Supporting Operation in Ad-Hoc Environments

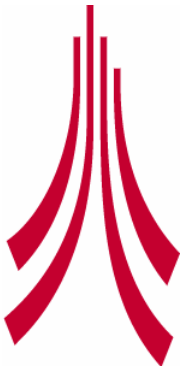
Matthew John Storey

B.Sc. Hons. (Lancaster 1997)

Computing Department,

Lancaster University,

England



Submitted for the degree of Doctor of Philosophy,

October, 2002.

Abstract

Supporting Operation in Ad-Hoc Environments

Matthew Storey

Computing Department,
Lancaster University, UK.

Submitted for the degree of Doctor of Philosophy,
September, 2002.

The increasing availability of devices with processing and communication capabilities, such as PDA's and mobile phones, has led to a desire to use various combinations of these devices for new and as yet unrealised operations. Not only are mobile devices expected to offer facilities like email and web browsing but also more demanding operations such as video encoding and playback. Attaining these operations within a stationary environment with high specification workstations is non-trivial; achieving this in a mobile environment with low power devices such as PDA's is a much more complex challenge.

The challenges posed in a mobile environment are diverse; this thesis focuses on the desire to operate in an ad hoc environment, one that is composed of devices that are capable of some form of computation as well as being able to communicate with one another. Operation within this '*active environment*' requires the ability to know what resources are available in the environment and provide a means for manipulation of these resources.

This thesis examines the currently available architectures for the detection and manipulation of resources in an active environment leading to a describing of the novel approach taken to help resolve these issues. The MARE approach described in this thesis is a novel combination of tuple spaces and mobile agents for enabling operation within active environments. The design and implementation of the MARE approach are detailed prior to performing an evaluation of the MARE approach for operating within an active environment. This evaluation highlights MARE as a

solution to facilitating operation within active environments. MARE is shown to operate favourably particularly in respect to resource discovery and configuration without reliance on a communications infrastructure, being sympathetic to bandwidth limitations and offering facilities to reconfigure resources to better suit environmental changes.

The overall hypothesis of this thesis is that through utilising tuple spaces and mobile agents in an active environment resource discovery and configuration can be achieved, and that this approach offers operation without reliance upon static servers and is capable of adaptation whilst utilising a minimum of communication bandwidth.

Declaration

This thesis builds upon the work carried out by Dr Stephen Wade examining the tuple space paradigm for operation within a mobile environment [Wade'99].

This thesis has been written by myself, and the work reported herein is my own. The following documented research has been carried out at Lancaster University.

The work reported in this thesis has not been previously submitted for a degree in this, or any other form.

Acknowledgements

The process of completing a Ph.D. thesis is a very personal journey; however it does require the help and support of many others to successfully complete the task. Here I wish to express my thanks to all who have been involved in supporting me during the completion of my Ph.D.

I would like to begin by thanking Professor Gordon Blair who has been my supervisor for the last three years. His guidance and continued support and encouragement throughout the process of writing this thesis is immeasurable. I would also like to thank Professor Doug Shepherd for helping me gain a position as a member of staff within the computing department at Lancaster University three years ago. Your support, kind words and long chats coupled with the trust you placed in me and the freedom you have given me is very much appreciated. Dr. Adrian Friday has been a long suffering friend whom has been a great source of encouragement as well as help through the maintenance of the L^2imbo tuple space implementation. There are many other people to thank from within the Computing Department at Lancaster, most notably Dr. Lee Johnston and my long suffering office colleagues, Dr. Keith Mitchell and more recently Dan Prince whom I thank for your patients and encouraging words and understanding when things are fraught.

Providing some light relief to the process of completing a Ph.D. has been Fylde College at Lancaster University, where I hold the position of dean. I have spent many an hour with you all in the college office and bar winding down after a long day in the department it is a pleasure to hold a position within the college.

I fully appreciate the support of my entire family, especially my grandparents and parents Chris and Derek for providing me with a good educational foundation and letting me disappear to the north to gain my university education. Your gift of a computer when I was a child has fuelled my desire to know how it works and how to fix it when it breaks leading me to where I am today.

The process of completing a Ph.D. is not easy and the person that has stood by me throughout the process bearing the brunt of the ups and downs has been my fiancée

Gill. Your love and support has been greatly appreciated, I would not have reached this point without the stability you have provided me and unwavering encouragement. I look forward to being your husband.

To you all, *thanks*!

Table of Contents

Figures

Tables

Chapter 1

Introduction

1.1 Overview

The emergence and development of technologies such as wireless modems, Private Mobile Radio (PMR), wireless LANs, IrDA and the developing Bluetooth technology is leading to a much wider acceptance and usage of mobile computing. Mobile phones are an example of the endorsement of mobile technologies that are commonplace in our society offering both voice and data capabilities. The increasing adoption of mobile computing is generating a heterogeneous pool of communication capable devices in the surrounding environment both in terms of mobile and static systems. Discovering and utilising devices in such a highly populated environment requires an approach capable of adapting to the continually changing availability and diversity of devices. This thesis investigates operation in a dynamic mobile environment containing a plethora of communication capable devices. The research presented is focused on the discovery and manipulation of services and devices to enable and enhance operations in an *active environment* as outlined in more detail later in this chapter.

This chapter outlines mobile computing and the issues associated with performing operations within a mobile environment. An active environment is defined and a scenario presented offering an illustration of the growing need to be able to operate in

such an environment. This chapter concludes by outlining the aims of the thesis and remaining structure.

1.2 Mobile Computing

1.2.1 Introduction

Mobile computing is playing an increasing role in everyday lives with mobile devices such as laptops and personal digital assistants (PDA's) are used both in work and recreational environments. The number and type of devices available form a *heterogeneous* collection offering and utilising different resources and operating systems. The availability of near constant connectivity through the use of wireless technologies such as Bluetooth [Bluetooth'99], GSM, GPRS and WaveLAN [AT&T'93] enables a device to be able to utilise resources whilst being mobile. The vision of *ubiquitous* computing is being realised through the wide scale adoption of mobile devices populating surrounding environments with devices offering functionality to users. Weiser offers this view of ubiquitous computing:

"Ubiquitous computing is the method of enhancing computer use by making many computers available throughout the physical environment, but making them effectively invisible to the user" [Weiser'93]

The heterogeneity of devices highlights the need to consider resource availability and interoperation techniques between devices of similar and differing type. This section provides an overview of the hardware and software developments forming a current mobile computing environment.

1.2.2 Hardware Support

The classical view of a computer being a powerful mainframe or desktop machine connected to a fixed network of some form is changing with the development of laptops and PDA's offering functionality comparable to desktop systems. The emergence of more powerful mobile devices such as handheld and palmtop computers is challenging the dominance of desktop machines for mainstream computing applications opening possibilities for mobile applications to develop. The devices outlined in Table 1-1 show the differences in mobile devices (including the increasingly important class of wearable devices).

Device (Example)	Weight (kg)	Hardware (Common)	Operating System
Laptop	> 1.5	LCD screen, keyboard, PCMCIA / USB/ serial / parallel ports, hard disk, cd-rom, floppy drive	UNIX, Windows 9x / NT / 2000 / XP, Mac OS
Handheld (Psion series 5)	< 0.5	LCD screen, keyboard, PC card / serial, compact flash storage	Symbian EPOCH32, Windows CE
Palmtop (Palm V)	< 0.2	LCD screen, serial port	Palm OS, Windows CE
Databank (Rolodex REX)	< 0.05	LCD screen, keyboard, 512KB storage	Custom
Wearable	Varied	Display, communications, very individual systems	UNIX, Windows 9x / NT / 2000 / XP, custom

Table 1-1 Mobile devices

The physical attributes of mobile devices such as available connectivity, computational power, storage and interaction methods differ from those typically available to desktop machines. These attributes require careful consideration when operating within a mobile environment often requiring some adaptation to best suit user and system requirements.

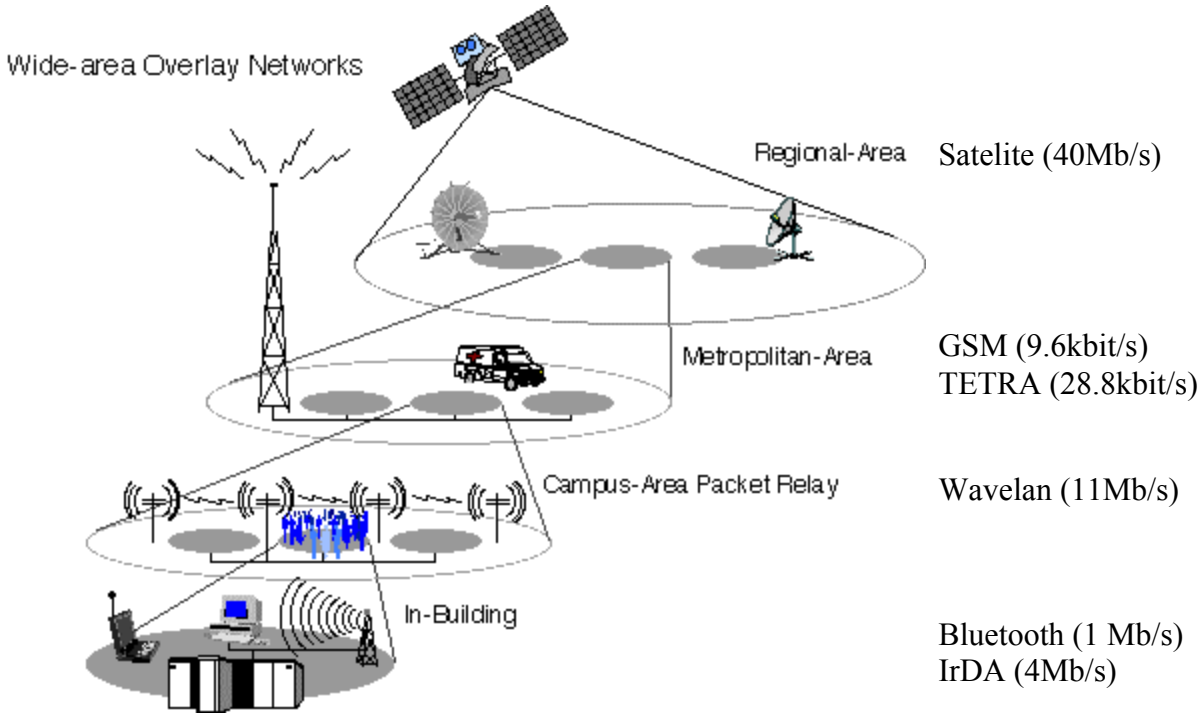


Figure 1-1 Wide-area overlay networks with available communications technologies [Stem'99]

Connectivity can be seen as a contributing factor to the operation of a device. Within built up areas technologies such as GSM, GPRS and WaveLAN may be available offering up to 11Mb/s, whereas in more remote areas solutions such as GSM may be the only available medium offering bandwidths in the order of 9.6Kbit/s. The potential to be able to utilise different communication types to maintain connectivity can be seen in Figure 1-1. Mobile connectivity in such an environment is typically variable and often intermittent due to the movement of a device through areas of differing type and fluctuating connectivity. An example of this can be seen in the migration of a user from a 11Mb wireless LAN environment on to a 9.6Kbit/s GSM connection and then moving into an area of no connectivity such as a tunnel. The handoff between connection types has to be considered carefully aiming to avoid a disconnection period without causing undue financial or performance costs and inconvenience to the user. Mobile devices are typically powered by battery offering a limited lifespan of the device requiring careful consideration of connectivity options on this basis.

Computational power of a device is often low or restricted to preserve battery power. The less power required by a device the longer it can operate on the same power source or the smaller the power source can be to maintain the same operational time. The operations that can be performed upon a device should be considered carefully to aid in the longevity of operational time. Techniques used in devices to save power include the slowing of the processor [Intel'96] or the placing of peripherals into a standby or a powered down mode. The entire device may also be turned off for periods of inactivity in an attempt to maximise operational life.

Storage capacity of a mobile device's working memory or storage area for data is typically limited compared to desktop machines ranging from a few kilobytes through to a few megabytes. Technologies offering thousands of megabytes of storage for data such as the Microdrive™ from IBM [IBM'00] lessons the importance of available storage however still offering far less storage than typical desktop machines. The restrictive storage capacity of most devices means that careful thought must be applied to the development of applications in terms of physical storage consumed both for applications, data storage and runtime memory usage. Techniques such as compression of data can be employed however consideration must be made to the tradeoffs against the extra time and computational power required to operate on such data.

Interaction methods employed by mobile devices include the more traditional keyboard and mouse pair as well as touch screens, assorted novel keypads, gestures and voice activations. Techniques for gaining contextual information by extending devices to include location, heat, light, accelerometers and pressure sensors are also extending the array of interaction methods potentially available to devices alleviating the need for a large display [Hinckley'00].

The previously outlined attributes require careful consideration in mobile systems despite developments such as larger storage capacities, novel interaction devices, more powerful batteries and bigger processors. The development of current mobile systems is spawning the deployment of smaller devices facing similar issues to those previously outlined. The newer smaller class of device is being developed for uses such as environmental monitoring [Kahn'99], [Gellersen'02], [Schmidt'99].

1.2.3 Application Support

Mobile applications such as tourist guides [Davies'99], parcel delivery tracking, traffic monitoring [TrafficMaster'02] and field worker support systems [Friday'96] have been and continue to develop exploring the possibilities offered by mobile computing. Experience in developing applications for traditional fixed network systems is filtering into mobile systems, examples of which include modifications made to network protocols, file systems and existing middleware platforms.

The most evident ***protocol*** used in fixed networks and mobile networks alike is the Internet Protocol (IP). However the current version four was not designed with the rigours of operation in a mobile environment in mind. IPv4 assumes the ability to form a direct connection from one host to another through a structure based on groupings of machines into subnets. When a mobile host moves out of a subnet into a new one it is required to gain a new IP address related to its new location. In order to address mobility, IP solutions have been defined that gain a temporary address in the new subnet [Perkins'92], [Ioannidis'93]. In addition solutions such as I-TCP [Bakre'95] and Snoop TCP [Amir'95] operate over the IP layer making use of routers at the edge of wireless networks to provide retransmission and forwarding of IPv4 packets. These solutions help in reducing hand off times, and retransmission costs between subnets when gaining a new address aiding in overall performance and efficiency. Finally the IPv6 protocol and mobile IPv6 extensions have been developed with mobility in mind employing a similar, however more refined approach, to previous IPv4 solutions to handle mobility [Perkins'96], [Finney'99].

Distributed ***file systems*** are used extensively in systems for the sharing of resources. Solutions such as the Network File System (NFS) [Sun'89] and the Andrew File System (AFS) [Satyanarayanan'85] enable this in real time however they offer little support for disconnected operation. In contrast disconnected operation has been addressed by systems such as Bayou [Demers'94] and CODA [Mummert'95], through caching or hoarding of records and files for synchronisation upon reconnection. The reintegration of information back into the primary store is complex due to the potential merging of multiple changed copies, this is exacerbated by long periods of disconnection requiring complex algorithms and often user interaction to resolve conflicts.

The heterogeneity of systems in terms of type and operating system has exposed issues in carrying out operations between devices. Approaches addressing this heterogeneity have been made through ***middleware*** offering an abstraction layer between the operating systems and the applications. Such systems include CORBA [OMG'98], DCE [OG'99], RM-ODP [ISO'98], DCOM [Microsoft'98] and Java RMI [Sun'02] offering a means of sharing tasks between different machines and operating systems through the use of a common interaction method. The techniques employed to enable middleware to operate in mobile environments include application code base size reduction such as can be seen in implementations such as PalmORB [Román'99] and UIC [Román'01]. The use of buffering of interactions on the edges of mobile environments provides a means of reducing the impact of disconnections that occur whilst mobile.

This thesis is focussed on enabling the discovery and manipulation of resources in a mobile environment. This thesis describes middleware as operating between applications and the underlying operating systems. This is a convenient place to address the discovery and manipulation of resources providing all applications with resource availability information and will be the focus of the remainder of this thesis. Further exploration of existing middleware systems can be found in chapter 2 of this thesis.

1.3 Active Environments

The diversity that can be found in the computational devices that surround and aid in the daily lives of many people is immense. The increasing development of hardware that is typically smaller, faster, lighter with a greater longevity in usage has provided users with the ability to utilise computing technology in new and previously unimagined ways. The shear number of available devices that can be envisaged in an environment is expanding. It is foreseeable that these devices may desire interaction with one another to achieve a goal. Mobile phones and PDA's are now commonplace with greatly varied operating systems and available peripherals. The issues exposed in how the devices will be aware of and utilise each other is compounded by the almost inevitable lack of communication infrastructure. The lack of freely available structured infrastructure for communication leads to a need for devices to be able to

form an ad hoc structure to facilitate interaction with one another. This thesis defines an active environment as follows.

> 'An active environment is a *loosely coupled*, *ad hoc* grouping of a number of *communication capable heterogeneous devices*. The population of this group is *highly dynamic* where there is a high frequency of membership changes.'†

Examples of such an environment can be seen at meeting points with or without structure, for example meeting rooms or corridors where several people come together bringing devices such as laptops, PDA's, phones and other assorted peripherals. As well as physical devices, users may bring software services that may be of use to other members such as encryption, manipulation and compression functionality. The active environment has a dynamic membership requiring constant adaptation in usage of available resources compensating for the changes in the operational environment. The active environment is more than an ad hoc grouping that is formed and then disbanded; a further more detailed example of an active environment is described in the following section.

1.4 Motivating Scenario

The British Lake District is renowned for its beauty and tranquillity crossed by narrow winding roads and walkways through both rugged and sparse terrain. The beauty of this area attracts many visitors intent on walking, running, cycling and climbing. With such outdoor activities in a beautiful and yet harsh environment come the almost inevitable accidents requiring the attention of the rescue services such as the mountain rescue team. In this section we describe a rescue performed by the mountain rescue team and issues raised by the introduction of multimedia support for the rescuers.

1.4.1 Basic Rescue

Upon the discovery of a person in distress a group of rescuers are dispatched to seek out the person and aid in recovery and treatment. The rescuers are dispatched from different locations travelling often by differing means. The rescuers converge upon the distressed party involving trekking across open terrain and rugged terrain such that climbing may be required (see Figure 1-2). Once the rescuers locate the distressed

† For the purposes of this thesis it is assumed that an active environment has a significant number of devices in the range 10-10,000 and that changes are frequent. Consideration of larger active environments is beyond the scope of this thesis. (See also Section 7.5.1)

party they converge forming a grouping of skills before bringing the distressed party to a point where they can be transported to a hospital for further treatment.

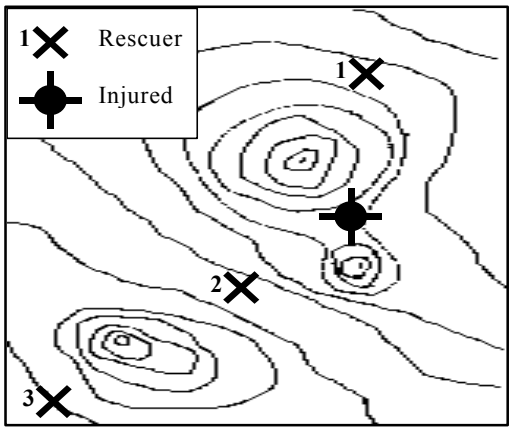


Figure 1-2 Rescuers converging on an injured party

In this environment there is a high likelihood of harsh weather in the form of wind, rain, sleet and snow requiring the rescuers to operate in a harsh terrain with typically unsympathetic weather conditions. Equipment carried by the rescuers is both for the distressed party and the rescuers amounting to a considerable quantity of equipment. The weight of such equipment is of paramount importance as excessive weight can hamper the progress of the rescuer.

1.4.2 Multimedia Enhanced Rescue

Through the development of a multimedia demonstrator project [Candy'98] the rescuers were augmented with multimedia systems aimed at enhancing the facilities available to the rescuer. Improvements included TETRA communications equipment, audio / visual capabilities, GPS compasses and medical monitoring equipment. Equipment provided to the rescuer includes a wearable computer consisting of a PC104 computer running Linux providing a point at which multiple devices can be connected such as TETRA communications equipment [ETSI'95], still and motion cameras, a medical monitor (PROPAQ [Allyn'02]) and GPS compass. The wearable is placed inside the back pack allowing some protection from the harsh surrounding elements whilst being less intrusive to the rescuer when carrying out a rescue. The peripheral devices are powered on and off as required to conserve battery life. The peripheral devices are heavy and often financially expensive resulting in a distribution of devices between rescuers.

With the multimedia enhancements the rescue of the injured party will start with the dispatch of the rescuers from separate locations with a subset of attachable devices. The distribution of devices amongst rescuers is due to the need to distribute weight between rescuers to limit the effect on progress of the rescue. The rescuers can keep in contact through the use of TETRA equipment gaining the location of themselves and other rescuers through the use of GPS compasses. Images may be gathered and transmitted between users from the digital motion and still cameras when required. Upon reaching the distressed party injuries can be captured by the cameras and encoded for transmission to interested parties such as hospitals who can then plan for the arrival of an injured party with some idea of the state of the arriving patient. The injured party can then be monitored feeding information to the hospital from the medical monitoring equipment as the rescue progresses.

1.4.3 Issues

This scenario has focussed upon an emergency service carrying out a rescue with and without multimedia enhancements. This scenario although focused upon the emergency services brings into focus other possible scenarios that could be envisaged forming an active environment. Other examples of active environments can be seen in any grouping or meeting such as a coffee break, lunch, informal meetings like the trip home in a bus, train or traffic jam. More subtle interactions can be achieved when moving through an area such as walking along a corridor, moving past a group of individuals, driving past other vehicles. The diversity of devices available and the possibility of static systems such as embedded servers to be within a grouping pose challenges to be addressed in discovery and facilitating interaction between these systems and making optimal use of surrounding resources.

1.5 Aims

This chapter has defined an *active environment* in which many communication capable devices can be present at any given time. This thesis aims to explore issues associated with operating within an active environment specifically the discovery and manipulation of resources. This thesis aims to:

- Explore in depth the active environment characteristics and requirements.
- Investigate techniques for resource discovery, focussing on the applicability of such techniques for operating within an active environment.
- Examine, in full, approaches for performing *efficient discovery* and manipulation of resources in an active environment.

These aims will be explored by taking the following steps:

- Examining existing approaches to discovery and configuration of resources in a hostile environment exploring weaknesses and strengths of each approach when applied to operating within an active environment.
- Exploring a novel approach for service discovery utilising the knowledge gained regarding existing approaches examined previously.
- Construction of a prototype of the novel approach previously outlined.
- Evaluate the prototype implementation against existing approaches before concluding on the suitability of such an approach for active environments.

This thesis will show that current solutions to operating within a mobile environment do not meet the requirements of an active environment, additionally proposing a solution that is designed to address these requirements. This will be achieved after a detailed examination of existing systems that address discovery of resources and systems that enable the manipulation of resources.

1.6 Thesis outline

This thesis examines the issues surrounding operation within an active environment proposing a means for discovery and manipulation of services and devices despite the unpredictable availability of them. Chapter 2 introduces mobile computing and address the differences between mobile and fixed environments. Chapter 3 then examines the desire to discover services and devices further examining methods of facilitating operation within a mobile environment. Chapter 4 introduces MARE a proposed architecture to aid in the discovery and operation within an active environment. Chapter 4 details the development of MARE whilst chapter 6 evaluates MARE offering concluding comments in chapter 7.

Chapter 2

Platform Support

2.1 Introduction

The previous chapter examined mobile and active environments noting that the majority of software operating within a mobile environment was not necessarily targeted specifically for it. Through examination of the active environment described in the previous chapter some issues have been highlighted relating to the discovery and manipulation of services. These operations are often performed as part of a middleware system offering an abstraction over underlying services. This chapter introduces and examines current mobile computing research, particularly middleware solutions developed specifically for mobile environments and as extensions to existing static environment solutions. The chapter then moves on to explore two key issues poignant to current research in mobile computing, i.e. discovery of services in the surrounding environment and agent technologies developed for fixed and mobile systems.

2.2 Mobile Distributed Systems

The development of distributed mobile systems has been largely focused on making existing software work on mobile systems; such approaches include extensions to CORBA [OMG'98], DCE [OG'99] and RM-ODP [ISO'98]. Other solutions that have been specifically designed to operate within a mobile environment include Odyssey [Satyanarayanan'94] and ROVER [Joseph'95], these approaches and others will be explored in the following section.

2.2.1 Extended Distributed Systems

Distributed systems developed for static networks typically consisting of servers and workstations were not targeted for the mobile environment where issues such as low bandwidth, intermittent connectivity, low computational power and small storage capacity are expected. Systems have been typically extended to better suit a mobile environment the most prevalent are explored in this section.

Common Object Request Broker Architecture and its Derivatives

Developed by the Object Management Group (OMG) as part of its effort to encourage use of object oriented techniques, the Common Object Request Broker Architecture (CORBA) [OMG'98] has been developed as an object based model of a distributed system. CORBA makes use of strongly typed interfaces to objects using an Interface Definition Language (IDL) to define the types used for interacting with an object. Interaction between objects is carried out though the Object Request Broker (ORB) forming a communication method for interaction between multiple objects of multiple languages and locations (Figure 2-1).

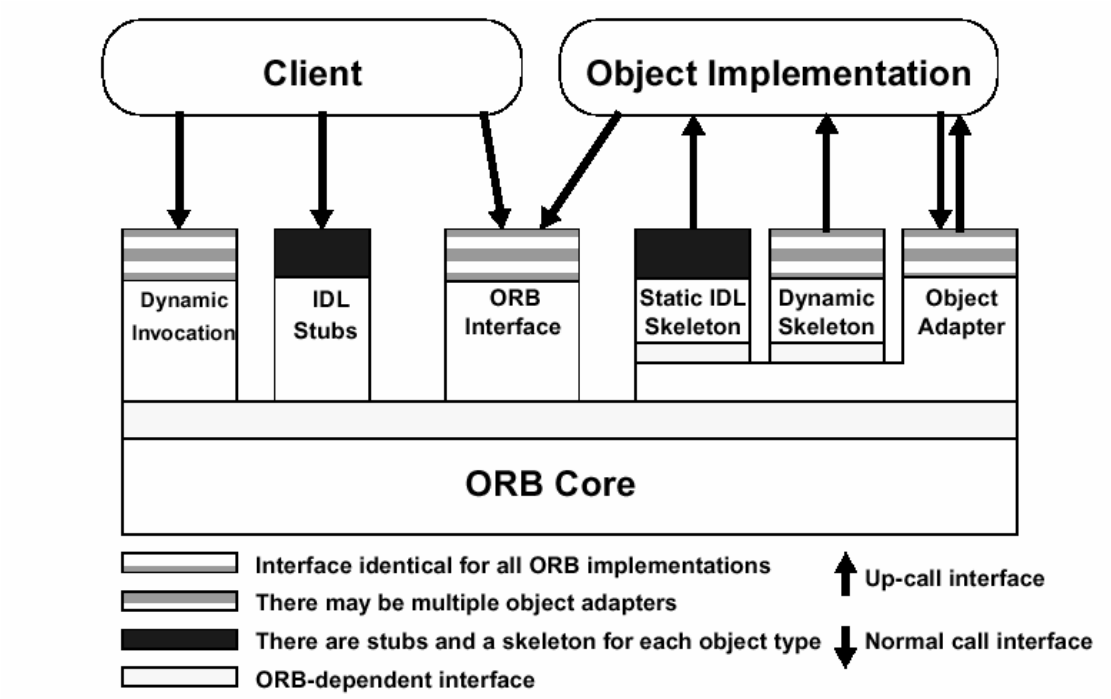


Figure 2-1 Structure of object request interfaces [OMG'98]

Extensions to CORBA have been developed aimed at supporting operation in a mobile environment. The Adapt project [Fitzpatrick'98] was developed from collaboration between Lancaster University and BT Labs; it was aimed at supporting mobile multimedia applications. This work developed QoS based adaptation through extending CORBA by adding stream interfaces, explicit bindings and open bindings. *Stream interfaces* are typed flows of data that have both control and data interfaces. *Explicit Bindings* can be used instead of *implicit bindings* placing more control over the formation of a binding with the programmer. *Open bindings* in effect enable the examination and manipulation of a binding and its sub components enabling adaptation to occur.

Using CORBA to facilitate a means of manipulating data flow can also be seen in systems such as the Reactive Adaptive Proxy Placement (RAPP) architecture [Seitz'98] and the Architecture for Location Independent CORBA Extensions (ALICE) Project [Haahr'99]. RAPP aims to support mobility through carefully placing of proxies using CORBA as a means of distributing the proxies to the most applicable location dependent on the operations being carried out, a example of which is the placement of a proxy near to or on a server for data intensive operations and near to a weak client for computationally intensive operations. Much of this work has

focussed on the placement of proxies with consideration to adapting the role and placement depending upon the status of QoS information. ALICE in contrast outlines *mobility gateways* placed at the edge of wireless networks acting as a point for proxies on behalf of mobile nodes.

Solutions targeted for small mobile systems such as PDA's include LegORB [Román'00] and PalmORB [Román'99] developed at the University of Illinois at Urbana-Champaign. These implementations offer a CORBA ORB small enough to execute on a Palm Pilot™ or Windows CE device. This work implemented a subset of the CORBA 2.0 functionality in effect reducing to only client side functionality of the ORB. Further work developed the Universally Interoperable Core (UIC) [Román'01]. The UIC is capable of generating small footprint solutions capable of incorporating multiple personalities such as a CORBA ORB or Java RMI [Sun'02] in a single implementation. The configuration of personalities allows for them to be included statically at compile time or dynamically at run time or a hybrid mix of the two approaches. These solutions offer compact implementations well suited to resource poor mobile devices. The solutions can be dynamically configured enabling the development of *What You Need Is What You Get* (WYNIWYG) systems. In essence, minimal solutions that contain or can obtain the elements required for a particular application to execute

Mobile DCE and Derivatives

The Distributed Computing Environment (DCE) [OG'99] was initially developed by the Open Software Foundation (OSF) which became The Open Group. The group aimed to develop a complete distributed computing infrastructure providing a scalable, secure environment capable of locating and utilising users, services and data within a heterogeneous environment. Extending DCE to operate within a mobile environment, Mobile DCE was developed at the University of Technology Dresden aiming to make mobility transparent to applications [Schill'95]. This was attempted through utilising domain managers acting as controlling authorities for station managers positioned upon fixed or mobile nodes. The use of domain and station managers provides a buffer allowing the ability to manipulate interactions to aid in handling disconnection and node movement as can be seen in Figure 2-2. Through the development of mobile DCE and related technologies it was found that transparent

operation is difficult to achieve and that further information from the application level is desirable [Adcock'99].

