some coordination tasks arising quite naturally which, although supported in the 'classical' LINDA framework, give rise to significant inefficiencies[2] under the classical model. This in itself is not a sufficient reason to support any proposed modifications to the LINDA model, since any change introduces extra complexity which must be weighed carefully against the potential benefits.

Examples of both of these types of task will be described in section 2.1.

2 The Design Space

The objects that we shall be concentrating on are *tuple-spaces*, and *tuples*, although it is possible also to consider tuple *elements* as being candidates for attribute-based coordination. There is a wide variety of possible object properties that could be incorporated in an attribute-based coordination scheme, and some of the more unusual candidates will be discussed in section 5.2. However, in this paper we shall focus on the widely-understood access attributes, which are analogous to those found in conventional file-systems.

The choice of objects, together with the set of attributes gives rise to a 'design space' formed from the object and attribute 'axes'. This space then enables the issues of semantics, implementability, expressibility etc. to be addressed when assessing which modifications to the LINDA model are required.

2.1 Access-control Attributes in LINDA

The object axis of the design space is clearly defined above — tuples, tuple-spaces, and tuple elements.[3] The attributes will include the right to perform the appropriate LINDA access operations on the objects — `in`, `rd`, and `out` — which approximately correspond to read and write permissions on files in conventional systems.[4] However there are other operations which are performed on objects in a LINDA-like system which must also be considered.

Figure 1 displays the part of the design space determined by these objects and attributes. A tick (✓) implies that a sensible meaning can be assigned to that attribute:object pair; a bullet (•) suggests that that pairing is incompatible; a question mark (?) indicates those pairings for which it is not clear that there is a sensible use, but might prove useful in some situations.

The attributes can be placed in the following groups based on their effects:

out, in, rd It is easy to see that out, rd and in attributes applied to tuple-spaces make sense — if a tuple-space has that attribute then an agent can access it accordingly, otherwise that type of access is prevented.

[2]Here, 'inefficiency' is to be taken in a broad sense, which will include run-time performance, but will also encompass *programming* or *software engineering* inefficiencies. An example of the latter would be the need for tuples output by an agent to have additional information 'just in case' some other agent *might* require this extra information.

[3]One further object could be identified: *templates*. However, 'standard' LINDA does not allow these to be placed in tuple-spaces and therefore they do not exist as coordination objects. It is, of course, possible that some LINDA derivation could make templates 'first-class', and then they could appear in the design space for that system.

[4]This correspondence is not exact since there is now a distinction between copying content from an object, and *removing* content., which has no direct analogue for file systems.

	Tuple-Space	Tuple	Tuple Element
in	✓	✓	•
out	✓	•	•
rd	✓	✓	•
wildcard	?	✓	✓
match	?	✓	✓
eval	✓	?	?

Figure 1: The Design Space

With individual *tuples*, only in and rd can apply — corresponding to permission to remove or copy that tuple, respectively — since `out`ing to a tuple is not defined.[5] Similarly, neither in nor rd nor out can be applied to tuple elements.

wildcard and match

The lack of wildcard permission on a tuple element would prevent an `in` or `rd` operation form retrieving that tuple is the operation's template specified a *formal* for that element. By extension, this would be applicable to whole tuples and tuple-spaces, affecting actual:formal matching on all elements of a tuple, and all tuple, respectively.

The match attribute is the dual of wildcard: its lack would imply that *only* actual:formal matches would succeed, and actual:actual matching would not.

eval

Permission to place an active tuple into a tuple-space is controlled by the eval attribute. This is clearly meaningful when applied to tuple-spaces, although it can also be of use when applied to individual tuples, as will be discussed later.

Denying eval rights in a tuple space would enable a finer control over the behaviour of the physical implementation of an open distributed tuple-space system.

There are a number of ways in which a LINDA system can be implemented on a collection of processors. Consider the simplest in which each processor is responsible for maintaining complete tuple-spaces (that is, tuple-spaces are not themselves distributed over multiple processors). The processors might have widely differing computational capabilities, ranging from super-computers to hand-held PDAs — a mobile phone might wish to make information available to, and communicate with, other processors using its local tuple-spaces, but it would probably not want to discover its CPU cycles being used by a distributed version of the Human Genome Project. It would however, want to `eval` its own processes locally — the eval attribute would control this behaviour.

The utility of eval permission on *tuples* depends on whether an element of an active tuple can be matched in a template.[6] If so, then the eval attribute could be used as a means of placing an active tuple in tuple space, but preventing its evaluation — the

[5] If there were some sort of `update` operator, then this attribute could be applied to a tuple — it would prevent or allow updating.

[6] This is not possible in the standard LINDA model, but a variation on the model could incorporate this relatively easily.

tuple's elements would then be 'suspensions' (closures or continuations) which, if `in`ed by a process and subsequently `out`ed, could be evaluated later.[7] The suspended active tuple would be, while it was in this form, a sort of passive tuple — call it 'pseudo-passive'. By using `rd` instead of `in`, a process or processes could create several copies of pseudo-passive tuples which could be activated later as required.

In fact, given this interpretation of `eval` permission on active tuples, there is a natural extension to individual active tuple *elements*. If an active tuple (created by `eval`) has only *some* elements with `eval` permission, then those elements would be concurrently executed, while those without the permission would remain suspended. The result would therefore be a pseudo-passive tuple containing some suspensions and some 'ground' values

2.2 Semantic Implications

Extending the LINDA model with an attribute framework will of course change the semantics — there now exist properties that could not be achieved previously. However, these properties might be purely *extensions*, that is they do not modify the semantics of the underlying model, or they might necessitate a revision of the underlying semantics.

In the first group — those which do *not* change the model — are all of the first six rows of the 'ticked' entries in the design space, except for the `out` attribute. For instance, due to the non-determinism in LINDA, a process would not be able to detect the difference between a `rd` which is blocked due to a tuple being unavailable in tuple-space, and being blocked due to a tuple (or tuple-space) denying `rd` permission. Similarly for the `in`, `wildcard`, and `match` attributes. However, with `out` the situation is different. If a tuple-space denies `out` permission to a process, then one of two outcomes could result depending on the implementation. Either the `out` operation *fails*, in which case the host *computational language* must have some exception mechanism to deal with this, or the operation *blocks*. Both of these represent changes to the user's view of the model's semantics. In the first case, the independence of the computational and coordination languages is broken. In the second, a fundamental alteration in the semantics of `out` has been forced — `out` *never blocks* in the standard LINDA model. Unfortunately, there is no clear choice between the two alternatives — to rely on integrating the LINDA coordination system with a host's exception mechanism would, at least, mean that existing LINDA programs would need modifying. On the other hand, enforcing a 'blocking `out`' semantics would introduce subtle (and not so subtle) problems in using existing techniques for analysing and designing LINDA process ensembles.[8] The problem of choosing between the two alternatives will be side-stepped in this paper by considering that processes are so well-behaved that there will be no attempt to do an operation that would be prevented by an attribute. In this way we shall be able to concentrate on the new coordination structures made available by incorporating an attribute framework into LINDA. There is, therefore, further work to be done in this area.

