Linda$_m$ and Tiamat: Providing Generative Communications in a Changing World

by

Gareth P. McSorley

A thesis submitted to the
Faculty of Information and Mathematical Sciences
at the University of Glasgow
for the degree of
Doctor of Philosophy

January 2006

If I could only be half the man you were,
I would be twice what I am now.
But, even that which I am, I owe to you,
And your eternal kindness and love.

For Grandpa

Abstract

When generative communications, as exemplified by Linda [Gel85], were originally proposed, they were intended as a mechanism for coordination of parallel processes. Since that time, they have been adapted to a variety of distributed environments with great success, as can be seen in commercial systems such as T Spaces [WMLF98]. The time, space and identity decoupling afforded to coordinating entities by generative communications also seems to be ideally suited to mobile environments where devices can come and go frequently and often without warning. Such a rapidly changing environment, however, presents a new set of challenges and attempts to introduce the generative communications paradigm into these environments have, so far, met with limited success. Indeed evaluation of research platforms, such as LIME (**L**inda **I**n a **M**obile **E**nvironment) [PMR99,MPR01] and L^2imbo [DFWB98] have led some to conclude that the generative communication paradigm is not well suited to mobile environments.

It is my belief, however, that it is the research platforms in question, rather than the paradigm, which do not fit well with mobile environments. These platforms either attempt to impose tight constraints on an inherently loosely constrained environment, or require significant alterations to the semantics of generative communications. I believe that these systems do not work well as they are not designed around the environment, rather they are forced onto the environment. I will begin by examining why these systems do not suit their environment. This done, I will then show that the conclusions drawn from these systems, namely that generative communications are unsuitable for mobile environments, are incorrect. Further, through construction and examination of a proof of concept system built around an environment-centric design, I will show that generative communications can be provided in a mobile environment with few (minor) semantic alterations. An evaluation of some of the mechanisms used will also be presented along with characterisation of the operation of the system. A comparison with existing mobile solutions will be used to highlight how the environment-driven design results in a system which better suits the nature of the target environment.

Acknowledgements

There is a plethora of people who have assisted, helped, encouraged, supported or just plain forced me through the course of undertaking this work. I am grateful to each and every one, whether I have remembered them as I write this list or not.

I would like to thank Dr. Huw Evans for his continued support, boundless "wisdom", for always being willing to participate in a good old fashioned rant and for introducing me to the single most disgusting word in the English language.

Jana Urban deserves endless credit for putting up with me while doing my corrections, as well as being a dab hand with a drawing package or two.

I would like to extend my sincerest gratitude to Dr. Peter Dickman for being my supervisor and for supporting me through my work. I am extremely grateful for his help and guidance when I found myself without supervisor, topic or direction.

I would also like to thank Dr. Karen Renaud for acting as my Second Supervisor.

I am grateful to Dr. Iain "Del-Boy" Darroch for being a good friend as well as a great officemate and for his limitless knowledge of useless sporting information.

I am grateful to (in no particular order): Dr. Michael Dales for his friendship, his scathing wit, and for showing everyone how a PhD *should* be done; Dr. Andy King for some truly wonderful turns of phrase and associated imagery; Dr. Tony Printezis for his invaluable advice and for making me feel less of a geek; Dr. Rolf Neugebauer for dispelling all my beliefs about healthy living; Lyndell St. Ville for redefining "laid-back" in the way only someone from a sun-drenched island can; Ji-Hyang Lee for all the coffee; Jonathan Paisley for possessing intimidating amounts of technical knowledge; and to my flatmate Gregory "Stress Bunny" Huczynski for stressing so much that no one else has to, may he one day be convinced.

I would like to extend my sympathies to my temporary officemates Dr. Nigel Harding and Susan Fairley — I'm impressed they managed to stick it out as long as they did.

I would like to thank the Engineering and Physical Sciences Research Council — without their financial input, this work would never have been possible.

Thanks also go to everyone at Little Italy, Tinderbox, The Atrium, Naked Soup, Domino's Pizza and Pizza Hut for keeping me fed and caffeinated.

Finally, I am, and always will be, grateful to my parents and family for their continued love, kindness and support.

Contents

List of Figures

List of Listings

Chapter 1

Opening Statements

The past few years have seen a growth in the number of devices available whose purpose is to provide computing power on the move. With advances in technology, this trend looks only set to increase. With these devices have come a wide and varied number of wireless networking technologies which allow them to communicate with fixed infrastructure and with each other. Developing software for this mobile space, however, presents application developers with a substantial amount of change in the system which must be accounted for and managed.

In order to relieve the application developer of some of the burden of managing this change, middleware that can model the change through some abstraction is needed. Providing an accurate system which abstracts over some elements of change is necessary to help manage the changing world. The middleware then becomes responsible for the detail of which devices or resources are available at a given time and leaves the application developer to develop applications. This work describes the issues and challenges of using the generative communications paradigm for providing that middleware. Generative communications involve processes communicating and coordinating with one another through collections of data called tuples. Tuples are placed in, and retrieved from, an independent shared memory known as a tuple space. The generative communications approach offers participating entities the advantages of being decoupled in time, space and identity. These decouplings help to address some of the issues involved in a mobile environment.

1.1 Thesis Statement

Generative communications were originally designed for the coordination of parallel processes. However, they have also found a home in a variety of distributed environments including environments involving mobility. Much of the research carried out in these environments has been problematic and has led some to conclude that generative communications are unsuitable for such mobility-oriented situations. I believe, however, that this is incorrect and is more a reflection on the systems used in this research than of the suitability of the approach. I will demonstrate how previous research platforms have been unsuitable for mobile environments. I will furthermore propose a model and construct a proof of concept implementation to demonstrate that, with some minor semantic alterations, the generative communications paradigm can be provided in a mobile environment. I will measure and examine the characteristics of the operation of such a system and will compare the system to existing research to demonstrate that an environment-centric design results in a system which is better suited to the defined mobile environment.

1.2 Contributions

The main contributions of this work are:

- The proposal of a novel model, Linda$_m$, for providing Linda-like semantics in a mobile environment.
- A proof of concept implementation, Tiamat, of that model to demonstrate viability and operability.
- Demonstration of the viability of Tiamat as a communications platform.
- Experimental evaluation of the characteristics of the Tiamat system.
- A demonstration of the value of a tuple space system in a mobile environment.
- Highlighting of an often overlooked problem that is exacerbated in a mobile environment (distributed consensus).
- A comparison of the new model and implementation with previous work, highlighting previous systems' unsuitability for mobile environments.

1.3 Document Outline

The remainder of the dissertation is organised as follows: Chapter 2 sets the context for the work as well as motivating the need for such an infrastructure. Chapters 3 and 4 outline the related work with chapter 4 taking a particular focus on those Linda systems which have previously attempted to operate in mobile environments. Chapter 5 describes the Linda$_m$ model for providing generative communications in a mobile environment with chapter 6 then discussing Tiamat, an implementation of that model. Chapter 7 evaluates the model and implementation through comparison with existing research. Chapter 8 follows on from this to provide quantitative evaluation of Tiamat through experimentation. Chapter 9 presents some future avenues for extension of this work. Chapter 10 provides a summary of and concludes this dissertation.

Chapter 2

Context and Motivation

This chapter provides an introduction to the context in which the dissertation is set. The chapter begins with an examination of the trend toward a more pervasive and mobile computing environment in section 2.1. There then follows an introduction to the traditional Linda system and semantics in section 2.2. A more detailed description of the proposed environment is presented in section 2.3 followed by a discussion of how the Linda model can be used to provide coordination in such an environment in section 2.4.

2.1 An Increasingly Mobile Environment

Recent years have seen a growing trend toward mobile computing. As technology advances, computing devices have become smaller and more powerful. Modern mobile phones are now capable of much more than just making phone calls and can perform many of the complicated tasks that were traditionally the domain of PDAs and computers. The most recent phones are even capable of running Java applications and processor-intensive games [Nok04]. With their increasing miniaturisation, computers are finding many new niches to occupy, from watches [NKR+02] to smart clothing [Man96]. Along with the pervasion of computing power has also come a pervasion of networking capabilities. Wireless protocols such as Bluetooth™ [Kar00] and 802.11 [OP99] have allowed portable devices to make use of existing networking infrastructure, or even to form ad-hoc networks of their own. This trend seems set to continue with the IEEE publishing details of a range of networking protocols for use in such devices [HG99, OP99, Swe04], allowing for a wide variety of types and scales of networks.

For application developers, however, this move toward mobile computing presents some

potential difficulties. With the increase in mobile networked devices comes a corresponding increase in the amount of change that a system will experience during its operation as the various devices come and go. Managing this potential change (i.e. maintaining a view of what resources/devices are present over time) is likely to prove time-consuming for application developers, thereby detracting from their focus on the application itself. Middleware which manages the change, removing from the client the need to determine which devices or resources happen to be available, is needed to once again allow the application developers to focus on developing interesting applications.

This dissertation focuses on one approach to providing such middleware which shows great promise, called generative communications. Generative communications provide an asynchronous coordination mechanism which are exemplified in a system called Linda [Gel85].

2.2 A Brief Primer on Linda

This section presents an overview of Linda, the canonical example of generative programming. The core concepts of the Linda system are described together with descriptions of the operational semantics. Readers who are familiar with Linda can skip this section.

The Linda system is based on three core concepts: *tuples*, presented in section 2.2.1; *tuple spaces*, presented in section 2.2.2; and *anti-tuples*, presented in section 2.2.3. This is followed by a description of the decouplings offered by Linda in section 2.2.4. Finally, there is a description of some types of application to which Linda is well suited in section 2.2.5.

2.2.1 Tuples

A tuple is an ordered collection of data of arbitrary type. An example of a tuple would be:

<"Andy", 27, Bald>

This example tuple has three fields: the first is the *actual* (or value) "Andy"; the second is the actual 27; and the third field is the actual Bald. As well as actuals, the fields of a tuple can contain *formals* which can be thought of as wild-cards for the appropriate types. For example, consider the following tuples:

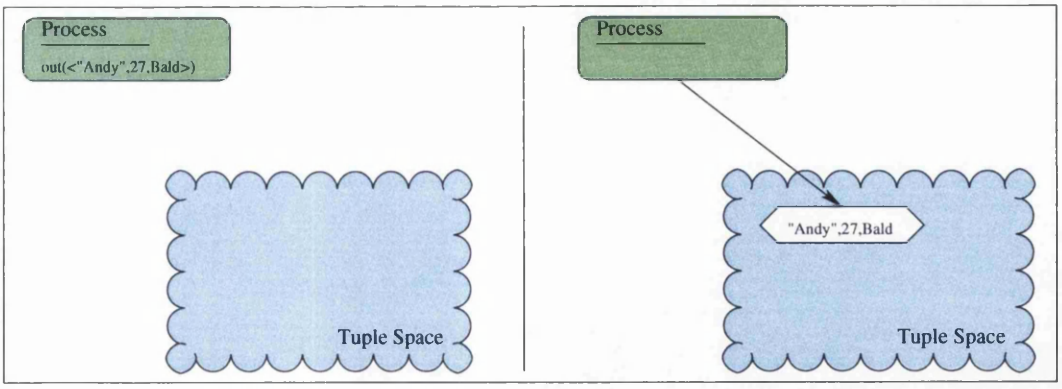


Figure 2.1: An **out** operation.

<"Andy", *Age*, Bald>

<"The Complete Robot", "Asimov, I", *ISBN*>

The first tuple contains two actuals, "Andy" and Bald and a single formal, *Age*. The second contains the two actuals, "The Complete Robot" and "Asimov, I" along with the formal *ISBN*. The role of actuals and formals will be elaborated upon in section 2.2.3 when anti-tuples and retrieval operations are discussed.

2.2.2 Tuple Spaces

Processes coordinate by placing tuples into, and retrieving them from, a tuple space, which behaves as an unordered bag. In the Linda originally proposed for parallel systems, the space would be an area of shared memory specifically set aside for that purpose. In distributed versions of Linda the space will most likely reside on a single server machine; nonetheless, its purpose and operation remain the same.

Linda provides two operations which can be used to populate the tuple space: **out**; and **eval**. The **out** operation[1], which can be seen in figure 2.1, takes a tuple and places it into the tuple space where it will be available to other processes.

The **eval** operation is used to place *active* tuples into the tuple space. Active tuples contain, in place of one or more actuals or formals, some calculation or piece of code which must be performed in order to obtain a normal (i.e., non-active) tuple. For example, in figure 2.2 a process can be seen placing an active tuple into the space. It is the responsibility of the Linda system, not the interacting processes, to perform the necessary calculations and place the resultant tuple into the space. Active tuples cannot be retrieved from the

[1]To begin with, the names of the operations can seem a little counter-intuitive. It often helps to think of the operations from the perspective of the process making use of the space.

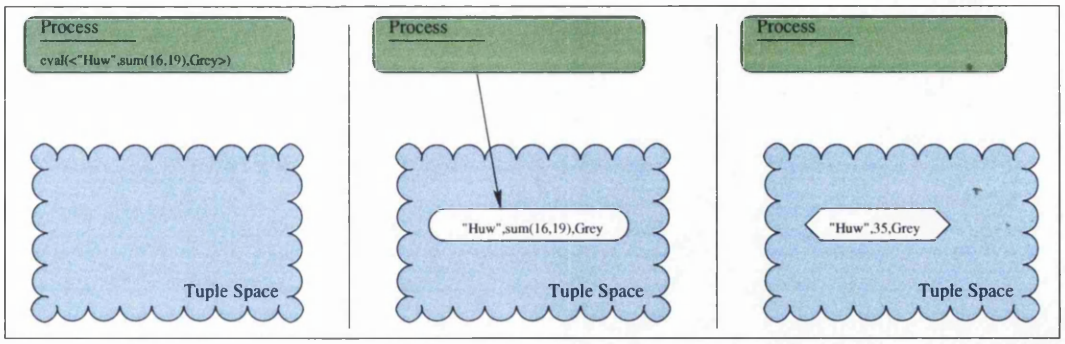


Figure 2.2: An **eval** operation.

space, only the resultant tuple can be retrieved.

Whichever mechanism is used to place the tuples into the tuple space, they can only be accessed through the use of anti-tuples.

2.2.3 Anti-Tuples

An anti-tuple is a tuple which is used as part of a retrieval operation. As such, anti-tuples consist of an ordered collection of actuals or formals. During retrieval operations (described below) anti-tuples are compared to tuples looking for a match. An anti-tuple is said to match a tuple if the following conditions are true:

1. Both anti-tuple and tuple have the same number of fields.
2. Each field in the anti-tuple has the same type as the corresponding field in the tuple.
3. If a field in the anti-tuple contains an actual, then the corresponding field in the tuple must contain either an identical actual or a formal.

Some examples of matching can be seen in figure 2.3.

There are two operations which are used to retrieve tuples from the tuple space: **in** and **rd**. Both operate in a similar fashion and both operations take an anti-tuple as a parameter. The anti-tuple is compared to the other tuples in the tuple space. If a matching tuple is found it is returned to the process that initiated the operation. If the space contains more than one matching tuple, one is selected for return in a non-deterministic manner. The **in** operation is called a *destructive* read because, if a match is found, the matching tuple will be removed from the space, as shown in figure 2.4. The **rd** operation is called a *non-destructive* read as it takes a copy of the matching tuple leaving the original in the space, as shown in figure 2.5. In both cases, if no match is found then

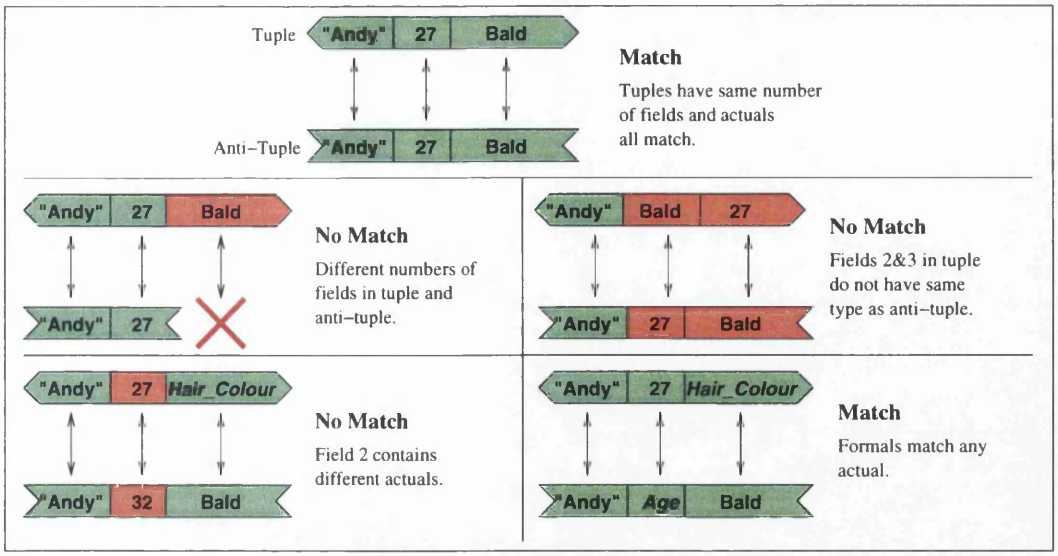


Figure 2.3: Some matching examples.

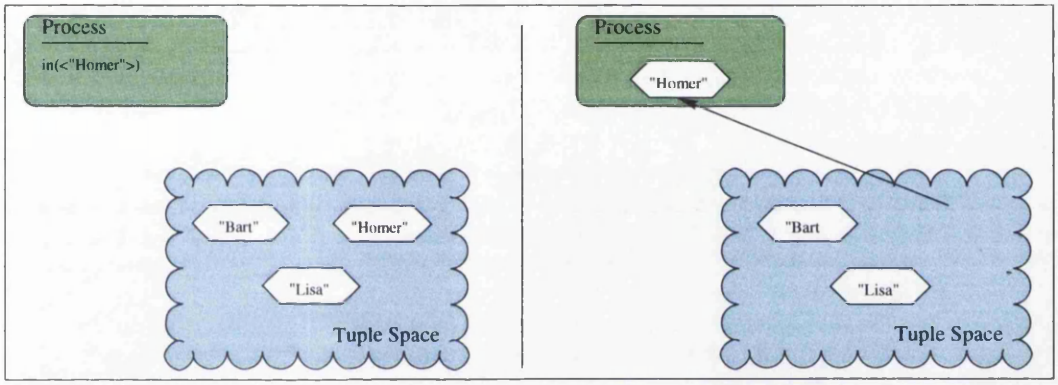


Figure 2.4: An **in** operation.

the operation will block until a matching tuple is placed into the space, as shown in figure 2.6. Note that there is no upper limit on how long these operations will block awaiting a match.

There are also probing versions of the **in** and **rd** operations which do not exhibit this blocking behaviour: **inp** and **rdp**. These will scan the tuples in the space as before. If a match is found it will be returned (and the original removed from the space in the case of **inp**). If no match is found, however, the operations will not block and will instead immediately return a null tuple.

2.2.4 Decoupling

The Linda system offers three forms of decoupling: *space*; *time*; and *identity*. Interactions through the tuple space are decoupled in terms of physical space as two coordinating en-

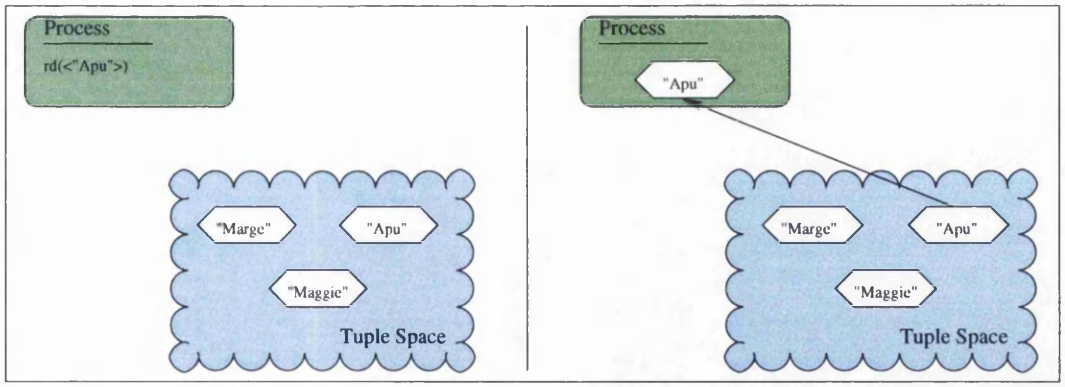


Figure 2.5: A **rd** operation.

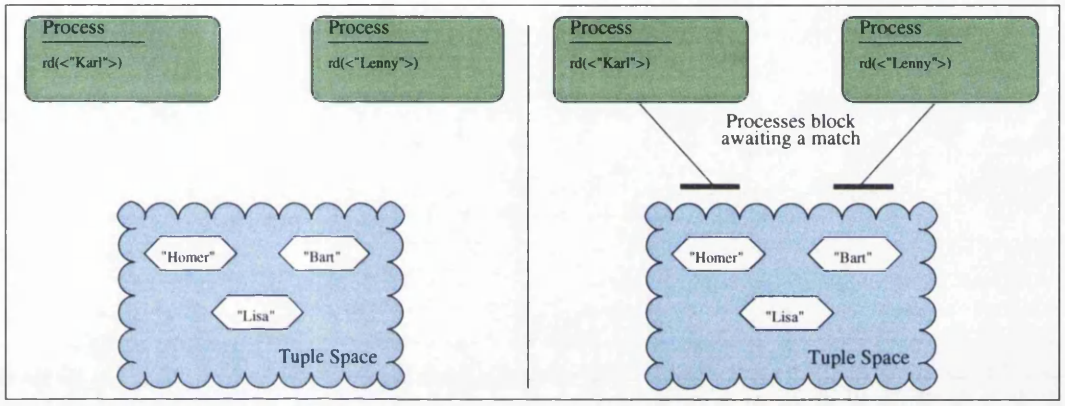


Figure 2.6: Blocking operations.

tities need not be co-located in order to make progress (although they must both have access to the tuple space at some point). The tuple space is an asynchronous communication mechanism and so also allows decoupling in time – two coordinating entities need not be connected to the tuple space, or even exist, at the same time in order to make progress. Finally, since all interactions take place through the tuple space, coordinating entities need not be aware of which other entity they are coordinating with, only that they are coordinating with someone. This undirected and anonymous form of communication will be termed "identity decoupling" in this dissertation. As will be shown in section 2.4 it is these decouplings which make Linda desirable in a mobile environment.

2.2.5 Applications

The Linda paradigm is a versatile coordination mechanism and can be used in a wide variety of situations. There are, however, some types of application to which Linda is ideally suited. The list that follows is adapted from [Cro00]. This list is by no means comprehensive, nor does it indicate that Linda is the *only* paradigm which can be used in

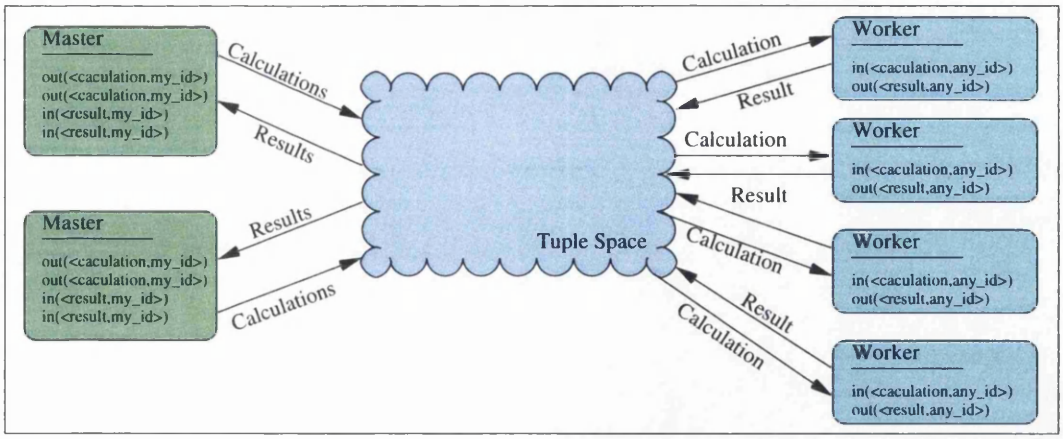


Figure 2.7: Replicated worker example.

these situations, merely that it is well suited to such situations.

Replicated Worker

The replicated worker (or master/worker) application is one in which a single master node wishes to have some operation or calculation performed by a number of worker nodes. The use of a tuple space fits naturally into this problem. The master node places tuples containing data which encapsulate the calculations that need to be performed. The workers retrieve these tuples from the space, perform the calculation and then wrap the results in a new tuple which is placed in the space and retrieved by the master. A simple example of the replicated worker application can be seen in figure 2.7.

This kind of application benefits greatly from the decouplings offered by Linda. The decoupling in identity means that the master does not have to know how many workers are available or how to identify them, this is dealt with through the space. The decoupling in time and space mean that the workers need not exist, nor be connected to the tuple space, at the same time as the master in order for work to be done.

Command

Command is a specialisation of Replicated Worker in which the actual code required to perform the calculations is also embedded in the tuples. In this instance the workers become generalised "dumb computers" which execute any block of code they are given.

This application benefits from the use of Linda in the same way as the replicated worker.

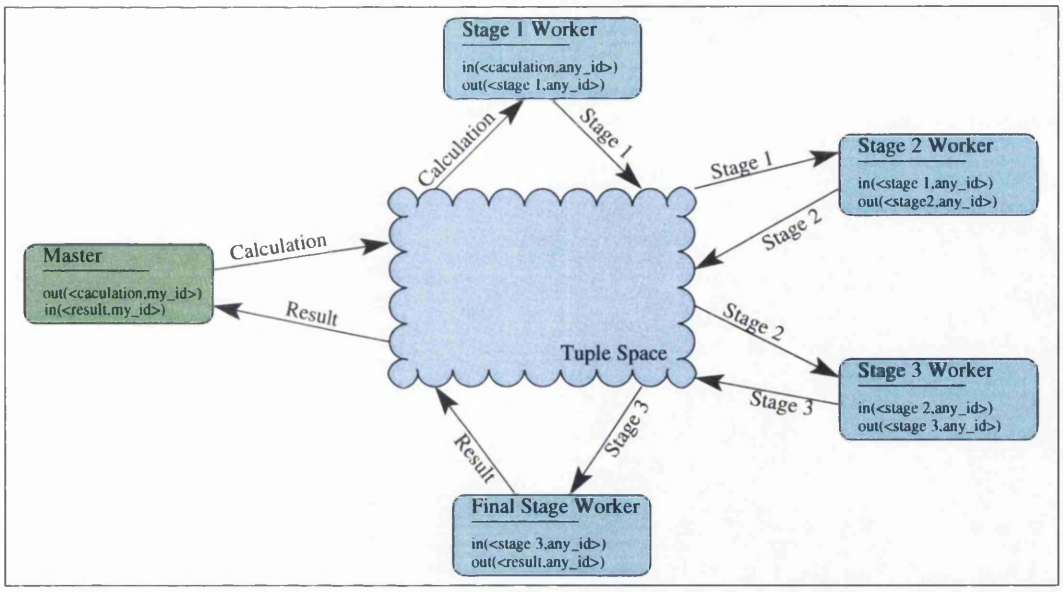


Figure 2.8: Trellis example.

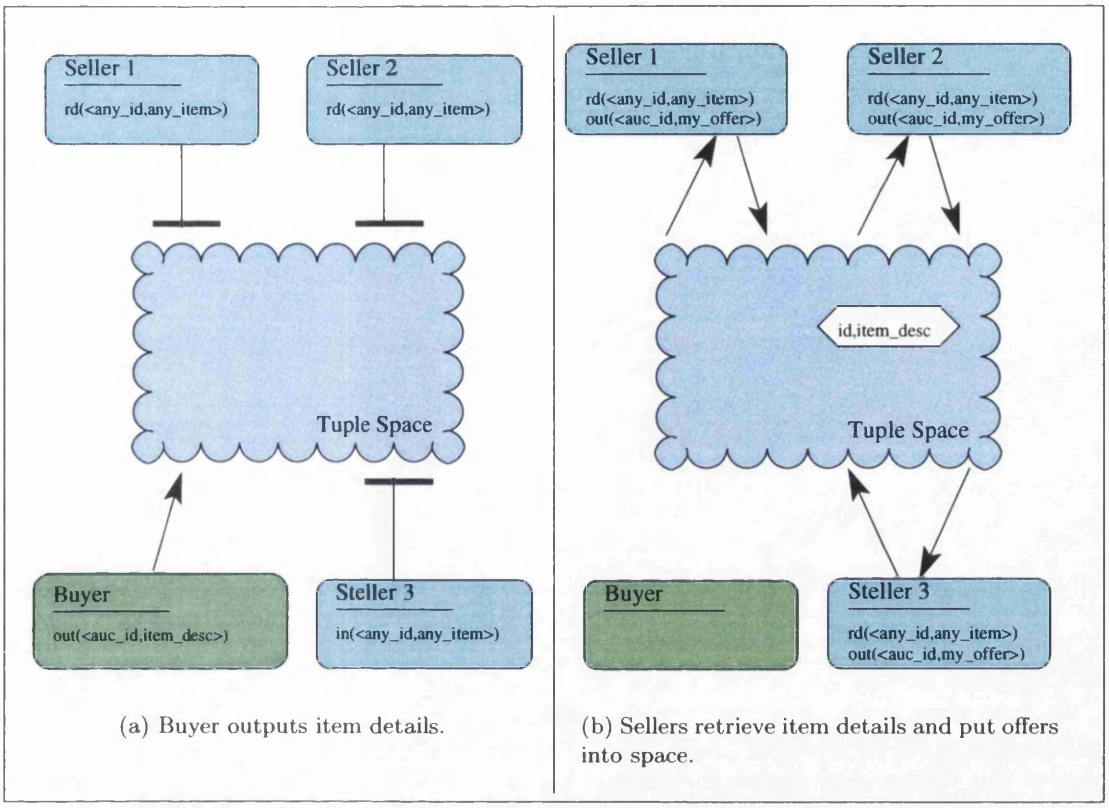
Trellis

The trellis is a multilayer form of the replicated worker in which the workers make up a chain, with each one taking the output from the previous node and performing some subsequent computation on it. Only once the data has passed through all of the layers is it retrieved from the space by the master node. This can be seen in figure 2.8.

The trellis benefits from the use of a tuple space in much the same way as the replicated worker. The decoupling in identity, however, has added benefits for a trellis as it allows for a separation of concerns when constructing the workers. Each worker has only to know the form of the tuple output by the previous layer and need know nothing about the layer above. This allows for extra layers to be added to the trellis with relative ease as none of the workers for the layers below the insertion point need be modified.

Marketplace

Marketplace problems operate much like an auction but in reverse. In a normal auction a seller would put an item up for auction. This item is then bid on by buyers until the close of the auction at which point the highest buyer has won. In a marketplace or reverse auction, the buyer places a description of the desired item into the tuple space (figure 2.9(a)). This description tuple is retrieved by the sellers who then decide what amount they would be willing to sell this item for. Each seller then places a bid acceptable to them into the space (figure 2.9(b)). Once the auction has finished, the buyer collects



(a) Buyer outputs item details.

(b) Sellers retrieve item details and put offers into space.

Figure 2.9: Marketplace example.

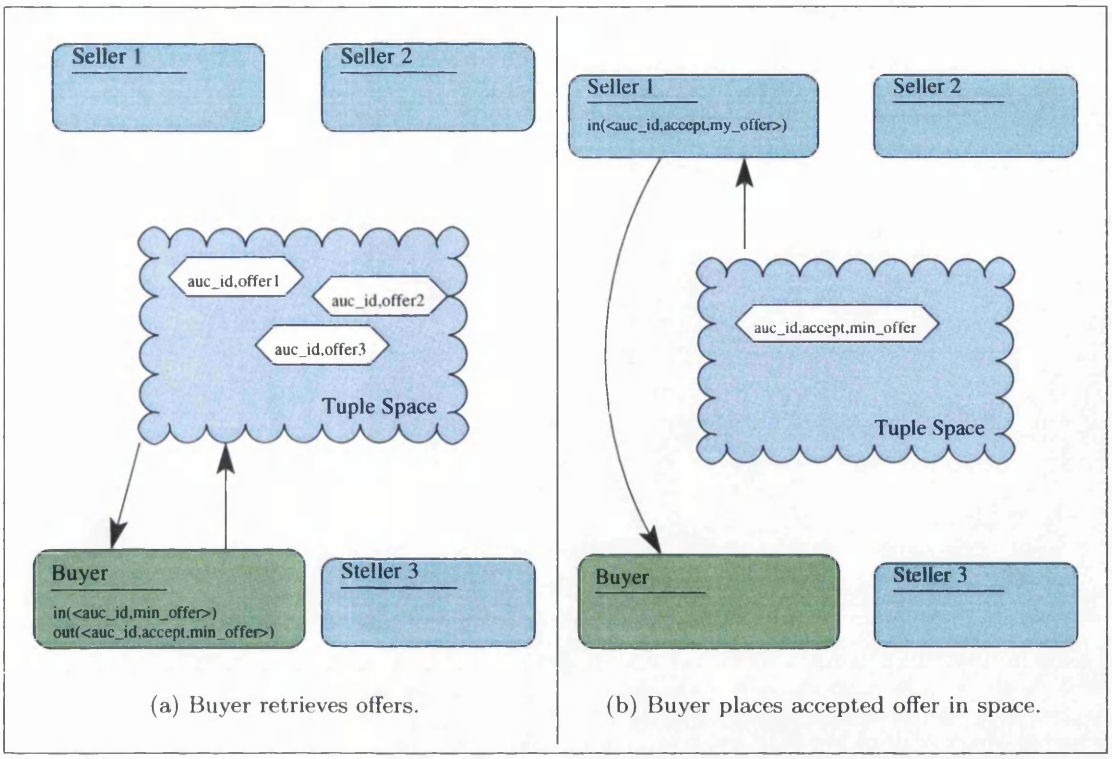
all the offers from the space and decides which bid it is willing to take (figure 2.10(a)). This bid is put back into the space where it is retrieved by the appropriate seller (figure 2.10(b)).

The marketplace benefits from the use of the tuple space as the buyer does not have to manage the acceptance of all the bids, this is dealt with through the space. This also means that the buyer process could initiate the auction and then go to sleep, or detach, only returning at the end of the auction and reading the bids. The buyer does not have to remain active for the duration of the entire auction. The identity of the buyer and of the sellers can also be kept secret through the space until the auction is completed.

2.2.6 Summary

The Linda system provides a coordination mechanism based around a tuple space which is an unordered bag of tuples. Six operations on the space are provided: **out** and **eval** are used to populate the space with tuples; **in**, **inp**, **rd** and **rdp** are used to retrieve tuples from the space based on a pattern match with a provided anti-tuple.

The tuple space provides three forms of decoupling to processes: space, two processes



(a) Buyer retrieves offers.

(b) Buyer places accepted offer in space.

Figure 2.10: Marketplace example (cont.).

need not be co-located to coordinate; time, two processes need not be active at the same time to coordinate; and identity, processes need not know the identity of the process they are coordinating with in order to make progress.

Before examining the value of Linda in a mobile environment, the next section will examine the various characteristics which define such an environment.

