Support for Dynamic Trading and Runtime Adaptability in Mobile Environments

Patty Kostkova
Department of Information Science
Institute of Health Sciences
The City University
London, UK
patty@soi.city.ac.uk

Julie A. McCann
Department of Computing
Imperial College, Technology and Medicine
London, UK
jamm@doc.ic.ac.uk

In the business climate an increasing number of people are expected to perform complex work-related tasks while on the move. In some of these systems, the movement of the mobile user to different physical locations results in the volatility of location-dependent information. For example the answer to the question, 'where is the nearest train station?' changes as a mobile user roams around its environment. Therefore, a key requirement of weakly connected mobile users (that is those dialling-in over a mobile network) is the availability of dynamically updated location-dependent information and the support for runtime adaptability to reflect the frequent changes in the physical environment.

Therefore, the need for dedicated support for mobile applications has become important since working while travelling is becoming more commonplace. In order to meet the needs of this type of dynamic mobile application, a software component enabling dynamic runtime matching of services to requests is needed. That is, when a request is made for data or a service, a third party software component must match the request to the services. This software component is known as a Trader. The unique thing about a mobile system's Trader is that it must monitor the environment to detect changes so that it can adapt the system accordingly.

Only recently have improvements in hardware support for wireless computing enabled mobile application requirements to be fulfilled. Key advances in hardware technologies allowing the current boom in mobile computing include: improvements in reliability, speed and coverage of wireless communication, decreasing hardware size/weight, the invention of the colour LCD display, the track-ball and touch-pads, combined with rapid improvements in mobile telephony. The timely combination of these achievements has enabled the widespread use of PDAs – small lightweight transportable computers, designed for specific mobile applications running dedicated software.

In this chapter, we focus on the problem of availability of dynamically updated location-aware information, rather than "classical" mobile problems dealing with the fluctuation in quality of a wireless communication network, or changing degree of connectivity. Therefore, we define a 'mobile application' as a distributed application run by mobile users (e.g., users with portables working while in transit, tourists while sightseeing, taxi-drivers etc.) processing location-dependent information (e.g., a local resource, local 'data', or location-based request) in a changing environment.

This chapter addresses the design of an adaptive architecture facilitated by dynamic trading. This trading system is called MAGNET and is designed to fulfil the requirements of users in frequently-changing environments. In particular, we present a framework for user-customized dynamic trading of services supporting runtime adaptation, and quality of service-based resource description. MAGNET introduces a different approach to information storage and matching to better meet the demands of mobile applications' adaptability. MAGNET is based on a shared information pool that stores application requests and available services. It permits operations to allow information insertion and their withdrawal which can be user-customized and/or location-aware. This we call the *matching function*. In addition, it supports the monitoring of information placed in the pool, and dynamic adaptation to changes detected in the environment.

In the next section, we discuss our motivations, and define the requirements for a dynamic information-sharing infrastructure. Section 2 presents a simple scenario to better illustrate the needs of mobile users. Also, here we provide an introduction into trading terminology and present existing commercial and academic architectures that support trading and adaptability. In Section 3 we will introduce the MAGNET model. Section 4 describes the support for information monitoring in greater depth, while section 5 demonstrates the support for adaptability in a mobile environment. Next, in section 6 we discuss the use of MAGNET on an example of adaptable resource allocation. Section 7 discusses the project's current status and the final section, 8, contains concluding remarks.

1 MOTIVATION

In the last decade we have witnessed significant technological advances in the areas of wireless communication and hardware component design that have fundamentally changed the computing environment. New software engineering techniques such as object-orientation and component-based computing have resulted in more structurally diverse software architectures, where component and system characteristics are frequently changing. For example, the availability of resources can change as new software or hardware enters or leaves the system. Also, the degree of connectivity (to the network) is continuously fluctuating. Furthermore, as mobile computers and laptops can join and leave computing environments (e.g., the local workplace or a remote office that the person is visiting) the hardware configuration effectively changes. As a result of these changes, a new class of application has emerged requiring a support for dynamic information exchange, trading and runtime adaptability.

1.1 Characteristics of Frequently-Changing Environments

Traditionally, computers use rather simplistic resource allocation strategies as the amount of change in the environment is relatively little. That is, resources and their requests are bound for the duration of the processing and rarely un-bound and re-bound to new services. Further, resource allocation in mobile computing has to deal with restricted hard disk space and limited battery life, in addition to the fluctuation in availability and other characteristics of traditional system resources (such as length of printer queue, processor load, network throughput, disk usage). However, these environments are characterised by their course-grained frequency of environmental change, that is frequency of change is in terms of minutes and hours rather that seconds and milliseconds.

Also, computing environments are no longer composed of monolithic code but are becoming finer-grained (for example, a word processor consists of independent components: editor, spell-checker, viewer, etc.). The structure of the computing environment, reflecting this trend toward 'componentisation' (Messer et. al. 1996, Law et. al. 2000), enables applications to tailor the selection and configuration of components, thus allowing composition of customized computing environments. This is essentially what drives the adaptation.

1.2 Characteristics of Dynamic Location-aware Applications

Besides classical resource allocation requests, there are other applications requiring dynamic resource management which rely on the availability of dynamically-updated location and time-dependent information. They include, for example, tourists running guide-like sightseeing information software on PDAs (Distributed Multimedia Group), taxi-drivers using PDAs to navigate to the next destination, or portable mobile users requiring local resources while in transit and in different company offices (e.g., a Web client running on a portable connected by a mobile phone while on the move needs to switch to the fast connection when the portable is plugged into the network in an office). Dynamic resource management also makes it feasible for mobile or non-mobile systems to provide continuous operation, which requires support for hardware upgrades and on-line software updates without the system shutting down.

Owing to recent significant improvements in wireless communication, weakly-connected applications no longer suffer from unreliability in the communication infrastructure (e.g., higher error-rate, frequent disconnections, limited coverage) (Forman et. al. 1994). In addition, as mobile users require location-dependent information it is essential that they be provided with *local information* ('local' meaning the user's current location). The high volatility of the local information encountered by mobile users necessitates dedicated support tailored to their new requirements and needs.

In addition, many Internet applications require type-free data exchange among different platforms via a common middleware infrastructure that cannot be satisfied by traditional database engines, as these offer data formats that are too restrictive. This is because they are typically based on relational (or object data modelling) and relational algebra-based query processing and typically do not provide any means for location and time awareness nor adaptation to changes in these characteristics.

1.3 Requirements for an Adaptive Trading Architecture

To reiterate, as a result of the combination of the above factors, a new type of application has emerged requiring adaptability to ever-changing conditions, which can be addressed at four key levels: dynamic trading, extensibility, dynamic information monitoring, mobile adaptability an scalability. These are discussed below in more detail.

Dynamic Trading

The primary role of the service manager for dynamic and mobile applications is to enable extensible dynamic service trading. In contrast to requesting services directly by *name* (e.g., I want 'printer P'), they should be allocated by the *type of service* they offer, such as a printer, a file system, an available taxi, etc. This implies communication between system components that did not know their identity *a priori.* Therefore, the system must provide a dedicated component, *Trader*, which collects information on services, and dynamically matches requests against demands. By doing this it can establish a communication between components, which it did not need to know their identity in advance. For example, a user in a mobile environment does not request a pre-configured printer P, rather it asks for a device that can offer printing functionality, and the Trader finds the most convenient match. At an application level, a customer calling for a taxi does not typically request a specific taxi by its number plate, rather they request the nearest available cab to their current location.

Extensibility

To achieve full generality, the Trader should not constrain the format or the semantics of information on services and should allow the user to customize the matching process. Therefore, applications can automatically adapt their behaviour to changes in the environment, and can therefore dynamically extend system functionality. This permits *extensibility* in two areas: firstly, existing services and data formats can be extended (new resources, services and user requests can be defined at run-time). That is, a new type of hardware device could be dynamically added an existing system, or new types of applications could be implemented in the same trading framework.
Secondly, the matching process, performed by the Trader, can also be dynamically redefined (that is, service allocation strategies can be user-customized). Therefore, users can dynamically define their preferences for matching, such as – the *nearest* colour printer, however if, when this request was issued, the colour printer is currently busy, an available black and white printer would be accepted. This switch would have been predefined.

In addition, the framework should be designed not only for computer system resource management purposes. It should enable potential utilization for any kind of application that requires dynamic trading of temporally important information, such as a tourist guide sightseeing system, dynamic taxi navigation system, etc.

Dynamic Information Monitoring

To provide up-to-date information on changing system components, a mechanism for the *monitoring* of selected services is required. This could be automated monitoring or manual update of information in the

Trader. A manual alteration is a sequence of operations performed by the components themselves. Automated update is carried out by monitors, independent of the actual components. For example, in order to keep information on a printer's availability in the Trader up to date, every time the printer's queue is changed, the Trader is updated accordingly. This could be performed by the printer itself, or by a third-party monitor.