A change in semantics is also introduced with `eval` permission which, while not modifying the underlying LINDA model, *does* change the behaviour of an ensemble in a way that a process *could* detect.

Without the existence of `eval` permission, as in the standard model, the fact that a process

[7] The subsequent process would `out` the tuple with `eval` permission allowed.

[8] It would, for instance, affect questions of deadlock and termination.

executes an `eval` operation can only be detected by some other process by accessing the resulting *passive* tuple. This will only occur after all the active elements (functional processes) in the active tuple have completed. When the active tuple elements execute they can have arbitrary side-effects, including the modification of tuple-space contents. Consequently, when a process detects that the `eval` has occurred, all of its side-effects will also have occurred.

With the introduction of the `eval` attribute, a process can 'know' that the `eval` was executed by accessing the pseudo-passive tuple, *before* any of the side-effects have occurred. This is not necessarily a *problem*, since the new behaviour can be accounted for when designing the process ensemble. However, it *is* a change of semantics, and should therefore be understood.

The following sections will consider only the basic access-control attributes in the examples. However, it is hoped that by discussing some of the other aspects of the basic 'design space', a feel for the potential improvements in expressiveness brought about by the introduction of an attribute framework can be gained.

3 Attribute-Supported Coordination Structures

The purpose of a coordination language is to enable the expression of coordination structures in the same way that a computation language supports the definition of control and data structures — both are means for specifying *algorithmically* the interactions between component objects that are composed to achieve a task. Deciding what constitutes a coordination structure is, however, rather awkward since there is no body of coordination theory to compare with the theory of computation. Consequently, an informal definition will be used here — a coordination structure describes and governs the temporal and spatial interactions between agents, without reference to their internal computational structure.

This formulation leaves some 'grey areas' at the interface between computation and coordination, but will suffice for the purposes of this paper since the examples used will be firmly coordinative.

Given the the difficulty in defining what constitutes a coordination structure, it is perhaps rather unfair to suggest that LINDA is deficient in its ability to express some coordination structures — it could be argued that such structures are *not* coordination structures, and therefore that LINDA is not deficient. However, at least one facility which appears in other non-LINDA coordination languages (for example IWIM [4]), the provision of private or point-point communication, cannot be implemented using the classical LINDA model. Consequently, there is a *prima facie* case that the classical LINDA model requires some enhancement. The rest of this section investigates such an enhancement.

The following two sections give examples of the types of coordination patterns or structures that become available once an attribute system is incorporated into LINDA. The first section addresses those structures that are generated by the basic access-control attributes discussed in section 2.1. Section 3.2 touches on attributes which do not address access control, but which may affect the ways in which a process ensemble behaves.

As mentioned before, all these coordination patterns can be split into those which could be achieved before but are now more efficient, and those which were previously unavailable.

3.1 Access-control Attributes

In this section, three attribute-supported coordination structures are described. In order to do so without preempting the discussion in section 4 on implementation choices, the notation used for expressing attributes is simplistic: only the attributes applicable to the structures are shown, and these are `in`, `rd`, and `out`, which are represented by their initial letters. Attributes are grouped into the two *domains* which are applicable — the owner or creator, and all others, respectively. In a full system there could be a large number of domains, defined in a variety of ways (see the discussion in section 4.2).

In the code examples, tuples are assigned access attributes by decorating the `out` operation's tuple parameter with the required attributes. Tuple-spaces are created with the `tsc` primitive, which is extended to take an attribute set as a parameter.

3.1.1 Private Channels — a new coordination structure in LINDA

It is often the case that (subsets of) agents need to establish means by which they can communicate without any interference from other agents in the system. Many coordination languages have explicit structures for setting up such 'private channels' — indeed, private, point-to-point communication is the *only* means of coordination in most common coordination systems. In LINDA-like systems, however, the coordination model is based on *broadcast* communication through tuple spaces which is more general, but less 'secure'. The lack of security can be demonstrated by considering a pair of agents which wish to coordinate by the communication of some tuple. Agent 1 `outs` the tuple into a tuple-space, and Agent 2 `ins` the tuple, *by specifying a matching template*. Unfortunately (for the purposes of a private conversation), there is nothing to prevent a third agent removing the tuple — and this could happen *accidentally* (as well as maliciously).

This can partly be solved in closed systems, since it is then assumed that the *whole* of the system of agents is known about, and is under the control of the designer. Consequently, there would be no 'malicious' removal of the tuple, and any 'accidental' removal would be a design fault (which could be fixed by a suitable re-design of the tuple formats). However, in the case of open systems, there is no such single locus of design (or control), and so there would be no way of ensuring that the private coordination tuple format is unique.

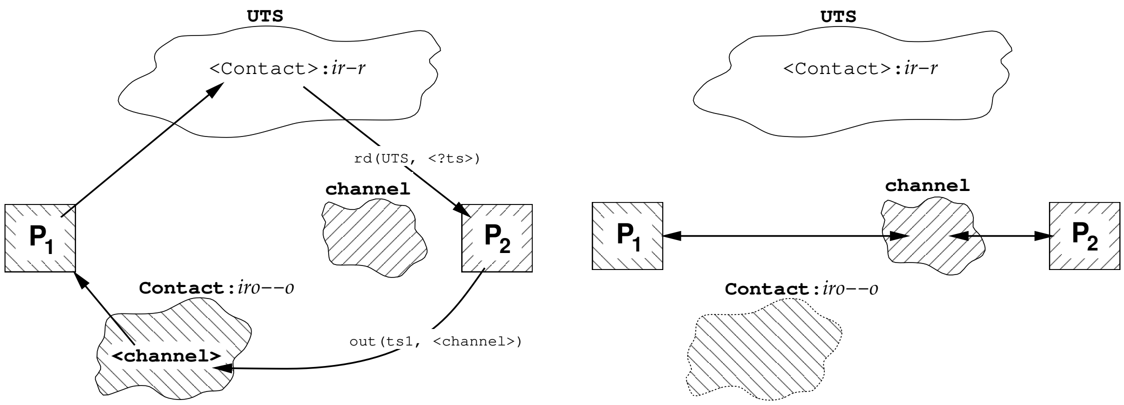
The original LINDA model was based on a single tuple space, globally accessible to all agents. This version was very quickly extended to allow multiple tuple-spaces, each of which had variable scope with respect to the agents.[9] At first sight this seems to solve the problem — a private channel is modelled by a tuple-space that is only known to (in the scope of) the two agents concerned. In the case of closed systems, this gives a better solution since there is then no need to make use of 'baroque' tuple formats in order to maintain their uniqueness — the tuples can contain just the required information, with the limited tuple-space scope providing the required protection against interference. The same is not true of *open* systems, however, since the problem of how to ensure that each agent has knowledge of a reference to the 'private' tuple-space remains.[10] In LINDA, the *only* way in which information can be conveyed between agents is via a tuple-space to which both have a reference. For an open system, initially this must be *the* globally accessible tuple space (referred to as the 'Universal Tuple Space' or `UTS` here) since in such a system no two agents have any knowledge of each other initially. Consequently, the information required to set up the private channel — the reference to the 'private' tuple space — will have to be transferred, *in a tuple*, through a non-private channel, which merely revives the original problem.