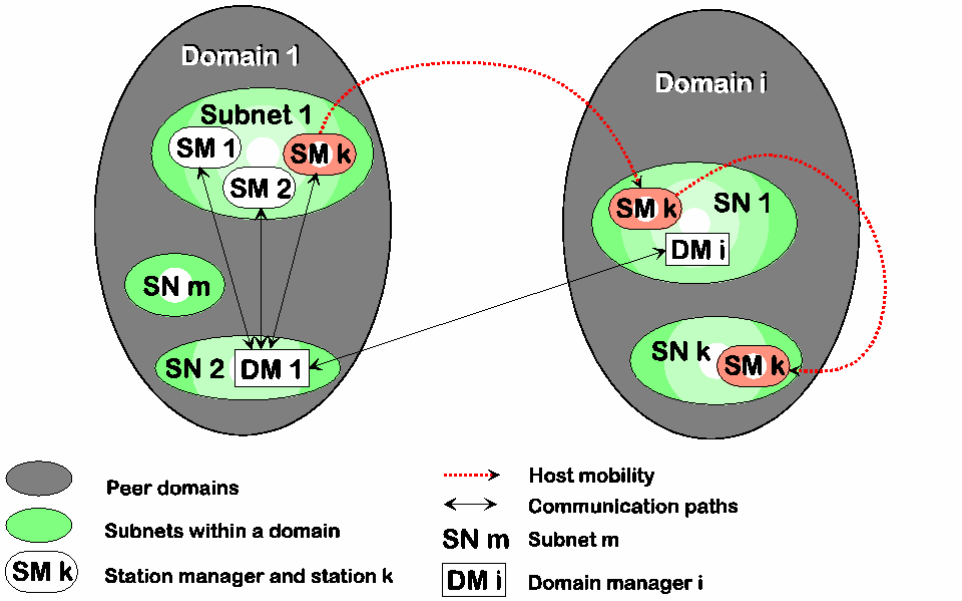


Figure 2-2 DCE overall generic structure [Schill'95]

RM-ODP and Derivatives

The Reference Model of Open Distributed Computing (RM-ODP) was created by a joint effort from the International Standards Organisation (ISO) and the International Telecommunications Union – Telecommunications sector (ITU-T). The RM-ODP defines a framework for distributed internetworking, interoperability and portability. The standard defines functions that provide services including a type repository, trader and relocator. These functions provide a means of storing and redistributing type, interfaces and location based information.

The development of ANSAware [APM'93] from the Advanced Networked System Architecture (ANSA) [APM'89] generated a partial implementation of the RM-ODP model. This system was further extended to better suit the mobile environment by the Mobile Open Systems Technologies (MOST) for the utilities industry platform at Lancaster University in collaboration with EA technologies [Friday'96]. The MOST platform implemented QoS-Managed bindings enabling querying of QoS parameters as well as attaining call-backs on QoS changes from a binding. Explicit bindings were implemented allowing more control over the formation of bindings; support for low

bandwidth communications media such as GSM was implemented through the development of Serial User Datagram Protocol (S-UDP) which the QoS enabled protocols QEX and G-QEX utilise. The MOST system enables an open approach allowing adaptation to changes within the system.

2.2.2 Targeted Distributed Systems

The following systems have been developed with mobility as a primary target environment although undoubtedly influenced by distributed systems developed for relatively static environments such as those previously examined.

Odyssey

Developed from work on the Andrew File System (AFS) and subsequent CODA [Satyanarayanan'90] file system, Odyssey [Satyanarayanan'94] is a set of extensions for mobility implemented in both the Unix operating system internals and system call level. The Odyssey architecture is formed from typed storage areas or volumes known as tomes that store data of the specified type. Odyssey provides an API of system calls providing the ability to negotiate a window of tolerance, receiving a notification upon exceeding of the tolerance. The architecture defines wardens that act as type specific managers and a viceroy that oversees the generic operations and administration of the wardens and clients. The system can be extended to have new wardens added to manage resources. Figure 2-3 shows the Odyssey client architecture with two Wardens installed.

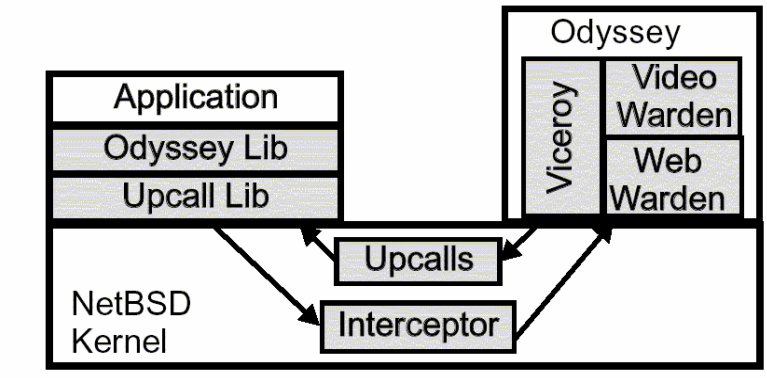


Figure 2-3 Odyssey client architecture [Satyanarayanan'94]

Rover

The Rover toolkit developed at the Massachusetts Institute of Technology (M.I.T.) is aimed at supporting the development of mobile applications [Joseph'95]. Rover combines Queued Remote Procedure Calls (QRPC) and Relocatable Dynamic Objects (RDO) that are akin to mobile agents examined later in this chapter to provide services for roving mobile applications. RDOs are imported, invoked and exported to and from servers based either on a pessimistic approach whereby RDOs are locked or an optimistic approach whereby conflicts from operations carried out on the RDOs are resolved upon reconnection. The QRPC defines a means of allowing non-blocking remote procedure calls to continue whilst a host is in a disconnected state, this is achieved through storing RPC interactions for replaying when the target becomes available upon reconnection. Figure 2-4 shows the three layers (application, system and transport) and core components (access manager, object cache, operation log and network scheduler) supporting disconnected operation in the Rover architecture.

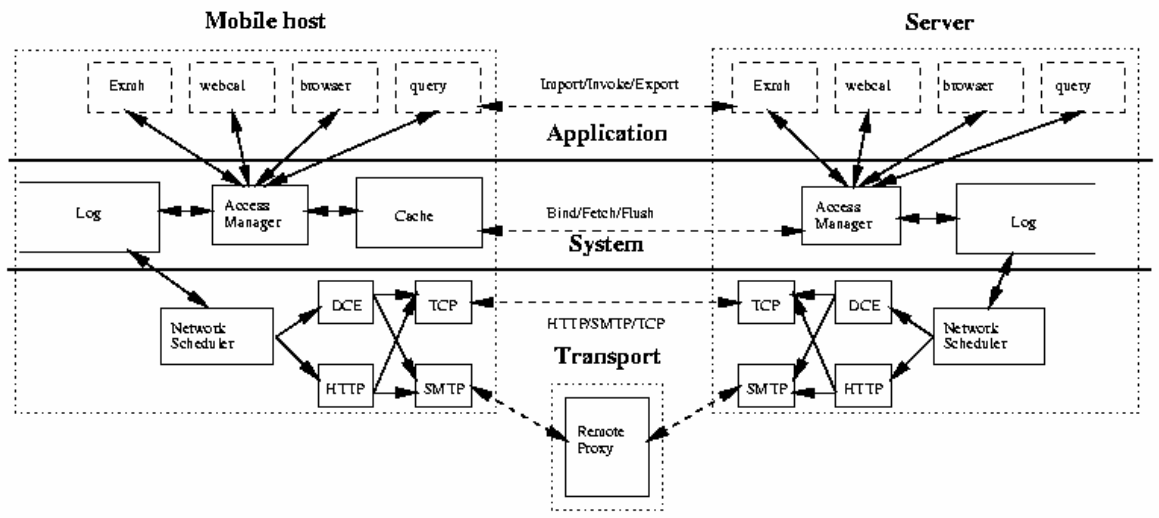


Figure 2-4 The Rover architecture [Joseph'95]

Bayou

Developed at Xerox PARC, Bayou supports data sharing amongst mobile users aiming to support application specific conflict detection, resolution and providing application controlled inconsistency [Demers'94]. The architecture prescribes to lightweight distributed storage on mobile devices allowing the distribution of data across multiple mobile devices. This is achieved through the careful replication of data with one primary copy and potentially many sub copies that may move between

hosts. This approach describes the reaching of eventual consistency where changes are propagated upon availability of connectivity such that the primary and all copies are identical.

2.2.3 Analysis

This section has examined systems developed from static network implementations and systems developed specifically for the mobile environment. Systems developed by extending existing approaches typically exhibit large implementations that are better suited to larger more powerful devices. Attempts such as the development of PalmORB, LegORB and UIC address the mismatch in resources available for the execution and those required in mobile devices.

One serious problem is that the synchronous RPC mechanism utilised by the previously examined systems can be disrupted at any time in a mobile environment. Attempts to address this issue can be seen as modification of RPC traffic such as buffering in ROVER, MOST and mobile DCE compensating for disruptions in RPC interactions. Other approaches include local caching of data and reintegration policies upon reconnection to the system as employed by Bayou, however, this approach and the approach made by mobile DCE poses problems in the reintegration of the cached data with a server version. This reintegration of data may also require user defined policies or user interaction to handle conflicts. The ability to adapt and operate within the surrounding environment is clearly useful and can be seen clearly in Odyssey with the negotiation of a window of tolerance.

In general the target of the previously examined systems is weak connectivity or short periods of disconnection often viewed as a graceful operation by detecting the degradation of connectivity. These systems do not inherently take advantage of surrounding resources preferring to operate with the resources they currently utilise. These systems address key issues when operating within a mobile environment demonstrating features such as adaptation to QoS changes, reintegration strategies for data access, interoperability with existing systems and distribution of operations between hosts. The examined systems do not offer an elegant unified approach thus warranting further examination in this area.

2.3 Resource Discovery

The ability to know which resources are available to a given host is the role of a service discovery protocol. Service discovery protocols that are prevalent include SLP [Verizades'97], SSDP [Goland'99] and Jini [Sun'02a] [Arnold'00]; these will be examined in more detail in this chapter.

2.3.1 Service Location Protocol

The Service Location Protocol (SLP) is designed to aid the discovery of resources for a user who does not necessarily know the specific location of the service they require [Veizades'97], [Guttman'99]. This protocol is aimed at enterprise networks with multiple shared services; it has not been aimed at global / Internet resource discovery. Other systems including Salutation [Salutation'99] make use of the SLP protocol by default when locating services in preference to its own service location protocol. There are two operational modes for SLP either with or without a directory agent. The directory agent is a repository for services such that a client can interact with a directory agent to discover services. The second mode of operation is used when a user agent cannot locate a directory agent therefore a user agent requests directly from the services using a service specific multicast address to request a specific type of service.

Operation with a directory agent is when a client generates a user agent the user agent will generate a multicast request to gain the location of a directory agent. If this fails the user agent enters a mode of operation that does not require a directory agent. The directory agent can also advertise its presence infrequently making itself available to user agents. The service agents acting on behalf of the services wishing to advertise their presence discover a directory agent in the same manner. A service agent registers with the directory agent when present and updates their registration periodically. The interactions can be seen in Figure 2-5.

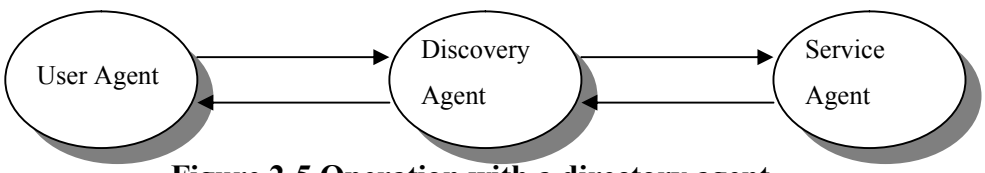


Figure 2-5 Operation with a directory agent

Operation without a directory agent requires the user agent to multicast its requests for a service to a predefined service specific multicast address that is responded to by appropriate service agents using a unicast response. Figure 2-6 illustrates the multicast request with multiple service agents responding to the request. If a directory agent is detected the service or user agents will register with and make use of the directory agent.

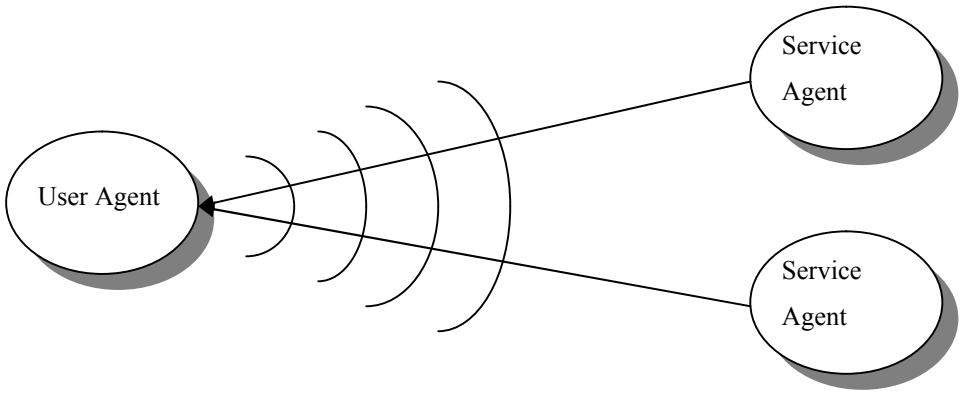


Figure 2-6 Operation without a directory agent

The SLP approach provides scalability for an increased number of service agents through the use of directory agents acting on behalf of multiple services. The use of directory agents does not preclude the use of direct interaction with service agents. A service is classified through its scope relating to the type of service.

When operating in an active environment the availability of a directory agent is unlikely due to the constantly changing content of the environment, requiring the reliance upon communication between user agents and service agents. In order to maintain a current view of surrounding services, a periodic multicast request for services from the user agent is required responded to by appropriate services using a unicast response. This can lead to the generation of a large number of requests and responses consuming potentially high quantities of available bandwidth.

2.3.2 Simple Service Discovery Protocol

The Simple Service Discovery Protocol (SSDP) [Goland'99] provides a mechanism for the discovery of services by networked clients with little if any configuration required. SSDP is an integral part of Universal Plug and Play (UPnP) [Microsoft'00]

initially developed by Microsoft. UPnP offers more than a service discovery protocol although the other features are outside of the scope of this section.

Operation can be viewed as client driven where clients request services or service driven whereby services update the information held by clients. Exchanged messages are in a HTTP format [Goland'00].

Service advertisements are performed upon generation of a service and periodically thereafter. A service announces its presence by multicast upon creation. The message consists of its type, expiry time, a unique identifier and a URL for accessing the service. The service is required to re-advertise itself within the expiry time otherwise the service should be deemed no longer available.

A ***client requests*** a type of service by multicasting a search request consisting of the type of service required or a request for all services. Each client that matches the request sends a unicast UDP message consisting of a unique identifier, type, expiry time and a URL for accessing the service back to the requesting client.

The specification for SSDP stipulates that a discovery multicast message should be transmitted three times to help to guarantee receipt by an appropriate client. For each arriving discovery message three response sets of six messages are required. Thus the number of messages produced for a client requesting a service from a single source can yield twenty one messages, of which three will be multicast discovery requests. As all responses are unicast UDP messages two clients requesting the same service will generate twice the number of responses.

2.3.3 Jini

Jini from Sun Microsystems [Sun'02a] provides a system for a client to locate and interact with services. Examined here is the ability to locate services for interaction by a host.

To advertise a service two protocols are used named *discovery* and *join*. During discovery a service will try and locate a *lookup service*; this is achieved by a multicast request for lookup services to identify themselves. Upon discovery of a lookup

service the service should *join* a lookup service gaining a lease that requires the service to update its entry in the lookup service before the lease expiry.

When a client wishes to use a service a *lookup* protocol is used. This involves the locating of a service by its type and perhaps its attributes from the lookup service. The service object containing the methods to enable interaction between the service provider and itself is loaded into the client. Once loaded the client and the server can interact directly using any available interaction method, as shown in Figure 2-7.

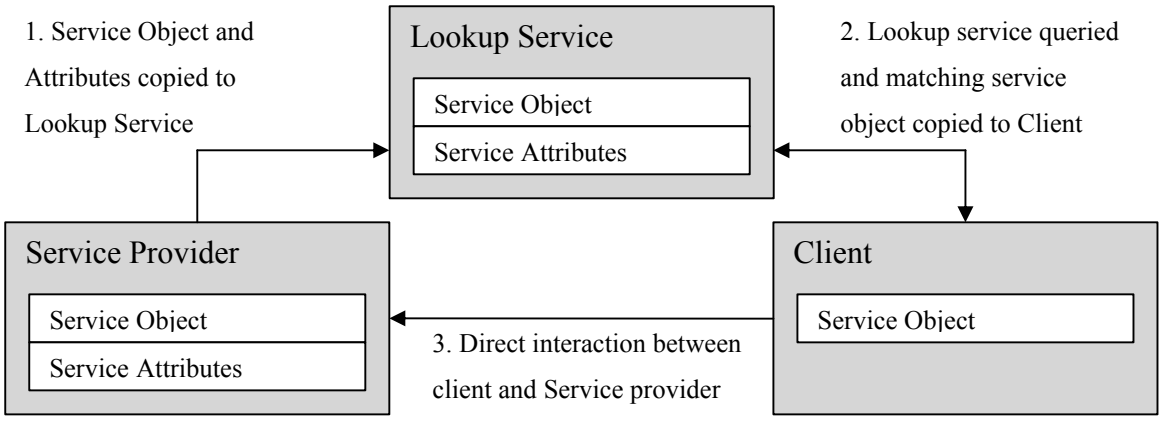


Figure 2-7 Jini Lookup Service interactions

In the absence of a lookup service Jini can resort to direct interaction with the service providers. It is possible to run a copy of the lookup server upon each participating node to form a distributed system that is not reliant upon a centralised server approach. Services can be represented by proxy to be able to be advertised if they are not capable of running the Java Virtual Machine and associated Jini services. A client can also register for the receipt of events in a similar manner as the service discovery, requiring lease renewal of the event registration and allowing the introduction of agents into the event path to allow for operations such as store and forward, combining and splitting of events. The Jini specification considers the desire for mobile hosts to interact considering the restrictions such devices place on applications. The specification however also assumes the existence of a network of reasonable speed and low latency, the active environment examined in this thesis will not provide such an environment.

> “The Jini system federates computers and computing devices into what appears to the user as a single system. It relies on the existence of a network of reasonable speed connecting those computers and devices. Some devices require much higher bandwidth and others can do with much less--displays and printers are examples of extreme points. We assume that the latency of the network is reasonable. ” [Sun’02a]

The practical implementation of Jini is relatively large and complex and although useful in larger systems it can currently be seen as a restriction upon smaller devices that may not have the resources to run the Jini system. The formation and disbanding of ad hoc groupings means that central lookup services will have fluctuating availability. When a lookup service is unavailable the clients are required to perform their own service lookups rather than talking directly to the lookup service. Detecting the absence of and operating without a lookup service produces extra communication overhead.

2.3.4 Summary

This section has concentrated on examining three prevalent service discovery protocols, SLP, UPnP and Jini. Note however that they have not been directly targeted for a mobile environment or the active environments outlined in the introduction to this thesis

The use of a server structure can be seen as a restriction in an active environment is unlikely to yield the correct server type due to constant membership changes. The back off strategies employed by these technologies place more computational overhead on a participating host as well as generating extra communications through operating in a synchronous manor.

An active environment does not lend itself to a service being available for long periods of time. There is a desire to be aware of the availability of a service and the arrival of other services that may meet the demands of the operation being undertaken as well if not better than the currently utilised service.

2.4 Mobile Agents

2.4.1 Introduction

The devices that are present within an active environment can be viewed as often being devices of a low power or restricted resources. In order to carry out operations on such devices in a similar manner as high power devices requires computation to be offloaded or shared between hosts. A technique for achieving this is explored in this section, namely mobile agents.

A mobile agent can be seen as a piece of code that is capable of movement to a host and execution at that host. The agent may travel to more than one host in order to complete its task. Mobile agents are commonly used for: -

- Distribution of processing through utilisation of other hosts.
- Fault tolerance through distribution of agents to carry out the same task.
- Reduction in bandwidth required by moving agents towards data sources for processing rather than moving large amounts of data across a network.
- Performing operations whilst a user has terminated an interactive session.

These attributes of mobile agents have made them particularly attractive to researchers examining mobile environments. In such an environment not only is the agent mobile but the environment in which it operates also changes. The potential hostility found in a mobile environment in terms of bandwidth, processing and connectivity can be addressed by the use of mobile agents. An agent can be sent to a more powerful node for execution or multiple agents may be produced to distribute the desired operation, this option also introduces a level of fault tolerance through the duplication of agents. The limited bandwidth with typically high latencies found in a mobile environment leads to a desire to move operations closer to the largest data source offering potential performance and reliability increases. A host can launch an agent and power down leaving the agent to carry out tasks on the originators behalf.

Mobile agents are typically written in portable code such as Java [Gosling'00] and Tcl [Raines'99] to execute the agents on hosts with differing operating systems. For larger

tasks more than one agent may be used and this leads to a desire for the agents to communicate with one another. What follows is an examination of several key mobile agent systems, namely Mole [Baumann'98], Concordia [Wong'97], Aglets [IBM'99], Agent Tcl [Gray'96], ARA [Peine'97], TACOMA [Johansen'95], Java-To-Go [Li'96], and Lime [Pico'98].

2.4.2 Mole

Developed at the University of Stuttgart and named after the project mascot [Baumann'98], Mole is a Java based middleware for the communication and execution of mobile agents on a standard Java virtual machine. The Mole system shown in Figure 2-8 offers an environment with a resource manager, directory service and a global naming scheme.

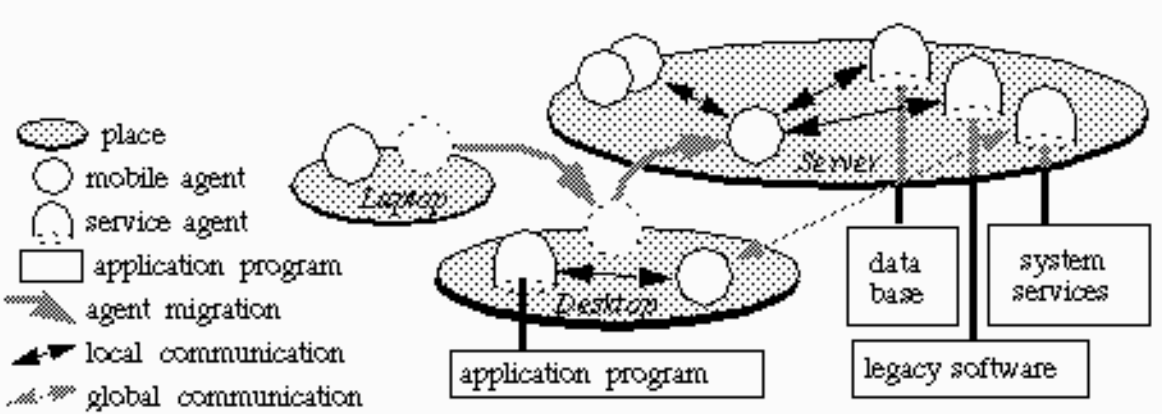


Figure 2-8 Mole system overview [Baumann'98]

The resource manager provides facilities for accounting and control of resources such as time at a location, network bandwidth and CPU time.

The directory service provides information on services on the local machine, such as agents. Each agent has a unique identifier based on twenty four bytes made up of two reserved bytes, two bytes for the port number being utilised, twelve for the IPv6 address of the host followed by two counters of four bytes each representing a system counter incremented on restarts and crashes as well as handling overflow from the lower four byte incrementing counter.

When an agent is created it is initialised and placed into the Mole system whereby an accepting node will call a prepare function on the agent prior to starting the agent. The

agent may register for a regular call-back from the system called a *heartbeat* to act as a trigger for the agent to potentially perform some operation, the heartbeat will continue until the *die* operation is called by the agent. The agent can migrate to another host through calling *migrateTo* that causes the agent to be suspended while the agent is serialised and moved before restarting. If an error occurs during moving or the target host cannot accept the agent an error message is received by the source host.

Communication between agents is performed through the use of badges attached to the agents identifying the agent as capable of receiving communication targeted at the badge. An agent may have more than one badge and the same badge can be worn by multiple agents providing the ability to perform group communications.

Mole uses Java serialisation for migrating agents between hosts. This has a side effect in that an agent cannot take its execution stack with itself. This means that a mole agent must compensate for the loss of execution state and recover when restarted. The agent programmer is responsible for maintaining state inside the agent. Code servers are used for storing class information required by the agent when it is generated or moved. If a code server crashes or is unavailable, the source of the agent may be asked for the classes required if it is available.

2.4.3 Concordia

Developed by Mitsubishi Electric Information Technology Center America, Concordia is a Java based mobile agent system [Wong'97]. Concordia has several key components, which interact to form a runtime environment for mobile agents. These components shown in Figure 2-9 include control for security, administration, persistent storage, events and migration.

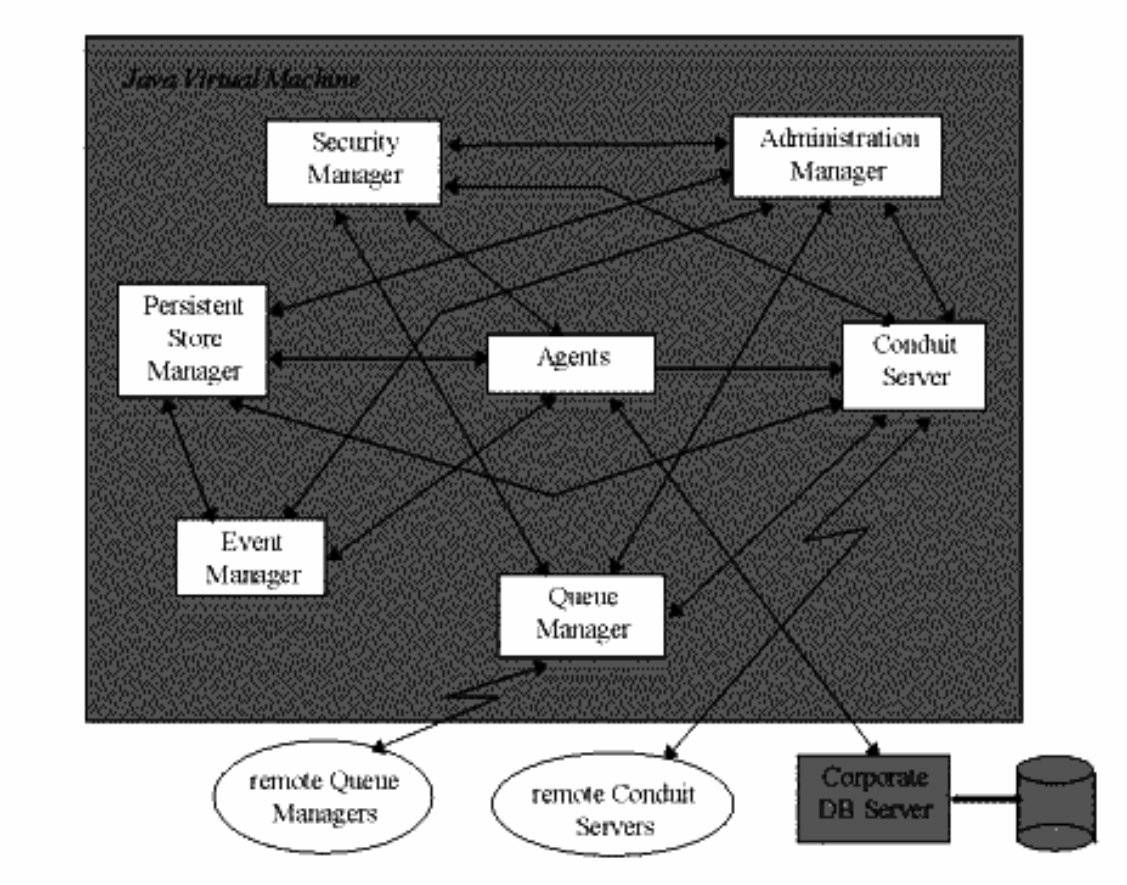


Figure 2-9 Concordia architecture [Wong'97]

The administration manager watches all other components with the view to recovery upon an error such as a system crash, also providing a graphical interface to the system.

Persistence has been introduced through the persistent store manager; agents as well as system state are stored by utilising Java serialisation. Using serialisation in this manner allows the system to be recovered to the point at which the serialisation was performed upon an event such as a system crash.

The conduit server controls agent transfer; when an agent calls the conduit server with a request to move the conduit server propagates the agent to the destination Concordia system providing the entry point function for the destination host to use. The destination and function to call is specified in an itinerary that moves with the agent. All classes required by the agent are transported with the agent for loading when the agent is restored. The transmission of all the required classes with the serialised agent

can lead to large transmissions upon the migration of an agent however removes the reliance of an agent upon another host.

Working with the local conduit server is the queue manager that is responsible for reliable transport of agents performing queuing of agent transfers inbound and outbound and handshaking with the remote queue manager. The queue manager is designed to facilitate fast recovery from errors by employing careful storage of state in a storage friendly manner.

Security is controlled by the security manager based upon users rather than creator's permissions placing the same permissions on the entire agent rather than the individual constructors of the constituent classes. Security in Concordia is employed in storage through the encrypting of stored information and in agent transmission through the use of SSLv3 protocol

Agent interaction is handled by two schemes i.e. asynchronous distributed events and agent collaboration. The events are managed and distributed to interested parties by the event manager while agent collaboration requires the programmer to construct an *AgentGroup* that enables agents to share information upon meeting.

2.4.4 ARA

Under development at the University of Kaiserslautern, Ara is an agent system that is not restricted to a specific language providing the agent language is capable of providing an interface that the Ara system can interact with [Peine'97].

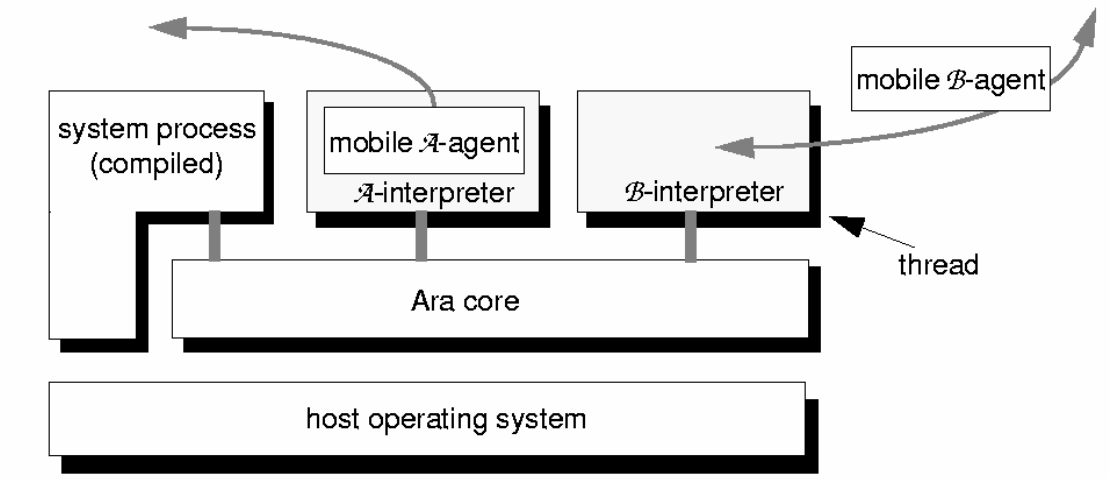


Figure 2-10 Ara system architecture [Peine'97]

Movement of agents is achieved with the capture of the internal state of the agent that then migrates transparently. Ara does not prescribe to attempting to shutdown and restore external relationships, instead relying upon the migration of an agent to be between two similarly equipped places. During the migration security is paramount both in terms of the admittance and hosting of agents as well as the encryption of the agent during transport.

