Department of Computing
The City University
London

MAGNET: A Dynamic Resource Management Architecture

Patricie Kostková

July 1999

A thesis submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science at City University, London, UK.

.

Copyright ©1999 Patricie Kostková

Abstract

This thesis proposes a new dynamic resource management architecture, MAGNET, to meet the requirements of users in flexible and adaptive systems. Computer systems no longer operate in centralized isolated static environments. Technological advances, such as smaller and faster hardware, and higher reliability of networks have resulted in the growth of mobility of computing and the need for run-time reconfigurability. The dynamic management of this diversity of resources is the central issue addressed in this thesis. Applications in environments with frequently changing characteristics are required to participate in dynamic resource management, to adapt to ever-changing conditions, and to express their requirements in terms of quality of service.

MAGNET enables dynamic trading of resources which can be requested indirectly by the type of service they offer, rather than directly by their name. A dedicated component, the Trader, matches requests for services against demands and establishes a component binding — resource allocation. In addition, the architecture is extensible — it does not constrain the information on services and allows user-customization of the matching process. Consequently, this allows resource definitions to be parametrized (to include QoS-based characteristics), and the matching process to be user-customized (to preform QoS-based negotiation). In order to fulfill the requirements of users relying on ever-changing conditions, MAGNET enables runtime adaptation (dynamic rebinding) to changes in the environment, constant monitoring of resources, and scalability of the architecture.

The generality of the MAGNET architecture is illustrated with several examples of resource allocation in dynamic environments.

Acknowledgements

Over the last three years, there were many people who have directly, or indirectly, intentionally or accidently contributed to this work having come to fruition. They are all owed my thanks.

First of all, I would like to thank all my official and unofficial supervisors: Tim Wilkinson, Peter Osmon, Steve Crane, Julie McCann and Kevin Murray.

Tim, the 'father' of the component-based resource management idea, gave me expert technical support in various aspects of this thesis, and desperately needed encouragement before public presentations at my first conferences.

Peter Osmon gave me constant encouragement and support in his ever-optimistic style. His guidance and helpful comments, particularly during the writing-up phase of my thesis, are much appreciated.

Many thanks are due to Steve Crane, who appeared in my final year, just in time to suggest Regis for my implementation environment. His perfectionist attitude, insight into the subject, and expert technical support helped me to improve, justify, and better formalize this work.

My thanks are also due to Julie McCann for moral support during all three years, many stimulating discussions, and valuable comments on my research. In particular, I would like to thank her for proof-reading the first draft of this thesis, and her constructive feedback greatly influenced the final presentation of this work.

I would like to thank Kevin Murray for many helpful suggestions during the initial 'shaping' of my thesis theme, and many constructive discussions.

Further, Nick Plumb and Kim Harries were always ready to provide helpful comments, and corrections of Czech-English drafts of my papers. In addition, Nick, Kim, Andy Whitcroft, James Green, Peter Loh and Irena Arambasic were excellent friends who put up with me over the last three years.

Many thanks are due to Nomi Harris, who volunteered to do the final proof-reading of this thesis.

Also, other PhD students and members of staff who provided a stimulating atmosphere in the office or a relaxing time off-site (or both) also contributed; not directly into my reseach, but significantly into my understanding of English culture. In particular, they include: Paul Howlett, Sheun Olatunbosun, Akmal Chaudhri, Shim Young, Chris Marshall, Tony Valsamidis, Gary Mullen, Nick Williams, Maia Dimitrova, Michael Schroeder and Greg Law.

Also, I owe thanks to my lectures from the Faculty of Mathematics and Physics, Charles University in Prague, where I received my Masters degree for preparing me for my PhD research.

Finally, many thanks go to my family and friends back in the Czech Republic. First of all, I have never been able to fully express my thanks to my parents, Daniela and František, for their remote support and constant encouragement during all my life. In addition, my great sister Jana who enjoyed with me many happy times, and supported me when things were difficult. My granny Josefa, with her inexhaustible source of energy and ability to battle against extreme adversity, has always been a source of inspiration for me. Finally, thanks for support from other friends, in particular, Dafe Šimonek, Markéta Starobová, Renata Škopková, Hynek Pikhart, Marek Zindulka and Eva Šimšová.

Thank you all!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The role of resource management has recently changed due to two factors: technological improvements, resulting in a diversity of computing environments, and the inability of traditional operating systems to provide a flexible dynamically-adaptable platform. This thesis addresses the design of a *resource manager,* MAGNET, fulfilling requirements of users in frequently-changing environments. In particular, we present a framework enabling user-customized dynamic resource allocation supporting runtime adaptations, and quality of service-based resource description.

Now we will discuss our motivations in greater detail (section 1.1), outline the high-level requirements for the resource management architecture (section 1.2) and then summarize our contributions (section 1.3).

1.1 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. It has become structurally diverse, with frequently-changing characteristics of system components, such as availability of resources, degree of connectivity, and local site hardware configuration. In addition, traditional operating systems (including microkernels) still suffer from high-level centralized resource management, and the inability to tailor resource abstractions to application needs [13].

1.1.1 Technological Advances

Technological improvements in reliability, speed and coverage of wireless communication, and the rapidly-decreasing size and weight of mobile phones are major factors enabling the current boom in mobile computing. Therefore, weakly-connected systems (e.g., Infra-red (IR) networks, cellular radio networks) no longer suffer from significantly low bandwidth, high error-rates, throughput fluctuations, frequent disconnections or limited coverage [21]. The affordability of mobile phones is another factor that has contributed to the change in the computing environment.

Other hardware technology advances have accelerated this process by enabling users to become mobile. These include: the invention of the colour LCD display, small disks, lightweight batteries, track-ball and touch-pad. The overall size and weight of hardware components has also decreased while their capacity and performance has increased.

The timely combination of these achievements has enabled the development of two types of transportable computers: Personal Digital Assistants (PDAs) such as

Palm Pilots, and 'portable' computers, such as laptops. PDAs are small, lightweight, transportable hand-held computers designed for specific mobile applications running unique software, for example an 'on-line' tourist navigation program [17]. Portables are transportable computers, typically running classical operating systems and applications, and are commonly used when complex work-related tasks are expected to be performed whilst on the move. Their use while in transit (e.g., on train or plane), or when movement is in the nature of the particular business (e.g., travelling salesmen) has become commonplace.

1.1.2 Characteristics of Frequently-Changing Environments

Here we describe four issues illustrating the dynamic nature of frequently-changing computing platforms. Primarily, we address environments with course-grained frequency of changes, that is minutes and hours rather that seconds and milliseconds (e.g., a typical example is a roaming portable user requiring to adapt to local resource configurations in offices where he arrives.) We further elaborate on assumption on our computing environment in chapter 3.

The classical resource allocation problems caused by fluctuation in availability and other characteristics of traditional system resources (such as length of printer queue, processor load, network throughput, disk usage) still remain. Resource allocation for mobile computers has to deal with restricted hard-disk space and limited battery life.

Owing to the enormous growth in wireless communication, resource allocation also has to reflect changes in characteristics of additional resources such as network connectivity of mobile users — the degree of connectivity may vary from totally disconnected, through weakly-connected (by wireless communication, such as, IR networks, cellular radio networks), to fully-connected (by Ethernet or high-speed optical-fibre networks such as FDDI and ATM) [21].

Above all, the mobility of users results in the high volatility of location and time-dependent information, such as local time (related to the user current position, changing while on the move, e.g., on a plane), local site hardware configuration (using local resources in offices where a portable computer is plugged in), and a useable Internet Service Provider (ISP) (according to town, state where a portable user is currently travelling).

Finally, computing environments are no longer coarse-grained and monolithic. System elements at all levels (hardware, software, and data) are becoming finer-grained [45] (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' [47, 45], enables applications to tailor the selection and configuration of required components, and allows composition of customized computing environments.

1.1.3 Limitations of Traditional Operating Systems

Operating systems form the interface between system resources and applications by providing abstractions of hardware devices, protection of applications, and resource management.

Classical operating systems (including microkernels) limit flexibility, performance and utilization of system resources by forcing applications to use inappropriate high-level abstractions, uniform protection schemes and high-level static resource management [13]. Unsatisfactory performance of both the operating system and applications, together with a lack of flexibility and run-time configurability, are the result of forcing applications to use inappropriate system services. Also, it has become clear that the requirements of all applications cannot be met by any

operating system in advance [65]. Therefore, applications require a platform where they can implement their own abstractions, tailor existing servers to their needs, define their own protection schemes, and customize resource management policies.

In addition, applications in environments with frequently changing characteristics impose additional requirements on operating systems, such as the ability to participate in dynamic resource management, and to support adaptation to ever-changing conditions. Due to the trend towards finer granulation [13] of system services (discussed in section 1.1.2), the resource manager's role, as the key component, has significantly expanded.

1.1.4 A New Role of Resource Management

The traditional resource manager operated within a set of pre-defined static policies for the allocation of resources to applications. Resources must have been connected to the system and configured in advance (typically at boot time).

Such resource strategies were sufficient for traditional computing environments, but recent technological improvements have extended the role of resource management. It now has to provide *dynamic resource allocation* strategies and support run-time adaptation to frequently-changing system conditions. In addition, the higher availability of distributed resources together with fluctuation of their characteristics have introduced a resource description specifying non-functional features, known as *quality of service* (QoS). Resource managers have to enable QoS-based resource description and QoS-based allocation policies.

In open systems, the requirement to enable requests for services to be described by a *type of service* (e.g., a printer), rather than directly by a *name* (e.g., the printer lwa) is a problem which is encountered by run-time resource allocators. This implies communication between system components which did not know their identity *a priori*. In addition, dynamic features such as the monitoring of selected resource features, and the provision of location and time-dependent information are also required. These features enable a resource manager to provide dynamic adaptation to variations in system environment.

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 [17] or portable mobile users requiring local resources while in transit and in different company offices [33] (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.

1.2 Requirements for a Resource Management Architecture

In order to design a dynamic resource manager, we need to identify the high-level requirements of typical applications utilizing the potential of frequently changing computing environments.

1.2.1 Dynamic Trading

The primary role of the resource manager for dynamic and mobile applications is to enable extensible dynamic resource allocation. In contrast to requesting resources

directly by *name*, they should be allocated by the *type of service* they offer, such as a printer, a file system, etc. 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 dynamic binding between components which did not need to know their identity in advance.

1.2.2 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. 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). Secondly, the matching process, performed by the Trader, can also be dynamically redefined (resource allocation strategies can be user-customized). Applications can adapt their behaviour to changes in the environment, and can therefore dynamically extend system functionality. However, this relies on the presence of the Trader component providing a framework for these extensions.

In addition, the framework should be designed not only for resource management purposes. It should enable potential utilization for any kind of applications requiring dynamic trading of up-to-date information.

1.2.3 QoS-based Management

The Trader performs component coupling based on the type of service provided or required. However, the extensibility of the architecture enables applications to describe system components in terms of non-functional characteristics of the service, the *quality of service*. For example, these might be static values (such as a printer resolution or speed), or dynamically changing characteristics (such as length of a printer queue, current network throughput, location of a resource regarding the location of a mobile user, etc.)

QoS-based resource description requires user-customization of the matching process, in order to enable applications to define their preferences. Therefore, in addition to basic *exact matching* which is sufficient if requirements are expressed exactly, the extensible Trader also supports *user-customized matching* requests against demands. This is necessary in situations where component characteristics include a QoS-based description, as components must express their preferences in order to define semantics of their matching process.

1.2.4 Dynamic Rebinding

In order to support runtime adaptation to system environment changes, such as the continuous operation of a system during a hardware upgrade or a software update (such as version upgrade), changes to existing bindings must be possible.

Therefore, in addition to the previously discussed dynamic binding of system resources to applications, the framework should also support *dynamic rebinding.* It can be originated by components themselves (*first-party rebinding*), or performed by a managerial third-party with knowledge of overall application semantics (so-called *third-party rebinding* [15]). In addition, rebinding should also enable *first-party renegotiating* (leaving the selection of a new component on the unbound peer), or presenting the rebound component with an appropriate replacement — that might be found either in the Trader (*third-party renegotiation*), or obtained from an external entity (in this case *no-renegotiation* is necessary).

Supporting dynamic rebinding introduces the problem of consistency. The semantics of this operation have to be defined; in particular, the circumstances under

which an existing binding can be broken, and the definition of who is responsible for maintaining end-to-end consistency.

1.2.5 Information Monitoring

To provide up-to-date information on changing system resources, a mechanism for automated periodical *monitoring* of selected services is required, in addition to the manual update of information in the Trader. A manual alteration is a sequence of operations performed by components themselves. An automated update is carried out by monitors independent of the actual components.

1.2.6 Scalability

Typical computing environments are *open distributed systems* consisting of interacting components — clients and servers which can join and leave without impairing system continuity [15].

As our framework is designed for applications running in such open distributed systems, it has to be scalable, in order to permit a larger physical area to be covered, to support mobility of users, and to allow a high number of users to dynamically join and leave the system.

As the key role of the system is to fulfill requirements of mobile users accessing local resources in various offices, it must support resource configuration as a result of frequent arrival and departure of system components.

A design of open systems that enable the architecture to *scale* faces a trade-off between response time and precision of provided information. Therefore, the framework has to define constraints under which scaling becomes feasible, such as constraints on the scale of the matching process.

1.3 Contributions

Contributions of this thesis lie in four areas:

- mapping the field of resource management in current systems and identifying its new role
- a unifying model of a third-party trading, forming a basis for the design of the MAGNET architecture
- a design and specification of MAGNET, a dynamic resource manager

1.3.1 Identifying a New Role of Resource Management

We have mapped the field of operating systems and resource management, and identified the new role of resource managers in dynamically changing adaptive systems.

1.3.2 A Model of Dynamic Third-party Trading

In order to design MAGNET, it was essential to identify entities involved in the resource management process, and specify features supported by the architecture. Therefore, we have elaborated a model of extensible service trading by a third-party, the Trader, enabling component coupling based on the description of a type of service, in contrast to direct name-based requests. The extensibility of the model enables user-customization of the service information and the matching process. The model forms a basis for our design of MAGNET.

1.3.3 MAGNET: A Dynamic Resource Management Architecture

The third contribution of this thesis is the design of MAGNET, a framework for dynamic resource management which aims to satisfy applications running in frequently-changing environments. Based on an information trading model, it provides a set of features which enable powerful dynamic resource management (dynamic trading, extensibility, QoS Management, dynamic rebinding, monitoring, and scalability). We have chosen the tuplespace paradigm for the design of the Trader, as it enables dynamic component matching and extensibility.

1.4 Thesis Structure

In this chapter we have discussed our motivations for research described in this thesis by identifying a new role of a resource manager in diverse computing environments. Based on our motivations, we have elaborated requirements for a dynamic resource management architecture and briefly summarized contributions of this work.

The second chapter analyzes various resource management architectures and extensible operating systems projects, both from academia and industry, that address similar problems.

In the third chapter, we define the terminology used in describing MAGNET and formulate a model of dynamic trading. We present a discussion of the binding process, then describe rebinding issues relevant to this thesis and present approaches to QoS management.

Based on this model, in chapters four and five we describe the resource manager architecture, MAGNET. The former presents the architecture, provides its specification and justification of its components, and describes the binding process. The latter presents advanced features, such as monitoring, QoS management, scalability and the support for rebinding.

Chapter six discusses implementation issues. We describe *Regis*, the computing environment used for implementing MAGNET's prototype, and present the implementation of key features of the architecture.

In chapter seven we describe several examples of MAGNET, as a resource manager, and illustrate how it provides support for dynamic resource allocation and runtime adaptation. In addition, we evaluate the framework by discussing features it provides, and providing comparison with existing architectures.

Chapter eight summarizes contributions of this work, and presents directions for future research in this problem area followed by closing remarks.

Chapter 2

Resource Management in Distributed Systems

Before we discuss the design of the proposed resource manager, MAGNET, we must examine existing resource management architectures in order to present the state of the art in this area (from both academic and industrial environments). We focus on systems providing support for mobile users in frequently changing environments, according to the requirements outlined in chapter 1.

As 'resource management' is a very broad subject which can be addressed at different levels (hardware level, operating systems level, user level, object interaction level, etc.), we will focus on major projects shaping the state of knowledge in this area. We consider resource management in the following three areas: extensible operating systems research (section 2.1), trading and reconfigurable architectures (section 2.2), and support for resource management in quality of service architectures (section 2.3). In addition, section 2.1 presents current trends in extensible operating systems as alternative approaches to the component-based architecture, BITS, proposed in this thesis.

2.1 Resource Management in Extensible Operating Systems

In this section, we will outline current research in operating systems design focusing on *extensible systems*. We discuss major projects, putting extra emphasis on aspects of resource management. However, a full description of the presented architectures is beyond the scope of this work. As MAGNET provides support for dynamic binding and adaptation to changes in computing environment, we will relate our discussion to these issues. We describe in greater detail extensible systems (Exokernel, SPIN, Inferno, Kea, DEIMOS and Nemesis), as they are relevant to BITS, then briefly mention other related architectures (section 2.1.7) and close this section with a summary (section 2.1.8).

2.1.1 Exokernel

Exokernel [19, 31], developed at the MIT Laboratory for Computer Science, is a major extensible system. It demonstrates that the separation of resource protection from management enables application-specific customization of traditional resource abstractions without impacting efficiency.

By pushing the kernel interface closer to the hardware, Exokernel allows greater

flexibility and more efficient user implementation of higher-level abstractions. Exokernel has proven that application-level virtual memory and interprocess communication primitives (IPC) can be implemented in an order of magnitude faster than state-of-the-art implementations [19].

Exokernel presents a flexible computing environment enabling users to build customized applications from available system services.

Resource Management

Exokernel presents an environment where resource management can be implemented at application level by untrusted servers. This is achieved by secure multiplexing of available hardware resources which are exported to library operating systems implementing desired high-level abstractions. Protection is achieved by tracking ownership of resources, using secure binding of applications to machine resources and event handles, and by visible resource revocations.

Exokernel achieves excellent performance by presenting applications raw hardware, and extensibility by enabling high-level libraries to be replaced or customized. However, its focus is more on 'static' issues, rather then on 'dynamic' features, such as adaptation to changes in system configurations, or runtime application-customized resource allocation.

Exokernel is a good example of a platform for a dynamic resource manager, like MAGNET, proposed in this thesis.

2.1.2 SPIN

The SPIN [5, 59] project provides user-level extensions of traditional operating system services by downloading user code into the kernel (user extensions written in a type-safe language are compiled by the kernel compiler and linked to the kernel). Extensible procedure calls, called events [59], can be executed by multiple handlers in response to an event. This technique provides dynamic binding of events to handlers by extended procedure call semantics, such as conditional execution, multicast, and asynchrony.

Although SPIN enables applications to write their own system calls (extensions), it does not allow them to implement their abstractions by accessing resources directly. This approach, together with a considerable additional cost of safety-guaranteeing code, impacts system performance [40].

Resource management

Although SPIN addresses the problem of providing user-level extensions on top of a traditional operating system, its resource management is rather static. A level of flexibility is provided by dynamic binding of events to multiple handlers; however, their dynamic selection is not supported.

2.1.3 Inferno

Inferno [18], an operating system developed at Lucent Technologies and Bell Labs, is a commercial project (a successor to Plan 9 [60]) contributing to the research of distributed services in network environments.

Inferno is designed to support a wide diversity of network environments — such as advanced telephones, hand-held devices, Internet computers, and above all, traditional operating systems. It provides standard interfaces to access system services, and can run as an application on top of a host operation system or on bare hardware.