2.3 Environment

Following on from the discussion in section 2.1, it can be seen that the environment of interest in this work consists of a large collection of heterogeneous computing devices and platforms that will range from small computers embedded in clothing up through mobile phones and PDAs to powerful desktop and server machines. These devices vary in terms of three primary characteristics: *resources*; *connectivity*; and *mobility*. This section highlights the environment under consideration in which these devices operate. Examination of the core characteristics of this environment are discussed in turn in sections 2.3.1, 2.3.2 and 2.3.3 before looking at the overarching nature of the environment in section 2.3.4.

2.3.1 Resources

Resources is a collective term describing the facilities, services and consumables which are available to a device. In this environment devices will possess resources of varying type which can be split into three categories: *consumables*; *recyclables*; and *services.* Consumables are resources which once used cannot be recovered by the device. Examples of consumables include battery life[2], CPU cycles and even money (if, for example, other resources charged for their usage). Recyclables are resources which once used may be recovered at some point in the future. Typical examples include backing storage, active memory space and screen real estate. Whereas recyclables and consumables represent the actual resources themselves, services represent a higher level of abstraction over a piece of software, hardware or some combined functionality provided to local applications or to other devices. Services will typically rely upon and consume one or more consumables or recyclables. Each recyclable or consumable may be used by multiple services. For example, memory (a recyclable resource) may be consumed when providing the services of persistence (to provide in-memory caching) or a graphical user interface (to store the window contents).

As well as possessing different types of resources, devices will possess varying numbers of resources, which refers to the amount of distinct resources possessed by the device (note that number is not the same as quantity, see below). For example, a server machine will possess a large number of resources, such as a backing store, an active memory, the capacity for processing, and a host of software services, such as persistent storage, network routing or file-sharing. Other devices will possess relatively few resources, for example, a simple PDA might only possess an active memory (which doubles as a backing store), some capacity for processing and a display.

Resources present on devices will also be of varying quantity. Where number described the amount of distinct resources possessed by the device, quantity relates to the amount of those resources available. Some devices will possess copious (albeit still finite) amounts of resources, such as server arrays with gigabytes of memory, terabytes of backing storage and multiple high-performance CPUs. Others will be resource impoverished, such as PDAs and mobile phones possessing a few megabytes of total storage and a single, low-end CPU.

Finally, resources will vary in terms of quality. The active memory on a server machine

[2]Even though batteries can be replaced or recharged, once battery power is consumed it cannot be recovered *by the device*, rather it requires outside intervention – hence its classification as a consumable.

will have fast access times and be connected by high-throughput buses, while in comparison the memory in a PDA will be slow to respond and have a more limited bus capacity.

2.3.2 Connectivity

Connectivity refers to a device's ability to connect to other devices in the system. Connectivity will vary in terms of *availability*, *performance* and *reliability*.

Availability describes how often a particular device is likely to be contactable. Some devices will be highly available, maintaining a near-permanent connection to a wide area network, while other devices connect only intermittently and only form a small, isolated, ad-hoc network with a small set of other devices. A device may have a very high performance satellite connection, but will only connect briefly, once a day due to the high power demands. The 3G (UMTS) network [KN05] is similar, its high performance is let down by poor coverage at present, whereas the GSM network [Har04] has very low performance (9.6 kbps) but the more pervasive infrastructure makes it far more available. Availability can depend on cultural, environmental, behavioural and technological factors and not simply the network technology available.

Performance describes the theoretical capabilities of the network and encompasses a variety of individual properties, for example, latency and bandwidth.

Reliability is a description of how well a connection performs in reality. As with availability, reliability also depends on external factors. An 802.11g [OP99] wireless network is capable of 54Mbps transfer rate, but if the access point is on the other side of a metal wall your maximum transfer rate is likely to be much lower.

Connectivity varies over the whole spectrum of devices, from well connected server machines with their high-bandwidth always-on Internet connections, to PDAs which may have only short-range, low-bandwidth connections such as Bluetooth [Kar00] and which may depend on other devices — such as a nearby mobile phone — to extend their connectivity. The connectivity of a device may even vary over time, for example, a laptop which is connected to a high-speed Ethernet connection (low latency, high bandwidth) may, at a later time, be operating through an analogue phone line and modem (high latency, low bandwidth). Much of a device's connectivity will depend on the resources available to it, however, external factors may also play a part.

2.3.3 Mobility

Mobility refers to the tendency of a device to move around in the physical world and can be defined in terms of *frequency*, *duration* and *promiscuity*. Frequency refers to how often the device is likely to move, from the mobile phone which is frequently carried by its owner to new locations, to the 80kg+ server array which may move only a couple of times in its entire life. Duration is how long the device is likely to remain at a given location (it is usually the inverse of frequency, but can be separate, e.g. a device which only very rarely connects to a network would have low frequency and low duration). Promiscuity describes how often these devices will visit new locations. Devices with low promiscuities may only move between two locations, such as a laptop being carried between work and home, while other devices with a higher promiscuity may rarely see the same location twice, for example, an on-board computer in a taxi. It is important to consider what drives the motion of these devices. While not excluding the possibility that some of these devices may be entirely autonomous (an Unmanned Aerial Vehicle [DGK+00], for example), the majority are carried by humans. Even those which are not carried by humans (e.g., in-car computer systems) are still likely to have their motion driven by the movement of humans. This presents a potential avenue for further research which is discussed in section 9.3.

2.3.4 Change Pervades

It is expected that the majority of devices in the environment will be the equivalent of modern PDAs, phones and laptops. While this may not be the situation at the moment, it is not hard to imagine a situation in the near future where each person may account for a mobile phone, a PDA, an on-board computer in their car, some smart clothing, probably a laptop, a couple of desktops and a share of some servers. This results in the majority of devices in the environment possessing a relatively small number of low quantity resources. Their connectivity will be defined by mid to low availability, reliability and performance. Their mobility will generally be high frequency with varying promiscuities.

All of the above factors combine to produce one overriding characteristic for this environment - *change*. This is not a static setup where homogeneous nodes attach to some well structured network and then remain there, barring any failure. Resources will constantly be in the process of being consumed and recycled. Devices will move frequently and unpredictably. This movement will carry them from areas of high connectivity to areas of isolation and back again. The set of available services will change as devices come and go.

Change pervades this environment and it is important that any model designed to operate within it considers change as a normal component of the system operation, rather than as an exception.

The notion of change as an integral component within mobile environments has also been identified in previous research [KMS+93, YJK98].

2.4 Mobility and Linda

The decoupling in space, time and identity which have made Linda useful in parallel and distributed environments also make it a desirable paradigm for mobile environments. In such a malleable environment it is difficult, if not impossible, for two devices to guarantee co-location for sufficient time to coordinate effectively. The decoupling in space and time offered by Linda helps to alleviate this problem by allowing two processes to coordinate even if they are never in the same place at the same time. Decoupling in identity will allow a device to move around and make use of any resources and services which happen to be available. This can be done without the need for the node to maintain a comprehensive list of which nodes provide which services in which locations. By using the tuple space the node will be able to coordinate with other nodes as they become available.

The desired operational platform for Linda in a mobile environment is as follows. Co-located nodes will share a tuple space through which they can coordinate using the basic operations. When a node becomes disconnected from the shared tuple space it should still be able to perform tuple space operations, so that coordinating applications located on the device can still make progress. When the node comes into contact with other nodes any tuples inserted into the shared tuple space during its isolation should become available to it. Also, any outstanding **in** or **rd** operations should check the newly available tuples for possible matches.

2.4.1 Examples

In order to outline how tuple spaces could be used in a mobile environment, and to highlight the advantages of this approach, it is useful to examine potential applications.

Web Proxy/Client

In order to simplify the management of a shared Internet connection, and to allow for monitoring or security, it is common practise to use a proxy server to provide Internet access. Rather than contact the other Internet hosts directly, all requests are given to the proxy server which contacts the remote hosts on the client's behalf. This is an example of the traditional client/server model where a single server machine, the proxy, is responsible for handling the requests of many clients.

Even in a static environment this model has some issues, foremost of which is the fact that the server acts as a bottleneck and a single point of failure. Although it is possible to provide static load balancing by providing multiple servers and using some scheme to spread the clients over them, dynamic load balancing is more troublesome. It either requires modification of the clients, or provision of a load balancing server which assigns the requests to clients. Modification of the clients is costly in terms of development time and resources. The provision of servers will provide better performance than doing the load balancing in software, but this creates a new bottleneck at the load balancing server. Load balancing can also be provided in the network routing hardware as well. Routers tend to be built on dedicated hardware with specialised operating systems and software and so will likely provide very high performance. Also, since this load balancing is now simply another facet of the network, the system is more resilient to failure[3] and as long as network communication is possible, then load balancing will take place. However, such routers tend to be quite expensive, are unlikely to be available in ad-hoc environments and their configuration may not be accessible to the system developer.

In a mobile environment other undesirable qualities emerge. Firstly, any mobile clients must be informed which machine acts as the proxy (or if one is needed at all), requiring the provision of a standardised discovery mechanism. Secondly, the traditional model requires that the client remain connected to the server for the duration of the exchange. In a mobile environment this may not be feasible.

When a mobile version of Linda is used to support this application, it becomes less of a client/server model and more of a master/worker model. The clients now act as masters, wrapping their requests into tuples which are then placed into the space. The proxy server becomes a worker, which takes request tuples from the space and contacts the appropriate

[3]Note that this depends on the network being well architected. If all of the servers connect to the same router, then there is still a single point of failure as before.

remote host. The result of the request is wrapped in a tuple and placed back into the space.

By using the tuple space for coordination, the bottleneck of the proxy has been removed. Instead, multiple servers can be started, each of which will retrieve request tuples from the space. Should one fail, another will take over transparently. Load balancing can also be dealt with by starting new proxies.

Discovery mechanisms for the proxies are no longer needed thanks to the decoupling in identity, as the client need not be aware of which host holds the proxy. A client places its request into the space and awaits a result. Also, intermittent connectivity is addressed by the decoupling in time and identity. Even if the client is not present when the result is placed into the space, it can retrieve it later.

Fractal Generation

The distributed fractal generator is one of the canonical examples of the master/worker architecture. While fractal generation specifically may not be a common requirement in a mobile environment, the more general pattern of master worker is, as it allows potentially resource impoverished devices to benefit from the collective resources of others. The fractal generator is presented here as an exemplar of this type of application and the benefits it can bring.

Fractal calculations are specified by one or more master nodes and then performed by some number of worker nodes. This arrangement places the burden of managing the workers, ensuring that each is kept active and is not overloaded, on the masters. This can prove difficult in the situation where multiple masters are using the same set of workers. For this reason such systems are often built with a load balancing server which manages the workers.

As with the previous example, there is a bottleneck and single point of failure in the load balancing server. Furthermore, in a mobile environment there may be a substantial amount of work necessary to keep track of the workers as they come and go.

This problem naturally maps to the tuple space paradigm. The masters wrap the required calculations or operations into tuples which are then placed into the space. At some point in the future workers will retrieve these tuples, perform the calculation, wrap the result in another tuple and place it back into the space for retrieval by the master.

This arrangement offers many of the same benefits as in the web client and proxy

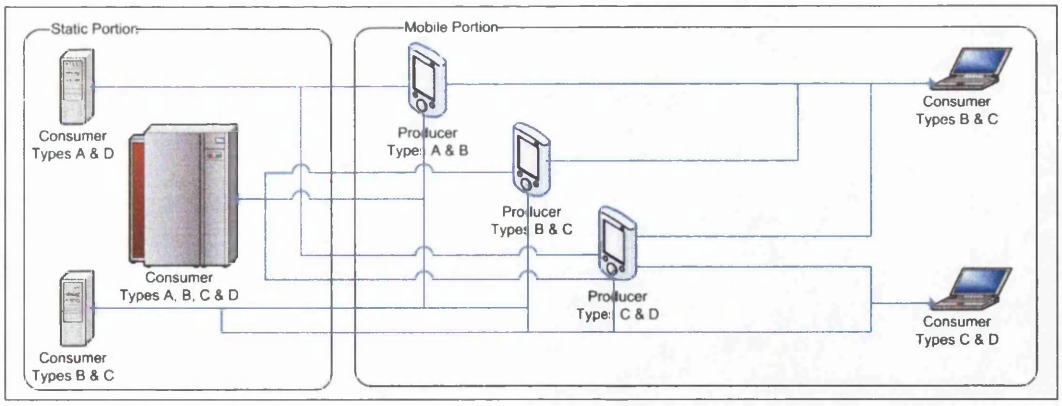


Figure 2.11: Mobile data delivery, original architecture.

example. Masters need not determine the identities of any worker nodes or load balancing servers in order to work, all that has to be done is to place the appropriate tuples into the space. The master may still be able to receive results which become available while it is absent by retrieving the relevant tuple from the space. The system can also engage in dynamic load balancing by starting new workers as necessary.

Mobile Data Delivery

This example describes a real-world problem to which the paradigm of generative communications was applied. The solution described has been successfully deployed in a commercial environment. Due to the commercial nature of the system, only a high level overview can be presented here.

A production system consisted of a large number of mobile nodes which produced various different types of data. These nodes would only connect intermittently to the network so that consumers could access their data. As well as these mobile data producers there were also a number of fixed nodes, and another couple of mobile nodes, which consumed various pieces of data stored on these devices. The situation is depicted in figure 2.11. Although it would be possible for the static nodes to monitor the set of visible[4] nodes and perform data extraction from those visible nodes when appropriate, this represented a significant amount of repeated effort, both in terms of development as well as runtime state management, across all the systems.

The solution to the problem was to provide a tuple space similar to that described above. The space could not simply reside on a single fixed node as some of the consumers were also mobile and were forming ad-hoc networks with the data producers so the space

[4]The set of visible nodes is a subset of the mobile nodes which can currently be communicated with.

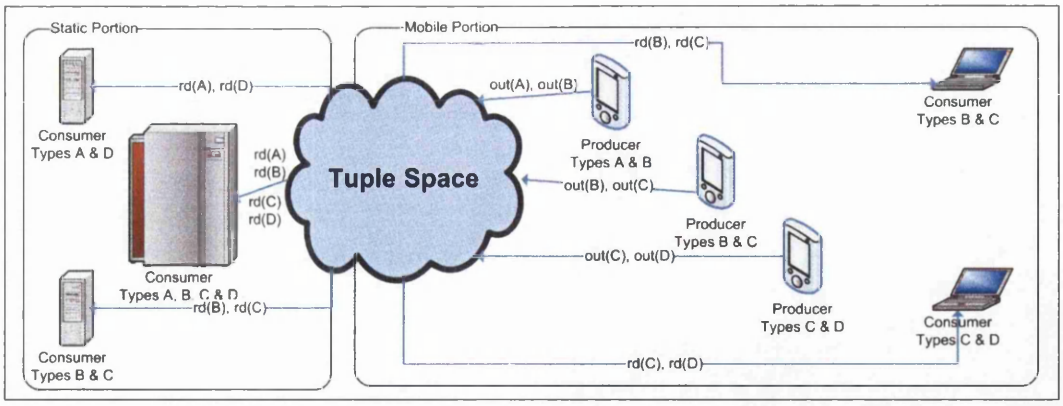


Figure 2.12: Mobile data delivery, tuple space architecture.

had to be mobility aware. The resultant architecture is similar to that depicted in figure 2.12. In this improved system, both the producers and the consumers benefit from the decouplings offered by the tuple space. The consumers of data are relieved of the burden of managing the changing set of visible node, instead they simply express their data need by performing **in** or **rd** operations on the space. The producers simply place all of their data into the space and do not need to be concerned about which server manages which data type. For the mobile consumers the interaction model remains the same as the fixed consumers, the tuple space absorbs the change and presents a unified set of operations to all nodes.

Although other solutions would be possible, the use of tuple spaces in this scenario offered a clean, simple model.

2.5 Disadvantages

At this point, it is worth identifying those applications to which the generative programming approach is not well suited.

Tuple spaces in general are ill-suited to situations where large amounts of data must be exchanged between two points. Although a tuple space may be used to set up such an exchange, other directed communications mechanisms will probably be better suited to the task of actually transporting the data. The identity decoupling in a tuple space prevents it from making some of the optimisations which a directed communication mechanism may utilise.

Tuple spaces are also ill-suited to carrying data streams or other forms of highly sequential data. The non-deterministic nature of retrievals from the space mean that anyone

attempting to receive the stream may have trouble receiving it in order. In these circumstances a separate mechanism (usually the implanting of a field containing the sequence number into the tuple) must be employed to ensure appropriate retrieval of the data.

The time decoupling offered by the space also makes it ill-suited to dealing with any time-critical or time-sensitive data as there are no guarantees as to how quickly such information will be returned by the space.

2.6 Summary

This chapter has described the context and environment in which the rest of the work presented in this dissertation will be set. An introduction to Linda has been provided along with examples highlighting its potential use and possible advantages in a mobile environment.

Chapter 3

Related Work

This chapter contains a discussion of other research which provides context and background for the work presented in this dissertation. Section 3.1 describes the historic Linda systems. By contrast, section 3.2 examines two "state of the art" commercial distributed Linda platforms alongside two prominent tuplespace research platforms. Section 3.3 examines other Linda systems and the extensions they have provided to the basic Linda model. This is followed by a discussion of peer-to-peer systems in section 3.4. Finally, other related work is presented in section 3.5.

3.1 Historic Linda Systems

Linda was originally proposed in [Gel85] and was designed to allow parallel processes to coordinate through a shared tuple space. C-Linda [CG90] represents one of the very first implementations of the Linda model. Basic by modern standards, it was only capable of operating on a single tuple space. All of the fundamental primitives were provided, including **eval**, which was performed by forking a new process to evaluate the tuple. The C-Linda implementation was incredibly faithful to the original Linda model, so much so, in fact, that many people came to think of it as actually *being* Linda. Although interesting from a semantic perspective (it was a direct implementation of the original Linda semantics as described in section 2.2) the system does not consider distribution.

3.2 State of the Art Systems

T Spaces [WMLF98] and JavaSpaces [W+98] are two powerful, commercial, distributed generative communication platforms. Both systems offer the tuple space abstraction to devices on a client/server basis. Event Heap [JF02, Joh02, JF04] and EQUIP [Gre02b, Gre02a], on the other hand, represent "state of the art" research platforms also based around the tuple space paradigm.

Although the centralised nature of the tuple space in these systems may not impact their usefulness in a distributed setting, it does reduce it in a mobile environment. Due to the changeable nature of a mobile environment, as outlined in section 2.3, the presence of other devices for the provision of services cannot be relied upon. This means that centralised architectures, where one machine must be visible to all others, are not appropriate in a mobile environment.

These systems presented as the "state of the art" in the current research and commercial fields will now be considered. T Spaces is presented in section 3.2.1, JavaSpaces follows in section 3.2.2, Event Heap is detailed in section 3.2.3 and EQUIP is discussed in section 3.2.4. The similarities and differences between the various platforms are then presented in sections 3.2.5 and 3.2.6, respectively, followed by an examination of how suitable they are for our environment in section 3.2.7.

3.2.1 T Spaces

T Spaces [WMLF98,LMW99,L+01] is a tuple space system designed and implemented by IBM. Based on a traditional client/server model, it provides a coordination infrastructure for networked applications. Implemented in Java [Sun02], T Spaces offers the traditional Linda primitives, albeit with different names — *Write*, *Read*, *Take*, *WaitToRead* and *WaitToTake* replace **out**, **rdp**, **inp**, **rd** and **in** respectively. No version of the **eval** primitive is provided. T Spaces offers the following extensions to the traditional Linda model:

Multiple Tuple Spaces

T Spaces servers can contain multiple distinct tuple spaces which can then be accessed by any connecting client (assuming appropriate group membership, see below). In addition to this, a client is permitted to perform concurrent operations on multiple T Spaces servers.

Client/Server Architecture

Tuple spaces are stored on and managed by T Spaces servers. Clients then connect to the server and perform operations on the tuple spaces stored there.

Access Control

Access to tuple spaces is managed through access groups. Each access group defines a set of permissions with regard to which tuple spaces members of the group can access and which operations they are allowed to perform on those spaces. The T Spaces server maintains a list of users and which access groups they are currently members of. When a client connects to a T Spaces server, the server determines which particular user the client is and, correspondingly, which access groups it is a part of. The server will prevent the client from accessing any tuple spaces or performing any operations for which it has no permission. Sufficiently privileged clients are allowed to modify group permissions or even create new tuple spaces.

Set Based Retrieval

Two extra primitives, *Scan* and *ConsumingScan*, are provided. These are analogous to *Read* and *Take*, but return the set of all matching tuples in the space.

The *Rhonda* Operator

Described as a rendezvous operator, *Rhonda* takes a tuple and an anti-tuple and matches them with the tuple and anti-tuple from another process executing a *Rhonda* operation. If, for example, process 1 executes *Rhonda*(<"A">,<*String*>) which writes the tuple <"A"> and requests any tuple with a string value, and process 2 executes *Rhonda*(<"B">,<*String*>), then process 1 will receive the tuple <"B">, while process 2 will receive <"A">. The *Rhonda* operator can be used to provide synchronisation between processes, although this would mean that the processes are no longer decoupled in time.

Event Notification

Clients in the T Spaces system can register interest in events. An event is simply any operation on the space. The client provides an object with a call-back method which is called when the specified event takes place.

Typed Tuples

Tuples in T Spaces are in fact Java objects and, as such, are typed. As per the Java type system, templates (anti-tuples) can match subtypes of the specified class. The fact that tuples are Java objects also means that it is possible to associate methods and, therefore, behaviour with the tuple. The only real disadvantage with this extension is that it alters the matching semantics of the Linda model. A tuple is no longer defined solely in terms of its contents, rather the tuple itself has a definite type. Imagine two unrelated classes A and B, which have exactly the same number and type of fields. Since the Java type system is not based on structural equivalence, but rather on name equivalence[1], a search for a tuple of class A will never return a tuple of class B even though they may be equivalent in all other respects.

Typed Tuple Fields

The fields contained within tuples are also Java objects and, as such, also benefit from subtype matching and the ability to have associated behaviour.

Extensible Primitive Set

Each T Spaces server has a series of factories which provide the actual implementations of the various primitives. These factories can also be set, at runtime, to distribute new implementations of the various primitives or even new operations altogether. The system allows different implementations or operations to be provided to different clients.

Queries

The T Spaces system offers the facilities of a query language to allow more powerful searches of the tuple space. Queries are based on searching for named fields within the objects. Examples of the query operations possible can be seen in table 3.1. This table, and the following explanations are reproduced verbatim from [WMLF98].

1. Query 1 is a regular structure Match query, where the query values are fed directly into the read operator. In this example, the query will return the first tuple of the form <"Superman", 75, Rock("Kryptonite")>.

[1]The full name of a class in Java comprises the name given to the class, for example, `java.lang.Object`, along with the identity of the classloader which originally loaded the class.

Query #	Query Type	Query Example
1	Regular	resultTuple = ts.read("Superman", 75, new Rock("Kryptonite"));
2	Match	queryTuple = new Tuple("Superman", 75, Rock); resultSetTuple = ts.scan(new MatchQuery(queryTuple));
3	Index	queryTuple = new Tuple(new IndexQuery("Superheros", "Spiderman")); resultSetTuple = ts.scan(queryTuple);
4	Range	queryTuple = new Tuple(new (IndexQuery("Superheros", new Range("A", "L"))); resultSetTuple = ts.scan(queryTuple);

5	And	queryTuple = new Tuple(new AndQuery(
		new IndexQuery("Superheros", new Range("A", "L")),
		new IndexQuery("Age", new Range(new Integer(10), new Integer(30)))));
		resultSetTuple = ts.scan(queryTuple);

6	Or	queryTuple = new Tuple(new OrQuery(
		new IndexQuery("Age", new Range(new Integer(10), new Integer(30))),
		new IndexQuery("Age", new Range(new Integer(60), new Integer(90)))));
		resultSetTuple = ts.scan(queryTuple);

Table 3.1: T Spaces Query Examples

2. The Match query's functionality is similar to the regular structure Match query, but it takes a query tuple as input. In this example, the query will return all tuples of the form <"Superman", 75, Rock>, where the values for the third parameter, Rock, can be any valid Rock value.

3. The Index query is either an exact match or a range. In this example, it is an exact match on the value "Spiderman". This query will return all tuples of any structure that have a Superhero field of the String type, with the value "Spiderman".

4. The fourth one is an example of an Index query using a Range predicate. This query will return all tuples of any structure that have a Superhero name in the range of "A" through "L".

5. The fifth one is an example of an And query. And and Or queries can be arbitrarily nested and used in any combination with other query types. This query will return all tuples of any structure that have a name in the range of "A" through "L" and an age in the range of 10 through 30.

6. Query 6 is an example of an Or query and is left as an exercise to the reader.

Transactions

T Spaces provides a transaction system which allows multiple tuple space operations to be applied as if they were one single atomic operation.

3.2.2 JavaSpaces

The JavaSpaces package [W+98, FAH99] consists of a series of interfaces and abstract classes which comprise a Java-based, Linda-like model proposed by Sun Microsystems for use in distributed systems. The main Linda primitives are present although with different names — *Write*, *ReadIfExists*, *TakeIfExists*, *Read* and *Take* replace **out**, **rdp**, **inp**, **rd** and **in** respectively. As with T Spaces, no equivalent to the **eval** primitive is provided. JavaSpaces itself is provided as a model to be implemented by other developers, such as the GigaSpaces system from GigaSpaces Technologies Ltd. [Gig02a, Gig02b, Gig03]. Sun also provide their own JavaSpaces implementation called Outrigger [Out02]. JavaSpaces offer the following extensions to the traditional Linda model:

Client/Server Architecture

JavaSpaces, much like T Spaces, uses a strict client/server model.

Multiple Tuple Spaces

Each JavaSpaces client is allowed to access multiple JavaSpaces servers concurrently.

Timeouts

Each of the JavaSpaces primitives can be provided with a timeout value. In the case of the *ReadIfExists* and *TakeIfExists* primitives, the value is used in the case where the only matching tuples are currently locked as part of a transaction (see below). The timeout value determines how long the primitive will wait for the transactions to settle and see whether the tuples become available or not. For the *Read* and *Take* primitives the timeout value states how long the primitive will block, waiting for a suitable tuple to become available. The timeout value can range from "no time at all" (i.e., return immediately) to "wait indefinitely" (i.e., block).

Event Notifications

Clients in the JavaSpaces system are allowed to register interest in any tuples matching a given template which are entered into the space. The client registers a listener with the server which is called in the event of a suitable tuple being added to the space. This allows the client to continue executing while awaiting the event rather than blocking.

Leases

Leases are a particularly interesting feature of JavaSpaces which avoid any garbage being left in the system. Whenever a tuple is placed in the tuple space, the client is returned a lease object which states how long the object is guaranteed to remain in the space. Specific lengths of lease can be requested by the client, although the decision of how long the lease will be is ultimately left up to the space itself. Once the lease has expired the space can safely remove the object. If the creating client wishes the object to remain, it must renew the lease. An expanded description of the leasing facilities provided can be found in [Sun00]. Event notification requests are also leased.

Typed Tuples and Fields

As in T Spaces, both the tuple fields and the tuples themselves are Java objects and have the same associated benefits and disadvantages.

Transactions

JavaSpaces provides a transaction system which allows multiple tuple space operations to be applied as if they were one single atomic operation. It has been shown that the transaction system currently presented is not serialisable[2] [BZ02], although extensions to make it serialisable are also presented in the same work.

GigaSpaces Extensions

As well as the above extensions to the Linda model, GigaSpaces also provides some further extensions. Batch operations for each of the basic primitives are provided, allowing large numbers of tuples to be placed into or read from the space. GigaSpaces also provides clustering technology [Gig02a, Gig03], designed to allow access to multiple spaces through a single proxy. The clustering technology is based on replication, although the exact mechanism used is not well described. It is hard to determine, from the available literature, what associated consistency issues there may be.

[2] A transactional mechanism is serialisable if operations which take place within a transaction could be modelled as taking place one after the other without the need to interleave them with operations outwith the transaction. Serialisability is a criterion for correctness in the execution of transactions.

3.2.3 Event Heap

Stanford's Event Heap [JF02, Joh02, JF04] was designed to support the development and operation of collaborative applications. To this end the designers made some modifications to the basic tuple space model:

Client/Server Architecture

Again, Event Heap is based around a centrally stored data space accessed by clients.

Self Describing Tuples

Event Heap tuples (called events) are composed of a set of fields which bear three attributes: type; value; and name. Types and values are used in the same manner to other systems. The name attribute is used to identify the field and, according to the designers, to allow developers to infer the meaning of tuples. These names are also used in the matching mechanism to identify which fields you want to match on.

Typed Tuples

Each tuple in Event Heap has a special field called "EventType". This field stores the type of the tuple itself. A type implies a certain minimal set of other fields will be present in a tuple. Tuple types can be extended simply by adding extra fields to the type.

Sequencing

Event Heap employs a FIFO[3] sequencing mechanism. This mechanism ensures that, if a client performs an operation which matches multiple tuples, then the system will return the earliest matching tuple which the client has not seen already. The system also provides a mechanism to snoop on the tuples without affecting the sequencing.

Expiration

Tuples in Event Heap also have a special field called "TimeToLive" which allows tuples to expire after a given time period has elapsed. This allows for garbage collection of tuples which have not been retrieved. It also allows developers to express immediacy in their applications where something should happen soon or not at all.

[3]First In, First Out.

Query Registration

Event Heap allows applications to register templates (anti-tuples) with the system. A callback mechanism is also registered along with the template. Should any tuples which match these templates be inserted into the space then the system will callback the registered client through the mechanism given.

Directed Tuple Routing

The Event Heap system also defines another set of tuples fields to the tuples to allow for some degree of direction to be established over the communication channel. For example, by default, the system will populate a field called "SourceApplication" with the name of the application which produced the tuple. This allows other applications to explicitly consume only data from that application. Similar fields also exist which note which application consumed a particular tuple to allow for subsequent tuples to be targeted to that consumer.

3.2.4 EQUIP

The EQUIP data space [Gre02b,Gre02a] was developed as part of the EQUATOR Interdisciplinary Research Collaboration in the UK [EQU05]. Much like the Event Heap, EQUIP's dataspace is primarily aimed at the support and construction of collaborative applications and workspaces. One of the strongest focuses in EQUIP is in facilitating the interoperation of applications developed in different programming languages. The following extensions to the basic Linda model are provided in EQUIP:

Replicated Client/Server

The EQUIP architecture is largely client/server based, however some of the tuples (called events in EQUIP) on the server are replicated down to the client. Clients can express a desire for tuples by expressing patterns to the server. Any tuples which match these patterns are automatically passed down to the client's local space. Clients can also use this local space for local-only tuples which are not passed up to the server. Otherwise all tuples are passed to the server and then replicated to interested clients.

Multiple Tuple Spaces

An application using EQUIP can create dataspace servers as and when required. These servers are accessed through a simple URL scheme where servers are given a name of the form "equip://host:port/spacename".

Event Subscription

Although based on the tuple space paradigm, EQUIP primarily provides event based interaction with the system. Clients register for events using patterns and provide a callback mechanism to receive them. When events are generated which match those patterns, the appropriate call back is made.

State Sharing

As well as the even mechanism, EQUIP makes provision for collaborating applications to share state. A shared piece of state will be stored in the local data space for the client interested in that state. When the state is changed, then "update" events are sent out that change the client's local copy of the state. If the item represented by the state goes away, then a delete event will be sent out and the state will be removed from all client spaces.

Language Independent Pattern Matching

One of the key goals of EQUIP was to allow interactions across programming language boundaries. To that end, EQUIP employs a completely language independent type and class system by using a subset of the CORBA Interface Definition Language (IDL) [Vos97]. EQUIP presently provides language bindings for C++ and Java. This IDL is also used in the serialisation, equality testing and pattern matching facilities within EQUIP. This allows applications to produce data for, or consume data from, other applications independent of their language.

3.2.5 Comparison

Similarities

The systems described above represent the "state of the art" in commercial and academic Linda-like systems. An examination of their similarities will identify those extensions of

the Linda model which have been deemed the most important.

All of the systems described above allow clients to access multiple different spaces. However, they do differ slightly in their models of server composition. In T Spaces and EQUIP each server can contain multiple distinct tuple spaces. In the JavaSpaces and Event Heap model[4], the server is the space. In these systems, there is nothing to prevent the running of multiple servers on the same machine, but this is not quite the same. The T Spaces and EQUIP architectures allow clients to create and destroy tuple spaces as needed. Therefore, clients can create a tuple space to achieve a particular task, and then remove it once the task is complete. This is not as easy to do using the JavaSpaces or Event Heap model (it would require external operating system calls to launch/kill instances of JavaSpaces as desired).

T Spaces and JavaSpaces both include typed tuples and fields. This means that both tuples, and their associated contents are stored as Java objects. This allows for a greater degree of flexibility during the matching process through Java's support for polymorphism. This object-oriented nature also extends the power of the tuple space since it enables tuples to contain functionality (in the form of method implementations) as well as state. This extended functionality does, however, prevent the use of structural matching. EQUIP also represents tuples using objects using the language independent IDL. Event Heap is the only system presented here which does not base its tuple typing around objects. Tuple types are represented by a field on the tuple. Types can be extended by adding fields, but there is no concept of inheritance or relationships between types.

The final commonality between all of the systems is the use of event notification. JavaSpaces, Event Heap and EQUIP all provide facilities to register interest in a particular pattern or type of tuple and then be notified whenever a tuple matching that pattern or type is inserted into the space. The T Spaces system provides a much more flexible event notification system. In T Spaces each and every operation on the server can be viewed as an event and, as such, any client may register interest in it. Thus, clients in the T Spaces environment are not limited to only watching for tuple insertions.

[4]The original implementation of Event Heap was built on top of T Spaces and, therefore, would have allowed for multiple spaces in a single instance. However, recent implementations have replaced T Spaces with a custom built space and this is no longer the case.

3.2.6 Differences

Aside from the common operations, each system offers its own distinct set of extensions not offered by the others.

The JavaSpaces system offers a means of garbage collection in the tuple spaces through the use of leases. Due to the nature of tuple spaces, it is impossible to identify when a particular tuple can be considered garbage. There is no knowledge available of which clients exist, and even if there was, it could not be guaranteed that an interested client would not appear at a later time. The use of leases addresses this problem by only allowing tuples to exist for a specified amount of time. This does, however, mean that clients must bear the burden of coping when their tuples are removed from the system. Event Heap offers similar facilities through the use of the "TimeToLive" fields on tuples. T Spaces and EQUIP provide no means of garbage collecting tuples.

Only the commercial solutions provide any form of transactional facilities, allowing the client to perform multiple tuple space operations as one atomic operation. The JavaSpaces system allows multiple servers to participate in a single transaction. Transactions in the T Spaces system, while permitting operation over multiple tuple spaces, require that all of those tuple spaces must be located on the same server.

Event Heap is the only system which employs a sequencing mechanism to allow for unique retrieval of tuples from the space.

EQUIP is the only system to provide language independence. It is also the only system to provide an explicit state sharing mechanism.

On the commercial side, aside from leases and distributed transactions, the T Spaces system offers all the functionality of the JavaSpaces system plus more. The T Spaces system offers expanded event notification, an extensible operation set, a new operator (*Rhonda*), set based operations and the facility to perform queries over the tuple spaces. In order to provide these operations the current implementation of T Spaces uses a fully fledged relational database to provide its storage and retrieval facilities.