Mobile Adaptability
Typical computing environments are open distributed systems consisting of communicating components – clients and servers that can join and leave without impairing system's continuity. As the key role of the system is to fulfil requirements of mobile users accessing local resources in various offices, it must support resource configuration and reconfiguration as a result of the frequent arrival and departure of system components. That is, the Trader must allow a mobile user to automatically adapt their laptop configuration to the hardware configuration of the office that they have just entered.

To summarize, the primary role of such framework is to enable user-customized trading for services and allow adaptation to changing environments by supporting constant monitoring, mobile adaptability and scalability.

2 TRADING AND ADAPTABILITY

In this section we will illustrate a typical adaptation situation on a simple scenario. Then, we will define terms for trading in distributed systems and finally, we will briefly describe existing trading and adapting architectures and look closer at their support for runtime mobile adaptability and extensibility.

2.1 Scenario

The recent trend is that more and more employees have to perform work-related tasks while on the move (e.g., on plane, on train) and at home, in addition to working in their offices. Mobile computers (laptops, Palm pilots), allowing this everyday mobility, are often replacing traditional PCs and becoming the only computer many people tend to use.

A typical everyday situation of such a particular salesperson is to work on train on the way to a client, then at the client's office he may give a presentation or perform other business-related tasks, then he travels to his own office, and work the rest of a day there. Finally, in the evening, he may prepare for his next day duties at home. Each situation requires a dynamic adaptation to allow such user to perform all essential tasks at different environments and to reconfigure to the local hardware settings.

Firstly, on train to a client, the user works disconnected from the network or can only rely on mobile wireless communication, then at a client office he needs to adapt to use local resources, e.g., printer, projector, etc. and utilize the local LAN or/and modem connection. At work, he can use all his local resources (e.g., file system, etc.) and can utilize full connectivity via the local LAN. At home, he needs to adapt to his home hardware configuration and can connect only via a modem link.

For simplicity, we will illustrate adaptation of the system to changes in connectivity on two typical applications – a word-processor and a Web client. We will further elaborate on this example in section 6.

In order to better illustrate the support for adaptability, the user needs to adapt to the following three configurations during the day:

1. on train (disconnected from the network) – he has finished his presentation and needs to print out the handouts. Also, he needs to check up-to-date information on the Internet through WWW. Neither of the requests can be fulfilled, as the network is not available.
2. preparing for a presentation at client's office (weakly connected via modem) – here the user can download the data over the Internet, however, printing handouts is not desirable.

3. giving a presentation at client's lecture theatre (fully connected via LAN adapter) – here he can plug his laptop in the network and take the advantage of the fast connection to download the data over the Internet and to print out the handouts.

2.2 *MAGNET: Trading Framework*

Here we briefly introduce the basic trading elements to clarify the concept undertaken in this work. The model is based on the RM-ODP architecture (ISO X.930 1998, ISO Ed 6.4.19 1998), which was tailored to our approach.

Components
Components are basic functional units of distributed systems. They could represent a wide range of system elements, such as hardware resources, abstraction servers (such as file systems), and user-level programs. They can also represent any objects in terms of object-oriented languages or any component existing in a applications' architecture.

Service Interface
Components act as *black boxes* and their functional behaviour is fully described by a service interface that defines services provided to, and services required from other components. Components requiring a service are called clients; components offering a service are servers. There terms are defined for a particular service-interface pair, therefore, a particular component can concurrently play both roles in different interactions. Server-client interaction is defined as *one-to-many*, that is, one server can communicate with many clients over the same interface.

Service Definitions
In addition to the *name* of the service (called interface reference, e.g., port LPT 1), services may also describe a type of service they offer (e.g., printer) and additional characteristics (e.g., colour, HP deskjet 940c, speed 20 pages per minute). A combination of all these parameters is called a *service definition*.

Component Binding
In order to enable an interaction between distributed components, a binding between their interfaces has to be established. This establishes the connection between the client and server to allow an interaction between them.

The Trader
Communication between components in open systems (i.e., one whereby a component can dynamically join or leave the system) requires a third-party component to collect service definitions defined at runtime, perform a matching process and establish the resultant client-server binding.

A Matching Process
The process of finding corresponding requests (expressed in terms of service definitions) performed by the Trader is called a matching process. If service definitions are expressed exactly, the Trader finds an exact match. However, matching of server characteristics include additional constrains, such as quality of service definitions, then it requires parameterisation, defining preferences of particular user. As these are impossible to define for all users *a priori*, the Trader supports user-customisation of the matching process which allows system extensibility and dynamic adaptability.

Example
Figure 1 illustrates binding between three components – the Trader, Client and Server. Darwin, an architecture-description language (Magee et. al. 1995), provides a convenient formalism for defining bindings in distributed systems shown in the figure. A rectangle represents a component, a circle stands for a service interface. A service that is provided is represented by a filled circle, a required service by an empty circle. A line between filled and empty circles represent the binding implemented by a particular communication channel. In Figure 1, Server and Client find corresponding interfaces for the resultant communication using the Trader. Numbers by the lines represent phases in which relevant bindings must be

established. Firstly, Server and Client export their service definitions by binding to the Trader (phase 1). These two steps can be performed in any order. Secondly, when the matching process discovers the required interface reference, the resultant end-to-end binding between Client and Server can be established (phase 2).

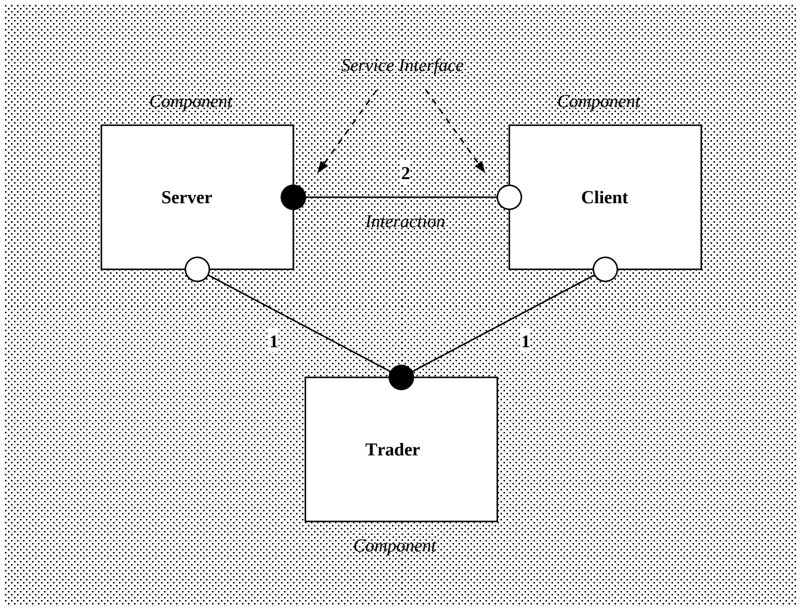

Figure 1: Binding between Server and Client established by the Trader

Federations
In scalable distributed systems Traders must be networked in order to cooperate on providing remote information. The framework should enable a *global trading system* that any Trader may dynamically join or leave. As components often interact on a local scale (such as resource managers within a particular domain), the Trader should primarily support a local trading scheme. However, inter-Trader communication must be also enabled to allow service export for components beyond the current local domain. Trader domains with domain-specific security and propagation information, internetworking with other Trader domains are called *federations*. Passing service definitions across the federation boundary, consequently, must be handled by appropriate communication channel reflecting the "beyond-federation" distribution and security issues (ISO X.930 1998). In Figure 2 a trading system consisting of two federations is illustrated.