Inferno applications are developed in Limbo, a module-based concurrent language, which compiles into byte-code, and is interpreted by a virtual machine DIS, enabling wide-range portability for applications and services.

Resource Management

All resources in the Inferno system, both local and remote, are represented by a hierarchical file system; users or processes assemble a private customized view of the system by constructing a file system containing only required resources. This approach provides unification of all system resources, user-customization of the computing environment, and distribution transparency by applying a uniform communication protocol, Styx, to all local and remote resources. However, dynamic features targeted by our architectures are not addressed.

2.1.4 Kea

The operating system Kea [73, 74] provides an environment enabling dynamic binding and runtime rebinding. Having inherited its design from the microkernel (a lightweight abstraction of physical resources), it does not allow runtime extensions and suffers from efficiency problems due to cross-domain procedure calls [73].

Its abstraction is based on the notion of *portals* describing entry-points to domains (virtual address spaces) through which interprocess communication is achieved. Interactions, based on RPCs generated by the Kea kernel, permit the remapping of a portal into a different domain at runtime.

Resource Management

Kea provides dynamic binding of applications to services, and enables runtime adaptations — transparent rebinding to new services (so called, portal remapping). However, application participation in dynamic resource allocation and other features required by users in dynamic environments are not addressed.

2.1.5 DEIMOS

DEIMOS [14], an extensible operating system developed at Lancaster University, addresses the problem of runtime dynamic extensibility and enables applications to build customized execution environments. Applications can load and unload modules on demand. A special module, the *configuration manager*, although itself subject to being unloaded, is responsible for configuration of system resources on application request (described as a system graph in terms of modules and bindings).

Resource Management

System resources, represented as modules, can implement abstractions and a range of protection schemes. The configuration manager enables runtime bindings to be established between system modules on demand, supporting system configuration and on-the-fly reconfiguration.

Although dynamic binding mechanisms supported by DEIMOS are very flexible, they cannot be parametrized. System scalability is limited as the communication manager has to maintain a system graph representing the configuration of the current system.

2.1.6 Nemesis

The Nemesis [64, 39] single address space operating system, developed at the University of Cambridge under the aegis of Pegasus and Pegasus II projects, aims to support time-sensitive applications requiring a consistent Quality of Service, such as those which use multimedia. The Nemesis kernel consists of a scheduler and the Nemesis Trusted Supervisor Code, which is used for Internet Domain Communication and interaction with the scheduler. The kernel also handles memory faults and other low-level processor features. It was driven by the idea of providing only the necessary functionality in the kernel, and leaving applications the flexibility to build customized environments on top of it.

Resource Management

The design of Nemesis was driven by the aim to provide QoS support (which we discuss in greater detail in the next section 2.3.2), which resulted in the design enabling applications to execute their code directly rather than via shared servers. Shared servers are used only for security or concurrency control.

Nemesis provides dynamic allocation of resources to applications by the *QoS Manager* which allocates applications a *share* of the processor and ensures that short term demands can be always met. Also, the QoS Manager uses algorithms considering a long term view of the availability of resources, provides a consistent guaranteed resource to the application. In additional, users are expected to provide overall control of resource allocation in terms of observation and by defining QoS specification.

The approach to inter-domain communication supports implicit and explicit bindings. Supported *name servers* or *traders* performing interface reference matching provide clients with an interface reference to requested servers. However, user-customization and extensibility of the matching is not addressed.

2.1.7 Other Systems

Other extensible system, such as Vino [71], or kernel protection by proof-carrying code [54] address techniques for ensuring security of kernel extensions implemented by untrusted user-level code downloaded into the kernel.

Also, the non-extensible network-based system Scout [49] provides 'static' *specialization* — by creating dedicated 'paths' (multi-layered communication channels), it provides advanced application customization. Synthetix [62] also investigates incremental specialization of existing systems code. It focuses on reducing the length of 'paths' in the kernel in order to provide kernel optimization which is done without application-specific requirements.

QNX microkernel [27] offers a flexible environment, realtime support, and enables upwards scalability for large, multiprocessor applications, as well as downward scalability for resource-constrained PDA hardware. A customizable operating system, Arena [44], provides operating system-level resource management at user-level where it is accessed by libraries. Hardware is presented through low-level abstraction; customization is enabled only at the user-level by instantiation resource managers for particular policies.

2.1.8 Discussion

In recent years, research in operating systems has focussed on investigating issues of extensibility enabling applications to implement their own abstraction by presenting them raw hardware, or providing user extensions of existing system services.

Although research results have proven that this is a step in the right direction, the majority of existing systems (except Kea and DEIMOS) still lack the support for dynamic reconfiguration, enabling adaptation to changing conditions. In addition, issues such as QoS-based resource allocation enabling application participation, parametrized resource selection (as opposed to name-based allocation) and issues of scalability are still to be addressed. However, operating systems such as Exokernel, Nemesis, provide flexible environments where the required dynamic functionality can be provided by a resource manager running on top of them.

2.2 Other Trading and Reconfigurable Architectures

In this section we describe representative systems that support dynamic third-party binding. We focus on systems supporting trading and reconfigurations. Firstly, we describe the tuplespace approach, as it was applied to MAGNET, and discuss several related projects (section 2.2.1). Then, we discuss other trading and dynamic architectures, ANSAware, CORBA, DCOM, Aster and Matchmaking. In section 2.2.7 we briefly introduce other related architectures, and in section 2.2.8 we close with a discussion.

2.2.1 Tuplespace-based Architectures

Distributed applications often need to establish communication without *a priori* knowledge of their peer identity. In addition, in mobile environments it is desirable to enable services to be described dynamically by their parameters, rather than to refer to services directly by their names. Therefore, the *tuplespace* [22] fulfilling these requirements represents a very successful distributed communication scheme. In this section, we will discuss the original tuplespace with the programming language Linda, and several frameworks derived from this idea: Limbo[1], Osprey, JavaSpaces, Jini, FT-Linda, and T Spaces.

Tuplespace and Linda

The original tuplespace was designed by D. Gelernter at Yale University, with a set of operations (called Linda) enabling tuple manipulation [22]. The architecture enables communication by exchanging information in the form of tuples placed into or withdrawn from the tuplespace. Applications can use the tuplespace for communication and synchronization purposes. Its features include *free-naming* (parties need not know each others' identity in order to communicate), *time decoupling* (parties need not exist at the same time) and *space decoupling* (parties from different address spaces can communicate).

Linda is discussed in greater detail in chapter 4, as it inspired the approach undertaken in this thesis.

Limbo

Limbo, a QoS-based distributed system platform developed at Lancaster University [8] represents a successful attempt to utilize the tuplespace paradigm in a mobile environment. The framework extends the basic architecture by the notion of multiple tuplespaces (specialized for application-specific requirements, e.g., security), an explicit tuple-type hierarchy supporting dynamic subtyping and QoS management. QoS is supported by providing QoS-aware tuplespaces; residing tuples are

[1] The tuplespace-based architecture Limbo, disccussed here, should not be confused with the programming language Limbo designed for Inferno operating system, discussed in section 2.1.

enhanced with QoS attributes, such as expiration time, priority, etc. Changes in system features are kept up-to-date by QoS monitoring agents acting as proxies to a service. Reacting to these changes, Limbo performs an adaptation implemented using specific components such as filtering agents and bridging agents.

Osprey

Osprey [10], designed at The City University in London, uses Linda for application-server coupling — resource allocation. Information about system services and their requests are exchanged in the form of tuples. It provides more flexibility than the traditional Linda, by adding additional semantics into the tuple format. For example, a result-based tuple naming scheme — a client describes the request by its result (e.g., Time, LocalTime), rather than by the name of the server itself. By implementing a hierarchy of tuplespaces, Osprey provides scalability and protection. The architecture provides higher flexibility, but it does not address issues of user-customized matching and extensibility. This project is still in its early stages, therefore we cannot provide a more detailed description of the Osprey architecture.

JavaSpaces and Jini

JavaSpaces, developed at Sun Microsystems, provide a tuplespace-like distributed environment manipulating objects rather than data tuples. It enables global scalability and forms a base for the Jini technology [75]. The Jini infrastructure provides automated configuration mechanisms for devices (such as desktop and portable computers, printers, scanners, Webcams, etc.) to join and leave the network — it establishes dynamically (without drivers) the communication, sharing, and exchange of services between any hardware or software on a network.

The key techniques used in Jini are: leasing (a grant of guaranteed access over a time period), transactions (two-phase commit-based service protocol encapsulating a series of operations), events (enabling object-defined event handling) and lookup service (finding and resolving system services defined by their operational interface).

Although the lookup service provides dynamic binding between clients and servers by passing over server proxies to clients, it is rather restrictive — neither parametrized requests (such as describing services by their types and characteristics), nor user-customization of the interaction protocol are supported.

FT-Linda

The communication framework FT-Linda [24], developed at the University of Arizona as a part of the x-kernel project [29], is based on a fault-tolerant version of the Linda language. Its design techniques include: the notion of stable and volatile tuplespace, shared and private tuplespace, failure detection and ordered atomic multicast. However, as the primary goal of the framework is to provide fault tolerance, it addresses issues of reliability, stability and ordering, in contrast to the user-customization and dynamic flexibility required by the mobile users we are targeting. If our architecture was providing fault tolerance, approaches used in FT-Linda could have been adopted for MAGNET, as it is also based on a tuplespace framework, however, our platform does not aim to provide this functionality.

T Spaces

A Linda-based technology developed at IBM Almaden Research Center, T Spaces [76] is a network communication 'buffer' with database capabilities. In addition to Linda operations, T Spaces provide services (data indexing and query capability),

and event notification services and group communication services. Allowing applications and services to describe their functionality in terms of tuples, T Space 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 mobile environment.

2.2.2 ANSAware Distributed Systems Platform

The ANSAware software model [1], developed at APM Ltd., is based on a location-independent object model providing uniform interaction schemes between communication objects — the model is based on the RM-ODP architecture [66, 67]. A special object, *the trader*, acts as a mediator for services wishing to advertise their services (by exporting operational interfaces), and clients requiring them (by importing operational interfaces). Clients are enabled to specify their requests in terms of attribute values. Interfaces in ANSAware are defined in an Interface Definition Language (IDL), and the operations *import*, *export*, and interface implementations are described by a second language, Distributed Processing Language (DPL). If matching candidates are found by the trader, an implicit binding between the peers is created.

Although attribute-based matching provided by the trader enables enhanced flexibility, there is no notion of runtime adaptation and user-customization.

2.2.3 CORBA

The Common Object Request Broker Architecture (CORBA) [56] provides an architecture for communication in distributed object-based systems. Building elements of the architecture, objects, written in different programming languages, are described by an IDL. The architecture provides distribution transparency by implementing RPC-like *remote object invocation* which hides the physical location of interacting objects.

Although CORBA is very popular architecture for distribution object interaction, from the 'dynamic' point of view it is restrictive. Its naming service supports parametrized nameservers, but unlike MAGNET, user-customized trading of objects defined by their types is not supported. However, MAGNET running with CORBA could provide dynamic trading at an object level.

2.2.4 DCOM

Distributed Communication Object Model (DCOM) [48], developed at Microsoft, is an application-level platform enabling objects on different physical locations to communicate through common protocols, including Internet and Web-based protocols. Objects, defined by strongly-typed multiple interfaces described by an interface definition language, interact by an RPC-like communication enabling authentication and security.

As the architecture provides an environment for a distributed object communication, it would be a suitable computing platform for MAGNET which can enhance it with the dynamic features, such as object trading and runtime user-customization.

2.2.5 Aster

The Aster project [30], developed at IRISA/INRIA, addresses middleware reconfigurations based on software specification matching that selects the components of the

middleware (such as ORB), customized to the application needs. The Aster Environment provides three elements — *Aster Type checker* (implements type checking of components described using the Aster language), *Aster Selector* (retrieves middleware components that satisfy the interaction requirements), and *Aster Generator* (responsible for interfacing the source code files with the middleware objects). Non-functional properties, described in terms of formulas of the first order predicate calculus, are processed by the selector in the three-stage selection process: *exact match selection*, *plug-in match selection* (the selected component implements behaviour that satisfy the application, but does not match exactly the application's requirements), *closest match selection* (the selected middleware needs to be customized through complementary components).

Although Aster presents a powerful framework for parametrized component selection enabling automated customization, it does not address dynamic runtime adaptations nor enable users to participate in the selection and customization process.

2.2.6 Matchmaking

The Matchmaking framework [63], developed at the University of Wisconsin-Madison is a part of project Condor [41]. The environment is based on components describing their requirements and provisions in classified advertisements which are matched by a designated service — the *Matchmaker*. Classified advertisements enable components to be described in terms of parameters enhanced with arithmetical and logical operators (e.g. Type = "Machine", Activity = "Idle", Arch = "INTEL", Rank >=10, etc.) The Matchmaker compares relevant parameters of component advertisements, and notifies components; then the client contacts the server using a claiming protocol to establish a dynamic binding.

Powerful parameter-based matching resource allocation provides the required flexibility, however it does not allow user-defined service selection from a group of matching ones, nor does it support decentralization and runtime adaptation.

2.2.7 Other Systems

As was mentioned above, in this section we will briefly introduce other architectures supporting dynamic binding, or some kind of reconfiguration. However, as they are not directly related to our research, we will not discuss them in detail.

Regis [15, 42], an environment for constructing distributed systems, provides a unified framework for dynamic binding and runtime rebinding of components in distributed systems. As this architecture was used for implementation of MAGNET, the resource management framework presented in this thesis, we will describe Regis in greater detail in chapter 6.

The problem of dynamic adaptation to an environmental change at low level (device-driver level) has been successfully addressed by the PC Card (former PCMCIA). Popular ethernet cards can be added and removed from the system without powering-off or rebooting the computer. The Linux *kernel daemon* is another successful attempt, enabling operating system kernel adaptation by adding or removing modules transparently on demand [70]. At an application level, a Java-based object abstraction, JavaBeans, provides a dynamic platform for object communication.

Zero downtime operating support for dynamic data objects communication has been explored in 'The information bus architecture' [57] which is based on principles such as self-describing, anonymous communications and minimal semantics communication protocol. Nevertheless, this name-based approach does not support more flexible parameter-based addressing.

Open bindings, implementing component interactions which are constructed from a chain of objects performing particular functions, were investigated by Fitzpatrick *et al* [20]. They support the inspection and adaptation of the communication paths required by mobile multimedia applications. However, the flexibility of open bindings is gained at the expense of performance. Also, we believe that incorporating additional functionality into the complex communication path rather than into the communicating components themselves takes the control out of applications, which contrasts to the approach undertaken in MAGNET (enabling them to participate).

Guarana' [58], an architecture based on meta-object protocols, presents a tool for structuring and building fault-tolerant distributed programs. Meta-objects can be combined through *composers* that provide the glue code for them to work together, delegating control to them and resolving conflicts when they arise. It supports meta-level security policies and, by further composing composers, it enables construction of a dynamic reconfigurable object hierarchy.

2.2.8 Discussion

In recent years, dynamic issues such as providing greater flexibility, supporing dynamic runtime adaptations or designing loosely coupled communication schemes have been successfully addressed by many research projects and commercial technologies. However, these architectures typically target one issue, rather than providing a unified architecture and therefore are unable to support the requirements of mobile applications (see chapter 1). Nevertheless, MAGNET dynamic resource architecture can be used for trading objects (such as CORBA objects, DCOM objects, JavaBeans etc.) rather then resources. Then, platforms such as CORBA, DCOM are suitable for implementing the proposed architecture at an 'object level'.

2.3 QoS Architectures

A service provided by a resource is described as its *functional behaviour*. Additional service characteristics such as timeouts, are described as the *non-functional behaviour* of the resource. *Quality of service* is a general term for an abstraction covering aspects of the non-functional behaviour of a system. In particular, it includes not only the specification of non-functional service characteristics, but also necessary data models, operational constraints, and information about data measurement, monitoring and maintenance.

In recent years, research into QoS has typically targeted the area of continuous data transmission — multimedia: video and audio, and computer music. In this class of application, the aim is to provide acceptable quality in real-time[2]. In this case, the QoS manager must maintain agreed end-to-end service characteristics through all layers of the communication channel. A brief description of architectures providing this kind of QoS support is given in section 2.3.2.

However, our aim is to provide *QoS-based trading* (e.g., dynamic resource allocation, software upgrades, etc), as opposed to maintaining the QoS of the communication channel. For our purposes, QoS-based resource description covers in particular: guaranteed characteristics of system resources (such as monitor resolution, processor speed), consistency (defined in terms of 'up-to-dateness' of information on resources, accuracy, precision, as granularity of environment change), timeliness (described in terms of availability, delay), location-dependent information. Leading architectures addressing this subset of QoS support are discussed in section 2.3.1 (a general overview of services and mechanisms for QoS resource management is

[2]Quality, in this context, means both accuracy of the timing and the accuracy of output values.

discussed in [53], and detailed review of the state of the art can be found in [2]). In section 2.3.3 we summarize the discussed architectures.

2.3.1 QoS-based Trading Architectures

The Cactus project [28], developed at the University of Arizona, addresses fine-grain customization of QoS in distributed middleware. As the relationship between QoS attributes (such as consistency, correctness, timeliness, security) is a tradeoff, the ability to customize QoS is especially important in resource-constrained systems (e.g., mobile computing). By adaptation of *micro-protocols* (collections of event handlers) that ensure different QoS attributes and a configuration protocol, a user-customized environment can be constructed.

QoS-based resource management for distributed multimedia applications was addressed by the QoS Broker [51] developed at the University of Pennsylvania. It is based on the notion of a duality of communicating system components: *broker-buyers* and *broker-sellers*. According to the component activating the process, the QoS Broker distinguishes between a sender-initiated brokerage, and receiver-initiated brokerage. The QoS Broker Protocol facilitates the negotiation of resource characteristics, and runtime adaptation requiring a renegotiating process. QoS Brokers are a basis for the QualMan architecture [50] providing soft real-time QoS guarantees to multimedia applications, such as video-on-demand and MPEG players. In addition, QoS Brokers are also used in the Omega architecture [52] to ensure end-point resource guarantees (such as responce time, etc.), presuming it is coupled with networks which can make bandwidth and delay guarantees.

In addition, several QoS-trading projects were discussed in section 2.2: Limbo [8], a tuplespace-based project provides quality of service, in particular monitoring and adaptation. Matchmaking [63], providing the coupling of applications to servers based on classified advertisements (equipped with arithmetical and logical operators), also supports a level of QoS-based selection. Aster [30], a framework based on software specification matching, provides complex QoS-based resource selection. Nevertheless, none of these architectures supports a user-customization of the selection process which adds an extra flexibility over QoS-based resource definition.

2.3.2 Other QoS Architectures

The QoS-Architecture (QoS-A) [11], developed at University of Lancaster, addresses the support for performance of multimedia applications over high-performance ATM-based networks. Kendra [46] is investigating adaptive techniques to improve the performance of data delivery over the Internet. Specifically, runtime adaptation occurs when network bandwidth falls or improves.

The Nemesis project [64, 39] (discussed in section 2.1 in greater detail) provides a probabilistic guarantee of resources and expects applications to monitor their performance and to adapt when resource allocation changes. It is based on a QoS Manager providing QoS-based scheduling (discussed in section 2.1) and two techniques: feedback for QoS Control supported by *QoS Controller* which defines the policy to be followed and can be directly dictated by users, and QoS Crosstalk between time-related data streams in network protocol stacks.