[9] How this scope was defined varied between different variations on the LINDA model.

[10] It is assumed that, in the case of a closed system, this knowledge is available at design time, and would therefore be programmed into both agents.

The solution is to make use of access control attributes. It will be seen that attributes attached to both tuples and tuple-spaces will be needed.

A sketch of the core of the method is shown in figure 2, with the skeleton of the required code (in a `C`-like syntax) given in figure 3. One agent (process P_1 in the example) broadcasts that fact that it is willing to engage in a private conversation by placing a tuple containing a reference to a tuple-space (of which it is the creator) in UTS. The tuple has `rd`-only access to all other agents. This prevents any accidental or malicious *removal* of the tuple, which is the first part of the guarantee against interference in setting up the private channel.



(a) Establishing contact

(b) Channel created

Figure 2: Creating a Private Channel using Access Attributes (shading matches a tuple-space with its creator.)

The tuple-space, `Contact`, is intended to be the means by which the other agent (P_2) informs P_1 of a reference to the private-channel (which is a tuple-space known only to the two agents involved). `Contact` is created with `out`-only access with respect to all other agents, so that its contents cannot be altered by any other agent. This gives the second part of the guarantee against accidental or malicious interference in the process of setting up a private channel — no third-party would be able to gain knowledge of a reference to the 'private-channel' tuple space which is passed in a tuple in `Contact`.

```
Contact = tsc( iro--o );
out( UTS, <Contact>:ir-r);
channel = in(Contact, <?ts>);
/* Use channel */
```

(a) Process P_1

```
channel = tsc( iroiro );
contact = rd( UTS, <?ts> );
out( Contact, <channel>:irir );
/* Use channel */
```

(b) Process P_2

Figure 3: Skeleton Code for Figure 2

Once `channel` has been created and its reference passed to P_1 via `Contact`, P_1 and P_2 have a secure means for coordination. Of course, it is likely in practice that more negotiation would take place between P_1 and P_2, which would probably require more information than just the tuple-space references to be passed in the tuples. However, this can easily be accomplished now that private channels can be created.

This simple example sets up a single channel between two cooperating processes, and therefore tuple-space `Contact` will be unused (and thus garbage collected[11] if possible). However, it is possible that further channels to P_1 might be required later, in which case the protocol for establishing a channel could be repeated (consider the case where P_1 provides a service, such as printing).

3.1.2 Provider-Users — aiding performance

The previous example showed how a coordination facility not available in the classical LINDA model could be provided by means of object attributes. This section demonstrates how the same access-control type of attribute can be used to increase system *performance* by providing extra information to the implementation.

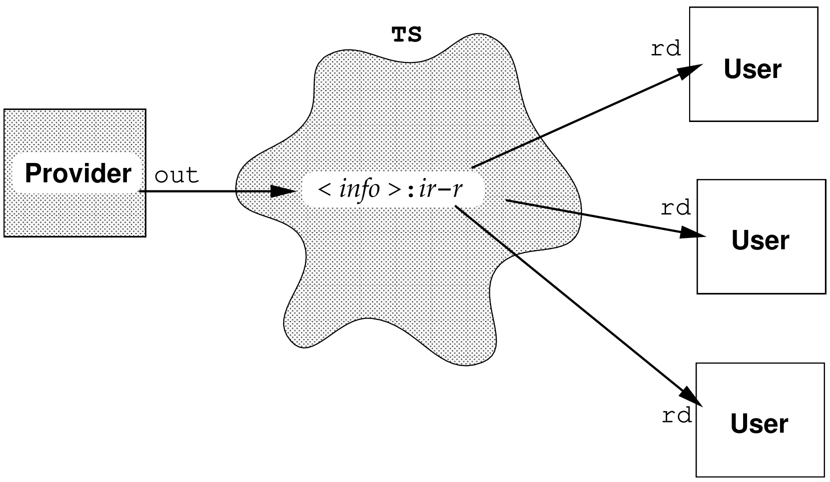


Figure 4: The Provider-User Model

Figure 4 illustrates an extremely common situation — a single agent provides information for a number of clients, or users of that information. The WWW is an obvious example.

The structure of the coordination is superficially the same as a Master-Worker style of computation. However, in the provider-user situation, it is assumed that the information provided does not change very often (relative to the number of retrievals by the users), and the units of information might be quite large.

The first need for access-control is obvious — it should not be possible for any agent except the provider to remove, or otherwise modify, the information. Hence, either the tuple-space or the individual tuples should be `rd`-only for users. The *creator* (Provider in this case) would need to be able both to `in` (remove) existing information, and to `out` new tuples. In the classical LINDA model, enforcing this type of control in an open system would be impossible

[11] Garbage collection, and other resource management issues in open distributed tuple-space systems are currently research problems [13].

(and even in a closed system, it might be difficult to engineer, especially if the system were large).

One efficiency problem associated with the provider-user mode of operation, especially when there are a large number of widely-distributed users (as is commonly found in the WWW, say), is that the single source of information becomes a severe bottleneck. The traditional technique for overcoming such a bottleneck — replication of the data — causes a problem for the LINDA model. Since tuple-spaces are *bags* and not *sets*, the number of copies of a tuple in a tuple-space is significant. Consequently, if replication is to occur (at the implementation level), this must be achieved while presenting all agents with a view of the system which is still consistent with the LINDA model. In general, this would be prohibitively expensive in practice, since it would require global synchronisation of all tuple-space 'removal' operations. This would defeat the object of having a fully-distributed system, and in any case would move the bottleneck from the single source of information to the single point of synchronisation.