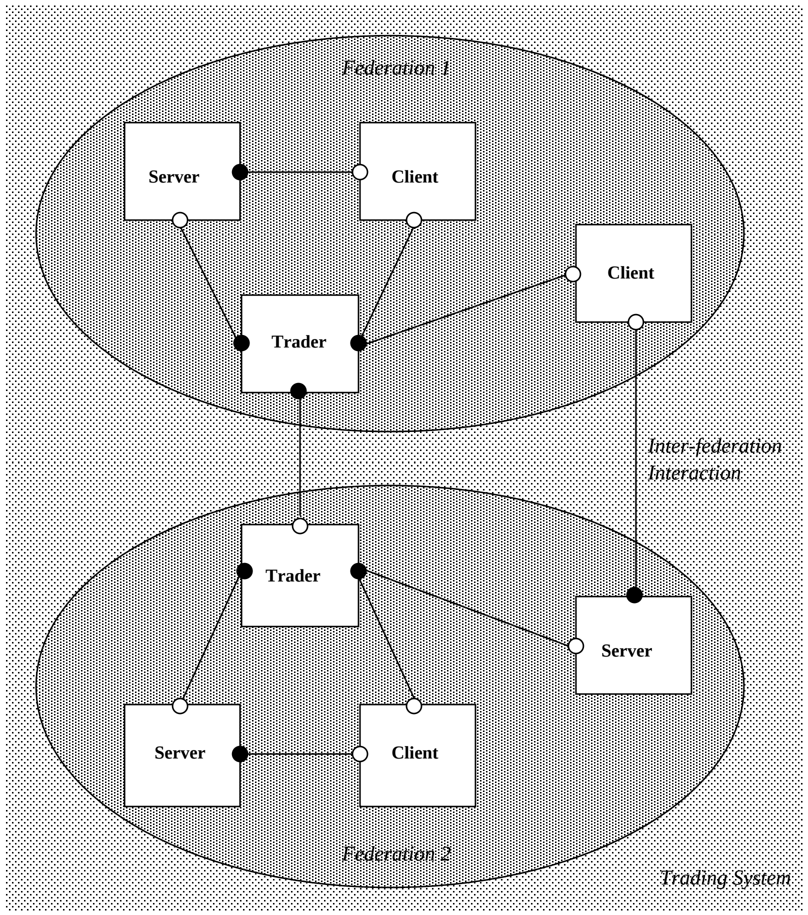

Figure 2: A Trading system consisting of two Federations

2.3 Existing Trading Architectures

In recent years, dynamic issues such as providing greater flexibility, supporting dynamic runtime adaptation or designing loosely coupled communication schemes have been successfully addressed by many research projects and commercial technologies. However, these architectures, discussed above, typically target one issue, rather than providing a unified architecture and therefore are unable to support the set of requirements of mobile applications.

There are many successful projects providing support for dynamic trading and adaptation: tuplespace-based architectures (such as Linda (Gerlernter 1985), Limbo (Blair et. al. 1997), Jini (Waldo 1998), FT-Linda (Guedes et. al. 1995), T Spaces (Web Technologies IBM), Osprey (Bolton et. al. 1993), etc.), and others (such as, Matchmaking (Raman et. al. 1998), Aster (Issarny et. al. 1998)) enable more flexible component coupling based on a parameter description.

Recall that MAGNET consists of an information pool that allows requests and services to be advertised. This has been implemented using a data structure called a tuplespace. The original tuplespace was designed by D. Gelernter at Yale University, with a set of operations (called Linda) enabling tuple manipulation (Gerlernter 1985), Limbo (Blair et. al. 1997) is an example of a distributed system utilizing the tuplespace paradigm in a mobile environment. This framework extends the basic tuplespace architecture to express application-specific requirements, e.g., security. Further, JavaSpaces provide a base for Jini technology

(Waldo 1998) enabling a configuration mechanism for devices to join and leave the network. However, a lookup service provides dynamic binding between clients and servers but is rather restrictive. That is, like Limbo, their types or characteristics cannot describe services and user-customisation is not supported. Consequently, this cannot be used for mobile computing, as the roaming device will not necessarily know the service names *a priori*.

The communication framework FT-Linda (Guedes et. al. 1995) also uses tuplespace to provide fault tolerance; therefore it also does not support the customisation and dynamic flexibility required by mobile systems. Likewise, a Linda-based technology, called T Spaces (Web Technologies IBM), enables communication between applications and devices in a network of heterogeneous computers and operating systems. The architecture presents a rather universal high-level framework; it does not deal with support for particular requirements of applications in a mobile environment. Furthermore, Osprey (Bolton et. al. 1993) also based on the tuplespace paradigm, provides resource allocation. It enables more flexibility by adding semantics into the tuple format. However like most of the systems discussed above, it does not address issues of user-customized matching and extensibility.

Architectures providing service matching based on a special ‘matching’ component, such as Matchmaking (Raman et. al. 1998) (the Matchmaker Component), or Aster (Issarny et. al. 1998) (the Aster Selector) also perform dynamic service coupling. In contrast to MAGNET, they rely on a ‘knowledgeable’ component that decides on component coupling, but can only perform non-customizable matching. In addition, Limbo and Osprey implement tuple-typing in contrast to the universality of our approach.

In addition, similar trends towards a greater degree of flexibility, user customisation and runtime adaptability have been seen at the area of resource management, at operating systems level. Although research results have proven that this is a step in the right direction, the majority of existing systems (except Kea (Veitch et. al. 1998) and DEIMOS (Clarke et. al. 1998)) still lack the support for dynamic reconfiguration, enabling adaptation to changing conditions. In addition, issues such as QoS-based resource allocation enabling application participation, parameterised resource selection (as opposed to name-based allocation) and issues of scalability are still to be addressed. Therefore they are very limited in that they essentially do not provide a framework for an adaptable architecture.

Next, we will present MAGNET dynamic trading architecture, which satisfies the needs for adaptability of applications in dynamic frequently changing environments.. As it has been designed as a dynamic resource architecture that is not restricted to resource allocation alone, it can be used for trading objects (such as CORBA objects (The Object Management Group), DCOM objects (Microsoft Corp.) JavaBeans etc). These platforms such as CORBA, DCOM, etc. would then be able to implement an adaptive architecture at an ‘object level’

3 THE MAGNET ARCHITECTURE

MAGNET is a high-level framework enabling applications in mobile environments to match location-dependent information on services to requests. The full description of the architecture and its usage in dynamic resource management and other application areas could be found in (Kostkova 1999).

The key component of the framework is a *Trader* that collects information on services, records and all application data and dynamically matches requests against demands. It achieves this by essentially performing a user-customized match to couple service offers to user demands. The Trader is based on a modified tuplespace paradigm and the data are kept in a form of tuples.

As one of the key features of the tuplespace is not to constrain the format or the semantics of stored information, but to allow type-free dynamically-defined data to be stored and searched by a user-customized matching process. This provides *extensibility* in terms of enabling new records, service requests and actual services to be dynamically generated but also in terms of customizing the matching process itself. That is, different mobile applications, based on completely different data semantics, could share the same Trader to match their tuples.

Further, to support runtime adaptation and system reconfiguration *dynamic rebinding* is required. That is, the old binding is dropped and a new binding is established in order to better meet application requirements. This may be as a result of client, server or a third party initiation. For example, a mobile client currently using its local disk may wish to join a new, more stable environment in an office to upload data. Therefore it will unbind from its current disk and rebind to the office disk file-system. Information on client demands and service capabilities is maintained either manually (i.e., carried out by the components themselves) or automatically (by a monitoring process).

The stateless nature of tuples saves the pool from having to provide a state-maintaining scheme, for example, check-pointing or recovery procedures. It is assumed that components are fully responsible to maintain its consistency and recover from possible failures which simplifies semantics of the Trader. In addition, stateless nature of tuples improves the generality and reliability of the system. If state is required, it can be incorporated as a parameter of the tuples. Decoupling the server from the client (servers produce tuples of interest to any client) permits communication to proceed anonymously. Consequently, this feature enables free-naming – communication can be established without previous knowledge of the other party's identity.

3.1 The Trader

The Trader is the key component in the MAGNET architecture. The Trader accesses a shared data repository available to all applications and objects represented by components. We call this data structure an *information pool*, its structure is similar to the tuplespace (The information pool is actually a tuplespace. However, the term "tuplespace" is often associated with the Linda distributed programming language (Gerlernter 1985), therefore, we decided to call our data structure 'information pool' to avoid confusion as we do not limited ourselves only to Linda operations).

The Trader consists of three distinctive elements:

1. The information pool (a tuplespace-like data structure),
2. The Trader operations on tuples for their manipulation, and
3. The tuple matching function (an operation providing the actual matching).

Figure 3 illustrates the structure of the Trader, and its three components.

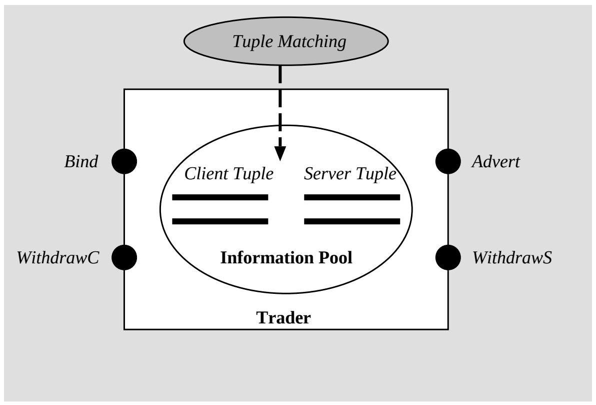

Figure 3: The Trader Structure

3.2 The Information Pool and the Tuples

The information pool is a distributed data structure accessible by all components using MAGNET. Tuples can be inserted into, or withdrawn from, the tuplespace by a set of clearly defined operations. Tuples describing data of mobile components often contain additional information, such as interface references for accessing the component. These are all expressed as tuple elements.

Therefore, the tuple distinguishes between the number of all tuple elements *n* and the number of matching elements *m*. This extension, which we have incorporated into traditional tuple matching, enables the restriction of the matching process to matching the first *m* elements.