Formal approaches to QoS specification, based on description of the resources in Z [61] language, has been investigated by Staehli *et al* [69]. Focusing on end-to-end service quarantees for continuous media, they distinguish between content, view and quality specification and define the presentation quality in terms of a subjective error interpretation. The error models extend the opportunity for optimization of resource utilization.

Other architectures providing comprehensive end-to-end QoS support include the CESAME project [6], TINA [55], the Heidelberg Transport System (HeiTS) [25] and an Extended Integrated Reference Model (XRM) [38].

2.3.3 Discussion

In recent years, support for QoS has been investigated at various system levels (such as hardware, operating system, middleware, etc.) and targeted to different classes of application (multimedia, resource allocation, adaptation, monitoring).

Our approach focuses on QoS-based trading of resources, in contrast to ensuring agreed QoS of the underlying infrastructure. There are several successful attempts in providing parametrized QoS-based trading of applications requirements and server offers (Matchmaking, Aster, QoS-Broker, ANSA trader, QoS Broker, etc.). However, none of them provide user customization of the matching process, nor scalability of the architecture.

2.4 Chapter Summary

In this chapter we have presented existing architectures supporting dynamic resource management. As it is a broad issue, we focussed on major projects in three following areas: extensible operating systems, trading and reconfigurable architectures, and quality of service architectures.

Extensible Operating Systems

Research in operating systems has focussed on investigating issues of extensibility enabling applications to implement their own abstraction by presenting them with raw hardware (e.g., Exokernel), or providing user extensions of existing system services (e.g., SPIN, Vino). Although it is a step in the right direction, support for dynamic reconfiguration enabling adaptation to changed conditions and user-customized parametrized resource allocation still need to be addressed. In addition, due to the diversity of hardware resources, changing degree of connectivity, and current technology improvements, the need for user-customization and runtime adaptation has increased. But from the computing environment point of view, we have identified that architectures, such as Exokernel and Nemesis, provide a sufficient flexibility for these features to be supported by a dynamic resource manager.

Trading, Reconfigurable and QoS Architecture

There are many successful projects providing support for dynamic trading and reconfiguration: tuplespace-based architectures (such as Linda, FT-Linda, Jini, Osprey, Limbo, etc.), and others (such as Aster, Matchmaking) enabling more flexible component coupling based on a parameter description.

QoS support is investigated by many ongoing projects. Typically, it focuses on providing a guaranteed QoS of the underlying infrastructure at runtime. However, as the primary goal of this work is a dynamic resource manager, we have discussed architectures providing QoS-based resource trading (such as QoS-Broker, Cactus, Limbo, Aster, etc.)

All trading and QoS architectures provide a certain level of flexibility and adaptability, however, applications in mobile environments seek a unified framework offering in addition user-customization of the trading process and extensibility.

Thesis objective

This thesis seeks to develop a dynamic extensible resource management architecture, MAGNET, enabling trading of resources based on requests for a type of a service, rather than for a service name. Extensibility of the architecture should enable users to define services and requests at runtime, and to user-customize the trading process. This further allows QoS-based description of services to be implemented. In addition, runtime adaptation to changes in the computing environment, monitoring of information about services and scaling the architecture should be also supported.

Chapter 3

A Model of Dynamic Third-party Trading

From an 'abstract component interaction' point of view, the problems of dynamic resource management that MAGNET attempts to solve are trading of information about services and dynamic third-party binding of components. In order to provide the required generality of the MAGNET architecture, we will define its framework in general 'component-binding' terms. This chapter is devoted to formalizing this architecture.

We start by defining basic terms for a component environment (section 3.1), then we summarize our assumptions (section 3.2), and give an overview of phases of a binding process, section 3.3. Next, in section 3.4, we focus on rebinding issues relevant to this thesis, and finally, in section 3.5, we discuss several aspects of quality of service management.

3.1 Terms and Definitions

In this section we define basic functional elements and relevant terms involved in service trading (components and their service interfaces, interface references, service definitions, component bindings, binding idioms, the Trader, a matching process, communication channels, and communicational protocols). Definitions of these terms are derived from the RM-ODP standards [66, 67, 68] tailored to the trading approach used in this work.

Components

Distributed systems comprise basic functional units — *components* of varying granularity that represent a wide range of system elements, such as hardware resources, abstraction servers (such as file systems), and user-level programs. Also, components can represent any objects in terms of object-oriented languages and environments (such as, C++ objects, CORBA objects, JavaBeans, etc.)

The functionality of components is assumed to be mutually independent; any dependencies are expressed in terms of component interaction. Structure-wise, they can be primitive or composed of other components.

Service Interfaces

Components act as 'black boxes' and their *functional behaviour* is fully described by a *service interface* which defines services provided to, and services required from

other components in the system.

Components requiring a service are called *clients*; components offering a service are *servers*. These 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' (one server can communicate with many clients over the same interface). In addition, clients are considered *active* entities, while servers are *passive* (requiring an external third-party to manage them, as this approach simplifies the semantics of rebinding, as will be discussed in section 3.4.2).

Unlike traditional objects, components can have multiple interfaces to meet the need to express QoS requirements and describe various service characteristics which cannot be attached to a single interface.

Interface References

Every service interface can be located and accessed by its 'name', called an *interface reference*. The interface reference must be unambiguous within the system range, embodying sufficient information to allow the required interaction to be established. Service interfaces, together with corresponding interface references, can be created during component creation, or dynamically at runtime.

Services Definitions

In addition to the 'name' (an interface reference), services may also be described by a *type of service* they offer (such as a printer or a scanner), which need not be unambiguous, and by additional *characteristics*, such as a QoS description of the required interaction (e.g., resolution, speed, etc.) This issue is covered in greater detail in section 3.5. A combination of an interface reference, a type of a service and its characteristics is called a *service definition*.

Component Bindings

In order to enable interaction between distributed components, *a binding*[1] between their interfaces has to be established. A binding is a result of a process, called the *binding process* (defined in section 3.3), consisting of a sequence of actions to be performed which result in the creation of a communication channel between component service interfaces.

Binding Idioms

Once corresponding interfaces and relevant interface references have been created, components can be bound by a first-party (so-called *first-party binding*), or by a third-party (so-called *third-party binding*) [15].

First-party binding is established by a client component, and can be performed if the peer interface reference is known to the binding initiator, and therefore the service interface can be accessed directly.

Third-party binding is established by an intermediate component (neither server nor client), and is performed by a sequence of first-party bindings — a client and a server link to the intermediate component, requesting or offering a particular service, then the third-party, with a knowledge of overall system behaviour, can establish the resultant client-server binding.

[1] This term suffers from being 'overloaded'. In addition to the resultant *interaction* (used as a noun), it can also mean the *process* of establishing it (used as a verb, to bind).

The Trader

Communication between components in open systems can be based on an Interface Definition Language (IDL) which predefines the syntax of the interaction, like in CORBA [56]. Another solution, offering higher flexibility, is a third-party component, called the *Trader*, which collects service definitions defined at runtime, performs a matching process, and establishes a 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 which include additional constraints, such as QoS definitions, requires parameterization defining preferences of particular components. As these are impossible to define for all components *a priori*, the Trader supports user-customization of the matching process. It is described in detail in section 3.3.2.

Communicational Channels

A channel represents the actual communication path enabling a binding between components. It is obliged to satisfy requirements on the properties of the interaction, including maintaining the agreed QoS. It comprises objects such as stubs, protocols, and binders [66].

Communication Protocols

The syntax and semantics of the established binding over a communication channel is defined by a *communication protocol*. It specifies, typically a set of functions (called a *functional interface*) provided by the server to its clients, and their guarantees (such as reliability).

Example

Figure 3.1 illustrates bindings between three components — the Trader, Client and Server. Darwin, an architecture-description language, provides a convenient notation for specifying interactions in distributed systems [43]. We use its graphical form throughout the thesis to represent interconnections between system elements. A rectangle represents a component, a circle stands for a service interface. A provided service is represented by a filled circle, a required service by an empty circle. A line between filled and empty circles represents a binding implemented by a particular communication channel.

In Figure 1, Server and Client find corresponding interface references 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 required interface reference is discovered by the matching process, the resultant end-to-end binding between Client and Server can be established (phase 2).

3.2 Assumptions

In this section, we will define the assumptions we made concerning our computing environment and system components' behaviour. The MAGNET framework pro-

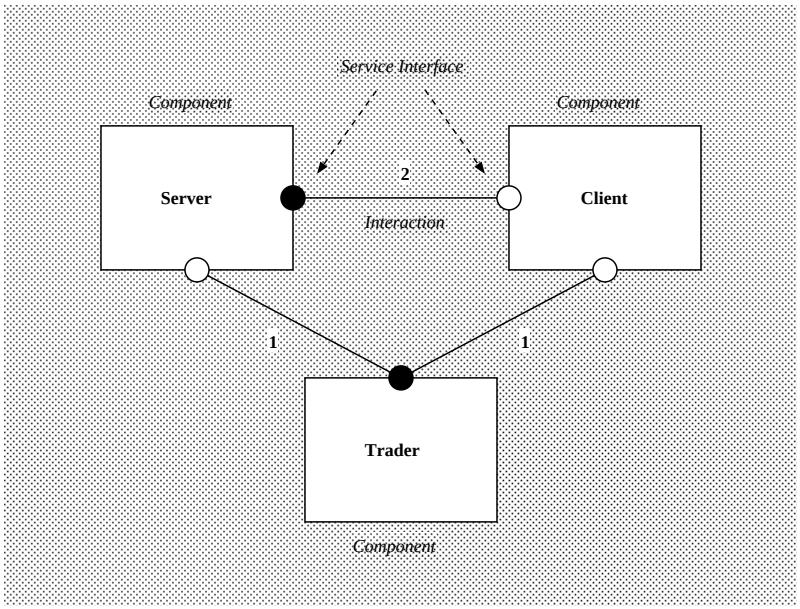

Figure 3.1: Binding between Server and Client established by the Trader

posed in this thesis is feasible only in systems where these assumptions are valid. In addition, in order to keep the problem tractable, we left out support for application areas which are beyond this, such as those requiring the environment to change very frequently, real-time and fault-tolerant applications, etc. In chapter 7, we elaborate on the implications for the architecture if these assumptions were not the case. The assumptions on which the framework is built include:

1. **Consistency**. All system components are assumed to maintain overall consistency. That is: rebinding can be performed only when the system is in a safe state, unexpected component crash cannot happen, and a component finishing its operation must leave the MAGNET structures (the Trader) in a consistent state and release the allocated resources.

2. **Protection**. Components are responsible for ensuring the validity of information on their services. This prevents components from advertising misleading information on non-existent resources.

3. **Synchronization**. Components are responsible for synchronization. This includes communication with the MAGNET framework as well as component-to-component interaction.

4. **Security**. The architecture supports a user-defined matching process. This is assumed to be secure in that control is returned back to the Trader while not altering other system data.

5. **Federation Scale**. A domain-type unit in the architecture is called a federation (discussed in detail later). We assume the number of components in a federations to be roughly tens, they can generate roughly tens to hundreds service requests placed into the Trader, not more than ten at the same time. It implies that a federation can have roughly tens of processors, as they are also components for our architecture.

6. **Frequency of Change.** We assume each component in the environment to change its features with frequency of minutes and hours, rather than seconds and milliseconds.

7. **Service Characteristics.** We assume that that component requests and offers have not more than tens of elements. In addition, the system is more suitable for processing requests and offers if they are equally distributed according to the number of elements in the request. In addition, we expect the number of types of service characteristics to be not more than tens. However, we do not constrain the semantics of the elements.

8. **Naming**. The architecture uses the naming scheme of the environment where it operates (e.g., the Internet with its IP addresses). We assume that the names are unambiguous and are constructed in a way that they can form a hierarchical tree structure with a single root, and an unambiguous 'path' in the tree (at the naming level) between two federations can be determined (however, nothing about the network topology at the implementation level is assumed.)

3.3 The Binding Process

A sequence of actions to be performed preceding the establishment of a component interaction is called the *binding process*. This can be achieved by linking to the peer component directly, if its reference is known to the initiator in advance, or establishing the binding indirectly, via a third party. In open systems due to their required flexibility, the latter case prevails. A trading of service definitions is performed by the Trader in our model.

In traditional systems, the binding process is typically implemented as an integral action inhibiting component customization, and disabling the participation of a third party (e.g., Unix system calls [3], RPC [7], CORBA [56], etc.) However, in open distributed systems, the binding process requires clearly delimited phases [66] in order to achieve flexibility and provide user-customization.

In our model, the binding process comprises the following phases:

1. Exporting service definitions
2. Negotiating service definitions
3. Establishing a communication channel

When a communication channel is established (the third phase is successfully performed), the required end-to-end binding takes place. Although this is the final goal, it is not a part of the actual binding process. In this section we describe these phases in greater detail.

3.3.1 Exporting Service Definitions

A service definition created for a service interface has to be passed to a component containing a complementary interface via the Trader which finds the requested match. A *service export* is a process of offering a service (defned by its service definition) to the Trader. A complementary operation, *service withdrawal*, is a process of removing a service definition from the Trader. Although it is not necessary for the binding process, service withdraw is an essential operation of the Trader. Figure 3.3 illustrates service export, service withdraw and negotiation of service definitions.

Policies

In order to achieve an agreement between components exporting their service definitions, policies defining operation semantics must be formulated. An *export policy* is a set of rules controlling the service export to the Trader including, for example, an obligation for a specific format, permission rules, timeout, etc. The complementary set of rules defining service withdrawal is called a *withdrawal policy*. In the case of distributed trading systems, these policies must define the propagation of service export between remote parts of the system.

It is up to system designers to decide whether all three types of policy are enabled. In addition, each particular application should be able to define and tailor available policies for its components.

Federation

In scalable distributed systems Traders must be networked in order to cooperate on providing remote information. One approach is to form a *global trading system* that any Trader may dynamically join or leave. Although there are systems requiring global shared information (e.g., distributed database engines, Internet search engines), this approach encounters problems similar to designing a global nameserver [37].

As components often interact on a local scale (such as resource managers within a particular domain), an alternative solution to the global trading system is a *local trading system*. In this case inter-Trader communication must be supported to enable service export for components beyond the local domain. Trader domains with domain-specific security and propagation policy information, internetworking with other Trader domains are called *federations*. Passing service definitions across the federation boundary, consequently, must be handled by appropriate communication channels reflecting the 'beyond-federation' distribution and security issues [66].

In Figure 3.2 a trading system consisting of two federations is illustrated.

3.3.2 Negotiating Service Definitions

A service definition (a type and characteristics of the requested binding together with enough naming information to locate the interface, the interface reference) is exported via the Trader. These characteristics form the set of requirements, from both components involved in the communication, which must be met before the binding can be established. A communication channel can only be established if the requested properties of both involved parties are satisfied.

Rules

When all service definitions are exported into the trading system according to system policies, the actual matching process takes place. In the general case, it comprises two phases — *search* and *select*, defined by the client and the Trader.

Matching rules define the *search* operation; they include [67]:

- pre-conditions of the binding ensuring that the interaction can be technically established, e.g., interfaces must be of the same type, and complementary roles (client, server).
- a minimal set of requirements of both components (defined by the service definitions and according to user-customizable matching process).

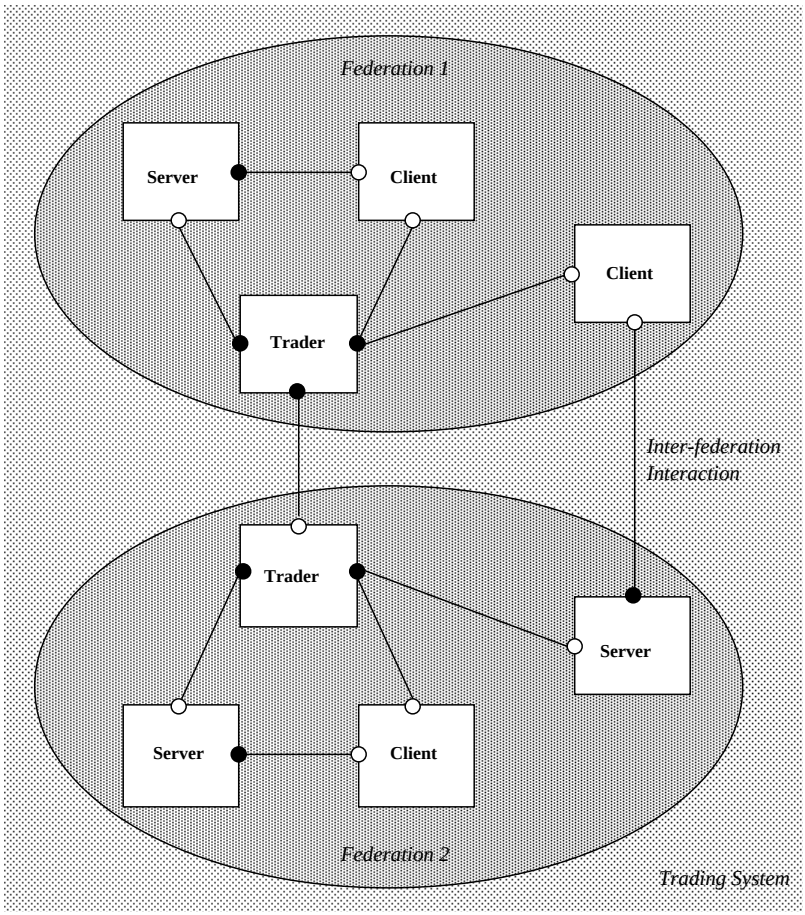

Figure 3.2: A Trading System consisting of two Federations

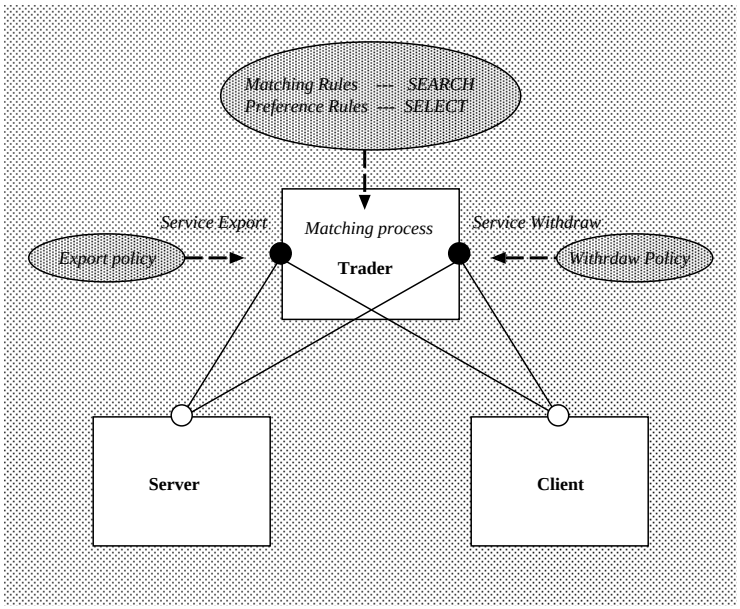

Figure 3.3: Trading

Matching rules can be formulated exactly, or component requirements can be parameterized (user-customized matching process) enabling a set of different service definitions to match the requirement.

Preference rules define the operation *select* that extracts one service from a set of service definitions fulfilling the matching rules.

Service characteristics expressed in terms of parameters typically cannot be linearly ordered (e.g., "Is speed better than resolution?") without preferences expressed by components themselves. Therefore, components using a trading system supporting this level of negotiation are obliged to express their preferences in terms of preference rules when exporting service definitions into the trading system, performed as a user-customized matching process. Otherwise, the *select* operation must be performed non-deterministically (e.g., first-fit strategy).