However, by introducing the necessary access controls the implementation (or LINDA *kernel*) is given some extra information about the way in which the tuple space is to be used (since the maintenance of the access-controls will be implemented by the kernel). In this case, the kernel will know that the tuple-space can only be *modified* by a single agent ... all other agents being `rd`-only. Since the only way that an agent can *count* the number of instances of a tuple in tuple-space is by repeatedly *removing* them,[12] whether a tuple is replicated does not destroy the 'LINDA-consistency' with respect to agents which can only `rd` such tuples. Consequently, when the kernel identifies the situation shown in figure 4, it is at liberty to replicate the information with respect to the User agents.

The problem remains, of course, of how to maintain the LINDA-consistent view of the tuple space for the provider (which *could* count the tuples since it has `in`-access). This will still be costly for the kernel since *all* copies of a replicated tuple would have to be removed from the (probably distributed) implementation of the tuple-space if the provider were to `in` the replicated tuple. However, this need not mean that a global synchronisation occurs, nor that the distributed tuple-space need be 'locked' for any time. This is because of the asynchrony, and non-determinism that are fundamental to the LINDA model.

Finally, it is interesting to note that these considerations have much in common with the issues involved in the implementation of 'Proxy Caches' for the WWW. In fact, the scheme described here is a kind of 'dual' of the most common form of Web cache in which Web clients' accesses to pages are mediated through a proxy, which retrieves the page, but caches it locally before passing it on to the client. Then, any further requests for the page, either by the same or another client, can be serviced from the local cache rather than the distant server.[13] This caching scheme has been called 'client-pull' since it is the client (User in the provider-user situation) that initiates the demand. The method described here is much closer to the 'server-push' caching schemes that are the subject of current research [5, 11].

[12] In some variants of LINDA, for instance the York LINDA Kernel [19], there are *bulk access* primitives which remove tuples from a tuple-space, and return information on how many were moved. This provides another (more efficient) way of counting tuples, but it still involves tuple *removal*.

[13] Of course, there is the need to detect whether the page has altered on the remote server before servicing the request by means of the local caches, but this normally involves a communication from the proxy to the server which is significantly shorter (and hence quicker) than the requested page itself.

3.1.3 Global Software Distribution — an application of access attributes

This final example of access-control attributes shows how private channels, scoped tuple-spaces and controlled access to tuples can be combined to provide an architecture for the 'hard media free' distribution of software. In addition, if some further technological problems were to be solved together with the opportunities for *caching* described in the previous section, this architecture could provide a platform for the fully distributed operating systems of the 'Network Computer' type [20], or the 'Millenium Project' [6].

When a user acquires a piece of software from a producer, whether by paying money to a commercial suplier or by copying it from a free source, several things must happen. Firstly, the potential user must know *how* to acquire it — its existence, who can supply it, how to contact the supplier, etc. Secondly, a request for the software must be made to the supplier — this will involve different protocols depending on whether or not money must change hands, whether the software is shareware or freeware, whether or not a formal licence is required, etc. Next, the user must obtain a copy of the software — normally on hard media if commercial, or electronically if free/shareware. Finally the user must install it.

This procedure (or set of procedures) work reasonably well, but there are a number of inherent problems with it:

- Once installed, the user will make use of the software's facilities. This might seem rather an obvious statement, but it will be seen to be important later, as not every user will use every facility. Consequently, especially for a complex piece of software (such as an operating system or large word-processor or spreadsheet package), it is unfortunate that multiple copies of little used parts of the software will use resources unnecessarily.

- After some time it is almost certain that bugs in the current version will be discovered. In order to obtain the latest 'debugged' version, at best the supplier would have to send patches to all the users — for commercially produced software this would involve multiple mailings of hard media, and for freeware at least an emailing to users informing them of the location of the patches or corrected version. Both of these 'best case' situations are very unlikely to occur. More likely, the user just has to wait for the next release (with new bugs, of course) to be announced from a commercial producer, or to keep revisiting newsgroups and/or the original site for news of patches.

- In the case of commercial distribution, every time a new user requests the software, a new copy has to be produced, and this will be supplied on some sort of hard media. This medium is often only used once, and is discarded once the installation process has been completed. This is both wasteful and expensive.

- One of the major problems that software producers encounter is unauthorized copying. Once the authorized copy has left the producer (electronically or on hard media), almost all control over it has been lost. In the absence of any separate key-based licence-activation scheme (such as those used by major workstation manufacturers), the only forces preventing unbridled copying are personal ethics, fear of litigation, and technical or economic difficulty (as when copying CDs). Experience shows that neither of the two former forces are very strong, and technology continually overcomes the latter.

Now consider how a tuple-space based system, with access-control attributes, might be used to overcome these problems, while admitting some new possibilities. Figure 5 outlines the skeleton of such a system.

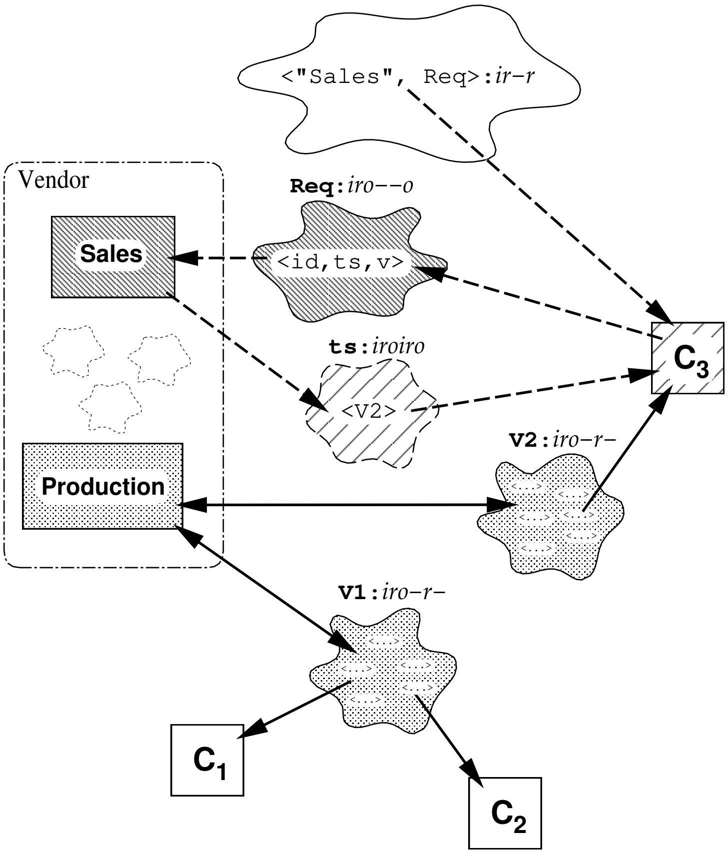


Figure 5: The Use of Access Attributes in a Global Software Distribution System

The configuration in the lower part of figure 5, involving the customers C_1 and C_2, the Production agent (part of the vendor), and tuple-space `V1`, represents the goal — a number of users of version 1 of the software, which is stored as a set of tuples in `V1`. `V1` can only be modified by Production, and is only known to the vendor and the authorized customers.