A **tuple** T is a set of elements $T=(n, m, p_{1,} p_{2, \ldots} p_{n})$, *n* represents the number of tuple elements and *m* is the number of "matchable" tuple elements p_i are the values of tuple elements i.e., the actual parameters.

For example, to describe a component book Romeo and Juliet by William Shakespeare we may use the following server tuple:

A = (6, 5, 12345, William, Shakespeare, Romeo and Juliet, Penguin, ISBN 654321)

That is 6 tuple elements, 5 of which can be matched*: 12345 (Author ID), William, Shakespeare, Romeo and Juliet, Penguin (publisher). ISBN* is a reference to the book (service) described by this tuple. Naming for interface references is derived from the naming scheme used in the computing or application environment, e.g., ISBN, library identification.

An equivalent client tuple looking up *Romeo and Juliet* would be:

*B = (6, 5, *, William, Shakespeare, Romeo and Juliet, *, reader ID)*

requesting this book published by any publisher (* sign) and ignoring the *Author ID*. This tuple definition incorporates advanced operators, such as *. These are defined in details in (Kostkova 1999).

3.3 The Matching Function

By *matching*, we mean an equality of tuple "matching" elements, or a user-defined "match" enabling quality of service to be taken into account. (However, this is beyond the scope of this chapter, further details can be found in (Kostkova 1999). The remaining elements, "non-matching", containing additional information about the service, such as an interface reference, are not part of the matching process. In this way we have extended the traditional matching, as defined for Linda operations (Gerlernter 1985).

As incorporating non-matching values into tuples is optional, and may differ between a client and a server-tuple, the equality of tuple size is not a required matching condition. Above all, the user-customized matching function, enabling extra flexibility and extensibility of the framework, is a powerful mechanism needed in dynamic frequently-changing environments. This is essentially providing the adaptability.

3.4 The Trader Operations

The information pool has defined operations for manipulation with tuples, such as insert and delete. MAGNET's Trader includes the operations: *Bind*, *Advert,* and *WithdrawC*, *WithdrawS*. These are described below in more detail.

Operation **Bind (T)**, T is a client-tuple. The Trader searches the information pool for a complementary matching tuple. If such a tuple is found, T is returned to the server component (which inserted the matching tuple) without being withdrawn from the pool. If no such tuple exists, the operation results in inserting tuple T into the information pool until a match becomes available and the request is fulfilled.

Operation **Advert (T)**, T is a server-tuple, which is inserted into the information pool. The trader also searches the pool for all complementary matching tuples. If such tuples are found, they are removed from the pool, and returned to the calling server component.

Operation **WithdrawC (T)**, where T is a client-tuple, results in removing tuple T from the information pool; while operation **WithdrawS (T)**, where T is a server-tuple, results in removing tuple T from the information pool.

3.5 Components for the MAGNET Architecture

Figure 4 illustrates the low-level structure of the MAGNET architecture, including components implementing the trading functionality. The system consists of four classes of component: *the Trader, Client, Server* and *Tree* (specific components implementing the matching process.) There is only a single instance of the Trader component per federation in contrast to multiple instances of Client, Server and Tree. In addition there are two types of subcomponent performing dedicated functions: these are a pair of Binders (the *Client-Binder* and the *Server-Binder*) present in all Clients and Servers; and the *GlueFactory* included in all Trees. The GlueFactory hands over a client tuple to the Server to initialise the establishment of the binding carried out by the Binders. Therefore, Binders in cooperation with the GlueFactory establish the resultant client-server binding.

In order to allow the framework to scale, the trader information pool is distributed into federations. Full details of the architecture can be found in (Kostkova 1999, Kostkova et. al. 1999, Kostkova et. al. 2000).

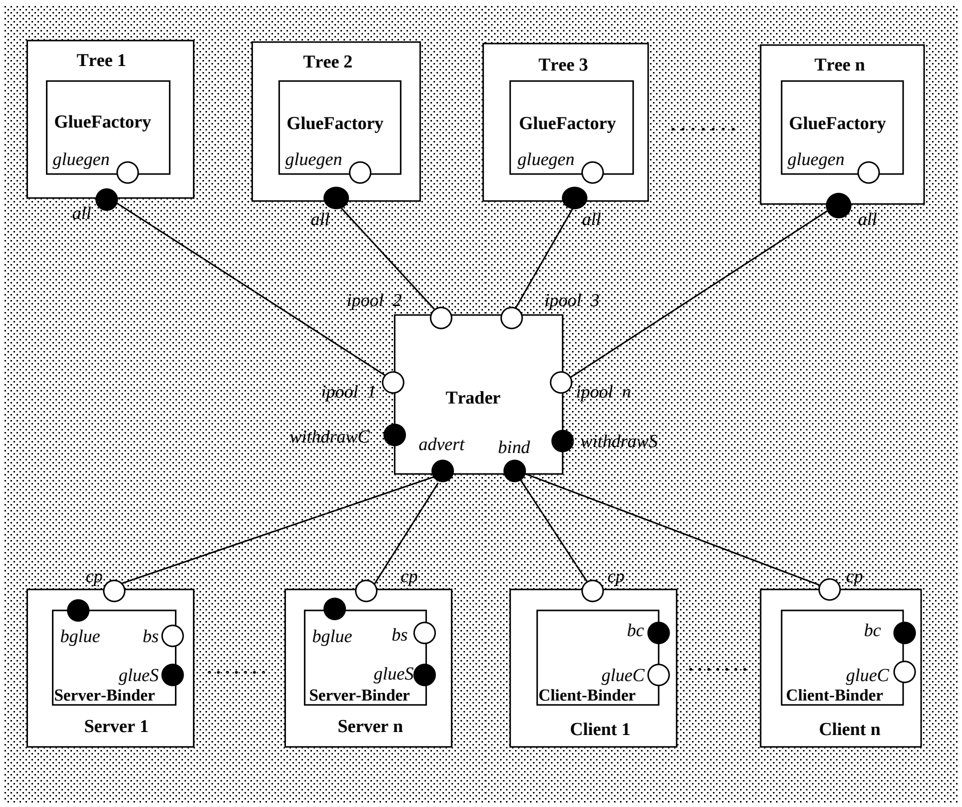

Figure 4: MAGNET's architecture

4 DYNAMIC INFORMATION MONITORING

In order to enable adaptation to changes in system characteristics, service definitions, which are placed in the Trader, must be kept up-to-date. Therefore, MAGNET must *monitor* resource characteristics. In this section, we will describe the semantics of these two components and the actual monitoring process.

4.1 Components for Monitoring

As the MAGNET framework distinguishes between the roles of the client and server, it is necessary to approach their monitoring differently. Therefore, MAGNET has two monitoring components providing this functionality – the *Monitor* and the *Updater* – both application-level components are attached to server or client respectively. They are created together with the components they serve, and are instructed by them to provide component-tailored functionality. A system administrator, not using MAGNET, establishes server-Monitor and client-Updater interactions statically in advance. Here we discuss their interface to MAGNET and expected functionality. The components are illustrated in Figure 5.

4.1.1 The Monitor

The task of the Monitor component is to observe the changing characteristics of the server it is attached to, and keep the server tuple data up-to-date. Tight cooperation with the server enables the Monitor to be informed about current service characteristics, so that it can periodically update relevant tuples in the pool (by removing them and replacing with updated ones).

The granularity of this operation (how often it is performed) depends on the server strategy, in particular on the actual feature being updated, and on the overall character of an application (for example, real-time applications rely on finer-grained updates). Also, it depends on the 'out-of-dateness' accepted (how much can a tuple in the pool differ from current characteristics). However, in accordance with our assumptions, we expect the monitoring to be performed with frequency of minutes, rather than seconds and milliseconds.

An example update would be the printer tuple that represents a printer that was moderately used and which becomes heavily used. This is useful for a client that requires a printout rather soon.

Technically, the Monitor supervising service provision, keeps performing a sequence of operations; *WithdrawS* and *Advert* respectively. From the Trader's point of view, monitoring is performed transparently, indistinguishable from a sequence of operations *WithdrawS* and *Advert* performed by the server itself.

4.1.2 The Updater

As there are not many clients requiring rebinding after having found a requested service, the monitoring of client requirements is less crucial. Also, client-tuples do not reside in the pool (if a match was found), and therefore there is no need to keep them up-to-date. However, clients in systems with frequently changing characteristics may rely on a guaranteed level of service (e.g., network throughput). For those, adaptation to change in conditions is unavoidable (e.g., switching to lower-quality audio and video, etc.) For these reasons, the framework must also provide equivalent support for monitoring clients.

The Updater is a dedicated component instructed by the client it is attached to. It searches the pool for a tuple meeting the client's *current* requirements more precisely, or looks for a different tuple if the client's requirements have changed (e.g., mobile users on the move need to update a requirement for the nearest server, etc.) In this case, the initiative is on the Updater component, in contrast to the Monitor that acts only when invoked by the server.