Agent communication can be performed between different hosts using an asynchronous messaging system or through an application specific manner. Local interactions can be performed through the use of a service point where agents can subscribe and interact on the same host.

The utilisation of a resource is controlled by an agent's *allowance* of access to resources such as the CPU, memory, network bandwidth etc. This allowance can be capped at creation of the agent to restrict the creator's liability as well as being used as an entry bond to a host of the resources it requires. Ara also defines a means of transmitting an allowance between agents. The agent itself may also move if it discovers that the host it has been resumed upon is not capable of supporting its resource requirements. The transmission of agents can also utilise encryption methods such that authentication of the agent is required at the arrival of a host. Persistence is introduced as a means of fault recovery by allowing an agent to produce a *check point*

version which is stored and can be used as a means of rolling back in the event of system failures.

2.4.5 TACOMA

TACOMA (Tromsø And Cornell Moving Agents) is a system developed by Tromsø University and Cornell University supporting agents written in C, Tcl/Tk, Perl, Python, and Scheme [Johansen'95], [Jacobsen'99].

Data and state including code is stored in *folders* that an agent may access. Folders are defined as being stored in a static storage called a *file cabinet* or in mobile storage called a *briefcase* shown in Figure 2-11. This provides the agent the ability to leave data at a site in folders within a file cabinet until required whilst taking information with the agent in a briefcase. The briefcases can be signed using PGP (Pretty Good Privacy) and decoded by servers with the correct key providing a mechanism for incorporating some security into TACOMA.

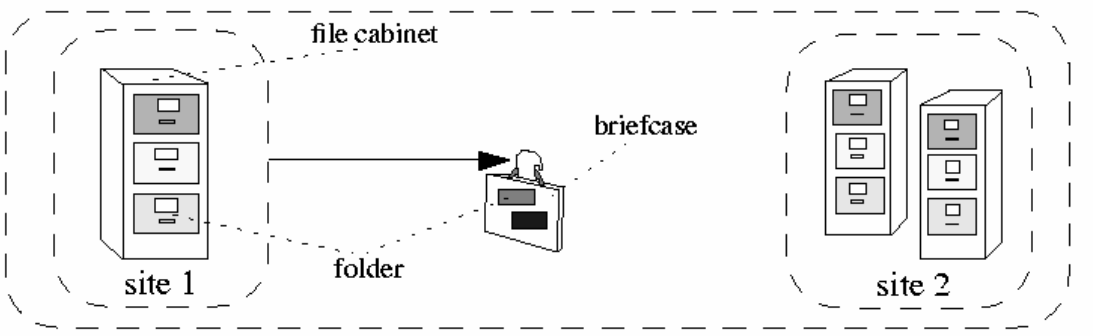


Figure 2-11 TACOMA: File Cabinet and Briefcase with Folders [Johansen'95]

Agent communication is facilitated through a *meet* operation specified by the agent to meet another specified agent and exchange a specified briefcase. In order to meet at a location a location is specified in the briefcase and then the meet operation is called on the briefcase containing the code and data.

The design of TACOMA is well suited to supporting multiple agents written in multiple languages making it a more generic environment capable of supporting a more heterogeneous pool of hosts. The desire to meet a specific agent or at a specific location forms a reliance upon an entity that can conceivably be unavailable within an active environment.

Development of TACOMA Lite for operation on small devices such as the Palm and Windows CE PDA required the refining of the TACOMA API to better suit the facilities offered by the devices [Jacobsen'97]. This implementation uses the same techniques as TACOMA also utilising technologies such as Short Message Service (SMS) and e-mail to encapsulate agents. To address disconnection issues TACOMA Lite assumes the existence of a *hostel* on a stable host, typically the synchronising host of the PDA for an interaction point between PDA and the rest of the world.

2.4.6 Lime

LIME [Picco'98] is a system that utilises the Linda [Carriero'89] tuple space principals discussed in more detail later in this thesis in combination with mobile agents.

An agent has an *interface tuple space* associated with the agent that moves with the agent when it is migrated containing information specific to the agent and unsent information from the agent. Interacting agents utilise a *transitively shared tuple space* forming a federated tuple space that can be spread over multiple hosts providing they are capable of communications forming a *federated tuple space*. Each host provides a *host-level tuple space* for all agents on that host to share and all hosts share a *LimeSystem* tuple space that is read only for agents containing information on agents. Figure 2-12 shows the migration of a Lime agent with transiently shared tuple spaces encompassing physical & logical mobility.

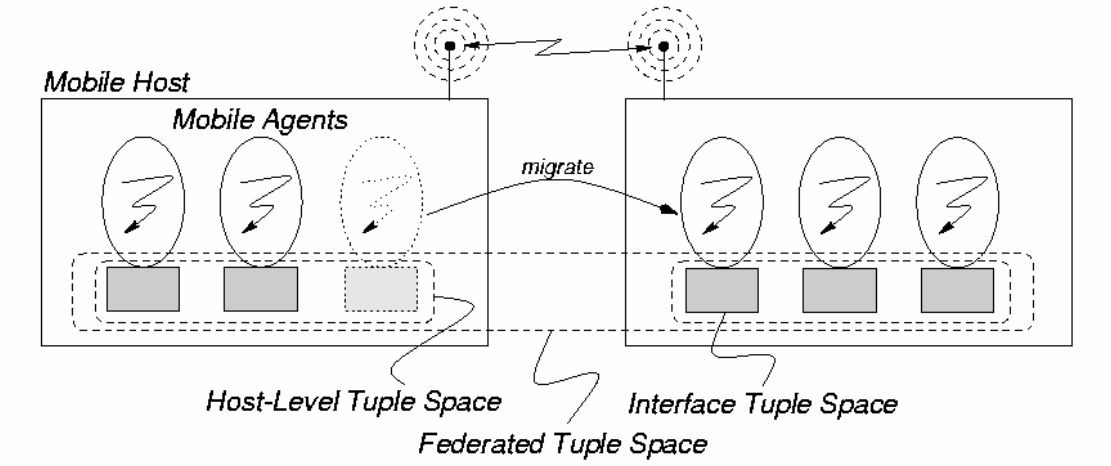


Figure 2-12 Lime: migration of an agent [Picco'98]

The carrying of information can generate large agents potentially holding key information required by other agents that may not have been transmitted; this could be exacerbated if the agent is lost by system failures destroying the information. The use of the tuple space provides decoupling of hosts removing the reliance upon directed communications.

2.4.7 Agent Tcl

Developed at Dartmouth University and extended further under the name of D'Agent, Agent Tcl is an agent system for the distribution and execution of agents [Gray96], [Gray'02]. This architecture supports multiple languages including Java and Tcl with facilities to use agents written in other languages. Agent Tcl has been extended to facilitate operation within a mobile environment [Gray'96a].

The Agent Tcl approach uses fixed permanently connected *dock* machines that acts as a buffer for the mobile host. When the mobile host is contactable the system interacts directly. Upon disconnection operations from the fixed network requiring access to the mobile host are placed in local storage at the dock for the mobile host until a connection is detected. When a mobile device is migrating an agent, the agent is stored in local storage by the mobile host's dock master awaiting reconnection for the agent to be able to move. The internal state of the agent is captured, encrypted and digitally signed before transmission.

Navigation of agents utilises a hierarchical structure whereby agents can query a database for information about an agent they require use of to discover its location. The database is developed through navigation agents that traverse the mobile environment gathering information on agents and locations sharing it with hosts they may visit.

2.4.8 Aglets

IBM has developed and implemented an agent system called Aglets [IBM'99] designed to operate using web technologies and implemented in the Java language.

Aglets are generated and initialised before being placed into the environment where they are consumed and executed by a willing node. Upon registration at a node the aglet receives callbacks upon certain events such as when an aglet is to be cloned,

dispatched, retracted, deactivated, disposed and desired to migrate. The notification of a migration is required due to the use of standard Java serialisation such that an aglet can store its state ready for migration. Aglets are capable of transmission utilising a fetching of required classes on demand or through the encompassing of required classes with the serialised aglet during transmission. Aglets may be stored for latter retrieval providing a means of persistent storage. Communication between agents is performed through a proxy that is generated at the same time as the aglet. Communication is carried out on the same host making the proxy responsible for forwarding the messages to the recipient aglet that may be located on a separate host providing location independence for the aglet. Security is provided by means of examining the source of the aglet classes for the correct execution permissions.

2.4.9 Java-To-Go

Java-To-Go is a simple architecture for experimentation with mobile agents developed at University of California Berkeley [Li'96]. Agents are executed in *Hall Servers* that form the runtime environment for mobile agents. A ClassLoader is provided for each agent to load the required classes. The Agents that require extra classes can be collected from *Class servers*. Both the *Hall servers* and the *Class Servers* are at well-known locations exposing weaknesses in the potential for the Class servers to become unavailable rendering the agent unusable.

2.4.10 Analysis

This section has focused on agent based systems operating in static and mobile environments. These systems facilitate the movement of agents between environments and hosts either by allowing an agent to choose the most appropriate host (autonomous agent) or by directing an agent at run time or by a pre arranged route.

The active environment outlined by this thesis is not suitable for many features of the previously examined agent systems, most notably the reliance on stable servers for operations. Java-To-Go and Mole have reliance upon hosts for gaining classes required by an agent when it arrives at a host. Whilst this approach saves in the transmitted agent size it forms a reliance that cannot be guaranteed in an active environment. TACOMA and Ara require a specified meeting point to perform agent interaction requiring a host to be stable for the duration of a meeting. Agent Tcl uses

its dock machine for interaction where the dock machine is placed on a static or highly available network. Aglets require the hosts that the agent has previously visited to execute a proxy such that communications can follow the agent. Aglets requires all hosts the agent has visited to remain available to relay communications to the agent. Lime offers a different approach utilising a tuple space for interactions and migration of all associated data with an agent. When the Lime agent migrates associated data with that agent are moved as well. Other agents may wish to utilise the data encapsulated and moved with the agent causing the removal of critical data from a working environment.

2.5 Summary

This chapter has introduced currently available distributed systems targeted at operation in a mobile environment. The chapter has investigated two key aspects of performing operations within a mobile environment, service discovery and utilisation of services. Through the examination of existing systems it is clear that an approach is required that exhibits several key features to operate in an active environment.

- The removal of reliance upon a server based architecture as the nature of mobility produces fluctuating connectivity and periods of disconnection. This style of system can be seen in mobile agent systems relying on servers for meeting and storage of classes. Service discovery protocols also make heavy use of servers storing service descriptions. Furthermore reliance on static servers should be avoided.

- Using a minimal amount of bandwidth in keeping an updated view of resources and the transmission of agents.

- Adaptation to environmental changes to make better use of available resources as can be seen in existing mobile distributed systems such as Odyssey by notification and adaptation to changes.

The previously examined approaches to resource discovery and agent systems have shown a potential for discovery and configuration of resources. The following chapter will outline the approach taken by MARE to achieve discovery and configuration in an active environment using resource discovery and agent systems.

Chapter 3

Resource Discovery

3.1 Overview

The previous chapter examined methods of operation within mobile distributed systems examining in particular service discovery and agent based systems. Through this examination, the lack of support for discovery in ad hoc environments has been highlighted noting that existing approaches focus on discovery of services in static networks before adapting these solutions for mobile environments.

This chapter revisits the resource discovery approaches examined in the previous chapter before proposing a novel approach to resource discovery, namely the use of tuple spaces. The tuple space paradigm is then examined including suggested enhancements before examining individual implementations. This chapter concludes by examining the attributes of the tuple space paradigm for operation within an active environment before selecting an approach to be used as a resource discovery mechanism within an active environment

3.2 Resource Discovery

This thesis has examined service discovery protocols that provide processes with information of the services requested. Within discovery protocols physical devices are often seen differently to services. As in Jini, MARE views devices and services as one, in the case of Jini as services and MARE as resources. Both approaches make the assumption that both devices and services expose an interface that can be utilised by processes thus making no distinction between them.

The service discovery protocols examined in the previous chapter have a preference for operation within a centralized architecture. They aim to introduce scalability through the collection of data at central points requiring a single query to a server rather than multiple requests to many hosts; however reliable servers are typically unavailable within an ad hoc environment. The UPnP, SLP and Jini service discovery approaches are heavily reliant on the synchronous query / response approach generating two or more interactions for a single query. These approaches specify that a host responds to a multicast request for information with a unicast response. As several clients may be interested in the same service, multiple multicast requests and unicast requests are made for a service. This approach produces a potentially large amount of traffic especially when retransmission strategies are employed [Microsoft'00]. The caching of service information can be seen as advantageous as requests can be satisfied locally requiring no external access. However the information within a mobile environment requires continual updating due to the dynamic arrival and departure of resources potentially requiring a highly effective caching policy.

In order to address the issues raised previously a different approach to service discovery is required for operation within a dynamic mobile environment. There is a body of work examining the potential for adapting tuple spaces for facilitating operations in distributed and mobile environments [Wade'99] [Johanson'02]. What follows is an examination of the tuple space paradigm as a candidate for enabling service discovery in an active environment.

3.3 Tuple Space Paradigm

Gelernter et al conceived the tuple space paradigm in the mid-1980's at Yale University calling their implementation Linda [Gelernter'85], [Carriero'89]. The paradigm has been researched extensively in the parallel and artificial intelligence domains and more recently the potential of operating in distributed and mobile environments has been explored. The tuple space was developed to offer a means of interaction in concurrent systems other than message passing and shared variables. The approach offers an abstraction allowing the insertion, reading and removal of data from any participating host.

Tuple space

A tuple space can be viewed as an abstract shared space where processes on one or more hosts can insert and query data. In effect the shared space is analogous to distributed shared memory between one or more participating processes across one or more hosts.

Tuples

A tuple is a typed data structure consisting of a number of fields of either an *actual* or *formal* type. A formal field within a tuple is a field that has a type with no associated value. In contrast an actual field consists of type and value information. A third field type exists that consists of an *active* tuple differing from the previously examined *passive* tuples. The active tuple contains operations to be carried out on the tuple within the tuple space eventually forming a passive tuple. Once a tuple is placed within a tuple space it can only be destroyed or modified by removal from the tuple space before reinsertion if required.

Anti-tuples

To search a tuple space a means of creating a template for tuples to be matched against is required. The templates are termed *anti-tuples* consisting of actual and formal fields to be matched against other tuples. Actual fields are matched fully whilst formal fields act as a field to be matched against type only. The anti-tuple may match with more than one tuple in which case a matching tuple is chosen at random. An anti-tuple can be seen as destructively removing tuples from a tuple space or as a non-destructive query upon the tuples returning a copy of a matched tuple. Figure 3-1

shows the interactions of a producer and consumer on a tuple space, emphasizing the requesting of tuples from a tuple space using an anti tuple to subsequently match tuples against.

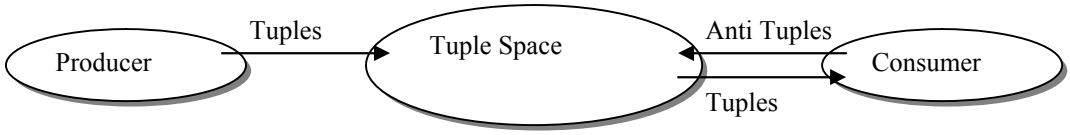


Figure 3-1 Communiction through a tuple space

Temporal decoupling

A tuple space is seen as offering temporal decoupling through its ability to act as a persistent store for tuples. This storage of tuples means that a host can insert tuples for consumption at a future time by the same or another process (See Figure 3-2). Anti-tuples can also be placed into a tuple space prior to the production of the matching tuple. This is unlike existing RPC mechanisms that, unless employing buffering in some way, require the explicit formation of a connection between client and server.

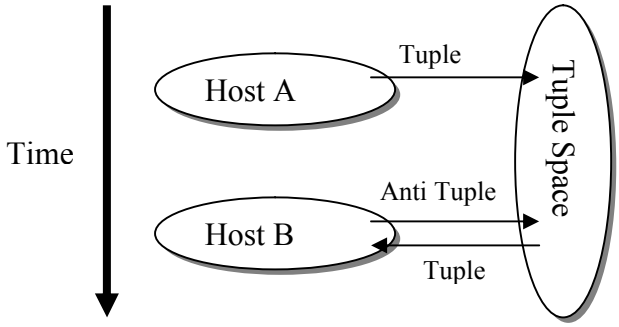


Figure 3-2 Temporal decoupling

Spatial decoupling

As processes do not have to directly interact with one another, a tuple space can be seen as providing spatial decoupling whereby the location of a process is irrelevant to the tuple space. This decoupling enables transparent support of group interaction through the use of a tuple space. In particular, a process can produce tuples that can be transparently read by multiple clients (see Figure 3-3). In the event of failure of a host the tuple space can compensate through utilising other hosts holding the same information introducing a level of fault tolerance into the tuple space.

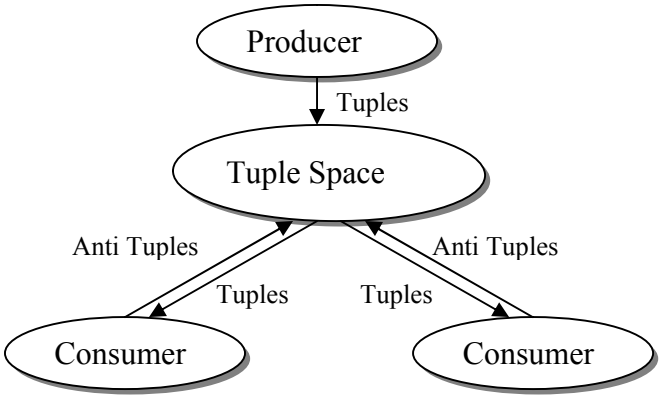


Figure 3-3 Group communication

Communication

Interaction between processes acting upon a tuple space has been defined as *generative communications* when interaction is exclusively carried out through a tuple space [Gelernter'85]. In this approach communication between processes is typically anonymous however directed communication can be achieved through the use of identification fields inside a tuple such that a tuple can be uniquely identified [Friday'99].

Application Programmer Interface

The application programmer interface (API) available to tuple space programmers typically consists of four operations in(), rd(), out() and eval() (see Table 3-1).

Primitive	Description
in	Blocking operation to remove a tuple from a tuple space, an anti-tuple is placed into the tuple space to be matched causing the return of the matching tuple.
rd	Blocking operation to return a matching tuple non-destructively from a tuple space. An anti-tuple is placed into the tuple space that is then matched against a tuple returning a copy of the tuple.
out	A non-blocking operation to insert a tuple composed of a number of formal and actual fields.
eval	A non-blocking operation to insert an active tuple composed of a number of formal / actual parameters and fields to be evaluated eventually forming a passive tuple.

Table 3-1 Standard Linda primitives

The four operations were initially designed as functions in the C programming language providing a simple yet powerful interface for distributed programmers. Further extensions to the API have been made for manipulation of tuple spaces and will be examined later in this chapter.

3.3.2 Tuple Space Enhancements

Linda was the initial implementation of the tuple space paradigm that offered a single tuple space with a simple API consisting of four simple yet powerful commands outlined in Table 3-1. The participating processes in the tuple space have access to all of the global tuple space. The tuple space paradigm defined by Gelernter et al has been developed further, notably through the development of multiple tuple spaces, bulk primitives and distributed implementations. Each of these enhancements will be examined in turn below.

Multiple tuple spaces

Tuple space implementations have been developed that support multiple tuple spaces as opposed to a single global tuple space [Davies'98] [Hupfer'90] [Bakken'94] [Carriero'94] [Rowstron'96]. Utilising multiple tuple spaces enables the tuples to be logically grouped into different tuple spaces allowing interested parties to participate in tuple spaces that contain information of interest to them (Figure 3-4). The use of

multiple tuple spaces helps reduce *unintended aliasing* of tuples where tuples are consumed non-intentionally because they match the template of an anti-tuple [Hupfer'90]. Performance can also be gained through the partitioning of tuples into multiple tuple spaces. The internal matching being performed in a tuple space is therefore on a reduced number of tuples.

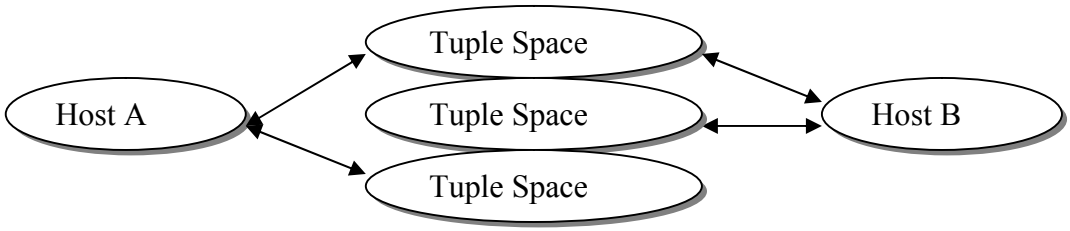


Figure 3-4 Multiple tuple spaces

Extensions to the application programmer interface

As previously mentioned the initial Linda API consisted of in(), out(), rd() and eval() operators providing the ability to add, remove, copy and evaluate tuples in a tuple space. These operators have been seen by many tuple space implementers as restrictive leading to the development of operators capable of exploiting multiple tuple spaces, multiple tuples and non blocking operations. The creation and destruction of tuple spaces has been examined in a number of differing ways considering attributes a tuple space may require such as if the tuple space is local or global to a particular process or machine, and other non functional properties such as security, fault tolerance or persistency attributes. Operations that manipulate multiple tuples such as insertion or removal in one operation are also found in many tuple space implementations. These operations often make use of multiple tuple spaces such that an operation can move or copy multiple tuples from one tuple space to another with a single operation call. Non blocking operations such as inp() and rdp() requiring examination of the tuple space to determine if an operations has completed have been developed. As noted by some tuple space implementations these operations are not precisely defined, leading to hard to predict behaviour. An example of the unpredictable behaviour of inp() and rdp() is proving that no tuple in the tuple space matches the request made. The whole tuple space would have to be searched whilst locking all interactions on the tuple space until the search completes

Distributed tuple space implementations

Distributed tuple space implementations aim to reduce the reliance upon a single host spreading or replicating tuples across multiple hosts. This provides the potential for a tuple space system to offer a level of fault tolerance. There is also potential to aid performance by caching or placing tuples near to the host requiring the tuples thus providing a local and potentially faster access to tuples within the tuple space. This distribution however adds complications in maintaining consistency across tuple spaces. For example, the removal of a tuple from a centralised tuple space can potentially offer a guarantee that the tuple has been removed whilst removing a tuple from a distributed system requires all hosts to agree that the tuple has been removed.

3.3.3 Tuple Space Implementations

Implementations of the tuple space paradigm have been developed offering differing features. Some prominent tuple space implementations are examined below.

Bauhaus Linda

Developed at Yale University Bauhaus Linda [Carriero'94] aims to generalise Linda by eliminating the distinction between tuples and tuple spaces considering the tuple space to be a set that can contain other sets hence providing support for multiple tuple spaces. This idea is further extended to allow the standard Linda operations in(), out() and rd() to act not only on single tuples but also on sets of tuples, offering support for the addition or removal of tuple spaces. Bauhaus Linda makes no distinction between tuples and anti-tuples utilising simple set inclusion rather than a more complex type and position of matching members of a tuple. The distinction between active and passive tuples is removed allowing the insertion using the out() operator removing the need for the eval() operation. Bauhaus Linda aims to provide a simpler view of the tuple space paradigm than the initial Linda implementation reducing the API providing support for multiple tuple spaces through overloading of its API.

Law-Governed Linda

Minsky et al at Rutgers University have developed Law-Governed Linda (LGL) [Minsky'94] providing a Linda model adhering to rules that control behaviour on and in the tuple space model. LGL supports three operations in(), out() and rd() on a single shared tuple space providing support for multiple tuple spaces by allowing the

shared tuple space to be divided into named sub spaces with associated criteria. The division of the shared tuple space effectively provides multiple tuple spaces accessed by processes meeting the required sub space criteria enforced by the LGL law. LGL offers a relatively simple API whilst internally enforcing its law onto the tuples / tuple spaces it contains, supporting the claim of introducing safety into tuple space communications by making invalid operations illegal.

Melinda

Developed at Yale University, Melinda [Hupfer'90] also aims to extend the tuple space paradigm set out by Gelernter et al to encompass multiple tuple spaces. The approach considers tuple spaces to be first class objects allowing operations to act on a tuple space as they would on a tuple. A tuple space may also have various parameters associated with them, for example fault tolerance, priority, protection and security. A tuple space is treated as a partially evaluated entity referred to as a live *tuple space image*. The tsc() (tuple space create) operation is added to generate a tuple space with optional properties defining the behaviour of the tuple space. The tsc() call does not return execution back to the calling program remaining live until the tuple space is destroyed.

The York Linda Kernel

Developed at the University of York, the York Linda Kernel [Douglas'95] has been targeted at running Linda across a network of transputers, each running a tuple space manager. The four Linda primitives in(), out() rd() and eval() are implemented as well as the collect primitives providing a means of manipulating multiple tuples within a single call to the tuple space. Each tuple space manager must be able to communicate with all other tuple space managers. A tuple is sent to one of the tuple space managers making removal of a tuple predictable as only one copy exists. The partitioning of tuple space managers would make tuples on either side of the partition unavailable to the other; this however breaks the requirement that a tuple space manager must be able to see all other tuple space managers. The York Linda Kernel provides a *local* or *remote* tuple space offering access to one or multiple processes respectively. Local tuple spaces are stored on the local machine.

Bonita

Bonita developed at the University York [Rowstron'97] defines a set of primitives for distributed coordination addressing weaknesses in previous approaches holding back the performance of tuple space operations. Although not specifically targeted at a particular tuple space system, the primitives assume the availability of a system capable of multiple tuple spaces. The proposed primitives provide the functionality of in(), out(), rd(), inp(), rdp(), collect() and copy-collect() in four primitives:

rqid = dispatch(tuple space, tuple | [template, destructive | non destructive])

> This is an overloaded primitive allowing the insertion of tuples or templates (anti-tuples) of a destructive or non destructive nature into a specified tuple space. When an anti-tuple is inserted, a *request identifier (rqid)* is returned that can later be used to test if the operation has completed. The rqid can then be used to copy or remove the tuple. This operation in effect provides a non blocking in() of a tuple or anti-tuple as defined in Linda using the rqid to check for completion of the desired operation.

Rqid = dispatch_bulk(tuple space 1, tuple space 2, template, destructive | non-destructive)

> This primitive provides for the destructive move or non destructive copy of multiple matching tuples from tuple space 1 to tuple space 2 providing a request identifier to examine the operation.

Arrived(rqid)

> Allows a request identifier to be queried returning true or false as to the arrival of a tuple as defined by the request identifier creating primitive.

Obtain(rqid)

> Unlike the previously examined primitives obtain() is a blocking primitive waiting for the completion of the primitive that created the request identifier.

These primitives provide a non blocking set of operations addressing the same concerns that inp() and rdp() examined previously. The inp() and rdp() primitives provide a polling mechanism of the entire tuple space whereas Bonita can collect results near to the requesting host. The arrived() primitive can simply consult the local

system allowing it to manage the marking of completed requests avoiding the continual polling of an entire tuple space. When considering a distributed tuple space implementation these operations reduce reliance on blocking operations. These operations whilst detecting tuples can generate less traffic throughout the tuple space.

FT-Linda

FT-Linda provides multiple tuple spaces of *private* or *shared* scope allowing access by single or multiple processes respectfully [Bakken'94]. A tuple space can be defined as being *volatile* or *stable,* the latter offering a tuple space capable of withstanding a node failure at the price of greater management overhead, in effect producing a distributed tuple space system.

JavaSpaces

Developed by Sun Microsystems *JavaSpaces* [Sun'99] offers a tuple space implementation built using Jini [Sun'02]. JavaSpaces utilises the Jini system to run a tuple space as a service on a server in a centralised manner facilitating multiple tuple spaces by using multiple services. There are four operators exposed to the JavaSpace programmer write(), read(), take() and notify(). The first three are analogous to the in(), rd() and out() operations found in Linda. The notify() operator provides a means of inserting a template for receipt of an event when the template is matched. Two other operators exist, readIfExists() and takeIfExists() providing the equivalent to the non blocking rdp() and inp() operators. Through the running of a JavaSpace service on all participating nodes, distribution of tuple spaces can be achieved thus reducing potential bottle necks at central servers. This technique can also be seen as offering a local tuple space through running a Java Space service on a local machine.

T Space

A TSpace [IBM'99a] is an implementation of a tuple space offered by IBM incorporating features from the database community such as transactions, persistency, flexible queries and XML support. TSpace operations include write(), waitToTake(), take(), waitToRead() and read() which are roughly analogous to out(), in(), inp(), rd() and rdp(). Other supported operations include scan(), equivalent to a copy-collect() operation, and capable of returning a matching set of tuples. eventRegister() provides a callback upon a (Write, Delete/Update) operation on a certain tuple type. There is

also a countN() operation that performs an operation similar to the scan() operation except returning the number of matching tuples. A TSpace system consists of servers capable of holding multiple TSpaces in effect providing a centralised multiple tuple space implementation; running the service on a local machine can also provide a local tuple space.

L^2imbo

Developed at Lancaster University L^2imbo [Davies'98] is a distributed tuple space implementation offering the four standard Linda primitives as well as the non blocking operations inp() and rdp(). Multiple tuple spaces are supported in both local and remote formats supporting the collect() and copy-collect() operations. L^2imbo also supports callbacks from the tuple space by placing an anti-tuple into a tuple space and receiving a callback upon addition or removal of a matching tuple. Fully distributing the tuple space across all participating hosts provides a local store of tuples providing matching firstly from the local store, reducing the need to contact remote hosts to satisfy request performance. If a host becomes disconnected from the network upon reconnection the host listens for changes made to the tuple spaces such as inserted or removed tuples. The local cache is updated to incorporating these changes eventually making the local cache consistent providing *eventual consistency*. The L^2imbo implementation also offers *unique withdrawal* of tuples, a method whereby each tuple will be removed from the tuple space once rather than potentially being removed multiple times by multiple hosts. Tuple spaces are distributed across multiple hosts with each host maintaining a cache of the tuple space such that requests will be satisfied locally when possible increasing performance and communication overheads. Through the development of the L^2imbo approach it was noted that the tuple space was not best suited to end-to-end communications as the lack of a bound communication path makes it difficult to monitor and adapt to changes in QoS. It was noted that the tuple space approach represents part of the communication solution requiring explicit bindings with QoS [Wade'99], in essence the tuple space is not best suited to directed communications.