3.2.7 Suitability

While the systems outlined above describe the state of the art in commercial and academic systems, neither system is designed with an ad-hoc, heterogeneous network in mind. Such an environment makes different demands of the system. One of the goals of the T Spaces system, for example, is to make the client side as lightweight as possible to allow for use

on PDAs and other, similar devices. While this is useful, no such claims are made of the server end. Indeed, given the heavyweight query facilities provided by the server, it is likely to incur storage and processing overheads as a consequence of maintaining indexes. While this may seem ideal in a client/server environment where it is acceptable to rely on the provision of a powerful server machine with plenty of storage, it is unsuitable for the kind of mobile environment previously outlined where the provision of a server machine cannot be assumed.

Another problem with these systems in a mobile context, is that, while they are all distributed in nature, they are based on traditional client/server models. Clients connect to a server, perform their operations and then disconnect. Due to this strict connection-oriented operation, this model is likely to prove unsuitable for the environment of interest in this work. In a mobile networking environment, it cannot be guaranteed that the devices which wish to cooperate will be in constant contact with any given server, or, for that matter, any other device. Even the replication mechanism in EQUIP is unsuitable for this environment as all replicated tuples are removed when disconnected from the server.

3.3 Other Linda Systems and Extensions

This section examines the wide variety of implemented Linda systems and the extensions to the basic model which they propose.

3.3.1 Javelin

The Javelin [Gre97] tuple space system was developed at the University of Glasgow by Robert Greig. Developed shortly before the release of the JavaSpaces system (section 3.2.2) the primary goal of Javelin was to implement the Linda coordination model in Java and, using the facilities provided by Java, attempt to construct a fault-tolerant distributed implementation. The implementation allowed for typed tuples as seen in JavaSpaces and T Spaces (section 3.2.1), with the same subclass matching.

All of the basic Linda primitives are provided in Javelin. Javelin supports the **eval** primitive through use of a preprocessor. The preprocessor, which is run prior to compilation, looks for subclasses of tuple which have **eval** methods defined. It then takes these classes and wraps them up in a custom class which implements Java's *Runnable* interface. This allows the custom class to be spawned as a separate thread at runtime in order to

perform the required calculations.

Javelin also provides support for various forms of distribution. In all cases the client connects to the outside world through a local "communicator" object which hides the details of the distribution from the client. In the basic version, the clients connect to a single tuple space on another server. In a second version, replication is used to achieve some degree of fault-tolerance, but with a significant performance trade-off. Tuple space servers are bundled together in "group spaces", each of which consists of one master and a number of slaves. All operations are performed on the master and then replicated onto the slaves. If the master fails, one of the slaves takes over. While this system can tolerate the failure of all but one host in a group space, it results in a significant performance drop due to the cost of replicating every action across multiple machines. This approach also struggles in the face of network partitions. Imagine a replicated space spread over a collection of hosts which are then subject to the network partitioning into two fragments. From the point of view of each fragment, it can be difficult to discern the network partition from the simultaneous failure of all of the hosts in the other fragment. As such, each fragment simply picks up where it left off, assuming the other one has ceased operating. This has the potential to result in multiple copies of the same tuple existing in separate parts of the network which could later reconnect. Even if the system were to try and keep track of all operations performed for later synchronisation should the network reform, it still does not prevent the system from distributing multiple copies of a given tuple in the meantime. Also it does not know how long it may be disconnected for, or even if it will ever become connected to the same set of hosts again, resulting in a potential waste of space in storing a theoretically infinite amount of synchronisation information which may never be used.

A third form of distribution attempts to improve performance by associating a particular class of tuples with each server. By spreading out the data types across multiple machines, Javelin hopes to reduce the load on any given machine. This is a simple form of a more general technique known as hashing. Hashing has recently become a popular mechanism for use in peer-to-peer systems (see section 3.4.2).

In all three cases the system relies on a known name server to locate the tuple or group spaces.

3.3.2 York Kernel

The York Kernel [RW96] is a distributed Linda implementation developed at the University of York in England. The York Kernel is designed to operate with multiple tuple spaces and includes a number of new primitives. The first of which is the **collect** primitive. The **collect** primitive allows a client to move all tuples matching a given anti-tuple from one tuple-space to another. The **copy-collect** primitive, a non-destructive version of the **collect** primitive, is also provided. It was proposed as a solution to the multiple **rd** problem [RW98]. The multiple **rd** problem is characterised by a client which wishes to non-destructively read all tuples matching a given anti-tuple in a tuple space (e.g., to collect statistical information). If the client simply performs multiple **rd** operations, it is not guaranteed to read all of the tuples due to the non-deterministic way in which the tuple is selected. Instead the client must perform repeated **in** operations, copy all of the tuples and then place all of the tuples back in the space through repeated use of the **out** primitive. The **copy-collect** primitive avoids the need for this disruptive and expensive alternative solution.

The York Kernel also has a set of extended primitives referred to as the BONITA primitives [RW97]. These primitives allow fully asynchronous interaction with the tuple space. While the Linda model promoted asynchronous message passing between clients, the clients interacted with the system in a very synchronous way. The **in** and **rd** primitives are good examples as the client must block until a matching tuple is inserted into the space.

The BONITA primitives all follow a similar path of operation. First, the client connects and uses the **dispatch** primitive. The dispatch primitive is overloaded and has a version for each of the other primitives in the system. The **dispatch** does not block, but immediately returns a request id. The tuple space performs the requested operation without any further intervention from the client. The client can check whether the operation has completed using the **arrived** primitive, which takes a request id and returns true if the request has been completed and false if it has not. The **obtain** primitive is a blocking primitive which takes a request id and returns the tuple or result associated with that request when it arrives. Through the use of these three primitives all operations on a tuple space can be performed asynchronously.

3.3.3 LogOp

The LogOp system [SM02a] proposes the use of logical operators for interacting with multiple tuple spaces. The three logical operators, **OR**, **AND** and **NOT**, allow the implicit parallelisation of the operations over multiple tuple spaces, improving the expressiveness of the model and providing a performance boost over the alternative of serialising the operations over multiple spaces. The basic Linda operations are provided, but can be combined with logical primitives with the following effects:

OR: This operator causes tuple space operations to affect one or more tuple spaces from a given list. In the case of **out**, one space, chosen in a non-deterministic manner, will receive the tuple. In the case of **rd** and **in**, the operation, if at least one space contains a matching tuple, will return a list containing a single matching tuple from any of the specified spaces which possess one. If there are no matching tuples in any space the operations will block until at least one matching tuple is inserted into one of the specified spaces. **rdp** and **inp** behave the same as **rd** and **in** with the exception that they will not block if no tuples can be found.

NOT: This operator is given a list of tuple spaces and it then performs the equivalent of an **OR** operation over the *complement* of that list. Although this adds no extra functionality, it may be more convenient if an application developer only wishes to exclude a small number of tuple spaces from a large set.

AND: This operator causes tuple space operations to affect a given list of tuple spaces. In the case of **out**, this results in a replication of the tuple. In the case of **rd** and **in** the operations will return a single tuple from every specified space *only* once a match has been found at *every* space. In the case of **inp** and **rdp** the operations will either return a single matching tuple from each space, or they will return nothing.

3.3.4 Ligia

The Ligia system [MW98] is a distributed Linda implementation, which is an implementation of previous work on tuple space garbage collection [MW97]. In Ligia, there is a single universal tuple space which can be accessed at all times. Processes can also create new spaces to which they are given handles. These handles are used to form a reachability graph for the tuple spaces. This graph is then used to determine which tuple spaces are no longer necessary in the system, as a consequence of being unreachable, and can be

removed. This means that garbage collection is only done on the level of spaces. Ligia does not provide a mechanism for collecting garbage tuples within a space.

Other than garbage collection, Ligia provides little else in the way of extensions to the Linda model. It is a simple centralised implementation. Clients can access multiple tuple spaces. The **eval** primitive is implemented in some form, but does not generate a tuple as in the Linda model. Its exact behaviour cannot be determined from the literature presently available.

3.3.5 Optimising Destructive and Non-Destructive Reads

Work by Rowstron [Row00] has proposed a potential optimisation to Linda systems by allowing tuples which have been removed as a result of an **in** operation to be returned to subsequent **rd** operations under a strictly defined set of conditions. This is designed to allow optimisation in situations where, for example, one process was responsible for updating a list while other processes were reading it. In this example, the items of the list are stored as tuples with a field indicating their position in the list. There is also a tuple which stores the length of the list. Whenever the process responsible for modifying the list wants to add or remove an item it must remove the tuple containing the length of the list from the space, modify it, and return it. As long as the tuple is removed, the processes reading the list will be blocked.

The system instead allows these processes to read a copy of the tuple even though it has actually been removed, known as a "ghost" tuple. "Ghost" tuples will remain in the space even after the destructive read, but:

- They cannot be returned as the result of another destructive read.
- The process which removed the tuple from the space cannot see the "ghost" tuple.
- The "ghost" must be removed from the space when the process which removed the original tuple terminates or inserts any tuple into the space.

This set of conditions is designed to ensure that no individual process is capable of seeing the inconsistency inherent in "ghost" tuples (i.e., that it can **rd** the tuple which it knows cannot be in the space).

In the above scenario, once the optimisation has been put in place, the reader nodes are no longer blocked by the updates performed by the list management node. They are

instead allowed to read a "ghost" of the list length tuple even if it has just been removed by the list manager.

The optimisation has been shown to provide benefit and has also been proven to maintain the correctness of the system [NPR00].

3.3.6 Physical Mobility and Linda

There have been a number of attempts to provide the Linda coordination model in an environment with physical mobility: Limbo [DWFB97]; L^2imbo [DFWB98, FDS+99]; Lime [PMR99, MPR01, MPR03]; CoreLime [CVV01a, CVV01b]; and PeerSpaces [BMMZ02, BMMZ03]. Due to the high-degree of relevance of these systems for the work presented in this dissertation, they are presented in chapter 4 to allow for a more in-depth discussion.

3.3.7 Logical Mobility and Linda

Logical mobility describes the ability of software components, usually referred to as software *agents*, to move from one device to another. The space, time and identity decouplings offer similar advantages to mobile agents as they do to coordinating mobile hosts. This has led to a number of systems which make use of generative communications to enable coordination between mobile agents [OZ98, Row98, CLZ99, BLP00, CVV01b, MPR03]. A good examination of the issues and systems involved in providing generative communications can be found in [CIZ99]. The work in this dissertation is focused only on physical mobility and is not concerned with providing facilities or support for logical mobility.

3.3.8 Linda for the Grid

Grid computing [FK99, DRBJS03, GGF04] is a varied field concerned with the provision and use of computing resources and services over well-connected, but geographically disparate sites. Grid computing has traditionally relied upon web services to provide access to the services in the system. Work by Bjornson *et al.* has suggested that a tuple space could be used in place of these web services [BS04]. They have built a system which coordinates the performance of tasks by grid based systems and the returning of results from those tasks through a single centralised tuple space. Work by Hawick *et al.* [HJP02] has proposed an architecture for providing grid based SuperSpaces. These SuperSpaces are formed by connecting separate tuple spaces together using software components called Transactional Workers. Transactional Workers link together a subgraph of the tuple spaces

and are responsible for forwarding queries to appropriate spaces and then routing the tuples back to the space where the request originated. At present, the construction of these SuperSpaces appears to be statically defined.

3.3.9 Emergent Technologies and Linda

Emergent technologies rely upon the interactions of simple localised behaviours (often inspired by natural or biological phenomena) to produce more complicated global behaviours. A good introduction to this class of systems is presented in [Res94].

Many of these systems are oriented around ants which use random walks and pheromone trails to produce a wide variety of behaviours. It has been suggested that such algorithms could be used to provide generative communications across a number of hosts [MT03, MZL03]. Due to the often random or unpredictable nature of these technologies, it is difficult to provide an evaluation of their applicability. This will be discussed further in section 9.7.

3.4 Peer-to-Peer

The controversy and associated expansive media coverage surrounding Napster has resulted in that system becoming almost synonymous with the phrase peer-to-peer (p2p). In reality, p2p encompasses a much larger class of system of which Napster is only one example. p2p systems can be identified by their decentralised architectures and methods of operation. Their intended environment has a lot of commonality with the one described in 2.3 — the systems are designed to operate without the provision of dependable, centralised nodes[5] and must adapt to changes in the environment as nodes arrive or depart (although this arrival or departure is not necessarily due to physical mobility). This section examines some of the more interesting facilities commonly provided in p2p systems. A good introduction to the problem space can be found in [Ora01].

3.4.1 Searching

One of the most common uses for p2p systems is to provide a decentralised distributed searching facility. The nature of the item being searched for can be anything from files or

[5]This was not the case in Napster, which used a central server to index the data on the peers and provide the search facilities over that data. It was this centralised architecture which proved to be its downfall as it gave the Recording Industry Association of America a target for litigation. It is also the reason why it is not a particularly interesting p2p system.

documents to the best recipe for raspberry cheesecake. Many of these searching algorithms [Gnu03, JXT04] use a flooding broadcast to locate the information. This can result in individual nodes receiving (and in some cases responding to) multiple copies of the same search request (although a time to live is usually defined for the broadcast packets to attempt to reduce this). Some systems attempt to remove this potential inefficiency, either through the provision of more structured overlay networks [RFH+01,ED02], through alternative query routing algorithms [HHL01, VvRB03] or through a combination of both [LRS02].

As well as basic searches some systems offer other forms of search. Freenet [CSWH01] focuses on providing anonymous searching for sensitive information as well as protecting the identity of the publisher of such data. Waldman *et al.* in Publius [WAC00] takes this idea further by making the data stored on the network resistent to tampering or censorship.

There are also a variety of techniques used to speed up searches or improve their chances of finding relevant data. Some systems make use of replication to pull the data towards the node which requests it in order to make it more available to others. This also has the useful property that data which is in high demand will quickly be propagated throughout the system reducing the load on individual nodes.

3.4.2 Hashing

One other technique used primarily to speed up searching is the use of distributed hashing which involves splitting the data set across a number of hosts based on some hashing function, the aim being to reduce the burden on any single server. However, within a p2p environment it presents new challenges as the system needs to be able to adapt the hashing algorithm and ensure data availability as devices come and go. If a static hashing algorithm is used, then the algorithm will continue trying to place data on or retrieve data from machines which have already departed. There are a number of approaches to solving this problem [RD01,SMK+01,HW02].

The use of such hashing to improve the scalability in client/server based tuple space systems has already been proposed in [OG02].

3.4.3 Summary

This section has looked at some of the interesting systems within the domain of peer-to-peer systems. Although there are some important differences between mobile and peer-to-peer

environments (for example, p2p nodes are not usually resource impoverished), there are also many similarities. As such, it is likely that at least some of the research conducted in one environment may be applicable in the other. One possible such application will be discussed in section 9.1.

3.5 Other Work

This section describes other related work which does not fit into the earlier structure. Publish and subscribe systems are outlined in section 3.5.1 while the Jini connection technology is described in section 3.5.2.

3.5.1 Publish and Subscribe

In publish and subscribe systems, participants are split into two categories: publishers, who are responsible for producing data; and subscribers, who consume the data. Subscribers register interest in types of data they are interested in and it is the responsibility of the middleware to attempt to route any published data to the appropriate set of subscribers. The publish and subscribe paradigm offers an identity decoupling similar to that exhibited by generative communications as subscribers do need to be aware of which entity is acting as the publisher. However, time and space decoupling are not always provided in publish and subscribe systems. Although some systems will store published data for late subscribers, many only provide delivery to those who have registered interest and are available at the time of publication. There are a variety of publish and subscribe systems designed for use in mobile environments [CFH+03, FGKZ03, FPM04].

Analysis by Busi *et al.* [BZ01a] has proven that the publish and subscribe paradigm is interchangeable with generative communications[6]. This means that the choosing what paradigm to use is analogous to choosing which programming language to choose. The final decision will depend on the nature of the problem along with the developer's personal competency or familiarity with either approach. Some problems to which generative communications are particularly well suited have been already been outlined in section 2.2.5.

[6]The conversion from publish and subscribe was performed through the provision of agents responsible for managing the state of the dataspace. As such the publish and subscribe system largely became a communication mechanism between the tuple space and the consumers.

3.5.2 Jini

Jini is a decentralised connection technology designed by Sun Microsystems [Edw99]. It is designed to provide a dynamic resource discovery service among networks of connected devices. Jini devices join communities of devices by registering their services in one or more lookup servers. Services have a set of named attributes associated with them represented by strings. Other clients can then query the lookup server to find services with matching attributes. The lookup server returns a proxy object which provides the client with access to the (possibly remote) service (this proxy object is published by the service provider at the same time as the other service information, such as attributes). This proxy object contains all the functionality needed to access the service it represents. It is important to note that the lookup server is in fact just another service in the system. This is crucial to providing some form of decentralised structure and also facilitates the creation and use of specialist lookup servers. Clients use multicast to obtain an initial reference to a lookup service. While this structure ensures that clients do not have to have knowledge of any specific lookup service, there is still a requirement that a lookup server must be running in order for the system to function. If mobile devices are taken into account, it cannot guaranteed that there will always be a lookup server available when a service is desired. Also, if there is only one lookup server in a given community, then it marks a single point of failure for that community. Recent work in adapting Jini to mobile devices has taken this into account and requires that all mobile devices run their own lookup server [Kam00].

Jini provides a highly adaptable framework for building networks of communicating device and of particular interest is the use of leases [Sun00][7]. Leases allow Jini clients to come and go in a lightweight manner. When the provider is advertising a service via its proxy in a lookup server, a lease is negotiated. When the lease expires, the service becomes unavailable and the lookup server will discard it. To avoid this happening, the provider can renew the lease. Proxy objects contain similar leases to the service provider. Leases prevent service providers from having to announce the departure or removal of that service, they simply allow the leases to expire. Once the leases have expired, the service will be removed from lookup servers and clients will dispose of the proxy objects. The use of leases also allows the system to repair itself in the event of a failure. In the case of a provider crashing, much like that of a provider leaving the network, the provider which

[7]The leases in JavaSpaces (section 3.2.2) are based on the set of specifications as defined by the Jini specification.

crashes will fail to renew any leases for any of its advertised proxy objects. Clients will also know to dispose of any existing proxy objects for the service provider. Thus the system eventually removes all reference to the failed service. In the case of a network failure the procedure is the same again, with any affected services eventually being removed from the system.

3.6 Summary

This chapter has highlighted various pieces of other research which either set context, provide background information or serve as comparison to the work presented in this dissertation. A more detailed discussion of the most similar pieces of research is presented in chapter 4.

Chapter 4

Mobile Linda Systems

This chapter takes a closer look at all of the available previous attempts to provide generative communications in a mobile environment. L^{2}imbo is presented in section 4.1. The LIME system is examined in section 4.2. CoreLime, a derivative of LIME, is discussed in section 4.3. Finally, PeerSpaces is presented in section 4.4.

4.1 L^{2}imbo

The L^{2}imbo[1] system [DWFB97, DFWB98, FDS+99, Wad99], developed at Lancaster University, is designed to provide support for adaptive mobile applications through intelligent use of quality of service (QoS) information. The L^{2}imbo system attempts to provide generative communications through a decentralised architecture.

4.1.1 The L^{2}imbo Model

The model provides the traditional Linda primitives **in**, **rd** and **out**. Four extensions to the basic Linda model are provided: multiple tuple spaces; tuple typing; QoS attributes; and system agents.

Multiple Tuple Spaces

The model provides support for clients to use multiple tuple spaces. Tuple spaces are created and destroyed by placing appropriate tuples in a common, universal tuple space (for more details see "system agents", below). Tuple spaces can be created with particular

[1]The system was originally named Limbo, but was renamed due to the name already being in use by another research group.

characteristics, e.g., persistence or access control. The clients of the system can then access these tuple spaces by obtaining a handle from the universal tuple space.

Tuple Typing

All tuples in the system are typed in a similar way to T Spaces (section 3.2.1) and JavaSpaces (section 3.2.2). However, instead of using the type hierarchy defined by the language, the Limbo model provides facilities for clients to define their own hierarchies dynamically. In brief, a client can nominate one tuple-type to be considered a subtype of another by placing a special tuple in the universal tuple space. The hierarchies are scoped per space and any restrictions (e.g., multiple-inheritance, cyclical hierarchies) are imposed by a type-manager for that space, if one is implemented.

QoS Attributes

The model proposed introduces the concept of deadlines which can be associated with either tuples or anti-tuples. Deadlines function in a manner similar to that of leases in systems like JavaSpaces (section 3.2.2). In the case of a tuple, the deadline represents the upper limit for how long the tuple is guaranteed to remain in a tuple space (barring any **in** operations on it). In the case of an anti-tuple, the deadline represents the time for which the appropriate **rd** or **in** operation is allowed to block. As in JavaSpaces, **inp** and **rdp** are implemented in terms of **in** and **rd** respectively with a low or zero deadline. Deadlines can also be used in the system to reorder tuple space operations to provide QoS guarantees; the system could, for example, decide to order operations in terms of closest deadline first in order to meet as many deadlines as possible.

System Agents

The Limbo model introduces the concept of system agents. System agents provide facilities for tuple space clients to interact with the system. One example of a system agent is the agent responsible for the creation and destruction of tuple spaces. As mentioned above, when a client wishes to create a new tuple space, it places a tuple into the universal tuple space. The system agent reads the tuple, creates the appropriate tuple space (if possible) and then places a tuple containing the tuple space handle into the universal tuple space. Another type of system agent is the *type management* agent. Type management agents are responsible for maintaining the user defined type hierarchy and deciding if a particular

request can be serviced by a sub-type. Bridging agents are used to connect tuple spaces. They organise the movement of tuples from one space to another. QoS monitoring agents watch various aspects of the system and make the information available to the clients via the tuple space. Other forms of agent can be implemented as necessary.

4.1.2 The L^{2}imbo Implementation

L^{2}imbo provides a decentralised implementation of the above model. The L^{2}imbo system provides decentralisation through a combination of replication and the Distributed Tuple Space (DTS) protocol. The DTS protocol provides facilities for spreading tuple spaces out among separate mobile hosts. Each tuple space belongs to its own multicast group, and clients attempt to maintain a consistent replicated version of the space. This is achieved by multicasting messages whenever an operation is performed on the space. Clients monitor these messages and use them to update their copy of the space. The universal tuple space mentioned in the model is implemented as one of these shared tuple spaces.

The DTS provides facilities for disconnected operation. Each tuple within a tuple space has a single owner associated with it. Only the owner is allowed to remove a tuple from the space, but the current owner can pass ownership on as desired. When a host disconnects from the network it retains its local copy of all tuples. While it is disconnected it can **rd** any tuples in the space. However, it is only permitted to perform an **in** operation on tuples in the space for which it is the current owner. It can be sure that no one else can perform a similar **in** operation in its absence since it is the owner. For tuples it does not own it cannot assume that no one is performing an **in** operation and so, to avoid allowing multiple **in** operations on the same tuple, it will not perform the **in**. Once the host reconnects, it informs the rest of the system of any removals performed. The host must therefore buffer any information regarding the removal of tuples during the period of disconnection, which could potentially constitute a large amount of data. The host then uses the contents of subsequent messages to determine if any tuples were placed in the space in its absence. If they were, then the host sends out a request for a copy of the appropriate tuple.

There are a variety of issues with this particular implementation, the majority of which stem from the DTS protocol and its implications. By introducing the concept of ownership, the L^{2}imbo system forfeits many of the characteristics which make Linda desirable in the first place. The decoupling in identity is lost, as a client must have knowledge of the

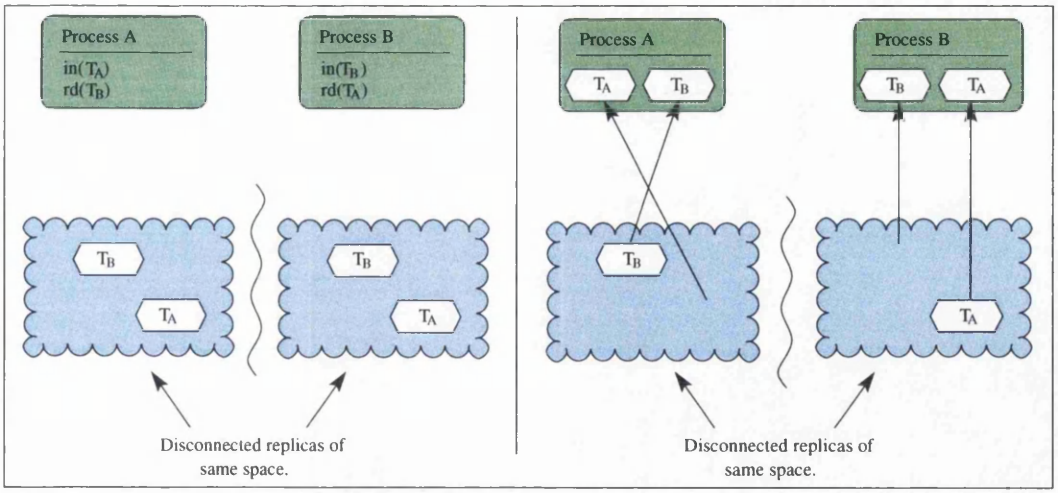


Figure 4.1: Linda semantic alteration in L^{2}imbo.

intended recipient of a tuple in order to pass on ownership. The client is also required to communicate directly with the recipient in order to transfer ownership (although this is concealed from the application), breaking the space and time decoupling.

The replicated nature of L^{2}imbo raises the issue of resource consumption. In order to make use of a tuple space as a coordination mechanism, a client must be willing to keep its own replica of the tuple space — a potentially substantial burden on a resource impoverished device. This problem is then exacerbated by the issue of ownership. Since only owners are permitted to remove tuples from a space, there is the potential for infinite resource consumption. If a client, which shall be labelled Bob, deposits a sizable number of tuples in the space and then leaves, no other client can remove those tuples until Bob returns, if ever. If Bob does not return then the tuples will continue to consume resources on *all* of the clients participating in that space.

The behaviour of the DTS protocol when the node is disconnected also causes a significant modification of the traditional Linda semantics. Due to the manner of operation, it is possible that clients in the system can continue to perform **rd** operations on a given tuple after that tuple has been removed from the space and returned to some client as a result of an **in** operation. This does not adhere to the Linda model, where the subject of an **in** operation is removed from the space.

This break in the semantics can most clearly be seen in a simple example shown in figure 4.1. On the left of the picture, two processes, A and B, are shown making use of a L^{2}imbo space containing only two tuples. Tuple T_A is owned by process A and tuple T_B is owned by process B. At some point the two spaces become disconnected, but due to

the replication mechanism they both still contain the two tuples. After this disconnection, each process performs two operations, shown on the right in the figure: A performs an **in** on tuple T_A followed by a **rd** on tuple T_B; B performs an **in** on tuple T_B followed by a **rd** on tuple T_A. In the traditional Linda semantics, regardless of the timing of the operations, there is no way of serialising these operations such that all four will be satisfied. Either A will have removed tuple T_A from the space as the first operation, causing the subsequent **rd** by B to fail, or B will have removed tuple T_B from the space as the first operation, causing the subsequent **rd** by A to fail. In L^{2}imbo, if the two replicated spaces are not in communication (or the appropriate messages are lost), it is possible for all four of these operations to be satisfied as the **in** operations will have no impact on the other replicated space. Although the ability to return as the result of a **rd** operation, tuples which have already been returned as the result of a **in** operation was proposed as an optimisation in [Row00, NPR00] (section 3.3.5), it was only allowable under strict conditions. The L^{2}imbo system does not meet these conditions.

Issues could also arise from the need to propagate large numbers of messages. Every operation on a tuple space generates a message to the multicast group. While some of the messages can be queued and sent in bulk to reduce overheads, it is not clear how suitable this is for a mobile environment. If the tuple space is under heavy load the messages could begin to consume significant amounts of network bandwidth, which may be a precious resource. Part of the problem stems from the unreliable nature of multicast communications. Since the system cannot be sure that every message will reach every participant, it must be pessimistic and multicast as many operations as possible in the hope that, eventually, some messages will reach each client. This is the reason for notifying the group of **rd** operations on tuples, as, through this, clients can learn of tuples for which they may have missed the **out** multicast.

4.2 LIME: *L*inda *I*n a *M*obile *E*nvironment

The goal of the LIME [PMR99, MPR01, MPR03] system, developed at Washington University at St. Louis, is to provide Linda-like facilities in a mobile environment. It was designed to handle both physical mobility (host machines moving around, joining and leaving the network) and logical mobility (in the form of mobile software agents which can move from one host to another) through the use of transiently shared tuple spaces.

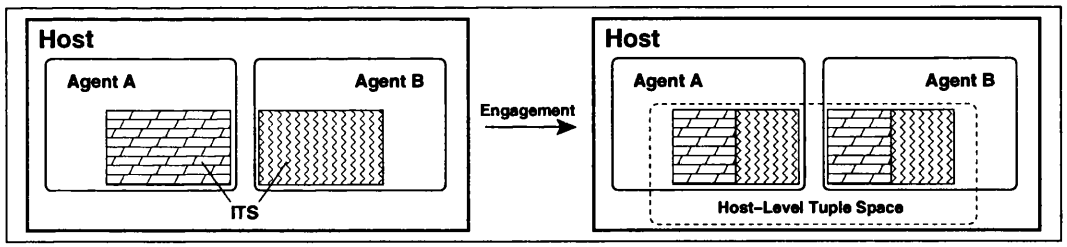


Figure 4.2: Engagement of ITS's to form host-level tuple space.

4.2.1 Transiently Shared Tuple Spaces

The LIME model is oriented around mobile agents. Each mobile software agent in the system has access to at least one tuple space, called the Interface Tuple Space (ITS), which is permanently associated with that agent. An ITS contains any tuples the agent wishes to share with the rest of the world (they can also have private spaces, visible only to the agent). This ITS supports the basic Linda operations **in**, **rd**, **inp**, **rdp** and **out**. When the agent is alone on an unconnected host, the ITS only provides access to that agent's tuples. However, the extent of what is visible through the ITS can be altered. When more than one agent exists on a host, their tuple spaces are "engaged" creating a *host-level* tuple space. This tuple space is then shared among the agents, becoming visible through the ITS. Figure 4.2 shows this model in operation. Two agents, A and B, are located on the same host. Each has access to the set of tuples in their ITS (represented by the different fill patterns). When the two agents engage, they form a host-level tuple space and now have access to the union of the two collections of tuples.

When two hosts become connected through the network, a similar engagement takes place between the two host-level tuple spaces. This creates a *federated* tuple space which, again, becomes shared among all the agents in the system. This process can be repeated as more hosts become connected, increasing the size of the federated tuple space. This is shown in figure 4.3. Hosts 1 and 2 become connected and engage. All four agents from the hosts now view the same shared tuple space through their ITSs.

The goal of the LIME system is that all tuple space primitives should maintain the same semantics irrespective of the nature of what is currently viewed through the ITS, be it local, host-level or federated.

Whenever a host disconnects, disengagement takes place and all departing spaces are removed from the federated view. Logical mobility is also supported through this engagement and disengagement mechanism. If an individual agent wishes to migrate, it first

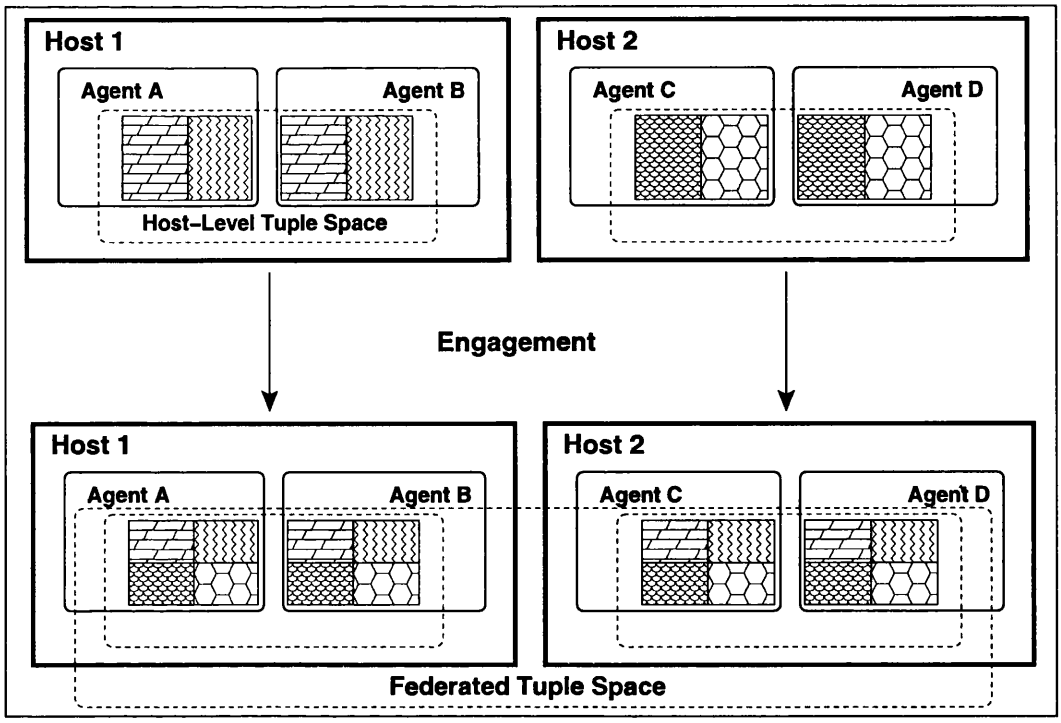


Figure 4.3: Engagement of host-level tuple spaces to form federated tuple space.

disengages its ITS from the system and then re-engages when it arrives at its new destination (which may or may not be part of the same federated tuple space). The migrating agent takes with it any tuples stored within its ITS.

4.2.2 Reactive Programming

LIME also introduces a reactive programming model to Linda. This is similar to event notification in JavaSpaces (section 3.2.2) and T Spaces (section 3.2.1). In LIME, a process can register a reactive statement, consisting of an anti-tuple and a section of code, with a tuple space. Immediately after a tuple is inserted into the space, each reactive statement in turn (selected in a non-deterministic manner) is evaluated. If the tuple being inserted matches the anti-tuple for the reactive statement then the corresponding code segment is executed. Each reactive statement must be evaluated before any further tuple space operations can be performed. Reactions can be defined as executing once only, or once per tuple. If once only is specified, the reaction will unregister itself the first time the code fragment is executed (i.e., a matching tuple is inserted). When once per tuple is specified, the reaction will remain registered, but will only execute at most once for any given tuple.

4.2.3 LIME in a Mobile Environment

The tightly constrained nature of the LIME model is in conflict with the highly dynamic nature of a mobile environment. This disparity between model and environment also manifests itself in the implementation. This section examines the issues present in both the LIME model and the associated implementation.

The LIME Model

The primary problem with the LIME model lies in its attempt to maintain a globally consistent view across all tuple spaces. While this is feasible within a given host, or perhaps even a small number of hosts, it is likely to prove impractical for large networks involving many hosts, where large latencies may result in operations on the federated space becoming increasingly expensive.

One proposed solution to this problem, presented in [PRM00], is to assign each host a set of preference vectors defining information or activities in which the host is interested. These vectors are then used to group the hosts in an attempt to place hosts with the most interests in common in the same group. Essentially, the preference vectors are fed into a mathematical function which calculates how "happy" two hosts would be together (essentially a measure of how many interests they have in common). The system then employs various techniques to attempt to maximise the happiness of all the groups in the system. This approach suffers from a number of problems.