In the case of a federation, additional rules and protection checks defining the propagation of exported service definitions may be defined, such as reflecting the domain scope, and defining constraints under which a *search* or *select* operation may take place beyond federation boundaries.

Figure 3.3 illustrates the negotiation phase of the binding process.

3.3.3 Establishing a Communication Channel

A special component attached to every functional component responsible for creating a communication channel is called a *binder*. When the required component pair has been found by the Trader (service definitions have matched), client and server binders are invoked to create the requested communication channel. If this task is successfully performed, the required binding is established and components may start communicating.

Communication channels implementing an actual binding may use primitives of varying complexity, from simple message-passing primitives, procedure calls, through dispatchers marshaling functions and parameters, to network protocols. They can be primitive (represented by one functional unit), or composed (featuring a sequence of functional units cooperating on the task — for example, network

protocols). Bindings might operate locally, remotely within a domain, or between federations.

If the interaction comprises service characteristics, such as QoS constraints, it is the channel's responsibility to maintain the agreed level of service characteristics by constant monitoring of relevant features [46].

3.4 Rebinding

As we focus on computing environments which change characteristics frequently, adaptations to new conditions must be possible. In addition, the support for mobile users dependent on location and time-aware information also necessitates adaptation. Breaking an existing component binding in order to establish a different one is called *dynamic rebinding*[2].

We distinguish between *fault management* — handling the breaking of a binding as a result of a component failure or a communication channel failure without previous agreement of both parties, and *change management* — breaking interaction after both components have reached a safe state. Fault management is not an issue of the architecture outlined in this thesis; more detailed discussion of related problems, such as failure discovery and dealing with inconsistency can be found in [36]. However, as we have taken the black-box approach, we assume that all components in the system are able to decide when they can be rebound. In addition, our model is based on the assumption that the component initiating the operation in cooperation with other components involved in rebinding, is responsible for maintaining overall system consistency. If a third party is involved it needs to cooperate with both of the components which are to be rebound.

3.4.1 The Rebinding Process

In section 3.3 we defined three phases of the binding process: exporting service definitions, negotiating service definitions, and establishing a communication channel. The *rebinding process* comprises four phases, semantically similar to those of the binding process:

1. Exporting service definitions
2. Renegotiating service definitions
3. Destroying a binding
4. Reestablishing a communication channel.

In the case of rebinding, ordering of the phases is not strict — destroying a binding may precede renegotiating service definitions in cases when a client performs the destruction, and then searches for a better service.

Exporting Service Definitions

In section 3.3.1 we discussed issues of the framework for exporting service definitions into the Trader. This is used for both binding and rebinding purposes, as this is the only repository of service definitions in the system.

[2]Strictly speaking, every rebinding is dynamic as it is performed as a result of changes in the system. Non-dynamic ('a priori') rebinding can only take place when the purpose of the original interaction was accomplished, therefore, it could be described as a sequence of two independent interactions. For this reason, we will use the terms *rebinding* and *dynamic rebinding* interchangeably.

In addition to manual modification of relevant information in the Trader, an automated *monitor* continuously checking the status of service exports assists with keeping service definitions up-to-date. Monitors are separate components running at the application level cooperating with the monitored servers and clients.

Renegotiating Service Definitions

Semantically renegotiating service definitions follows the matching process performed in the Trader, defined for the negotiation phase. It involves finding a new peer according to matching and preference rules, discussed in section 3.3.2.

In the binding process, this phase was performed by components themselves, exporting their service definitions into the Trader. However, the renegotiation phase leads into three different cases according to the component responsible for selection of a new peer. The unbound peer might be left to perform the renegotiation itself (*first-party renegotiation*), or it is presented with an appropriate replacement (*third-party renegotiation*, or *no-renegotiation* — in cases when an external entity presents components with a replacement without contacting the Trader).

Destroying a Binding

There are two semantically-different situations leading to the destruction of an interaction. Firstly, it is performed when the initial purpose of the communication has been satisfied. Secondly, destruction is required as a first step before a rebinding is performed, as it is the case here.

As components are represented as black boxes, it is impossible to reveal their semantics. Therefore, the third party component can only request the destruction which is then performed by both involved peers. This functionality can be built in to the communication protocol as a specific function, or it can be performed by a dedicated component attached to both client and server. Component themselves have to transfer the state of the binding, if appropriate, enabling them to continue operation with a new peer.

Reestablishing a Communication Channel

Once the previous interaction has been destroyed, and a new peer component has been found, the reestablishment of a communication channel may take place. Semantically, this phase does not differ from the establishing a channel phase described in section 3.3.3.

3.4.2 Rebinding Situations

We have described the rebinding process. Now we look at possible situations in which the system transforms if different components *initiate* the rebinding or *renegotiate* the new peer (the second phase of the process).

Rebinding can be initiated from within the component as a result of changed requirements (*first-party rebinding*), or by an external third-party, either human or automated, providing an overall application strategy (*third-party rebinding*). In addition to first-party and third-party rebinding distinguishing between the role of the initiator of the rebinding process, we also have considered which component performs the renegotiation of service definitions, because trading requests is the primary goal of the architecture. As was outlined above in section 3.4.1, we distinguish between *first-party renegotiation*, *third-party renegotiation*, and *no-renegotiation*. Therefore, there are four rebinding situations which the adapting system might use:

- first-party renegotiated first-party rebinding

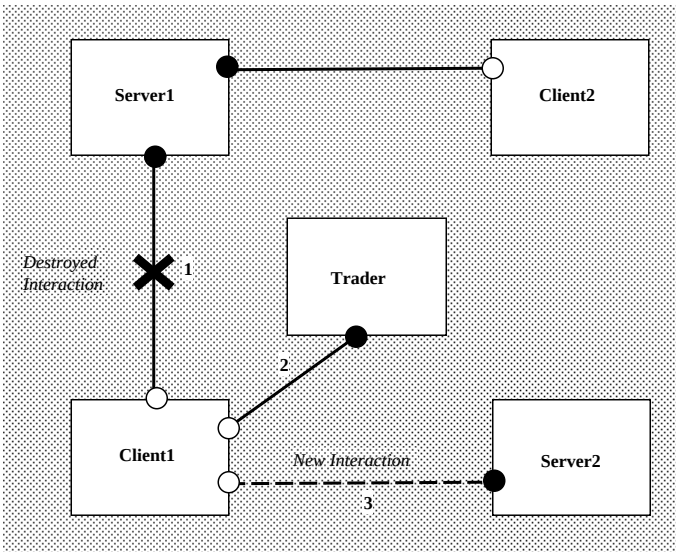

Figure 3.4: First-party Renegotiated First-party Rebinding

- first-party renegotiated third-party rebinding
- third-party renegotiated third-party rebinding
- no-renegotiation, third-party rebinding.

Remaining combinations for the first-party rebinding (third-party renegotiated, and no-renegotiation) are not valid, as the former one would not lead to a different situation from 'third-party renegotiated third-party rebinding', and there will be no rebinding performed in the latter case. These four situations have to be considered separately as they differ in several points, in addition to different components involved:

1. phases of the rebinding process are performed in different order
2. the number of components rebound in a single action vary (from one to many) as a result of the asymmetrical binding between clients and servers.
3. unbound clients originally attached to the same server might be all rebound to a single new peer, or left to find new servers themselves. This might result in different overall system configurations.

First-party Renegotiated First-party Rebinding

A component (client) initiates a rebinding if it requires an adaptation to changes in the computing environment. As a client is considered an active entity, we assume it can decide whether and when it needs to adapt (e.g., to change to a faster connection when a portable is connected to the Internet). As for the order of rebinding phases, the destruction of a binding might precede renegotiation of a new service, or vice versa. Other interactions between the original server and its remaining clients are not affected.

Figure 3.4 illustrates the situation where Client1 initiated a rebinding, destroys the original binding, performs a renegotiation, and links to a Server2. However, other bindings (between Server1 and Client2) remains unaffected.

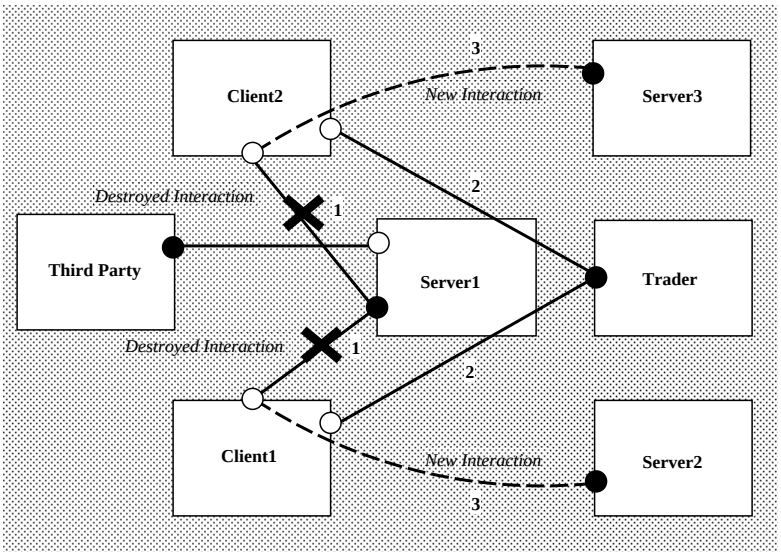

Figure 3.5: First-party Renegotiated Third-party Rebinding

First-party Renegotiated Third-party Rebinding

A server, as a passive component, is disconnected from the system by an external third-party. Renegotiation is carried out independently by all client components which might result in them finding different servers. A typical example of this case is a server shutdown.

This situation is illustrated in Figure 3.5 where Server1's bindings are destroyed by a third-party (the third party initiates the operation which is performed in cooperation with the servers). As renegotiation is left to the unbound Client1 and Client2, the figure illustrates that different servers (Server2 and Server3) have replaced Server1.

Third-party Renegotiated Third-party Rebinding

In this case, a Third-party has renegotiated a new peer in the Trader. Now, it faces a tradeoff between the better offer on one side, and the 'phase' of the operation and an overhead of the rebinding on the other side. However, its external knowledge of system behaviour enables the Third-party to decide whether the rebinding is beneficial. A typical example is an adaptation as a result of changes to system state, resource availability and quality of service.

This case is illustrated in Figure 3.6. Client1 is rebound from Server1 to Server2, found by the Third-party. Remaining bindings (between Server1 and Client2) are preserved.

No-Renegotiation Third-party Rebinding

In this case, the operation is initiated by an external Third-party replacing a component, e.g., a server upgrade. The third-party ensures the 'upgrading' of all necessary structures in cooperation with the original component and announces the interface reference of the replacement to all connected clients. Therefore, no trading in the Trader has to be undertaken. Consistency, which in this case might be nontrivial, must be handled by the upgrading server or by the manager which initiated the upgrade.

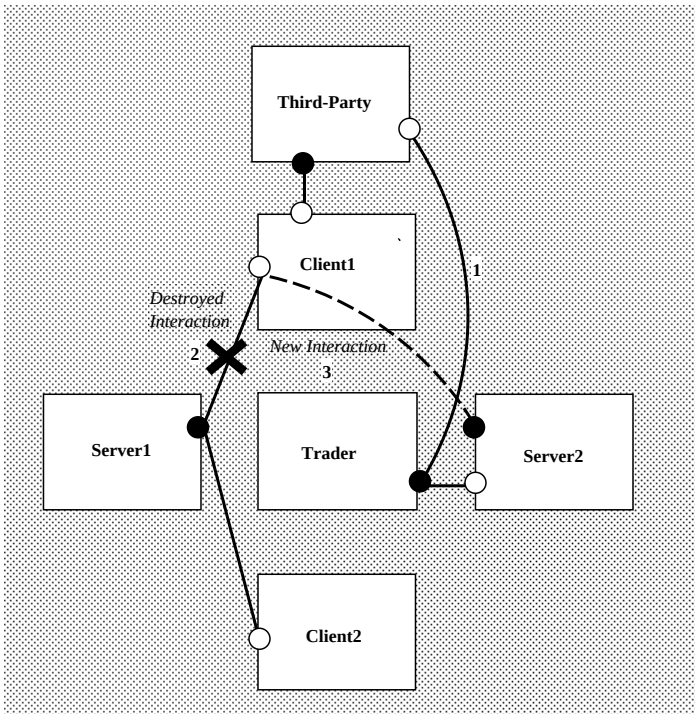

Figure 3.6: Third-party Renegotiated Third-party Rebinding

In Figure 3.7, Server1's bindings are destroyed by the Third-party. Both clients (Client1 and Client2) are bound to the replacing server (Server2).

3.5 Quality of Service Management

As was defined above, a service interface describes the functional behaviour of a component, either provision or requirement, and is accessed by a service interface. Also, it is desirable to enable the service to be requested by the type of service it provides and its characteristics. We termed this a service definition.

Service characteristics, including features such as timeouts, and service characteristics under which the particular service is provided, describe the non-functional behaviour of the component. As was introduced in chapter 2, a unifying term for various aspects of the non-functional behaviour is *quality of service*.

As QoS is a quickly evolving research area, we describe these terms in greater detail, in order to clearly define the angle from which we approach this problem. As in section 3.3 (discussing the binding process), we base our term definitions on the RM-ODP standard [68].

3.5.1 Introduction

QoS aspects span a wide spectrum of non-functional requirements. 'Static' systems, at one end of the scale, consider QoS characteristics during system design and configuration (such as OS structure, task priorities, static resource allocation, etc.) At the other end of the scale, fully 'dynamic' systems manage QoS characteristics at runtime (using techniques such as monitoring, routing, filtering, application adaptation, etc.).

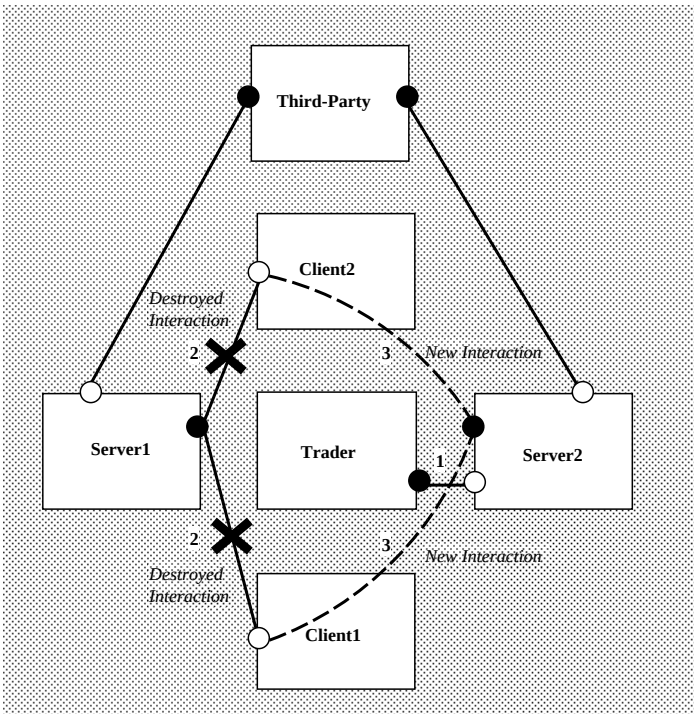

Figure 3.7: No-Renegotiation Third-party Rebinding

As MAGNET is primarily designed for resource allocation, we focus on the QoS of system resources, not necessarily changing at runtime (e.g., a printer resolution). The aim is to provide a *QoS-based trading* (e.g., resource allocation, software upgrades, etc), as opposed to maintaining QoS of the communication channel. Therefore, it can be said that our approach is closer to the 'static' end of the scale. However, resource allocation is never fully static. Characteristics of classical resources, such as length of a printer queue, or processor load, have never been static and predictable. In addition, we target our design to open systems where components may join and leave at runtime which precludes their characteristics from being known in advance. The design of our QoS framework reflects the needs of the mobile applications.

As component lifetime (based on binding of its services) is a dynamic process, QoS Management must reflect the needs of components at different phases of execution. Therefore, we distinguish a sequence of three phases performed in this order: *QoS Definition, QoS Negotiation,* and *QoS Maintenance.*

3.5.2 QoS Definition

The *QoS Definition* comprises QoS characteristics known *a priori*, static or dynamically changing at runtime. In addition, high-level system requirements defining system policies — for example, dedication or reservation of particular resources — might also be expressed in terms of their QoS Definition.

The QoS Definition covers a wide range of non-functional resource features. In our resource management framework, it covers, in particular: guaranteed characteristics of system resources (such as monitor resolution, processor speed, network throughput), timeliness (described in terms of availability, delay, or response time), consistency (for example, 'freshness' or 'up-to-dateness', accuracy, precision, as granularity of information expressed) and possibly failure-related behaviour.

Level	*Printer*	*CPU*
Characteristics	B&W, laser, 600dpi, queue=3, floor=5	Pentium, 200MHz, 4MBCache
Service	laser **WITH** queue < 5	Pentium **WITH** 200 MHz
Application	laser printer with queue < 5 **AND** CPU Pentium, 200 MHz	
Distribution	laser, queue < 5 on floor 5	Pentium, 200 MHz, anywhere

Table 3.1: QoS-based Definition of a printer and a CPU

Components in open distributed systems can contain subcomponents, each of them providing and requiring services. The QoS Definition framework must provide a mechanism for expressing QoS-based service characteristics at all levels of component hierarchy, as well as their combination priorities. We consider service features at four levels, bottom-up: *characteristics level*, *service level*, *application level*, and *distribution level* where each one forms the basis for the ones above:

1. *Characteristics level*: represents values of potential resource characteristics (e.g., printer resolution, printer speed, colour versus black-and-white printing, printing technology — matrix, laser, ink-jet, etc.) They can have relevant discrete values, intervals, limits (max, min), or thresholds.

2. *Service level*: service definitions may comprise service characteristics applicable for a particular service at higher granularity (e.g., printer service: a colour printer, 600dpi with no queue).

3. *Application level*: expresses a *combination* of components with certain QoS characteristics (e.g., a processor and a memory — Pentium running on 200MHz with 32MB RAM memory)

4. *Distribution level*: components are distributed spatially; location and time-dependent information must be able to reflect the mutual distance and time difference between components in order to ensure realistic estimation of location-dependent parameters (e.g., delay, 'nearest' resource, time-based operation scheduling, location-dependent time: server time, client time, or 'absolute' which can be, for example, 'office time').

Example

Table 3.1 illustrates a hypothetical QoS-based resource allocation requirement. For simplicity, it consideres only two servers — a printer and a CPU. Characteristics of both resources with assigned values are described at the Characteristics level (e.g., black-and-white, laser, 600dpi, queue length=3, floor=5). Their combination defines the resource characteristics. However, they can be requested by clients defining their QoS requirements *not exactly*. At the Service level, a combination of values represents a particular resource which is to be satisfied (e.g., laser, queue length$<$5).

At the application level, a combination of both resources is requested (e.g., laser printer with queue length$<$5 *and* CPU Pentium, 200 MHz). There are resources that might be allocated separately, but often it is necessary to reserve a group of resources at the same time in order to be able to use them, for example CPU and memory.

The Distribution level features represent location-dependent information. In our example, it is the floor on which the printer should be located. This issue is essential for mobile users changing location while requesting services.

3.5.3 QoS Negotiation

The *QoS Negotiation* phase covers the negotiation of the service definitions according to matching rules (defined in section 3.3.2) that are extended to express a QoS-based matching process, in particular QoS-based operations *search* and *select*. They define the semantics of the matching process for service characteristics (search), and component priorities 'ordering' matching services. For example, a request for a printer could be defined as: *search* for printers with resolution over 600dpi, and *select* the one with the shorter queue.