The monitoring of the information pool is not the only function of the Updater. As changes might result in rebinding the client to a new server, the primary functionality of the Updater is to assist in this third party rebinding. For example a client might want access to the nearest network cell, but as the cell's bandwidth lowers the client may wish to unbind from that cell and move to a less congested one.

Technically, the Updater calls the operation *Bind* on a tuple with higher requirements or performs *WithdrawC* and *Bind* operations when the requirements of the client have changed. The bind-tuple, inserted by the Updater, waits in the pool until it finds a match.

According to the Updater protocol and the 'stage' of client interaction, the Updater decides if rebinding is beneficial (rebinding of a client close to finishing might not be beneficial, taking the overhead of the rebinding process into account). Therefore, the new server tuple can be ignored, or client rebinding can be performed. Details of the rebinding issues are beyond the scope of this chapter but can be found in (Kostkova 1999).

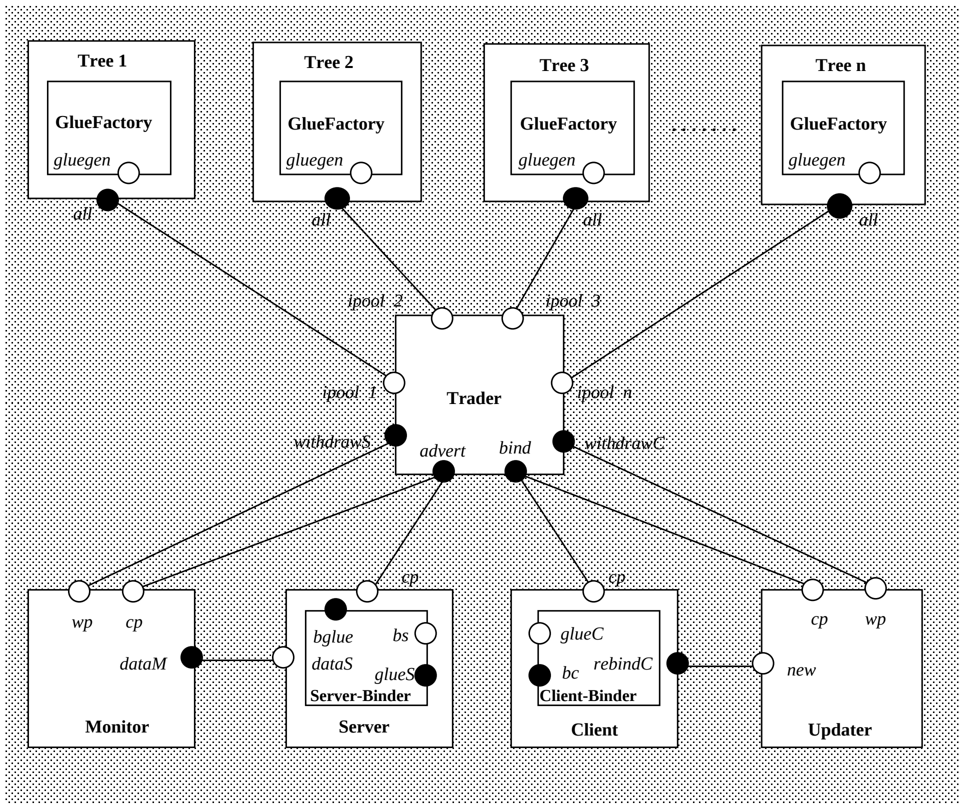

Figure 5: The architecture with the Monitor and the Updater

5 MOBILE ADAPTABILITY

The majority of computing systems are based on scalable, connected 'domain-size' units – such as the Internet with its domains, cellular phone network divided into cells, etc. Likewise MAGNET, designed to support the topology of existing computing systems, consists of connected domain-like units – *federations,* discussed in section 2.2

In this section, we describe *dynamic reconfiguration* enabling computers to join and leave the system at runtime and will discuss adaptability to changes in system configuration. Adapting federations to new configurations, such as when a mobile client with a laptop connects to office resources, requires a join of the mobile federation and the office-based one. Also, MAGNET must support the interconnection of Traders that can enable inter-federation communication – that is the laptop applications can user the resources in the office-based federation.

5.1 Dynamic Adaptation

This section describes how clients temporarily join a local federation where they arrive. This operation needs a special type of support, as users need not know the identity of the Trader they want to connect to. MAGNET provides this support by operations *Join* and *Leave*. The semantics of these operations is targeted to users joining a local site *temporarily* (e.g., mobile users) in order to use its services (such as printer, scanner, file system, etc.). This presumption leads into three design decisions

- operations *Join* and *Leave cannot be transparent* (therefore, each portable client waiting for a resource has got the right to decide whether it is able to accept a resource from an office-based site. This is necessary to avoid mis-allocations; such as a disk space allocated in the office-based computer will be useless when the portable is disconnected).
- Consequently, clients and servers *do not act symmetrically* – portable clients might take advantage of the portable being temporarily on-line by using the office-based services (this is the main goal of the operation), while portable servers will not offer their services to an office-based clients for two reasons. Firstly, the connection is assumed to be temporary, and secondly, the portable computer resources are typically very limited to be offered to other clients. Therefore, the operation *Join* is designed as 'one-way' – the office-based resources are offered to the portable clients, but not vice-versa.
- As client components are responsible for deciding on the usage of office-based resources, they are also responsible for a *maintaining consistent state* when the portable is disconnected from the office-based site. Consequently, every portable component using the office-based Trader is responsible for withdrawing all inserted tuples in order to leave the office-based information pool up-to-date and consistent.

We assume that the communication channel can be established between the portable computer and the office-based domain in the same way as within a single federation. It is the responsibility of the portable applications and the office-based site administrator to ensure that the inter-federation interconnections can be physically achieved. As the join is only temporary, full Trader connection necessitating the merging of information pools is not required.

Now we define operations *Join* and *Leave* – we assume two local trading systems, a portable and an office-based domain, each consisting of one Trader.

5.1.1 Trader Connection

In order to perform the operation *Join*, an identification of the local Trader is not necessary as it is not known in advance which site the portable will be plugged into.

The Trader component can also participate in resource management provided by MAGNET. In order to be able to take part in dynamic binding, it must contain the *Trader-Binder* subcomponent (although, its functionality slightly differs from traditional Binders). To perform the *Join* operation, the office-based Trader offers its information pool to the portable Trader by calling an operation *Advert* inserting a tuple *T1=(2,1,join,bglue)* into the portable information pool. The only matching element in this specific tuple is the third element, "join" (according to the definition of the matching process). As Traders are being connected, this primary binding is established statically by a system administrator (human or automated, such as support for plug&play Ethernet cards). This does not use MAGNET as the Traders cannot be used for connecting themselves together.

5.1.2 Operation Join

A portable client requesting a service that might be fulfilled by office-based site servers when the portable is temporarily connected inserts *three* bind tuples into the portable pool. These are: the classical bind tuple *C1* defining the request (such as a printer request), the second tuple of the form: *C2=(n,1,join,C1)* encapsulating the actual tuple *C1* tuple and having one matching element, keyword "join", and the third tuple *C3=(n,1,leave,C1)* which will be used for disconnection, again, this one has only one matching element, the keyword "leave".

When the portable is connected to an office-based domain by inserting the office-based Trader tuple *T1=(2,1,join,bglue)* into the portable information pool by the system administrator, and a matching between *T1* and *C2* can be achieved. *Bglue*, the office-based Trader's interface reference, is used to connect to the Trader-Binder as illustrated in Figure 6.

As tuple matching between *T1* and *C2* does not differ from any other client-server tuple matching, the operation is performed as usual (the particular GlueFactory binds to the office-based Trader and passes *C2* tuple to the office-based Trader-Binder). Nevertheless, the operational semantics of the Trade-Binder differs: instead of establishing a binding between the office-based Trader and the client, it retrieves the tuple *C1* (the actual client request) from the received tuple *C2* and reinserts it into its information pool by calling operation *Bind*. This operation features a recursion. The client tuple is handled as an ordinary local tuple in the office-based information pool – if a matching server-tuple is available, an inter-federation binding is established.

Figure 6 illustrates operation *Join*, for reasons of simplicity, we omit the Tree components.