First of all, connection no longer guarantees communication, that is, just because two hosts are connected does not mean that they will eventually be in the same group. As such, two hosts who wish to perform some coordinated task may be unable to do so as they never end up in the same group. There is also a naming issue in the preference vectors. Imagine, for example, two hosts who are both interested in cats. One host has the word "cats" in its preference vector, but the other has "felines" and, as a result, they never end up in the same group. To address this issue would require either a sophisticated matching algorithm able to incorporate the nuances of language, a centralised name service, agreement on the part of developers on a common ontology or the intervention of a third party standards body to define an appropriate ontology.

Preference vectors also only store the concept of interest with no concept of intent. Consider a large group of hosts, some of which are interested in obtaining information on coffee and some of which have lots of information on coffee. Each of the hosts then has

the word "coffee", and only the word "coffee", in their preference vectors. In this way every host will be just as happy with any other host in the system and the division into groups performed by the system will be arbitrary. However, this means there will be no way of ensuring that the system does not result in groups of hosts all seeking information on coffee with no one to obtain it from, or groups of hosts all with lots of information on coffee, but no one interested in reading it.

The LIME model also calls for the engagement/disengagement operations to be atomic across all hosts in the federated space. This means that other operations cannot proceed while hosts are engaging/disengaging. This could prove disastrous in an environment where machines come and go rapidly, potentially causing significant delays in normal tuple space operation processing as the engagements/disengagements are dealt with.

The reactive programming model included in LIME also raises certain issues. In particular, the option to have a reaction occur "once per tuple" implies that either any "once per tuple" reaction must maintain a list of all of the tuples which they have acted on already[2] or the tuples must maintain a list of those "once per tuple" reactions which have been encountered. In either situation, assuming a sufficiently large system there can be no guarantee that there will be sufficient storage space to maintain either of these lists.

Also, in theory, the contents of the code fragments provided in a reactive statement can be an arbitrary piece of code of the programmer's choice. This means that these code fragments can have adverse effects on the running of the LIME system. In the worst case, the code fragment could execute one of the blocking primitives on the tuple space (i.e., **in** or **rd**). If no matching tuple is ever inserted then the operation will never return, the reactive statement will never complete and the tuple space will be unable to make any further progress. In other cases the reactive statement may contain an infinite loop or even form an infinite reaction loop by performing an **out**, which triggers some other reactive statement which then does the same, resulting in livelock[3].

The LIME Implementation

There are a number of issues which arise from the current implementation of LIME. While not inherent flaws in the model, they are a consequence of trying to implement a model

[2]Also implying that each tuple must be assigned some unique identifier as content is not guaranteed to be unique.

[3]This has the potential to be an even more significant problem as it may result from a reaction that has been placed into the system by some other client application. As such, the programmer will be unable to foresee it until runtime.

that is not sympathetic to its intended environment and as such merit discussion.

The majority of the disadvantages to the LIME approach stem from the engagement and disengagement operations required in the model. Firstly, the need for explicit disengagement does not allow machines to come and go as they please, as they will do in a mobile environment. Instead, all machines must announce their intention to leave and allow the system to atomically remove them from the federated space. This is impractical in an environment where the machine can disappear from the network without any notice (imagine a user with a PDA leaving the catchment area of the network, a laptop running out of battery power or, indeed, a process crashing).

Secondly, the mechanics of engagement/disengagement do not stand up well to the rigours of a mobile, ad-hoc network environment. The implementation requires that a single host acts as the engagement leader, and it is through this host that all other machines join a federated space. This approach has the inherent problem that each federated space exists only as long as the leader is present. If chosen badly, the machine may depart before construction of the federated space can even finish. Finally, since machines can only depart or join one at a time, no provision is made for the network partitioning, or for two federated spaces combining.

4.3 CoreLime

CoreLime is a simplified LIME variant developed by Carbunar *et al.* at Purdue University in order to address many of the scalability issues which LIME presented [CVV01a, CVV01b]. It attempts to simplify the ambitious model presented by LIME whilst still trying to maintain the semantics of the various primitives.

The most fundamental difference between LIME and CoreLime lies in the federated tuple spaces, or, indeed, the lack thereof. CoreLime does away with federated tuple spaces altogether. Mobile agents still have ITSs and they can form host-level spaces similar to that shown in figure 4.2. Host-level tuple spaces are no longer permitted to form federated tuple spaces. This removes many of the global synchronisation problems which arose from trying to maintain a consistent view of the world. All the LIME operations are now carried out only on co-located ITSs. No remote communications are permitted at all. Instead, clients are expected to take advantage of the logical mobility facilities to access other host-level tuple spaces. If a client wished to perform an **in** on a remote, host-level tuple space,

it would first create a new mobile agent which would then migrate to the specified host. There, it would engage with the other agents and become a part of the host-level tuple space. This agent would then be able to perform the requisite **in** before migrating back to the original host and delivering the retrieved tuple to the client. Similar steps can be taken to use the other basic primitives on remote tuple spaces.

The CoreLime model also alters somewhat the semantics of reactive statements. Reaction statements are now executed concurrently with the user code, avoiding the termination issues present in the traditional LIME model.

The CoreLime system is a step in the right direction. It removes many of the inefficiencies present in the LIME model, and yet retains much of the functionality in the form of host-level tuple spaces. However, CoreLime loses much of what made LIME interesting in the first place. CoreLime removes the ability to federate the tuple spaces. While this removes those issues related to global consistency, the application developer must now bear the burden of discovering which tuple spaces are available, connecting to them and performing operations on them. This is in stark contrast to the model originally envisioned by the LIME team, where the application developer interacted only with the ITS and the underlying infrastructure dealt with the communication to, and operations on, other tuple spaces.

As far as can be told, there is, at the time of writing, no implementation of the CoreLime model.

4.4 PeerSpaces

PeerSpaces [BMMZ02, BMMZ03] is a system designed to provide generative communications in a peer-to-peer environment. Although PeerSpaces is not strictly designed for use in a mobile environment, there are enough similarities between mobile and peer-to-peer environments to warrant discussion of the system.

The PeerSpaces model has been implemented as a service on top of the JXTA [JXT04] framework developed by Sun Microsystems. JXTA provides a generalised set of low-level services to aid in the development of peer-to-peer applications.

PeerSpaces uses the JXTA framework to construct an overlay network of PeerSpaces nodes with each node containing a local tuple space. Operations which cannot be handled by the local space are passed out to the remote instances using a flooding broadcast. A

time to live (TTL) field determines the horizon over which the search is performed.

The PeerSpaces system has the advantage of providing a self-organising network which can deal with nodes coming and going through the JXTA framework. PeerSpaces also benefits from ongoing efforts to incorporate improved security features into the JXTA framework. However, there are some disadvantages which make PeerSpaces less than ideal for mobile environments. Firstly, the JXTA framework makes extensive use of XML in all of its communication protocols. The extra information embedded in the XML creates an increase in the amount of data which must be sent as well as an increase in processing and parsing overhead. Both of these are likely to have an impact on resource impoverished devices.

Secondly, the PeerSpaces system does not provide a resource management mechanism. Nodes at present are not capable of controlling the amount of work they carry out on behalf of other nodes or local applications. This is crucial in a mobile environment where resources are scarce. The concept of data expiry (similar to leases) has, however, been proposed as a future extension.

The costs of constructing and maintaining an overlay network in the face of increasing amounts of network change are not, at present, well studied or understood. It is possible that, as the amount of change experienced in the system increases, the amount of effort expended in maintaining the overlay network could outweigh any benefits derived from its presence. In addition to this, the flooding broadcast is wasteful of bandwidth and processing resources, especially at high TTLs. In a mobile environment such waste could prove troublesome.

4.5 Summary and Conclusions

This chapter has taken a closer look at previous and current attempts to provide Linda-like semantics in a mobile environment. In every case there are some incongruities between the model and the environment which lead to various issues either in the model itself or in the resultant implementations. In the case of L^2imbo, there is a substantial drain on precious resources for participating in the space combined with the breaking of the Linda semantics at the expense of its useful decouplings. In LIME, the drive to provide global consistency along with explicit connection and disconnection operations has led to a model which does not gel with a highly dynamic environment and a problematic

implementation. CoreLime, while attempting to solve some of the issues with LIME, has discarded LIME's core abstraction, namely the federated spaces. Instead the burden of managing the changes in the environment is once again passed to the application developer to bear. PeerSpaces, due to its design as a peer-to-peer system, has not had call to consider resource consumption as a priority and as such lacks the resource control that is necessary for impoverished mobile devices.

The problems with the various attempts to provide Linda in a mobile environment have led some to conclude that the paradigm is ill-suited to such environments. The next two chapters, however, present a model (chapter 5) and corresponding implementation (chapter 6) which show that, by taking an environment-centric approach to design, the advantages of Linda can be made available to application developers wishing to develop applications for mobile environments.

Chapter 5

The Linda$_m$ Model

The previous chapter highlighted the various issues present in existing systems which attempt to provide generative communications in a mobile environment. This chapter presents Linda$_m$, a model for providing generative communications which has been designed around the constraints and demands of the underlying environment. Linda$_m$ provides the abstraction of opportunistic logical spaces allowing Linda-like semantics to be provided in the face of environmental change. Linda$_m$ also provides leases as a means of fine-grained resource management.

The chapter opens with some brief notes on the terminology used within the chapter in section 5.1. Section 5.2 takes the discussion of the environment from section 2.3 and highlights the resultant implications which must be considered in the design of Linda$_m$. Section 5.3 enumerates the core design assumptions. The primary features of the Linda$_m$ model are outlined in section 5.4. Finally, section 5.5 summarises the extensions which have been made to the Linda model in Linda$_m$. Tiamat, an implementation of the Linda$_m$ model, is presented in chapter 6.

Much of the work presented in this chapter has previously been published in [ME03].

5.1 Definition of Terms

Mobility/Mobile: The work described is interested solely in physical mobility (i.e., when devices are moving around the physical world) and is not concerned with logical mobility (i.e., where software components move from one location to another). Unless explicitly stated otherwise, it can be assumed that the words mobile and mobility refer exclusively to physical mobility.

Node/Device: The word node is used to indicate any active participant in a Linda$_m$ system. A single device can contain several nodes (running in separate virtual machines for example); however, for the purposes of this discussion, it is assumed that each device represents a single node and hence the terms node and device are used interchangeably.

5.2 Design Principles

As discussed in the previous chapter, many of the existing systems for providing generative communications in mobile environments did not fully consider the environment they were supposed to operate in, resulting in a variety of issues in both the resultant models and implementations. In Linda$_m$ the goal is to avoid the same issues by adopting an environment-centric design.

This section examines what impact, if any, the various characteristics outlined in the description of the environment from section 2.3 will have on the design of the Linda$_m$ model. It begins by looking at the impact of resources in section 5.2.1, followed by connectivity in section 5.2.2 and the effects of mobility and change in section 5.2.3.

5.2.1 Resources

It is assumed that the majority of devices in the environment will possess relatively few and often limited resources, and, as such, these resources must be carefully managed. While it is true that, with advances in technology, certain resources could become more abundant in the future, there are also likely to be corresponding increases in the complexity and resource requirements of devices. This situation is already experienced by mobile phone manufactures who, despite numerous improvements in battery technology, are finding that even the improved power supplies are insufficient to meet the demands created by the increasingly complicated features packed into their phones [Bie04]. Although some devices within the environment will be resource rich, they will not be in the majority. It would not be prudent to design a system in which resources are assumed to be bountiful. This would be designing for the exception rather than the rule. It is better to design for the general case and provide extensions or optimisations for the exceptions at a later stage.

It is important to note that Linda$_m$ will represent only one piece of software on a device. While it is important that Linda$_m$ be capable of managing the consumption of resources

resulting directly from its actions or use, its purpose is not to manage the resources of the entire device. Such decisions must be taken at a lower level where the state and operation of the entire device can be assessed. One discussion of how such decisions can be made and managed can be seen in work by Neugebauer [Neu03]. For Linda$_m$, the implication is that, while the mechanism for managing resources must be provided, the policy will come from elsewhere.

5.2.2 Connectivity

The issues involved in designing and implementing low-level network protocols for operation within mobile ad-hoc environments, as well as the potential interactions with existing network protocols, is a vast research area and is outwith the scope of this work. It is through such work that the characteristics of a device's connectivity will be defined. As such, Linda$_m$ focuses on how to provide generative communications once such communications channels are in place rather than the intricacies of establishing and managing those channels at low-levels of protocol.

5.2.3 Mobility and Change

It is important that any model designed for an environment which is characterised by change should incorporate change as part of the normal system operation, not as an exceptional circumstance. This is another example of programming for the majority. This means that the Linda$_m$ model should allow devices to come and go frequently without causing disruption to their own or other devices' operations.

The frequent mobility and high degree of change exhibited by the environment mean it would not be prudent to depend upon the presence of other machines, as mobility may eventually separate devices from those they are dependent on. In extreme cases, devices may become completely isolated from others. It is therefore assumed that all devices are operating in an ad-hoc fashion rather than rely expressly on the provision of certain infrastructure. As such, it is important that Linda$_m$ be designed to allow devices and their applications to operate in such isolated conditions and can take advantage of, but not depend upon, other devices if they happen to be present

5.3 Assumptions

This section takes the environmental discussions from sections 2.1 and 5.2 and distills them into a set of assumptions which drive the remainder of the design.

It is assumed in this work that...

- ...the majority of devices will be improverished in one or more resources. It is not assumed that any resource in particular is in short supply, rather that the middleware must provide mechanisms for the management of resources which can be driven by external policy.
- ...the majority of devices in the system will be independantly mobile. This means that no machine can rely upon the provision of a specific other machine to make progress.
- ...operation in isolation is desireable. In other words, any coordinating applications or processes residing on a single machine should be able to make progress without the provision of any other machines.
- ...the environment will be heterogeneous in terms of architecture and capabilities of devices. No single architecture or platform can be relied upon.
- ...the environment will be heterogeneous in terms of network availability. Even when other machines are present, it should not be assumed that it will be possible to establish or maintain connection with them.

5.4 Linda$_m$

It was established in section 2.4 that the decoupling offered by generative communications would be advantageous in a mobile environment. As such, the primary goal of the Linda$_m$ model is to provide the well understood semantics of the traditional Linda system in the environment outlined in section 2.3. This section describes how the Linda$_m$ model provides those semantics in a mobile environment: opportunistic logical tuple spaces are described in section 5.4.1; the facilities for direct remote communications are presented in section 5.4.2; and section 5.4.3 introduces leases, the mechanism for resource management.

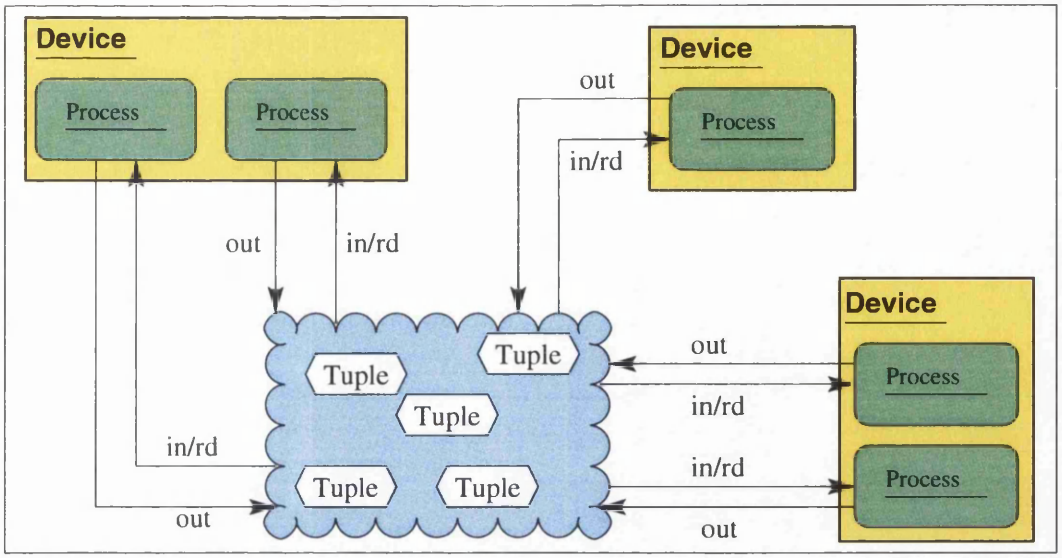


Figure 5.1: The Linda$_m$ model.

5.4.1 Opportunistic Logical Tuple Spaces

From the perspective of a local, application-level process[1], Linda$_m$ provides a single space, as depicted in figure 5.1, through which the process can coordinate with other processes. The single space allows a process to interact with other processes through a single abstraction providing a single set of operations. Processes need not concern themselves with whether the other process they are coordinating with resides locally or remotely (although they can do so if desired, see section 5.4.2). The processes can perform the normal tuple space operations (as described in section 2.2) on this space and receive results accordingly.

Design

As detailed in section 5.2.3, devices in this environment cannot depend on any other device to provide services or facilities. This immediately dismisses the possibility of client/server architectures. Instead, each node must be able to operate independently and, as a result, must contain its own tuple space. While this does place extra resource demands on the participating devices, it is the only way of guaranteeing the device has access to a tuple space[2]. The provision of local tuple spaces allows coordinating processes located on the same device to make progress even while the device is in complete isolation. The local

[1]Henceforth referred to as a 'process'.

[2]While it would, in theory, be possible to remove some of the local spaces by identifying those devices which primarily operate in the presence of other devices (e.g., when someone is using their PDA, their phone is rarely far away), determining when/where this would be applicable would be far from trivial for any real system. It would also become even more complicated should the dynamics of the system change.

tuple space must be capable of handling the basic Linda operations (see section 2.2). Aside from this requirement, all other aspects of the operation of the local space are left to the implementation.

In order to provide the appearance of a single space two possibilities presented themselves: replication and composition. Replication would involve each device maintaining its own replica of the whole tuple space. Along with the issues of maintaining consistency, as discussed in section 4.1.2, replication places substantial resource demands on a device since each device must agree to store a complete replica of the space. Composition involves forming a single space out of a number of other spaces. This has the advantage that each device must only maintain a subset of the overall space locally. Since the majority of devices in this environment are resource impoverished (as discussed in section 2.3.4) it was decided that replication would place too great a burden on participating devices and, for this reason, the single space presented to the processes should be a composition of the local spaces from each process. This single space presented is called an Opportunistic Logical Tuple Space (OLTS).

Operation of Opportunistic Logical Tuple Space

The OLTS presented to a process is composed of the local space on the device along with the local spaces of any other devices which are currently visible. Another Linda$_m$ node is considered visible if it can be communicated with via some mechanism. The exact means of this communication may be *implemented* in different ways, e.g., through direct communication only, or routed through other nodes, as can its scope, e.g., local nodes only, or those connected via the Internet[3]. The concept of visibility is depicted in figure 5.2. Part (a) shows two isolated Linda$_m$ devices. In this case the logical tuple space presented to processes residing on the device consists solely of its local space. If these devices become visible to one another, then the single logical tuple space will now be a combination of the two physical local spaces, as shown in (b). This allows the process on each device access to the tuples stored both locally as well as those stored on the other devices.

When a process performs a tuple space operation on the OLTS the same operation is performed on the local space. When writing to the space with **out** or **eval**, the default operation only contacts the local space. In the case of reading from the space with **in**, **inp**, **rd** or **rdp**, as well as performing the operation on the local space, Linda$_m$ determines

[3]The Linda$_m$ model does not depend on any particular implementation of visibility, only the concept of visibility.

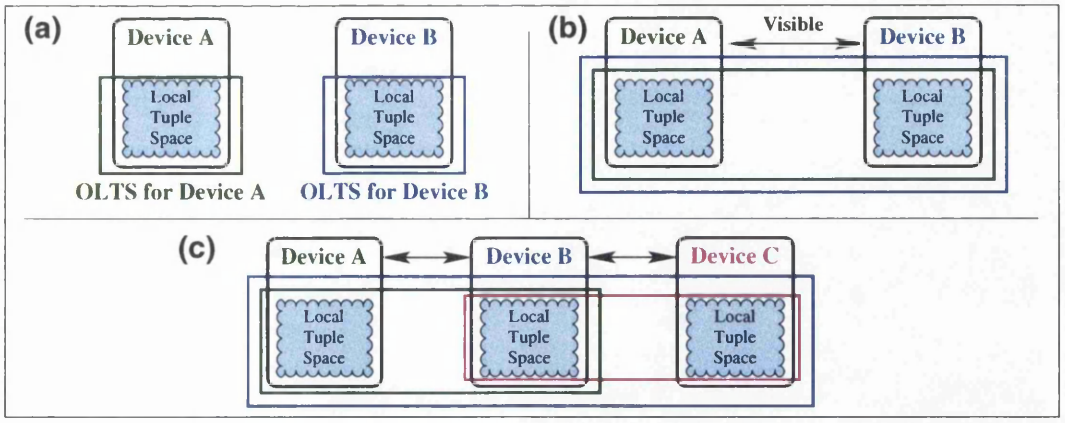


Figure 5.2: Opportunistic Logical Tuple Space operation.

which nodes are currently visible. The operation, including a copy of the associated tuple, is then propagated to the set of currently visible nodes, thus forming the OLTS. These nodes will then perform the operation on their respective local spaces. If a matching tuple is found in a remote node, it will be returned to the node from which the operation originated, assuming it is still visible. If the node is no longer visible then the tuple is placed back into the space. In the case where multiple remote nodes locate matching tuples, the first one to be returned to the originator will be accepted and the others will be returned to their respective spaces.

Opportunistic vs. Global Consistency

The devices in the previous example are both presented with identical OLTSs. Linda$_m$, however, makes no guarantees that such views will always be globally consistent; the opportunistic nature of the space means it is possible for separate processes using Linda$_m$ to see different logical spaces. This scenario is depicted in part (c) of figure 5.2 where a third node, C, becomes visible to node B, but not to node A. Node B now has a logical tuple space consisting of all three tuple spaces. The logical tuple spaces for nodes A and C, however, consist only of their own local space along with the space from node B.

Although global consistency is not provided, constructing the logical tuple space opportunistically, as operations are performed, removes the need for explicit connection and disconnection operations[4]. In accordance with the design principle identified in section 5.2.3, this means that, from the perspective of the individual nodes, other Linda$_m$ nodes

[4]Although the model does not require explicit connection and disconnection operations, it does not preclude a particular implementation from providing such operations as optional. This could be desirable to allow application developers to choose whether nodes passively or actively perceive change in the set of visible nodes.

can enter or leave the scope of visibility without affecting the semantics of any ongoing operations (although, if their local space contains matching tuples, their arrival or departure may affect the *result* of the operation). As such, the opportunistic model gels well with the environment.

It should also be noted that none of the applications described in section 2.2.5 actually required global consistency to function — only the ability for a single application to see the tuples of another is required and this functionality is provided by the OLTS.

This opportunistic model also allows Linda$_m$ to adapt to changes in the mobile environment and, from the process' perspective, such change is absorbed by the model and manifests as the removal or insertion of a number of tuples from or into the OLTS. By being absorbed by the model, change becomes a normal part of the lifecycle of the system, as called for in section 5.2.3, rather than an exceptional circumstance.

5.4.2 Direct Remote Communications

The abstraction over underlying change provided by the logical tuple space allows processes to function without direct knowledge of remote Linda$_m$ nodes. However, it is important to appreciate that in some situations, processes can make good use of such information and that Linda$_m$ should not prevent a process from obtaining this knowledge if it so desires. As such, Linda$_m$ offers processes the means to interact with specific nodes when required. This functionality is particularly useful in the case of **out** and **eval** where the local space may refuse to accept the tuple due to resource shortages (see Section 5.4.3), or in the case where the process wants to make tuples available to other Linda$_m$ nodes even after it leaves.

In order to support this, each local tuple space in Linda$_m$ contains a special tuple. This tuple contains a handle on the space as well some information about that space, e.g., whether the local space provides a persistence mechanism or not. Processes can read these tuples and use the handles to perform operations on specific remote spaces. All of the operations have special versions which take these handles and perform the operation requested on the remote space specified. If the destination is no longer available, the operation is abandoned.

An alternative means of supporting direct remote communications is also provided in the case of **out** and **eval** by way of a third version of each operation. These take a tuple that was returned as a result of a prior **in**, **inp**, **rd** or **rdp** operation. Linda$_m$ will then

attempt to satisfy the operation at the node where the given tuple was obtained (which may be a remote node or may be the local node). If the desired destination is no longer visible, the operation fails.

5.4.3 Resource Management

As discussed in section 5.2.1, Linda$_m$ should provide a mechanism through which the potentially limited resources of the device can be managed. For this reason, the Linda$_m$ model includes leasing as a mechanism for fine-grained resource management within Linda$_m$ nodes. These leases operate in a similar fashion to the leases used in JavaSpaces (see section 3.2.2) and the QoS guarantees in L^{2}imbo (see section 4.1), although Linda$_m$ leases can encompass more than just time information. Due to the asynchronous, identity-separated nature of generative communications, it is not normally possible to identify tuples as being garbage, meaning that any resources consumed by the tuple can never be recovered. In Linda$_m$, the leasing model allows tighter controls to be placed on how long tuples may reside in the space before being removed. By also extending the leasing mechanism to all operations, and by allowing lease expiration to be defined in terms of resources used, as well as time, a Linda$_m$ node can control access to its resources on a resource by resource basis. The leasing mechanism also allows application programmers to specify upper boundaries on the availability of their tuples.

Linda$_m$ defines a leasing model in which every operation on the tuple space is leased. Whenever a process performs an operation, it must first negotiate a lease with a Linda$_m$ node. These leases represent the effort, in terms of resources, a Linda$_m$ node is willing to dedicate to carrying out the operation. These leases may be based on time or on other measures such as the number of remote nodes contacted. Each lease incorporates the concept of expiration, after which the leased resource may be reclaimed if applicable. The final decision as to what lease is actually granted, or if a lease is granted at all, is made by the Linda$_m$ node. Each node is responsible for managing only its own resources and, as such, cannot make guarantees on behalf of another node. For this reason, leases are only valid for the node which grants them and are not transferable across nodes. Any Linda$_m$ node which, during the course of performing an operation, places demands on another, is responsible for negotiating any further leases.

Due to the unpredictable nature of the environment, the leases offered do not represent absolute guarantees. Rather they represent a best-effort on the part of the system to

satisfy the process' request. If circumstances change substantially, a Linda$_m$ node may revoke the lease; although this behaviour should only be employed as a last resort to avoid undermining the leasing system altogether.

For the **out** operation, once the lease expires, the tuple may be removed from the space at any time. For the **eval** operation, when the lease expires the resultant computation (if it has not already finished) may be halted and the tuple may be removed. In the case of **in**, **inp**, **rd** and **rdp**, once the lease expires the Linda$_m$ node may stop trying to satisfy the request and, assuming no match has already been found, return nothing.

5.5 Linda Semantics

The Linda$_m$ model attempts to provide the well understood Linda semantics in a mobile environment. Over the course of designing the Linda$_m$ model, however, it became clear that certain extensions to the semantics could prove useful. The extensions presented in this chapter are:

out, eval: The **out** and **eval** operations have changed somewhat due to the introduction of leases. Tuples placed into the space will no longer remain there indefinitely, and instead may be deleted from the space at any point after their lease has expired. This is vital in order to allow the control and reclamation of space in order to preserve resources.

in, rd: The **in** and **rd** operations will no longer block indefinitely but may be terminated at any point after their leases have expired. This too is necessary to avoid the indefinite consumption of resources.

inp, rdp: Remain unchanged.

Further extensions to the semantics as a result of the implementation will be presented in section 6.5 and section 7.2 will summarise these extensions.

5.6 Summary

This chapter has presented the Linda$_m$ model for providing Linda-like semantics in a mobile environment. At the heart of the model is the concept of the Opportunistic Logical Tuple Space which provides each process with the abstraction of a single tuple space. The

OLTS allows each device to possess its own tuple space, as required in section 5.2.3, which are then connected opportunistically as operations are performed to provide the abstraction of a single logical space. The opportunistic nature of the OLTS means that no explicit connection or disconnection operations are required, meeting the design principle in section 5.2.3. As discussed in section 5.4.1, the OLTS also allows change in the underlying system to be modelled as part of the normal operation of the model, rather than as an exceptional circumstance. Also described were the extensions to the basic Linda model which have been incorporated into Linda$_m$, namely leases and direct remote communication. Leases allow for fine-grained resource control to meet the design principle introduced in section 5.2.1, as well as the capacity for tuple garbage collection. Direct remote communication is provided to allow application developers to break through the OLTS abstraction when absolutely necessary.

An implementation of the Linda$_m$ model, Tiamat, will be presented in the next chapter.

Chapter 6

Tiamat

This chapter describes Tiamat, a proof of concept implementation of the Linda_m model. An overview of the architecture of Tiamat is discussed in section 6.1. This is followed by a closer examination of the three main components of the architecture: the lease manager in section 6.2; the tuple space in section 6.3; and the communications manager in section 6.4. Section 6.5 contains a discussion of the modifications to the Linda semantics made during the implementation of Tiamat.

Some of the work presented in this chapter has previously been published in [ME03].

6.1 Tiamat Architecture

The Tiamat system has been implemented in Java. Java was chosen due to the ease of portability and the fact that VM implementations are available for a wide variety of devices including mobile devices such as PDAs and mobiles phones that are expected to be prevalent in the environment (section 2.3.4). An overview of the architecture of a Tiamat instance is presented in figure 6.1. The system consists of three components: the lease manager, which is responsible for the allocation and management of leases; the tuple space, which stores the tuples; and the communications manager, which is responsible for propagating operations to and receiving responses from other remote Tiamat instances, thus implementing the Opportunistic Logical Tuple Space (OLTS).

A single Java VM can host multiple Tiamat instances. Each instance operates independently of the others and instances operating in the same VM do not share any Tiamat runtime data structures or other resources. A Java RMI [PM01] interface is also provided to allow multiple processes on the same machine to use a single space.

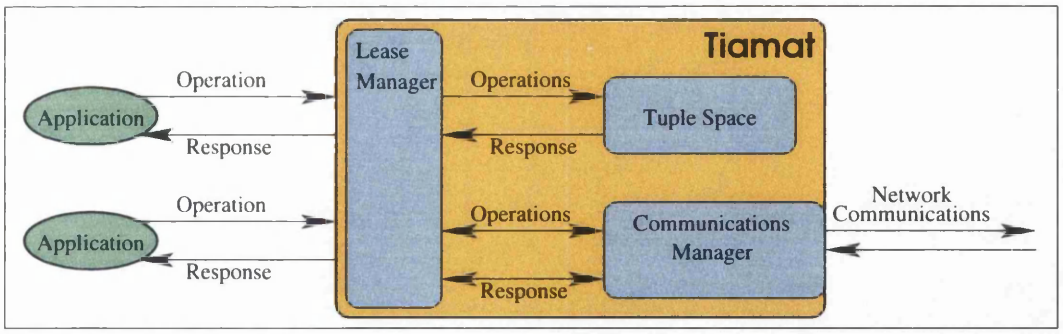


Figure 6.1: A Tiamat instance.

Semantic separation is necessary when two different applications make use of the same type of tuple, but for very different purposes. If the two applications share the same tuple space, they will begin to interfere with each other due to the reuse of the particular tuple class. In Tiamat, the separate applications would be run in separate VMs and the Tiamat instances in those VMs set to use a different port for visibility (see section 6.4.1). In this way the applications are allowed to operate without interfering.

6.1.1 Tuples

A tuple in Tiamat is implemented as a Java object which implements the provided interface `tiamat.tuples.Tuple` presented in listing 6.1. While modifying a tuple in situ might be useful for certain applications it would make it more difficult to manage resources; the system would not be able to make a static assessment of the storage requirements for a particular tuple at the time of the operation. Instead, the system would be forced to trap each modification of the tuple and ensure that it does not exceed its resource limits. The cost associated with trapping every field modification makes this impractical. To remedy this, whenever a tuple is passed to Tiamat, before any other operation takes place, a copy of the tuple is taken using the Java serialisation mechanism. As such, all tuples must implement the `java.io.Serializable` interface. To enforce this, the `Tuple` interface extends `Serializable`. The Java serialisation mechanism was chosen to remove the need to custom-craft a copy mechanism. Most Java programmers are also likely to be

```
package tiamat.tuples;
import java.io.Serializable;
public interface Tuple extends Serializable{
}
```

Listing 6.1: The `Tuple` Interface

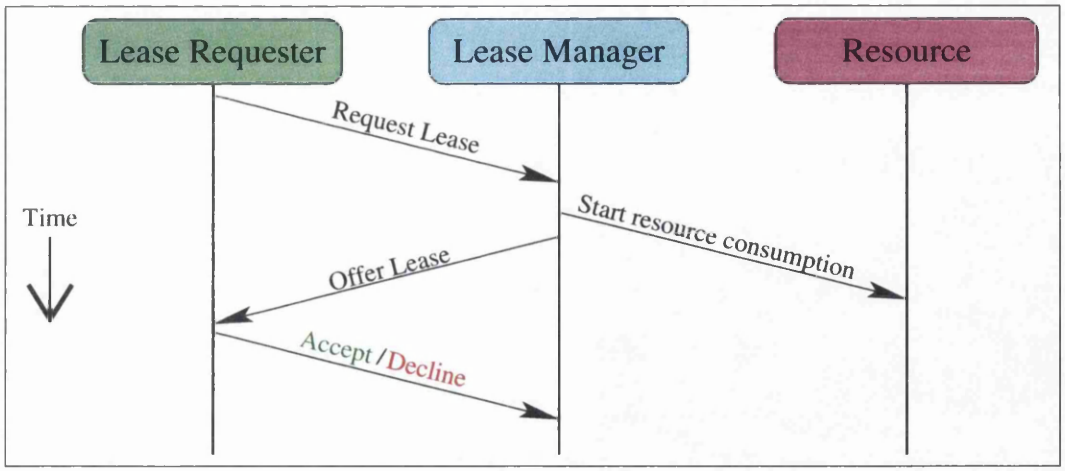


Figure 6.2: Lease negotiation time-line.

familiar with its semantics. Although copying a tuple can be a potentially costly operation (particularly for large tuples), it ensures that the application cannot modify the tuple once it has been placed into the space. It would also have been possible to provide a class which represents tuples rather than an interface. This approach, however, would make it more difficult to adapt existing applications to use Tiamat. With an interface, any object can be designated as being a tuple without having to modify the class hierarchy. Since Java does not allow multiple-inheritance existing class hierarchies would have to be modified to accommodate a tuple class.

6.2 Lease Manager

As described in the Linda$_m$ model in section 5.4.3, every operation is leased to allow for resource management. The assignment of leases in Tiamat is handled by the lease manager which acts as the first port of call for any operation.

6.2.1 Programmer Model

For the application programmer, all interaction with the lease manager takes place through Lease Requesters. Every operation in Tiamat requires that a lease requester is provided along with the tuple for the operation. The lease requester is responsible for negotiating the exact details of the lease for a single operation on the space. This negotiation consists of three stages: request; offer; accept/reject. This is shown in figure 6.2. The lease requester

requests a lease with the duration[1] desired by the application. The lease manager takes this request and offers the requester a lease based on the resources available to it. The lease that is offered may have a smaller duration than that requested if insufficient resources are available (the lease manager may refuse to offer any lease at all). It may also have a larger duration if, for example, the lease manager allocates particular resources in blocks to simplify their management. Once a lease is offered, the lease requester must either accept or decline the lease. If the lease is declined, or if no lease is offered, then no further action is taken by Tiamat. If the lease is accepted then the operation is passed to the tuple space (see section 6.3).