The upper part of the figure shows how a new customer would acquire access to the software. The situation shown is that C_3 wants version 2 of the software, in which case it would communicate with the Sales agent of the vendor via a private channel established using the protocol described in section 3.1.1. The information passed to Sales would be the customer's `id`, the version of the software wanted, and the private-channel tuple-space reference. Once Sales is satisfied[14] that C_3 is entitled to the software, it would obtain a reference to the appropriate tuple-space from Production (using private channels local to the Vendor), and pass this to the customer. Then C_3 would be in a situation similar to customers C_1 and C_2, but with access to version 2.

This method for distribution does not depend on the actual representation of the software as tuples in the tuple-space. In the simplest case it would be possible to have the entire piece of software as an element of a tuple. Then authorized customers (those with a reference to the tuple-space) could either copy the software to local storage, or run the software from the

[14]The details of how Sales would negotiate with the customer for payment etc. are not important here.

tuple-space (in effect 'copying' it by means of their system's program loader). This would require no fundamental alteration to the way in which the software would be installed and used ... only the method of delivery would have changed.

However, by representing the software as a *set* of tuples, a number of novel possibilities for software architectures can be considered. The tuples might then correspond to library routines (as in pre-'object-oriented' systems), objects, or services (as in operating systems). Some of the potential advantages of this arrangement are:

Bug fixes. When modifications to the software are required that don't imply a new full release, only those components that require correcting need be replaced. An *update* of a tuple (routine or object) would correspond to the Production agent removing the original version (using an `in`), and `out`ing the replacement (the mechanisms by which the customers might then learn of the change is dealt with later). Therefore, a completely up to date version of the software would always be available to both existing and new customers alike — management of bug-fixes by the vendor would become relatively trivial.

Copy protection. A degree of copy protection — more accurately, copy *discouragement* — becomes available. It is likely that most of the tuples accessed by customers would contain 'interface' routines, or 'wrappers' which call other services/objects. Within a tuple-space based system, these lower-level routines would be accessed via tuple-space operations as would the routines that they require. Consequently, it would not be necessary to have the *complete* code for the software existing in the customer-accessible tuple-space — much of the system could exist in tuple-spaces local to the vendor which is only accessed by the *code* of the routines contained in the customer tuple-space. This has two effects. Firstly, since only the 'interface' tuples formats need be made known to the user, it would be practically difficult to take a copy of the *entirety* of the tuple-space, since this would require being able to specify templates to match *every* tuple, and this would not in general be possible. Secondly, even if a complete copy of the tuple-space's contents were able to be made, which is doubtful, there would be no simple way of obtaining copies of the lower-level routines.[15]

The execution model resulting from this system is similar to demand-paging — the routine (code) for the required service is loaded by `rd`ing the appropriate tuple. When the execution of this routine caused the retrieval of the contents of further tuples this would be equivalent to 'demand' loading the required code — the newly loaded code could then retrieve code-tuples itself, and so on. This could form the basis of a Network Computer style of operating system in which the operating system's services were represented as code contained in tuples.

However, currently this execution model would not be feasible since it implies a *single* source of the code, which would be frequently accessed by a very large number of agents. A single tuple-space could not cope. For this to work a very efficient caching and replication scheme would need to be implemented. Therefore the techniques described in section 3.1.2 would need to be developed.

If the caching and replication problems were satisfactorily solved, then this execution model could be comparable in efficiency to the disk-based demand-paging schemes in common

[15]This would at least require disassembling the routines' code to determine all the references to vendor's local tuple-spaces. Even then this would not necessarily be the end of the unethical copier's task, as the routines thus copied could call the contents of tuples in yet other unknown tuple spaces.

use.

3.2 Other Attributes

The previous sections have shown, in detail, how *access-control* attributes can be used to significantly enhance a system's coordinative vocabulary. It has been argued that, despite focusing on LINDA-like systems, incorporating such an attribute framework is *generally* useful in any coordination language. However, once the notion of attribute-based coordination is admitted, there are many attributes other than access-control that might be considered — often these will be more or less specific to the actual coordination platform being used. This section will briefly describe attributes not concerned with access-control, which are of particular interest in LINDA systems. Other coordination systems will have similar examples.

An important point about both the following attributes is that neither affects an agent's view of the LINDA model. Although the behaviour of a LINDA system incorporating these attributes *is* modified, the behaviour is one of the several that are consistent with the model.

3.2.1 Ordering

It is sometimes the case that the order of insertion of tuples into a tuple-space is important to an agent retrieving those tuples. For instance, if the tuple-space were to represent a keyboard buffer, then the order in which the tuples are inserted would reflect the order in which the keys were pressed. Consequently, the tuple-space (or its controlling LINDA kernel) would need to maintain sequence information. This could be specified as an attribute of the tuple space.

In the absence of attribute-supported ordering, this would have to be emulated when needed by the inserting agent producing a sequence number as an extra element in the tuples that it `out`s. The retrieving agent could then be written to access tuples with the modified format. Superficially, then, there is no *essential* need for ordering as a kernel-supported attribute. This would be true for *closed* systems. However, for open systems the need for ordering *attributes* is clearer. If a consuming process requires ordered access, it would be difficult to enforce that requirement on an *arbitrary* producer — in an open system an agent can have no prior knowledge of the agent(s) which might retrieve the tuples that it produces. Consequently, it could not know that ordering would be required, and thus would not be written to add the sequence information. In the case of the 'keyboard agent' mentioned above, it might be argued that it is natural to include this information. However this will not always be the case, hence the need for attribute supported ordering.