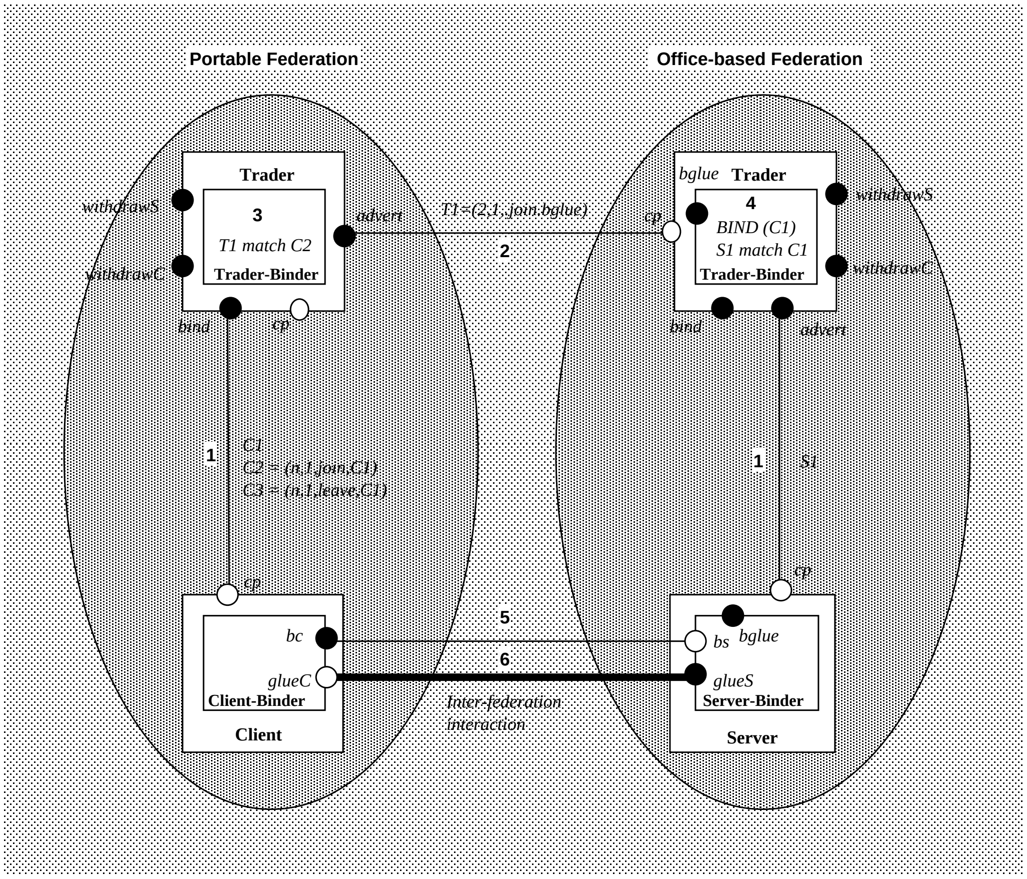

Figure 6: Operation Join

5.1.3 Trader Disconnection

Disconnecting the portable from the office-based site must return the system to a consistent state. Firstly, the ‘connecting’ office-based Trader’s tuple *T1* is withdrawn from the portable’s information pool by operation *WithdrawS* called by the administrator. Client tuples waiting to be served in the office-based information pool must be also removed. As the local Trader cannot distinguish between an office-based client-tuple and a portable client-tuple, the client components themselves must perform the withdrawal. In order to inform them that their tuples should be removed from the office-based pool, another local Trader advert-tuple is inserted into the portable information pool: *T2=(2,1,leave,bglue2).*

5.1.4 Operation Leave

Inserted *T2* tuple matches with waiting client tuple *C3=(n,1,leave,C1)* which is passed to the office-based Trader-Binder over an binding established between particular Glue Factory and *bglue2* service interface, tuple *C1* is extracted from *C3*, and operation *WithdrawC* on *C1* is performed does not succeed – it means an office-based server is already serving the client. However, all clients must be informed. If it about the disconnection, therefore the office-based Trader-Binder establishes a binding with them (between *bs* and *bc* obtained from the tuple *C1*) and notifies them. Client-Binders participating in inter-federation binding (in particular, the service interface *bc*) must handle this additional functionality. That is the ‘notification’ about disconnection results in: the termination of the client-server binding (if the client was bound to an office-based server), or the reinsertion of tuples *C2* and *C3* into the portable information pool (if the client was still waiting to be served).Portable client components having finished their communication with local servers have already removed all their tuples, therefore no tuples can be left behind.

Figure 7 illustrates this operation. There are two clients (Client1 and Client2) using the option of the portable being temporarily connected to an office-based site. Client1 (described by tuples *C1*, *C2* and *C3)* is being served by an office-based server (Server), while Client2 (described by tuples *X1, X2* and *X3*) is still waiting. The notification about disconnection from the office-based Trader results in different actions: Client1 must terminate its binding with Server, while Client2 just reinserts its 'connecting' tuples *X2* and *X3*.

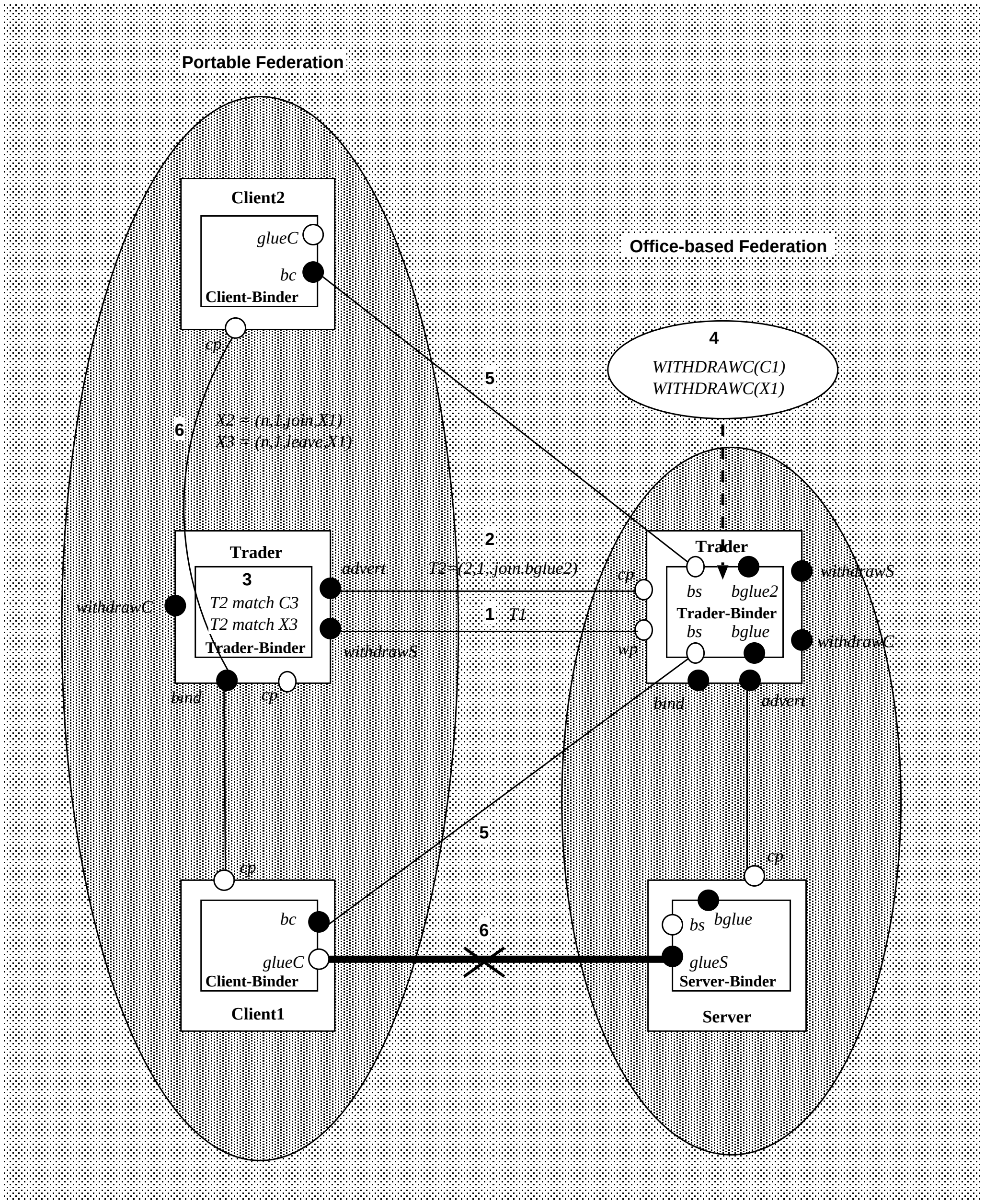

Figure 7: Operation Leave

6 CASE STUDY: NETWORK CONNECTIVITY ADAPTATION

In this example, which we outlined in our scenario (section 2.1), we demonstrate the dynamic capabilities of the MAGNET architecture in terms of adaptability, dynamic rebinding and monitoring, as discussed in previous sections. Dynamic network connectivity is a good example of an application requiring adaptation. To summarize, the system must be able to adapt its network connection from disconnected operation (on train), through weakly connected (at clients' office), to fully connected (at client's lecture theatre). Again, we will illustrate how the system adapts to changes in connectivity on two typical applications – a word-processor requiring a printer and a Web client requiring a Web server. We can see how MAGNET populates its information pool in the three cases (disconnected, weakly connected and fully connected) and illustrate the support for adaptability to changes in configuration.