Where appropriate, any resources which are to be consumed should be considered to be consumed at the point of offering the lease to prevent the application disrupting the system by delaying its response. Such potential for disruption can be seen clearly in the case of leases based around time. If an application requested and was offered a lease with a long lifetime at a time when the system was under low load, but delayed its acceptance until the system was under heavy load, it could effectively usurp the authority of the lease manager.

6.2.2 Implementation

Lease requesters are represented by the `tiamat.leases.LeaseRequester` class, the code for which can be seen in listing 6.2. `LeaseRequester` is an abstract class, each subclass of which is designed to request a different type of lease (e.g., one which is limited by time or by number of remote communications permitted). Providing the `LeaseRequester` as an abstract class allows for the use of new types of lease by creating a new subclass of `LeaseRequester` for that type and modifying the lease manager to evaluate and issue leases of that type.

The programmer configures a `LeaseRequester` instance to look for a particular type

[1] As discussed in section 5.4.3 the duration of a lease may be temporal or may be defined in terms of other resources.

```
package tiamat.leases;
public abstract class LeaseRequester{
  public abstract Lease request();
  public abstract LeaseResponse offer(Lease l);
}
```

Listing 6.2: The `LeaseRequester` Class

of lease by choosing the appropriate subclass and to look for a particular duration (either temporal or another appropriate measure) by passing the appropriate parameters to the constructor. This `LeaseRequester` is then passed in during the method invocation which starts the operation. The lease manager calls the request method on the `LeaseRequester` which returns a `tiamat.leases.Lease` object representing the lease which it desires. The lease manager examines the lease and then either constructs another `Lease` object representing the lease it is willing to offer, or throws a `tiamat.leases.LeaseRefusedException` if it is unwilling to offer any lease at all. A copy of this lease is then passed to the offer method on the `LeaseRequester`. An instance of `tiamat.leases.LeaseResponse` is then returned by this method indicating acceptance or rejection of the lease. A copy of the lease is passed to ensure that the application cannot make modifications to the duration of the lease.

6.3 Tuple Space

Assuming that the lease offered by Tiamat has been accepted by the application, the tuple for the operation, along with its newly assigned lease, is then passed to the local tuple space, as described in the Linda$_m$ model in section 5.4.1. The local space will either store the tuple and lease or search for possible matches, as appropriate. The local tuple space currently provided in Tiamat is a custom-built tuple space which features extended matching semantics and a fine-grained locking mechanism for concurrent access. The main reason for custom building a tuple space for Tiamat, as opposed to using an existing tuple space implementation, was to establish exactly what was required of a tuple space implementation in order to be used to provide the Linda$_m$ semantics. The resulting requirements are discussed further in section 6.3.5. The tuple space in Tiamat can be used as a stand-alone tuple space with no modifications.

6.3.1 Matching Semantics

As discussed in section 2.2, the traditional Linda semantics only allow a search for fields in a tuple with either an exact value match (actuals) or a type match (formals). As introduced in section 3.3, in Java-based tuple spaces, the matching semantics are often extended to take advantage of Java's polymorphism and better fit the semantics of Java (and those of object-oriented languages in general). The tuple space created for use in

Tiamat will allow subclasses to be matched to tuples. It also allows for the creation and use of user-defined match semantics.

The default matching semantics provided in the tuple space work as follows: Assume that tuple A is the tuple being used for the search (i.e., the input to an **in**, **inp**, **rd** or **rdp**) and tuple B is the tuple it is being compared to for a match. The first level of comparison is at the type level, if B is not the same class as or a subclass of A, then there is no match. If it is, then the fields must be compared. Each field in A is compared to the corresponding field in B using the field's equals method. If the result is true, then the fields match. Java 'null' values are used as wild-cards in the search tuple, so if an object reference field in A is null then it will match any value in the corresponding field in B.

The matching mechanism will always prefer an exact class match to a subclass match. This is done to try and increase the number of potential matches the system can make. If the matching mechanism were to match on subclasses first, then any later requests for tuples specifically of that subclass may not find a match.

Using a different set of semantics, for example searching for an exactly null value, is made possible through the `AntiTuple` interface presented in listing 6.3. The `AntiTuple` interface contains two methods: match; and tupleID. The match method takes a single `Tuple` and returns true if the given `Tuple` instance matches the `AntiTuple` instance and false if it does not. The `tupleID` method returns a `Class` object representing the class of object being sought by this `AntiTuple`. By implementing the `AntiTuple` interface and providing an implementation for the match method an arbitrary set of matching semantics can be provided. In order to know what class is being searched for, an appropriate `Class` object must be provided to the `AntiTuple`; the interface does not explicitly define the mechanism for doing this.

Allowing the match method to contain arbitrary code raises the possibility of the `Tuple` object which is passed in being modified. This would result in an in situ modification, already identified as being undesirable in section 6.1.1. To avoid this situation, the match

```
package tiamat.tuples;
public interface AntiTuple extends Tuple{
    public boolean match(Tuple t);
    public Class tupleID();
}
```

Listing 6.3: The `AntiTuple` Interface

method is passed a copy of the `Tuple` instead.

6.3.2 Eval

Often overlooked in many tuple space implementations is the **eval** operation. However, it can provide a useful mechanism in a mobile environment for allowing processor impoverished devices to offload computational tasks to more powerful machines without having those machines know the nature of the computation beforehand.

The tuple space used in Tiamat provides the **eval** operation in the form of an interface presented in listing 6.4. The `tiamat.tuples.Evalable` interface extends `Tuple` and exports a single method doEval. When an **eval** operation is started, the tuple in question is passed to a pool of threads. At some point, one of these threads will take the tuple and call the doEval method. This method should contain the application level code. Once the doEval method returns, the tuple is placed into the space as if it were part of an **out** operation.

While this offers a quick and simple way of providing the **eval** operation, it has a number of drawbacks. The fact that the user can put arbitrary code into the doEval method raises the potential for deadlock, livelock or infinite loops to arise, all of which provide a drain on resources. While the **eval** threads can be (and are) kept at the lowest priority to ensure they have minimal impact on the operation of the rest of the system, this does not altogether solve the problem — even at low priority the threads will still be consuming resources. It also makes providing meaningful leases to **eval** operations very difficult. Static code analysis to determine how long an arbitrary piece of code will take to complete could only provide a guess at best and is, in fact, an instance of the halting problem. To compound matters, since Java no longer allows the forceful termination of threads, once a lease is expired there may be no way of bringing the **eval** operation to a close.

Determining how long an arbitrary piece of code will take to run or identifying badly behaved code through static analysis is incredibly difficult, if not impossible, and is far

```
package tiamat.tuples;
public interface Evalable extends Tuple{
    public void doEval();
}
```

Listing 6.4: The `Evalable` Interface

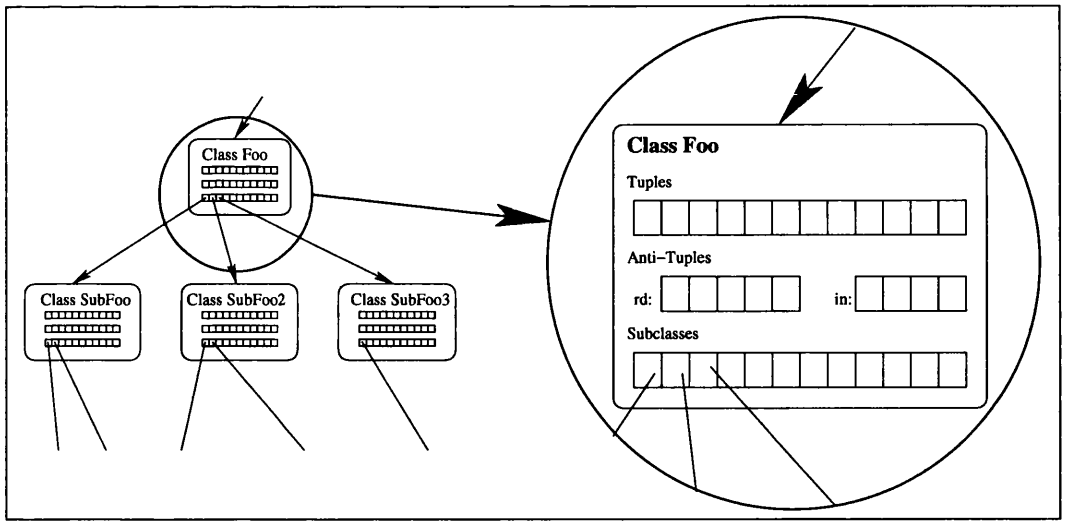


Figure 6.3: Tiamat tuple space — core data structure.

outside the boundaries of this work. Developers should already be aware of these issues as they are also present when making use of Java threads. For the time being, it is suggested that application developers exercise the same caution in the development of doEval methods as when they are developing threads. Also, the lease manager must be pessimistic about the potential costs of an **eval** operation.

6.3.3 Core Data Structure

The core data structure of the tuple space is a tree which represents the hierarchy of the Java classes which are in use in the space. A portion of the structure is depicted in figure 6.3. Each node represents a single class and contains four lists: an unordered list of all the tuples of that class currently stored in the space; two unordered lists containing anti-tuples[2] for that class which are still awaiting a match; and a list containing references to the nodes for the subclasses of that class. These nodes are also stored in a hash-table keyed on class, to allow for quick retrieval of the class during searches. Note that although `Object` objects cannot be placed into the tuple space (they cannot implement the `Tuple` interface), an entry is still maintained for them. This is used since `AntiTuples` can still be used to search for items of class `Object` (indicating they will accept *any* tuple which meets their other criteria).

[2]An anti-tuple is a high-level, general term for any tuple which has been provided as the input for an **in**, **inp**, **rd** or **rdp** operation. This should not be confused with the specific, implementation level, Java interface, `AntiTuple` which is a tuple that is implemented to provide extended matching facilities. An anti-tuple can be a class which implements `Tuple` (indicating that it uses the default matching semantics) or `AntiTuple` (indicating it provides its own matching semantics).

Separate lists are kept for those anti-tuples which represent **rd** operations and those that represent **in** operations. Splitting the anti-tuples into two can raise the number of matches created by a single tuple, since an individual tuple can be used to satisfy multiple **rd** operations but only a single **in** operation. During a matching operation, therefore, tuples are checked against the **rd** list before the **in** list.

Unlike the list of subclasses, there is no need to maintain an explicit reference to the superclass as this can be obtained when needed from the Java system.

Operations

The function of the data structure is best explained through examination of the various tuple space operations. Throughout the following discussion it is assumed that the system is operating in a single-threaded environment. For a description of how the system copes with concurrency, see section 6.3.4.

For the **out** operation, assuming the tree is already constructed, the system gets the `Class` object for the tuple and uses the hash-table to retrieve the appropriate class entry. The lists of anti-tuples contained in that entry are then scanned. If a match is found in the **in** list, the tuple is returned to the caller of the **in** operation, the anti-tuple is removed from the list and the **out** operation concludes. If no matching anti-tuple is found or the only matches are in the **rd** list, then the system works its way up the tree of classes and checks for matching anti-tuples in the superclasses. Again, a matching **in** anti-tuple will result in the conclusion of the operation. If the only matching anti-tuples are from **rd** operations then the process will continue until the top of the tree is reached. At this point the system returns to the node at which the search began and adds the tuple to the list of tuples stored there.

For the **in** and **rd** operations, assuming the tree is already constructed, the process is similar to that of the **out** operation, only in the opposite direction. The system begins by retrieving the hash-table entry for the class and searching the list of tuples there. If no match is found, then the tuple lists in each of the subclasses are searched. If no match is found there, then the subgraph for that subclass is searched. Once the operation has searched all of the subclasses, if no match has been found, it returns to the starting node and adds the anti-tuple to the appropriate list. The **inp** and **rdp** operations omit this final step. Since these operations return immediately if no match is found, there is no need to store the anti-tuple.

Construction

Construction of the tree takes place on-the-fly as operations are performed. Most of the construction is done during **out** operations. When an **out** operation is performed and the appropriate node is not located, then that node is constructed. The node must then be attached to the appropriate point in the tree. This is done by taking the node's superclass and checking the hash-table for a node representing it. If the superclass already has a node then all that remains is to update the subclass list on the superclass's node. If the superclass is also lacking a node, then it too must be constructed and the system must then check for its superclass. This continues until a class is reached for which a node already exists and which contains a matching anti-tuple or the top of the class hierarchy is reached.

For the **in** and **rd** operations, if no node is found for the appropriate class, then that node, and only that node, must be constructed. There is no point in traversing up the hierarchy as only subclasses can provide a match. There is no point in (nor mechanism for) searching down the hierarchy, as if there were any matching tuples from subclasses, then the node would already exist.

For the **inp** and **rdp** operations, if no node is found for the appropriate class, the operation can return since there are no tuples of either this class or any of its subclasses in the space. No tree construction takes place in this instance.

6.3.4 Locking Mechanism

The description above assumes a single thread of operation. The Tiamat system is designed to operate in concurrent environments providing generative communications to multiple applications across a number of devices. For this reason it was important that the system deal with concurrent accesses to the tuple space. While a global lock on the tuple space is a possible solution, it would have been impractical and limiting. If, for example, two applications were looking for tuples in separate branches of the class hierarchy, it would make sense to allow them both to search the space at once. In order to provide finer grained concurrent access, the appropriate locking mechanisms had to be built into the tuple space itself.

The primary goal of the locking mechanism designed for this tuple space is to allow concurrent accesses and modifications to the core data structure while maintaining one invariant: there should never be a situation where a tuple and a matching anti-tuple should

both be stored in the space.

Initially it will be assumed that all necessary portions of the data structure have been constructed. When an **out** operation is performed, the system starts by retrieving the appropriate node from the hash-table and then attempts to take out a node-level lock on that node. Each node in the tree has its own node-level lock which ensures that only one process can be reading or modifying the lists in that node at any given time. Once the node-level lock is obtained, the system searches the lists as before. If no match is found, then the superclass's node is retrieved and the system attempts to obtain a node-level lock on that. Note that the node-level lock on the original node is maintained (the reasons for which will become clear below). Once the lock is obtained, the superclass's node is searched. If no match is found then the lock is released and the next superclass's node is retrieved and a lock taken out on it. This continues until either a matching anti-tuple from an **in** operation is found or the top of the hierarchy is reached. The node-level lock on the original node is maintained throughout. Once searching is complete, the tuple is added to the list of stored tuples in the original node (assuming it was not matched during the search) and the node-level lock is released.

For the **in** and **rd** operations, the locking operation is almost identical, with one exception. Once the starting node has been retrieved, locked and searched, the operation's anti-tuple is added to the appropriate list and the lock is released. The system then attempts to get the node-level lock for the first subclass which is then searched and released. This continues in a depth-first search of the entire sub-hierarchy from the initial node.

As the operations proceed, the **in** and **rd** will become blocked by any **out** operation; however, since they will not be holding any locks, the **out** operations will continue unhindered. Once the **out** operations have finished searching the tree, they release their locks and the **in** and **rd**s are allowed to continue. Since the **in** and **rd** operations are blocked by an ongoing **out** they cannot skip over that class until the appropriate tuple has been placed into the space, thereby ensuring that a potential match cannot be overlooked.

6.3.5 Using Alternative Tuple Spaces

As was stated at the outset of this section, part of the motivation for implementing a new tuple space for Tiamat was to investigate what demands were made of such a tuple space. Having built the space, it is now clear that, while other tuple space implementations could be used at the heart of Tiamat, there is one complicating factor — leases. The Linda$_m$

model requires that every operation be leased. In practical terms, this means that every tuple (which is representative of an operation) will have an associated lease which must be stored along with it. Any implementation must therefore be able to manage the association between tuples and their lease. In the tuple space which was custom built for Tiamat, leases are incorporated into the design. In Tiamat's tuple space implementation, leases are wrapped up with the tuples when they are placed into the space. The majority of tuple space implementations, however, do not expect to have to deal with leases. There are three suggested approaches to address this issue: bundling; hashing; and source-code modification.

Bundling involves placing the tuple and its lease in a wrapper object which is then passed to the space as if it were a tuple. As long as the matching semantics provided ignore the lease and look at the contents of the tuple during matching this will work. However, this could make garbage collection impractical, if not impossible. Assuming the space provides only the basic tuple space operations, the collector would first need to know a set of operations which, when performed, would retrieve every tuple from the space at least once. Assuming such a set of operations could be determined, the collector would then have to examine each lease, decide which tuples are garbage and then remove them from the space. If the space provides some of the extended Linda operations (see section 3.3), **copy-collect** in particular, or provides some way to iterate over all of the tuples in the space, then garbage collection is not difficult.

Hashing would involve setting up an auxiliary hash-table. The hash-table would be keyed on tuple and would store the lease associated with a given tuple. The tuple itself would then be stored in the space. This workaround would allow for the easy scanning of lease information for the purposes of garbage collection. In addition, since a reference to the tuple is already stored along with the lease as the hash-table key, retrieving the appropriate tuple for each lease is easy. This workaround could fail in cases where the tuple space takes a copy of the tuple to be stored. Depending on the tuple's `hashcode` and `equals` methods, once the copy has been taken, it may no longer be possible to match it to the original tuple or its associated lease. Tiamat cannot defer the copying process to the tuple space as it must have a static copy of the tuple for the lease manager to decide on an appropriate lease. There could also be complications if the space made any modifications to the tuple it was storing as, again, it may not be possible to match the tuple to its lease afterwards.

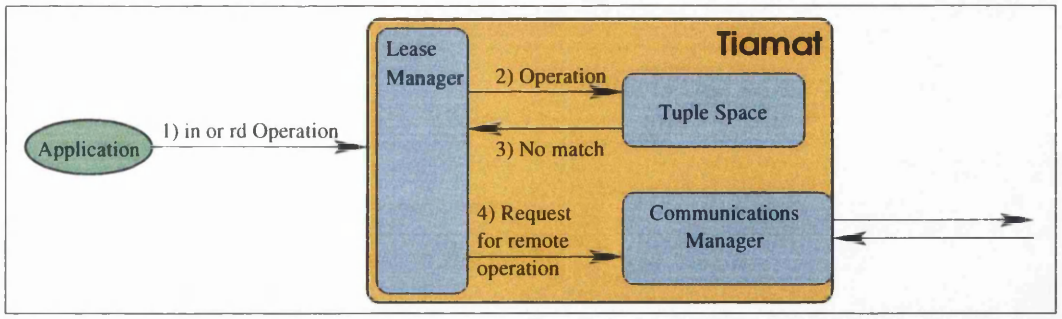


Figure 6.4: Unsatisfied operations are passed to the communications manager.

Modification is the most drastic workaround and would entail modifying the behaviour of the tuple space to deal with leases in an appropriate manner. Although this would result in the best outcome with the fewest complications, it would only be possible if the source code to the tuple space was available. It is also likely to be the most time-consuming and labour-intensive of the workarounds depending on the complexity and nature of the particular tuple space implementation.

6.4 Communications Manager

As described in the Linda$_m$ model (section 5.4.1), in the case of a basic **out** or **eval** operation, once the tuple has been inserted into the local space, no more action is taken. In the case of the **in**, **inp**, **rd** or **rdp** operations, there may be a further stage. If no appropriate match is found in the local tuple space, these operations are passed to the Communications Manager as shown in figure 6.4. The Communications Manager is responsible for establishing communications with, receiving operations from, and propagating operations to, other Tiamat nodes. The following sections describe an initial prototype of the communications manager (section 6.4.1) and an improved version (section 6.4.3).

6.4.1 Initial Prototype

Operational Description

An operation is passed to the Communications Manager only if it cannot be satisfied locally. The communications manager determines which nodes are visible (see section titled "Visibility" below) and then contacts each visible node in turn and propagates the operation to them until either the operation is satisfied or all visible nodes have been contacted.

As described in section 5.4.3, leases are only valid for the Tiamat instance in which they are issued, since no instance can be responsible for the allocation and management of the resources of another. As such, the Communication Manager is responsible for negotiating new leases with the remote instances. The `Lease` object currently attached to the tuple is given to a `LeaseRequester` and this is passed along with the tuple to the remote instances. The `LeaseRequester` will request a lease identical to the one it has.

Visibility

Visibility is a core concept in the Linda$_m$ model (see section 5.4.1), with its exact definition being left up to the implementation. A Tiamat instance is defined as visible if it responds to a multicast on a known port. This approach to visibility is not necessarily the most reliable, as there may be instances that can be communicated with, but do not get the multicast, because standard multicast is a lossy protocol. However, lossy multicast is appropriate for a simple implementation, works well in small scale networks and is ideal for a proof of concept[3].

More elaborate schemes are possible using other instances as proxies to forward information, most likely forming some sort of overlay network. This, however, is non-trivial and is discussed further in section 9.1.

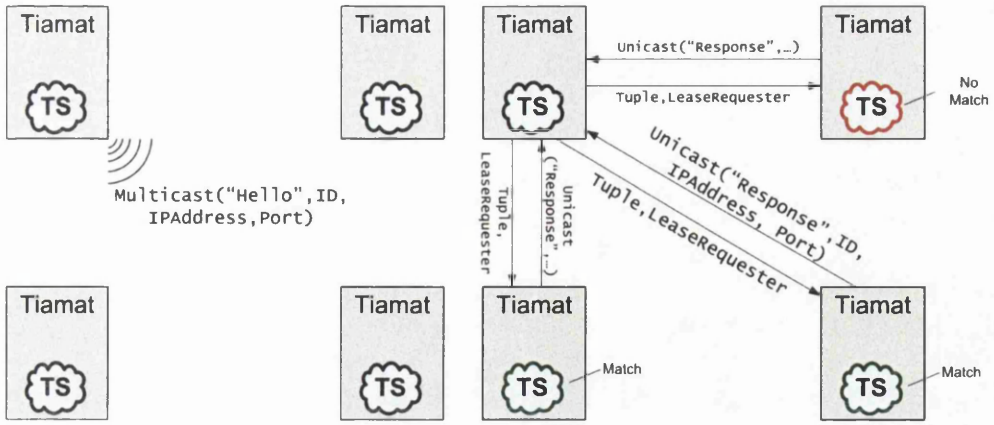
As mentioned in section 6.1, multicast can also be used to provide semantic separation for applications. If two applications, A and B, which use the same class of tuple for different purposes[4] both want to use Tiamat, they can avoid interference by each having their Tiamat instances use a different multicast port. Since the Tiamat instances being used by A will now not respond to multicasts coming from instances being used by B, they will not be considered visible and no interference will occur.

6.4.2 Protocol Operation

When the communications manager receives an operation which needs to be propagated to other nodes, the first step is to find out which nodes are visible. This is achieved by sending out a multicast packet with the appropriate message as shown in figure 6.5(a).

[3]This mechanism was also employed in favour of an existing discovery protocol as it is provided by default in the JDK and has low overheads in terms of space used in the UDP packet which will become important in section 6.4.3.

[4]Determining where and when this can happen is a deployment issue. The system architect would need to identify the potential clash before-hand. This is a general problem with any generalised data store as the storage is independent of semantics. Exact mechanisms for determining these clashes are outwith the scope of this dissertation.



(a) Originator multicasts to discover other nodes.

(b) Node receives responses and propagates operation.

Figure 6.5: Initial prototype discovery operation.

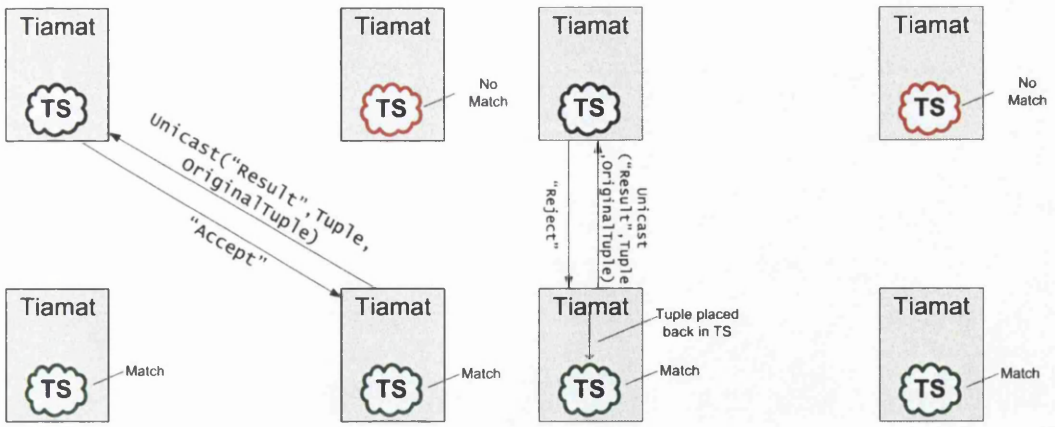
Any visible nodes will respond to this packet via unicast. Once they are in communication, the communications manager will negotiate a lease with and pass the appropriate tuple on to the receiver as shown in figure 6.5(b). If any receiving node has a match for the operation, then it contacts the originating node via multicast once again to inform it of the match. Assuming this is the first such response, the communications manager will accept the result and return it to the calling application, depicted in figure 6.6(a). If any subsequent nodes contact the originator with a result, then the communications manager will reject them as shown in figure 6.6(b).

Cache Lists

While the above implementation of visibility is simple, it is also inefficient — every remote operation requires the same phase of multicasting even if nothing has changed. In order to remove some of this inefficiency, Tiamat implements cache lists where references to Tiamat instances that have previously responded to the visibility multicast are retained.

When a remote operation has to be performed, Tiamat begins by attempting to contact the Tiamat instances in the cache list first. If any instance on the cache list cannot be reached via unicast, its entry is removed. Only if the operation has not been satisfied when the end of the list is reached does the system resort to performing the multicast. Any instances which respond to the multicast are added to the end of the list.

This mechanism of removing Tiamat instances which do not respond, and adding new instances to the end of the list, also has the effect of pushing those instances which have



(a) Originator receives and accepts first response.

(b) Originator rejects subsequent response.

Figure 6.6: Returning of results.

remained in contact for the longest to the top of the list. In the case of devices whose movements are closely related (for example, a person's phone and their PDA), this has the advantage of ensuring that the instances at the top of the list are most likely to respond. This is an unexpected benefit of the list management.

6.4.3 Improving the Communications Manager

The above implementation results in one core weakness which must be addressed, only nodes which respond to the initial multicast or are already in the cache list will receive the operation. The protocol does not make any provision for propagating operations to nodes which appear after that point. This can be broken down into two smaller problems: detection, noticing that a new node has become visible; and reconciliation, passing on any outstanding operations which the new node is not presently aware of. These two problems are addressed by heartbeats and synchronisation respectively.

Discovery: Heartbeats

In the original implementation, a node only checks for visible nodes at the time the operation takes place. It would, in the general case, be undesirable for each operation to depend on another, subsequent operation in order to detect that new nodes have become visible as the time period between operations is completely unpredictable. A more reliable mechanism is needed and this is where a heartbeat comes in.

In the simplest implementation, every node in the system would emit a periodic heartbeat via multicast. By monitoring the set of heartbeats received, a Tiamat instance can keep track of the set of visible nodes. However, this approach is not very scalable. If large numbers of nodes are present, then the amount of heartbeat traffic could drastically reduce the amount of bandwidth available for normal application-level traffic and have a negative impact on the performance of the system. Also, where the degree of change is low, the amount of unnecessary heartbeats being generated represent a significant waste of energy and other resources in the system as a whole.

The standard for 802.11 [OP99] wireless ad-hoc networks solves this problem by having a single node be responsible for broadcasting the beacon packet which maintains the network. The beacon packet is broadcast at known intervals so that, if a packet is not seen, one of the other network participants can take on the responsibility within a short, random timeframe (the randomness being used to reduce the chances of clashing). This approach is better in terms of scalability, but places all the resource costs of maintaining the network on a single node. This also means that clients can only perceive a change in the presence of the node currently responsible for emitting the beacons. If one of the other nodes should leave, there is no mechanism in place for detecting this.

In a different, but not entirely dissimilar, situation, fireflies can be seen to exhibit the desirable property of global synchronisation without centralisation. A firefly flashes through the buildup of a mixture of chemicals. Once the buildup of chemicals reaches a certain threshold they react releasing a bright flash of light. A short time before flashing, the firefly reaches a "point of no return" from which point a flash is inevitable. At any time up until that point the firefly can "abort" the current flash and restart the process. When gathering in numbers the fireflies attempt to flash in unison. This is achieved using a very simple algorithm[5]. Each firefly monitors its surroundings looking for flashes. If it sees a certain threshold of light during its chemical buildup, it will abort and start again (assuming it has not reached the point of no return). The end effect of this algorithm is that all the fireflies in the group wind up flashing at the same time.

Drawing on inspiration from the mechanism fireflies use to synchronise their flashes, Tiamat deploys a variant of the system used in 802.11 networks in which each node takes a fair share of the effort needed to maintain the heartbeats. The operation of the system is depicted in pseudo code in figure 6.7. The operation of an individual node will be

[5] A simplified version of this algorithm is described here. For more details the interested reader can refer to [Res94].

```
Idle:
    TimeToBeat = 2 mins
    while(TimeToBeat-- != 0){
        if(beat heard){
            if(new node) add to list
            goto Active
        }
    }
    send beat
    goto Idle

Active:
    TimeToBeat = square_root(TimeToBeat)
    while(TimeToBeat-- != 0){
        if(beat heard){
            if(new node) add to list
        }
    }
    send beat
    goto Passive
```

```
Passive:
    TimeToBeat = 2x
    while(TimeToBeat-- != 0){
        if(beat heard){
            if(new node) add to list
            else set counter to 0
            ConsecutiveBeats = 0
            if(beat is passive){
                if(TimeToBeat < x){
                    TimeToBeat += x
                }
            }
        }
    }
    send beat
    increment counter for all nodes in known list
    remove any nodes with count of 3 from list
    ConsecutiveBeats++
    if(ConsecutiveBeats > 3) goto Idle
```

Figure 6.7: Pseudo-code for heartbeat algorithm.

explained first, followed by a description of the overall system behaviour. Some experimental evaluation of this mechanism is described in the next chapter along with some improvements to the basic system.

Each node has three states: idle; active; and passive. These states, and the transitions between them, are shown in figure 6.8. Every node begins in the idle state. The node is currently unaware of any other nodes in the surrounding area and has not heard any heartbeats from other nodes. In the idle state the node will send out an idle beat periodically with relatively low frequency to reduce power consumption during extended periods of disconnection (once every two minutes in the current implementation, although this is configurable at runtime). The heartbeat packet contains the ID for the node, its IP address, the port on which it can be contacted and its current state[6]. The node remains in the idle state until it hears any heartbeat from any other node. At this point, it switches to the active state and the new node is added to a list of known nodes[7]. The purpose of the active state is to make that node's presence known quickly. The node takes the square

[6]State plays an important role in this algorithm and, as such, the state of a node is often used to describe it and any heartbeats it emits while in that state. For example, a node in the active state would be described as an active node and would be said to emit active beats.

[7]This list replaces the cache lists above, but has the same functionality.

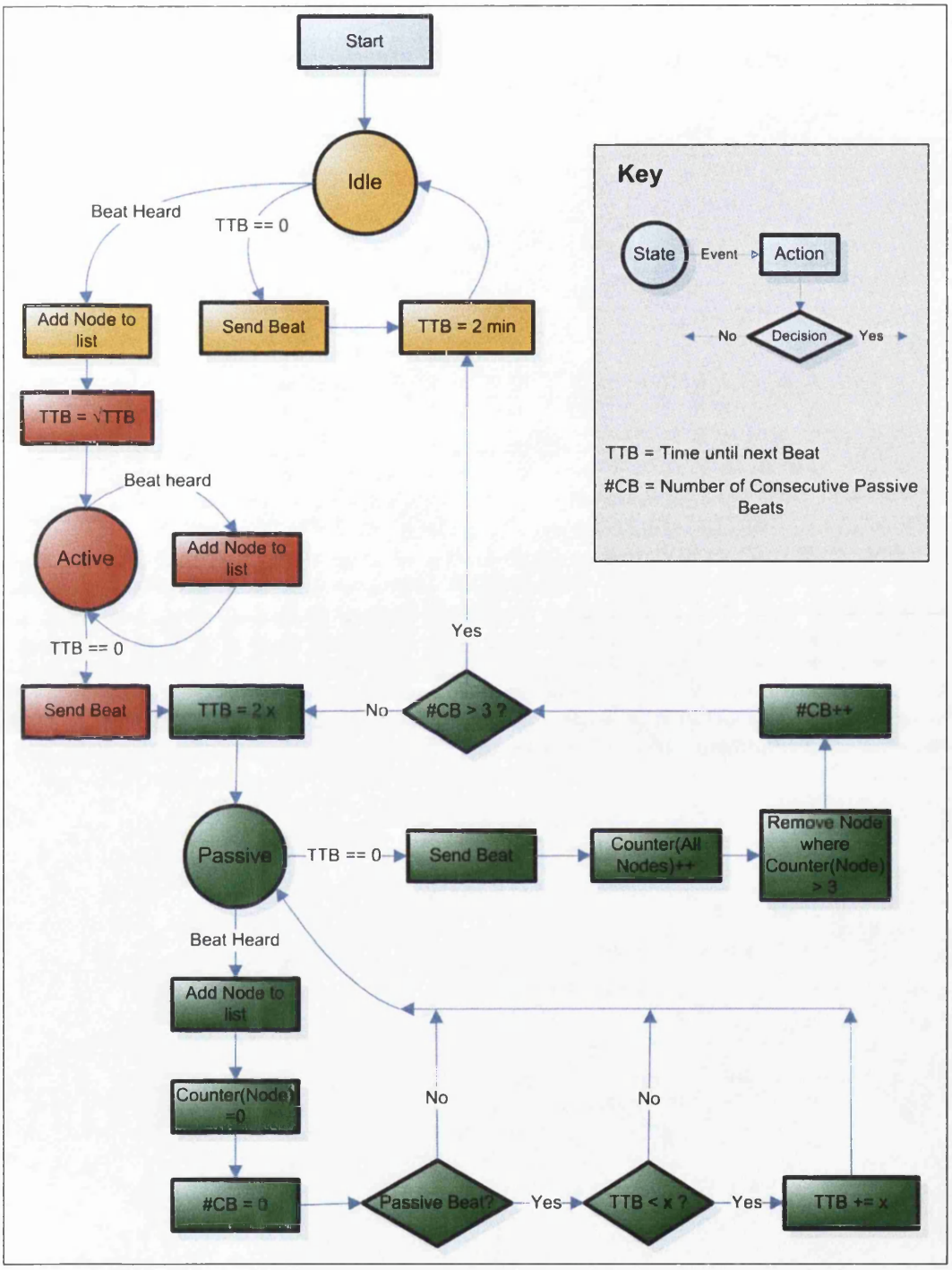


Figure 6.8: Heartbeat state transition diagram.

root of the current time left until its next heartbeat and uses this as the new time until its next heartbeat[8]. Once a node has emitted an active heartbeat, it switches into passive mode. In passive mode, the node is attempting to enter a loop of heartbeating in which each node takes a turn. The mechanism for this is as follows. The node delays for a given time period, say $2x$, before emitting its next heartbeat. This delay is split evenly into two distinct phases. During the first x of the delay the node will only note any new nodes in its known nodes list, but will not change its behaviour, nor its state, in any way. If, at any point during the second period it sees another passive beat, then it will add x to its current timer. No action is taken on active or idle beats save to add that node to the known nodes list if it is not already known. Only delaying in the second half of the delay ensures an upper bound for how long the node will be delaying for. This ensures the node cannot be continuously usurped by newly arrived nodes (who will delay only by $2x$ when first entering the passive state) and ensures the responsiveness of the system in the face of nodes departing or failing. The value chosen for x affects the rapidity with which the system detects change: the smaller x is, the faster the system reacts, but the more network traffic it generates over a given time frame. Since this value is configurable, Tiamat allows the application developer to decide upon the tradeoff. A passive node which sees no other heartbeats in between a configurable number (usually 3) of its own passive heartbeats will assume there are no nodes nearby and revert back to the idle state to conserve energy.