3.2.2 Non-determinism

LINDA's associative access mechanism is *non-deterministic*. That is, when there are two or more possible outcomes to an `in` or a `rd`, it is not possible to predict which will be chosen — running the same program on different LINDA implementations, or at different times on the same implementation, could cause different choices to be made. However, this is not the same thing as *random* access, which is related to the *fairness* properties of the system. Consider a process which repeatedly executes a `rd` of a tuple-space containing several tuples which match the `rd`'s template. LINDA's non-determinism ensures that the process cannot predict which tuple it will receive, but it is quite valid for the kernel to supply the *same* tuple every time.[16]

[16] This is not only valid, but highly likely in practice — the 'natural' way for a LINDA kernel to represent

A random access scheme, on the other hand, would identify *all* candidate tuples and randomly choose one to be returned. This would guarantee fairness (all candidates would be chosen equally often) over a sufficiently large number of repeated accesses. However, it is not merely in matters of fairness (which affects a program's *correctness*) that the distinction between random and indeterminate access is important. In some circumstances an improved *performance* can be obtained with a random system compared with a purely non-deterministic implementation [9], Therefore, 'random choice' is an example of a useful behaviour-modifying attribute.

4 Implementation Considerations

So far attributes have been represented abstractly as a set of flags attached to representations of the objects. This has enabled discussion of the issues without being concerned with how the abstractions are actually to be implemented. In fact, if the only attributes were access-control, and it were only necessary to distinguish an object's creator from other agents, then the implementation would follow this representation. However, in the context of open distributed systems, it will be necessary to provide a level of attribute control that is both more general, and finer, than can be achieved with this simple model. This section discusses the implementation issues and alternatives in the open system environment.

4.1 Conventional Approaches

There are two approaches to the implementation of access (and other) attributes described in the standard operating systems literature (see for example [17, 21, 22]).

Access Control Lists The first approach is essentially that assumed in the earlier sections of this paper[17] in which an object, generally a file or device, has attached to it a list of sets of attributes. Each set in the list represents the attributes that are appropriate for a given *domain* of agents when accessing the object. In UNIX there are three domains — the owner, a 'group', and all others. This method of implementing attributes is called the *Access Control List* (ACL). The exact means by which the domains are defined and maintained will vary from system to system.

Capabilities In the second method, each domain holds rights to the object's attributes that it requires, rather than the object maintaining a list of domains' attribute rights. If a domain needs to perform some action on an object, it requests the right to do so from the object. If the request is granted, the object returns an unforgeable 'passport' — a *capability*[18] — which identifies the domain, the object, and the rights granted. When an element of the domain (a process) needs to perform some action on the object, it presents the object with a copy of the capability along with the access request so the object can determine whether the process possesses the right to do the requested action.

a tuple-space is as a list. Then a retrieval operation would return the first tuple in the list that matches. Repeating the operation would merely find the same tuple again.

[17] This is the system used in UNIX (and to a limited extent to DOS/Windows).

[18] As far as the domain's elements (processes) are concerned this would be some 'opaque' bit-string. For full details of how this is implemented, see the references [22].

Although the technique still encompasses the idea of a domain consisting of several elements, in practice it is normal to have a capability *per process* and to identify domains with processes.

These two techniques have several complementary properties. The following section compares the two in the context of coordination in the type of open distributed system that has been the main concern of this paper.

4.2 Linda-like Systems — Requirements and Opportunities

The major difference between capabilities and ACLs is that the former require that the objects to which they apply have *names*, otherwise there is no way for a capability to be matched to an object. Consequently, an open distributed system using capabilities for its attribute framework must be able to supply a global name-space for all the objects which might be part of the framework. In a Linda-based system, this *must* exist for tuple-spaces, otherwise there would be no way for arbitrary tuple-space references to be passed between agents. Consequently, it would be possible to implement capability-based attributes for tuple-spaces.

However, one of the most fundamental aspects of Linda-like systems is that accessing the contents of tuple-spaces is associative. That is to say, *tuples have no names* — they are only identified by their contents, and this identification is, of course, not unique. Therefore, there would be no way in which a capability for a specific tuple could be formed.

On the other hand ACLs, being *attached* to the objects to which they apply, do not require the objects to be named, and thus could readily be used with tuples. However, they could also be applied to tuple-space attribute management — the fact that tuple-spaces *are* named would be irrelevant. So it would seem that ACLs are the best means for implementing an attribute framework. This would be the case if it weren't for the need to introduce the framework into an open distributed system, which exposes the main *functional* disadvantage of ACL-based systems.

Since ACLs are attached to each attribute-controlled object, either each object will have a huge list of dynamically varying length (containing attribute sets for each agent in the system), or there must be a limit to the number of domains in the system (thus fixing the size of the list). The former is not practical in anything other than small closed systems, and in open distributed systems would be impossible to implement since, in addition to arbitrarily long lists attached to each object, it would require that every object's ACL be modified whenever any new agent enters the system, or old agent leaves. The latter case, limiting the number of domains, would require a means for managing domains (creating new ones, adding/deleting agents to/from domains, etc.) which would need to be distributed in order to prevent the domain manager from becoming a bottleneck. Whilst this is possible (and might be usefully implemented using tuple-spaces), it could severely limit the flexibility of the system.

Summarising the pros and cons of ACLs and capabilities in the context of open distributed tuple-space systems, then:

ACLs

+ Applicable to un-named objects.

  Therefore they can be used with tuples, tuple elements, and tuple-spaces.

– Require distributed domain management.

  This limits the flexibility and increases complexity.

– Or, require arbitrarily large lists attached to each object.

  This is not practical, *especially* in an open system.

– The set of applicable attributes is fixed by the system, since each ACL element must account for the attribute *set* for that domain.

Capabilities

+ Easily implemented in an open distributed system.

  Agents store their own set of capabilities, putting no storage overhead on the objects to which they refer.

– Cannot easily *withdraw* premission, or otherwise modify a capability once granted.

– Cannot be used with anonymous objects.

  This limits their use to tuple-spaces alone.

Therefore there is no clear choice between the two methods for an open distributed tuple-space system, and so a compromise solution must be fund. Since there is no choice but to use ACLs for tuples and tuple elements, it is proposed that the simple, two domain ACL system implicitly used in section 3 be implemented — the domains being *creator*, and *others*.

For tuple-spaces, however, a capability system should be used for several reasons.

Firstly, the lack of 'domain flexibility' inherent in the use of ACLs for tuple-space contents can be overcome by designing process ensembles to use tuple-spaces (with capabilities) as the prime means of attribute-control thereby regaining access to the richness of domains that is a property of capability-based systems. Secondly, user-defined attributes can easily be handled — a kernel needs only to know that an attribute *exists* in order for it to be able to grant, and verify, a capability for that attribute. Finally, a capability-based scheme is not restricted to a fixed set of attributes — this set may change over time, which is essential for an open system since its life-span might be very long.