As the negotiation process takes place in realtime, its time complexity also contributes to the QoS characteristics being negotiated. Therefore, the traditional 'best-effort' strategies are often disappointing. This is an additional reason for enabling the negotiating process to be component-defined and customized.

3.5.4 QoS Maintenance

During component binding, QoS characteristics may change due to a user moving physically from one location to another, changes in the environment, or explicit indications (e.g., the number of processes on a processor, or a number of jobs in a printer queue) from any of the involved components.

QoS characteristics may change in both directions — improvement or degradation — both requiring appropriate actions to be undertaken according to component requirements.

There are two fundamental strategies for dealing with fluctuation in QoS: *resource management* and *application adaptation* [68].

1. *Resource management* attempts to fulfill the QoS requirements originally agreed by allocating additional resources, or, by extending the service provided by existing resources (e.g., requesting more disk and memory space, extra CPU-cycles). However, where QoS is improved, allocated resources can be released.

2. *Application adaptation* deals with resource degradation or improvement by providing a service of a different quality within an accepted range (e.g., presenting lower quality video and audio, switching into text mode instead of providing a full-graphics interface.)

Considering these two strategies for the binding model, resource management can be implemented by a first-party or a third-party rebinding of the client to a different server providing the required service; application adaptation keeps the established binding, but changes appropriate communication protocols within the communication channel.

Not every change in system resources can be adapted to. There might be cases when changes in QoS characteristics are so drastic that none of these strategies can provide a sufficient adaptation; then an external component (e.g., a system administrator) must interfere in the rebinding process, or a binding cannot continue.

QoS Monitoring

QoS adaptation strategies are performed as a result of a change in QoS characteristics. In order to achieve this, the current level of QoS must be kept up-to-date at all times (according to a 'accuracy-grain' provided by the Trader. Therefore, *QoS Monitoring* must be supported as an essential part of QoS Maintenance. It uses component-dependent techniques to obtain QoS values (depending on a particular resource) that are actually achieved by a particular binding.

As in our model we treat components as black-boxes, services are obliged to provide *monitor* components which, by directly interacting with the component, keep the information about varying QoS characteristics up-to-date in the Trader.

3.6 Chapter Summary

In this chapter we have defined a unifying model for third-party trading based on a resource description by a type and characteristics of a service. The model is appropriate for our primary purpose — dynamic resource management — but its generality makes it suitable for any architecture requiring a dynamic trading architecture.

We have set the scene by defining terms and summarising our assumptions. Then, we have defined the *binding process* as a sequence of actions that must be performed in order to establish an interaction. It comprises the following phases: exporting service definitions, negotiating service definitions, and establishing a communication channel.

Rebinding of interacting components enables system adaptation to changed conditions. This is performed as a sequence of four steps: exporting service definitions, renegotiating service definitions, destroying a binding, and reestablishing a communication channel. We have distinguished four rebinding situations according to which component initializes the process, and which performs the renegotiation of a new peer.

QoS Management, dealing with non-functional behaviour of a component, is an essential part of support for applications in open dynamic systems. In our model, it consists of three phases performed in sequence during a component's lifetime: *QoS Definition, QoS Negotiation,* and *QoS Maintenance.*

Every proposed framework must be based on well-defined and well-known research terminology — we have derived our terminology from RM-ODP standards [66, 67, 68]. It is important to clarify the difference between these approaches.

The concepts defined in the RM-ODP standards such as the trader and quality of service management are described independently and in very broad terms. Our approach, MAGNET, attempts to unify the relevant concepts and tailor the RM-ODP framework to its primary purpose — dynamic resource management. Our definition of the Trader does not follow the RM-ODP standard [67], which distinguishes between two operations: service export and service import. Also, unlike the RM-ODP standard, our rebinding approach emphasizes issues concerning the renegotiating phase. Also our approach to QoS Management significantly differs from the RM-ODP standard [68]. In particular, it emphasizes the negotiation phase and moves the target of the problem from maintaining agreed QoS to providing a QoS-based negotiation and selection.

Having defined an abstract model for trading, in the next chapter we can focus on a design of an actual architecture implementing the trading functionality.

Chapter 4

A Resource Management Architecture

In this chapter we present the design of MAGNET — an architecture for third-party dynamic trading of service provisions and requirements in open distributed systems with frequently changing characteristics. The model and terminology for this architecture were described in chapter 3, here we focus on the design of MAGNET. For reasons of clarity and readability we have split the description of the architecture into two chapters, 4 and 5.

This chapter describes the design of the core of the MAGNET framework. Advanced features, such as information monitoring, QoS Management, dynamic rebinding and scalability, are discussed in chapter 5.

Firstly, we summarize the requirements of the architecture outlined in chapter 1. In section 4.2 we discuss our reasoning for the chosen infrastructure model. Then, in section 4.3 we define functional semantics of system elements, and describe the binding process established using MAGNET (section 4.4). Finally, we focus on naming and protection issues, in sections 4.5 and 4.6.

4.1 Requirements for Dynamic Resource Management

Fundamental changes in computing environments have affected the role of resource management. Here, we summarize the requirements of dynamic resource management, as discussed in chapter 1, to refresh the main goals MAGNET attempts to achieve.

Dynamic Trading

The primary role of the resource manager is to enable resource allocation — dynamic binding performing component coupling based on information on the type of service. Therefore, the system must provide a third-party component, the Trader, collecting information on services and matching requests against demands.

Extensibility

The architecture should not constrain the format nor semantics of its data, and should enable user-customization of the matching process.

QoS-based Management

Unconstrained description of services and user-customization of the matching process enables support for QoS-based management. This provides QoS-based resource description and parametrized QoS-based matching process.

Dynamic Rebinding

In order to support runtime adaptations to changes of system environment, MAGNET should support dynamic rebinding of both types — *first-party rebinding* (initiated by component itself) and *third-party rebinding* performed by a managerial third-party with knowledge of overall application semantics. In addition, rebinding should also enable *first-party renegotiating* (leaving the selection of a new component on the unbound peer), or presenting the rebound component with appropriate replacement which can be found either in the Trader (*third-party renegotiation*), or obtained from an external entity (*no-renegotiation* is required).

Information Monitoring

Monitoring resource characteristics is a crucial requirement of dynamic resource management, enabling varying system features and time-dependent information to be kept up-to-date in the Trader.

Scalability

Scalability is an essential feature of all open distributed systems. The framework must enable mobile users dynamically joining and leaving the system to use its full potential in a local scale, and also provide support for scaling.

4.2 Using the Tuplespace Paradigm for the Trader

As was defined in chapter 3, in the MAGNET architecture, the key component in third-party role performing the service coupling is called the *Trader*. In order to provide an information infrastructure for trading service properties, the Trader must contain a shared data repository available to all components (however, it is not directly accessible). We call this data structure, derived from the tuplespace paradigm[1] [22], an *information pool*. Structured data items placed into the information pool are *tuples*. In this section we describe the design of the Trader and the information pool, and discuss our reasons for choosing this paradigm.

4.2.1 Overview of the Trader

The Trader is the key component of the MAGNET architecture available to all components, such as system services, hardware resources, and mobile users, for establishing dynamic bindings. 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 communication).

Figure 4.1 illustrates the structure of the Trader, and its three components.

[1] Speaking strictly about the data structure, the information pool is actually a tuplespace. However, the term 'tuplespace' is often associated with the Linda distributed programming language [22]; therefore, we decided to call our data structure 'information pool' to avoid confusion.

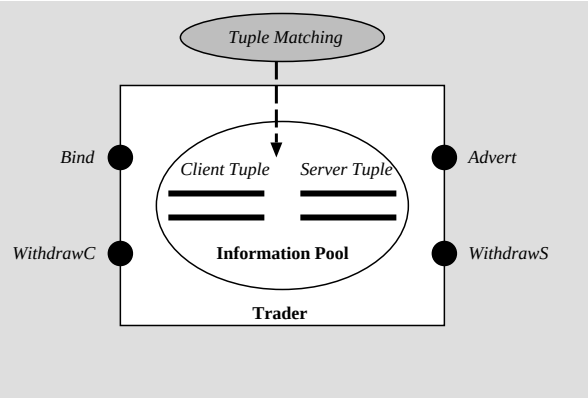

Figure 4.1: The Trader Structure

4.2.2 The Information Pool

The information pool is a distributed data structure accessible by all components using MAGNET. As was mentioned above, the information pool design was influenced by the notion of a *distributed tuplespace* [22]. Data items (tuples) can be inserted in, or withdrawn from, the tuplespace by a set of clearly defined operations. The internal organization of the structure is not defined, and is irrelevant for the framework semantics.

Tuples describing requirements and provisions for resource management often contain additional 'non-matching' information, such as interface references for accessing offered services, or requirements on the establishment of the communication channel. For clarity of the framework, it is desirable to express this information in the form of tuple elements. Therefore, the tuple (defined below in Def. 1) distinguishes between the number of all tuple elements n and the number of matching elements m. Traditional tuples containing only matching elements simply set $n = m$; neither the semantics of the structure nor the matching process have to be modified.

A tuple, a structured data item, is defined below (N is the set of natural numbers).

Definition 1. A ***tuple*** T is an ordered set of (n+2) elements $T = (n, m, p_1, p_2, ..., p_n)$, $n \geq m$ where $n \in N$ is the number of tuple elements, $m \in N$ is number of 'matchable' tuple elements, and $p_i \in P_i$ are values of tuple elements, all actual parameters.

4.2.3 The Trader Operations

The information pool must be equipped with the Trader operations defining the semantics of manipulation of tuples, such as insert and delete. Linda [22] is a well-known distributed programming language defining the original set of operations on tuples built around the traditional tuplespace (IN, OUT, READ).

Although MAGNET's Trader is based on the tuplespace paradigm, the operations and their semantics were redefined and extended to better meet requirements of users in dynamic environments. Operations offered by the Trader include: BIND, ADVERT (implementing service export), and WITHDRAWC, WITHDRAWS (implementing service withdraw).

Semantics of the Operations

The crucial feature of Linda, which the Trader has inherited, is that it does not treat communicating components equally, but distinguishes between the roles of client and server. Therefore, the Trader operations also express this 'duality' of character by providing *client-operations* (BIND, WITHDRAWC) used to manipulate *client-tuples*, and *server-operations* (ADVERT, WITHDRAWS) for manipulating *server-tuples*. Tuples themselves are syntactically identical (following Def. 1); client-tuples and server-tuples are identified by the operation used to insert them into the pool. In order to ensure the matching operation is performed only on complementary-type tuples, information about tuple type must be preserved. In section 4.2.5 we will argue for this feature in greater detail, here we define the semantics of the four fundamental operations provided by the Trader.

Definition 2. Operation ***Bind (T),*** where T is a client-tuple (Def. 1), searches the information pool for a complementary-type matching tuple (Def. 6). 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 one becomes available.

Definition 3. Operation ***Advert (T),*** where T is a server-tuple (Def. 1), results in inserting the tuple into the information pool. It also searches the pool for all complementary-type matching tuples (Def. 6). If such tuples are found, they are removed from the pool, and returned to the calling server component.

Definition 4. Operation ***WithdrawC (T),*** where T is a client-tuple (Def. 1), results in removing tuple T from the information pool.

Definition 5. Operation ***WithdrawS (T),*** where T is a server-tuple (Def. 1), results in removing tuple T from the information pool.

Operations WITHDRAWC and WITHDRAWS do not perform the matching operation (Def. 6) restricted to a subset of the tuple elements, but find an equal one (all tuple elements are checked).

Like in Linda, all these operations are performed atomically and selection from more than one matching tuple currently available in the information pool is performed non-deterministically, unless defined by components themselves in terms of preference rules.

Discussion

In order to allow user-customization, blocking the calling component is not performed by the Trader, but left to components themselves. According to the nature of the application, a component can block itself immediately, or after a sequence of 'insert' operations.

Operations WITHDRAWC, WITHDRAWS are semantically identical. The only reason why we did not define one WITHDRAW operation for both clients and servers is performance. As the information pool is typically very large, it would be inefficient to search all tuples (client and server ones) when it is clear which tuple type is being searched for. As we define the actual operations used by components, there is not any 'lower' layer at which this information could have been passed into the pool.

The stateless character of tuples in the pool (discussed in greater detail in section 4.2.5) enables the Trader not to worry about the state of possible ongoing bindings while performing WITHDRAWS and WITHDRAWC operations. This information is

maintained by the communicating components themselves, and is usually ensured by their communication protocol.

4.2.4 The Tuple Matching

The primary purpose of the framework is not to store information, but to provide trading of data placed into the repository. This communication model is known as *generative,* because a tuple generated by a component has an independent existence in the tuplespace until explicitly withdrawn by any component [22].

Tuple matching[2] is a concrete implementation of the matching process discussed in chapter 3 which enables actual communication between components in the form of exchanging information in tuples. In the classical tuplespace, exact matching only was supported [22]. Definition 6 defines the matching process in the MAGNET framework which, in addition, enables matching to be performed on a subset of tuple elements.

Definition 6. A client-tuple $T_1 = (n_1, m_1, p_1, p_2, ..., p_n)$, $n_1 \geq m_1$, where $n_1, m_1 \in N$, $p_i \in P_i$ and a server-tuple $T_2 = (n_2, m_2, q_1, q_2, ..., q_n)$, $n_2 \geq m_2$, where $n_2, m_2 \in N$, and $q_i \in Q_i$ ***match*** iff $m_1 = m_2$ & $(P_i = Q_i$ & $p_i = q_i)$ for $\forall i \in \{1, m_1\}$.

As incorporating non-matching information into tuples is optional, and may differ between a client-tuple and a server-tuple, the equality of tuple size ($n_1 = n_2$) is not a required matching condition.

The matching operation, as it is defined, is not a symmetrical operation — it does assume client and server roles for the matching tuples. Here we discuss the basic (exact) matching process. User-customization of the matching process is covered in chapter 5.

4.2.5 Reasoning about the Trading Paradigm

In order to justify the tuplespace framework chosen for our information pool design, we have to discuss key characteristics of the paradigm; in particular, the necessity of distinct roles for client and server. Finally, we compare our approach to Linda and a traditional namespace and clarify their differences.

Key Characteristics

Like a tuplespace, the information pool supports the following communication features: *multi-party asynchronous communication*, *stateless* character of tuples, and *decoupling* of the communication parties permitting *free-naming*. More general asynchronous communication prevents applications from forced undesired synchronization. The stateless nature of tuples saves the Trader from having to provide a state-maintenance scheme; for example, checkpointing or recovery procedures. In addition, it improves the generality of the system. If state is required, it can be incorporated as a parameter of tuples. Decoupling the server from the client by the Trader permits communication to proceed anonymously, therefore servers can produce tuples of interest to any client. Consequently, this feature enables free-naming — communication can be established without previous knowledge of the other party's identity. Similarly to the states of tuples, names can be expressed as parameters of tuples if required. All these features provide additional flexibility over traditional direct one-to-one communication schemes.

[2] Tuple matching is also called the *Matching Function* or the *Matching Operation.*

The Necessity of Distinct Roles of Client and Server

As was said above, distinct roles of client and server assigned to interacting components is one of the crucial feature of the Trader. Why is this sacrifice of generality necessary?

The primary purpose of the Trader is to enable an information exchange for establishment of component bindings. A component binding, as defined in chapter 3, is an interaction between two parties — client and server, not equal system elements. In order to tailor our framework to this model of binding, the roles of client and server must be incorporated into the Trader semantics. Therefore, the operations provided by the Trader — BIND, ADVERT for service export, and WITHDRAWC, WITHDRAWS for service withdraw — reflect the duality of client and server roles by providing 'built-in duality'. This corresponds to Linda's view of tuples represented by operations IN and OUT [22].

In contrast with this dual approach, general data structures (such as 'heap') treat all data items equally by providing operations with no additional semantics (such as INSERT and DELETE).

Therefore, the built-in semantics approach outweighs the loss of full generality. As the contents of tuples are not predefined and the matching process can be user-customized (see chapter 5), distinguished roles do not constrain the extensibility of the Trader.

The Trader Operation Set versus Linda

As the Trader was motivated by the Linda programming language, it is desirable to compare these two approaches. In order to clarify our discussion, we briefly summarize Linda's operations below (details can be found in [22]):

OUT$(N, P_2, ..., P_j)$, where $P_2, ..., P_j$ are parameters (actual or formal) and N is an actual parameter of type name, results in inserting of the tuple $(N, P_2, ..., P_j)$ into the tuplespace; the process (which called the operation) continues immediately.

IN$(N, P_2, ..., P_j)$ where $P_2, ..., P_j$ are parameters (actual or formal) and N is an actual parameter of type name. If a type-consonant tuple whose first component in N exists in the tuplespace, the tuple is withdrawn, the values of its actuals are assigned to the IN()-statement's formals, and the process executing the IN()-statement continues. If no matching tuple is available, IN() suspends until one is available and then proceeds as above.

READ$(N, P_2, ..., P_j)$ is identical to the IN()-statement except that, when a matching tuple is found, assignment of actuals to formals is made as before but the tuple remains in the tuplespace.

The key similarity was discussed in the previous section — both Linda and the Trader enforce components to become client or server, by providing a set of dual operations (e.g., IN and OUT in Linda; BIND and ADVERT in the MAGNET Trader).

However, there are also many significant differences between these two approaches. Here we highlight several fundamental ones. Firstly, there is the differences in semantics of the operations. For example, ADVERT matches all tuples inserted by BIND waiting in the pool, while OUT matches only one waiting tuple inserted by IN; a tuple inserted by OUT can be removed from the pool by any other component calling IN, while in the Trader clients calling BIND have no right to remove a server tuple inserted by ADVERT. (Semantically, operation BIND is equivalent to operation READ).

Secondly, tuples in the information pool can be 'signed' for identification purposes. This does not constrain the tuple format, as the 'name' can be expressed as one of the non-matching tuple elements (Def. 1), nor does it restrict the free-naming feature (that is requesting a service by its type) because the matching process can be restricted to a subset of the tuple elements, excluding the 'name' (Linda does not support this feature).

Also, a particular tuple can be withdrawn from the pool by operations WITHDRAWC and WITHDRAWS which can identify it by searching for an equal tuple. In Linda, designed primarily for decoupled communication, any component can withdraw any tuple by calling an appropriate complementary operation (IN removes OUT tuple; and OUT removes IN and READ tuples), without paying attention to an exact tuple identity.

In addition, the Trader does not support substitution of formal parameters because there is no need for this kind of communication in the binding process.

The Information Pool versus a Namespace

A *nameservice* 'maps a name for an entity (an individual, organization or facility) into a set of labeled properties[3], each of which is a string. It is the basis for resource location, mail addressing, and authentication in distributed computing systems' [37]. A nameservice is based on a data structure (a repository of addresses) called a *namespace.*

The primary difference between the Trader and a nameservice lies in the distinguished roles of client and server discussed above. A namespace keeps equal data items, while the information pool consists of server-tuples and client-tuples. In addition to this fundamental difference, there are several other important semantic nuances.

Firstly, data items in the namespace can be considered as a *static 'pair'* (name and address), while in the information pool, client-tuples are independent of complementary server-tuples, and only the matching operation joins them into a *dynamic 'pair'*. Secondly, name—address mapping can be classified as one-to-one mapping, unlike client-tuple—server-tuple matching which is defined as one-to-many. Thirdly, name and address are semantically nondetachable in the namespace (an address with no name does not have a valuable meaning and vice versa), while client-tuples and server-tuples have an independent existence in the pool.