In order to allow the detection of node departure, the list of known nodes contains a counter, when a node sends out a passive beat it increments the counters for all the nodes in its known nodes list. Whenever it sees a heartbeat from a node, the counter is reset to zero. If the counter reaches some threshold, then the appropriate node is removed from the known node list. Setting the threshold low gives an aggressive eviction strategy which will be responsive to change, but will suffer from premature eviction if heartbeats are being lost in the network or nodes briefly move out of visibility before returning. A higher value results in fewer premature evictions, but more stale node entries in the known node list. A threshold of three was chosen for Tiamat as it was felt this gave a good balance between caution and responsiveness.

The overall effect of this algorithm is best described in an example system of two nodes, A and B, that both begin in the idle state. A and B come into communications contact

[8]The square root is an arbitrary choice here, the important point is that the node should not use a fixed value. If a fixed value were used and a number of idle nodes happened to all hear the same heartbeat, then all the idle nodes could flood the network with their responses. The square root ensures a reasonable random spread of responses.

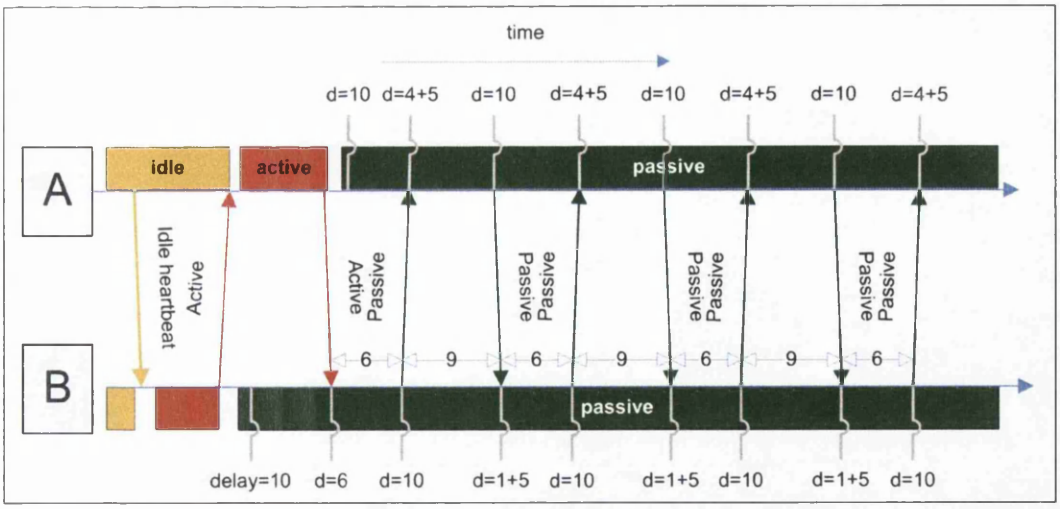


Figure 6.9: Heartbeat operation with two nodes.

with one another as shown in figure 6.9. At some point one of the nodes, in this example A, emits an idle beat. B detects this idle beat and switches into the active state. Shortly thereafter, B emits an active heartbeat and switches into passive mode. Node A sees the active beat from B, switches into active mode and emits an active beat of its own. At this point both nodes know about each other (i.e., their known node list is populated), they are both in passive mode and have delayed by $2x$ (in this example x is 5, the time units are undefined) before emitting their first passive beat. Since B entered the passive state before A, it will emit the passive beat before A, and delay by $2x$ again. When A sees this beat it adds x to its current delay, this still leaves it with a delay of less than the $2x$ of B so it will beat next. When A beats, it delays for $2x$ again and B adds x to its delay. This pattern continues with each node taking a turn to beat. The exact period of the passive heartbeats will depend on the timing of the various events but the system is guaranteed to emit a heartbeat every $2x$ time units in the very worst case, and no more often than every x time units[9].

To continue the example, a third node, C, now comes within contact of the other two nodes. There are two possible scenarios: either C will see a beat from A or B before it is able to send out an idle beat; or C will send out an idle beat before seeing a beat from A or

[9]It is possible with this algorithm for two nodes to heartbeat at the same time (or more accurately within d of each other, where d is the network induced delay in the heartbeat packets). In this instance both nodes delay by the same amount and so will beat at the same time again next time. This situation represents a waste of effort, but is also unlikely (the exact timing of heartbeats is largely random as the various nodes are started independent of one another). As such, it was felt that avoiding it would only complicate the existing algorithm for little end benefit and so no workaround was incorporated. It might be possible to have nodes in such a position induce a random element to their next delay, but this may only succeed in them colliding with the beats of other nodes.

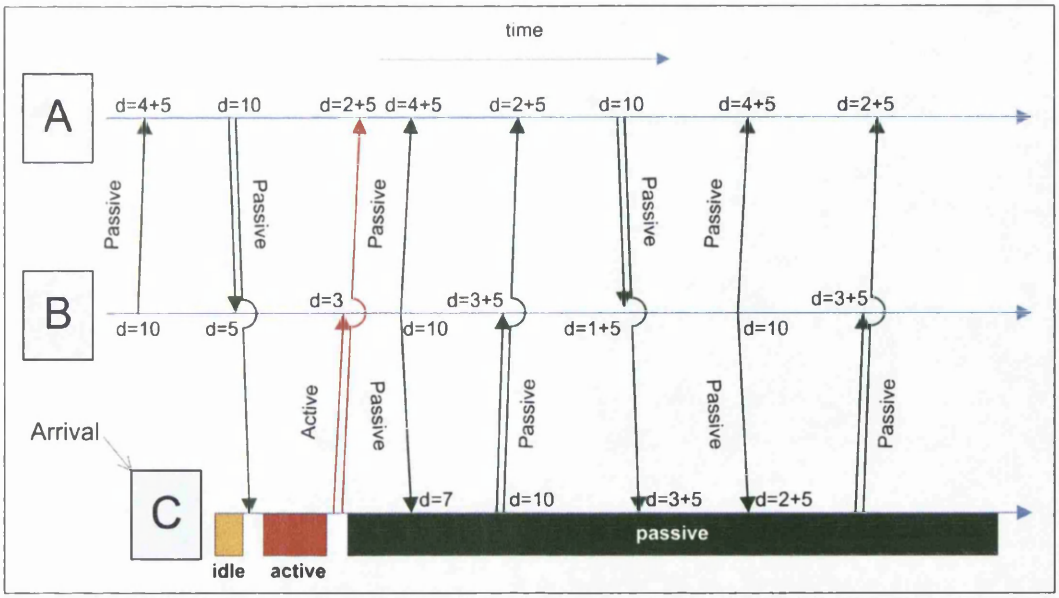


Figure 6.10: Heartbeat operation as new node arrives.

B. The only difference between the two cases is that, in the latter case, A and B will know about C before C sends out its active beat. Other than that, they operate identically, so the former instance will be considered here. The example is shown in figure 6.10.

C sees the passive beat from one of the nodes already in the cycle, in this case A. It switches to active mode and, soon after, emits an active beat and switches to passive mode with a delay of $2x$. A and B see this active beat, but do not change their behaviour, they simply note the node in the known nodes list (if C had managed to send an idle beat before seeing a passive beat from A or B, then they would already know about it and would not add it at this point). As can seen in the figure, C then falls into step within the passive cycle with A and B and each node now sends one out of every three heartbeats.

If C now departs from the system, then one of the other nodes will pick up the slack, as shown in figure 6.11. Once again, the timings may vary from the example given, but in the case of a node failing to heartbeat, another node is guaranteed to pick up the slack within $2x$ time units of a missed beat in the worst case, and within x of the missed beat in the typical case. In terms of responsiveness, the time taken by A and B to register the departure of C is roughly given by the formula:

$$|P| \times x \times Thresh_{evict}$$

Where $|P|$ is the number of passive nodes currently left in the passive cycle, x is the passive delay constant noted above, and $Thresh_{evict}$ is the threshold for evicting nodes from the

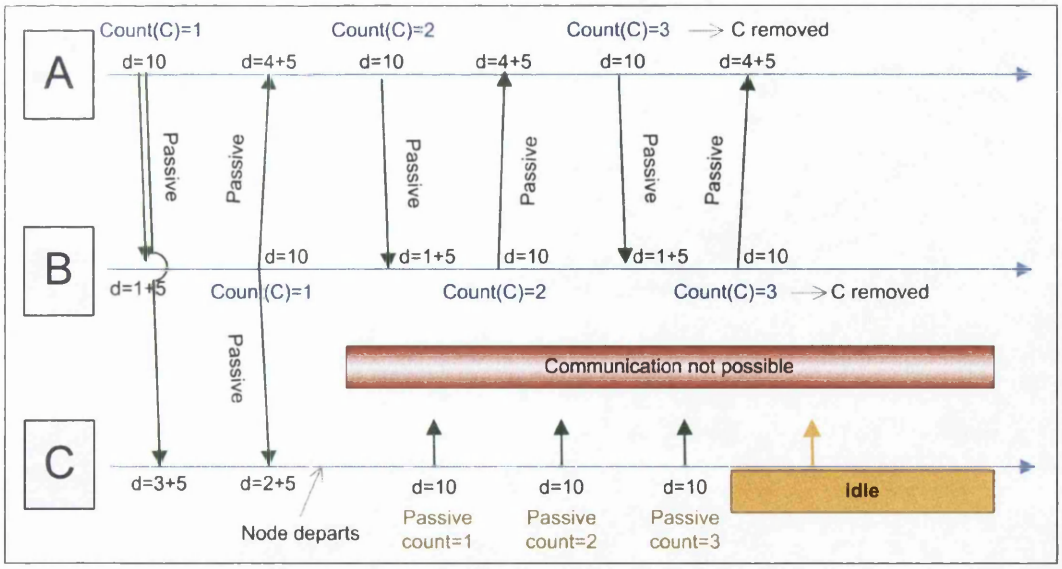


Figure 6.11: Heartbeat operation after node departure.

list of known nodes.

C, which is now on its own, will revert back to its idle state after a specified number of passive beats unless it encounters other nodes. In other words, its responsiveness is given by the formula:

$$x \times Thresh_{idle}$$

Where x is the passive delay constant and $Thresh_{idle}$ is the number of consecutive passive heartbeats which a single node will emit before reverting to the idle state. It is interesting to note that an isolated node will typically realise it is alone before any passive nodes it has left behind realise it has gone.

Reconciliation: Anti-Tuple Synchronisation

While the heartbeats allow nodes to detect change in the set of visible nodes, there is still the issue of reconciliation to be dealt with. There are a number of solutions to this issue, each with its own advantages and disadvantages. Independent of these solutions is the decision of whether or not a node should pass on anti-tuples which it has received from other nodes. How fruitful this is depends on how the results from tuples are routed back. If it is the case that the routing mechanism is closely tied to visibility (as is the case in Tiamat) and any contactable node is very likely to also be visible, then there is little point in passing on tuples from other nodes for two reasons: if the originating node is not visible, then, even if the new node contains a satisfying tuple, it will likely be unable to return it;

and, if the originating node is visible, then it will be able to do the synchronisation itself and the system will avoid doing repeated work. This latter case is the approach used in Tiamat.

Once that initial decision is made, the focus becomes the tradeoffs between the various solutions which lie across three main dimensions: computational cost; space overhead; and network cost. The simplest solution is to send a newly discovered node a copy of all of the anti-tuples[10] stored within the local space and receive a copy of their anti-tuples in return. Although obviously costly in terms of network usage, this places no storage overheads on the system. The solution also incurs some computational cost as each node must scan the list of anti-tuples it receives and extract only the new ones. This solution will be referred to as *send-all.*

If it is important to reduce network usage, an alternative is to timestamp every anti-tuple and pass on only those which are new to the new node. This solution reduces the network cost since duplicate anti-tuples are not sent. Computational cost is also kept low, although there is still some expenditure, usually to organise the anti-tuples in such a way as to allow for quick retrieval. However, this introduces a potentially large storage overhead. For each anti-tuple a timestamp must now be kept (this can actually be a logical timestamp so its cost is dependant on the number of anti-tuples likely to be active at any given time), and for each node, we must keep a note of the highest timestamp received from that node. This list could be potentially very large if the number of distinct devices seen during any time period is high. As such, a mechanism to manage the size of that list over time must be employed. One solution would be to drop any entries more than a given time period old. For any nodes whose entries are removed, the system would simply fall back to a complete exchange mechanism.

This approach also incurs one problem as a result of the resource management mechanism in Linda$_m$. In a traditional timestamping system, the assumption is, that if you have seen the item with timestamp x you have also seen all the items with timestamp $< x$. In Linda$_m$ and Tiamat, it is possible, through the leasing mechanism, for a node to refuse to accept anti-tuples. There are no guarantees that because one anti-tuple is rejected, all subsequent ones will also be rejected. As such, the set of stored anti-tuples at a remote node can be more "patchy" than the basic timestamping algorithm assumes. There are two possible approaches to dealing with this. One is to adopt a *once refused,*

[10]Remember, tuples are not replicated in the Tiamat implementation, so only the anti-tuples need to be passed across.

always refused policy, so that timestamps can be used once again. The other is to keep more complicated lists of exactly which timestamps have been seen and which have not. The former means that anti-tuple refusal is permanent, even if the resources to accept that anti-tuple become available in the future, but does not introduce the storage and computational overheads of maintaining the more complex lists of timestamps.

Timestamps also do not help detect the removal of anti-tuples. For example, if the originating node has found a match for a given anti-tuple, it will be removed from the space. However, this fact is not conveyed in the timestamping alone. As such, a secondary mechanism would have to be put in place to convey this information if desired. Note that this information is not required for operation — any stale anti-tuples will be caught by the originator when a match is returned, as depicted earlier in figure 6.6(b) — but does reduce wasted effort in the system.

If storage is at an absolute premium, then storage overheads can be reduced at the expense of added computation through the use of checksums [SM02b]. When a synchronisation takes place, anti-tuples are divided into predetermined groups, for example by class. For each group of anti-tuples a checksum is calculated over the group. The other node does the same. The checksums are then compared and, if they match, the nodes know they have the same set of anti-tuples for that group. If they are different, then the nodes exchange a list of anti-tuples for each of those groups which did not have matching checksums. If the groups of anti-tuples are particularly large, it may be worth further subgrouping them and repeating the algorithm if the first checksum fails. Since checksums are calculated on the fly there is no storage overhead, although some storage space may be used to cache checksum values to allow for quicker synchronisation when a set of anti-tuples does not change. If the tuplespaces contain very similar sets of anti-tuples, then the network usage will also be reduced compared to the complete exchange mechanism. This improvement in network usage reduces as the number of differences within groups increases and can even result in an increase in network usage in extreme cases where the total size of the mismatched checksums sent is greater than the size of the anti-tuples which did not have to be sent. The computational cost of generating these checksums is dependent on the number of anti-tuples represented by each checksum, since each anti-tuple must be incorporated into the checksum in some way. The most common mechanism for generating checksums is to make use of hashing algorithms [Kno75]. Note that checksums are not infallible. It is possible, albeit unlikely, that two distinct sets of anti-tuples could generate the same

hash and, as a result, a false positive could be registered. It is important to choose a good hashing algorithm in order to reduce the chances of this occurring as much as possible.

If computational cost is not an issue at all, checksums can be improved further through the use of rolling checksums, similar to the mechanism employed in rsync [Tri99] to synchronise the contents of files on different machines. This mechanism is a uni-directional synchronisation for sending additional information from a *sender* to a *receiver*. First of all the receiver splits its copy of the data into discrete chunks of some size, say s. For each block the receiver calculates two checksums. The first is a strong checksum, whose purpose will become clear later on. The second is a weaker, rolling checksum. The rolling checksum has the property that, if the checksum for a number of sequential items, say the first 24 anti-tuples in a list, is known, then the checksum for the next overlapping set of sequential items, i.e. the 2nd to the 25th anti-tuples in the same list, can be easily calculated from the known checksum along with the values of the first item from the previous block and the last item from the new block (i.e. the 1st and 25th anti-tuples in the list). This allows the sender to easily find the matching block in its list by "sliding" the rolling checksum down the list until a match is found. Once a match is found, the sender calculates the stronger checksum for that block and compares it with the one it was sent. This step bolsters the weakness of the rolling checksum with the extra reliability of a stronger checksum without the need to calculate it for each overlapping block. Once the match is confirmed, the sender sends any data before the start of this block, but after the end of the previous matched block, to the receiver. This allows the receiver to adapt its copy of the data where appropriate to match the copy at the sender. This technique works best when some sort of ordering over the data set can be assured (but this is not required). This ordering should be such that, if data item A comes before data item B in one node, the same should be true in the other node, although there may be other items in between A and B which the first node does not possess. Rolling checksums are incredibly expensive in terms of computational demands due to the need to repeatedly calculate the checksums (even with the reduced cost of the rolling checksum). However, they can result in an improvement over the basic checksum system.

Tiamat implements a flexible synchronisation mechanism. The communications manager holds a set of `SynchronisationManager` instances. Each instance defines a mechanism for synchronising a class of anti-tuples. Which manager to use is defined on a class

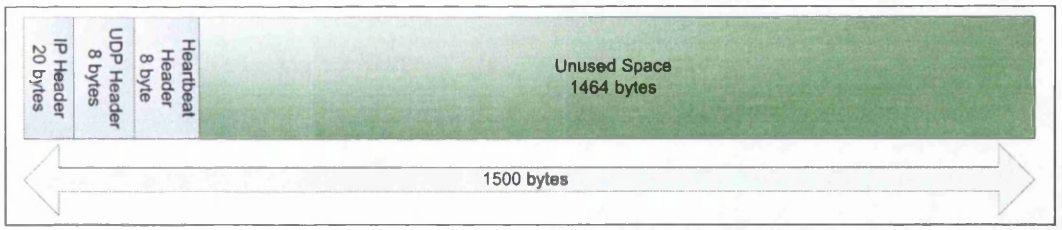


Figure 6.12: Space available in network packet.

by class basis and is changeable at runtime[11]. This allows the system to make the best use of the resources available. At present the decision is made by the application developer, but this could later be improved by the automation of this selection based on the presently available resources.

The current prototype provides only a `SynchronisationManager` for the send-all mechanism described above. This was chosen for two reasons: firstly, both timestamping and checksum based synchronisation rely upon it as a fall back mechanism; secondly, it is simpler to implement than the rolling checksums.

Payload Utilisation

As well as the two extensions to deal with anti-tuple propagation, the improved version of the communications manager contains one further extension. When a node emits a heartbeat, any new nodes will immediately contact it and attempt a synchronisation. This occurs even if the nodes have no tuples/anti-tuples in common. It would be helpful to either reduce the number of unnecessary exchanges or at least prioritise exchanges to make the best use of them. For this reason it was decided to make use of the extra payload space in the heartbeat packets.

When sending out the heartbeats, the amount of data sent in an individual heartbeat is relatively small when compared to the maximum packet size for a typical wireless network (as shown in figure 6.12). Research has shown [XP99] that when using UDP multicast the size of an individual packet has little impact on the probability of that packet being lost. This means that the extra space in heartbeat packets is essentially going to waste.

Instead of wasting this space a node can place a set of data items in here representing the types of anti-tuples it currently has stored that are awaiting a match. The identifier

[11] It is worth pointing out that changing the synchronisation mechanism at runtime for a particular class may not reap immediate benefits. For example, if the synchronisation is switched from checksums to timestamping, then the system must do one full send-all synchronisation in order for the timestamp information to be gathered. The benefits of the timestamping will only be seen on the second synchronisation.

for the type (comprising the fully qualified name of the class combined with the fully qualified name of the classloader which loaded it) is hashed into a 64-bit number using the MD5 hashing algorithm [Riv92] (to reduce the space consumption). If the number of distinct types is less than one hundred and eighty, then all the hashes can be placed into the packet. If not, then they are ordered in terms of number of anti-tuples of that type, highest first, and a special marker is placed at the end of the packet to signify that more types are available. Using this information, another node can make a decision on whether or not it can satisfy the anti-tuples at another node. In the case where there are more tuples than can be represented, at least the client can prioritise the contacting of nodes based on the number of outstanding tuples.

6.4.4 Distributed Consensus

During the implementation of the Tiamat system, a problem was encountered which could not be solved. The problem arose only in an unlikely set of circumstances, but could have potentially significant consequences. The problem is a variant of the well studied distributed consensus problem [Lyn96], which, unfortunately, has been shown to be unsolvable in the general case. In Java it is possible that, during an exchange of data over a network, one Tiamat instance could experience an `IOException` while the other thinks that the exchange has been satisfactorily completed. This gives two possible situations: the sender thinks that the datum has been sent but the receiver has not received it, called *receiver exception*; or the receiver has received the datum, but the sender thinks it has failed, called *sender exception*. For a given network protocol, only one of these situations should arise[12].

This problem is not limited to Java but can occur in any communications system in which failure of the communication channel is possible. The problem seems to go unmentioned in the majority of distributed systems work; indeed, none of the other mobile Linda systems discussed in chapter 4 make any mention of the problem. In normal circumstances, the problem is unlikely to occur often (it requires the loss of the last packets of the communication in only one direction), however, the constantly changing nature of a mobile environment is likely to increase the failure rate of traditional communication channels as devices move out of communication range or move into particularly noisy areas, and so cause a corresponding increase in the chances of this problem arising.

[12] A receiver exception arises in the case of a protocol which uses NACKs and a sender exception in the case of a protocol which uses ACKs.

It is also important to note that, while further communication between sender and receiver could possibly remedy the problem, it is impossible to guarantee that such communication will be able to take place (particularly in the face of the failure of a communication channel). As such, it is important to consider the potential impact of the problem and examine what possible actions can be taken in such an event.

There is the potential for this inconsistency to have an effect on an application's semantics. In the case of a receiver exception, the system will experience tuple loss since the sending instance will have no reason to hold onto the tuple. This could possibly be circumvented by holding on to the tuple for a period of time after it has been sent, but, as discussed, there is no guarantee that the receiver will be able to get in touch again. This would also represent a drain on resources proportional to the length of the time the tuple is held for, which would occur every time a tuple was sent, not just those in which a problem occurred (as the sender is unaware of any problem). Given the unlikely nature of the problem, along with the resource impact of this solution, it is highly uneconomical and for this reason has not been implemented in Tiamat.

In the case of a sender exception, the tuple will be duplicated. The copy that was received will be used normally, while the sender, having failed to send the tuple, will place the tuple back into the local space. Depending on the expected semantics, this could be disastrous (if, for example, the tuple represents exclusive access to a resource). This could be circumvented by always discarding the tuple when the sender sees an exception. Remembering that, in the vast majority of cases, when the sender sees an exception it means that the tuple has not been sent, this solution will result in tuple loss every time there is a communication error. This high loss rate is likely to be undesirable in most situations. If, however, duplication *must* be avoided then these losses may be acceptable. Such a decision can only be made at the application level.

In Tiamat, the default behaviour is to assume that a sender exception indicates that the tuple has not been sent and that the tuple should be placed back into the space. However, Tiamat also makes allowances for those applications which cannot handle tuple duplication. Tuples can be marked as *unique*, which indicates to the communications manager that they should always be discarded should a sender exception occur. Assuming the number of tuples labelled as unique is going to be relatively small, it may be worth implementing a quarantine system for these tuples in order to reduce the frequency with which they are lost. The sender, upon seeing the exception, would place the tuple into

quarantine. It would then spend a predetermined amount of time trying to contact the receiver and determine if the tuple was received correctly. If the receiver cannot be reached then the tuple must be discarded. Receivers must keep track of any unique tuples they have received. However, since the number of these tuples is likely to be low and the information only needs to be stored for the same amount of time as the sender will spend trying to re-establish communications, this will not be a substantial burden.

Since the Tiamat system cannot determine which tuples should be unique, this decision is left to the application developer. A special version of the **out** operation is provided which allows a tuple to be flagged as unique. This also serves a dual purpose in that it makes the application developer aware of the problem. It is important that the application developer understands that these situations may occur during operation of the system. This gives the developer the option of coding to deal with them. If the application developer is not informed, it could lead to later problems as applications start to exhibit undesirable behaviour. This is also likely to be difficult for the application developer to debug as he will not be aware such things are possible.

6.5 Linda Semantics

The Tiamat implementation of the Linda$_m$ model makes two major modifications to the basic Linda semantics in the form of loss and duplication. These are in addition to the extensions which resulted from the Linda$_m$ model described in section 5.5. Since the original Linda system was designed for single memory space systems, these problems did not arise[13], but in the context of a distributed system they are unavoidable.

Loss can arise either due to the distributed consensus problem or due to the departure/failure of other nodes in the system. As such it is important for application developers to consider the potential for loss and program accordingly.

Duplication can only arise in the case of distributed consensus problems. As such, it is unlikely to occur, but application developers must still be aware that it is possible. If duplication will have catastrophic effects on the operation of their system, then a workaround is available.

A summary of all the modifications or extensions made to the Linda model made during the course of this work will be presented in section 7.2.

[13]To be more accurate, neither could arise due to the Linda system itself. Obviously, if applications began to misbehave, then the resultant behaviour is undefined.

6.6 Summary

This chapter has described the Tiamat implementation of the Linda$_m$ model, highlighting each of its constituent components and discussing how they work together. The chapter has also identified and discussed the distributed consensus problem.

Chapter 7

Analysis

This chapter provides an analysis and evaluation of the Linda$_m$ model and the Tiamat implementation. Two sample applications, adapted from third party code, are presented in section 7.1, to demonstrate that the Linda$_m$ model and its implementation in Tiamat are operable. There then follows a discussion of the ways in which Linda$_m$ and Tiamat deviate from or expand upon the traditional Linda semantics and why each was felt to be necessary in section 7.2. Finally, Linda$_m$ and Tiamat are evaluated through a personal comparative analysis between them and existing research (from chapter 4) in section 7.3.

7.1 Applications

In order to examine the functionality of the Tiamat system and, at the same time, examine the consequences of programming with the Linda$_m$ model, two third-party applications were ported to use Tiamat as their coordination infrastructure. Both applications stem from the examples discussed in section 2.4.1. The first is the web client and proxy server, the second is the fractal generator.

7.1.1 Web Proxy Server/Client

As outlined in section 2.4.1, the traditional architecture used for web proxies is strictly client/server and is shown in figure 7.1. Web clients (e.g., a browser) connect to the proxy and make HTTP requests. The proxy retrieves the relevant item, be it a page, image or application, and returns it to the client. Although this architecture is sufficient in a static network setup, it has some disadvantages in a mobile environment.

Firstly, when a mobile client moves around the network, the proxy which it is using

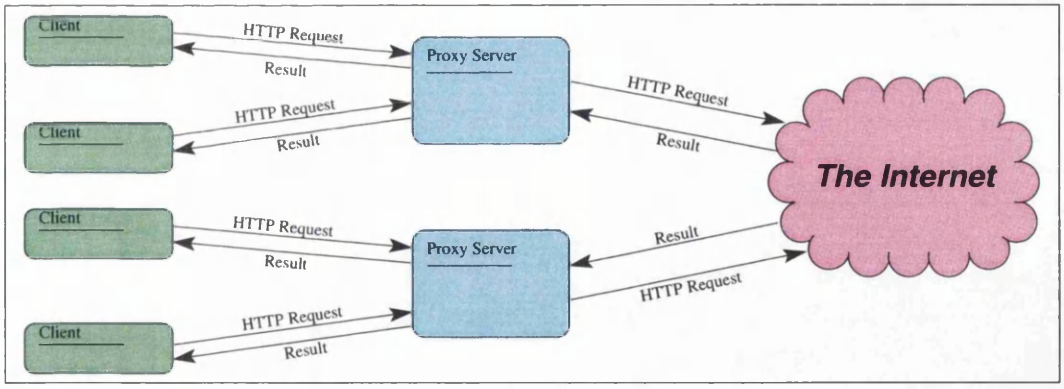


Figure 7.1: Original web proxy/client architecture.

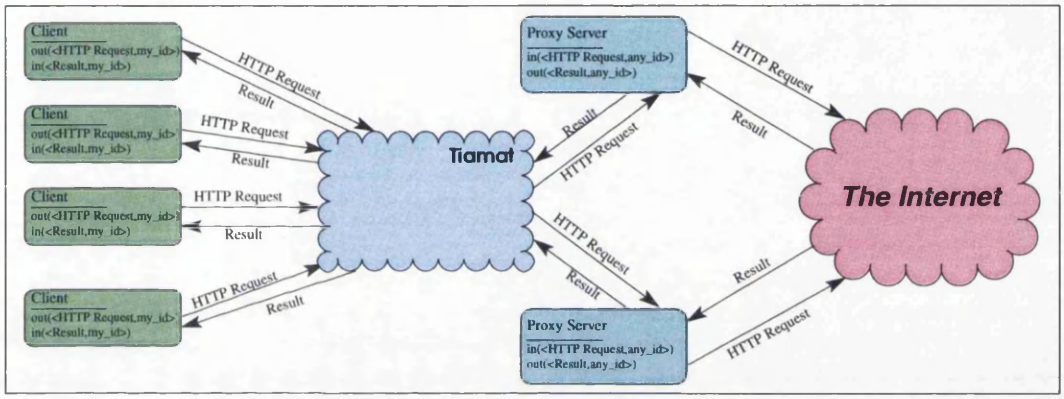


Figure 7.2: New web proxy/cient architecture.

may become inaccessible. This means that there must be an infrastructure which allows the client to find and connect to new proxy servers. Secondly, if a mobile client is only connected intermittently, then it may have trouble using the proxies. In a traditional setup the client must remain available in order to receive the web item. If it is disconnected from the server, then the server will discard the item. By adapting the proxies and clients to use Tiamat to coordinate, it should be possible to overcome these problems.

Figure 7.2 shows the modified architecture of the web client/proxy using Tiamat. The client, instead of connecting to the proxy, connects to a small adaptor program on the same device. This adaptor takes HTTP requests, wraps them up into tuples, attaches an ID and then places them in the tuple space. The client adaptor then performs an **in** operation for a tuple with the same ID field. There is another adaptor program for the proxy. It should ideally be run on the same physical machine as the proxy itself to improve performance and to simplify administration, but can also be run on another static node within the same network. This proxy adaptor performs **in** operations looking for HTTP request tuples. These tuples are removed and the HTTP request is unwrapped and then passed to the

proxy, which processes it normally. When the proxy returns a result, the proxy adaptor wraps the returned item along with the original request ID in a new tuple and places it back in the space. This result tuple is then retrieved by the original requesting client adapter and the result is given back to the web client.

This system circumvents the problems of proxy discovery and intermittent connectivity outlined above. Due to the decoupling in identity offered by the Linda$_m$ model, clients do not need to know which proxy they are using, only that there is a proxy available. As a result, a mobile client application will not have to modify its behaviour or configuration as it moves around. The decoupling in time and space offered by Linda$_m$ mean that, should a client with intermittent connection make a request of a proxy, the client may still be able to receive the tuple from the space the next time the proxy is visible (assuming the lease has not expired).

In addition, this improved architecture makes it easier to replace a web proxy for whatever reason (e.g., maintenance, fault rectification etc.) without having to inform all of the clients. It also allows for dynamic load-balancing by starting up more instances of the proxy and proxy adaptor as needed.

The code for the two adaptors consists of around 200 lines of code. The client used was the Mozilla web browser [Moz04]. The server was the Squid web proxy. The system required no modification to the code of either the client or proxy. The system required no understanding of the client and proxy beyond their paths of communication.

7.1.2 Fractal Generator

The distributed fractal generator is one of the canonical examples of the master/worker architecture described in section 2.2.5. While fractal generation specifically may not be a common requirement in a mobile environment, the more general pattern of master/worker is as it allows potentially resource impoverished devices to benefit from the collective resources of others. The fractal generator is presented here as an exemplar of this type of application and the benefits it can bring. The fractal calculations are generated by one or more masters node and then performed by some number of worker nodes. The architecture of the original fractal generator can be seen in figure 7.3. The master nodes connect to a load balancing server which then farms out the fractal calculations to a series of worker nodes. These worker nodes then return the result of the calculation directly back to the appropriate master.

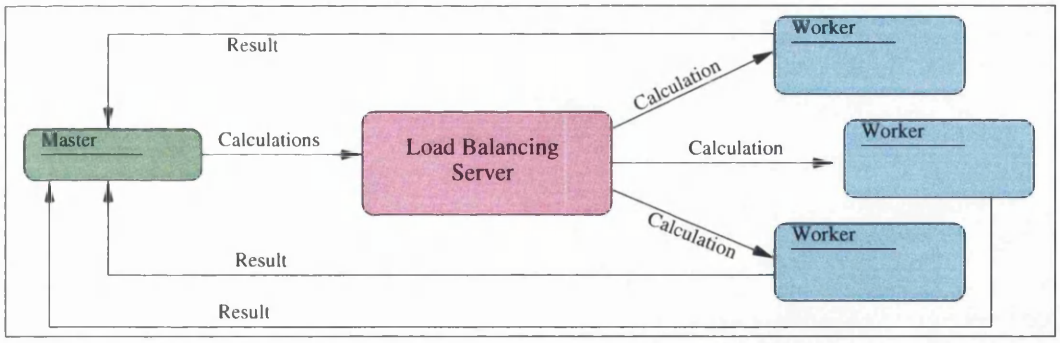


Figure 7.3: Original fractal generator architecture.

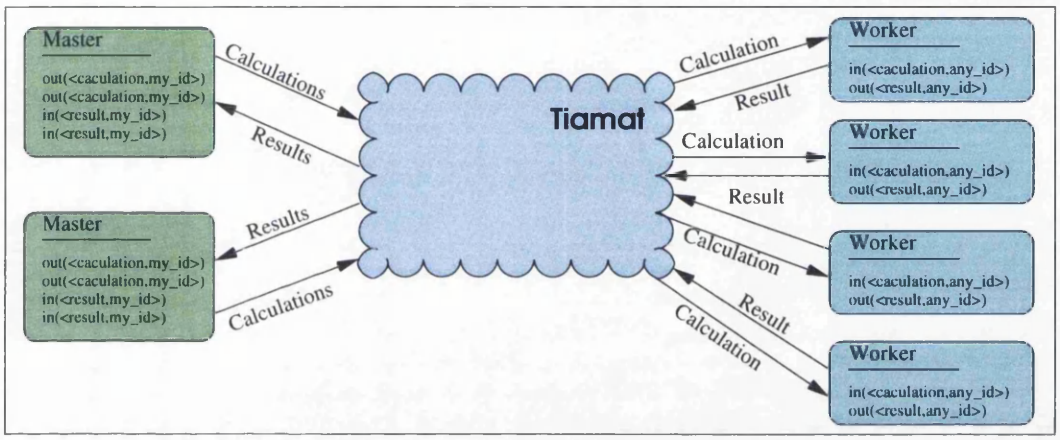


Figure 7.4: New fractal generator architecture.

Although the traditional architecture works fine in a static environment, it causes problems in a mobile environment. The two core issues are the same as for the web proxy server and client — namely the need for a discovery mechanism for the load balancing server and dealing with the possibility that the master may not always be connected to receive results. In addition, it is worth noting that, in this particular architecture, the load balancing server constitutes a single point of congestion and failure in the system.