3.3.4 Analysis

This section has focussed on examining the tuple space paradigm as a potential resource discovery mechanism for active environments. The tuple space paradigm

describes a shared space that tuples can be inserted, removed and read from using the in(), out() and rd() operations respectively. These blocking operations have been seen as restrictive spawning the examination of non blocking operations such that a process does not have to block until the operation completes. Bonita, JavaSpaces, TSpaces and L^2imbo have developed operations to address this issue. A desire for multiple tuple spaces to enable the grouping of related information into a single space or assign properties to a tuple space such as persistency or security has also been explored in all the examined implementations. The API calls have also been examined aiming to provide a more intuitive interface to the tuple space. The examination has resulted in renaming the in(), out() and rd() operations as can be seen in JavaSpaces and TSapces. L^2imbo also aims to address the distribution of a tuple space to improve performance and fault tolerance by careful caching of tuples. JavaSpaces has also aimed to address this issue however the relationship it has with Jini is not clear and in practice the reliance of JavaSpaces on Jini makes the approach inappropriate for active environments as outlined in chapter 2 of this thesis.

The L^2imbo implementation offers non blocking operations, bulk primitives, multiple tuple spaces and a fully distributed approach making it of particular interest as a mechanism for resource discovery within an active environment.

3.4 Summary

Through the exploration of existing service discovery protocols in the previous chapter it is clear that the majority of focus has not been directed at mobile devices wishing to perform service discovery operations. A different approach that harnesses the experience and benefits from the current service discovery techniques is required when operating in an active environment. This chapter has focussed on the tuple space paradigm as a candidate for facilitating resource discovery in a mobile environment. The benefits of using a tuple space for service discovery are a combination of the core properties of anonymity, spatial and temporal decoupling. These properties allow the natural use of group communications between multiple hosts utilising a tuple space. This allows a request to be serviced by any host with the appropriate information without the need to form explicit connections. These properties also allow a host to place information into a tuple space for retrieval at a later point in time by a host that may not currently be participating offering a degree of communication persistency.

It is proposed that resource discovery can be achieved through a distributed tuple space approach and configuration of such resources will be best achieved through the utilisation of mobile agents. The L^2imbo implementation of a distributed tuple space system offers a comprehensive list of operations and a truly distributed approach allowing for the continued operation of a system despite partitioning of a tuple space. The following chapter explores the benefits of combining the tuple space paradigm with agent technologies to perform discovery and configuration of resources in an active environment. The chapter will then outline the design of a prototype to explore the potential benefits of such integration.

Chapter 4

The Design of MARE

4.1 Introduction

The active environment has been introduced in chapter 1 providing a description of an environment that has a constantly changing membership with a lack of structured communications infrastructure. An example of the active environment was also outlined emphasizing the importance of knowing the availability of resources and being able to interact and configure such resources. Chapter 2 has highlighted weaknesses and strengths in systems designed to discover and manipulate services when applied to an active environment. A novel approach for the discovery of resources has been examined in chapter 3 offering the tuple space paradigm and more specifically the L^2imbo implementation as a resource discovery mechanism.

This chapter describes the MARE (**M**obile **A**gent **R**untime **E**nvironment) approach to resource discovery and configuration in an active environment [Storey‘00], [Storey’02]. The chapter draws together the discovery of resources through the use of the tuple space paradigm and configuration through the use of mobile agents. An analysis of the combination of these technologies is undertaken relating to the initial goals of the thesis. The chapter then outlines the MARE key design decisions before examining the architecture describing the role of each constituent component in turn.

4.2 Resource Discovery and Configuration

4.2.1 Resource Discovery

Current service discovery approaches have been previously examined in this thesis concluding that, whilst such approaches are undoubtedly useful within static networks and useable within mobile networks, they appear unable to operate effectively and efficiently within an active environment.

A resource discovery mechanism operating in a mobile environment must offer a robust system capable of addressing the requirements highlighted previously in this thesis. The resource discovery mechanism must operate without reliance upon server based architectures allowing the discovery of resources without the use of static servers. Resource discovery should be performed in a manner that consumes a minimal amount of bandwidth whilst being capable of adapting to operate within an active environment.

This thesis has introduced the tuple space paradigm concluding that this paradigm offers the potential to provide a resource discovery mechanism for operating within an active environment. In particular it has been proposed to use the L^2imbo tuple space implementation as a means of resource discovery in an active environment. The L^2imbo approach incorporates the key characteristics of the tuple space paradigm that makes it an attractive proposition for resource discovery in an active environment through the key features of anonymity and spatial and temporal decoupling. Furthermore to these core characteristics the L^2imbo implementation offers a fully distributed approach through the distribution and caching of information at participating instances. The approach also incorporates a recovery mechanism to allow a host to acquire updates, such as deletions and insertions to its local cache in case of periods of disconnection. This comprehensive set of features offered by L^2imbo makes the system an attractive proposition for resource discovery in an active environment. The approach however must be supported by a mechanism for configuring and combining resources for performing tasks in an active environment such as adaptation to environmental changes.

4.2.2 Resource Configuration

Once appropriate resources have been discovered the resources can be configured and utilised. The interaction with resources can be made in a direct unmodified manner such as direct calls to a device or through the use of proxies that can act as an intermediary. The use of proxies allows transformation or caching of data into a more applicable format for the environment a system may be operating within, for example a video stream compressor can be used to save bandwidth as shown in Figure 4-1.

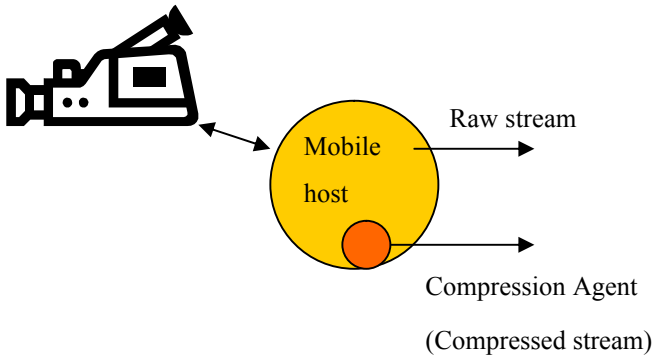


Figure 4-1 Compression agent on mobile host

The techniques used to interact with a resource are dependent on the environment the resource and interacting party are contained within. The environment this thesis is primarily examining is the active environment. When considering the active environment issues such as bandwidth and poor computational power lend weight to considering distribution of operations between hosts. This thesis proposes a method of performing the distribution of operations through utilising mobile agent techniques.

Mobile agent systems have been examined previously; through the analysis of these systems the mobile agent approach offers many beneficial features for utilisation in an active environment. Utilising resources in an active environment requires a solution where resources can be dynamically configured, adapting the configuration as required. Performing, monitoring and adjusting a configuration to maintain a relationship between a resource and a consumer can be seen as a problem well suited to autonomous mobile agents capable of seeking out and performing operations without user interaction.

The examination of existing agent systems in chapter 2 typically used in static and mobile environments outlines the major features of the respective systems. Chapter 2 concludes that an agent system must avoid reliance upon hosts as shown in agent implementations that require a host to remain contactable for relaying of information as seen in Aglets and Agent Tcl, class downloading as seen in Java-To-Go and Mole and meeting points as seen in TACOMA and Ara. Agents will be required to be autonomous to handle periods of disconnection, adapt to surroundings, detect available resources and make appropriate use of them.

None of the systems examined previously allow simple incorporation with a third party resource discovery mechanism, instead typically employing custom methods of resource discovery. Movement of and communications between agents is also performed by many differing techniques. To operate effectively in an active environment the transport of agents and associated requirements must be done in an efficient manner.

4.2.3 Analysis

This thesis proposes the merging of an agent system with the tuple space paradigm. This is achieved in part with the use of the tuple space `eval` operation developed in the original Linda implementation. The `eval` operation was intended to inject active code within a tuple space for computation. This operation is often seen as a simple operation designed to manipulate other tuples returning a tuple back into the tuple space. This thesis argues that a less restrictive approach is required allowing an agent to be inserted into the tuple space which has access to a broader range of operations. The active code should have as much access as is required to perform its tasks which may be more complex than simple tuple manipulations.

Revisiting the goals highlighted in chapter 2 the following examines the novel use of the tuple space paradigm and an agent runtime environment to facilitate resource discovery and configuration.

The removal of reliance upon a server based architecture as the nature of mobility produces fluctuating connectivity and periods of disconnection

The use of the distributed L^2imbo tuple space implementation allows operations to continue despite periods of disconnection. Requests and insertion of tuples can be

carried out on the local cache for incorporation into the caches of other hosts when they become available. The L[2]imbo approach transparently supports the notion of eventual consistency within the tuple space. Hosts will ultimately gain a consistent view of of the tuple space in their local cache through monitoring tuple traffic and adjusting the local cache contents based on the operations that are seen. Update such as deletions and insertions of tuples, piggybacked on normal tuple traffic reduces the number of connections required if transmission of updates occurred separately. The removal of reliance upon static and central servers is achieved through the use of the L[2]imbo distributed tuple space where central servers are not used. MARE uses the tuple space to advertise the resource descriptions at periodic intervals in a proactive manner. The approach not only provides updates to all participating hosts but also provides information about resources that a user may be unaware of but may be of interest to the host.

Using a minimal amount of bandwidth in keeping an updated view of resources and the transmission of agents

Attaining awareness of new resources in an active environment is achievable using existing service discovery techniques by performing a request for all information. The extra communications produced in a synchronous approach with multiple responses for a single resource, as can be seen in the UPnP approach examined in chapter 2, is clearly undesirable in a low bandwidth environment. MARE adopts an asynchronous advertisement of resources. Agents are placed in the tuple space once and then moved by the tuple space to all participating hosts. The agent is formed of a set of requirements and descriptive elements followed by the agent's executable code including associated code bases. Whilst the code base adds size to the transmitted agent it allows the agent to construct an instance without requiring extra communications acquiring required code base. This approach does not prohibit the ability to acquire code from remote hosts; however by default the agent will be migrated with its required code base.

Adaptation to environmental changes to make better use of available resources

The MARE system makes use of tuple spaces in particular incorporating a local tuple space that contains local system information. Each agent executing upon the host is made aware of system changes that are placed within the local tuple space. The use of the local tuple space provides a level of awareness as to the status of the local system such that an agent can adapt to the detected changes. Further to the local detection of changes, the availability of resources is relayed to the executing agents providing the agent with a list of resources that could be used instead of its current resources. This enables the agent to choose a better resource or a set of resources that can be combined to provide a more appropriate resource.

The MARE approach aims to explore and develop the tuple space paradigm approach combined with mobile agents particularly targeted at active environments. The details of the approach adopted by MARE are explored in more detail in the remainder of this chapter.

4.3 Major Design Issues

The merging of the tuple space and agent technologies is a novel approach to resource discovery and configuration. The combining of the technologies offers an efficient method of discovery through the use of the tuple space paradigm and utilising agent technology, providing a flexible mechanism for configuring and maintaining the availability of resources in an active environment. Some core aspects of the combination of these two technologies have been explored in this Chapter. This section aims to examine other major design decisions.

4.3.1 Eval

Eval was introduced in the tuple space paradigm as a means of distributing computation between multiple nodes. A specific tuple type is inserted into the tuple space and executed by a participating node often leaving a result in the tuple space in the form of a tuple. The approach has been implemented both as a specific operation call similar to an *out()* operation or as a specific tuple type that could be inserted by using the *out()* command. Further descriptions of the eval approaches can be found in chapter 3 of this thesis.

MARE implements eval as a new data type which avoids extending the L^{2}imbo API with a specific *eval()* operation. The new data type can be used in the same way as other data types in L^{2}imbo in the generation of tuples. The insertion of such a tuple is performed by calling *out()* with the tuple as a parameter. This approach enables a node to consume and run the operations contained in the eval field of the tuple.

Note that the contents of the eval field may not be a simple routine and may require code bases to be present to execute, such as other classes. To enable MARE to execute the contents of an eval field a method of making available the required classes has been implemented. MARE offers three approaches to moving associated code bases in an eval field.

I. The movement of only the core class relying on all required classes to be present at the consuming node.

II. The transportation of all required classes generating a larger eval field however all required components are available at the consuming node.

III. The implementation of a bootstrap mechanism that acquires the required classes at runtime, this offers a smaller eval field however requires the required classes to be available to the consuming node.

By default the MARE approach seamlessly takes all required classes within the eval field which allows for disconnected operation. In the active environment disconnected operation can be expected and is preferred as no further fetching of classes is required and no reliance on a node having all required classes is assumed. MARE can take either a new class or an initialised class that has been paused with associated runtime state. Upon starting at a participating node a common interface is used to restart the class.

4.3.2 Resource Advertising

Knowing what resources are available to perform an operation is clearly desirable in any environment; the method in which a host discovers what resources are available to it is not trivial in static networks with static resource repositories and large quantities of available bandwidth. Discovery of resources in an active environment

where bandwidth is a premium resource and availability of any host cannot be guaranteed is clearly a more complex scenario. The discovery of resources previously explored in this chapter is achieved in conventional service discovery techniques by a request and response approach as can be seen in SLP and UPnP. This approach is supplemented by announcements of the changing of a service's status such as removal as can be seen in the UPnP approach. In the case of SLP a request is made and a single response issued. However in the case of UPnP a request is issued three times followed by multiple responses for each request. The MARE approach is designed to reduce the overhead of continually requesting all available resources by proactively announcing all resources allowing all listening hosts to have knowledge of all available resources.

4.3.3 Resource Consistency

Having knowledge of available resources is important in utilising them, however maintaining a consistent view of resources is also vital in adapting usage to best suit changes in the operational environment. Within a static environment the changes in operational environment are typically infrequent. In contrast in an active environment maintaining a consistent view of resources is an issue complicated by constantly changing resource availability. Gaining a view of all resources allows the most appropriate resource to be used when a choice of resources is available. In the SLP approach continual requests and responses would provide a consistent view of resources, whilst in UPnP the same operation with multiple requests and responses is required. UPnP supplements the asynchronous discovery approach with announcements of a resource to try to alleviate the need for continual requests for resource updates. A request can be targeted at a specific type of resource reducing the amount of responses produced. However, to gain an impression of all available resources, a request for all resources must be made producing one or more requests with multiple responses. Within the MARE approach resources are announced on a periodic basis through the tuple space. The announcement serves to make every listening host aware of all available resources. The proactive approach to resource discovery enables a host to be made aware that a new type or instance of a resource exists. The awareness of all available resources enables the acquisition of information and potential utilisation of as yet unknown resources. A resource once received by the MARE instance is maintained and marked with an expiry time that is greater than the

announcement period. This allows the MARE instance to detect the availability of a resource by updating the expiry time when a resource is next announced and remove it from its cache if the expiry time is exceeded. All executing agents in the MARE instance are made aware of the arrival of a new resource and the removal of a resource determined by the lack of an update announcement. To minimise the number of connections required, resources from the same host are grouped into *resource bundles* reducing the number of connections required to transmit the resource descriptions. The bundling of resources can also be made more efficient by compressing the stream incurring additional overhead at the recipient and source in performing the compression / decompression. MARE supports the use of bundling of resources and compression transparently to the agents executing within the MARE instance.

4.3.4 Agent Execution

An agent needs to be executed in a safe manner both in terms of the agent towards the executing host but also the host affecting the agent. An agent is therefore required to trust the host and the host in turn must trust the agent before execution begins. Approaches typically focus on the protection of the executing host rather than the executing agent. Mole and Concordia use permissions associated with the agent to allow access to system objects such as IO routines providing a level of protection for the system against malicious damage instigated by the requesting agent. The use of the L^2imbo tuple space implementation allows multiple instances of MARE to execute utilising a single instance of the L^2imbo as shown in Figure 4-2.

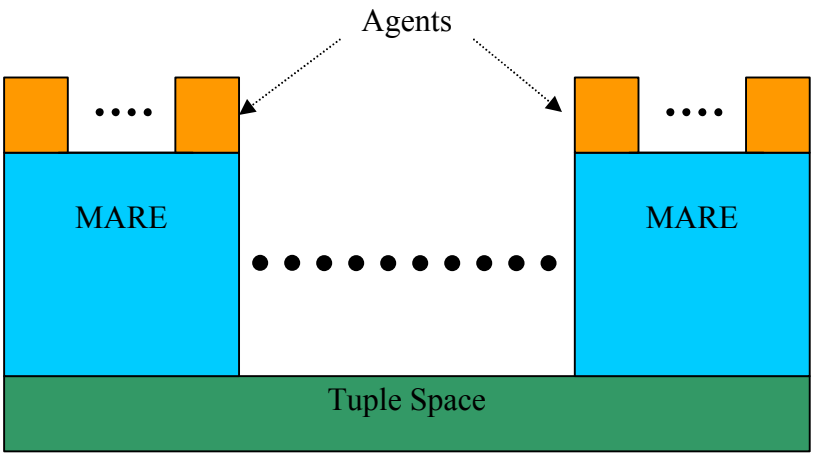


Figure 4-2 Multiple MARE instances utilising a single tuple space stub

In particular the approach depicted in Figure 4-2 allows multiple MARE instances to execute at different priority or security levels. Having multiple instances of MARE running in different security levels allows the execution of agents in an appropriate environment, for example an agent with a low level of trust determined by a host may choose to execute that agent in a restricted environment of low permission status. The overhead of running multiple MARE instances is minimal as all instances share the same distributed tuple space instance allowing the reuse of tuple space data such as resources and agents.

4.3.5 Agent Movement

The use of mobile agents to perform operations on behalf of a user at an appropriate point within a system requires the agent to be able to migrate to that point. The movement of code is not simplistic, particularly when state is also required to be moved with the agent. The movement of an agent has further associated issues requiring further examination including, the size of a transmitted agent, access to required resources and timeliness in achieving access to resources. The runtime state of an agent is typically not captured as this requires the storage and rebuilding of the runtime environment the agent was executing in. An agent is either run in a stateless mode such that it can be stopped, migrated and restarted without loss of information or sent notification that it will be stopped such that the agent can store its state. Further to this the tuple space can also act as a repository for state information of the agent. Allowing an agent to avoid serialising itself instead placing its state in the tuple space and then running a bootstrap mechanism to reload the agents state and continue execution.

The Agent systems examined previously in chapter 2 use assorted techniques to move agents. Targeting of an agent at a particular machine either prior to construction as can be seen in Concordia with the use of an itinerary or from one host to a neighbour as shown in Aglets can be seen as unwise if the target host becomes unavailable. The availability of a host for a sustained period of time is unlikely in an active environment. The target of an agent's migration is not the only consideration; the reconstruction of the agent requires careful consideration. An agent of a reasonable level of complexity is likely to consist of more than one element or code base such as classes and libraries. Whilst many are likely to be present on a participating host such as system libraries, many will not, such as user developed classes. This issue is examined in agent systems examined in chapter 2 through class servers which hold a repository of classes, the movement of required elements with the agent or fetching from the source of the agent. These approaches suffer in an active environment through the lack of available servers for class downloading, increased transmission size and potential for the source of an agent to become unavailable.

The MARE approach places the agent into the tuple space that then disperses the agent to all participating hosts. This approach makes the agent available for all listening hosts to execute. There is no required itinerary and no specified target for the agent, instead letting the description carried with the agent enable the decision to load and execute the agent. Each agent is executed only once controlled by the L^2imbo tuple space allowing the removal of an item from the tuple space only once. Agents carry with them any code base required for execution which may consist of classes or libraries. As highlighted in chapter 1 when examining the active environment the reliance on a central server is unwise; the combining of the code base with the agent removes a reliance on the source of an agent. This adds stability to the MARE approach at the expense of some bandwidth overhead of transporting agents with their code base. MARE also allows agents to load classes once they have started the execution providing the agent has the correct permissions on a host to perform such a task. An agent is migrated by either the MARE instance or by the agent itself. The MARE instance makes a request to the agent to make itself ready for migration before performing the actual migration. If the agent has not complied with the request, the MARE instance will stop and serialise the agent. However this can result in the loss of

state of the agent. The agent itself can also place itself back into the MARE instance thus allowing both parties to perform the migration of an agent.

4.3.6 Agent Loading

Consuming of an agent by a host is complicated by not just the mechanics of unpacking, reassembling and instantiating an agent but also the decision to do so on behalf of both involved parties, namely the host and the agent. The decision to consume an agent is made through not only the resources the agent requires being available on the host but also the host's willingness to accept and execute the agent. There are also associated security issues related to the trust the agent has in the host and the host in the agent and this should be maintained throughout the lifetime of the agent on the host. Agent systems examined in chapter 2 address the loading of agents relying on available resources on the executing host. The agent moves when further resources are required or operations are completed. The agent approach adopted in MARE allows agents to be executed in a MARE instance after meeting a set of criteria transmitted with the agent. This allows the system to gain an initial set of requirements to decide if the agent is appropriate to be executed within the instance of MARE before actually executing the mobile agent. In turn, once the agent is executed, it too can examine the MARE system to examine the chances of a successful execution. The criteria for loading an agent will be visited in detail in the following chapter.

4.3.7 Agent Communications

Interaction between agents is desirable for performing an operation requiring more than one agent such as searching where synchronisation may be required. Communication between agents in systems examined in chapter 2 is achieved by several techniques. Meeting at a specific location before exchanging information as is the case in TACOMA. Utilisation of proxies on each visited node as a route to the agent, as is the case for Aglets. These approaches suffer from a lack of communications infrastructure where such meeting points or communications relays can become unavailable before a meeting or communication takes place. Mole uses a different approach based on badges attached to each agent allowing the direction of information at the specified badge. Multiple badges can be used and shared between different agents and the Mole system controls relaying of messages. MARE adopts a

similar approach to Mole utilising badges that are carried by agents and revealed to the MARE instance upon arrival allowing the MARE instance to direct messages to the correct agent. The messages are transmitted through the tuple space distributed to all listening hosts. MARE encourages the transport of bulk communications such as multimedia streams through other techniques rather than through the tuple space (this is explored later in section 4.3.9). Other communications subsystems including peer to peer communications are not restricted; however they are limited in their use by the level of permissions an executing agent has on the MARE instance.

4.3.8 Interoperability

Allowing interoperability with resources not capable of executing an instance of MARE expands the number of available resources to the MARE system. Furthermore providing interoperability with existing agent and resource discovery systems such as SLP and UPnP is also a desirable feature. The MARE system has multiple options available to resources wishing to participate but unable to execute a full environment despite the minimal requirements of the environment. For example a digital camera can be attached and act as a resource on an executing MARE host by placing a stored agent into the MARE environment to act on behalf of the resource as shown in Figure 4-3.

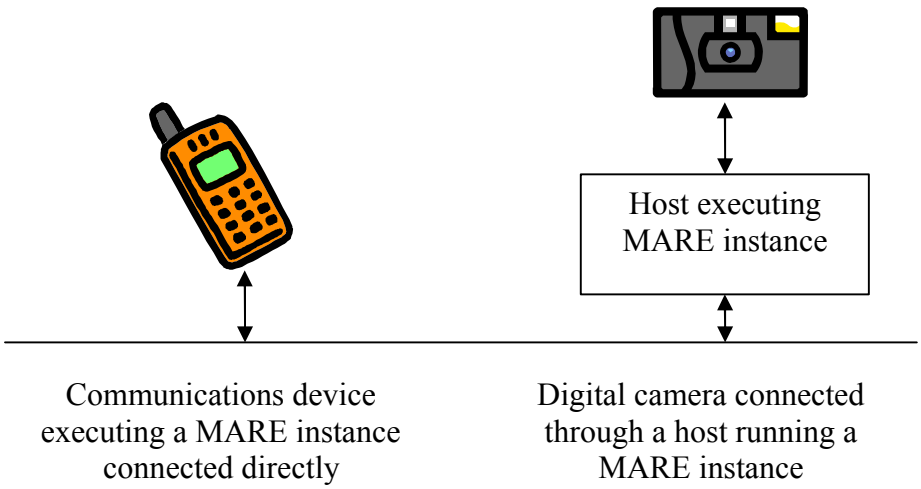


Figure 4-3 Interoperability with existing devices

Interoperability with existing standards can be achieved through the use of agents acting as a proxy between discovery protocols and agent systems. Whilst this level of

interoperability is not a goal of the MARE system such interoperability can be achieved through utilising agents as wrappers for other discovery services as depicted in Figure 4-4. This approach also offers the ability to execute and deploy agents targeted at other systems such as TACOMA or Mole by in effect running an agent in MARE encapsulating agents of other systems. The ability to perform such interoperation does however bring drawbacks in effect adopting the undesirable features found in existing systems explored in Chapter 2 such as inefficient use of bandwidth and reliance on central servers. In effect these approaches use a MARE agent as a proxy for resources enabling operations to be performed with devices that have inadequate facilities to execute a full MARE instance. Such resources include legacy devices without computational and communication capabilities as well as software that cannot be rewritten to utilise the MARE environment.

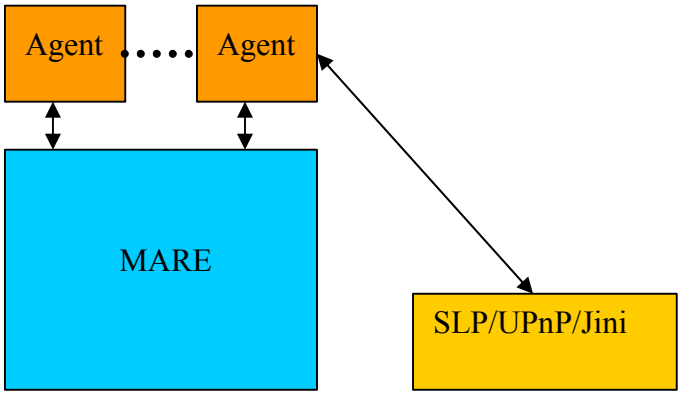


Figure 4-4 Service discovery interoperability

4.3.9 Bandwidth Consumption

The consumption of bandwidth is a prime concern within mobile networks which utilise low bandwidth communication mediums. Considering the active environment described in chapter 1 the consumption of bandwidth is also a major concern. The approaches to service discovery and agent systems explored in chapter 2 address bandwidth consumption by multiple techniques. The combining of tuple spaces and agents is a novel approach thus incorporating techniques from both areas is required. The use of the tuple space for resource, message and agent traffic and not for bulk data streams is inline with the conclusions by Wade et al [Wade'99] as outlined in chapter 3 when examining the L^2imbo approach. The tuple space is not best suited to

bulk communications such as multimedia streams. Instead it is more useful for the transmission of control information. MARE upholds this by offering methods for message passing between agents through the tuple space addressed by the identifiers associated with each agent. Further bulk communications is possible by allowing agents to utilise communications interfaces such as point to point connections, agent access permissions allowing.

4.3.10 Analysis

The MARE approach to resource discovery and configuration has been refined in this section by examining areas of importance when operating in the active environment explored in chapter 1 and summarised in Table 4-1.

Design Issue	Notes
Eval	Provides the fundamental ability for executable code to be moved through the tuple space and consumed by MARE instances, and hence facilitates integration of tuple spaces and agents.
Resource Advertising	Resources are advertised through the tuple space. Resources can be grouped on the same host to form resource bundles to reduce the number of connections required.
Resource Consistency	Resources are kept consistent through periodic announcements.
Agent Execution	Agents are executed at MARE instances of which there may be more than one on any given host.
Agent Movement	Agents are transported through the tuple space with related code and state to alleviate further interactions after the initial movement of the agent.
Agent Loading	Agents are checked for appropriate criteria before loading and in turn check the environment once loaded.
Agent Communications	Agent communications are performed through the tuple space by addressing a UID which can relate to a single or group of agents.
Interoperability	Agents are used to interact between different systems and act as a proxy for small or incompliant devices.
Bandwidth Consumption	Consumption is minimised for resource discovery through a proactive announcement of resources through the tuple space. Bundling of resources together reduces bandwidth and required connections.

Table 4-1 Key design issues

The key areas addressed within this section include tuple space, resource, agent and general environmental areas. The methods adopted by MARE to achieve awareness of available resources and maintain a consistent view of resources available is achieved through utilisation of the L^2imbo tuples space. The eval data type has been added enabling the movement of executable code through the tuple space. This has facilitated agent movement and enabled seamless reconstruction of agents for execution. Agents are moved through the tuple space and used for configuration and manipulation of resources. Agent issues addressed in this section include the *movement*, *loading*, *communications* and *execution*. Agents are moved with an initial descriptor and required code bases through the tuple space. The agent descriptor is read for suitability before being consumed by the MARE instance running at an appropriate security level. The tuple space provides a mechanism for communication between agents through the use of identifiers attached to the agents; however bulk communications are discouraged through the tuple space. The design points also address the interoperability with existing service discovery mechanisms and agent systems. This interoperability is viable through the use of agents however introduces the flaws in alternate systems that MARE is addressing in its approach.

4.4 Architecture

4.4.1 Overview

MARE has an architecture designed to be flexible allowing development of components in isolation to one another. The design issues previously mentioned in this chapter are able to be explored further within the separate components to help ease development and testing times.

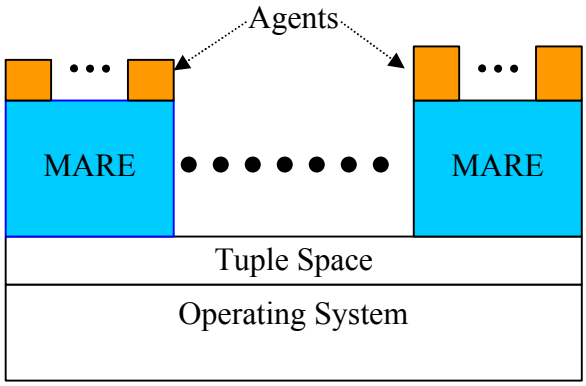


Figure 4-5 MARE host environment

The MARE approach requires each participating host to execute a tuple space stub and MARE runtime environment. This enables each host to have a tuple space for communications not only of resources and agents but for other application data that can utilise the tuple space approach. The position of the MARE system operating on a host can be seen in Figure 4-5, a further decomposition of the MARE system can be seen in Figure 4-6

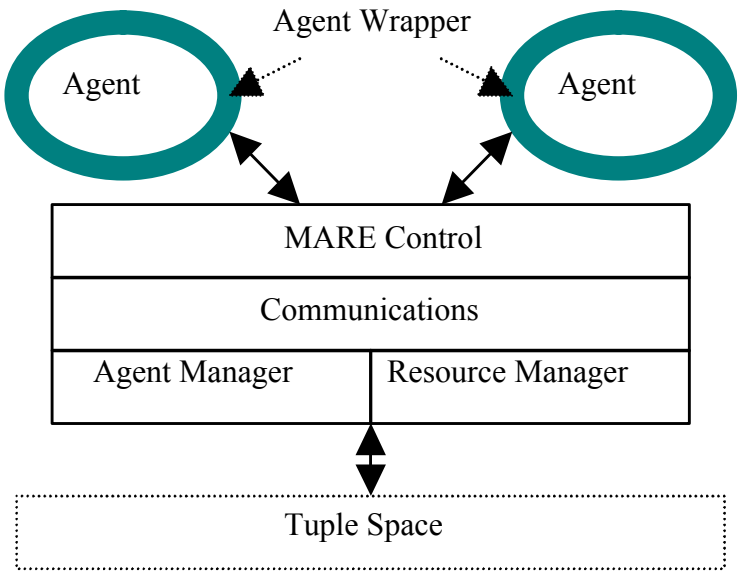


Figure 4-6 MARE architecture

The components have been designed such that they abstract over the surrounding components allowing for the development and introduction of new components at compile time if required. The abstraction can be seen as useful when trying different communications techniques for agents and resources. The components that make up the MARE architecture as shown in Figure 4-6 will now be defined in turn.