In addition there are three more minor differences: a nameservice requires unique naming, supports only simple name mapping and considers global scalability an important issue. The Trader, in contrast, supports optional names expressed as tuple elements, enables parametrized QoS-based matching, and is primarily designed for a local scale, a federation.

4.3 Components for the MAGNET Architecture

Having described in detail the core approach undertaken in MAGNET's architecture (the Trader based on the tuplespace paradigm), now we present an overview of the framework, and describe individual system components.

Figure 4.2 illustrates the structure of the MAGNET architecture distributed over a single federation. The system consists of four classes of component: *the Trader*, *Client*, *Server and Tree* (components performing the matching process). There is only a single instance of the Trader component, in contrast to multiple instances of Client, Server and Tree. In addition to these four high-level components, there are two types of subcomponent performing dedicated functions: these are a pair

[3] often called 'addresses'

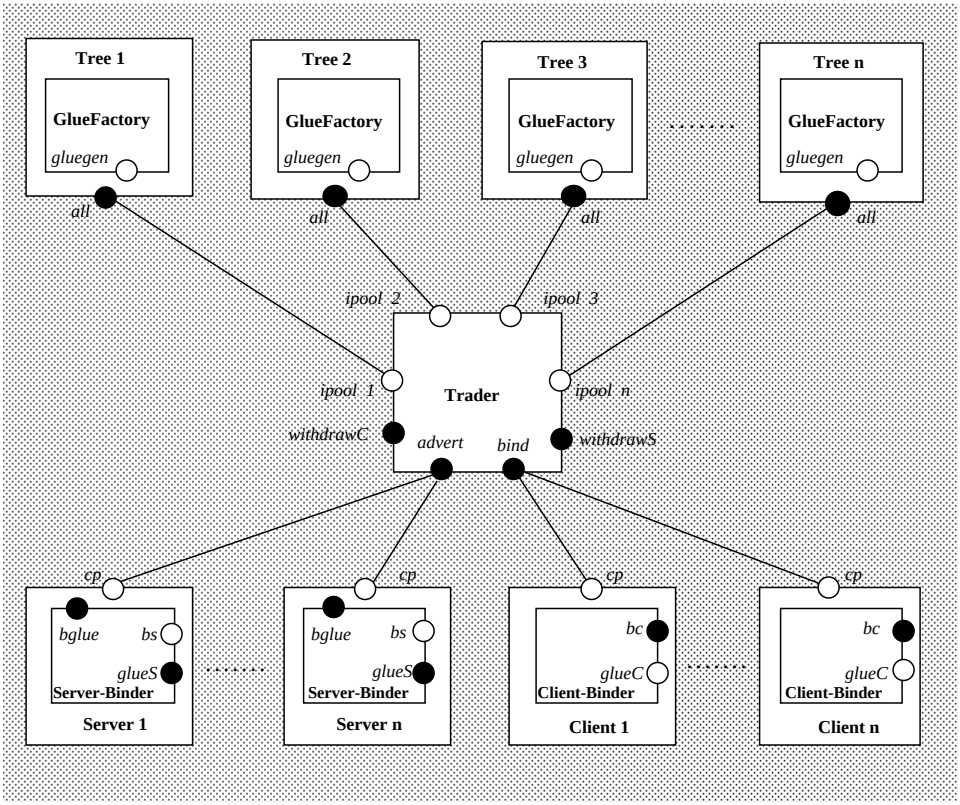

Figure 4.2: MAGNET's architecture

of *Binders* (the *Client-Binder* and the *Server-Binder*) present in all Clients and Servers; and the *GlueFactory* included in all Trees. Binders in cooperation with the GlueFactory establish the resultant client-server binding.

In addition to the static components, Figure 4.2 also illustrates the interconnections between them. In this section we will describe the functionality of every individual component, while in section 4.4 we will focus on their mutual interaction that realizes the primary goal of the framework — the binding process.

4.3.1 The Trader

The Trader is the key system component offering four fundamental operations (BIND, ADVERT, WITHDRAWC, WITHDRAWS) used by clients and servers requesting to establish dynamic bindings. As the framework is primarily designed for resource management purposes, it is desirable to have one centralized Trader component within a federation keeping information of the available resources. This is sufficient as resource allocation is mainly performed within a given domain (however, distribution of the information pool preventing the Trader becoming a bottleneck in the system is discussed below).

Other option would be running the Trader component on every system processor and 'escalating' (forwarding) tuples which do not match locally to other Traders. However, this would cause consistency problems (e.g., what happens if a single tuple is matched in more than one Traders? What is the topology of Trader intercomection and how is it dynamically adapted?) As resource allocation *is* primarily domain-scale problem (e.g., most of requests for resources are fulfilled in a local Internet domain), we have chosen the federation to be the smallest architecture en-

tity. However, scalability of the framework in terms of inter-Trader communication is also supported. We discuss these issues in the next chapter.

In order to prevent the Trader being a bottleneck in the system, the four operations are actually implemented in distributed Tree components. The Trader forms a *single interface* to all components calling its operations, while multiple physically distributed Tree components actually *implement* these operations. The Trader forwards the request (the operation and the tuple) into the appropriate Tree for processing. The inner structure of Trees ensures that there is only one Tree component able to fulfill the request.

However, the configuration of Trees and the type of tuples they hold is determined by the Trader, therefore they cannot be accessed directly by components.

From the semantic point of view, distributed Tree components form the information pool, and are an indispensable part of the Trader functionality. For this reason, in the previous section 4.2.5 we considered the Trader as a single component. However, structure-wise, the information pool is physically distributed into Trees, therefore in this section, as we are dealing with the system structure, we treat the Trees as independent system components. Figure 4.2 illustrates the relationship between the Trader and Trees.

4.3.2 The Tree

As was said above, Trees provide the actual functionality by implementing the Trader operations, yet are not accessible by components directly. As was outlined in section 4.2.5, the information pool distinguishes between client and server tuples, therefore this feature must be also provided by every Tree component in order to ensure the matching process will be performed between complementary tuples. Trees perform the requested operation forwarded from the Trader and, if the matching process is successful, initialize the last phase of the binding process — establishment of a communication channel — implemented by a dedicated subcomponent, the Glue Factory.

4.3.3 Distribution Issues

The Trader acts as an interface filter for all components involved in resource management and is responsible for distribution of Tree components. The initialization of the Trader in a particular domain performs an instantiation of Tree components which are physically distributed over available processors. The number of processors, given by a system administrator initializing the Trader, determines the way Trees are distributed. These issues are discussed in section 6.5.

4.3.4 The Glue Factory

If the matching process called by the operations ADVERT and BIND finds a match, client and server components must be informed about each other in order to establish a binding, which is the final goal of this procedure. A dedicated component responsible for informing components of matching tuples is the Glue Factory, a subcomponent of every Tree.

When the Glue Factory is given a matching tuple pair (a result of a successful matching process), it establishes a communication with the Server-Binder (the interface reference is obtained from the server-tuple), and is responsible for passing over the client-tuple.

Information about the established server-client pair is not kept by the GlueFactory, nor the Tree, in order to ensure the stateless nature of the Trader. As said above, this functionality belongs to the scope of components' responsibility.

4.3.5 Client and Server

Every component using MAGNET must interact with the Trader as a client which requests the four Trader's functions. Therefore, every component is equipped with an additional service interface (`cp`) through which it calls the Trader's operations — advertises or binds its tuples.

In Figure 4.2 the client's service interface is labeled `glueC`, while the server's service interface is `glueS`. In addition, both client and server must also contain a dedicated subcomponent — the Binders — responsible for establishing the binding.

4.3.6 Binders

The primary purpose of Binders is to perform the last phase of the binding process — the establishment of a communication channel. On this task, Binders cooperate with the underlying computing environment where MAGNET operates: Binders form the MAGNET environment-independent interface, while the resultant binding at the low-level is established by a communication mechanism available in the system (such as a message passing, etc.)

From the semantic point of view, there are two classes of Binder — Server-Binders and Client-Binders. The service interface to the Server-Binder (`bglue`) is inserted into the tuple sent to the Trader by an operation ADVERT. When matching of service definitions is achieved, the Server-Binder is provided with the matching client-tuple containing an service interface to the Client-Binder (`bc`) of the client component. Then a binder-to-binder interaction can be established to exchange required information to enable the final end-to-end client-server binding to take place. The actual protocol and semantics of Binders is application dependent, and for MAGNET's purposes irrelevant.

If components do not trust each other, Binders might also contain a built-in admission protocol, providing protection for servers from untrustworthy clients. This feature is discussed in section 4.6.

4.4 The Binding Process

Having discussed the functionality of MAGNET's components, we focus on their interaction resulting in dynamic binding. In chapter 3, we defined three phases of the binding process: export service definitions, negotiating service definitions, and establishing a communication channel. In this section we will discuss these three phases as they are performed by the MAGNET framework.

In this chapter we discuss a *local trading* system; *federations*, as a solution to the problem of scaling, are covered in chapter 5.

Figure 4.3 illustrates a simple binding configuration consisting of a server (the Printer), a client (the Application), the Trader and the Tree component into which the Printer's and the Application's tuples are forwarded for processing. For simplicity, remaining distributed Tree components are omitted, as they are irrelevant to this particular instance of the binding process. We define and illustrate each phase of the binding process in this example. Detailed description of each particular phase (in this example) is given at the end of each section defining the phase.

4.4.1 Export Service Definitions

The goal of the first phase of the binding process — export service definitions — is to place services into the Trader where they are negotiated. In our framework, the Trader offers four functions (BIND, ADVERT, WITHDRAWC, WITHDRAWS). In order

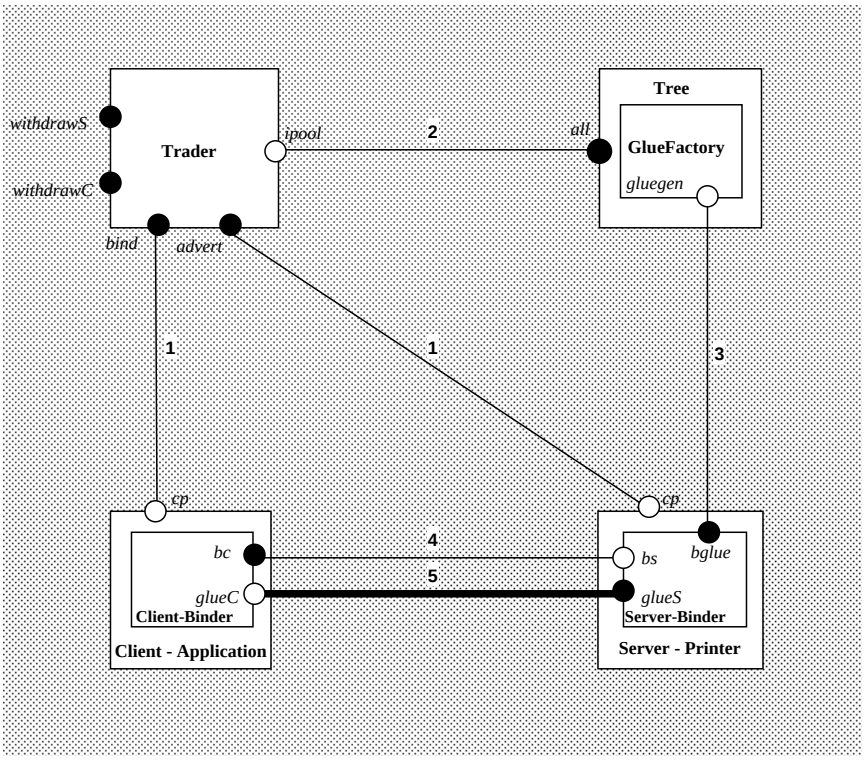

Figure 4.3: Binding establishment in MAGNET

to call them (they are represented as services provided by the Trader), components need to bind to the Trader component. However, in this 'first case', it cannot be the Trader, which establishes the binding, as it is itself being bound. Therefore, a nameserver resident at a well-known address provides components with interface references to the Trader's functions so that the component-Trader bindings can be established. The Trader forwards incoming tuples into the appropriate Tree where the required operations are performed.

Example

In our example, the Application describes its requests by a tuple:

$T1 = (4, 3, printer, laser, 600dpi, bc)$

(4 determines the tuple size, 3 sets the number of elements which have to match). The application exports service by calling the Trader's operation BIND($T1$). The Printer describes its offer by a tuple:

$T2 = (4, 3, printer, laser, 600dpi, bglue)$

(again, 4 determines the tuple size, 3 sets the number of 'matching' elements, therefore service interfaces `bc` and `bglue` used for locating components are omitted at the matching process). It exports service by calling the Trader's operation ADVERT($T2$). Performing operations in this order results in forwarding tuples $T1$ and $T2$ into the Tree, where the next phase, the matching process, takes place.

4.4.2 Negotiating Service Definitions

The goal of this second phase is to perform the matching process that searches the information pool (the particular Tree) for a matching tuple. MAGNET's export policy consists of operations ADVERT and BIND; the withdraw policy consists of WITHDRAWS and WITHDRAWC. In addition, the tuple format (Def. 1) for accessing the Trader's operations is enforced.

The matching process consists of two phases — operations *search* and *select*. The matching function (Def. 6) defining the matching rules can be user-customized. For each pair of tuples an equality of the 'matching size' m (Def. 1) is checked (3=3, in our example); then for each tuple element, the type of the element and the value are compared (value comparison can be user-customized, therefore strict mathematical equality is not required). In our example, three comparisons are performed (printer=printer, laser=laser, 600dpi=600dpi). If this procedure succeeds for all tuple matching elements, the two tuples are said to match, and are forwarded to the GlueFactory.

Removing or inserting matching tuples into the pool is performed in accordance with the particular operation definition (e.g., a matching ADVERT tuple is inserted into, or left in, the pool, while a BIND tuple is not inserted, or is withdrawn from it, etc.)

Example

In our example, tuples $T1$ and $T2$ match, therefore, they are passed into the GlueFactory; the server tuple $T2$ remains in the Tree, while the application tuple $T1$ is withdrawn as its request is satisfied.

Here we have discussed basic exact matching. Advanced QoS-negotiation which enables the user to further customize the matching rules and define the preference rules is discussed in chapter 5.

4.4.3 Establishing a Communication Channel

The establishment of an end-to-end binding is performed by the particular GlueFactory in cooperation with both Binders, the Client-Binder and the Server-Binder. The GlueFactory passes over the client-tuple to the Server-Binder via a binding established temporarily for this purpose. The interface reference of the Server-Binder, necessary for establishment of this temporary binding, is obtained from the server-tuple.

Then, the Server-Binder invokes the Client-Binder (the interface reference is obtained from the client-tuple), and performs the binder-to-binder protocol, passing over the required server interface reference, so the client can bind to it, the resultant communication channel is established; and components can start interacting.

Example

In our example, the `gluegen` request service is bound to the `bglue` provision service, and via this temporary binding, the tuple $T1$ is sent to the Server-Binder. Then, the Server-Binder's service reference `bs` is bound to the Client-Binder's reference `bc`, over which the resultant communication channel between server provision reference `glueS` and the client request reference `glueC` is established.

4.5 Naming

Naming is an important issue in open distributed scalable systems. In MAGNET's framework, there are three levels at which naming must be considered: *tuple naming*, *interface reference naming*, and *Trader naming*.

4.5.1 Tuple Naming

As the Trader provides an environment for *generative* programming (discussed in section 4.2.4), tuples are considered *anonymous*. Therefore, the tuple definition (Def. 1) does not enforce any names. As was discussed above, this feature enables free-naming — accessing a peer component without previous knowledge of its identity. An identification which is necessary for accessing services (interface references) is expressed as a tuple element, invisible to the high-level MAGNET framework. Therefore, no dedicated naming scheme for tuples is necessary.

4.5.2 Interface Reference Naming

In order to establish a binding between service interfaces of two components, they must be identified by interface references. As components offer or require unique services, naming of their interface references must be unambiguous. The design of such a naming scheme depends on the particular computing environment and identification scheme used within it, and can vary between different applications using the same Trader.

For example, the typical computing environment where MAGNET can operate is the Internet. Therefore, 'names' of interface references can be composed as a combination of an IP address, a process number (PID), and an internal process (component) reference which enable unambiguous scalability. The chosen interface reference naming scheme, and its mapping derived from a particular computing environment, are irrelevant for MAGNET's design.

4.5.3 Trader Naming

The third level at which naming is an issue in MAGNET is the Trader level. There are two different 'scales' to consider — the local domain consisting of one Trader, and federations enabling scaling of the architecture.

Firstly, a local domain (as assumed throughout this chapter) consists of only one Trader serving distributed Trees. The Trader is contacted via a shared nameserver resident at an address known in advance. Therefore, in the local domain Trader naming is not an issue.

Secondly, in order to enable scaling of the architecture, Traders are internetworked to connect federations. There must be a naming scheme enforced to allow components to identify Traders in remote domains, in order to enable Trader-to-Trader communication. Issues related to scalability and the actual design of federations, including naming of federated Traders, are covered in chapter 5.

4.6 Protection

Protection in open distributed systems is always a complex issue. MAGNET's protection scheme targets the following crucial areas: *Trader protection*, *tuple protection*, and *components protection*.

4.6.1 Trader Protection

In order to allow the Trader to distribute Tree components across the domain transparently, Trees are not directly accessible by components. Therefore, Tree service interfaces (`all`) are hidden by the public Trader's operations (service interfaces: `advert`, `bind`, `withdrawC`, `withdrawS`).

4.6.2 Tuple Protection

In order to enable free-naming, tuples are considered anonymous. This feature allows any client to insert a type-complementary tuple that matches any server tuple in the information pool. The matching process does not include any protection checks.

Frequently, additional protection is necessary, as components using MAGNET do not necessarily trust each other. Therefore, a tuple can be protected by a 'password' expressed as one of its elements. Trustworthy components know the password (from system designers, or administrators) and are able to produce a matching tuple, while untrusted components cannot. As a result of this, the distributed system can be shared by groups of components (possibly overlapping) able to produce only mutually matching tuples and gain access to services only within their groups.

Also, capabilities could be expressed as tuple elements, if this way of protection is required by system administrators or applications using MAGNET. Nevertheless, the universality of the architecture is preserved.

As the protection of tuples using signatures or capabilities is user-defined, it does not require altering the high-level design of the framework, nor the matching process.

4.6.3 Component Protection

In order to prevent service references from being directly passed to untrustworthy components, bypassing the signature-matching process, components are equipped with two additional means for their self-protection. Firstly, interface references can be protected by Binders; secondly, an admission protocol carried out by Binders can result in refusing service to untrusted components.

Interface Reference Protection

All bindings are established using the pair of Binders (Client-Binder and Server-Binder). In addition to this primary purpose, Binders prevent interface references from being advertised, and therefore exposed for misuse. Therefore, Binders' service interfaces (`bglue` in Server-Binder and `bc` in Client-Binder, as illustrated in Figure 4.3) are inserted into tuples which are placed into the information pool. Service interfaces of the actual services provision (`glueS`) and requirement (`glueC`) are never advertised, and can only be accessed by trusted Binders.

Admission Protocol

Protecting interface references from being misused by untrusted components by hiding them behind the Binders can only have the desired effect if the Binders have the right to refuse to establish a binding. As Binders' interface reference can be passed over to any untrusted component, an *admission protocol* between Client-Binder and Server-Binder is required.