The protocols described in the literature for managing capabilities are quite straight-forward, and are easily transferred to a tuple-space based system. For example, in the Amoeba system [16], the creator of an object has a capability for that object returned from the kernel as a result of the creation request.[19] This gives the creator *carte-blanche* permission over the object. Then, in this system, when another agent wishes to access the object it sends a request to the owner (creator). The owner then requests a new capability for the object from the kernel, passing (a copy of) its own capability with the request (in order to verify that it has the right to request a new capability). The new capability is requested to have the attribute that the owner is willing to grant to the other agent, to which it is finally returned. It is easy to see how this mechanism can be translated into a sequence of tuple-space operations between the agents, especially now that private channels can be set up.

Finally, there are four important properties that this capability scheme has which positively affect the implementation.

[19] A good description of the format of a capability in Amoeba, which illustrates its essential properties such as unforgeability, can be found in [22].

1. The kernel is only involved in generating and checking capabilities — initially deciding whether or not an agent is to be granted some attribute rights to an object is the responsibility of the object's creator.
2. All the information required to generate capabilities (object identity, agent identity etc) would naturally exist in any 'conventional' LINDA kernels.
3. Since all LINDA operations must be serviced by the kernel, the only overhead involved in incorporating a capability-based scheme is the extra communication costs due to the need to provide the capability in addition to the other parameters in an operation.
4. No information regarding capabilities need be stored in the kernel — the information contained in the capability itself is sufficient for the kernel to asses its validity for the specified object.

The remaining problem with capabilities is more awkward to solve — withdrawing an agent's capability for an object is difficult. Several methods have been reported which all result in revoking the capability of *all* agents for a particular object. Any finer degree of control seems very difficult to achieve. However, it is most likely that, in a LINDA-like system, the only need to revoke a capability would be when the creator (or one of its proxies — agents which have been given creator capability) needs to block access temporarily — for example, if it needs to 'freeze' a tuple-space when doing a series of operations. In this case, a request to the kernel to temporarily withdraw non-creator access permission would be sufficient — all processes would merely block until the permission were reinstated.

5 Discussion

It has been argued that an attribute framework is necessary for complete coordination in open distributed systems. In particular, access-control attributes have been shown to be *essential* for LINDA-like tuple-space systems to be able to provide at least one coordination structure — private channels – that could not be achieved previously. Other desirable coordination properties also emerge from such an attribute framework.

The implementation proposed is a mixed system combining both Access Control Lists when necessary, with Capabilities when possible. It has been argued that a capability-based implementation is particularly natural in the context of LINDA.

The ideas discussed in this paper are being implemented in the latest in the series of LINDA kernels developed at York. This will also incorporate recent advances in distributed resource management (in particular, garbage collection [13]) and the confluence of Input/Output with the LINDA model [14].

5.1 Related Work

The deficiencies in LINDA, in particular the inability to express directed, or point to point, communication have been noted by several authors. In general the response has been to propose entirely new systems (or existing systems extended to work in a distributed environment) which overcome the perceived deficiencies, but at the expense of losing the advantages that LINDA-like systems have over many of the proposals in the area of open systems.

An example of an attempt to solve the LINDA problems by improving LINDA, is that of Pinakis's Joyce-Linda [18]. The focus was solely on the directed communication (private channels) problem, and developed a solution which was based on a type of public-key encryption technique inspired by the capability-like port concept in an early version of the Amoeba operating system. There are, therefore, strong links with the ideas presented in section 4 regarding implementation of the attribute framework. However, Pinakis' solution had difficulties in communicating "tickets" (his proto-capabilities) through tuple-space securely, since a basic property of ticket values was that they could not be matched by formals in templates. Consequently he had to resort to the rather unsatisfactory solution of introducing a second special data type (tickets being the first) to which ticket values could be coerced, and hence matched by formals. Hence it can be seen that his implementation incorporated some of the properties of the capability-based implementation of the attribute framework discussed here.

Finally, the Joyce-Linda system "relies on the compiler and runtime system", and therefore would be difficult to adapt to the *open* systems problems addressed in this paper.

Two proposal taking the more radical approach to the LINDA deficiencies are those of Law Governed LINDA [15], and ActorSpace [2].

Minsky and Leichter's motivation in developing Law Governed LINDA was to address LINDA's "inherent lack of safety", specifically citing its inability to support "pair-wise communication" (private channels). The reason why they do not abandon LINDA completely is that they regard it as "particularly suitable as a coordination model for open systems", and that it is by implication merely the lack of safety that prevents it from being more widely used. They identify the principal safety problem as the need for component processes in an ensemble to rely on "voluntary conventions" regarding tuple/template formats and hence tuple-space accessing. This mode of operation is just acceptable for closed systems, but cannot be expected to work in open systems. Therefore they propose to replace these voluntary rules with globally enforceable 'laws'. This is to be achieved by the system assigning a 'controller' to *each process* which mediates between the process and the system, thereby ensuring adherence to the built-in law.

Again there are parallels with the attribute framework described in earlier sections — in this case there is a rough duality between LGL and the attribute framework. Whereas LGL's integrity is maintained by the *agents* (or their controllers in conjunction with the system), in the attribute framework the control resides with the *objects* (in conjunction with the kernel). Which system has the performance/resource advantage would depend on the relative complexities of the controlling components and the relative numbers of objects and agents.

The LGL system is very powerful and the laws that can be expressed are extremely general. The authors do comment, however, that the efficiency of the system is still to be determined.

The ActorSpace proposal of Agha and Callsen [2] also criticises LINDA for its insecurity, but makes no attempt to fix LINDA, preferring to develop Agha's Actors [1] to work in the open distributed environment. The result is a new coordination language, rather more complex than LINDA, in which the focus is on a 'generalized' direct (point to point) communication based on a pattern matching mechanism. The 'patterns' which are involved in the match in essence define attributes in the sense used in this paper. In their proposal, the authors make the case for attribute-based control of system activity, and choose to provide security by means of capabilities. Their work is therefore in several ways directly comparable to that described in this paper, only using a different coordination basis.