4.4.2 Key Components

Tuple Space

The tuple space implementation L^{2}imbo provides the transport mechanism for agents, resources and agent messages. Tuples of a specific format are used to define the type of transmission and constituent structure. A single tuple space instance can support multiple interactions; for example multiple MARE instances operating at different permission levels as well as other undefined applications that can utilise the tuple space system. The ability to serve multiple instances / applications from the same instance provides a local point for resolution of queries that can be satisfied by another local application.

Resource Manager

Resources and Agents are handled separately in the MARE architecture having associated managers as seen in Figure 4-6. The resource manager receives and

transmits information regarding resources, passing information up to executing agents and placing resource descriptions into the MARE environment. The manager is responsible for announcing the resource details from the MARE instance periodically. The manager is also responsible for the generation and transmission of resource bundles to conserve on the number of connections formed when transmitting resources discussed earlier in this chapter.

Agent Manager

The agent manager constantly listens for agents entering the tuple space. Upon the detection of an agent, its requirements are checked against the constraints of the MARE instance. If the agent requirements are acceptable, the manager acquires it and passes it to the execution environment. The agent requirements are explored in more detail in the implementation details in chapter 5. The agent manager can be set to not receive agents when carrying out insertion of resources, agents or shutting down to maintain consistency within the local system. The agent manager also handles the forwarding of agent messages between instances of the execution environment both local and remote, through the receiving and placing of messages into the tuple space and the forwarding of appropriate messages to agents in the local MARE instance.

Communication

Acting as an interface between the MARE control above and the resource managers below is the communication component. The communication component is specifically aimed at allowing different managers to be used or integrated beside or in place of existing managers. The component provides relaying of information from multiple communication sources for example the resource and agent managers in Figure 4-6 to the MARE control component.

MARE Control

The MARE control component examines agent requirements including the resources that the instance of MARE has, evaluating if an agent can be executed against them. The control component also delivers messages to the correct agents executing in the MARE instance. The control of the migration of agents is carried out at this point in the case of an event such as a system shutdown or changing of execution power caused by an event such as power saving.

Agent Wrapper

The agent wrapper is used to control the agents' operations and acts as an interface to the MARE control. An instance of this wrapper is passed to the agent upon being executed. This wrapper controls registering for receipt of messages such that more than one agent can receive the same message by addressing it to a shared identifier. In effect this component provides the interface to the underlying MARE system.

Agent

An agent implements a set of methods to allow the restarting and movement of the agent. The agent will be able to be serialized and serialize itself into a stream of data to be transported across a network. The agent will carry a set of requirements separate to the agent itself when transported such that a host examining the agent can examine the agent requirements without the overhead of reconstructing the agent and executing it if the agent is inappropriate. Allowing agents to have access to system libraries provides the agent programmer with freedom as to the operations of the agent restricted by the agent's permissions.

4.4.3 Analysis

The previously examined architecture outlines the components that make up the MARE approach to resource discovery and configuration. The movement of agents and resource information is maintained by the MARE resource and agent managers respectfully. When an agent is accepted it is examined for applicability to execute within the MARE environment before the agent is executed. When the agent is running it may also examine the MARE instance as to its suitability for the executing agent. Allowing both parties a say in weather the agent can execute at the current location.

4.5 Summary

The MARE approach has been designed to make use of the tuple space paradigm and mobile agents to facilitate a means of distributing and configuring resources. The key design issues examined within this chapter are targeted at providing a means of resource discovery and configuration within the active environment defined in Chapter 1. This chapter has explored the MARE approach regarding its potential efficiency when operating in an active environment through the use of tuple spaces

and mobile agents, and has outlined the MARE architecture and design decisions made when developing the MARE approach. The MARE prototype implementation is examined in the following chapter before evaluation of the MARE approach using the prototype.

Chapter 5

Implementation

5.1 Introduction

The prototype MARE implementation has been constructed using the design issues raised in the previous chapter. The prototype has been developed to address the requirements highlighted in chapter 2 for operating in an active environment. The prototype provides an approach that has no reliance on servers when performing resource discovery and configuration in an efficient manner. Usage of minimal bandwidth and being adaptable to environmental changes are also goals of the MARE prototype.

This chapter describes the implementation issues related to the components in the MARE architecture emphasising the techniques used within the MARE approach including weaknesses and strengths.

5.2 Implementation Language

The MARE system is required to operate on multiple devices being capable of migrating code with ease between MARE systems making the choice of implementation language restrictive. The Java programming language was chosen due to its widespread adoption on mobile devices, servers and desktop systems alike. Examination of upcoming languages such as the Microsoft Common Language Infrastructure (CLI) [Microsoft'01] may offer further choices. However at the time of writing little support is available for this option on UNIX derivates and non Microsoft powered handheld devices. MARE is written in the Java language. However components, such as agents written in other languages can be used but will lack the close integration with the MARE system. These components will require a level of abstraction typically through the use of a proxy between MARE and the agent of a different language.

5.3 MARE Structure

The structure of MARE has been introduced in Chapter 4 emphasizing the key components and their respective roles. This section looks in more detail at how these elements operate by examining the components in turn highlighting the prototype MARE implementation interfaces. The components have been designed in a modular fashion to aid in development of components for different or additional transport mediums such as incorporating support for bulk communications seamlessly. This ability is not explored in detail here except in overview. This chapter focuses on the use of agents and tuple spaces as the primary communications medium for Agents, resources and simple messages leaving directed communication to individual agent programmers to implement.

5.3.1 Tuple Space

MARE is required to operate in an active environment consisting of heterogeneous devices running different operating systems with different resources. The tuple space implementation MARE uses is L^2imbo this is written in ANSI C and ported to Microsoft platforms as well as UNIX derivates. The L^2imbo implementation provides C, C++ and Java Interfaces; the Java interface was used in the development of MARE.

There is only a requirement for a single instance of the L^{2}imbo stub; the instance can be accessed simultaneously by multiple instances of MARE or other applications as shown in Figure 5-1. L^{2}imbo provides methods for inserting and removing tuples as well as the ability for the MARE instance to register with the tuple space stub for the receipt of callbacks. The MARE implementation registers for the callbacks upon the arrival or deletion of specified tuple types namely agents, messages and resources.

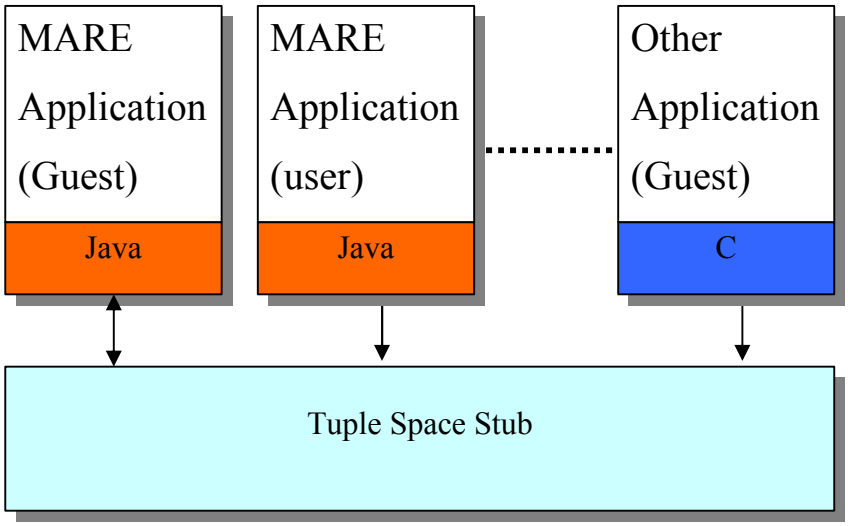


Figure 5-1 Tuple Space Stub with different interface languages and run levels

The API the MARE instance utilises consists of the Linda primitives, in, out, rd and out. Bulk primitives collect and copy collect are implemented as well as non blocking inp and rdp operations. Furthermore specific elements include the addition of callbacks on the arrival or removal of a specific tuple type as demonstrated in the simplified Resource manager code in Figure 5-2.

```
package MARE;

import java.io.*;
import java.util.*;
import TupleSpaceAPI.*;

class ResourceManager implements java.io.Serializable,
          java.lang.Runnable, TupleSpaceAPI.TupleCallback {
  //Tuple space handle
  private TupleSpace tupleSpaceHandle = null;
  //Handles for callbacks
  private int arrivedRegistration, deletedRegistration,

  //Callback methods
  public void TupleArrived(TupleSpace tupleSpace, Tuple newTuple)
          {
      System.out.println("Resource descriptor = " +
          ((StringArg)newTuple.getArgument(2)).toString());
  }
  public void TupleDeleted(TupleSpace tupleSpace, Tuple oldTuple)
          {
    System.out.println("Resource descriptor = " +
          ((StringArg)newTuple.getArgument(2)).toString())
  }

  public void run() {
    try {
      //initialise tuple space handle for UTS
      tupleSpaceHandle = new
          tupleSpaceStub().use(TupleSpace.UTSHANDLE);
      //Add tuple type & format information (Data & String types)
          tupleSpaceHandle.addTupleType("RESOURCE","DS");
      //Register for callbacks for the RESOURCE tuple
      arrivedRegistration = tupleSpaceHandle.register(this,
          TupleSpace.TUPLE_ARRIVED, "RESOURCE");
      deletedRegistration = tupleSpaceHandle.register(this,
          TupleSpace.TUPLE_DELETED, "RESOURCE");
    }
    catch (Exception e) {
      System.err.println("Error initialising tuple space.");
    }
  }
}
```

Figure 5-2 Registration for call backs in L^{2}imbo

The sample code in Figure 5-2 generates a handle to the Universal Tuple Space (UTS) used as a known tuple space for all hosts. New tuple spaces can then be generated to hold agents, messages and resource descriptions. This allows a MARE instance to subscribe to tuple spaces of interest avoiding filtering messages form tuple spaces it is

not interested in. The example highlights the subscription for callbacks on a specific type of tuple (RESOURCE) for deletion and insertion.

```
……
  if (tupleSpaceHandle != null) {
    Tuple t = new Tuple("RESOURCE");
    t.addArgument(new
          DataArg(resourceUID.getBytes(),resourceLength));
    t.addArgument(new StringArg(resourceDescriptor));
    tupleSpaceHandle.out(t);
  }
……
```

Figure 5-3 Inserting a tuple into the tuple space

Figure 5-3 illustrates the insertion of a tuple consisting of two arguments a data argument for the unique identifier and a string argument for the resource descriptor. In comparison the consumption of a tuple can be seen in Figure 5-4 where a blank tuple is created for the consumed tuple contents to be placed within. The call used in the consumption example is a blocking call however the same technique is used for non blocking operations such as rdp and inp.

```
……
  Tuple tuple = new Tuple("RESOURCE");
  tuple.addArgument(new DataArg());
  tuple.addArgument(new StringArg());
  tupleSpaceHandle.in(tuple);
……
```

Figure 5-4 Consuming a tuple from the tuple space

The agent manager acts as an interface between the communication layer and the tuple space. The manager offers the ability to detect the arrival and removal of agents and messages from the MARE system. The management of transmitting agents and messages from the MARE instance is also maintained by the agent manager component. This role requires the manager to know how to decode and encode both messages and agents.

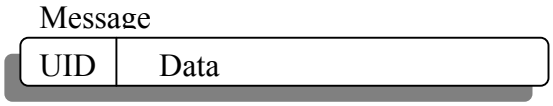


Figure 5-5 Message format

The message format shown in Figure 5-5 shows the target for the message identified by the targets, unique identifier (UID) gained from the MARE system followed by the message itself. The UID could refer to a single destination or a collection of recipients that share the same UID to enable group communications. An agent wishing to communicate with another agent uses the agent runtime handle passed to the agent when the agent becomes instantiated as demonstrated in Figure 5-6.

```
……
   agentRuntime.send(targetUID,message);
……
```

Figure 5-6 Sending a message

The agent format shown in Figure 5-7 shows the agent, the unique identifier of the agent and initial requirements for execution. The agent is transmitted with its classes such that the agent consists of a grouping of required java classes that are user generated with the agent class itself. The agent classes are extrapolated at runtime by examining the agent class hierarchy. Each new class is placed in a data stream for transmission with the agent by the MARE system.

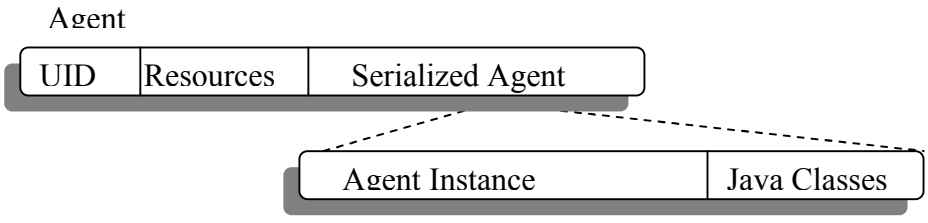


Figure 5-7 Agent format

The insertion of an agent with an attached resource list can be seen in Figure 5-8. The agent is assigned a unique identifier by the MARE runtime environment while the agent is prepared for transmission.

```
……
  agentruntime.send(resourcesRequired, agent);
……
```

Figure 5-8 Inserting an agent

The agent is serialised ready for transmission by the tuple space type added to the L^{2}imbo implementation whilst developing MARE called an *EvalArg*. The evaluation argument has facilities to take the agent and serialise it and all related classes automatically or just the agent class. In the prototype MARE implementation the required resources are assumed to be described as resource unique identifiers. The MARE approach is extensible where further definitions can be added such as a description of the environment and runtime level required. Furthermore the prototype only provides facilities for serialising the agent and all related classes at present, however an override of the transmission method can facilitate a different method of transmission.

5.3.1.1Eval

To facilitate the encoding of mobile agents the L^{2}imbo Tuple space implementation was extended to contain the eval data type. This enabled the transportation of executable code through the tuple space facilitating the movement of mobile agents as featured in the MARE approach. A mobile agent can consist of a single class. However multiple classes are often required by a mobile agent. A consuming node is required to reassemble and instantiate the agent at the recipient host. Three approaches are highlighted in section 4.3.1, the movement of just the serialised agent which relies on all required classes being present on the consuming node. The movement of all classes required to reassemble the agent identified by examining the agent being serialised. The third approach is to move the serialised agent which gathers the required classes when being reconstructed from the tuple space or other storage areas. MARE by default takes all the required classes with the agent. The state of the agent is captured through Java serialisation and placed in the eval field. If required the agents associated classes are identified through examining the serialised agent. The classes are then collected in a binary format from the executing system ready to be distributed. Upon reconstruction of the agent a Java class loader is used to

identify and load the required classes from the appropriate store allowing the agent to execute.

5.3.2 Resource Manager

Performing a similar role as the agent manager, the resource manager acts as an interface between the communication layer and the tuple space. The resource manager offers the ability to detect the arrival of new resources and removal of resources passing on any new resource descriptions as well as any notifications of removed resources to the upper layers for relaying to the executing agents on the MARE instance. This role requires the manager to know how to decode and encode both messages and agents.

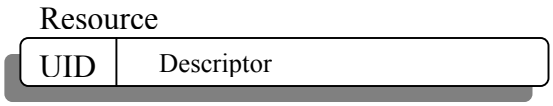


Figure 5-9 Resource structure

The resource format shown in Figure 5-9 shows the unique identifier of the resource followed by the resource description that is constructed by a resource and advertised by a MARE instance. The descriptor is a series of key-value pairs as shown in Figure 5-10; the keys are separated by a semicolon and passed as an ASCII string. The descriptor contains a few system pairs such as source address. The user defined section is checked for syntactical correctness prior to transmission. However interpretation of the content is left to the consuming MARE instance.

```
Descriptor -> Pair | Pair;Descriptor
Pair       -> Key = value
Key        -> DESCRIPTION | TYPE | IP | PORT | user-defined
```

Figure 5-10 Resource descriptor

If the MARE instance has more than one host the resource may be part of a *resource bundle* whereby the resource is grouped with other resources to reduce the number of separate connections required to transmit all the resources held on a host. This can be seen in Figure 5-11. The resources can be compressed to save transmission bandwidth

however this requires greater processing at both source and destination hosts to compress and decompress the resource descriptors.

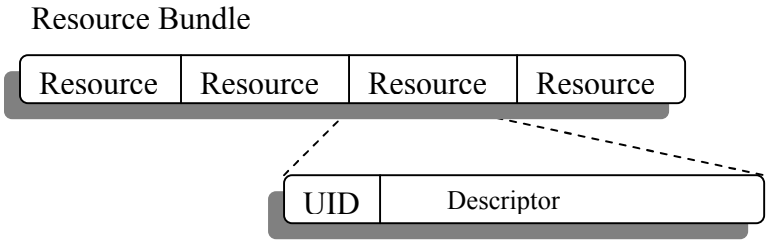


Figure 5-11 Resource bundle

Resources can be added to the resource manager through calls from an agent or from an outside source by placing the resource description within the MARE system by a static call to the MARE instance as shown in Figure 5-12.

```
……..
AgentRuntimeEnvironment.addResource(resourceDescriptor);
……
```

Figure 5-12 Static call to insert a resource descriptor

A resource is assigned a UID when it is added to the MARE system upon the generation of a valid descriptor. To aid in the correct construction of a resource descriptor a helper class is part of the MARE system. The helper class shown in Figure 5-13 consists of addition and removal of individual items as well as the ability to take a transform into and out of a string representation of the resource descriptor.

```
public class ResourceDescriptor
{
  // Create an empty descriptor.
  public ResourceDescriptor();
  // Construct a resource descriptor from a compliant string
  public ResourceDescriptor(String descriptor);

  // Generate a string from the resource descriptor
  public String toString();
  // Get descriptor as a string
  public String getDescriptor();
  // Get a specific value from
  public String getValue(String key);
  // Remove an item from the resource descriptor
  public void removeDescriptor(String key);
  //Add an item to the resource descriptor
  public void addDescriptor(String key, String value);
  //Get a hastable copy of the resource descriptor
  public Hashtable getPairs();
}
```

Figure 5-13 Resource descriptor helper class

The current approach to resource descriptors within MARE is a simple key value pair approach. The approach would benefit from a more generic approach to describing resources for interaction, such an approach may utilise an XML descriptive language. This thesis believes that the currently used key value pair approach is adequate for testing purposes however admits this is an area that can benefit from further examination.

The resource manager is responsible for maintaining the resource's availability and placing into the tuple space the resource the host contains periodically. Each resource in the resource manager has an associated *lease time* based on time of arrival of a resource. The lease time is set to be longer than the expected next beacon of information. This allows for the beacon to refresh the timer. If a beacon is not received during this period the resource will timeout and be removed from the active list. The resource manager, upon detecting the failure of an update, passes a notification to the MARE control of the change in resource availability and this will in turn alert interested agents. It should be noted that ultimately a programmer must allow for a resource being reported as present by the system and being unavailable at the time of use. Such availability issues are due to the nature of the changing active environment; a system cannot guarantee the availability of a resource within such an environment without keeping an open channel to the resource and getting immediate

notification of communication failure. This would consume valuable resources, which is clearly undesirable.

The L^{2}imbo tuple space approach has no notion of an expiry time of a tuple that can be user specified. Requesting the removal of a tuple from the tuple space generates a message transmitted separately or piggybacked on a tuple announcement. The MARE approach will place a deletion of a resource or resource bundle on the corresponding next announcement. Due to precedence in transmission in the L^{2}imbo tuple space approach the tuple space maintains a consistent view of available resources with a minor increase in tuple size and no extra connections are generated.

5.3.3 Communications

The communications component is positioned between the lower agent and resource managers and the MARE control component. The component provides a separation between control and data layers in the MARE implementation. The component provides a small set of interfaces such that new agent and resource managers as well as control components can be generated with ease. The development of different managers can be achieved when interaction with other communications mediums with differing characteristics. These can be used in place of, or alongside the tuple space aware implementation used in the MARE prototype.

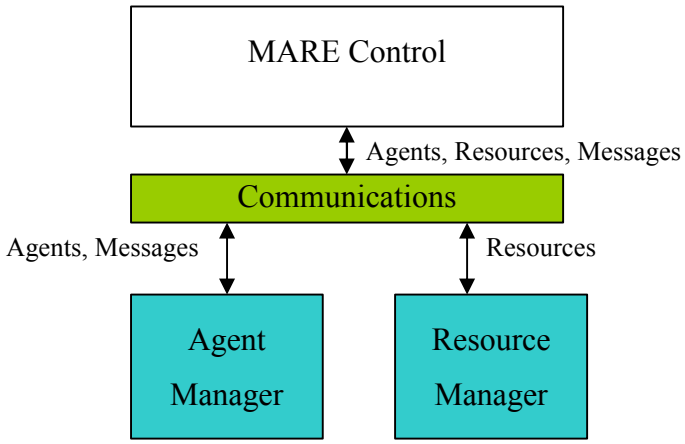


Figure 5-14 Communications component control routes

The communications component shown in Figure 5-14 is positioned between the resource and agent managers and the MARE control component. This component directs the specific request from the MARE control to the required manager as well as

receiving information presenting a uniform interface to the MARE control component.

5.3.4 MARE Control

The MARE control component is at the heart of the prototype MARE implementation providing checks for incoming agents, instantiation of agents and destruction of agents executing upon the MARE instance.

The component is passed a new agent by the communication layer to decide if it is appropriate to execute the agent through examining the resource requirements. If the resources are appropriate for the MARE instance, an agent wrapper is generated and the agent is constructed and started in a new thread inside the agent wrapper as shown in Figure 5-15. At this point a Java security profile [Sun'02b] can be applied to the incoming agent inside the Java framework offering a finer level of security checks on the constituent classes used by the agent.

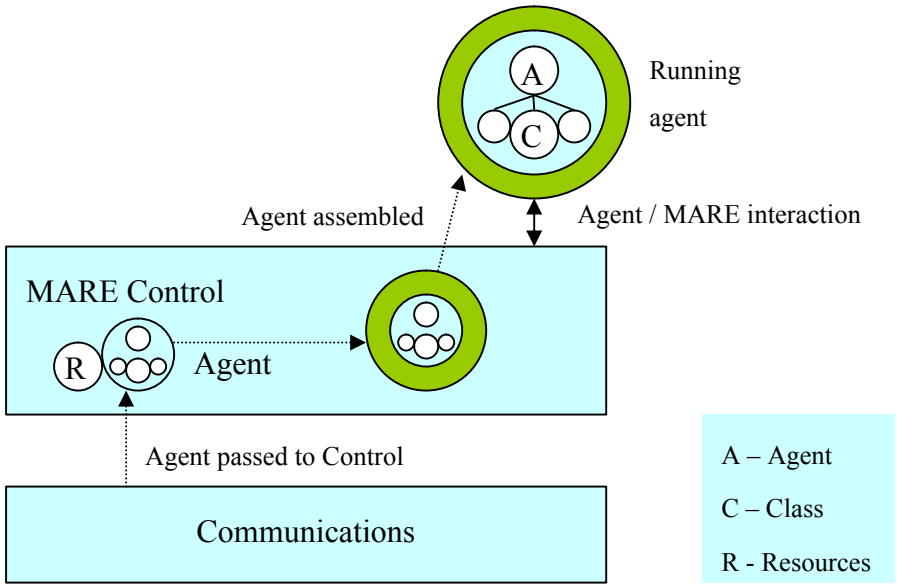


Figure 5-15 Agent decompressed and executed

It is the task of the control component to monitor the environment the MARE instance is executing in to decide if an agent is operating incorrectly and if so terminates the offending agent. Whilst the detection of an inappropriately operating agent is currently non trivial within the Java framework external factors can be detected such as power status, connectivity and physical position. These can then be reported to the

MARE instance through a local tuple space. The use of a local tuple space for a transport mechanism for system information generated by external components provides information of use to the MARE instance. Indeed other MARE instances can place information in a local tuple space allowing interaction between MARE instances in an efficient manner requiring no external communications.

5.3.5 Agent Wrapper

The agent wrapper is generated by the MARE control component for the execution of an agent. In essence the serialised incoming agent is finally constructed after being accepted within the environment inside an Agent wrapper, the wrapper provides the interfaces for interaction with the MARE control component and thus the MARE implementation.

5.3.6 Agents

A MARE agent is a class that uses an interface that provides a set of functions for interaction between the MARE instance and the executing agent. The interface also inherits from a class defined in the tuple space designed to enable the serialisation and de-serialisation of the class inclusive of its component parts such as other defined classes.

```
/**
 * Base interface for generating an agent
 *
 * @author Matthew Storey
 * @version 1.1
 * @see java.lang.Runnable
 */
package MARE;
import java.io.*;
import TupleSpaceAPI.Eval;

public interface Agent extends Eval
{
  /*
   * Called to start / restart an Agent
   */
  public void run();

  /*
   * Called by MARE when arriving at a new Location to
   * provide an interface to the agent runtime environment
   * (access the agent wrapper)
   *
   * @param agentRuntime The agent runtime environment.
   * This variable may be used to interact with the local
   * runtime environment
   * @see MARE.AgentRuntime
   */
  public void agentEnvironment(AgentRuntime agentRuntime) ;

  /*
   * Called by the Agent Environment to request the agent
   * moves.
   */
  public void close() ;

  /*
   * Called by the agent runtime environment to let the
   * agent know of incoming data.
   * The agent may register for data from multiple sources
   * via calls to the agent runtime environment
   *
   * @param destination The destination of the data
   * @param data The data for the given destination
   * @see MARE.AgentRuntime
   * @see MARE.UID
   * @see java.lang.byte
   */
  public void receive(UID destination, byte[] data) ;
}
```

Figure 5-16 Agent interface

The Agent interface shown in Figure 5-16 defines a *run* method for starting or resuming an agent and *agentEnvironment* a method for passing a handle to the MARE interface to the agent. A communications method, *receive* is supplied to allow the MARE instance to direct communications to the executing agent. The *close* method is

provided to allow the agent to be requested to serialise itself and migrate itself to another host if required.

Within the standard release of Java the serialisation of an agent with its runtime environment and associated state is not possible. In the MARE approach the calling of the *close* method by the runtime acts as a trigger for serialisation allowing the programmer to pause the execution of the agent and serialisation the agent. This approach allows the programmer to decide which parts of the agent are required to be moved and which can be discarded before moving to a new location. This enables a higher level of control over the manner in which an agent is serialised and typically a smaller footprint for the transmitted agent as only the required information needs to be moved rather than the entire agent runtime. An agent can be forcibly terminated through the Java threading classes. However this does not offer a clean way for state to be preserved for restarting at a different location. The ability to stop the execution of an agent and restart from the same execution point is not possible without a means of stopping execution and storing state in the Java runtime environment.

An agent can be placed in the MARE system from both within an executing MARE instance and from outside an executing MARE instance. An agent can be generated by another agent with appropriate permissions and placed into the MARE system. Alternatively an agent may be generated by an external application and placed into the MARE system by utilising calls available in the MARE implementation. Inserting an agent from outside an executing MARE instance effectively places a serialised agent inside the tuple space for consumption by an executing MARE instance.

Figure 5-17 shows an example simple agent that once placed into the MARE system will be consumed and executed displaying a message in all of its states, namely running, being closed and receiving a message. The handle to the agent runtime is passed to the agent once it has been consumed by a host and then the run method is called which displays its message. If a message arrived addressed to the agent's UID the agent would be notified through the receive method and if the agent did more work the close method may be called which would display an appropriate message.

```
/**
 * Test agent for displaying status on executing host.
 *
 * @author Matthew Storey
 * @version 1.0
 * @see MARE.Agent
 * @see java.io.Serializable
 */
import MARE.*;

class TestAgent implements MARE.Agent,java.io.Serializable {
  /* Handle to agent runtime*/
  private AgentRuntime  agentEnvironment = null;

  public void agentEnvironment(AgentRuntime agentRuntime) {
          agentEnvironment = agentRuntime;
  }
  public void close() {
    System.err.println("Request to go");
  }
  public void receive(UID destination, byte data[]) {
    System.err.println("data arrived");
  }
  public void run() {
    System.err.println("I have arrived");
  }
}
```

Figure 5-17 Agent to display status

The insertion of the agent into the MARE system can be achieved without having to initialise a MARE environment by inserting the agent directly into the MARE system as shown in Figure 5-18. This example shows the insertion of the agent with no defined prerequisites such as other resources.

```
new AgentRuntime.addAgent("",new TestAgent());
```

Figure 5-18 Insertion of an agent into the MARE system

The agents are given a high level of freedom; once assembled the agent is restricted only by the security permissions applied to the agent through a security manifest or through the maximum permissions available to the MARE instance the agent is running in. In effect this approach is providing the agent with all the permissions associated with the environment the MARE instance is executing within. The ability

to have multiple MARE instances on the same machine allows multiple instances in different security levels.

5.4 Summary

The prototype implementation described within this chapter has been described by examining each component of the architecture in turn. The operations each component performs and the implementation issues associated with each component have been examined. The generation of resource descriptions and agents have been shown through an examination of code examples from the MARE system. This chapter has explored the MARE prototype implementation, the validity of this approach will be further examined in the following evaluation chapter.

Chapter 6

Evaluation

6.1 Overview

The previous two chapters have proposed the MARE design and described the MARE prototype implementation. The majority of this chapter focuses on performing a qualitative evaluation of the MARE prototype. The qualitative examination explores resource discovery through the use of the L^2imbo tuple space implementation and resource configuration through the use of mobile agents. A quantitative evaluation is also performed on the underlying resource discovery mechanism exploring the efficiency of the MARE approach.

This evaluation aims to explore the hypothesis of this thesis that utilising tuple spaces and mobile agents in an active environment enables resource discovery and configuration to be achieved. The evaluation examines the core requirements of the MARE approach exploring the validity of the approach described in chapter 4 for operating within an active environment. To facilitate the evaluation the emergency multimedia example from chapter 1 is further explored and refined. Finally the requirements detailed in chapter 2 will be revisited at the end of this chapter aiding in summarising the effectiveness of the MARE approach.

6.2 Case Study: Emergency Multimedia

The emergency multimedia example described in chapter 1 is referred to and used throughout the evaluation performed within this chapter. The example explored is of a rescue performed in the British Lake District where three rescuers converge upon a stricken party. The rescuers and injured party desire knowledge of each other such as resources they hold and can be utilised aiding in the rescue. The scenario is summarised in Figure 6-1 illustrating four nodes each with a resource to advertise and desiring knowledge of the remaining three hosts.