6.1 Disconnected Case

As was discussed in our scenario, the user is on train, disconnected from the network, working on his presentation that needs to be printed out. Also, he needs to look up information on the Internet. On train, neither of the requests could be satisfied, as the network is not available.

In this disconnected situation there are two applications running on a disconnected portable computer – a word-processor and a Web client. As there is a MAGNET system installed, both applications are represented as components. In this instance, the word-processor requests a printer and the Web Client requests a Web Server. Therefore, the client-tuples (*PRINT*, and *WebClient*) are inserted into the Trader to represent these requests:

Printer	a request for a laser printer with resolution of 600dpi, any speed, attached to any processor serving as print server: *PRINT = (8, 7, CPU, *, *, printer , laser, 600, *, ref)*
WebClient	Web client requests a Web server, as this is an abstraction server, specification of hardware (such as a processor), are not necessary. *WebServer = (3, 2, WebS, modem, ref)* . The fourth element, *modem*, represents the required hardware device connecting the Web client with the Web server. This feature will be discussed in detail bellow.

At this point, their requirements cannot be fulfilled, as there is no printer component or Web server connected to the system to match the client tuples.

Table 1 illustrates the information pool for the described system configuration. Again, the table is schematic; implementation issues are omitted. Also, for reasons of simplicity, only components featuring in this example are illustrated.

Server-Tuples	Client-Tuples
	*PRINT = (8, 7, CPU, *, *, printer , laser, 600, *, ref)*
	WebClient = (3, 2, WebS, modem, ref)

Table 1: The Portable Information Pool - the disconnected case

6.2 Weakly Connected Case

In this next example, the user has arrived to the client's office and is working his portable, which is weakly connected by a modem. Here, the *WebClient* could match *WebServer* and the user could download up-to-date data form the Internet. Below, we describe the establishment of a binding between the Web Client and the Web Server. As he is not interested in printing the handouts over the slow modem connection, the *PRINT* tuple keep waiting.

The word-processor wants to take the advantage of the portable being connected to the Internet later on, therefore, it inserts the 'joining' and 'leaving' tuples to the portable Trader:

*PRINT2 = (12, 2, join, network, 8, 7, CPU, *, *, printer , laser, 600, *, ref)*
*PRINT3=(12, 2, leave, network, 8, 7, CPU, *, *, printer , laser, 600, *, ref)*

To distinguish the hardware device and consequently the type of link connecting the computers, the fourth tuple element (*network*) expresses this information (other option would be *modem*, etc.)Similarly, the Web Client also wants to take advantage of the portable being connected to the network, but it can also operate over modem, therefore it inserts following two tuples:

WebClient2 = (7, 2, join, modem, 3, 2, WebS, modem, ref)
WebClient3 =(7, 2, leave, modem, 3, 2, WebS, modem, ref)

Unlike the word-processor, in this example the Web Client can be connected to the server by any network hardware. In this example, transmitting Web pages over a mobile line is feasible, however, printing a job at a printer in a remote office is not desired. Hence, the printer tuple does not require the network option.

6.2.1 Trader Connection

The portable, being equipped with a mobile phone and a modem, can be weakly connected to an office-based server. In order to enable portable applications to use resources from the office-based server, the operation *Join* must be performed. Therefore, a 'joining' tuple *T1=(3, 2, join, modem, bglue)* representing a reference to the office-based Trader is inserted into the portable Trader. Table 2 illustrates the information pool with all the client tuples and the 'joining' tuple T1 inserted. Again, the table is schematic; implementation issues are omitted.

Server-Tuples	Client-Tuples
T1=(3, 2, join, modem, bglue)	*PRINT = (8, 7, CPU, *, *, printer , laser, 600, *, ref)*
	WebClient = (3, 2, WebS, modem, ref)
	*PRINT2 = (12, 2, join, network, 8, 7, CPU, *, *, printer , laser, 600, *, ref)*
	*PRINT3 = (12, 2, leave, network, 8, 7, CPU, *, *, printer , laser, 600, *, ref)*
	WebClient2 =(7, 2, join, modem, 3, 2, WebS, modem, ref)
	WebClient3=(7, 2, leave, modem, 3, 2, WebS, modem, ref)

Table 2: The Portable Information Pool – the weakly connected case

Before we describe the remaining steps of the operation *Join*, we have to define resources available in the office-based information pool. For reasons of clarity, we consider only the resource requested in our example: a laser printer and a Web server.

That is, the laser printer component with resolution 600dpi and speed of printing 40 pages per minute is described by the server-tuple:

PRINTER= (8, 7, CPU, Pentium, 300, printer , laser, 600, 40, ref)

Recall that the processor running the Web Server can communicate via two network links – it can use a modem port or a LAN adaptor connected to the Internet. Therefore, the Web Server component offers two service interfaces according to the network medium. They are described by server-tuples, WebServer1 and WebServer2:

WebServer1 = (3, 2, WebS, network, ref)
WebServer2=(3, 2, WebS, modem, ref)

Table 3 illustrates the configuration of the office-based information pool before the portable computer dialled in.

Server-Tuples	Client-Tuples
PRINTER = (8, 7, CPU, Pentium, 300, printer , laser, 600, 40, ref)	
WebServer1 = (3, 2, WebS, network, ref)	
WebServer2 = (3, 2, WebS, modem, ref)	

Table 3: The Office-Based Information Pool

6.2.2 Operation JOIN

Next, we can return to the remaining steps of the *Join* operation – in the portable information pool, the sever-tuple *T1* matches with a client-tuple *WebClient2* , the office-based Trader-Binder obtains the client-tuple, removes the encapsulated tuple *WebClient=(3, 2, WebS, modem, ref)* and reinserts it into the office-based information pool. It matches with the Web server tuple *WebServer2 = (3, 2, WebS, modem ,ref)* and a resultant binding between the Web Client and Web Server over a modem can be established. However, the second application, the word processor, remains waiting due to no match being available.

6.3 Fully Connected Case

In the final stage of our scenario, the user is giving a presentation at client's lecture theatre. Here, his laptop is fully connected via a LAN adapter, so the WebClient can adapt the current binding to the WebServer to utilize the faster connection and the handouts could be print out on the available laser printer.

The Web Client is currently communicating with the Web Server through a modem. If a LAN adapter connects the portable to a network, the Client requires to be rebound in order to take advantage of the faster connection. A dedicated Updater component instructed by the Web Client is inserted into the system to perform this task – to monitor the pool and perform the adaptation. The Updater inserts following two tuples into the portable's information pool:

WebClient4 = (7, 2, join, network, 3, 2, WebS, network ,refU)
WebClient5 = (7, 2, leave, network, 3, 2, WebS, network, refU)

These tuples refer to an service interface of the Updater component (*refU*), in contrast to the original Web Client reference (*ref*), as it acts as a third-party in the rebinding process.

6.3.1 Trader Connection

When the portable is plugged into the network by its Ethernet card. A connecting tuple *T2=(3, 2, join, network, bglue)* is inserted into the portable information pool to enable portable applications to use all available office-based resources. At this stage, the portable information pool (illustrated in Table 4) contains two joining server-tuples (*T1* and *T2*), all previous client-tuples (*PRINT, WebClient, PRINT2, PRINT3, WebClient3*), except the tuple *WebClient2* which has been removed when the WebClient was weakly connected to the WebServer. In addition, two tuples from the Updater *WebClient4* and WebClient5 for 'join' and 'leave' have been inserted.

Server-Tuples	Client-Tuples
T1 = (3, 2, join, modem, bglue)	*PRINT = (8, 7, CPU, *, *, printer , laser, 600, *, ref)*
T2 = (3, 2, join, network, bglue)	*WebClient = (3, 2, WebS, modem, ref)*
	*PRINT2 = (12, 2, join, network, 8, 7, CPU, *, *, printer , laser, 600, *, ref)*
	*PRINT3 = (12, 2, leave, network, 8, 7, CPU, *, *, printer , laser, 600, *, ref)*
	WebClient3 = (7, 2, leave, modem, 3, 2, WebS, modem, ref)

	WebClient4 = (7, 2, join, network, 3, 2, WebS, network, refU)
	WebClient5=(7, 2, leave, network, 3, 2, WebS, network, refU)

Table 4: The Portable Information Pool – the fully connected case

6.3.2 Operation JOIN

The server-tuple *T2* matches two client-tuples: *PRINT2* and *WebClient4* . Both tuples are sent to the office-based Trader-Binder which retrieves the original printer tuple: *PRINT= (8, 7, CPU, *, *, printer , laser, 600, *, ref)* and the Updater tuple *(3, 2, WebS, network, refU)* which does not represent a request, as it is used for searching for better service for rebinding. *Both* tuples are reinserted into the office-based information pool (see Table 3) by the operation *Bind*.