The improved architecture using Tiamat is presented in figure 7.4. The load balancing server is removed entirely and the master and workers now coordinate entirely through the tuple space. Two small adaptor programs were written. One thread in the master adaptor takes the calculations from the master node and wraps them up in tuples, along with an ID code representing the master node (as there may be more than one in operation), which are then placed in the tuple space. Another thread performs **in** operations for any result tuples bearing the appropriate ID. The worker adaptors perform **in** operations looking for calculation tuples. These are then retrieved and the calculation is passed to the worker. The result from the calculation is then wrapped up in a result tuple and placed back in

the space.

As before, the various forms of decoupling offered by the Linda$_m$ model offered significant benefits to the application. Masters do not need to determine the identities of any worker nodes or load balancing servers in order to work, all they have to do is place the appropriate tuples into the space. The master may still be able to receive results which become available while it is absent by retrieving the relevant tuple from the space (assuming that the lease has not expired). The system can engage in dynamic load balancing by starting new workers as necessary.

7.1.3 Summary

The porting of two third-party applications to Tiamat demonstrates both the functionality and operability of the platform as well as its usefulness. Both ports were performed with the writing of small (<200) amounts of code, yet the inclusion of the tuple space for coordination provided clear advantages.

7.2 Extensions to Linda

The Linda$_m$ model and Tiamat are designed to provide the Linda semantics in a mobile environment. During their conception and implementation there were some extensions to the traditional Linda semantics. In each case this was done to either improve the functionality of the system or to make it fit the environment. This reviews the extensions presented in sections 5.5 and 6.5.

7.2.1 Leasing

The leasing mechanism represents the biggest modification to the traditional Linda semantics (traditionally tuples live forever and blocking operations block indefinitely until a match is found), but this particular extension is necessary to allow for resource management, one of the design principles from section 5.2.1. Without leases there would be no mechanism for garbage collecting tuples, which could lead to wasted resources. On devices which are likely to be resource impoverished this is highly undesirable. Providing the leasing mechanism offers a simple and well studied [BGZ00, BGZ01, BZ03] means of controlling resource consumption. Finally, due to their use in many other distributed systems, developers are likely to be familiar with their use and function.

7.2.2 Consensus Problem

Distributed consensus is not a modification to the semantics, as such, but rather the acknowledgement of a peculiarity of the environment. In an environment where the communication channels are subject to failure, it is impossible to guarantee that two distributed nodes will be able to reach a consensus [Lyn96]. In the case of Tiamat, the consensus involved is whether or not a tuple has been successfully sent from one node to another. This consensus problem, when it arises, has the potential to alter the semantics of the system by duplicating tuples. While there is no way of preventing the problem, it is still important to make the application developer aware that the problem can exist and, where appropriate, allow the application developer to select the policy that best suits the desired semantics for his application.

At the time of writing, Linda$_m$ and Tiamat are the only mobile tuple space systems which are explicit about the impact of and policies available for distributed consensus.

7.2.3 Direct Remote Communications

Direct remote communication allows the application developer to break through the abstraction provided by Tiamat and place tuples in, or direct operations to, a specific remote space. Although this extension is not necessary to allow Linda$_m$ or Tiamat to function, it allows for application-level optimisations. For example, imagine one of the worker nodes in the fractal generator system (section 7.1.2) that resides on a mobile node. While performing a calculation, the application becomes aware (through some other mechanism) that the device is about to be disconnected from the network. In this case, it would be better for the worker not to place the tuple back into its own space as no one else may be able to reach it for the foreseeable future. The worker would instead place the tuple into the master's space, or the space of another worker, to increase the chances of it being retrieved successfully. This is an example of how being able to perform direct remote communication can be used to perform application-level optimisations.

7.2.4 Summary

The extensions made to the basic semantics are necessary either in order to fit the environment (as in sections 7.2.1 and 7.2.2) or to empower the application developer to make application level optimisations (as in section 7.2.3). Aside from these modifications,

the Linda semantics are preserved and provided to the application developer through the abstraction of an opportunistic logical tuple space.

7.3 Comparative Analysis

This section will evaluate Linda$_m$ and Tiamat by examining the major differences between them and each of the tuple space systems outlined in chapter 4.

7.3.1 LIME

One of LIME's primary weaknesses was its attempts to enforce global consistency on a potentially large and rapidly changing network of devices[1]. As well as being ill-suited to the environment, this also required the provision of explicit connection/disconnection operations. It is unrealistic to expect devices in a mobile environment to announce their departure in such a way, as departure will often be unpredictable. The design principles distilled from the discussion of the environment in section 5.2.3 have resulted in Linda$_m$ and Tiamat providing an opportunistic mechanism for accessing remote spaces. This approach has avoided the difficulties in providing a globally consistent view and has provided a solution that more naturally fits the environment.

7.3.2 CoreLime

CoreLime tries to address some of the issues presented by federation in the LIME system, but goes too far. It strips the system of any kind of automated remote access, instead requiring the application developer to bear the burden of locating, contacting and sending agents to other remote tuple spaces. Linda$_m$ and Tiamat still allow the application developer access to potentially many remote and local spaces through the abstraction of a single logical space. The application developer is not required to concern himself with the details of which spaces are located where[2] and how to access them. This provides a simpler model for the application developer to deal with.

7.3.3 L^{2}imbo

The L^{2}imbo system made extensive use of replication to provide access to a single tuple space for multiple applications. In an environment where the participating devices are

[1]The weakness of global consistency has also been identified in [BZ01b].
[2]Although he may do if desired, see section 7.2.3.

likely to be resource impoverished, the substantial burden represented by having to maintain a replica of the entire space may be more than such devices are willing or able to bear. The design principles in section 5.2.1 have led Linda$_m$ and Tiamat to avoid the use of replication in favour of distributing the logical space over the set of opportunistically visible hosts. As a result, Linda$_m$ and Tiamat do not place such high demands on participating devices. Furthermore, through the provision of the leasing mechanism, each device has fine-grained control over how many resources are consumed by its local space.

One advantage of the replication mechanism used in L^{2}imbo is that it is not affected by the issue of Distributed Consensus in the same way at Tiamat. Because tuples are replicated and not moved from one node to another, there is no communication failure mechanism through which tuple loss can occur. Also, by requiring that a tuple can only be removed from the space by its owner, unique withdrawal is guaranteed.

7.3.4 PeerSpaces

Although the PeerSpaces system is not intended for use in mobile environments, it still provides an interesting basis for comparison. PeerSpaces uses a structured overlay network to perform routing for its queries. Although such overlay networks have already been shown to perform well in wired and relatively static networks [Ora01], their ability to operate in mobile and rapidly changing environments is not so well understood. In particular, it is conceivable that such networks will show deterioration in the face of increasing amounts of change as the work done to maintain the network begins to obscure the work performed through the network. Linda$_m$ and Tiamat at present opt for a more free-form approach, where no overlay network is created in order to perform better in the face of increasing frequency of change. Further research is needed to evaluate which of these approaches is most appropriate, or whether the actual solution lies in hybrid overlay networks which try to take the best of both solutions. This is discussed further in section 9.1.

As the PeerSpaces system is designed to operate on relatively resource rich machines, no mechanisms are provided to allow for resource management.

7.3.5 Summary

In each case, Linda$_m$ and Tiamat can be seen to provide some distinct advantage over the other systems. Importantly, each of these advantages can be traced back to one or more of the design principles given in section 5.2. This highlights the importance of

the environment-centric design, which has resulted in Linda$_m$ and Tiamat fitting more naturally with their environment.

7.4 Summary

This chapter has demonstrated the functionality of Tiamat through sample applications. It has also provided a clear discussion of why the extensions to the Linda system, which were provided in Linda$_m$ and Tiamat, were deemed necessary. Finally, it has provided a limited evaluation of Linda$_m$ and Tiamat in the form of a personal comparative analysis. The following chapter will follow on from this to provide quantitative evaluation of the Tiamat system through experimentation.

Chapter 8

Experiments

This chapter describes some experimental evaluations of the Tiamat system. A general characterisation of the system is established through a series of experiments using the web proxy/client application originally described in section 7.1.1. These experiments are presented in section 8.1. This is followed by some further evaluation of the heartbeat mechanism from section 6.4.3. Details of this evaluation can be found in section 8.2.

Details of all of the machines used for these experiments can be found in appendix A. Here they will be referred to by their designations in that appendix.

8.1 Tiamat Evaluation

This section describes three experiments performed to establish various characteristics of, or costs associated with, the Tiamat system. The first experiment, presented in section 8.1.1, was designed to measure the communications overhead and compare this with an existing communications platform. The second experiment, described in section 8.1.2, establishes the costs involved in the synchronisation mechanism used to propagate existing operations to new nodes (previously discussed in 6.4.3). Section 8.1.3 describes the final experiment in which the costs involved when multiple nodes are present are discussed.

8.1.1 Communications Overhead

Purpose

The purpose of this experiment is to establish the communication overhead involved when using Tiamat for generative communication between two processes. This will establish a base "cost", in terms of communication time, for using Tiamat. This baseline can

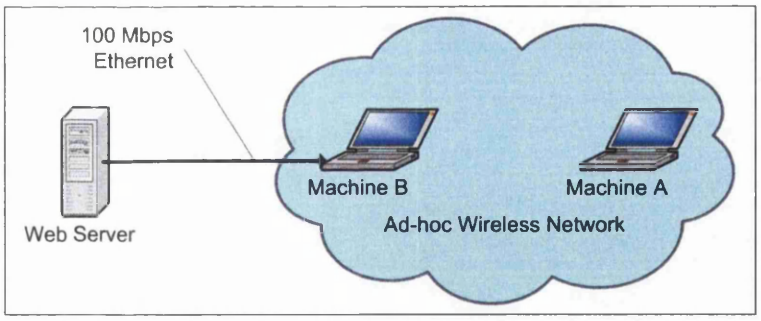


Figure 8.1: Experimental Setup for First Experiment.

then be compared to the cost of using another communication system, in this case Java RMI (Remote Method Invocation) [PM01], to establish the viability of Tiamat's run-time performance.

Method

The experimental setup is described in figure 8.1. Two laptops, A and B, are connected via an ad-hoc wireless network. Machine B is connected to a web server via a high bandwidth, low latency wired connection. This web server is set to serve a series of locally stored web pages of predetermined size. This high speed connection, along with the fact that pages are served locally, ensures that there is no significant delay in obtaining the web page, only in communicating it between the two laptops.

Two communications mechanisms are used, Tiamat and Java RMI. When Tiamat is used, it makes use of the web proxy/client application described in section 7.1.1. Machine B, which is connected to the web server, acts as the proxy, removing request tuples from the space, retrieving the appropriate page from the web server, wrapping the page in a tuple and placing that tuple into the space. Machine A runs the client, wrapping the address of the page required in a tuple, placing it in the space and then blocking while awaiting a result.

The RMI setup has machine B, which is connected to the web server, advertise a web proxy object via an RMI registry also running on machine B. This web proxy object has a single method which takes in a page address, the appropriate page is retrieved from the web server and then returned to the caller.

Seven sizes of page were tested ranging from 10 bytes to 10,000,000 bytes and increasing by a factor of ten at each step. Each size of page was retrieved 10 times without being timed to allow any appropriate caching mechanisms to be primed. A timer is then started

Page Size (bytes)	100	1,000	10,000	100,000	1,000,000	10,000,000
Tiamat	26.42	32.10	56.38	212.45	1399.36	12674.76
RMI	57.77	56.78	66.39	170.58	1263.38	11841.68

Table 8.1: Web page retrieval times in ms.

at the client and the page retrieved a further 100 times before the timer is stopped. This is important for the smaller pages where the time to retrieve a single page is relatively small and could be heavily perturbed by the inclusion of measurement code. Aggregating over a series of retrievals reduces this effect. This process is then repeated a further nine times and an average is taken from the time to retrieve the page one thousand times.

In the case of Java RMI, the advertised object is retrieved only once at the start of the run, not for each retrieval. As such, the times noted do not include the cost of the object lookup and retrieval, only of the method invocation itself.

Results & Conclusions

The results from these experiments are presented in table 8.1. This graph shows the average time for a single retrieval of a page of a given size for each of the two systems. Here it can be seen that the time taken for Tiamat to retrieve each page is comparable to that of RMI. This is especially important with the smaller pages where the communications overhead dominates the exchange. Tiamat's marginal slowdown on the larger pages is most likely the result of the two serialisation/deserialisation steps used in Tiamat (first when the tuple is entered into the space and again when it is communicated across the network).

Tiamat has been shown to have similar communications overhead and performance to an established communications mechanism. This establishes Tiamat's viability for use to provide communications between two nodes in an ad-hoc network.

8.1.2 Synchronisation Costs

Purpose

The purpose of this experiment is to establish the costs of the synchronisation mechanism described in section 6.4.3.

Method

The experimental setup is shown in figure 8.2. Machine B is connected to a web server

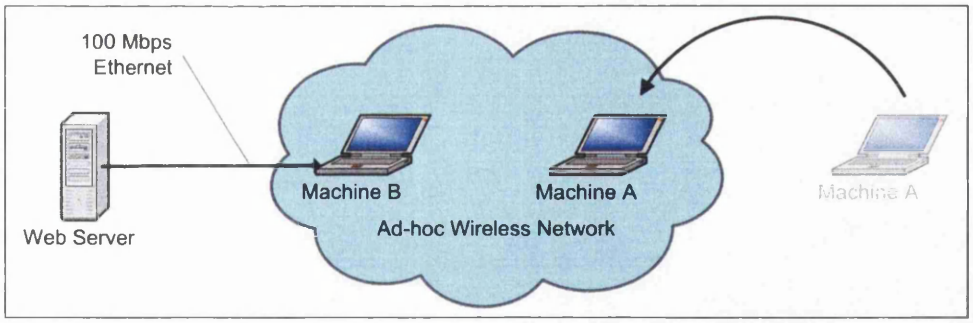


Figure 8.2: Experimental Setup for Second Experiment.

via a high bandwidth, low latency link. B is running the Tiamat web proxy described in section 7.1.1. This results in a single outstanding **in** at the proxy node. In addition to this, one hundred[1] outstanding **in** operations for other types of tuple are performed at both the client and the proxy. A second node, running the web client application, is then brought within communications distance of the proxy node. Once an ad-hoc network is established between the two nodes (monitored by determining when a packet from one can reach the other) a timer is started which stops once the appropriate page is returned to the client. Since the outstanding operations are for a different type of tuple, the outstanding operations will not return but must still be exchanged as part of the synchronisation. The sizes of the anti-tuples used in these outstanding operations was varied during the experiments to test the synchronisation cost. The values used for the sizes were 200, 1000, 2000, 4000, 6000, 8000 and 10000 bytes. Each experiment was repeated twenty times[2] and an average was taken.

Results & Conclusions

The results of this experiment are presented in figure 8.3. This shows that the overall synchronisation cost is relatively low and scales linearly with the size of the outstanding operations, which is to be expected with the synchronisation mechanism employed.

[1]It is important to bear in mind that each outstanding blocking operation represents a corresponding blocked thread. As such, it is not expected that each Tiamat node will experience a substantially large number of outstanding operations at any given time. It was felt that 100 represented a suitable high end limit for this testing.

[2]This experiment was run fewer times than the others as it required physical interaction from a human being.

Anti-Tuple Size	Retrieval Time
200 bytes	320.2 ms
1000 bytes	548.2 ms
2000 bytes	677.2 ms
4000 bytes	1193.0 ms
6000 bytes	1605.4 ms
8000 bytes	1971.2 ms
10000 bytes	2822.2 ms

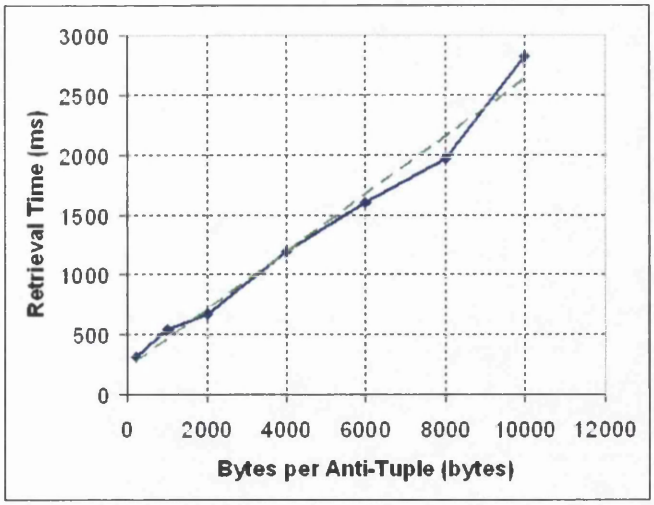


Figure 8.3: Synchronisation cost experimental results.

8.1.3 Multiple Nodes

Purpose

The purpose of this experiment is to establish the costs associated with having multiple nodes participate in the system.

Method

The experimental setup is similar to that of the experiment described in section 8.1.1. However, in this experiment, as well as the web client and proxy, there are a number of other nodes running Tiamat instances present in the ad-hoc network. These nodes are not performing any operations, but must still be contacted when new operations are performed. For this experiment, the operation of the system was altered slightly, instead of traversing the known node list in order when propagating operations, the nodes were set to traverse the list in a random order, to simulate the nodes arriving in a random order. A fixed size of page (100 bytes[3]) is retrieved ten times without being timed in order to prime caching mechanisms and then a further one hundred times while the response time was measured. This measurement is then repeated ten times and an average is calculated.

The tests are run with 2, 3, 6, 9, and 18 nodes. In the case where 18 nodes were used, each machine was running two Tiamat instances. Machines A through to I (A-I) are used, with A-C being used for the three node setup, A-F being used for the six node setup

[3]The smallest size of page was chosen to ensure that the cost of contacting new nodes would not be dwarfed by the cost of retrieving and transmitting the web page itself.

Nodes	Retrieval Time
2	26.42 ms
3	30.21 ms
6	51.43 ms
9	73.41 ms
18	167.23 ms

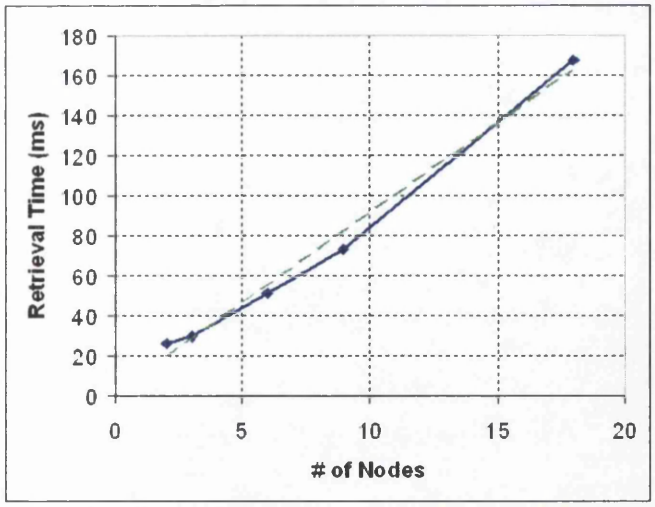


Figure 8.4: Multiple node cost experimental results.

and all nine being used for the remaining two setups, with two instances running on each machine in the final setup.

Results & Conclusions

The results for this experiment are presented in figure 8.4. It can be seen that the overhead for contacting each node averages about 7ms in total, this time is accounted for purely in the cost of contacting that node and sending it the anti-tuple.

It is worth noting here that the web client/proxy system performs especially well here because the anti-tuples in use are very small. If an application made use of larger anti-tuples then it would pay a greater overhead for sending them to each node resulting in an increase in communications time. However, generality is not lost since, for the majority of applications, anti-tuples are less specific than the tuples they are matching against. This means they will contain less information and typically be smaller as a result.

8.2 Heartbeat Evaluation

The two experiments described below, in sections 8.2.1 and 8.2.2, are designed to examine the advantages and disadvantages, respectively, of the heartbeat mechanism described in section 6.4.3.

8.2.1 Communications Savings

Purpose

To examine the savings, in terms of the amount of data sent, resulting from using the heartbeat mechanism described in section 6.4.3 when compared to simply having each machine emit a periodic heartbeat.

Method

For this experiment, nine machines are joined in an ad-hoc wireless network. A number of Tiamat instances are then started. The Tiamat instances do not have any operations running, as all the experiment is interested in is the heartbeat mechanism which operates even when no operations are being performed. Machines A-I are used for all experiments even where the number of instances being started is less than nine, the remaining machines remaining idle on the network[4].

Two heartbeat mechanisms were used. In the first, each node emits a heartbeat every 500ms. The second mechanism is the one described in section 6.4.3. For these experiments, the value of x was set at 500ms and the period for idle beats was set at 1000ms[5].

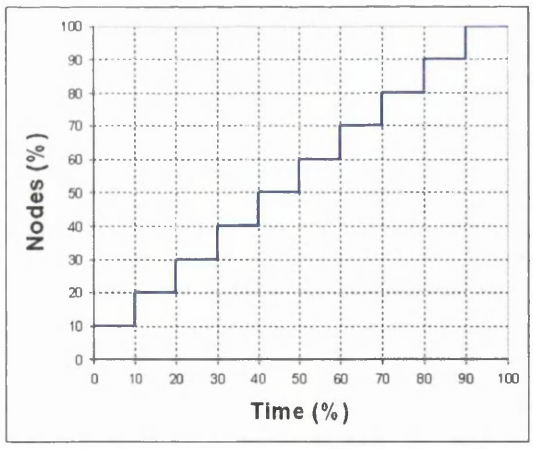
The experiments were run with 3, 6, 9, 18 and 36 Tiamat instances. Only machines A-C ran instances for the three instance setup, A-F were used for the six instance setup and all nine machines, A-I, were used for the remaining three setups.

The arrival and departure of the nodes was defined by four scenarios: climbing; declining; hill; and random. In the climbing scenario, 10% of the total number of nodes start every 10% of the total time, so the arrival pattern is as shown in figure 8.5(a). Declining is similar, but opposite, with all nodes present at the start and 10% leaving every 10% of the time resulting in the pattern from figure 8.5(b). Hill is the climbing scenario immediately followed by the declining one and can be seen in figure 8.5(c). In the random case, the nodes arrive and depart at random.

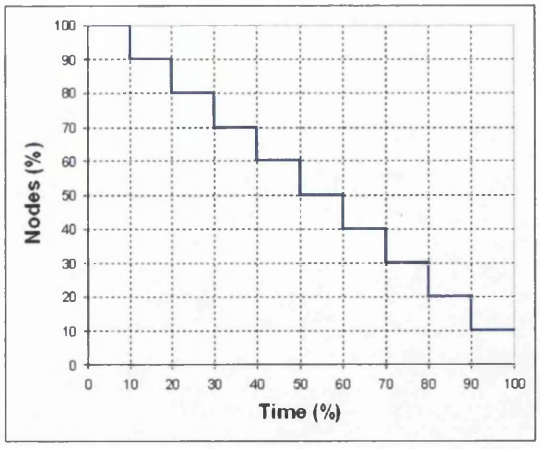
For each setup the experiment was left to run for 5 minutes and the number of heartbeats sent by each node in the system was monitored. An average was then taken over the number of nodes present in the system. In the case of the random arrival/departure scenario, the experiments were run twenty times and the results averaged.

[4]This setup made it easier to run multiple sets of experiments consecutively, without the need for human intervention

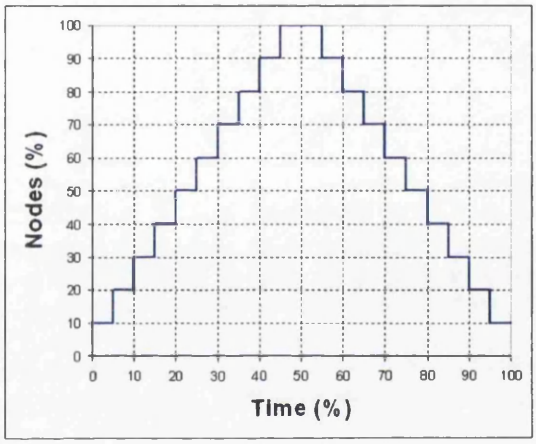
[5]This is probably slightly lower than the desired value for idle heartbeats, but was chosen in order to ensure the experiments made reasonably rapid progress.



(a) Climbing arrival pattern.



(b) Declining departure pattern.



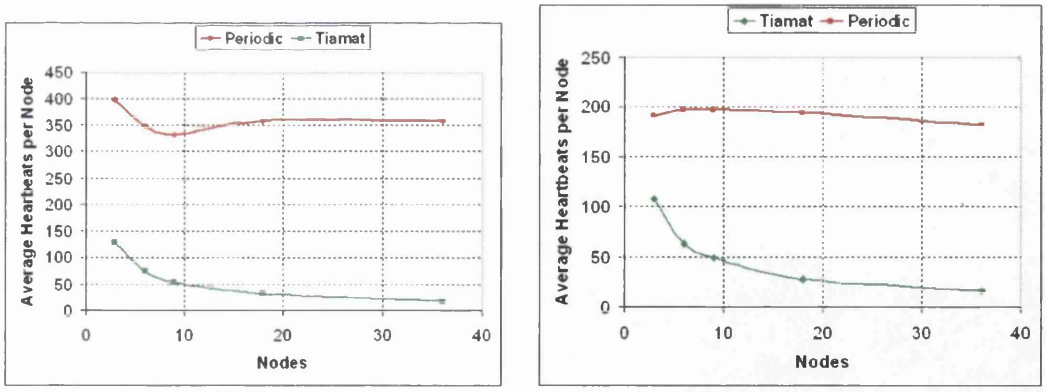
(c) Hill arrival/departure pattern.

Figure 8.5: Node arrival/departure patterns.

Results & Conclusions

The results for these experiments can be seen in figure 8.6. Figure 8.6(a) shows the average number of heartbeats emitted by each node during the climbing arrival scenario. The results for the hill and declining scenarios produced virtually identical results to that of the climbing scenario[6] (to within 0.2% of each other). Figure 8.6(b) shows the results for the random arrival/departure pattern. The use of the heartbeat mechanism provided as part of Tiamat shows a significant drop in the amount of traffic produced by the nodes compared to the cruder approach of having every node emit a periodic heartbeat. However,

[6]So much so that when plotted on the same graph it was impossible to distinguish between them — hence the reason only one is plotted here



(a) Climbing, declining and hill patterns.

(b) Random pattern.

Figure 8.6: Average heartbeats per node for various arrival/departure patterns.

as can be seen from the next experiment, this approach has a tradeoff.

8.2.2 System Awareness

Purpose

To examine how the overall "awareness" of the nodes in the system is affected by the heartbeat mechanism described in section 6.4.3 as opposed to simply having each machine emit a periodic heartbeat. The awareness of an individual node is defined as the ratio of nodes which it knows about to nodes which it could know about (i.e., nodes with which it is able to communicate).

Method

The setup is identical to that presented in the previous experiment (section 8.2.1). Also, what is being monitored is different. Rather than monitoring the traffic in this instance it is the list of known nodes at each node which is being monitored. Since the arrival/departure patterns are known, it is easy to determine how many nodes should be visible at a given point in time. From this it is possible to determine the awareness of an individual node by dividing the length of the known node list by the number of nodes which should be visible. This gives an awareness for each individual node.

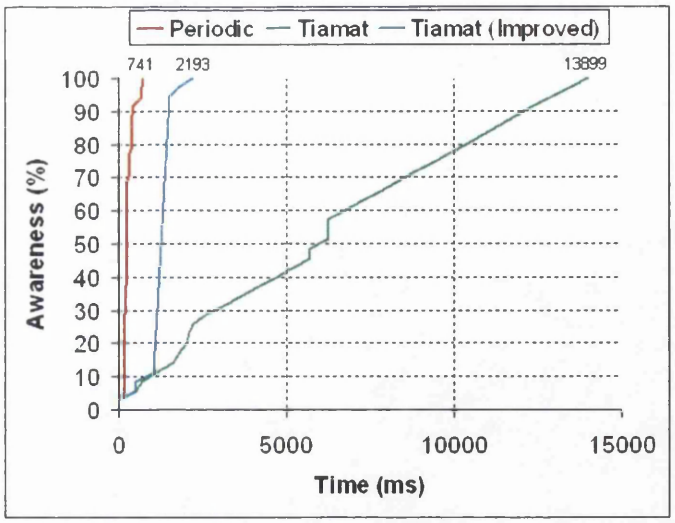


Figure 8.7: System awareness over time.

Results & Conclusions

The results from one of these experiment are shown in figure 8.7. This graph shows, for the last node to enter the system in the 36 node, climbing scenario, how its awareness varies over time. This is shown for both the periodic heartbeat (shown in red) and the heartbeat mechanism used in Tiamat (depicted in green). The emission of periodic heartbeats by all nodes results in a system in which awareness is kept high as a newly arrived node quickly hears from all other nodes present. The heartbeat mechanism proposed in section 6.4.3 does not have this advantage. As nodes which are in passive mode only emit heartbeats in turn, a newly arrived node must wait for one complete cycle before it is aware of all the nodes already present. As a result, the awareness of newly arrived nodes takes longer to rise.

To combat this, a suggested improvement is proposed which would allow nodes to more quickly discover an existing set of nodes. Whenever a node in the idle or active state hears a passive beat, it contacts the node which sent the beat. It then retrieves the known node list from that node. Since the passive node is already part of the passive cycle it should already be aware of all of the nodes in the group (either from having watched them arrive, or from the use of the same mechanism when it arrived). This should ensure a more rapid rise in awareness levels. The reason for only doing this while in the idle or active states is to avoid unnecessary traffic between nodes which have already settled into a passive cycle.

This improvement was implemented and also tested in this configuration. The results for this can be seen in blue alongside the others in figure 8.7. It can be seen that the

improvement drastically reduces the time taken for the node to reach full awareness.

However, there are two further issues with this improvement. Firstly when two groups of nodes, each of which have formed a passive cycle in isolation of the other, come within contact range of one another, then the system falls back to having to wait for a complete passive cycle in order for new nodes to be discovered. Secondly, if a large number of nodes come into contact with a passive group at the same time, then the next member of the passive group to emit a heartbeat could be swamped by responses. Neither issue is particularly critical in the case of Tiamat. In the former case, the system does not stop working, it merely works as it did before. The latter case is only relevant when large numbers of nodes are present. Since Tiamat is designed for small, ad-hoc networks, this is unlikely to be a major problem. For this reason, the mechanism has been implemented in Tiamat as described immediately above, to help improve the system awareness.

Although not critical for Tiamat, solutions to these issues still merit further discussion. The former may be resolved by having any node in a passive state, which hears a passive beat from a node it is not already aware of, contact the node which sent the beat and retrieving its known node list. The latter may be resolved by having nodes delay for a small, random amount of time before contacting the node in question to retrieve the known node list. In both cases, further investigation of these proposed solutions and any alternatives is required.

8.3 Summary and Conclusions

Tiamat has been demonstrated to have basic communication overheads similar to that of an established communications mechanism demonstrating the runtime viability of its communications system. This chapter has also server to characterise the behaviour and performance of Tiamat in the face of increasing numbers of nodes and outstanding operations.

This chapter has also examined and quantified the tradeoff inherent in the heartbeat mechanism employed within Tiamat. An improvement to the existing heartbeat mechanism has also been proposed and evaluated.

Chapter 9

Future Work

This chapter examines some avenues of future research which have been opened up by, or could be used to benefit, the work contained within this dissertation.

9.1 Adaptive Overlay Networks

As mentioned in section 7.3.4, overlay networks have proved successful in a variety of Peer-to-Peer systems. It is likely that their routing properties could also help improve the propagation of operations (and hence extend the scope of visibility) in Tiamat. However, at present, little research has been carried out into how such overlay networks behave or perform within the context of a mobile environment, and, in particular, how well they scale as the degree and frequency of mobility increases. Such performance and scalability information could be used to identify those structures of overlay networks that are well suited to use in a mobile environment or perhaps even propose new overlay networks for use in such environments.

Following on from this, it would be interesting to see how these mobile overlay networks relate and interact with overlay networks which are well suited to static environments. The resulting taxonomy would provide the groundwork for an examination of hybrid overlay networks.

Even in a highly mobile environment, there are still portions of the network that do not exhibit a high degree of change (e.g., the computing infrastructure in a computing science department). Since not all devices in the environment behave the same way, it could be useful to treat them in different ways. Providing a highly structured overlay network which is periodically costly to either construct or maintain but provides excellent routing

properties would be useful for the static portions of the environment, but unsuitable for the mobile portion. Providing a highly adaptable, low-cost overlay network will benefit the mobile devices, but will leave much of the potential of the fixed portions of the network untapped. It would be better to provide some hybrid of these two overlay networks which could allow static nodes to form highly structured overlays while providing facilities for mobile nodes to connect to that network in a lightweight and dynamic manner. The need for such research has also been identified in [CDHR02].

9.2 Localised Temporal Topologies

Even if such hybrid overlay networks are in place, there still remains another issue: how to identify which nodes tend to be static and which are more mobile. Although such identification may be performed statically, it would be of more benefit to have the system itself make the decision. This could allow the system to better adapt to changes in the behaviour of devices in the environment (e.g., if a node previously identified as static becomes mobile, or vice versa).

The nodes in the system would not be defined by a global or static topology as with traditional networks. Instead, nodes would now have to examine the nature of the environment around them and how it changes over time — which shall be referred to as a Localised Temporal Topology (LTT). It would be interesting to see whether nodes can determine, on-the-fly, useful and usable patterns in their own LTT. For example, a node could attempt to identify other devices with which it is frequently in contact. These associated devices could then act as proxies for routing information to this device (see next section) and vice versa.

Although LTT can give a device some information about the behaviour of the environment around it, it alone cannot provide information about the device itself. For example, if the device observes a large amount of change in the devices it can see, does that mean that the device itself should be classed as mobile or the devices around it. In order to address this issue, devices would have to collaborate to try and resolve such relativity problems.

9.3 Social Routing

As stated previously, the mobility behaviour of devices is likely to be driven by the movement of the users who carry them. This raises interesting potential for routing of information based on the social interactions of users. This social information could either stem from the examination of Localised Temporal Topologies or from higher-level applications. For example, if a calendar program on Bob's PDA shows that he will be in a meeting with Jane later that day, Sue's machine, which has non-urgent data to be sent to Jane, could use Bob's PDA as a communication medium. For Linda$_m$ and Tiamat, the social routing facilities would increase the scope of the visibility by allowing the communication of tuples and anti-tuples through social interactions.

Exploiting such behaviour will depend on how predictable these behaviours are. Again, the examination and classification of these sorts of behaviours is largely outwith the remit of Computing Science.

Another issue in this research would be the personal security and privacy implications of using such data. If, in the above example, Jane and Bob are having a secret meeting to have Sue ejected from the company, they may not wish to advertise the fact. Exposure of LTT information could similarly reveal much about a device's (and the associated person's) recent behaviour. Identifying information which could be sensitive is a non-trivial problem, since the distinction is highly dependent on human perception rather than directly discernible information.

9.4 Secure Tiamat

The Tiamat system, at present, makes no provision for security of any kind, depending instead on other software layers to provide such mechanisms. One interesting piece of research would be in how security could be provided through the Tiamat system itself in order to prevent abuse of the facilities offered.