When the Server-Binder obtains the client-tuple containing the reference to the Client-Binder, it performs a protection checking procedure of varying complexity. That is, the admission can be based on simple user identification (UID), public key

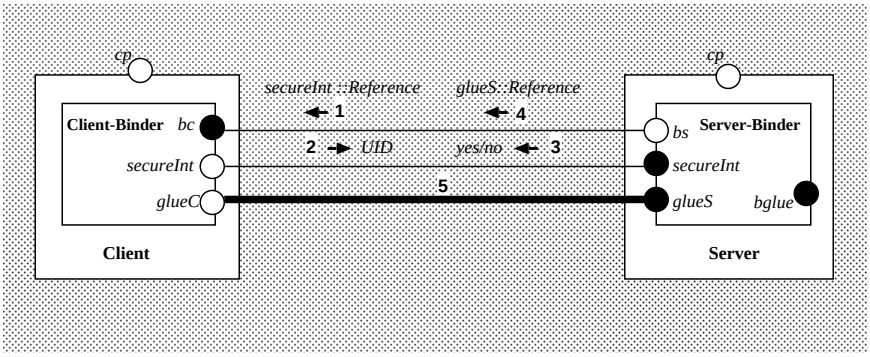

Figure 4.4: Admission Protocol

(PGP), or any other common or proprietary protocol. Only when the admission protocol is successful is the access to the service is gained, and the required end-to-end client-server binding is established.

Figure 4.4 illustrates an admission protocol between Binders based on classical UID protection.

4.7 Chapter Summary

Based on the model defined in chapter 3, we have described the core of MAGNET's design, leaving its advanced features to chapter 5.

MAGNET provides a framework for establishment of dynamic binding using third-party trading. Based on a tuplespace paradigm, the Trader consists of three components: the *information pool* (a tuple-space like data structure), the *Trader operations* on tuples for their manipulation, and the *tuple matching* operation.

The information pool is a distributed data repository for structured data items — tuples consisting of an ordered set of items. The set of operations provided by the Trader consists of: BIND, ADVERT, WITHDRAWC, and WITHDRAWS. In this chapter we have provided definitions of these operations and of the matching process. The key characteristics of the chosen paradigm include *multi-party asynchronous communication*, *stateless* character of tuples, and *decoupling* of the communication parties permitting *free-naming*. In other words, communication can be established without previous knowledge of the peer identity. We have also argued why it is necessary to distinguish between the roles of client and server, and we have compared the Trader to Linda, and the information pool to a namespace.

MAGNET consists of four classes of components: the *Trader*, *Client*, *Server*, *Tree* and two types of dedicated subcomponent: a pair of *Binders* (*Client-Binder* and *Server-Binder*), and the *GlueFactory*. The Trader is the key system component providing an interface to the four operations on tuples. According to a local site configuration, the Trader initializes distributed Trees where the actual operations are performed. As they belong to an internal structure of the Trader, for protection reasons, they are not accessible by components directly. If the matching process, implemented by the Trees, is successful, the GlueFactory subcomponent (present in every Tree, dedicated to the establishment of a binding) cooperates with Binders (present in every component) in order to establish the binding. Then the Client-Binder and the Server-Binder perform the final phase of the binding process, resulting in the establishment of a communication channel between client and server.

Having discussed the semantics of all these components separately, we have

described their interaction which performs the actual binding process: exporting service definitions, negotiating service definitions, and establishing a communication channel. We illustrated each particular phase of the binding process using an example of a Printer and an Application components.

Naming in MAGNET is considered at three levels: *tuple naming*, *interface reference naming*, and *Trader naming*. As tuples are considered anonymous, no names are enforced at the high-level. Interface reference naming requires an unambiguous scalable naming scheme and depends on the actual computing environment where MAGNET is being used. Naming of Traders, interconnected into federations, was mentioned in this chapter, and will be discussed in chapter 5, together with scalability issues.

Protection is another indispensable requirement of open distributed systems. In MAGNET, we provide protection at three levels: *Trader protection* (hiding the Tree components from users by the public interface provided by the Trader), *tuple protection* (enabling components to incorporate 'signatures' into their tuples in order to prevent them from being misused by untrusted components), and *component protection* (hiding the actual service interace reference behind publically advertised Binder references, enabling an admission protocol to be performed by the Binders).

This chapter has covered the design of the core system elements and their functionality. The next chapter discusses the design of MAGNET's advanced features, including information monitoring, QoS Management, rebinding and scalability issues.

Chapter 5

Advanced Features of the Architecture

Besides the basic functionality (described in chapter 4), MAGNET must support additional features permitting adaptation to the requirements of changing environments. Based on the fundamental system elements (defined in chapter 4), here we present the advanced features of the architecture: *information monitoring*, *QoS-based Management*, *dynamic rebinding*, and *scalability*. Information monitoring, an indispensable requirement of users in changing computing environments, is discussed in section 5.1. Section 5.2 covers issues of QoS Management, section 5.3 provides a description of MAGNET's support for dynamic rebinding. Finally, section 5.4 presents issues related to scalability of the architecture.

5.1 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. For this reason, the framework presented in this thesis is equipped with two additional components providing monitoring: the *Monitor* (monitoring server provisions), and the *Updater* (monitoring changing client requirements). In this section, we will describe the semantics of these two components and the actual monitoring process.

5.1.1 Components for Monitoring

As the MAGNET framework distinguishes between the roles of 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. Here we discuss their interface to MAGNET and expected functionality.

The Monitor

The task of the Monitor component is to observe changing characteristics of the server it is attached to, and keep the server tuple up-to-date. Figure 5.1 illustrates the structure of the framework with the Monitor.

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 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). However, in accordance with our assumptions, we expect the monitoring to be performed with frequency of minutes, rather than seconds and milliseconds.

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., a network throughput). For those, adaptation to change in conditions are 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 presisely, 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.).

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 third-party rebinding. Here we focus on the monitoring issues, while in section 5.3 we discuss the role of the Updater in the client rebinding process.

5.1.2 Monitoring

In this section we describe the monitoring process, as provided by the dedicated components: the server-attached Monitor, and the client-attached Updater. Figure 5.1 illustrates the bindings discussed below. Server-Monitor and client-Updater interactions are established statically in advance by a system administrator, not using MAGNET.

Monitoring Server Provisions

The Monitor component is attached to the server by a binding established between service interfaces `dataS` and `dataM`. The server keeps the Monitor informed about relevant changes. Then, according to the granularity of update (how often it is performed), and the 'out-of-dateness' accepted (how much can a tuple in the pool differ from current characteristics), the Monitor decides when to perform the operations WITHDRAWS and ADVERT. That is, the actual update in the pool (through service interfaces `cp` and `wp`). 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.

Monitoring Client Requirements

The Updater component is instructed by a client about service requirements it should search for. These two components communicate through a statically established binding between service interfaces `new` and `rebindC`. In this case, the

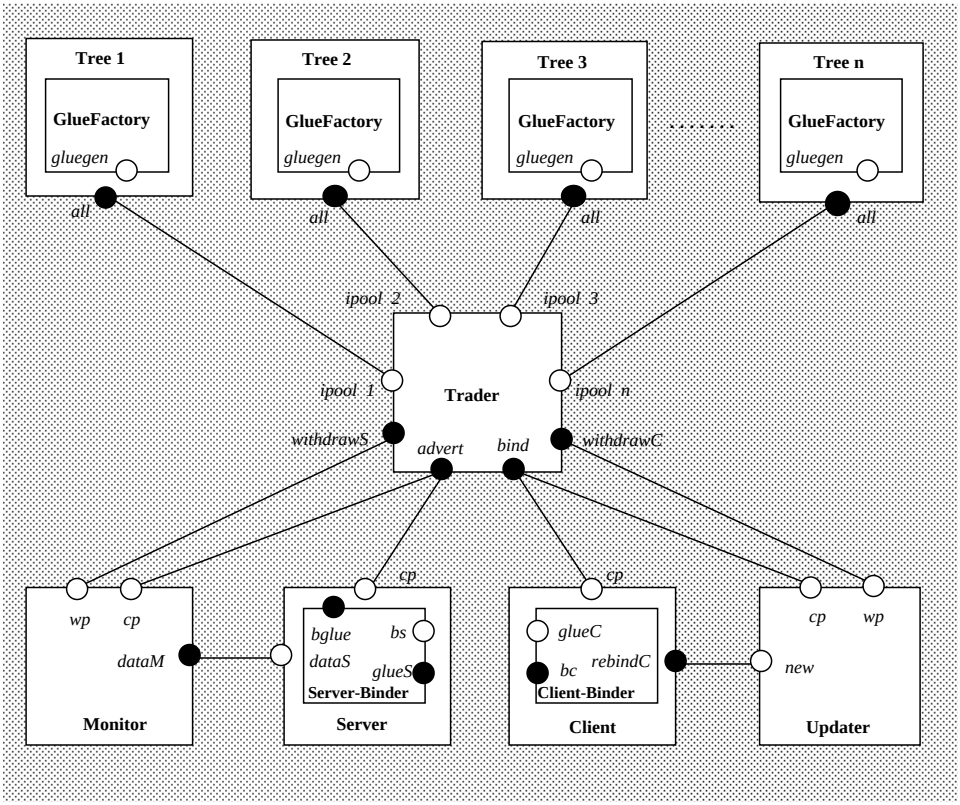

Figure 5.1: The architecture with the Monitor and the Updater

initiative is on the Updater component, in contrast to the Monitor that acts only when invoked by the server.

The Updater calls the operation BIND on a tuple with higher requirements (through service interface `cp`), 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. Rebinding issues are discussed in the section 5.3.

5.1.3 Discussion

There are two important issues related to monitoring that deserve further investigation. Firstly, we discuss issues related to monitoring of the communication channel in contrast to monitoring of component characteristics. Secondly, we briefly discuss the efficiency of the monitoring operation.

Component Characteristics versus a Communication Channel

Monitoring of system characteristic changes ensures the maintenance of an agreed level of service provided by the communication channel, as investigated for example in [16]. However, the primary focus of MAGNET is to present an environment for

resource allocation based on up-to-date information, rather than maintaining the agreed quality of a resultant binding.

Applications requiring the monitoring of a service provided by the communication channel can describe the channel as an additional component. This can be equipped with a Monitor that keeps channel tuples in the pool updated.

Efficiency

Data monitoring efficiency is an important issue. For applications requiring only course-grained monitoring strategies (with frequency of minutes), tuple updates performed by a withdrawal and reinsert (as discussed in this section) are sufficient. However, for applications requiring finer-grained updates of their data in the pool (with frequency of seconds and milliseconds), the complexity of the Trader operations must be added to the complexity of the update operation (the complexity of the operations will be discussed in chapter 6).

In order to improve efficiency, specific trusted Monitors and Updaters might be authorized to have direct access to the Tree holding their tuples. However, this solution fundamentally violates protection of the information pool (encapsulating Trees behind the public Trader's operations). For the reason of protection of other data in the pool, and protection of Trees that might be misused by untrustworthy Monitors, this approach is not a part of the framework design.

5.2 Quality of Service Management

Quality of service describes the non-functional behavior of the system components — characteristics under which service is provided. MAGNET's approach addresses QoS-based selection, targeting resource allocation, software upgrades, and dealing with dynamic characteristics of system resources, such as length of a printer queue, processor load etc.

According to the model defined in chapter 3, QoS Management is defined at three levels: *QoS Definition*, *QoS Negotiation*, and *QoS Maintenance*. In this section, we describe how MAGNET supports QoS Management, following the model discussed in chapter 3.

5.2.1 QoS Definition

The QoS Definition covers a wide range of system characteristics and their combinations. In our model, the QoS Definition comprises four levels: *characteristics level*, *service level*, *application level*, and *distribution level*.

A common formal approach to defining the QoS is the Z notation [61]. However, our approach, derived from Regular Expressions [3], extended with an 'evaluation' function expressing combination priorities (section 5.2.2), better suits the requirements of the QoS-based matching function.

In this section we discuss MAGNET's support for QoS Definition and illustrate its utility with simple examples.

Characteristics Level

Based on the tuple definition $T = (n, m, p_1, p_2, ..., p_n)$, $n \geq m$ where $n, m \in N$, $p_i \in P_i$ (Def. 1), resource parameters are represented as *tuple elements* $p_i \in P_i$. Typically, p_i gains discrete actual values within the definition range P_i.

However, in order to enable advanced QoS description, MAGNET also defines a set of QoS operators enabling parameterized description of service characteristics:

Definition 7. The ***set of QoS operators*** defined on tuple parameters $p_i \in P_i$ consists of:

1. operator ***interval*** p_i — p_j , where $p_i, p_j \in P_i$, and P_i is a linearly ordered set, gains any value $p_k \in < p_i, p_j >$.
2. operator ***negation*** $\neg p_i$ where $p_i \in P_i$ gains any value $p_k \in P_i$ & $p_k \neq p_i$ and operation equality $=$ can be defined according to the relation on the definition range P_i.
3. operator ***or*** p_i / p_j where $p_i, p_j \in P_i$ gains either value p_i or value p_j.
4. operator ***all **** gains any value $p_k \in P_i$.
5. additional user-defined operators.

As part of enabling user-customization of the tuple format, the framework allows customization of the set of QoS operators. The four operators have been predefined, as they are typical for QoS requests. Additional user-defined operators may be, for example $< x$ (values smaller than x), or $> x$ (values greater than x). The following tuples illustrate the usage of QoS operators:
$X1 = (3, 2, a - d, 555, ref)$
$X2 = (2, 2, a, *)$
$X3 = (3, 2, a|b, \neg 66, ref)$.

Service Level

As tuple elements (gaining operator-enhanced values, according to Def. 7), represent only partial information, the aim is to form a combination of parameters (forming the tuple) that defines the final service definition. It is necessary to extend the matching function in order to allow user-customized matching according to the set of operators on tuple elements:

Definition 8. Let T_1 and T_2 be tuples defined: $T_1 = (n_1, m_1, p_1, p_2, ..., p_n)$, $n_1 \geq m_1$, where $n_1, m_1 \in N$, $p_i \in P_i$ and $T_2 = (n_2, m_2, f_1(q_1), f_2(q_2), ..., f_n(q_n))$, $n_2 \geq m_2$, where $n_2, m_2 \in N$, $q_i \in Q_i$, and $f_i(q_i)$ is one of the QoS operators or gains a value q_i[1].
T_1 and T_2 ***QoS-match*** iff $m_1 = m_2$ & $(P_i = Q_i$ & $p_i \in f_i(q_i))$ for $\forall i \in \{1, m_1\}$.

The tuples $X1$, $X2$ and $X3$ that were defined above also illustrate QoS-matching process. Assume there is a server-tuple $S = (3, 2, a, 555, ref)$. It QoS-matches with all tuples $X1$,$X2$ and $X3$.

Application Level

Resources that must be allocated together (such as a processor and its operational memory) form a compound component consisting of two or more subcomponents.

However, for the trading system, such a component is described by one tuple composing characteristics of both subcomponents. The complementary clients-tuple must also express the requirement for the combination of components. Therefore,

[1] QoS operators are optional, clients can express their requests by tuple elements gaining only exact values, or as a combination of operators and exact values. Therefore, is it necessary to express the option of elements gaining an 'exact value' (q_i) in the definition.

the matching process follows the definition above, as at the Trader level it is irrelevant how many services the tuple actually describes. The binder-to-binder communication, performing the establishment of a binding, ensures that all subcomponents are connected as requested.

As an illustrative example we consider a tuple describing a CPU Pentium 200MHz with 32MB RAM memory:

$CM = (6, 5, CPU, Pentium, 200, memory, 32, ref)$.

Distribution Level

Time and location-based information can be expressed in the form of ordinary tuple elements that match by the QoS-matching function (Def. 8). The monitoring components (Monitor and Updater) ensure that tuples containing this kind of element are kept updated at all times. The matching function is performed as usual.

5.2.2 QoS Negotiation

The Matching rules, including the QoS-based operations *search* and *select*, are performed in the QoS Negotiation phase. In MAGNET, the operation *search* is performed by the QoS-based matching function (Def. 8) enabling components to define service definitions in more flexible way by the QoS operators (Def. 7). The tuple model is universal and extensible by supporting user-defined types and the user-defined semantics of matching. Other models, such as typed objects, tagged trees, etc. do not provide the level of universality, we wish to support.

The exact matching results in a single tuple or a set of identical tuples matching the request. In the latter case the selection is performed non-deterministically (the first one is returned). However, a result of a user-customized matching can be a set of different tuples. Therefore, the QoS Negotiation process necessitates different semantics which enable clients to 'order' matching tuples which express their preferences. All tuple elements in an exact matching are required to match equally; however, tuple elements in a QoS-matching can deviate from an 'ideal' value. Therefore, they can be assigned 'rating' values expressing their deviation according to the requirements of a particular component.

Definition 9. Let $T = (n, m, f_1(q_1), f_2(q_2), ..., f_n(q_n))$, where $n \geq m$, $n, m \in N$ be a client-tuple. The ***tuple element rating*** is a function $\Omega(f_i(q_i)) = k_i$ for $\forall i \in \{1, m\}$ defined on the matching tuple elements, where $f_i(q_i)$ is one of the QoS operators or gain a value q_i, and $k_i \in N$ is the ***rating value*** for tuple element q_i.

Definition 10. Value $X \in N$ is the ***threshold value*** attached to a tuple $T = (n, m, f_1(q_1), f_2(q_2), ..., f_n(q_n))$, where $n \geq m$, $n, m \in N$.

By default, all tuple elements gain an equal rating value $k_i = 1$ for $\forall i \in \{1, m\}$ and $X = m$.

Table 5.1 illustrates a use of rating values. Having defined the rating value k_i to express the deviation of every tuple element from the ideal value, and the threshold value X, we have to define the combination of these partial rates to express the overall component preferences — the QoS rating matching function.

Definition 11. Let $T_1 = (n_1, m_1, p_1, p_2, ..., p_n)$, where $n_1 \geq m_1$, $n_1, m_1 \in N$, $p_i \in P_i$ be a server-tuple, and $T_2 = (n_2, m_2, f_1(q_1), f_2(q_2), ..., f_n(q_n))$, a client-tuple where $n_2 \geq m_2$, $n_2, m_2 \in N$, $q_i \in Q_i$ and $f_i(q_i)$ is one of the QoS operators, or gains a value q_i.

	matching condition	tuple elements rating	threshold value
	$m_1 = m_2 \&$ $\& P_i = Q_i \forall i \in \{1, m_1\}$	$\forall i \in \{1, m_1\}$	
match	$f_i(q_i) = q_i \Rightarrow$ $\Rightarrow p_i = q_i$	$\Omega(f_i(q_i)) = 1 \Rightarrow$ $\Rightarrow k_i = 1$	$X = m_1 \Rightarrow$ $\Rightarrow \sum_{i=1}^{m_1} 1 = m_1 \geq m_1$
QoS-match	$p_i \in f_i(q_i)$	$\Omega(f_i(q_i)) = 1 \Rightarrow$ $\Rightarrow k_i = 1$	$X = m_1 \Rightarrow$ $\Rightarrow \sum_{i=1}^{m_1} 1 = m_1 \geq m_1$
QoS-rating match	$p_i \in f_i(q_i)$	$\Omega(f_i(q_i)) = k_i$	$\sum_{i=1}^{m_1} k_i \geq X$

Table 5.1: Matching functions

Let $\Omega(f_i(q_i)) = k_i$ for $\forall i \in \{1, m_2\}$ be the tuple element rating function and $X \in N$ the threshold value attached to T_2.
Tuples T_1 and T_2 ***QoS-rating match*** iff $m_1 = m_2$ & $(P_i = Q_i$ & $p_i \in f_i(q_i))$ for $\forall i \in \{1, m_1\}$ and $\sum_{i=1}^{m_1} k_i \geq X$.