The printer tuple is matched against the waiting server-tuple *PRINTER*, so it is not inserted into the pool, but an inter-federation binding is established between the word-processor and the printer. As a result of this, the word-processor job can be printed.

The Updater tuple case is more complex: the extracted tuple *(3,2,WebS,network,refU)* matches the waiting server-tuple *WebServer1* (see Table 3). However, when the binding between the Web Server (by the network port) and the Updater is established, the Updater performs all the steps necessary for the third-party rebinding to adapt the WebClient to the faster connection. Finally, the resultant binding between the WebClient and WebServer takes place using the network.

7 ASSUMPTIONS AND PROJECT STATUS

In designing MAGNET we assume all system components maintain their own *consistency*. That is, we assume that rebinding can be performed only when the system is in a safe state and that when a component has finished its operation it must leave MAGNET in a consistent state. Similarly, components are responsible for the validity of their tuples.

Further, user defined functions are assumed to be *secure* in that they return control back to the Trader. To overcome this we would have to extend the trader's functionality to finish any matching function by force after a timeout period. Also, we assume that unambiguous *naming schemes* are used. However, this can be derived from common naming schemes, such as IP addresses.

As for *performance*, the estimated numbers of components in are in the region of tens and they have the potential to generate tens to hundreds tuples placed in the Trader. Likewise, the number of concurrent components accessing the Trader at one time are estimated to be in the region of tens. A higher number of components can result in the Trader becoming a bottleneck. A possible solution would be to implement the information pool in distributed shared memory.

7.1 Implementation

MAGNET has been implemented in Regis (Crane 1997), an environment for constructing distributed systems. The tuple is implemented in C++ as a high-level base-class (Tuple) comprising the tuple size, the tuple matching size, and encapsulating the tuple-elements. All standard and user-defined tuple-element classes are inherited from a base tuple-element class. Trees contain tree data structures supporting the search (and matching) for non-parameterised requests. The complexity of the Trader operations was calculated and was found to be linear to the number of tuple matching elements. The Trader is responsible for the efficient distribution of tree data structures over Tree components.

The matching function is implemented as an overloaded member function of tuple-element classes inherited from the base class. A tuple-element type matches only the same type, and the "equality" of values can be re-defined according to the type.

As the focus of the architecture is to provide dynamic features, such as runtime adaptability, user-customisation and flexibility, the implementation results cannot be described in terms of performance. However, critical analysis of various features of the framework can be found in (Kostkova 1999)

In addition, the MAGNET architecture also supports advanced QoS support. The extensibility of the framework allows applications to define and negotiate services using QoS characteristics. However, support for QoS is beyond the scope of this chapter. Further details of our QoS model, its design and implementation can be found in (Kostkova 1999).

8 CONCLUSION

This chapter has targeted a fundamental problem of mobile users requiring their computer systems to adapt to them moving to various locations and wishing to access various services at those locations on demand. This requires that the system is aware of dynamically-updated location-aware information. We have argued that the problem has become crucial, owing to a combination of recent improvements in wireless communication, and advances in hardware technology. As a result of these fundamental changes, there is a new class of application requiring type-free data storage, and dynamic user-customized matching of service offers and client requests. These applications need extensibility, and support for runtime monitoring, and adaptability to ever-changing system conditions.

As traditional resource management and trading systems do not provide support for all aspects of this type of mobile application, we have designed and implemented a novel tuplespace-based framework, MAGNET, allowing dynamic user-customized trading of service information and user requests in frequently changing mobile environments. This extends the notion of the tuplespace paradigm to provide a universal solution, which is not tied to a specific application domain. In addition to user-customized trading for services, MAGNET enables adaptation by supporting constant monitoring of computing environment, mobile adaptability and is able to scale to mobile computers dynamically joining and leaving the system. We illustrated how MAGNET meets the specified requirements by a case study – switching network connectivity.

References

Blair G. S., Davies N., Wade A. P. *Quality of service support in mobile environment: an approach based on tuple spaces*. The 5th IFIP International Workshop on Quality of Service, New York, USA, May 1997.

Bolton D., Gilbert D., Murray K., Osmon P., Whitcroft A., Wilkinson T., Williams N.. *A Question based approach to Open systems: OSPREY* Internal TR, SARC, City University, London. March 1993.

Clarke M., Coulson G. *An Architecture for Dynamic Extensible Operating Systems*. In the Proceedings of the 4th International Conference on Configurable Systems, pages 145-155, Annapolis, Maryland, USA, May 1998.

Crane J. S. *Dynamic Binding for Distributed Systems*. PhD thesis, Dept. of Computing, Imperial College, London, UK, 1997.

Distributed Multimedia Research Group. *ABTA: The Active Badge Tourist Application*. Computing Department, Lancaster University, Lancaster, UK. Electronic document available at http://www.comp.lancs.ac.uk/computing/research/mpg/most/abta_project.html

Engler D. R., Kaashoek M. F., O'Toole J. W. Jr. *Exokernel: An Operating System Architecture for Application-Level Resource Management*. In Proceedings of the 15th ACM Symposium on Operating Systems Principles, pages 251-266, Colorado, USA, December 1995.

Forman G. H., Zahorjan J. *The Challenges of Mobile Computing*. IEEE Computer, 27(4), pages 38-47, April 1994.

Gelernter D. *Generative Communication in Linda*. ACM Transactions on Programming Languages and Systems, 7(1), pages 80-112, January 1985.

Guedes D. O., Bakken D. E., Bhatti N. T., Hiltunen M. A., Schlichting, R. D. *A Customized Communication Subsystem for FT-Linda*. In Proceedings of the 13th Brazilian Symposium on Computer Networks, pages 319-338, May 1995.

Issarny V., Bidan C., Saridakis T. *Achieving Middleware Customization in a Configuration-Based Development*. In Proceedings of the 4th International Conference on Configurable Distributed Systems, pages 207-214, Annapolis, Maryland, USA, May 1998.

Kostkova P., McCann J.A. *MAGNET: An Architecture for Dynamic Resource Allocation.* Proc. of International Workshop on Data Engineering for Wireless and Mobile Access, ACM, August 1999.

Kostkova P., McCann J.A. *Inter-federation Communication in the MAGNET Architecture.* published in the Third Grace Hopper Celebration of Women in Computing, Mass. USA., September 2000

Kostkova P. *MAGNET: Dynamic Resource Management Architecture*. PhD Thesis. Dept. of Computing, The City University, London, March 1999.

Law G., McCann J. A. *Decomposition of Pre-emptive Scheduling in the Go! Component-Based Operating System*, ACM SIGOPS European Workshop, 2000.

Magee J., Dulay N., Eisenbach S., Kramer J. *Specifying Distributed Software Architectures*. Fifth European Software Engineering Conference, Barcelona, September 1995.

Messer A., Wilkinson T. *Components for Operating System Design.* Proceedings of the 5th IEEE International Workshop on Object-Orientation in Operating Systems (IWOOOS '96), Seattle, USA, October 1996.

Microsoft Corporation. *DCOM Technical Overview*. Electronic document available at http://www.microsoft.com/com/DCOM.asp

Raman R., Livny M., Solomon M. *Matchmaking: Distributed Resource Management for High Throughput Computing*. In Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing, Chicago, IL, USA, July 1998.

Reed D., Fairbairns R. *Nemesis: the kernel. Overview*. University of Cambridge, Computer Laboratory. Cambridge, UK, May 1997.

Secretariat. ISO/IEC JTC1/SC33. Standards Association of Australia, PO Box 1055, Strathfield, NSW, Australia 2135. *Processing – Interface References and Binding*. January 1998. Document ISO/IEC JTC1/SC33 N119, ITU-T Draft Recommendation X.930, 1998.

Secretariat. ISO/IEC JTC1/SC33. Standards Association of Australia, PO Box 1055, Strathfield, NSW, Australia 2135. *Information Technology – Open Distributed Processing – Trading Function*. January 1998. Document ISO/IEC JTC1/SC21 N10979 Ed. 6.4.19, 1998.

The Object Management Group, OMG Headquarters, 492 Old Connecticut Path, Framington, MA 01701, USA. *The Common Object Request Broker: Architecture and Specification*, July 1995. Version 2.0.

Veitch A., Hutchinson N. *Dynamic Service Reconfiguration and Migration in the Kea Kernel*. In the Proceedings of the 4th International Conference on Configurable Systems, pages 156-163, Annapolis, Maryland, USA, May 1998.

Waldo J. *Jini Architecture Overview* http://www/javasoft.com/products/jini/whitepapers/architectureovervoew.pdf, Sun Microsystems, 1998.

Web Technologies Department of Computer Science, IBM Almaden Research Center, San Jose, CA, USA, Electronic document available at http://www.almaden.ibm.com/cs/TSpaces.