The obvious point for such mechanisms to reside is within the lease manager (section 6.2). Since every operation in Tiamat must be leased, the lease manager is able to monitor the overall behaviour of the Tiamat node. It would be useful, therefore, if the lease manager could be adapted to identify and, possibly, prevent any abuses. Simple forms of security mechanism could involve black/white lists to control access. Nodes may be black listed if they exhibit undesirable behaviour, for example, starting **eval** operations

which (apparently) never complete. Identifying undesirable behaviour could in itself prove difficult. For example, how does the system distinguish between a node which is placing a large number of tuples in the space as part of its normal operation and one which is maliciously trying to consume resources?

Such security mechanisms would be strengthened by allowing lease managers to collaborate. This would reduce the impact of malicious nodes moving their attention from one node to another by forewarning other lease managers of the node's behaviour. However, this would also require mechanisms to allow the lease managers to form an appropriate network of trust. It would also be useful for the lease manager to collaborate with any existing security infrastructure so that they could both benefit from the information possessed by the other.

Another issue with securing a tuple space is that, in order to allow matching, the contents of tuples must be unencrypted. While it would be possible to encrypt the fields individually, this would still only allow an exact value matching and not any of the extended matching facilities.

Some of the issues involved in introducing security to tuple space systems, as well as recent proposals on how security could be provided, are presented in [BGLZ03, HR03].

9.5 Transactions

Although tuple spaces were originally proposed as a lightweight coordination mechanism, many implementations have provided transactional facilities in order to allow more sophisticated application behaviours. Most of these systems are either designed for use in distributed (but not mobile) environments or make use of the transactional facilities designed for such environments.

However, it has been shown that such transactional facilities are not well-suited to mobile environments as they result in high rates of transactional failure and unpredictable execution costs [Ser04]. It would be of interest to examine the use of a transactional facility designed for use in mobile environments to extend the Tiamat system [MB98, LLK01, Ser04].

9.6 Performance Improvements

In order to demonstrate the thesis statement (section 1.1), only a proof of concept implementation was required. Therefore, one piece of useful work would be to engineer an implementation oriented around performance in order to make the use of Linda_m more desirable to application developers. One example of a possible potential improvement stems from the storage of tuples. Indexing the tuples stored in a tuple space could reduce the cost of searching the space during matching. In providing any performance improvements, it would be important for the system engineer to consider the potential resource cost of providing the performance improvement and ensure that the improvement will be used often enough to warrant that cost.

9.7 Suitability of Ant Algorithms

Ant algorithms are one of the most prominent examples of a group of technologies, known as emergent technologies, where complicated global behaviours "emerge" from the interactions of simplified localised processes. The nature of these systems can make it very difficult to understand the overall behaviour of the system at runtime from an examination of the system's design and algorithms. These systems are also usually driven by random choices dictated by probabilities chosen at the design of the system. The values chosen for these probabilities can drastically alter the overall behaviour of the system and it is not always obvious what that impact will be or how to select appropriate values to achieve the desired behaviour.

One of the advantages of these algorithms is that the simple behaviours are very easy to program, reducing development costs. However, there are at present no clearly established metrics for evaluating the cost to benefit ratios for these algorithms from a resource usage perspective. There is also insufficient information on how several such algorithms interact.

The SWARM Linda system [MT03] referred to in section 3.3.9 employs two ant based algorithms to try and provide Linda across multiple hosts. Firstly, the system employs brood sorting to cluster similar tuples together. A collection of ants wander randomly between the spaces. Each ant has a template representing a type of tuple. If the ant encounters a space containing a tuple which matches that template, and it *is not* already carrying a tuple, it will pick up the tuple and move to a randomly chosen new space. If the ant encounters a space containing a tuple which matches that template, and it *is* already

carrying a tuple, it will drop the tuple and move to a new space. If the ant encounters a space containing no matching tuples and it *is* carrying a tuple, there is an increasing probability that it will drop the tuple before moving to a new space. The overall effect of this algorithm is that *some* similar tuples *may* be clustered together in a single space. The number of clusters which will be formed for a given type, the size of the clusters which will be formed and the time it takes for a cluster to be formed are all determined by the choice of the various probabilities which drive the system.

Secondly, the algorithm employs an ant based search for tuples. Ants wander the set of spaces randomly looking for a tuple which matches a template they carry. The ant maintains a short memory of where it has been to help it find its way back to the anthill (the space where the original request was made). To assist in this process anthills emit a pheromone which is diffused into the other nodes around it, weakening with distance from the anthill, which is also used to help the ant find its way back home.

Evaluating the impact or effectiveness of this approach is difficult given current techniques. For example, it is unclear how the two ant algorithms will interact. Since the search ants wander randomly, clustering the tuples into a small number of spaces may reduce the probability of an ant finding a match to its template. Also, the brood sorting algorithm imposes a constant cost on the system by placing it in a constant state of flux — even if no tuple space operations are taking place. However, the algorithm's impact is highly dependent on the input probabilities and as such is hard to measure.

Until some of these issues are addressed and more reasonable estimates of emergent technologies' costs and benefits can be made, it is difficult to accurately evaluate their effectiveness.

More work is also needed on techniques for crafting the probabilities which drive the system so that the desired behaviours emerge. However, it could be possible that systems which employs multiple ant algorithms could be driven by so many different variables that predicting any specific behaviours could become exceptionally difficult.

9.8 Summary

This sections has presented a variety of research avenues which could be used to improve or extend this research.

Chapter 10

Summary and Conclusions

10.1 Thesis Statement and Dissertation Overview

In the introduction (section 1.1) my thesis statement was given as:

> *Generative communications were originally designed for the coordination of parallel processes. However, they have also found a home in a variety of distributed environments including environments involving mobility. Much of the research carried out in these environments has been problematic and has led some to conclude that generative communications are unsuitable for such mobility-oriented situations. I believe, however, that this is incorrect and is more a reflection on the systems used in this research than of the suitability of the approach. I will demonstrate how previous research platforms have been unsuitable for mobile environments. I will furthermore propose a model and construct a proof of concept implementation to demonstrate that, with some minor semantic alterations, the generative communications paradigm can be provided in a mobile environment. I will measure and examine the characteristics of the operation of such a system and will compare the system to existing research to demonstrate that an environment-centric design results in a system which is better suited to the defined mobile environment.*

Chapter 2 highlighted the context in which the work was set, establishing the growing trend in mobile devices and highlighting how Linda could be used by application developers working in such environments. Chapter 3 provided an overview of related work. Chapter 4 supported the thesis statement by examining the previous mobile Linda solutions. In

each case, it could be seen that some aspect of the model or implementation did not fit well with the proposed environment. Chapter 5 presented a new model for providing Linda semantics in a mobile environment. This was followed by a description of a proof of concept implementation in chapter 6. The functionality of the model and implementation were tested successfully by porting third-party applications to the infrastructure with the outcomes and benefits being presented in chapter 7, demonstrating that generative communications can be made to function in a mobile environment. This chapter also compared the model and resultant implementations to the previous attempts presented in chapter 4 to highlight how, by keeping the environment at the heart of the design, the model and implementation were better suited to mobile environments. Chapter 8 provided demonstration of the viability of Tiamat as a communications platform as well as examining the characteristics of the implementation. Potential avenues for further research or development were presented in chapter 9.

10.2 Contributions and Achievements

The main contributions and significant achievements made during the course of this research are:

- The proposal of a novel model, Linda$_m$, for providing Linda-like semantics in a mobile environment.
- A proof of concept implementation, Tiamat, of that model to demonstrate viability and operability.
- Demonstration of the viability of Tiamat as a communications platform.
- Experimental evaluation of the characteristics of the Tiamat system.
- A demonstration of the value of a tuple space system in a mobile environment.
- Highlighting of an often overlooked problem that is exacerbated in a mobile environment (distributed consensus).
- A comparison of the new model and implementation with previous work, highlighting previous systems' unsuitability for mobile environments.

10.3 Conclusions

In this dissertation I have presented a novel piece of research in the field of mobile Linda systems and I have demonstrated the usefulness of this system and motivated the need for the research. I then examined and evaluated previous work in the field showing that there was still room for improvement. I examined the nature of the environment and used this to address shortcomings in the available literature. This was then followed by a concrete implementation of that model. I evaluated the functionality and usefulness of the implementation through the porting of third-party code to use the system. I then evaluated my work through comparison with the previous systems. I then performed numerous experiments to establish the viability of Tiamat as a communications platform as well as characterising the behaviour of the system. Finally, I looked to the future and examined other possible avenues of research.

Appendix A

Machines

This appendix describes the machines which were used for the experiments introduced in chapter 8.

A.1 Machine A

Manufacturer:	Fujitsu Siemens
Model:	Lifebook C1110
Profile:	Laptop
Processor Model:	Intel Pentium M 735
Processor Speed:	1.7 GHz
Bus Speed:	400 MHz
Memory:	1024 Mb
Wireless Adapter:	Intel PRO/Wireless 2200BG
Wireless Modes:	802.11b, 802.11g
Operating System:	Linspire Live 5.0
Java Version:	HotSpot Client VM 1.5.0_02-b09

A.2 Machine B

Manufacturer:	Fujitsu Siemens
Model:	Lifebook S7010
Profile:	Laptop
Processor Model:	Intel Pentium M 725
Processor Speed:	1.6 GHz
Bus Speed:	400 MHz
Memory:	512 Mb
Wireless Adapter:	Intel PRO/Wireless 2200BG
Wireless Modes:	802.11b, 802.11g
Operating System:	Linspire Live 5.0
Java Version:	HotSpot Client VM 1.5.0_02-b09

A.3 Machine C

Manufacturer:	Hi-Grade
Model:	Notino W6700
Profile:	Laptop
Processor Model:	Mobile Pentium 4-M
Processor Speed:	2 GHz
Bus Speed:	400 MHz
Memory:	512 Mb
Wireless Adapter:	Prism II
Wireless Modes:	802.11b
Operating System:	Windows XP Professional SP2
Java Version:	HotSpot Client VM 1.5.0_02-b09

A.4 Machine D

Manufacturer:	Olivetti
Model:	Xtrema 323S
Profile:	Laptop
Processor Model:	Intel Pentium II
Processor Speed:	233 MHz
Bus Speed:	66 MHz
Memory:	96 Mb
Wireless Adapter:	Compaq WL110
Wireless Modes:	802.11b
Operating System:	Windows 2000 SP4
Java Version:	HotSpot Client VM 1.5.0_02-b09

A.5 Machine E

Manufacturer:	Siemens Nixdorf
Model:	Scenic Mobile 710
Profile:	Laptop
Processor Model:	Intel Pentium II
Processor Speed:	233 MHz
Bus Speed:	66 MHz
Memory:	96 Mb
Wireless Adapter:	Netgear WG511
Wireless Modes:	802.11b, 802.11g
Operating System:	Windows 2000 SP4
Java Version:	HotSpot Client VM 1.5.0_02-b09

A.6 Machine F

Manufacturer:	Dell
Model:	Optiplex GXa
Profile:	Desktop
Processor Model:	Intel Pentium II
Processor Speed:	333 MHz
Bus Speed:	100 MHz
Memory:	128 Mb
Wireless Adapter:	Buffalo AirStation WLI-CB-G54 & Buffalo AirStation WLI-PCI-OP-PC PCMCIA-PCI Bridge
Wireless Modes:	802.11b, 802.11g
Operating System:	Windows 2000 SP4
Java Version:	HotSpot Client VM 1.5.0_02-b09

A.7 Machine G

Manufacturer:	Patriot
Model:	PII 300MMX
Profile:	Desktop
Processor Model:	Intel Pentium II
Processor Speed:	300 MHz
Bus Speed:	100 MHz
Memory:	128 Mb
Wireless Adapter:	Buffalo Airstation WLI-CB-G54 & Buffalo AirStation WLI-PCI-OP-PC PCMCIA-PCI Bridge
Wireless Modes:	802.11b, 802.11g
Operating System:	Windows 2000 SP4
Java Version:	HotSpot Client VM 1.5.0_02-b09

A.8 Machine H

Manufacturer:	Various
Model:	N/A
Profile:	Desktop
Processor Model:	AMD Duron
Processor Speed:	1 GHz
Bus Speed:	100 MHz
Memory:	512 Mb
Wireless Adapter:	Mentor Wireless USB 2.0 Adapter
Wireless Modes:	802.11b, 802.11g
Operating System:	Windows XP Professional SP2
Java Version:	HotSpot Client VM 1.5.0_02-b09

A.9 Machine I

Manufacturer:	Various
Model:	N/A
Profile:	Desktop
Processor Model:	AMD Athlon XP 2400+
Processor Speed:	1.9 GHz
Bus Speed:	133 MHz
Memory:	512 Mb
Wireless Adapter:	Mentor Wireless USB 2.0 Adapter
Wireless Modes:	802.11b, 802.11g
Operating System:	Windows XP Professional SP2
Java Version:	HotSpot Client VM 1.5.0_02-b09

Appendix B

Trademarks

- Windows, Windows XP and Windows 2000 are registered trademarks of Microsoft Corporation in the United States and other countries.
- Intel and Pentium are registered trademarks of Intel Corporation.
- AMD, Athlon, Athlon XP and Duron are registered trademarks of Advanced Micro Devices, Inc. in the United States and/or other countries.
- Linspire is a registered trademark of Linspire Inc.
- Java, HotSpot, JavaSpaces and Jini are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.
- Buffalo and AirStation are trademarks of Buffalo, Inc.
- LifeBook is a trademark of Fujitsu Limited.
- Bluetooth is a trademark owned by Bluetooth SIG, Inc.
- Dell and Dell OptiPlex are trademarks of Dell Inc.
- T Spaces is a registered trademark of International Buisness Machines Corporation.
- All other trademarks and service marks are the property of their respective owners.

Bibliography

[BGLZ03] Nadia Busi, Roberto Gorrieri, Roberto Lucchi, and Gianluigi Zavattaro. SecSpaces: A Data-driven Coordination Model for Environments Open to Untrusted Agents. In Antonio Brogi and Jean-Marie Jacquet, editors, *Electronic Notes in Theoretical Computer Science*, volume 68. Elsevier, 2003.

[BGZ00] Nadia Busi, Roberto Gorrieri, and Gianluigi Zavattaro. On the Expressiveness of Distributed Leasing in Linda-like Coordination Languages. Technical Report UBLCS-2000-5, Department of Computing Science, University of Bologna, May 2000.

[BGZ01] Nadia Busi, Roberto Gorrieri, and Gianluigi Zavattaro. Temporary Data in Shared Dataspace Coordination Languages. In *Proceedings of the 4th International Conference on Foundations of Software Science and Computation Structures*, pages 121–136. Springer-Verlag, 2001.

[Bie04] Celeste Biever. Phones Face Power Failure. In *New Scientist*, number 2436, page 21, 28th February 2004.

[BLP00] Lorenzo Bettini, Michele Loreti, and Rosario Pugliese. Structured nets in KLAIM. In *Proceedings of the 2000 ACM symposium on Applied computing*, pages 174–180. ACM Press, 2000.

[BMMZ02] Nadia Busi, Cristian Manfredini, Alberto Montresor, and Gianluigi Zavattaro. Towards a Data-driven Coordination Infrastructure for Peer-to-Peer Systems. In *Proc. of Workshop on Peer-to-Peer Computing Co-located with Networking'02*, 2002.

[BMMZ03] Nadia Busi, Cristian Manfredini, Alberto Montresor, and Gianluigi Zavattaro. PeerSpaces: Data-driven Coordination in Peer-to-Peer Networks. In *SAC'03*. ACM Press, 2003.

[BS04] Rob Bjornson and Andrew Sherman. Grid Computing & the Linda Programming Model: An Alternative to Web-Service Interfaces. *Dr. Dobb's Journal*, (364):16–17,20,22,24, September 2004.

[BZ01a] Nadia Busi and Gianluigi Zavattaro. Publish/Subscribe vs. Shared Dataspace Coordination Infrastructures: Is It Just a Matter of Taste? In *Proceedings of the 10th IEEE International Workshops on Enabling Technologies*, pages 328–333. IEEE Computer Society, 2001.

[BZ01b] Nadia Busi and Gianluigi Zavattaro. Some Thoughts on Transiently Shared Dataspaces. In *The Workshop on Software Engineering and Mobility (at ICSE 2001)*, 2001.

[BZ02] Nadia Busi and Gianluigi Zavattaro. On the Serializability of Transactions in Shared Dataspaces with Temporary Data. In *Proceedings of the 2002 ACM Symposium on Applied Computing*, pages 359–366. ACM Press, 2002.

[BZ03] Nadia Busi and Gianluigi Zavattaro. Expired Data Collection in Shared Dataspaces. *Theoretical Computing Science*, 298(3):529–556, 2003.

[CDHR02] M. Castro, P. Druschel, Y. Hu, and A. Rowstron. Topology-Aware Routing in Structured Peer-to-Peer Overlay Networks. Technical Report MSR-TR-2002-82, Microsoft Research, One Microsoft Way, Redmond, WA 98052, 2002.

[CFH+03] M. Cilia, L. Fiege, C. Haul, A. Zeidler, and A. P. Buchmann. Looking into the Past: Enhancing Mobile Publish/Subscribe Middleware. In *Proceedings of the 2nd international workshop on Distributed event-based systems*, pages 1–8. ACM Press, 2003.

[CG90] Nicholas Carriero and David Gelernter. *How to Write Parallel Programs: A First Course*. MIT Press, Cambridge, MA, 1990.

[CIZ99] Paolo Ciancarini, Andra Imicini, and Franco Zambonelli. Coordination Technologies for Internet Agents. Nordic Journal of Computing 6, 215-240, 1999.

[CLZ99] Giacomo Cabri, Letizia Leonardi, and Franco Zambonelli. Reactive Tuple Spaces for Mobile Agent Coordination. In *Proceedings of the Second International Workshop on Mobile Agents*, pages 237–248. Springer-Verlag, 1999.

[Cro00] David Wallace Croft. Tuple Spaces — Research — VerticalNet, Inc., February 2000. `http://alumnus.caltech.edu/~croft/research/agent/tuplespaces/`.

[CSWH01] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. *Lecture Notes in Computer Science*, 2009:46+, 2001.

[CVV01a] Bogdan Carbunar, Marco Tulio Valente, and Jan Vitek. CoreLime: A Coordination Model for Mobile Agents. In *ConCoord 2001: International Workshop on Concurrency and Coordination*, Lipari, Italy, July 2001.

[CVV01b] Bogdan Carbunar, Marco Tulio Valente, and Jan Vitek. Lime Revisited. *Lecture Notes in Computer Science*, 2240:54, 2001.

[DFWB98] Nigel Davies, Adrian Friday, Stephen Wade, and Gordon Blair. L^{2}imbo: A Distributed Systems Platform for Mobile Computing. *ACM Mobile Networks and Applications (MONET), Special Issue on Protocols and Software Paradigms of Mobile Networks*, 3(2):143–156, August 1998.

[DGK+00] Patrick Doherty, Gösta Granlund, Krzystof Kuchcinski, Erik Sandewall, Klas Nordberg, Erik Skarman, and Johan Wiklund. The WITAS Unmanned Aerial Vehicle Project. In W. Horn, editor, *ECAI 2000. Proceedings of the 14th European Conference on Artificial Intelligence*, pages 747–755, Berlin, August 2000.

[DRBJS03] D. De Roure, M.A. Baker, N.R. Jennings, and N.R. Shadbolt. The Evolution of the Grid. In F. Berman, G. Fox, and A.J.G. Hey, editors, *Grid Computing — Making the Global Infrastructure a Reality*, pages 65–100. John Wiley and Sons Ltd, 2003.

[DWFB97] Nigel Davies, Stephen Wade, Adrian Friday, and Gordon Blair. Limbo: A Tuple Space Based Platform for Adaptive Mobile Applications. In *Joint Interna-*

tional Conference on Open Distributed Processing and Distributed Platforms (ICODP/ICDP '97), Toronto, Canada, 1997. Chapman and Hall.

[ED02] Huw Evans and Peter Dickman. Peer-to-Peer Programming with Teaq. In *Workshop on Peer-to-Peer Computing*, co-located with Networking 2002, Pisa, Italy, May 2002.

[Edw99] W. Keith Edwards. *Core Jini.* P T R Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1999.

[EQU05] EQUATOR IRC. `http://www.equator.ac.uk`, October 2005.

[FAH99] Eric Freeman, Ken Arnold, and Susanne Hupfer. *JavaSpaces Principles, Patterns, and Practice.* Addison-Wesley Longman Ltd., 1999.

[FDS+99] Adrian Friday, Nigel Davies, Jochen Seitz, Matthew Storey, and Stephen Wade. Experiences of Using Generative Communications to Support Adaptive Mobile Applications. *Distributed and Parallel Databases, Special Issue on Mobile Data Management and Applications*, 7(3):1–24, 1999.

[FGKZ03] Ludger Fiege, Felix C. Gartner, Oliver Kasten, and Andreas Zeidler. Supporting Mobility in Content-Based Publish/Subscribe Middleware. In *Middleware2003*, number 2672 in LNCS, pages 103–122, Rio de Janeiro, Brazil, June 2003. Springer-Verlag.

[FK99] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann Publishers, Inc., 1999.

[FPM04] Umar Farooq, Eric W. Parsons, and Shikharesh Majumdar. Performance of Publish/Subscribe Middleware in Mobile Wireless Networks. In *Proceedings of the 4th International Workshop on Software and Performance*, pages 278–289. ACM Press, 2004.

[Gel85] David Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[GGF04] Global Grid Forum, 2004. `http://www.ggf.org/`.

[Gig02a] GigaSpaces Technologies Ltd. GigaSpaces Cluster: White Paper, March 2002. Available from `www.gigaspaces.com`.

[Gig02b] GigaSpaces Technologies Ltd. GigaSpaces Platform: White Paper, February 2002. Available from `www.gigaspaces.com`.

[Gig03] GigaSpaces Technologies Ltd. P2P Cluster Patterns: White Paper, March 2003. Available from `www.gigaspaces.com`.

[Gnu03] Gnutella — A Protocol for a Revolution, 2003. `http://rfc-gnutella.sourceforge.net/`.

[Gre97] Robert Greig. Javelin: A Distributed Linda System. Final Year Project Report, University of Glasgow, 1997.

[Gre02a] Chris Greenhalgh. Equip - extensible platform for distributed collaboration. In *Second Workshop on Advanced Collaboration Environments, held in conjunction with the Eleventh IEEE International Symposium on High Performance Distributed Computing (HPDC-11)*, July 2002.

[Gre02b] Chris Greenhalgh. EQUIP: a Software Platform for Distributed Interactive Systems. Technical report, University of Nottingham, 2002.

[Har04] Lawrence Harte. *Introduction to GSM: Physical Channels, Logical Channels, Network, and Operation.* Althos, November 2004.

[HG99] Bob Heile and GTE Technology Organization. Solutions for the Last 10 Meters: An Overview of IEEE 802.15 Working Group on WPANs. In *Proceedings of the 3rd IEEE International Symposium on Wearable Computers*, page 10. IEEE Computer Society, 1999.

[HHL01] Zygmunt Haas, Joseph Y. Halpern, and Li Li. Gossip-Based Ad Hoc Routing. Technical Report TR2001-1849, Department of Computer Science, Cornell University, 2001.

[HJP02] K.A. Hawick, H.A. James, and L.H. Pritchard. Tuple-Space Based Middleware for Distributed Computing. Technical Report DHPC-128, Computer Science Division, University of Wales, Bangor, North Wales, October 2002.

[HR03] Radu Handorean and Gruia-Catalin Roman. Secure Sharing of Tuple Spaces in Ad Hoc Settings. In Riccardo Focardi and Gianluigi Zavattaro, editors, *Electronic Notes in Theoretical Computer Science*, volume 85. Elsevier, 2003.

[HW02] S. Hazel and B. Wiley. Achord: A Variant of the Chord Lookup Service for Use in Censorship Resistant Peer-to-Peer Publishing Systems, 2002.

[JF02] Brad Johanson and Armando Fox. The Event Heap: A Coordination Infrastructure for Interactive Workspaces. In *WMCSA '02: Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications*, Callicoon, New York, USA, 2002. IEEE Computer Society.

[JF04] Brad Johanson and Armando Fox. Extending tuplespaces for coordination in interactive workspaces. *J. Syst. Softw.*, 69(3):243–266, 2004.

[Joh02] Bradley Earl Johanson. *Application Coordination Infrastructure for Ubiquitous Computing Rooms.* PhD thesis, Stanford University, December 2002.

[JXT04] Project JXTA, 2004. `http://www.jxta.org`.

[Kam00] Alan Kaminsky. JiniME: Jini Connection Technology for Mobile Devices. Technical report, Information Technology Laboratory, Rochester Inst. of Technology, Aug 2000.

[Kar00] James Kardach. Bluetooth Architecture Overview. *Intel Technology Journal*, page 7, May 2000.

[KMS+93] Kimberly Keeton, Bruce A. Mah, Srinivasan Seshan, Randy H. Katz, and Domenico Ferrari. Providing connection-oriented network services to mobile hosts. In *Proceedings USENIX Symposium on Mobile & Location-Independent Computing*, pages 83–102, August 1993.

[KN05] Sumit Kasera and Nishit Narang. *3G Mobile Networks: Architecture, Protocols, and Procedures.* Professional Engineering. Higher Education, February 2005.

[Kno75] G. D. Knott. Hashing functions. *Computer Journal*, 18(3):265–278, 1975.

[L+01] Tobin J. Lehman et al. Hitting the Distributed Computing Sweet Spot with T Spaces. *Computer Networks (Amsterdam, Netherlands: 1999)*, 35(4):457–472, March 2001.

[LLK01] Kam-Yiu Lam, Guo Hui Li, and Tei-Wei Kuo. A Multi-version Data Model for Executing Real-time Transactions in a Mobile Environment. In *Proceedings of*

the 2nd ACM International Workshop on Data engineering for Wireless and Mobile Access, pages 90–97. ACM Press, 2001.

[LMW99] Tobin J. Lehman, Stephen W. McLaughry, and Peter Wycko. T Spaces: The Next Wave. In *HICSS*, 1999.

[LRS02] Qin Lv, Sylvia Ratnasamy, and Scott Shenker. Can Heterogeneity Make Gnutella Scalable? In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 94–103. Springer-Verlag, 2002.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms.* The Morgan Kaufman Series in Data Management Systems. Morgan Kaufmann Publishers, Inc., 1996.

[Man96] Steve Mann. "Smart Clothing": Wearable Multimedia Computing and "Personal Imaging" to Restore the Technological Balance Between People and Their Environments. In *Proceedings of the 4th ACM International Conference on Multimedia*, pages 163–174. ACM Press, 1996.

[MB98] Sanjay Kumar Madria and Bharat K. Bhargava. On the Correctness of a Transaction Model for Mobile Computing. In *Proceedings of the 9th International Conference on Database and Expert Systems Applications*, pages 573–583. Springer-Verlag, 1998.

[ME03] Gareth P. McSorley and Huw Evans. Tiamat: Generative Communications in a Changing World. In *1st International Workshop on Middleware for Pervasive and Ad Hoc Computing*, pages 37–44, Rio de Janeiro, Brazil, 16-20 June 2003.

[Moz04] The Mozilla Organization, 2004. `http://www.mozilla.org`.

[MPR01] A. Murphy, G. Picco, and G.-C. Roman. Lime: A Middleware for Physical and Logical Mobility. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS-01)*, pages 524–536, Los Alamitos, CA, April 16–19 2001. IEEE Computer Society.

[MPR03] A. Murphy, G. Picco, and G. Roman. Lime: A Coordination Middleware Supporting Mobility of Hosts and Agents, 2003.

[MT03] Ronaldo Menezes and Robert Tolksdorf. Adaptiveness in Linda-based Coordination Models. In *Proceedings of the 1st International Workshop on Engineering Self-Organising Applications*, LNCS 2977, Melbourne, Australia, July 2003.

[MW97] Ronaldo Menezes and Alan Wood. Garbage Collection in Open Distributed Tuple Space Systems. In Wanderley Lopes de Souza and Rogério Drummond, editors, *Proceedings of 15th Brazilian Computer Networks Symposium — SBRC'97*, pages 525–543, São Carlos, São Paulo, Brazil, May 1997.

[MW98] Ronaldo Menezes and Alan Wood. Ligia: A Java-based Linda-like Run-time System with Garbage Collection of Tuple Spaces. Technical Report YCS 304 (1998), University of York, 1998.

[MZL03] Marco Mamei, Franco Zambonelli, and Letizia Leonardi. Tuples On The Air: A Middleware for Context-Aware Computing in Dynamic Networks. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, page 342. IEEE Computer Society, 2003.

[Neu03] Rolf Neugebauer. *Decentralising Resource Management in Operating Systems.* PhD Dissertation, Department of Computing Science, University of Glasgow, April 2003.

[NKR+02] Chandra Narayanaswami, Noboru Kamijoh, Mandayam Raghunath, Tadanobu Inoue, Thomas Cipolla, Jim Sanford, Eugene Schlig, Sreekrishnan Venkiteswaran, Dinakar Guniguntala, Vishal Kulkarni, and Kazuhiko Yamazaki. IBM's Linux Watch: The Challenge of Miniaturization. *Computer*, 35(1):33–41, 2002.

[Nok04] Nokia — Latest Mobile Phones, 2004. `http://www.nokia.co.uk/nokia/0,8764,18062,00.html`.

[NPR00] Rocco De Nicola, Rosario Pugliese, and Antony I. T. Rowstron. Proving the Correctness of Optimising Destructive and Non-destructive Reads over Tuple Spaces. In *Proceedings of the 4th International Conference on Coordination Languages and Models*, pages 66–80. Springer-Verlag, 2000.

[OG02] Philipp Obreiter and Guntram Gräf. Towards Scalability in Tuple Spaces. In *Proceedings of the 2002 ACM Symposium on Applied computing*, pages 344–350. ACM Press, 2002.

[OP99] Bob O'Hara and Al Petrick. *The IEEE 802.11 Handbook: A Designer's Companion.* Standards Information Network IEEE Press, 1999.

[Ora01] Andy Oram. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies.* O'Reilly & Associates, Sebastopol, CA, 2001.

[Out02] Getting the Most out of Outrigger. White Paper, June 2002. `http://java.sun.com/developer/products/jini/OutriggerMostOf-8.pdf`.

[OZ98] Andrea Omicini and Franco Zambonelli. TuCSoN: A Coordination Model for Mobile Information Agents. In David G. Schwartz, Monica Divitini, and Terje Brasethvik, editors, *1st International Workshop on Innovative Internet Information Systems (IIIS'98)*, pages 177–187, Pisa, Italy, 8–9 June 1998. IDI — NTNU, Trondheim (Norway).

[PM01] Esmond Pitt and Kathy McNiff. *Java.rmi: The Remote Method Invocation Guide.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[PMR99] Gian Pietro Picco, Amy L. Murphy, and Gruia-Catalin Roman. LIME: Linda Meets Mobility. In D. Garlan, editor, *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, pages 368–377, Los Angeles, CA, USA, May 1999. ACM Press. Also available as Technical Report WUCS-98-21, July 1998, Washington University in St. Louis, MO, USA.

[PRM00] Bryan D. Payne, Gruia-Catalin Roman, and Amy L. Murphy. Managing Growth in Mobile Ad Hoc Networks Based on Linda in a Mobile Environment (LIME). Dept. Computing Science Washington University at St. Louis, May 2000.

[RD01] Antony Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. *Lecture Notes in Computer Science*, 2218:329, 2001.

[Res94] Mitchel Resnick. *Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Micorworlds.* MIT Press, 1994.

[RFH+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A Scalable Content-Addressable Network. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172. ACM Press, 2001.

[Riv92] R. Rivest. RFC 1321 — The MD5 Message-Digest Algorithm. `http://ietf.org/rfc/rfc1321.txt`, April 1992.

[Row98] Antony Rowstron. WCL: A Co-ordination Language for Geographically Distributed Agents. *World Wide Web*, 1(3):167–179, 1998.

[Row00] Antony Rowstron. Optimising the Linda **in** Primitive: Understanding Tuple-Space Run-Times. In *Proceedings of the 2000 ACM Symposium on Applied Computing*, volume 1, pages 227–232. ACM Press, March 2000.

[RW96] A. Rowstron and A. Wood. An Efficient Distributed Tuple Space Implementation for Networks of Workstations. *Lecture Notes in Computer Science*, 1123:510, 1996.

[RW97] A. Rowstron and A. Wood. Bonita: A Set of Tuple Space Primitives for Distributed Coordination. In *Proc. HICSS30, Software Track*, pages 379–388, Hawaii, 1997. IEEE Computer Society Press.

[RW98] A. I. T. Rowstron and A. M. Wood. Solving the Linda Multiple **rd** Problem Using the **copy-collect** Primitive. *Science of Computer Programming*, 31(2–3):335–358, July 1998.

[Ser04] Patricia Serrano Alvarado. *Adaptable Transactions for Mobile Environments.* PhD Dissertation, LSR-IMAG Laboratory, Grenoble, 2004.

[SM02a] Jim Snyder and Ronaldo Menezes. Using Logical Operators as an Extended Coordination Mechanism in Linda. In *Proceedings of the 5th International Conference on Coordination Models and Languages*, pages 317–331. Springer-Verlag, 2002.

[SM02b] Torsten Suel and Nasir Memon. *Handbook of Lossless Compression*, chapter Algorithms for Delta Compression and Remote File Synchronization. Academic Press, August 2002.

[SMK+01] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

[Sun00] Sun Microsystems Inc. Jini Distributed Leasing Specification, 2000. `http://wwws.sun.com/software/jini/specs/jini1.1html/lease-spec.html`.

[Sun02] Sun Microsystems Inc. Java Technology, 2002. `http://java.sun.com/`.

[Swe04] Daniel Sweeney. *WiMax Operator's Manual: Building 802.16 Wireless Networks.* APress, 2004.

[Tri99] Andrew Tridgell. *Efficient Algorithms for Sorting and Synchronization.* Phd dissertation, The Australian National University, February 1999.

[Vos97] Gottfried Vossen. The CORBA Specification for Cooperation in Heterogeneous Information Systems. In *CIA '97: Proceedings of the First International Workshop on Cooperative Information Agents*, pages 101–115, London, UK, 1997. Springer-Verlag.

[VvRB03] Werner Vogels, Robbert van Renesse, and Ken Birman. The Power of Epidemics: Robust Communication for Large-Scale Distributed Systems. *SIGCOMM Comput. Commun. Rev.*, 33(1):131–135, 2003.

[W+98] J. Waldo et al. JavaSpace Specification — 1.0. Technical report, Sun Microsystems, March 1998.

[WAC00] M. Waldman, Rubin A.D., and L.F. Cranor. Publius: A Robust, Tamper-Evident, Censorship-Resistant Web Publishing System. June 2000.

[Wad99] Stephen P. Wade. *An Investigation into the use of the Tuple Space Paradigm in Mobile Computing Environments.* PhD thesis, Lancaster University, September 1999.

[WMLF98] P. Wyckoff, S. McLaughry, T. Lehman, and D. Ford. T Spaces. *IBM Systems Journal*, 37(3):454–474, 1998.

[XP99] George Xylomenos and George C. Polyzos. TCP and UDP performance over a wireless LAN. In *INFOCOM (2)*, pages 439–446, 1999.

[YJK98] Tao Ye, H.-Arno Jacobsen, and Randy Katz. Mobile awareness in a wide area wireless network of info-stations. In *MobiCom '98: Proceedings of the 4th annual ACM/IEEE international conference on Mobile computing and networking*, pages 109–120, New York, NY, USA, 1998. ACM Press.