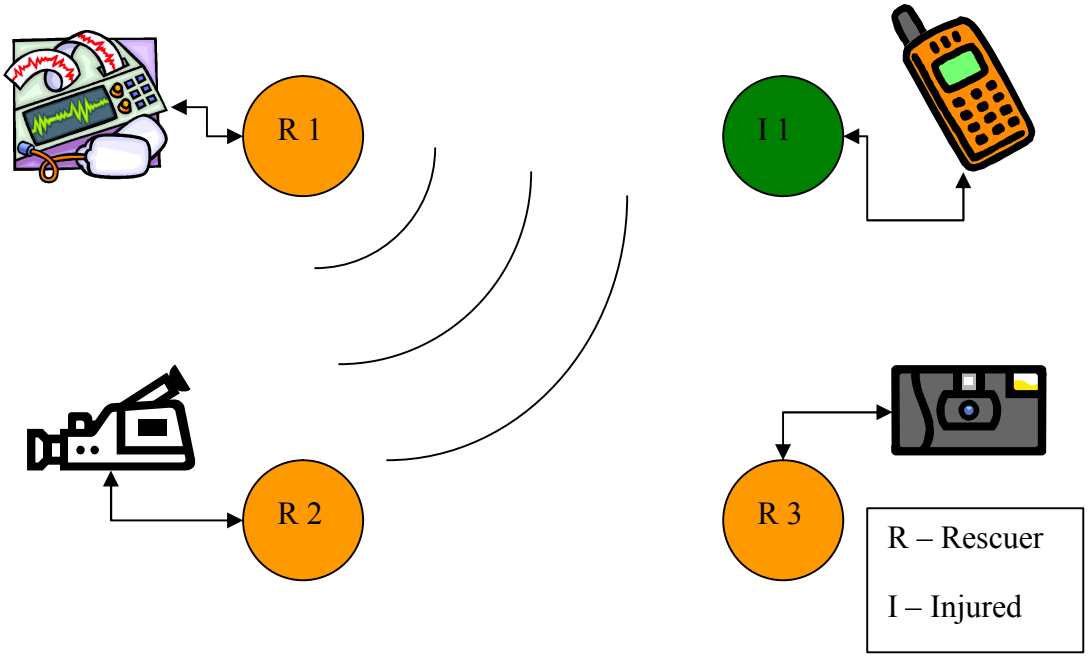


Figure 6-1 Scenario illustration

Expanding the illustration in Figure 6-1 further, the distressed party has a long range communication device equipped with Bluetooth; an example of such a device is a modern mobile phone. This device can be used initially to raise awareness of the incident through a phone call and to act as a resource that can be utilised as a long range communication device.

Each rescuer is equipped with a TETRA handset, a small embedded device with Bluetooth and a connected device. Rescuer one has a medical monitor that interfaces to the embedded device. Rescuers two and three also have embedded devices that are connected to a digital video recorder and camera respectfully.

Note that although this chapter focuses on the emergency multimedia scenario, many other scenarios can be envisaged exhibiting the attributes of an active environment. Examples demonstrating a grouping with a dynamic membership exhibiting a lack of communications infrastructure, requiring knowledge of surrounding resources and facilities to effectively utilise these resources. Such examples can be seen in groupings of communication capable devices such as office meetings, social gatherings, vehicles stationary or moving. In all of these examples interaction can lead to the discovery of new solutions to problems or enhancement of the situation, such as in car navigation, general sharing of information, and utilisation of surrounding audio video equipment.

6.3 Multimedia Enhanced Rescue Demonstrator

This section explores the construction of the emergency multimedia application as a demonstrator for the MARE approach. The developed prototype has a user interface device that can be powered on when required. To enable continuous advertising of resources a rescuer also carries a small embedded device providing a point of computation for mobile agents. A picture of the embedded device and user interface is shown in Figure 6-2 removed from the backpack and clothing of the rescuer.

Figure 6-2 Emergency rescue devices

The separate user interface and embedded device has led to the development of a small user interface application and a mobile agent that acts on its behalf. The agent collects information about available resources whilst executing on the embedded device. The agent relays information to the user interface device for collection when the handheld device is powered on; the link between the handheld and embedded device is assumed to be available at all times.

To enable the advertising of resources and execution of mobile agents upon powering up the embedded device it executes the L^{2}imbo stub and starts a MARE instance. The registration of any resources is performed as well as the insertion of the mobile agent acting on behalf of the user interface. The code to start the MARE runtime and insert some resources can be seen in Figure 6-3 illustrating the initialising of a MARE instance, the insertion of an agent and the insertion of some resources.

```
import MARE.*;

public class Go {

  public static void main(String args[]) throws Exception
  {
    //Initialise Agent Runtime Environment to accept agents
          without a graphical environment.
    AgentRuntimeEnvironment.Go(true,false);
    //Insert Mobile Agent into running environment
    AgentRuntimeEnvironment.addAgent("",new
          EmergencyRescueAgent());
    //Add resources (Still Camera, Digital Camera, GPS)
    AgentRuntimeEnvironment.addResource(new
          ResourceDescriptor("DESCRIPTION=Still Camera
          320x240;TYPE=VIDEO SOURCE;IP=10.0.10.10;PORT=3010"));
    AgentRuntimeEnvironment.addResource(new
          ResourceDescriptor("DESCRIPTION=Digital Camera
          320x240;TYPE=VIDEO SOURCE;IP=10.0.10.10;PORT=3020"));
    AgentRuntimeEnvironment.addResource(new
          ResourceDescriptor("DESCRIPTION=GPS for rescuer
          3;TYPE=GPS;IP=10.0.10.10;PORT=3030"));
  }
}
```

Figure 6-3 Starting MARE and inserting resources

The MARE instance executing on the embedded device listens for resource announcements from other devices whilst announcing resources it hosts. The mobile agent executing in the MARE instance monitors available resources passing this

information to the user interface device. The associated code for this operation can be seen in Figure 6-4.

```
import MARE.*;

public class EmergencyRescueAgent implements MARE.Agent,
          MARE.ResourceCallback {
  private AgentRuntime agentRuntime = null;
  private java.util.Hashtable resources = null;

  public void agentEnvironment(AgentRuntime agentRuntime) {
    this.agentRuntime = agentRuntime;
    //initialise temporary available resource list.
    resources = new java.util.Hashtable();
    //register for resource callback
    agentRuntime.registerCallBack(true, this);
  }
  //Called when agent is about to close.
  public void close() {
    //de-register for callback before move.
    agentRuntime.registerCallBack(false, this);
    //Reset the variables (saves space when serialising agent)
    agentRuntime = null;
    resources = null;
  }

  //A new resource has arrived
  public void resourceArrived(UID resourceUID, String
          resourceDescriptor) {
    //Add resource to our list
    resources.put(resourceUID, resourceDescriptor);
  }

  public void resourceRemoved(UID resourceUID, String
          resourceDescriptor) {
    resources.remove(resourceUID);
  }

  public void run() {
    //transmit resources to userinterface.
    ……
  }
}
```

Figure 6-4 Emergency rescue agent

The user interface application is started on the handheld device for receiving the resource descriptions. Connection with resources and the interpretation of resource information is performed on the user interface device allowing direct connection with available resources. An illustration of the process can be seen in Figure 6-5 demonstrating two embedded devices with a mobile agent listening for available resources passing this information to the user interface device. The user interface

device can then communicate with the resource directly, in the illustration a digital still camera. Further code listings for the mobile agent and user interface application are available in the appendices of this thesis.

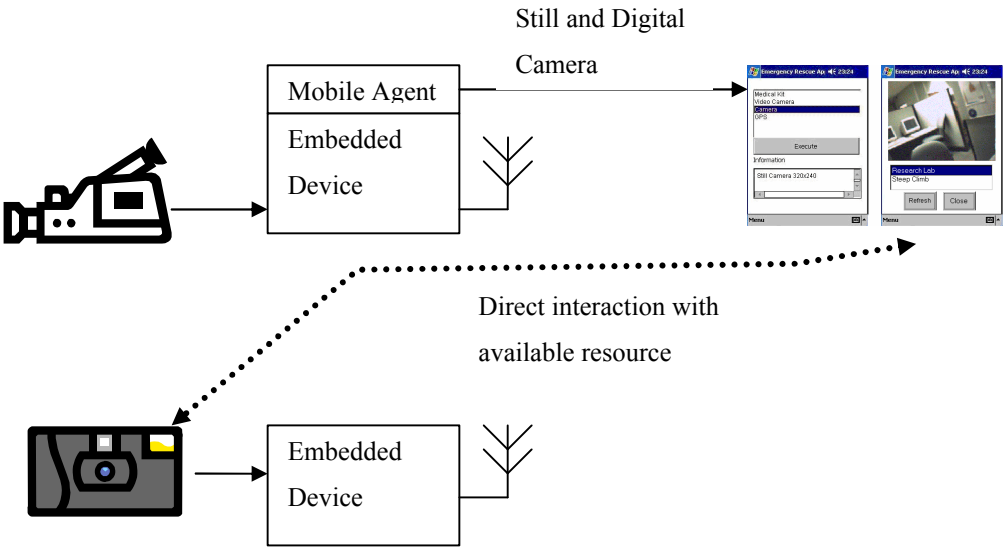


Figure 6-5 Emergency rescue process

The user interface application can be seen clearly in Figure 6-6 where a single resource is available. The agent running on the embedded device in the MARE environment reports changes in available resources such as arrival and departure of resources. The user interface device can power down whilst the agent acts on its behalf relaying information to the user interface device when required.

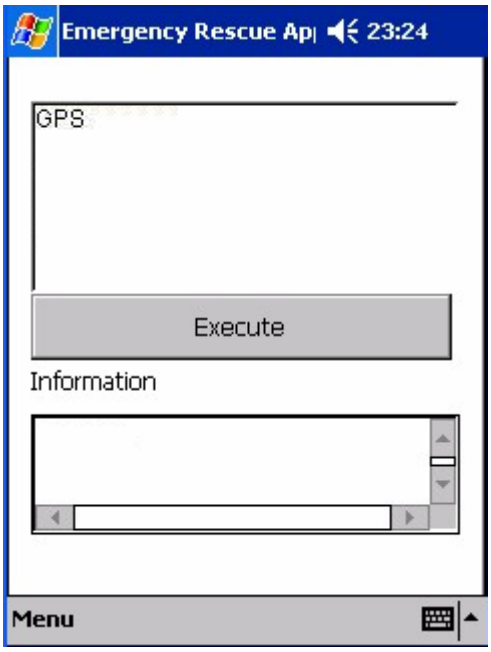


Figure 6-6 Initial resource monitor

Upon selecting an available resource it can then be utilised as shown in Figure 6-6. Initially the application shows only the rescuers device but upon being in range of another rescuer or the distressed party the application will show all available resources as can be seen in Figure 6-7.

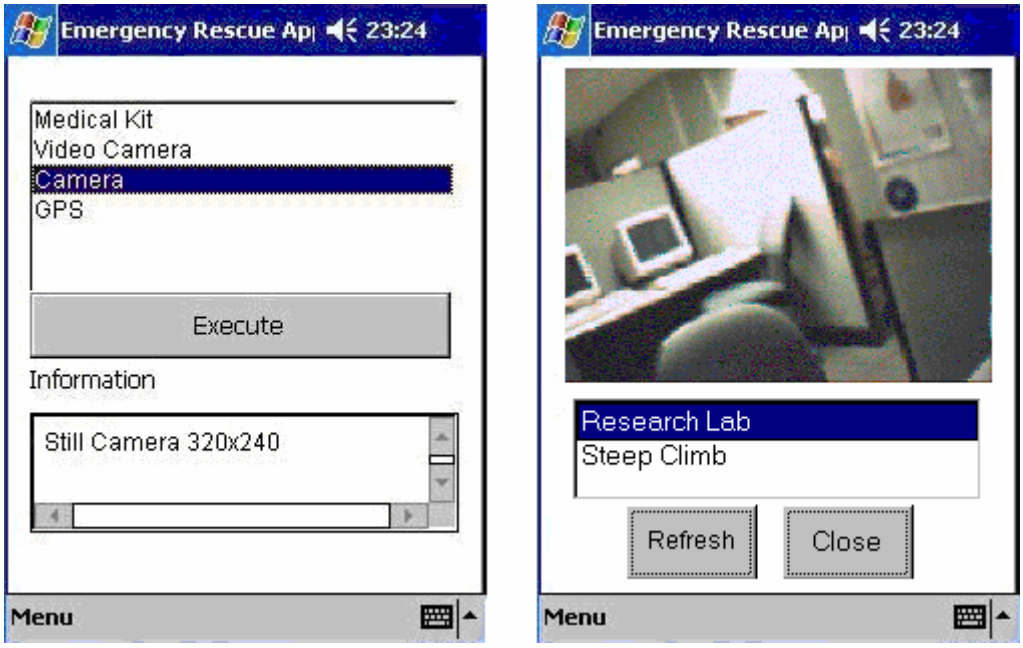


Figure 6-7 Rescuer view with multiple resources and resource in use

The construction of the emergency rescue demonstrator influences the following qualitative and quantitative discussion of the MARE approach.

6.4 Qualitative Evaluation

6.4.1 Resource discovery

Being able to discover what resources are available is essential in making efficient use of surrounding facilities. An example can be seen when a vehicle accident has taken place, a passer by is searching for a communications device to call for emergency support. The device may be integrated and hidden in some way making discovery of the device unlikely unless it can be automatically discovered and utilised. This section evaluates MARE with key issues when performing resource discovery, namely the method of advertising an available resource, the maintaining of consistency and the bandwidth consumed performing the discovery.

Resource Advertising

Resource advertising in MARE is performed by utilising the L^2imbo tuple space implementation to perform the distribution of resource descriptions between interested parties. The MARE approach is proactive as it transmits periodically an announcement of any resources a host has available. Approaches such as UPnP, Jini and SLP work on a more synchronous approach of one or more requests followed by one or more responses. These approaches make initial use of servers to act as repositories of resources that can be interacted with directly instead of querying all available hosts. The use of servers is usually advantageous in a static environment where hosts and services are typically fixed or change infrequently. In a mobile environment resources change more frequently and servers are less likely to be available. In an active environment hosts and resources are often transiently available making reliance on servers inappropriate. The active environment forces host to host discovery, making a different approach to resource discovery desirable to address the lack of infrastructure and constantly changing environment. In summary the use of the L^2imbo tuple space implementation in the MARE approach allows advertising of resources to be performed in an ad hoc network.

Resource Consistency

MARE achieves consistency through each host periodically advertising its resources. Every listening host that can hear the announcement can be assumed to be able to communicate with the resource. Whilst the ability to communicate with the advertised resource can be assumed, it should be noted that the host could have moved out of range after the announcement so caution should still be exercised in utilising the resource. In UPnP, SLP and Jini the use of central servers can lead to a resource being advertised as available however being out of communications reach of a host as shown in Figure 6-8. The server can see the client providing it with an announcement however the client cannot see the resource it need to interact with.

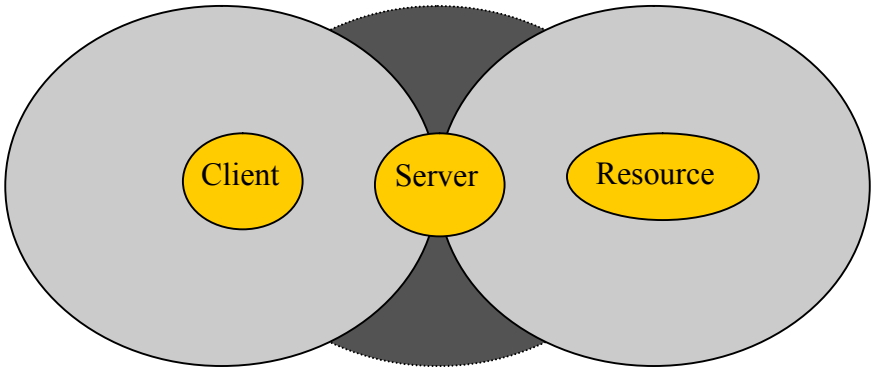


Figure 6-8 Resource communication range

When UPnP, SLP and Jini are operating in non server based mode maintaining a consistent view of resources is possible through a request response mechanism. This approach allows consistency when required however at the cost of extra bandwidth consumption. This will be further explored in the quantitative evaluation later in this chapter. MARE continually maintains a view of all available resources whereas UPnP, SLP and Jini when operating in a non server mode rely on client requests to update the available resources to a given host. The MARE approach is also proactive in detecting a change in resource state reporting it back to executing agents. UPnP and SLP require client actions unless a planned and advertised removal is performed whereby the removed resource advertises its state change. In summary MARE offers a simple approach to resource discovery where resources are advertised and timeout such that a view of available resources is achievable with no interaction from a client.

Bandwidth Consumption

MARE is designed for operation in an active environment where bandwidth is currently a precious resource. MARE can offer a deterministic model of bandwidth consumed performing discovery and consistency (this is further explored later in this chapter). The approaches offered by UPnP, SLP and Jini provide less predictable behaviour; the possibility of operation with one or more servers and requests upon demand for resource descriptions emphasize this. There is potential for less bandwidth consumption when performing resource discovery using available servers in a static network. However the active environment is constantly changing making existing server based approaches inappropriate. In the absence of server, UPnP and SLP approaches utilise peer lookup techniques generating extra messages and thus consuming more bandwidth. In contrast MARE adopts a combining of resource descriptions into a resource bundle reducing the number of individual messages. Furthermore MARE seamlessly supports compression of resource descriptions saving bandwidth at the price of some computation at the end points of the communication.

Usage

MARE is targeted at performing not only resource discovery but also configuration. The management of a resource is performed within the MARE system. A resource is constructed using a helper class called *ResourceDescriptor* allowing a convenient method for construction, encoding and decoding of resource descriptions Figure 6-8 demonstrates the generation of the still camera as shown in Figure 6-7. The resource is inserted into an available MARE system either on the device advertising the resource or a host executing the MARE system. In the case of the emergency rescue the embedded device carried by the rescuer can be used. A resource descriptor is generated and required key value pairs added prior to a static call to insert the resource in to a running MARE instance. A unique identifier is returned for identifying the resource for removal at a later date from the hosting MARE system.

```
Public static void main(String [] args) {
  //Generate a Digital Camera resource description
  ResourceDescriptor rd = new ResourceDescriptor();

  rd.addDescriptor("DESCRIPTION", "Still Camera 320x240");
  rd.addDescriptor("TYPE", "VIDEO SOURCE");
  rd.addDescriptor("IP", "10.0.10.10");
  rd.addDescriptor("PORT", "3002");

  //Add a resource to the MARE system (static call)
  UID uid = AgentRuntimeEnvironment.addResource(rd);
  ……
  //Remove resource from the MARE system
  AentRuntimeEnvironment.removeResource(uid);
}
```

Figure 6-9 Resource descriptor generation

A resource can be advertised by placing it into a MARE instance from within an agent or from a static call. Once a resource is added to a MARE system it exists until it is explicitly removed from the MARE system. The MARE instance advertises the resource with other resources the system may be responsible for with no further user interaction.

6.4.2 Mobile Agents

MARE provides the ability for resources to be configured by utilising mobile agent techniques. The agent approach adopted in MARE can be utilised for configuration of resources as well as user defined operations. This section will examine mobile agent operations in a qualitative manner in comparison to existing key mobile agent approaches.

Agent Generation

Mobile agents are generated and placed into an environment where they migrate to an appropriate position to be executed. Mobile agent approaches such as Mole and Concordia utilise an inherited class for routines key to interaction between the agent and their runtime environment. In these approaches implemented methods include initialisation, execution routine and stop routines. MARE also provides a class that is implemented by an agent offering initialise, start, stop methods as well as a method that is called by the runtime environment providing a handle to the MARE instance allowing manipulation of the runtime for operations including registering for receipt of message and system events.

Agent Migration

Enabling an agent to migrate to an appropriate position in an environment is a complex operation. An agent requires a method of storing itself or being stored in a manner which can be transported to a new location. Further to this the agent may require the carrying of some state between execution points. An agent also needs a target destination for migration, this can be directed, however the lack of a reliable structure in an active environment makes this impractical.

The storage of an agent for migration can be achieved through the movement of source code for interpretation or compilation on the destination machine. This can be seen in TACOMA and pre-compiled binary or intermediary language as used by Java or Microsoft IL solutions. Java is used in Mole, Concordia and MARE allowing the serialisation of code into a byte stream capable of movement to a new host. The Java approach allows current variable state to be serialised as well as related classes to be included where necessary for migration. Several approaches are adopted in mobile agent systems including the migration of the core agent class allowing the requesting of further required classes from a related host or repository of classes. Both of these approaches are useful in conserving resources where there is a high probability that classes are already available on an executing node. If the classes are not already present then further fetching of classes will be required resulting in further connections being formed and dependences made upon a specific host being present. MARE moves the entire agent by default including all related classes, (explored further below). This approach generates a potentially larger initial agent but removes the reliance on other hosts as required for operation in an active environment.

The storage of state is typically not possible due to the complexity of moving the current execution stack and any external links the agent may be utilising. The lack of support for runtime state migration in languages such as Java leads to a technique often described as ‘check pointing’. Check pointing is where an agent to be migrated is notified that it will be migrated allowing it the opportunity to store its state in some way prior to movement. Concordia, Mole and MARE all employ this approach.

The migration of agents is typically achieved through a targeted approach, such as discovering a host or having a predetermined host to move an agent to; this can be

seen in approaches such as Concordia, aglets and TACOMA. This approach is not appropriate in an active environment where a host cannot be relied upon to be available. MARE uses a novel approach utilising the tuple space to distribute the agent to any available host that can execute the agent. A list of requirements is transmitted with the agent such as required available resources which helps determine an appropriate host. The tuple space *eval* data type is used to transport the mobile agent through the tuple space with associated data representing the resources required for the agent to execute.

MARE incorporates features from other mobile agent systems extending and modifying them for use primarily in the active environment. The default transmission of agents including all associated state and classes required for execution to any available node allows for periods of disconnection and an opportunistic approach to execution. MARE allows extensibility of this approach through the utilisation of the tuple space to maintain classes and state for a migrating agent to be collected upon execution of the agent allowing a reduction in the transmission size of the agent. Other uses of the tuple space include the transportation of events and messages that can be placed in and consumed from the tuple space at any participating host at any time.

Agent Execution

The execution of a mobile agent requires a successful migration of the agent and its constituent components before an agent can be consumed. Mobile agent systems such as Mole and Concordia emphasize protecting the runtime from actions performed by the agents. In contrast MARE aims to consider both the agent and the runtime when operating. The consideration of both parties in execution helps prevent damage to the agent or runtime environment. The MARE environment checks the agent when it is consumed into an instance and likewise, once the agent is executed the agent can check the instance for its applicability to run. If an instance or agent is not appropriate, the agent can place itself or be placed back into the environment for consumption at another host.

Usage

The use of mobile agents for performing operations in a mobile environment requires a different style of programming to the standalone application model. Mobile agent programming when applied to the active environment is well suited to applications that can be broken into distinct components, or require specific resources that may not be readily available. A developer is required to think in terms of what is required for the application to execute and how an application can be decomposed into sub components. While applications can still be monolithic and simply moved to a more appropriate location for execution, there are benefits from decomposing a problem, e.g. distributing an application to increase processing power or implementing redundancy. Decomposing an application does impose extra overhead on the developer in terms of code generation for resource specification, starting and stopping an agent as shown in Figure 6-4. The extra effort is rewarded by the potential for a more optimal use of available resources such as bandwidth and processing.

Agent programming should be as intuitive to an application programmer as any other coding techniques they may be familiar with. To facilitate this MARE only requires the agent to support a single class interface implementing the methods listed within it. The agent can be extended to receive resource state callbacks through further interfaces, as shown in Figure 6-10. The code example is a simplified version of the agent run by each rescuer upon the embedded device they carry to collate resource views and report back to the user interface client. The agent maintains a separate thread for updating the user interface with the resource view when required.

```
class MyAgent implements Agent, ResourceCallback {
//Agent interface methods
……
//Resource Callback methods
public void resourceArrived(UID resourceUID,String
          resourceDescriptor) {
  //Update resource view
  }
public void resourceRemoved(UID resourceUID,String
          resourceDescriptor) {
  //Update resource view;
  }
}
```

Figure 6-10 Agent with resource availability call-back

The agent can be injected through a call from within the tuple space or by an executing agent such that an agent can also place itself back into the MARE system for migration to a different host. The insertion of an agent through a MARE call can be seen in Figure 6-11. The example demonstrates the agent injected by a rescuer initially onto the embedded device with no resources required for execution. Currently the resources required by the agent for execution are formed in the same way as a ResourceDescriptor, i.e. as key value pairs separated by a semi colon. This enables the ResourceDescriptor helper class to be used for the generation of required resource list. This is a simple approach that could potentially be extended through the use of a self describing technique such as XML allowing a more dynamic data representation with a large degree of extensibility.

```
//Generate the Agent
MyAgent myAgent = new MyAgent() ;
//specify any resourcesRequired
String resourcesRequired = "" ;
//Send the agent
agentRuntime.send(resourcesRequired, myAgent) ;
```

Figure 6-11 Inserting an agent

The MARE system by default serialises the agent and its state prior to it being placed into the L^2imbo tuple space. This operation involves no further involvement by the application programmer other than calling the addition method. The agent is then assigned a UID allowing the agent to be uniquely addressed; further unique identifiers

can be attached allowing the agent to register itself on arrival at a MARE instance for the receipt of messages, in effect allowing an agent to have multiple addresses.

6.4.3 Resource Discovery and Configuration

The technologies previously discussed combine to form a basis for resource discovery through the use of the tuple space and resource configuration through the use of agents. Whilst agent systems have implemented resource discovery mechanisms for static and mobile environments, MARE provides an approach targeted at operating in an ad hoc or active environment. The MARE approach has a novel resource discovery technique and through the use of mobile agents using the same transport mechanism enables dynamic configuration of the available resources. The agent will receive notifications of available and no longer available resources form the MARE instance it is executing upon. An agent may then adjust its configuration accordingly and offer different resources or add further agents into the MARE system.

6.4.4 Analysis

The issues examined here are summarised in Table 6-1 for operating in an active environment.

Issue	Notes	Addressed
Resource Advertising	MARE offers a proactive approach incorporating resource bundling and compression	YES
Resource Consistency	Achieved through a non centralised approach by announcements renewing a host's available resource list	YES
Bandwidth Consumption	Minimal consumption achieved through proactive resource announcements and bundling	YES
Resource Usage	A helper class is implemented mapping to a simplistic key value pair description.	LIMITED
Agent Generation	A single inherited class allows the generation of an agent.	YES
Agent Migration	MARE uses Java byte code. The migration is handled by the MARE system with the aid of the L^2imbo *eval* operation. The migration technique can be extended and controlled by the application programmer.	YES
Agent Execution	The agent's and runtime environment is checked for applicability to execute an agent at a particular runtime level.	YES
Agent Usage	A simple interface is extendable allowing callbacks on system state change such as resource availability changes.	LIMITED

Table 6-1 MARE issues summary

This table emphasizes the author's belief that the MARE approach addresses the key issues, but accepts that further refinement of the approach can be performed. In particular the descriptions of resources required for agents are simple key value pairs.

Whilst adequate for simple examples more complex resource requirements may require an approach such as XML.

6.5 Quantitative Evaluation

6.5.1 Introduction

This chapter has performed a qualitative evaluation of the MARE approach for the constituent technologies, resource discovery and mobile agents. To perform a more comprehensive evaluation of the MARE approach this section aims to quantitatively examine the MARE approach. This section emphasises the underlying resource discovery technique that utilises L^2imbo tuple spaces. The quantitative evaluation of mobile agent usage within MARE is not performed as mobile agents within the approach are autonomous and can react differently in any situation dependent on the programmer / user requirements and environmental conditions.

To perform the quantitative evaluation a simple view of the case study examined earlier in this chapter is used. Each host has a service relating to a physical device such as a digital camera, long distance communications device, digital video camera and medical monitor. Each of the hosts advertises their resources and aims to discover all available resources that are held on the other devices. Further simplification will be made by assuming all devices perform a discovery at or near the same time. This is a scenario in line with a trigger for discovery such as a change in environment or a power cycle. This example demonstrates a worst-case scenario where all resources need to be discovered. However, this scenario illustrates a similar case where a host moves into an existing ad hoc grouping and requests an update of all available resources triggering an updating of all hosts.

6.5.2 Resource discovery

The quantitative evaluation will be performed by examining the interactions performed by SSDP (Simple Service Discovery Protocol) the service discovery component of UPnP, SLP and the discovery aspects of MARE.

UPnP

As described in chapter 2 UPnP utilises a discovery protocol called Simple Service Discovery Protocol (SSDP). Each SSDP query in an ad hoc grouping will provide three multicast requests, each responded to by three sets of unicast responses. Each *response set* consists of three *root device* advertisements, two *embedded device* advertisements and one *service* advertisement. Where the *root device* is typically a description of the physical host, the *embedded device* is an attached component such as a digital camera or GPS compass. The *service* advertisement provides information regarding a service for example how to use an attached device. The number of messages made between two hosts where one desires knowledge of the other can be summarised as:

Messages = 3 x request + 3 x response sets

Messages = 3request + 3(3 x root device + 2 x embedded device + 1 x service)

Messages = 3 + 3(6) = 3 + 18

Messages = 21

This can be further simplified when considering one host desiring a complete view of the surrounding resources from the surrounding hosts (n). This can be expressed as follows.

Messages = 3 x requests + 3 x (n-1) x response sets

Messages = 3 x requests + 3 x (n-1) x (3 x root device + 2 x embedded device + 1 x service)

Messages = 3 + 3(n-1)(6)

Messages = 3+18(n-1)

Messages = 3+18n-18

Messages = 18n-15

To further extend this to examine the discovery of all resources for all hosts the expression needs multiplying by the number of nodes taking part in the discovery.

Messages = n (18n-15)

Messages = $18n^2 – 15n$

Relating to the test scenario, a single host requesting a view of the service held by each of its surrounding neighbours will generate *fifty seven* messages. Thus, for all hosts to achieve a consistent view of the services offered by its neighbours, a total of *two hundred and twenty eight* messages are needed.

SLP

Providing a much simpler approach than SSDP, SLP will attempt to work with servers which are unlikely to be found within an active environment due to changing membership and resource poor hosts. The lack of servers offering a directory agent requires all hosts to perform a multicast discovery followed by a unicast response to facilitate a discovery operation. This approach generates an RPC style interaction whereby each host multicasts a request for services and all other hosts respond with a unicast response. For example a host generates one request to which each host (n) generates a response. This can be expressed for a single host requesting information from one other host as:

Messages = request + response

When applied to a number (n) of hosts this expression becomes:

Messages = request + (n-1) x responses

Further to this if every host requires updating then the expression becomes:

Messages = n x (request + (n-1) x responses)

Messages = n + n(n-1)

Messages = n+ n^2-n

Messages = n^2

Thus in this scenario a single host requiring its three neighbours to update it will produce four messages whilst a complete update of all four hosts will require a total of sixteen messages.

Jini

Jini operates by default through a lookup service that acts as a server style application providing resource descriptions to hosts in request response manner. Operation without a lookup service that is unlikely to be found in an ad hoc grouping due to its operational requirements being in excess of many mobile devices. Making a comparison with Jini can be done when forcing each host to run a lookup service in effect performing peer lookup forcing direct interaction between client and server. This approach is mentioned in the Jini specification in a limited amount of detail making an educated calculation of figures currently infeasible.