The final work to be considered here does not directly address the issues in this paper, but since its main thrust is generalization of the tuple-space concept, it sheds light on several themes encountered here. The $\mathcal{TS}$ language reported by Jagannathan [12] is based on Scheme and is firmly 'higher-order' — that is, all the data structures, including the generalization of tuple-spaces, are first-class objects, implemented by higher-order functions. By using this representation, the behaviours of the tuple-spaces are highly customizable, to such an extent that LINDA's tuple-spaces become a (particularly simple) instance of $\mathcal{TS}$'s generalized tuple-space model. In $\mathcal{TS}$, tuple-spaces are defined by four higher-order functions, called *policies*. These policies control: how the tuple-space itself is seen by an accessing process; how tuples and templates are *matched*; under what circumstances an operation *blocks*; and under what circumstances an operation *fails*. This last 'policy' is interesting as it has a bearing on the blocking *versus* failing semantics for the `out` attribute discussed in section 2.2. The $\mathcal{TS}$ proposal is very powerful and, due to its extreme generality, would probably present the ideal solution to the problems addressed in this paper. Unfortunately it is extremely difficult to see how it could be efficiently implemented over multiple heterogeneous processors, especially when these might include severely limited capability devices (such as PDAs). Therefore it would not be suitable for open distributed systems. However, as a 'meta-model' of tuple-space based coordination, it deserves more attention.

5.2 Future Work

There are a number of areas for further research and implementation work. The first concerns a potential performance problem which arises from capability management in the LINDA kernel.

In a distributed system, 'the' kernel is, in fact, composed of a number of distributed sub-kernels, each handling some subset of the (possibly partitioned and distributed) tuple-spaces. Depending upon precisely how this is arranged, there might be considerable communication traffic between distant sub-kernels on capability generation, or verification, or both, since the identities of the objects and agents involved might not be available on the same sub-kernel. It is expected that a suitable caching scheme will overcome this problem, probably based on the way in which objects are hashed across the sub-kernels.

Another area for research will be to investigate potentially useful *new* attributes and their coordinative effects. Several possibilities are:

Persistence and Fault-Tolerance. There has been considerable research progress in these areas as applied to LINDA systems [3]. By making them attributes that could be applied at will to tuple-spaces, it is possible that improved performance might be achieved by using cheap, 'dirty', transient tuples spaces most of the time, and only expensive, clean and permanent ones when necessary.

Cost. In a distributed system, different people own different resources. The cost of storing a tuple in a tuple-space will vary, as will the cost of executing (using `eval`) processes in a tuple-space. Consequently it appears that there is some cost attribute associated with tuple-spaces. However, it is not obvious how this might be incorporated into the formal attribute framework.

Type Views. [20] Currently, an instance of a tuple in a tuple space is seen in only one way — the form in which the `out`ing process deposited it. However, as with 'Union' types in

[20] Thanks are due to Dr Simon Dobson of the Rutherford Appleton Laboratories, UK, for this suggestion.

languages like `C`, it would be possible for the tuple-space to offer to allow agents to view tuples in several ways. An obvious example would be for a tuple containing four bytes to be viewed as two short integers, and hence be matched by a template consistent with that type. In other words, a tuple-space would provide a system of type-matching and run-time negotiation which attempts to allow different *views* of the data to exist while maintaining its integrity.

These and other possibilities will be investigated.

The final area for research is to attempt to devise methods by which the 'name-full' capability system could be integrated with the 'name-less' associative retrieval method of LINDA. If this were achieved, then it might be possible to dispense with ACLs altogether and use capabilities for tuples.

References

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[2] G. Agha and C.J. Callsen. ActorSpace: An Open Distributed Programming Paradigm. *SIGPLAN Notices*, 28(7):23–32, 1993.

[3] B.G. Anderson and D.Shasha. Persistent Linda: Linda + Transactions + Query Processing. In J.P. Banatre and D. Le Metayer, editors, *Proceedings of Research Directions in High–Level Parallel Programming Languages*, volume LNCS-574, pages 93–109. Springer, 1992.

[4] F. Arbab. The IWIM Model for Coordination of Concurrent Activities. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models*, volume LNCS-1061, pages 34–56. Springer-Verlag, 1996.

[5] A. Bestavros. WWW Traffic Reduction and Load Balancing through Server-Based Caching. *IEEE Concurrency*, 5:56–66, 1997.

[6] W.J. Bolosky, R.P. Draves, R.P. Fitzgerald, C.W. Fraser, M.B. Jones, T.B. Knoblock, and R. Rashid. Operating System Directions for the Next Millennium. http://www.research.microsoft.com/research/os/Millennium/mgoals.html, 1996.

[7] N. Carriero and D. Gelernter. *How to Write Parallel Programs: A First Course.* MIT Press, 1990.

[8] N. Carriero and D. Gelernter. Coordination Languages and their Significance. *Communications of the ACM*, 35(2):97–106, 1992.

[9] D. Edwards. Genetic Algorithms in LINDA. Technical report, University of York, Department of Computer Science, 1997.

[10] D. Gelernter. Generative Communications in LINDA. *ACM Trans. Programming Languages and Systems*, 7:80, 1985.

[11] J.S. Gwertzmann and M. Seltzer. The Case for Geographical Push-Caching. In *Proc. 5^{th} Workshop on Hot Topics in Operating Systems*. IEEE Press, 1995.

[12] S. Jagannathan. Customization of First-Class Tuple-Spaces in a Higher-Order Language. In E.H.L. Aarts, J. van Leeuwen, and M. Rem, editors, *Proc. PARLE'91: Parallel Architectures and Languages Europe*, volume LNCS-506, pages 254–276, 1991.

[13] R. Menezes and A. Wood. Garbage Collection in Open Distributed Tuple-Space Systems. In *Proc. 15^{th} Brazilian Computer Networks Symposium — SBRC'97*, 1997.

[14] R. Menezes and A. Wood. Incorporating Input/Output Operations in Linda. In *submitted to 30^{th} Hawaii International Conference on System Science*, 1998.

[15] N.H. Minsky and J. Leichter. Law-Governed Linda as a Coordination Model. In P. Ciancarini, O. Nierstrasz, and A. Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems*, volume LNCS-924, pages 125–146, 1995.

[16] S.J. Mullender, G. van Rossum, A.S. Tanenbaum, R. van Renesse, and H. van Staveren. Amoeba: A Distributed Operating System for the 1990s. *IEEE Computer*, 23:44–53, 1990.

[17] J.L. Petersen and A. Silberschatz. *Operating System Concepts.* Addison-Wesley, 1985.

[18] J. Pinakis. Providing Directed Communication in LINDA. In *Proc. 15^{th} Australian Computer Science Conference*, pages 731–743, 1992.

[19] A. Rowstron and A.M. Wood. An Efficient Distributed Tuple Space Implementation for Networks of Workstations. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *EuroPar'96*, volume LNCS-1123, pages 510–513. Springer-Verlag, 1996.

[20] D. Simpson. Who Needs a Network Computer. *Datamation*, 42(16):96–99, 1996.

[21] A.S. Tanenbaum. *Modern Operating Systems.* Prentice Hall, 1992.

[22] A.S. Tanenbaum. *Distributed Operating Systems.* Prentice Hall, 1995.