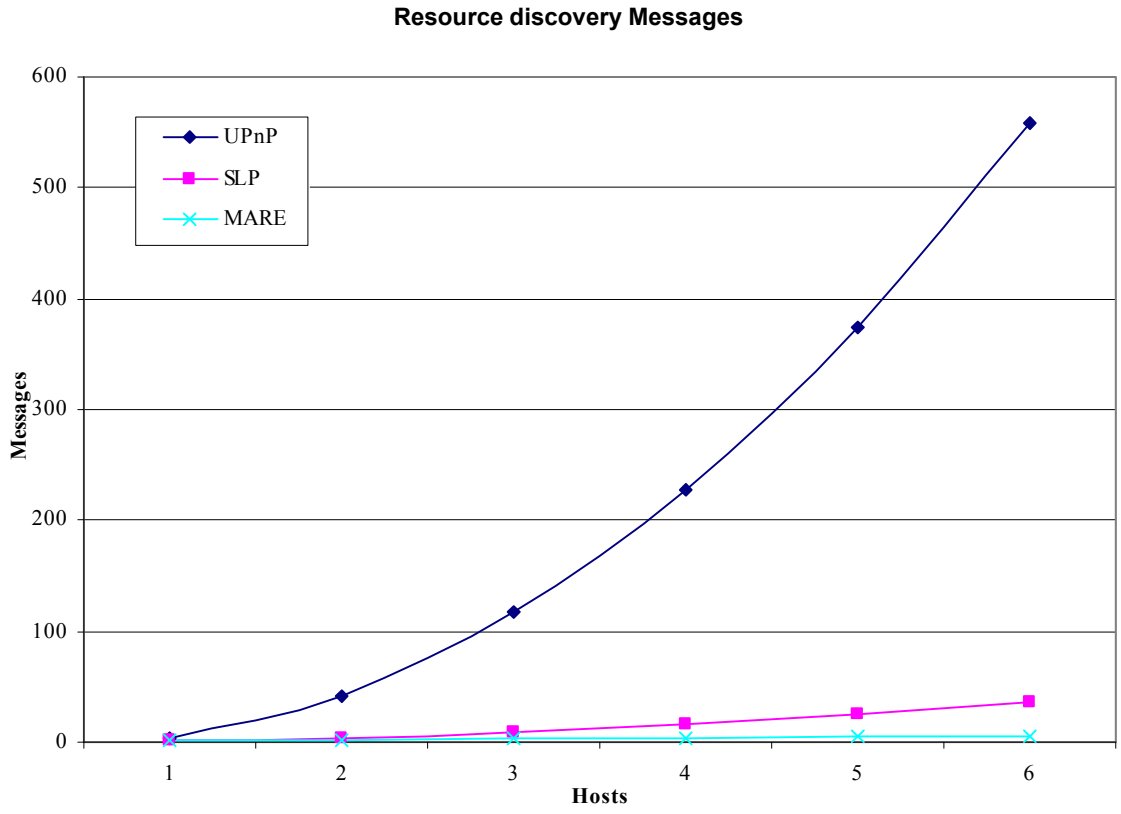
MARE

MARE adopts a much simpler approach using periodic announcements of services thus generating only four messages transmitted through a tuple space. Multiple services on a host are combined into a single message containing multiple service descriptions reducing the number of transmitted messages, message size permitting. The L^2imbo tuple space also acts as a cache whereby local requests can be satisfied locally saving external communications.

Messages = n

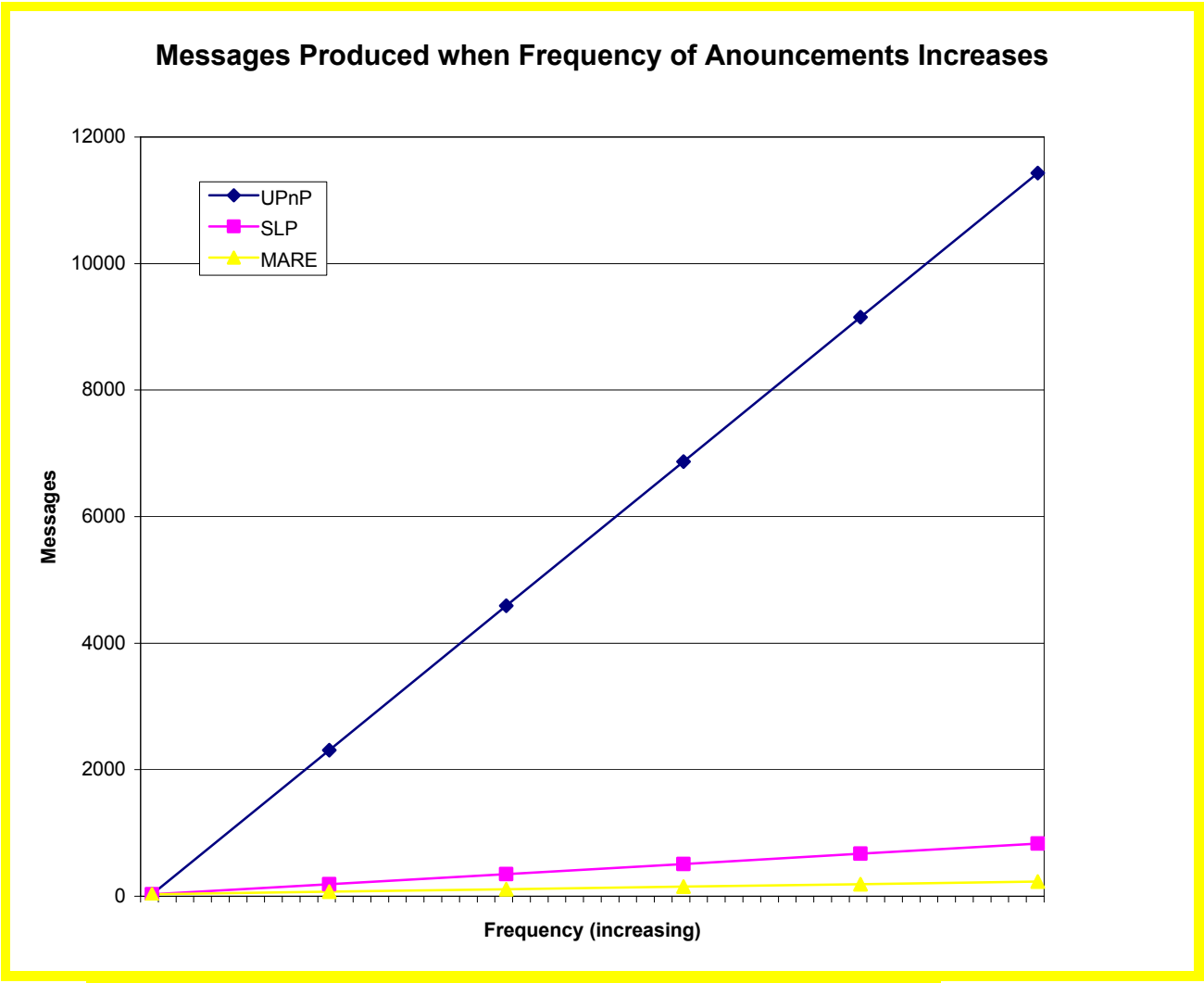
6.5.3 Analysis

The total number of messages produced when performing the discovery of four resources where each resource is held on a separate host can be seen in Figure 6-1. Whilst this is a simplified example of resource discovery, it demonstrates the potential for generation of large quantities of messages when performing a discovery. This illustrates that MARE performs very favourably when compared to SLP with both approaches producing far less messages then the SSDP approach utilised by UPnP.



Graph 6-1 Messages Produced in Resource Discovery

The resource discovery examination performed here only considers the costs of discovering all available resources. The possibility of a full discovery being required is high as frequent power cycles and movement performed by mobile hosts operating in an active environment is high. A full discovery is also the only way to discover resources that may not fall into specific type groups that can be utilised to achieve the same goal in combination. MARE provides knowledge of all resources whereas the other approaches require a number of requests to achieve the same goal unless performing a full discovery. Clearly service discovery protocols can be optimised to improve service discovery performance (e.g. SLP can use a directory agent). However in ad hoc environments such techniques typically lead to increased startup costs for example when a system must detect the absence of central resources. UPnP supports announcement messages from services however to be effective these messages must be transmitted frequently forming an approach similar to the approach taken by MARE without message combining requiring less connections to be formed.



Graph 6-2 Messages produced when frequency of updates increases

The impact of the frequency of announcement can be seen in Graph 6-2 demonstrating the impact of an increasing frequency on the number of messages produced in an environment with only 4 nodes each with one resource. Graph 6-2 again demonstrates the high volume of messages created by UPnP and SLP when compared to MARE. Further requests from clients performed in UPnP and SLP in between the complete updates are not added to this graph. Further requests can be seen as adding to the numbers of requests and responses generated and increasing the UPnP and SLP message count. MARE uses a local cache to service requests for resources generating fewer messages. As previously stated the need to request all resources is necessary to gain a full picture of available resources that, although not matching a client's immediate needs, may be able to be combined to meet requirements. For example when searching for a compressed video feed, a

compression agent could be utilised with a normal video source to provide the required output.

6.6 Requirements Revisited

The evaluation performed previously in this chapter has focused on key issues highlighted in the design and implementation chapters. This section relates the evaluation performed in this chapter to the requirements highlighted in chapter 2.

The removal of reliance upon a server based architecture as the nature of mobility produces fluctuating connectivity and periods of disconnection

The case study examined in this chapter has resources that are transient and may not be available for long periods of time. The dynamic nature of this environment is not well suited to server based architectures. Aspiring to the removal of reliance upon a server based architecture MARE has been designed to be able to distribute agents and resources using the L^2imbo tuple space implementation. The L^2imbo implementation is decentralised allowing insertions and reading from the tuple space in isolation. The implementation seamlessly updates the surrounding nodes when they are contactable aiming to achieve eventual consistency.

Using a minimal amount of bandwidth in keeping an updated view of resources and the transmission of agents

The amount of bandwidth consumed is an important consideration in any environment, especially mobile environments where available bandwidth can be a scarce commodity. MARE employs a proactive approach to resource discovery in that every resource is advertised periodically. Where a host has more than one resource, the resource descriptions are combined into a resource bundle before transmission. To further aid in utilising the minimal amount of bandwidth, MARE can seamlessly support compression of resource descriptions and bundles of descriptions. The assumption in MARE is that a host requires a continual view of available resources that is as accurate as possible. Traditionally a request is issued for available resources and a response generated from a server or all hosts that have the resources requested. This approach will not work if the server is missing or when the specific resource requested is not available. Furthermore gaining a consistent view of resources can result in many requests and responses from multiple hosts. The MARE approach

provides a continual update of all available resources such that each host periodically announces its resources to all listening hosts. If a host does not receive the announcement in the required time frame, the resource is deemed to be unavailable. This approach also provides hosts with resource descriptions that they may not have been looking for that can be used / combined to provide a resource they were seeking. For example a request for a video source in h.263 format can be achieved by locating a video source and an encoding agent that can process the stream to the required format.

Facilitating adaptation to environmental changes to make use of available resources

MARE adapts to environmental changes to enable more efficient use of available resources. This is important within any environment especially in an active environment where available resources change frequently. MARE reports any changes in resource availability to interested agents in the runtime environment through a call back mechanism implemented by the agent. Agents can also be used to dynamically configure and reconfigure resources to be offered as other resources.

6.7 Summary

The qualitative evaluation performed on the MARE approach has examined both the resource discovery and configuration aspects. Resource *discovery*, *advertising* and maintaining of *consistency* has been examined favourably with other approaches. The use of a proactive approach has enabled *bandwidth* saving features such as bundling of resources and seamless encryption. The *usability* of the resource interfaces of MARE has been examined using the emergency multimedia scenario. In the author's opinion MARE is simple and comfortable to users with basic programming abilities. The qualitative approach also highlights the configuration of resources through the use of mobile agents emphasising the *generation*, *migration* and *execution* combining proven features from other agent systems for application in an active environment. Again, in the author's opinion the combination of features from different approaches has led to an intuitive API making the generation and *usability* of agents a natural operation for users. The quantitative approach has focused upon the resource *discovery* aspects of MARE. This section highlights the efficiency of the MARE approach when compared to other popular resource discovery approaches.

This chapter has revisited the initial requirements from chapter 2 highlighting how the MARE approach has addressed these issues. Concluding remarks will be offered in the following chapter regarding the analysis performed in this thesis, the MARE approach and future directions for this work.

Chapter 7

Conclusion

7.1 Overview

This thesis has examined how resource discovery and configuration can be achieved in an active environment where availability of resources is transient and bandwidth is at a premium. An examination of resource discovery mechanisms has been performed concluding that a new approach to resource discovery is required as well as a facility to manipulate resources for the purposes of adaptation when operating in an active environment.

The MARE approach is a novel approach combining tuple space and agent technologies designed to address issues raised whilst examining resource discovery techniques. The approach has been described and shown to address many of the issues relating to operation in an active environment. This chapter summarises this thesis highlighting the key contributions and areas of potential further study before offering final concluding remarks.

7.2 Thesis Outline

The introductory chapter introduced the concept of an active environment highlighting issues when operating within such an environment. The chapter examined mobile computing devices and communications infrastructure that are currently available and likely to be found in a mobile environment. The emergency multimedia scenario was examined highlighting the rescue of a distressed party. This introduced and highlighted the need for awareness of available resources and the ability to adapt to utilise resources. The chapter concluded with initial aims of the thesis and outlined the remainder of the thesis.

Within chapter 2 a survey of existing middleware was performed; particularly platform support for mobile systems was examined. The chapter moved on to examine in more detail two areas of mobile computing research, namely resource discovery and configuration performed by mobile agent technologies. Through the exploration of these areas it was concluded that resource discovery and configuration techniques employed in static and mobile environments do not offer a solution for an active environment.

Chapter 3 examined in more detail resource discovery through utilising tuple spaces. The chapter examined the tuple space approach highlighting advances in the tuple space approach from the initial Linda implementation. Such advances include the development of multiple tuple spaces for partitioning of data; distributed approaches that do not use server based interaction, local tuple spaces for inter process on the same host and bulk primitives for operating on multiple tuples in a single operation. The chapter concluded that the tuple space approach is a good candidate for the distribution of resources amongst interested parties.

The MARE approach was introduced in chapter 4, highlighting the novel combination of tuple space and agent technology to perform resource discovery and configuration. The chapter highlighted the features of the tuple space and agent combination that MARE adopts and developed to offer a solution well suited to operating in an active environment. The chapter introduced the MARE architecture where the individual modular components are examined. The modular approach allows for different components to be developed in isolation to the remainder of the MARE system.

Chapter 5 discussed the MARE implementation examining the MARE prototype in terms of the individual components of the architecture described in chapter 4. The chapter discussed the overall component architecture examining issues related to each component in turn.

An evaluation of the MARE approach is performed in chapter 6, in both a qualitative and quantitative manner. The evaluation started by revisiting the emergency multimedia rescue scenario outlined in chapter 1 of this thesis. The scenario was examined and expanded to include the emergency rescue application. The scenario and prototype implementation was referred to throughout the qualitative and quantitative examination of the MARE approach. The evaluation emphasized the ability for MARE to operate without reliance on server based architectures, operation utilising a minimal amount of bandwidth and facilitating adaptation to environmental changes. The chapter concluded that MARE offers an approach well suited to resource discovery and configuration in an active environment.

The remainder of this chapter describes the contributions of the thesis and addresses several areas of potential future work before offering some concluding remarks.

7.3 Major Contributions

7.3.1 Definition and Analysis of the Active Environment

The target area of this thesis has been operations within ad hoc environments where membership of the grouping is dynamic. This type of environment has been termed in this thesis as an *active environment*. An example of an active environment can be seen in the scenario explored in chapter 1 of an emergency rescue. In this scenario a number of rescuers converge at a given point carrying different equipment to aid in the rescue. It is beneficial for devices to discover and interoperate without external interference utilising surrounding resources seamlessly for the efficient rescue of the stricken party. Further scenarios can be seen in other impromptu groupings formed in the automotive world where cars meet in car parks, a traffic jam or simply when passing on a road. Groupings of aircraft are on a much larger scale; however an ad hoc group formed when flying in formation or when loading at an airport can be envisaged. Such groupings are frequent and appear in everyday lives, the availability

of devices that are capable of communication leads to many diverse devices with little or no structured networking.

The development of the MARE approach through practical experimentation, published papers [Storey'00][Storey'02] and within this thesis has highlighted a number of requirements for operating in an active environment. Such requirements include removing reliance upon central servers, maintaining a consistent view of available resources and operating whilst utilising a minimum amount of bandwidth. A vast body of work has focused on facilitating operation within static and mobile environments. Far less work has focussed upon peer to peer environments and less still on operating in ad hoc environments. This thesis has highlighted these issues and provides a basis for further analysis and examination of these requirements.

7.3.2 Use of Tuple Spaces for Resource Discovery

The lack of stable network infrastructure in mobile environments has led to diverse techniques for performing interactions between member hosts. For example client server interaction cannot be relied upon due to the high probability of disconnection. Techniques for addressing disconnected operation include the queuing of messages for replaying when a connection is re-established. When considering the active environment a first step is considering the techniques employed in the static and mobile environments. Through this examination an approach is required that offers some resilience to disconnection and consideration to low bandwidth connections. One such technique involves the use of a tuple space allowing operations to continue within a mobile environment through periods of disconnection.

The L^2imbo approach offers a fully distributed implementation removing the reliance on central servers. The approach proves that the tuple space is a valid approach for interactions within a mobile environment, but not necessarily for point to point communications for such operations as large audio or video interactions [Wade'99]. This thesis applies the L^2imbo approach to the active environment for the distribution and management of resources but relying on other approaches for inter host communications. The L^2imbo implementation offers many attractive features for performing such an operation due to its distributed design providing the removal of reliance on servers. Maintaining consistency is achieved through synchronisation

when connected and bandwidth and priority constraints are satisfied. Partitioning of communications into separate tuple spaces allows tuning into and out of areas of interest. Furthermore the tuple space allows the generation and use of local tuple spaces for the sharing of information upon the local host. These features make the L^2imbo implementation a good mechanism for distributing and managing resource information.

7.3.3 The Combining of Tuple Space and Agent Technologies

This thesis has proposed the MARE approach that has involved the combination of tuple space and agent technologies to better facilitate operations within an active environment. The combination of the two technologies is novel and provides many features desirable within an active environment, not only the features for distributing resources as highlighted previously through the use of the tuple space but the ability to manage and control resources. The use of agents allows adaptation to be performed offering new resources built out of available resources to better suit a client. The use of agents also allows the rebuilding of configured resources to better suit the operational environment, for example when network bandwidth increases or a resource becomes unavailable, further resources can be brought into use allowing operation to continue seamlessly. This allows a user to generate an agent that can operate on their behalf configuring resources autonomously if required.

7.4 Other Significant Findings

7.4.1 The *eval* Operation

The *eval* operation defined in Linda was used to insert active tuples into a tuple space for computation within the tuple space placing a result within the tuple space for collection or further manipulation. The initial L^2imbo implementation had no support for the *eval* data type and associated operations, support for *eval* was added to L^2imbo as part of the development of the MARE approach enabling the distribution of active code. The L^2imbo implementation seamlessly takes Java code and transforms it into a format that can be moved through the tuple space. The *eval* data type is reassembled by the tuple space when required. This ease of insertion and reconstruction can be seen in the simplicity of inserting and consuming mobile agents as highlighted in the evaluation performed in chapter 6. The approach was further refined to allow for a

choice as to the transporting of associated classes or not with the main code. This option was provided due to the potential size that an agent may attain and the flexibility offered by the loading of constituent classes from other sources. For example constituent classes can be placed in the tuple space or at a central repository. These approaches offer flexibility and potential bandwidth savings at the expense of the potential failure of availability of constituent classes.

7.4.2 Development of a Lightweight Agent Architecture

The MARE approach utilises the tuple space as a transport mechanism for resource descriptions. To facilitate the manipulation of these resources the use of mobile agents has been adopted. After an examination of mobile agent systems developed for operating in static and mobile environments, it was concluded that they are not appropriate for operating in an active environment utilising tuple spaces. Amongst the reasons for this are a reliance upon servers for constituent classes, static communication meeting points and targeted migration. The likelihood of disconnection and membership changes also makes the availability of servers and specific hosts unlikely.

The MARE approach utilises the tuple space for transportation of mobile agents through the use of the *eval* data type and associated operations detailed previously in this chapter. Once an agent has been accepted for execution on a particular host it is assembled and started in the user level the MARE runtime is operating in. More than one MARE runtime can execute on a single machine allowing different runtime levels when available for different MARE instances such as guest, user or administrator. Both the agent and the runtime have the ability to move the agent through a request and then if required a forced migration will take place which places the agent back into the tuple space for consumption at an appropriate host. The approach is lightweight allowing for further extensions such as more specific security settings and refinement of dynamic class loading.

7.4.3 Analysis of Current Middleware Solutions

Through the development of the MARE approach an analysis of existing middleware solutions has been performed in chapters 2 and 3 of this thesis. This analysis focussed on resource distribution and agent execution environment. Through this analysis was

concluded that current middleware solutions do not fully consider operations in active environments. This thesis serves as a survey of existing middleware approaches that facilitate operations for resource discovery and configuration within an active environment. Clearly there is further work in addressing the issues raised in operating within an active environment highlighted by the examinations performed in this thesis Existing approaches have a heavy reliance on client server interactions and assume short term disconnection rather than complete termination of a connection, providing only a basic level of support for when such an event occurs.

7.5 Future work

7.5.1 Towards Large Scale Active Environments

MARE has been designed specifically for operation within an active environment, anticipating a small number of hosts. This thesis has not focussed on operating in an environment where the number of hosts is large for example of the order of millions. The impact of the number of hosts increasing can be seen as negligible to the tuple space approach however the number of resources may also increase. This is due to resources being proactive and hosts without resources simply listening for resource announcements. The number of resources available will directly affect the amount of bandwidth consumed. The consumption of bandwidth through resource announcements has been simply addressed in the MARE approach by employing compression and bundling of resources. Other techniques could include modifying the time interval of resource update announcements to be at a dynamic interval to spread the required bandwidth over a larger period of time and filtering at the tuple space level of resources, messages and agents.

7.5.2 Extensibility of MARE

The resource descriptions and required resource list transmitted with MARE agents are basic key / value pairs with a simple defined structure and user defined elements. This approach whilst economic and simplistic does not offer easy expansion to allow for a self descriptive approach to allow a more dynamic content and interoperability between existing systems. The eXtensible Markup Language (XML) offers the ability to perform such operations and is being widely adopted for describing data and passing of information between diverse systems. Parsing an XML document in an

active environment where bandwidth and computational resources are scarce needs to be examined and determined for possible benefits and drawbacks.

7.5.3 Securing the Active Environment

The security currently available in the active environment is basic. The environment and agent check each other's available and desired resources prior to execution and the host may employ a Java ClassLoader to establish a level of security through checking various aspects of an agent. An agent may also be used as a bootstrap for further agents in a controlled manner. These approaches rely upon some prior knowledge of acceptable security levels and signers of code bases, requiring updating and monitoring at regular intervals allowing basic authentication when credentials are available. Other techniques for verifying an agent's credentials can be seen in schemes that employ inferred trust where an agent is trusted by someone trusted thus they must be trustworthy. This is an approach with limitations, such as if the trusted party becomes compromised in some way then trust of a compromised host is performed. Further examinations of security techniques that are appropriate for operating in an active environment are clearly required.

7.5.4 Refining the Tuple Space

The MARE approach has utilised the existing L^2imbo tuple space implementation with the addition of the *eval* data type for transmission of active code. The tuple space can control the forwarding of tuples through itself by tunnelling and forwarding offering the potential for shaping agent and resource traffic. These features have not been explored in depth in this thesis; however they do offer the potential to optimise the tuple space for operating in active and mixed environments. A mixed environment may occur when a participating host has a link to a static network. Using the host as a conduit between the static and active environments can be achieved through forwarding or tunnelling of tuples. Further refinement to the tuple space operations can be developed such as the insertion of policies for the cleaning of stale tuples from within the tuple space maintaining a consistent tuple space. Work in these areas has been performed since the development of Linda and adapted and enhanced for operation in new environments such as the mobile environment and now the active environment. In summary further work can and should be undertaken in refining the behaviour of the tuple space.

7.6 Concluding Remarks

The diversity found in computing environments is growing; the mobile computing domain is occupied by a wide variety of devices with different characteristics. The proliferation of communication capable devices and the increasing levels of mobility are generating active environments where ad hoc groupings of diverse devices are formed with a dynamic membership. Middleware has an important role providing the ability for facilitating operations in mobile environments. This thesis has concluded that existing middleware is not appropriate for operating in an active environment.

This thesis has proposed the MARE approach, a novel combination of tuple spaces and mobile agents. This approach offers a middleware solution for allowing the discovery and configuration of resources within active environments. This is done in a manner that is sympathetic to the operational constraints of such an environment including limited bandwidth, periods of disconnection and adaptation to change.

The author believes that through further construction, examination and modelling of active environments the challenges and issues highlighted in this thesis can be expanded. It is hoped that the community of researchers and enthusiasts at large will build and develop the MARE approach, ultimately allowing users to utilise and manipulate resources in an efficient manner in active environments.

References

[Adcock'99] P. Adcock, Supporting Mobile Applications in a Heterogeneous Distributed Environment, Ph.D. Thesis, Distributed Multimedia Research Group, Computing Department, Lancaster University, Bailrigg, Lancaster, LA1 4YR, U.K., 1999.

[Allyn'02] Welch Allyn UK, Manufacturers of Propaq Medical monitoring systems http://www.welchallyn.co.uk/, 2002.

[Amir'95] E. Amir, H. Balakrishnan, S. Seshan and R. Katz, "Efficient TCP over Networks with Wireless Links", Proceedings of the 5th IEEE Workshop on Hot Topics in Operating Systems (HotOS-V), Orcas Island, Washington, U.S., IEEE Computer Society Press, 4th-5th May 1995.

[APM'89] APM Limited, ANSA: An Engineers Introduction to the Architecture, Technical Document release TR.03.02, APM Cambridge Limited, Poseidon House, Castle Park, Cambridge, CB3 0RD, U.K., November 1989.

[APM'93] APM Limited, ANSAware 4.1 Application Programming in ANSAware, Technical Document RM.102.02, APM Cambridge Limited, Poseidon House, Castle Park, Cambridge, CB3 0RD, U.K., February 1993.

[Arnold'00] Ken Arnold, Robert Scheifler, and Jim Waldo, The Jini Specification, Second edition, Penguin Books (N.Z.) Ltd.; ISBN: 0201726173, November 2000.

[AT&T‘93] AT & T, Development of WaveLAN, an ISM band wireless LAN, Technical Journal, pp. 27-37, July/August 1993

[Bakre’95] A. Bakre and B. R. Badrinath, I-TCP: Indirect TCP for Mobile Hosts, Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS), Vancouver, British Columbia, pp 136-143, 30th May-2nd June 1995.

[Baumann‘98] J. Baumann, F. Hohl, K. Rothermel, M. Schwehm, M. Straßer, Mole 3.0: A Middleware for Java-Based Mobile Software AgentsMiddleware'98 IFIP International conference on Distributed Systems Platforms and Open Distributed Processing, pp. 355-370, 1998.

[Bakken’94] D. E. Bakken, Supporting Fault-Tolerant Parallel Programming in Linda, Ph.D. Thesis, Department of Computer Science, University of Arizona, Tucson, Arizona 85271, U.S., 8th August 1994.

[Bluetooth‘99] Bluetooth, Bluetooth Specifications, http://www.bluetooth.com/, 1999.

[Candy ‘98] Ed. Candy et al, Emergency Multimedia, Simoco Europe Ltd., the Langdale Ambleside Mountain Rescue Team, and HW Communications Limited, Multimedia Demonstrator Programme funded by the Department of Trade and Industry, 1989.

[Carriero‘89] N. Carriero and D. Gelernter, Linda in Context, Communications of the ACM, Volume 32 Number 4, pp 444 – 458, 1989.

[Carriero’94] N. Carriero, D. Gelernter and L. Zuck, "Bauhaus Linda", Selected Papers from the Workshop on Models and Languages for Coordination of Parallelism and Distribution (ECOOP '94), Bologna, Italy, pp 66-76, July 1994.

[Chill’95] A. Schill and S. Kümmel, Design and Implementation of a Support Platform for Distributed Mobile Computing, Distributed Systems Engineering Journal, 2(3), pp. 128-141, 1995.

[Davies‘98] N. Davies, A. Friday, S. Wade and G. Blair, L2imbo: A Distributed Systems Platform for Mobile Computing, ACM Mobile Networks and Applications

(MONET), Special Issue on Protocols and Software Paradigms of Mobile Networks, Volume 3, Number 2, pp143-156, 1998.

[Davies'99] N. Davies, K. Cheverst, K. Mitchell, and A. Friday, Caches in the Air: Disseminating Information in the Guide System, Proc. 2nd IEEE Workshop on Mobile Computing Systems and Applications (WMCSA '99), New Orleans, U.S., IEEE Press, Pages 11-19, 1999.

[Demers'94] A. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer and B. Welch, The Bayou Architecture: Support for Data Sharing Among Mobile Users, Proceedings of the 1st IEEE Workshop on Mobile Computing Systems and Applications (WMCSA '94), Santa Cruz, California, U.S., IEEE Computer Society Press, 8th-9th December 1994.

[Douglas'95] A. Douglas, A. Wood and A. Rowstron, Linda Implementation Revisited, Transputer and Occam Developments, IOS Press, pp 125-138, 1995.

[ETSI'95] ETSI, Radio Equipment System (RES) Trans-European Trunked Radio (TETRA); Voice plus Data (V+D), Designer's Guide, ETSI Work Programme DE/RES – 0601, Subtechnical Committee (STC) RES 06, May 1995.

[Finney'99] J. Finney, Supporting Continuous Multimedia Services in Next Generation Mobile Systems, PhD Thesis, Distributed Multimedia Research Group, Computing Department, Lancaster University, Bailrigg, Lancaster, LA1 4YR, U.K, September 1999.

[Fitzpatrick'98] T. Fitzpatrick, G. Blair, G. Coulson, N. Davies and P. Robin, Software Architecture for Adaptive Distributed Multimedia Applications, IEE Proceedings - Software Vol. 145, No. 5, pp163-171 October 1998.

[Friday'96] A. Friday, Infrastructure Support for Adaptive Mobile Applications, Ph.D. Thesis, Distributed Multimedia Research Group, Computing Department, Lancaster University, Bailrigg, Lancaster, LA1 4YR, U.K., October 1996.

[Friday'99] A. Friday, N. Davies, J. Seitz, M. Storey and S. Wade, Experiences of Using Generative Communications to Support Adaptive Mobile Applications,

Distributed and Parallel Databases, Special Issue on Mobile Data Management and Applications, Volume 7, pp1-24, Number 3, 1999.

[Gellersen'02] HW Gellersen, A Schmidt and M Beigl Multi-Sensor Context-Awareness in Mobile Devices and Smart Artifacts, in Mobile Networks and Applications (MONET), Oct 2002

[Gelernter'85] D. Gelernter, Generative Communication in Linda, ACM Transactions on Programming Languages and Systems, Volume 7, Number 1, pp 255-263, 1985.

[Goland'99] Y. Goland, T. Cai, P. Leach., Y. Gu, S. Albright, Simple Service Discovery Protocol, IETF Draft. 1999

[Goland'00] Y. Goland, J. Schlimmer, Multicast and Unicast UDP HTTP Messages, UPnP forum Technical Committee Draft http://www.upnp.org/resources/specifications.asp, October 2000.

[Gosling'00] James Gosling, Bill Joy, Guy Steele and Gilad Bracha, The Java Language Specification,Addison Wesley; ISBN: 0201310082, July 2000

[Gray'02] R. Gray, G. Cybenko, D. Kotz, R. Peterson and D. Rus. D'Agents: Applications and Performance of a Mobile-Agent System. Software-- Practice and Experience, 32(6):543-573, May, 2002.

[Gray'96] R. Gray, Agent Tcl: A flexible and secure mobile-agent system, Proceedings of the 1996 Tcl/Tk Workshop, pages 9-23, 1996.

[Gray'96a] R. Gray, D. Kotz, S. Nog, D. Rus, and G. Cybenko ,Mobile agents for mobile computing, Technical Report PCS-TR96-285, Dept. of Computer Science, Dartmouth College, May 1996.

[Guttman'99] E. Guttman, The Service Location Protocol, IEEE Internet Computing pp71-80, July 1999.

[Haahr'99] M. Haahr, R. Cunningham and V. Cahill. Supporting CORBA Applications in a Mobile Environment. MobiCom '99: 5th International Conference on Mobile Computing and Networking. Seattle, August 1999.

[Hinckley‘00] K. Hinckley, J. Pierce, M. Sinclair, E. Horvitz, Sensing Techniques for Mobile Interaction, ACM UIST 2000 Symposium on User Interface Software & Technology, CHI Letters 2 (2), pp. 91-100, 2000.

[Hupfer‘90] S. Hupfer, Melinda: Linda with Multiple Tuple Spaces, Research Report YaleU/DCS/RR-766, February 1990.

[IBM‘99] IBM Corporation, Aglets Specification, http://www.trl.ibm.co.jp/aglets/, 1999.

[IBM‘99a] IBM Corporation, TSpaces: The Next Wave, Hawaii International Conference on System Sciences (HICSS-32), 1999.

[IBM‘00] IBM Corporation, IBM MicrodriveTM, 1Gb Microdrive specification, http://www.storage.ibm.com/hdd/micro/index.htm, 2000.

[Intel'96] Intel and Microsoft Corporation, Advanced Power Management (APM) BIOS Specification, Revision 1.2, 1996.

[Ioannidis’93] J. Ioannidis and G. Q. Maguire, The Design and Implementation of a Mobile Internetworking Architecture, Proceedings of the USENIX Winter Conference, January 1993.

[ISO’98] International Standards Organization, Information Technology; Open Distributed Processing; ISO Standard ISO/IEC 10746, 1998.

[Jacobsen’97] K. Jacobsen, and D. Johansen. Mobile Software on Mobile Hardware -- Experiences with TACOMA on PDAs. Technical Report 97-32, Department of Computer Science, University of Tromsø, Norway, December 1997.

[Jacobsen‘99] K. Jacobsen, D. Johansen, Ubiquitous devices united: enabling distributed computing through mobile code, SAC '99. ACM symposium on Applied computing, pages 399-404 1999.

[Johansen‘95] D. Johansen, R. van Renesse, and F. Schneider, An Introduction to the TACOMA Distributed System, Department of Computer Science, University of Tromsø, Norway, Technical Report 95-23, 1995.

[Johanson'02] B. Johanson, and A. Fox, The Event Heap: A Coordination Infrastructure for Interactive Workspaces, 4th IEEE Workshop on Mobile Computer Systems and Applications (WMCSA-2002), Callicoon, New York, USA, June, 2002.

[Joseph'95] A. Joseph, A. deLespinasse, J. Tauber, D. Gifford and M. Kaashoek, Rover: A toolkit for Mobile Information Access, Proceedings of the 15th symposium on Operating Systems Principles (SOSP'95), Copper Mountain Resort, Colorado, U.S., pp. 156-171, 1995.

[Kahn'99] J. Kahn, R. Katz and K. Pister, Mobile Networking for Smart Dust, ACM/IEEE Intl. Conf. on Mobile Computing and Networking (MobiCom 99), Seattle, WA, August 17-19, 1999.

[Li'96] W. Li, D. Messerschmitt, Java-To-Go: Itinerative Computing Using Java, Department of Electrical Engineering and Computer Sciences University of California at Berkeley, 1996.

[Microsoft'98] Microsoft Corporation, Distributed Component Object Model Protocol – DCOM/1.0 Specification, Microsoft Developer Network Library Document. Available on the Internet at http://msdn.microsoft.com/

[Microsoft'00] Microsoft Corporation, Universal Plug and Play Device Architecture Version 1, http://www.upnp.org/resources/documents.asp, June 2000.

[Microsoft'01] Microsoft Corporation, Common Language Infrastructure Standards, http://msdn.microsoft.com/net/ecma/, December 2001.

[Minsky'94] N. Minsky and J. Leichter, Law-Governed Linda as a CoordinationModel, Selected Papers from the Workshop on Models and Languages for Coordination of Parallelism and Distribution, Bologna, Italy, pp 125-146, July 1994.

[Mummert'95] L. Mummert, M. Ebling and M. Satyanarayanan, Exploiting Weak Connectivity for Mobile File Access, Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP), Colorado, U.S., ACM Press, Volume 29, pp 143-155, 3rd-6th December 1995.

[OG'99] The Open Group, http://www.opengroup.org/dce/, 1999.

[OMG'98] Object Management Group, CORBA/IIOP 2.3 Specification, OMG document formal/98-12-01, Object management group, http://www.omg.org/, 1998.

[Peine'97] H, Peine, T. Stolpmann, In Kurt Rothermel, Radu Popescu-Zeletin, The Architecture of the Ara Platform for Mobile Agents, (Eds.): Proc. of the First International Workshop on Mobile Agents MA'97 (Berlin, Germany), 1997.

[Perkins'92] C. Perkins and Y. Rekhter, Shortcut Routing for Mobile Hosts, Mobile IP Internet Draft, IBM Corporation, July 1992.

[Perkins'96] C. Perkins and D. Johnson, Mobility Support in IPv6, Proceedings of the 2nd ACM International Conference on Mobile Computing and Networking (MobiCom '96), Rye Hilton, Rye, White Plains, New York, U.S., ACM Press, 10th-12th November 1996.

[Picco'98] G. Picco, A. Murphy, and Roman, Lime: Linda Meets Mobility, G.-C., Technical report no. 98-21, 1998.

[Raines'99] Paul Raines and Jeff Tranter, Tcl/Tk in a Nutshell, O'Reilly UK; ISBN: 1565924339, April 1999.

[Román'99] M. Román, A. Singhai, D. Carvalho, C. Hess, and R. Campbell, Integrating PDAs into Distributed Systems: 2K and PalmORB., International Symposium on Handheld and Ubiquitous Computing (HUC'99). Karlsruhe, Germany. September 27-29, 1999.

[Román'00] M. Roman, D. Mickunas, F. Kon and R. Campbell, LegORB and Ubiquitous CORBA, IFIP/ACM Middleware'2000 Workshop on Reflective Middleware. IBM Palisades Executive Conference Center, NY, April 2000.

[Román'01]. M. Román, F. Kon, and R. Campbell, Reflective Middleware: From Your Desk to Your Hand, IEEE Distributed Systems Online (Special Issue on Reflective Middleware), Vol. 2, No. 5, July, 2001.

[Rowstron‘96] A. Rowstron, Bulk Primitives in Linda Run-Time Systems, University of York, Thesis, 1996.

[Rowstron’97] A. Rowstron and A. Wood, Bonita: A Set of Tuple Space Primitives for Distributed Coordination, Proceedings of the 30th Annual Hawaii International Conference on System Sciences, Volume 1, IEEE Computer Society Press, pp 379-388, 1997.

[Salutation’99] Salutation Consortium, Salutation Architecture Specification (Part-1), Technical Report Version 2.0c, 1999.

[Satyanarayanan’85] M. Satyanarayanan, J. Howard, D. Nichols, R. Sidebotham, A. Spector and M. West, The ITC Distributed File System: Principles and Design, Proceedings of the 10th Symposium on Operating System Principles (SOSP), Orcas Island, Washington, U.S., ACM Press, December 1985.

[Satyanarayanan’90] M. Satyanarayanan, J. Kistler, P. Kumar, M. Okasaki, E. Siegel and D. Steere, Coda: A Highly Available File System for a Distributed Workstation Environment, IEEE Transactions on Computers, Volume 39, Number 4, pp 447-459, April 1990.

[Satyanarayanan’94] M. Satyanarayanan, B. Noble, P. Kumar and M. Price, Application-Aware Adaptation for Mobile Computing, Proceedings of the 6th ACM SIGOPS European Workshop (Dagstuhl, Germany), 1994.

[Shill‘95] A. Schill and S. Kümmel, Design and Implementation of a Support Platform for Distributed Mobile Computing, Distributed Systems Engineering Journal, 2(3), pp. 128-141, 1995.

[Schmidt’99] A. Schmidt, K. Aidoo, A. Takaluoma, U. Tuomela, V. Laerhoven, and W. Velde, Advanced Interaction in Context, In the Proceedings of the 1st International Symposium on Handheld and Ubiquitous Computing (HUC '99),pp. 89-101, Karlsruhe, Germany, Springer-Verlag. September 27-29, 1999.

[Seitz’98] J. Seitz, N. Davies, M. Ebner, and A. Friday, A CORBA-based Proxy Architecture for Mobile Multimedia Applications, 2nd IFIP/IEEE International

Conference on Management of Multimedia Networks and Services (MMNS '98), Versailles, France, 1998.

[Stemm‘99] M. Stemm, R. Katz, Vertical Handoffs in Wireless Overlay Networks, ACM/Baltzer Mobile Networking and Applications (MONET), Special Issue on Mobile Networking in the Internet, V 3, N 4, pp. 319-334, January 1999.

[Storey‘00] M. Storey, G. Blair, Resource Configuration in Ad Hoc Networks: The MARE Approach, Third IEEE Workshop on Mobile Computing Systems and Applications (WMCSA), Monterey California, 2000.

[Storey‘02] M. Storey, G. Blair, A. Friday, MARE: Resource Discovery and Configuration in Ad Hoc Networks, MONET special edition, October 2002.

[Sun,89] Sun Microsystems, NFS: Network File System Protocol Specification, Request for Comments (RFC) number 1094, Sun Microsystems, 1989.

[Sun‘99] Sun Microsystems, Jini, JavaSpaces Specification, http://www.sun.com/jini/specs/, 1999.

[Sun’02] Sun Microsystems, Java Remote Method Invocation (RMI), Sun Microsystems, http://java.sun.com/products/jdk/rmi/, 2002.

[Sun‘02a] Sun Microsystems, Jini Specification, http://wwws.sun.com/software/jini/specs/index.html, 2002.

[Sun’02b] Sun Microsystems, Java Security, http://java.sun.com/security/index.html, 2002.

[TrafficMaster’02] TrafficMaster, http://www.trafficmaster.co.uk/, 2002.

[Verizades’97] J. Verizades, E. Guttman, C. Perkins, S. Kaplan, Service Location Protocol, rfc2165, June 1997

[Wade‘99] S. Wade, An Investigation into the use of the Tuple Space Paradigm in Mobile Computing Environments, Ph.D. Thesis, Distributed Multimedia Research

Group, Computing Department, Lancaster University, Bailrigg, Lancaster, LA1 4YR, U.K., 1999.

[Weiser‘93] M. Weiser. Some Computer Science Issues in Ubiquitous Computing, Communications of the ACM, 36(7):75-84, 1993.

[Wong‘97] D. Wong, N. Paciorek, T. Walsh, J. DiCelie, M. Young, B. Peet, Concordia: An Infrastructure for Collaborating Mobile Agents, Mitsubishi Electric ITA, First International Workshop on mobile agents, Berlin Germany, 1997.

1. Appendix A

MARE: Application Programmer Interface

8.1 Agent Interface

Public Methods

```
void run ()
void agentEnvironment (AgentRuntime agentRuntime)
void close ()
void receive (UID destination, byte[] data)
```

Detailed Descriptions

`void MARE::Agent::agentEnvironment (AgentRuntime agentRuntime)`

Called by MARE when arriving at a new Location to provide an interface to the agent runtime environment

Parameters:

agentRuntime: The agent runtime environment. This variable may be used to interact with the local runtime environment

See also: MARE.AgentRuntime

`void MARE::Agent::close ()`

Called by the Agent Environment to request the agent move.

`void MARE::Agent::receive (UID destination, byte data[])`

Called by the agent runtime environment to let the agent know of incoming data. The agent may register for data from multiple sources via calls to the agent runtime environmnet

Parameters:

To: The destination of the data

Data: The data for the given destination

See also: MARE.AgentRuntime , MARE.UID

`void MARE::Agent::run ()`

Called to start an Agent

8.2 AgentRuntime Class

Public Methods

```
AgentRuntime (AgentRuntimeEnvironment aRE, UID name, Agent a)
void run ()
void send (String resourcesRequired, Agent a)  throws Exception
void sendResourceDescriptor (String resourceDescriptor)  throws
      Exception
void send (UID destination, byte[] data)  throws Exception
void dataArrived (UID name, byte[] message)
void     resourceArrived     (UID     resourceUID,     String
      resourceDescriptor)
void resourceRemoved (UID name)
void close ()
UID getName ()
void registerData (UID dataSource)
void registerCallBack (boolean register, QoSCallback callback)
void  registerCallBack  (boolean  register,  ResourceCallback
      callback)
```

Detailed Descriptions

Constructor

`MARE::AgentRuntime::AgentRuntime (AgentRuntimeEnvironment aRE, UID name, Agent a)`

Called when generating a new Agent to run

Parameters:

aRE The AgentRuntime which allows access to the agent system for multiple agents.

name The name of the Agent

a The agent itself

Member Functions

`void MARE::AgentRuntime::close ()`
Called by the agent to release the agent from the runtime environment. A Clean shutdown

`void MARE::AgentRuntime::dataArrived (UID name, byte message[])`
Called by the AgentRuntimeEnvironment to indicate a message has arrived

Parameters:

name The destination for the data

message The message transmitted

`UID MARE::AgentRuntime::getName ()`
Get the name of the running agent

Returns :

The UID of the agent

`void MARE::AgentRuntime::registerCallBack (boolean register, ResourceCallback callback)`
Call to register for callbacks upon the delivery / removal of resources

Parameters:

register register for a callback true deregister for a callback false

callback class to receive callback

`void MARE::AgentRuntime::registerCallBack (boolean register, QoSCallback callback)`
Call to register or de-register for a QoS callback

Parameters:

register register for the callback = true deregister for the callback = false

callback QoS callback instance

See also: QosCallBack

`void MARE::AgentRuntime::registerData (UID dataSource)`
register for the receipt of data from a given source

Parameters:

dataSource UID of the source of data

`void MARE::AgentRuntime::resourceArrived (UID resourceUID, String resourceDescriptor)`
Called by the system when a resource is detected.

Parameters:

name The name of the resource

resource A handle to the resource

`void MARE::AgentRuntime::resourceRemoved (UID name)`
Called by the system to indicate a resource has been removed

Parameters:

name The name of the resource

`void MARE::AgentRuntime::run ()`
Starts the agent Runtime and triggers the Agent Called by the AgentRuntime Environmnet.

`void MARE::AgentRuntime::send (UID destination, byte data[]) throws Exception`
Called to send a message to another agent

Parameters:

destination The recipient of the message

data The data to transmit

Exceptions:

java.lang.Exception

`void MARE::AgentRuntime::send (String resourcesRequired, Agent a) throws Exception`

Called to send an agent into the environment This routine may be used to move the currently running agent.

Parameters:

resourceRequired resource requirements to run

a The agent to send

Exceptions:

java.lang.Exception

`void MARE::AgentRuntime::sendResourceDescriptor (String resourceDescriptor) throws Exception`

Called to send a resource descriptor into the environment

Parameters:

resourceUID UID of resource

resourceDescriptor descriptor of resource

Exceptions:

java.lang.Exception

8.3 AgentRuntimeEnvironment Class

Public Methods

```
AgentRuntimeEnvironment (boolean acceptAgents)
void run ()
UID addAgent (String resourcesRequired, Agent a) throws
      Exception
void sendMessage (UID to, UID from, byte[] message) throws
      Exception
```

Static Public Methods

```
UID addResource (ResourceDescriptor resourceDescriptor)  throws
      Exception
void removeResource (UID uid)  throws Exception
void  Go  (boolean  acceptAgents,  boolean  graphical)    throws
      Exception
```

Detailed Descriptions

Constructor

`MARE::AgentRuntimeEnvironment::AgentRuntimeEnvironment (boolean acceptAgents)`
Initiate MARE runtime environment

Parameters:

acceptAgents Accept incoming agents when initialised.

Member Functions

`UID MARE::AgentRuntimeEnvironment::addAgent (String resourcesRequired, Agent a) throws Exception`
Static method to add an agent

Parameters:

resourcesRequired Required resource list

a Agent to add

Returns:

The UID of the agent

`UID MARE::AgentRuntimeEnvironment::addResource (ResourceDescriptor resourceDescriptor) throws Exception`
Static method to add a resource

Parameters:

resourceDescriptor Resource description to add

Returns:

The UID of the resource descriptor

`void MARE::AgentRuntimeEnvironment::Go (boolean acceptAgents, boolean graphical) throws Exception`
Static call to start MARE

Parameters:

acceptAgents Accept agent by default

graphical Start with graphical monitor

`void MARE::AgentRuntimeEnvironment::removeResource (UID uid) throws Exception`
Remove a resource from MARE

Parameters:

uid resource to remove

`void MARE::AgentRuntimeEnvironment::run ()`
Start a MARE instance

`void MARE::AgentRuntimeEnvironment::sendMessage (UID to, UID from, byte message[]) throws Exception`
Call to send an message

Parameters:

to Where to send message

from From whom the message has come

message Message to send

8.4 QoSCallback Interface

Implemented by an agent wishing to receive QOS callbacks

Public Methods

`void updateQoS (String type, String value)`

Detailed Descriptions

`void MARE::QoSCallback::updateQoS (String type, String value)`

Called when QOS change detected

Parameters:

type Type of QOS change

value Value of QOS parameter

8.5 ResourceCallback Interface

Implemented by an agent wishing to receive resource change callbacks

Public Methods

```
void resourceArrived (UID resourceUID, String resourceDescriptor)
void resourceRemoved (UID resourceUID, String resourceDescriptor)
```

Detailed Descriptions

void MARE::ResourceCallback::resourceArrived (UID resourceUID, String resourceDescriptor)

Called when a new resource is made available

Parameters:

resourceUID unique identifier of resource

resourceDescriptor description of resource

void MARE::ResourceCallback::resourceRemoved (UID resourceUID, String resourceDescriptor)

Called when a resource is no longer available

Parameters:

resourceUID unique identifier of resource

resourceDescriptor description of resource

8.6 ResourceDescriptor Class

Used to help generate resource descriptions

Public Methods

```
ResourceDescriptor ()
ResourceDescriptor (String descriptor)
String toString ()
String getDescriptor ()
String getDescriptor (String tag)
String getValue (String key)
void removeDescriptor (String key)
void addDescriptor (String type, String value)
Hashtable getPairs ()
```

Detailed Descriptions

Constructors

`MARE::ResourceDescriptor::ResourceDescriptor ()`
Create an empty descriptor.

`MARE::ResourceDescriptor::ResourceDescriptor (String descriptor)`
Requires a string of a layout compliant with the resource descriptor layout.

Parameters:

descriptor The string format of a resource descriptor

Member Functions

`void MARE::ResourceDescriptor::addDescriptor (String type, String value)`
Add a tag value pair

Parameters:

type tag type

value value of tag

`String MARE::ResourceDescriptor::getDescriptor (String tag)`
Get a tag value i.e. TYPE=camera calling with TYPE will return camera

Parameters:

tag tag to search for

Returns :

value depicted by the tag or null.

Deprecated:

Replaced by getValue(String key)

`String MARE::ResourceDescriptor::getDescriptor ()`

Get resource descriptor

Returns :

A string representation of the descriptor. null if no descriptors present

`Hashtable MARE::ResourceDescriptor::getPairs ()`

Returns a hashtable of key value pairs

Returns :

a Hashtable of values and pairs.

`String MARE::ResourceDescriptor::getValue (String key)`

Get a Value value i.e. TYPE=camera calling with TYPE will return camera

Parameters:

key, key to search for

Returns :

value depicted by the key or null if not found.

`void MARE::ResourceDescriptor::removeDescriptor (String key)`

Remove key value pair

Parameters:

key Key name

`String MARE::ResourceDescriptor::toString ()`

Generate descriptor for screen output

Returns :

The descriptor string

8.7 UID Class

Unique identifier of a resource or agent within MARE

Public Methods

```
UID ()
UID (byte[] uID)
UID (String uID)
byte[] getBytes ()
String getUID ()
boolean equals (Object input)
int hashCode ()
String toString ()
```

Detailed Descriptions

Constructors

`MARE::UID::UID ()`

Create an empty unique identifier

`MARE::UID::UID (byte uID[])`

Construct from a byte representation

Parameters:

uID unique identifier

`MARE::UID::UID (String uID)`

Construct from a string representation

Parameters :

uID unique identifier

Member Functions

`byte [] MARE::UID::getBytes ()`

Get the byte representation

Returns

The byte representation of the unique identifier

`String MARE::UID::getUID ()`

Get the string representation

Returns

The string representation of the unique identifier

`String MARE::UID::toString ()`

Provides the string representation

Returns

The string representation of the unique identifier

2. Appendix B

Emergency Multimedia Demonstrator

9.1 Embedded Device Application

Calls made to MARE interfaces or from MARE interfaces are shown in turquoise.

```
/**
 * Emergency Rescue bootstrap.
 *
 * @author Matthew Storey
 * @version 1.0
 */
import MARE.*;

public class Go {

  public static void main(String args[]) throws Exception
  {
    //Initialise Agent Runtime Environment to accept agents
         without a graphical environment.
    AgentRuntimeEnvironment.Go(true,false);
    //Insert Mobile Agent into running environment
    AgentRuntimeEnvironment.addAgent("",new
         EmergencyRescueAgent());
    //Add resources we have e.g. GPS
    //AgentRuntimeEnvironment.addResource(new
         ResourceDescriptor("DESCRIPTION=GPS for rescuer
         3;TYPE=GPS;IP=10.0.10.10;PORT=3030"));
  }
}
```

9.2 Mobile Agent

Calls made to MARE interfaces or from MARE interfaces are shown in turquoise.

```
/**
 * MARE agent to relay resources to a client interface.
 *
 * @author Matthew Storey
 * @version 1.0
 */

import MARE.*;

public class EmergencyRescueAgent implements MARE.Agent,
         MARE.ResourceCallback {
  private AgentRuntime agentRuntime = null;
  private java.util.Hashtable resources = null;

  /*
  * Called by MARE with a handle to the agents runtime environment
  * @param agentRuntime handle to agents runtime environment
  */
  public void agentEnvironment(AgentRuntime agentRuntime) {
    this.agentRuntime = agentRuntime;
    this.resources = new java.util.Hashtable();
    //register for callback
    agentRuntime.registerCallBack(true, this);
  }

  /*
  * Called when agent is about to close.
  */
  public void close() {
    //deregister for callback before move.
    agentRuntime.registerCallBack(false, this);
    //Reset the variables (saves space when serialising agent)
    agentRuntime = null;
    resources = null;
  }

  /*
  * Receive a message
  */
  public void receive(UID destination, byte data[]) {
    //Not required in this example
  }

  /*
  * A new resource has arrived
  */
  public void resourceArrived(UID resourceUID, String
         resourceDescriptor) {
    resources.put(resourceUID, resourceDescriptor);
    sendResources();
  }

  /*
  * A resource has been removed
  */
```

```
  public void resourceRemoved(UID resourceUID, String
         resourceDescriptor) {
    resources.remove(resourceUID);
    sendResources();
  }

  /*
  * Send resources to the user interface if any
  */
  public synchronized void sendResources() {
    //Serialise and send hashtable to user interface
    ……
  }

  /*
  * Main routine to form interaction with user interface
  */
  public void run() {
    ……
  }
}
```

9.3 User Interface

Calls made to MARE interfaces or from MARE interfaces are shown in turquoise.

```
/**
 * Emergency Rescue user interface.
 *
 * @author Matthew Storey
 * @version 1.0
 */

import java.awt.*;
import java.awt.event.*;
import MARE.*;

public class ERescueApp extends Frame implements
          java.awt.event.ActionListener,
          java.awt.event.WindowListener {

  Update update;
  java.util.Vector list = null;
  String serverName = "localhost";
  java.awt.List lstResources = new java.awt.List(4);
  java.awt.Button btnExecute = new java.awt.Button();
  java.awt.TextArea txtResourceDescription = new
         java.awt.TextArea();
  java.awt.Label lbl = new java.awt.Label();
  java.awt.MenuBar menuBar = new java.awt.MenuBar();
  java.awt.Menu menu1 = new java.awt.Menu();
  java.awt.MenuItem serverSelect = new java.awt.MenuItem();

  public ERescueApp() {
    setLayout(null);
    setSize(240,300);
    setVisible(false);
    add(lstResources);
    lstResources.setBounds(12,48,214,96);
    btnExecute.setLabel("Execute");
```

```
  add(btnExecute);
  btnExecute.setBackground(java.awt.Color.lightGray);
  btnExecute.setBounds(12,144,210,33);
  txtResourceDescription.setEditable(false);
  add(txtResourceDescription);
  txtResourceDescription.setBounds(12,204,214,60);
  lbl.setText("Information");
  add(lbl);
  lbl.setBounds(12,180,152,15);
  setTitle("Emergency Rescue Appplication");
  setResizable(false);
  menu1.setLabel("Menu");
  menu1.add(serverSelect);
  serverSelect.setLabel("Server");
  menuBar.add(menu1);
  setMenuBar(menuBar);
  btnExecute.addActionListener(this);
  lstResources.addActionListener(this);
  serverSelect.addActionListener(this);
  update = new Update(this);
  new Thread(update).start();
  this.addWindowListener(this);
}

/*
* Start the interface
*/
static public void main(String args[]) {
  (new ERescueApp()).setVisible(true);
}

/*
* Main event method
*/
public void actionPerformed(java.awt.event.ActionEvent event) {
  Object object = event.getSource();
  if (object == btnExecute)
    btnExecute_ActionPerformed(event);
  else if (object == lstResources)
    lstResources_ActionPerformed(event);
  else if (object == serverSelect)
    serverSelect_ActionPerformed(event);
}

/*
* Start the appropriate resource application passing resource
       descriptor to the application. The applications will
       interact directly with resources.
*/
void btnExecute_ActionPerformed(java.awt.event.ActionEvent
       event) {
  String item = lstResources.getSelectedItem();
  if (item != null)
  {
    if (item.trim().equalsIgnoreCase("Camera"))
    {
       //Start Camera App, pass resource Descriptor info
    }
    else if (item.trim().equalsIgnoreCase("GPS"))
    {
       //Start GPS App, pass resource Descriptor info
```

```
    }
    else if (item.trim().equalsIgnoreCase("Medical Kit"))
    {
       //Start Medical Kit App, pass resource Descriptor info
    }
    else if (item.trim().equalsIgnoreCase("Video Camera"))
    {
       //Start Video Camera App, pass resource Descriptor info
    }
  }
}

/*
* Display the resources information
*/
void lstResources_ActionPerformed(java.awt.event.ActionEvent
       event) {
  try { //Use MARE.ResourceDescriptor to parse resource
       description.
    txtResourceDescription.setText(new
       ResourceDescriptor((String)list.elementAt(lstResources.ge
       tSelectedIndex())).getValue("Description") );
  }
  catch (Exception e) {
    txtResourceDescription.setText("");
    System.err.println(e);
  }
}

/*
* Update the server to connect with
*/
void serverSelect_ActionPerformed(java.awt.event.ActionEvent
       event) {
  try {
    ServerDialog serverDlg = new ServerDialog(this,"Server
       Name",true);
    serverDlg.txtServer.setText(serverName);
    serverDlg.setVisible(true);
    serverName =  serverDlg.txtServer.getText();
    this.update.Stop();
    new Thread(new Update(this)).start();
  }
  catch (Exception excep) {
    System.err.println("Error restarting with new Server " +
       excep);
  }
}

/*
* Update the displayed list
*/
public void updateList(java.util.Vector newList) {
  lstResources.clear();
  list = newList;
  for (int i= 0; i < list.size(); i++)
    lstResources.add(new
       MARE.ResourceDescriptor(list.elementAt(i).toString()).get
       Value("TYPE"));
}
```

```
/*
* Various windowing operations.
*/
public void windowActivated(WindowEvent e) {}
public void windowDeactivated(WindowEvent e) {}
public void windowDeiconified(WindowEvent e) {}
public void windowIconified(WindowEvent e) {}
public void windowOpened(WindowEvent e) {}
public void windowClosed(WindowEvent e) {}
public void windowClosing(WindowEvent e) {
  update.Stop();
  System.exit(0);
}

  /**
  * Class for listening for updates to available resources.
  */
class Update implements Runnable{
  ERescueApp parent;
  /*
  * Constructor to initialise the interaction class

  public Update(ERescueApp parent) {
    this.parent = parent;
  }

  /**
  * Called to stop listening for updates.
  */
  public void Stop() {
    ……
  }

  /*
  * Called to connect, receive and update parent with resources.
  */
  public void run() {
    ……
  }
}

/*
* Class to modify the server we are connecting to
*/
public class ServerDialog extends Dialog implements
        java.awt.event.ActionListener {
  java.awt.Label lblServer = new java.awt.Label();
  java.awt.TextField txtServer = new java.awt.TextField();
  java.awt.Button okButton = new java.awt.Button();

  public ServerDialog(Frame parent, String title, boolean modal)
        {
    super(parent, title, modal);
    setLayout(null);
    setSize(200,100);
    setVisible(false);
    lblServer.setText("Server:");
    add(lblServer);
    lblServer.setBounds(10,28,75,15);
    add(txtServer);
    txtServer.setBounds(85,28,100,22);
```

```
      okButton.setLabel("OK");
      add(okButton);
      okButton.setBackground(java.awt.Color.lightGray);
      okButton.setBounds(80,50,40,20);
      okButton.addActionListener(this);
    }

    /*
    * Close the dialog.
    */
    public void actionPerformed(java.awt.event.ActionEvent event)
          {
      Object object = event.getSource();
      if (object == okButton)
        setVisible(false);
    }
  }
}
```