Also, if there are more tuples fulfilling the QoS-rating matching condition ($\sum_{i=1}^{m_1} k_i \geq X$), the first one non-deterministically found is presented to the client. For efficiency reasons, a best-fit strategy is inappropriate. However, clients have the flexibility in setting the threshold value X in such a way as to narrow the gap between the 'worst-accepted' tuple and the 'ideal' one. Exact matching is an extreme case accepting only ideal tuples.

In addition, the three matching functions were designed in such a way that the exact matching function (Def. 6) is a special case of QoS-match (Def. 8) which is a special case of the QoS-rating match (Def. 11). Table 5.1 illustrates the relations between all matching functions. The first column compares the condition performed on tuple elements, the second column defines the tuple element rating, and the third column compares the sum of the partial rates compared to the threshold value.

Example

To illustrate the usage of QoS Definition and QoS Negotiation defined in this section, we elaborate a simple example of a processor-printer component. There are three Pentium processors in the system of different speeds with RAM memories of different sizes:

ProcessorA 200 MHz with 32 MB RAM memory described by a server-tuple A:
$A = (6, 5, CPU, Pentium, 200, memory, 32, ref)$

ProcessorB 200 MHz with 16 MB RAM memory described by a server-tuple B:
$B = (6, 5, CPU, Pentium, 200, memory, 16, ref)$

ProcessorC 300 MHz with 4 MB RAM memory described by a server-tuple C:
$C = (6, 5, CPU, Pentium, 300, memory, 4, ref)$

Definition ranges for all three tuples are $P_1 = P_2 = P_4 = N$ and $P_3 = P_5 = S$ where N is the set of natural numbers and S is the set of all words from English alphabet (see tuple definition, Def. 1).

A client requires the fastest available Pentium processor running at least at 200MHz with at least 16MB memory (numbers are hypothetical, chosen to illustrate QoS-based matching, rather than to demonstrate realistic resources). The client request is defined by a tuple D:

QoS-rating match	$m_1 = m_2$	$P_i = Q_i \ \& p_i \in f_i(q_i)$ $\forall i \in \{1, m_1\}$	$\sum_{i=1}^{5} k_i \geq 5$	result
tuples: A D	$5 = 5$	yes	$1+1+1+1+2 =$ $= 6 \geq 5$	$match$
tuples: B D	$5 = 5$	yes	$1+1+1+1+1 =$ $= 5 \geq 5$	$match$
tuples: C D	$5 = 5$	$for\ i = 5:$ $4 \notin 16-64 \Rightarrow no$		$do\ not$ $match$

Table 5.2: Results of QoS-rating match between tuples A,B, C and D

$D = (6, 5, CPU, Pentium, 200-300, memory, 16-64, ref)$.

The rating values are:

CPU $\Omega(CPU) = 1$ (exact match)

Pentium $\Omega(Pentium) = 1$ (exact match)

200-300 $\Omega(-(200)) = 1$, $\Omega(-(300)) = 2$

memory $\Omega(memory) = 1$ (exact match)

16-64 $\Omega(-(16)) = 1$, $\Omega(-(20)) = 2$, $\Omega(-(32)) = 3$, $\Omega(-(64)) = 4$

X 5 (threshold)

From Table 5.2 it is seen that tuple D does not match because there is one element (i=5) which does not fulfill the first condition $p_i \in f_i(q_i)$. Therefore, the result of the sum of rating values is irrelevant even though it would fulfill the second condition ($1+1+2+1+0 = 5 \geq 5$). The decision between matching tuples A and B is performed non-deterministically — the first one tested is offered to the client.

5.2.3 QoS Maintenance

After the QoS Negotiation process has successfully finished (the appropriate server-tuple has been found), the binding can be established. However, QoS characteristics might change due to user physical migration, or other changes in the computing environment.

In computing systems where change is frequent, QoS Monitoring must be supported in order to keep service information in the Trader up-to-date. When a change is discovered, an appropriate action reflecting new system conditions must be undertaken. In this section we discuss MAGNET's support for QoS Monitoring and Adaptation to change.

QoS Monitoring

QoS Monitoring does not differ from information monitoring, described in section 5.1. Therefore, the monitoring components — the Monitor on the server side and the Updater on the client side — perform this task in the same way they did the non-QoS monitoring (section 5.1).

Adaptation to Change

Service changed characteristics are expressed in terms of different tuples being placed into the pool by the Monitor, the Updater or by the client and server themselves. This might result in an adaptation to new conditions. According to the

model defined in chapter 3, we distinguish between two fundamental adaptation strategies: *resource management* and *application adaptation*.

Resource management, primarily client-initiated, attempts to obtain additional or different resources to fulfill the original client's requirements. In MAGNET, this is performed by the client-instructed Updater seaching the pool for a better match, and initiating a rebinding. The details of rebinding are provided in section 5.3.6.

Unlike resource management, application adaptation is a result of a server being unable to provide agreed service. In MAGNET, the server cooperating with the third-party Administrator has to be replaced and a new tuple is offered to all attached clients. The new service can be provided by the same physical component, or by a replacement. Details of this operation are described in section 5.3.7.

5.3 Rebinding

The rebinding process, as defined in chapter 3, comprises four phases: exporting service definitions, renegotiating service definitions, destroying a binding, and reestablishing a communication channel. According to the component initiating the operation, we distinguish between *first-party rebinding* and *third-party rebinding*. In addition, a situation where an unbound component is left to find a new peer is called *first-party renegotiation*, while if it is presented with a replacement, this is called *third-party renegotiation*, or *no-renegotiation* (if it is obtained from an external administrator).

In this section we describe MAGNET's additional components involved in rebinding, then we describe the rebinding process, and finally different situations into which the system transfers.

5.3.1 Components for Rebinding

Third-party rebinding and renegotiating requires the assistance of a dedicated component, attached to the client (*Updater*) or the server (*Administrator*). Both clients and servers involved in rebinding contain 'extended' Binder subcomponents — the *Rebinders*. The *Server-Rebinder* is present in the server component, while the *Client-Rebinder* is contained in the client component. All these components are illustrated in Figure 5.2.

Rebinders

The Server-Rebinder and the Client-Rebinder constitute a pair of subcomponents contained in all clients and servers requiring rebinding. They extend the functionality of classical Binders, described in chapter 4.

In addition to the establishment of a binding, the Server-Rebinder provides an additional service interface (`rebindS`) through which the Administrator performs the rebinding operation. Symmetrically, in addition to Client-Binder functionality, the Client-Rebinder provides an service interface `rebindC` through which the client is informed about rebinding. In addition, the Server-Rebinder keeps a list of clients currently attached to the server in form of their `rebindC` references. Via a dynamically established binding between service interfaces `rebindC` (on clients side) and `forwardS` (on the server side), clients are informed about charges in currently provided service, such as a server upgrade.

As all rebinding operations are performed in cooperation with Rebinders, they ensure that the rebinding takes place when components are ready to do so.

The Administrator

The key operation of the Administrator component is a server upgrade. The operation (performed by a human administrator or automated) consists of switching to the new server and upgrading server 'structures', if required. The procedure of upgrading server structures falls into one of the following three categories:

1. Resources offering a time-constrained service. Interacting clients are allowed to finish on the current server to avoid the upgrading procedure, while new clients are assigned to the new server. For example, in a printer upgrade, current jobs in a printer queue are allowed to finish, while new jobs are allocated the new printer.

2. Resources necessitating runtime rebinding but no state has to be maintained and therefore no additional data structures need to be upgraded; for example, switching between different levels of network connectivity does not require any additional operations (such as copying), in contrast to a disk upgrade, see point 3. In this case, the Administrator's task is to initiate the operation and provide a reference to the server replacement.

3. Resources necessitating an upgrade of server data structures, for example a disk upgrade. The Administrator, in cooperation with both servers (current one and the replacement), provides the upgrade and ensures that consistency is maintained.

The Updater

Another component involved in the rebinding process, instructed by the client, is the Updater. It keeps monitoring server-tuples in the information pool and, if appropriate, initiates rebinding. Its monitoring procedure was described in section 5.1.

5.3.2 The Rebinding Process

In chapter 3, we identified four phases of the rebinding process: exporting service definitions, renegotiating setvice definitions, destroying a binding, and reestablishing a communication channel. In this section, we describe how these phases are implemented in MAGNET.

Exporting Service Definitions

Every new component arriving into the system must export its service into the Trader, as usual. However, in case of rebinding three cases need to be described.

Firsly, two tuples need to be inserted into the Trader structures by ADVERT — the classical service tuple (contains service interface `bglue`) and a 'rebinding' tuple (with a service interace `rebindS`), see Figure 5.2. All servers in the system participating in the rebinding process are responsible for inserting these two tuples into the Trader.

Secondly, the Administrator, as a third-party, performs a replacement of a server. From an insertion of a new server's tuple by ADVERT, the new server is available to clients. However, the Administrator must perform the upgrade operation: it obtains the `rebindS` reference of the current server from the Trader by BIND (a binding is established between the service interface `destroy` and `rebindS`), and hands over the reference (`bglue`) to the 'new' server and provides required upgrades server data structures.

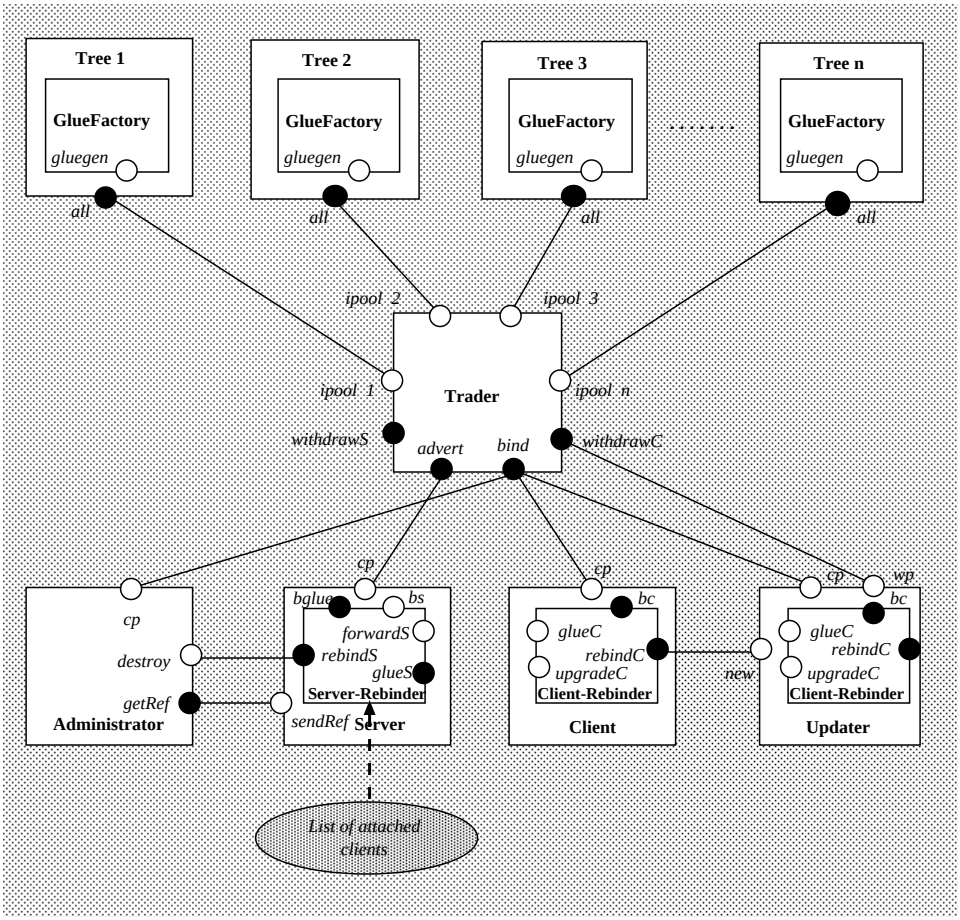

Figure 5.2: MAGNET with Components for Rebinding

Thirdly, Updater calls the operation BIND with a higher requirements (closer to the ideal resource) than the server currently serving the client and waits until a matching server-tuple arrives.

Renegotiating Service Definitions

Renegotiation service definitions might be performed before or after the current binding is destroyed. Semantically, this does not differ from the negotiation in the binding process. However, according to which component performs it, there are three cases:

Unbound clients (*first-party rebinding*) renegotiate a new server-tuple by calling BIND as a result of being unbound, yet not given a new peer. This will result in the client finding different servers, as client's requirements might have changed due to QoS requirements.

Updater, a *third-party*, renegotiates a new server, then it performs an additional check (rebinding of a client close to termination might not be worth it, because of an overhead of the rebinding operation). However, if it decides that rebinding is beneficial, it notifies the client.

No-renegotiation needs to take place if all clients are presented with a new server replacement obtained from an external administrator.

Destroying a Binding

Firstly, according to our assumptions, it is the responsibility of every component to keep its tuples in the information pool up-to-date, therefore, all tuples must be removed by calling the operation WITHDRAWS (servers only; clients do not keep tuples in the pool) before destroying themselves.

The server informs all clients about its shutdown, which results in destroying the communication channel (by establishing a binding between service interfaces `forwardS` on the server side and `rebindC` on the client side). Clients use a built-in function in the communication channel, to inform the server of its departure, so that the Server-Rebinder can keep the list of clients up-to-date.

Reestablishing a Communication Channel

Firstly, the Server-Rebinder needs to receive the new client tuple (from a client, or from the Updater via `upgradeC` and `bglue`), initiate the Rebinder-to-Rebinder communication protocol (between `bc` and `bs`) which performs the final stage of the rebinding process, to establish the new end-to-end communication channel. Again, this phase does not differ from the classical establishment of a communication channel. It performs the admission protocol, if required.

Transparency of the rebinding is achieved — it is impossible for the server to distinguish between classical cases (tuples passed over from the Trader), and rebinding components (tuples passed over via a third-party). Finally, a new client-server binding is established.

5.3.3 Rebinding Situations

Here we will investigate the four rebinding situations described in chapter 3:

- first-party renegotiated first-party rebinding
- first-party renegotiated third-party rebinding
- third-party renegotiated third-party rebinding

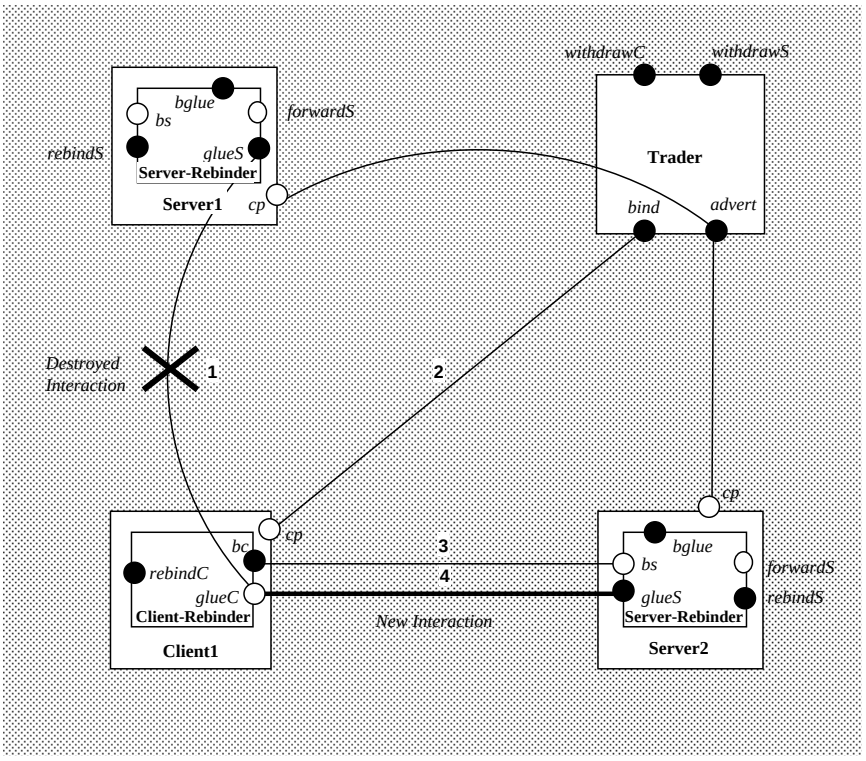

Figure 5.3: First-Party Renegotiated First-Party Rebinding in MAGNET

- no-renegotiation third-party rebinding.

All situations are illustrated with simple examples. However, for reasons of simplicity we represent rebinding of only one component (differences in number of component rebound we described in chapter 3). Further, all figures are equipped with numbers representing the order in which each binding is established. Therefore, we will not describe in words each rebinding phase for each situation. Also, for reasons of clarity, we decided to omit the Tree components and present only interactions with the Trader. Nevertheless, the matching process *does* take place in Tree components, as described in chapter 4.

5.3.4 First-Party Renegotiated First-Party Rebinding

In this case, it is the client component which requires a new service. Therefore, it destroys the current binding with the server and renegotiates a new server in the Trader. Figure 5.3 illustrates this rebinding situation with a simple example.

5.3.5 First-Party Renegotiated Third-Party Rebinding

In this case, a server (as a passive component) is disconnected by a third-party, the Administer. It announces shutdown to all attached clients kept in the list in the Rebinder. Clients are left to renegotiate a new peer themselves. Figure 5.4 illustrates this rebinding case.

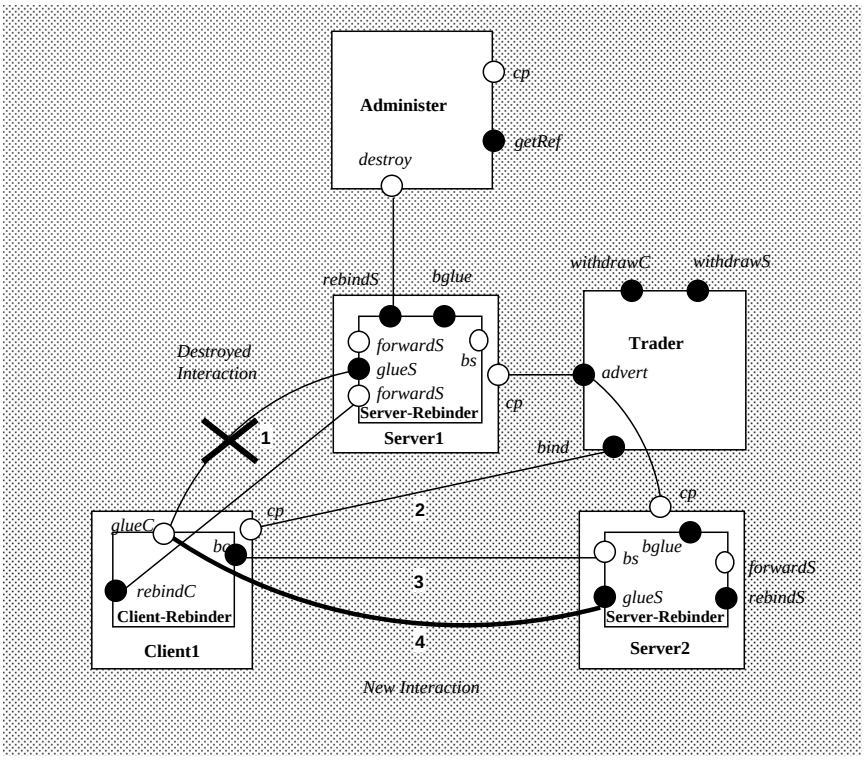

Figure 5.4: First-Party Renegotiated Third-Party Rebinding in MAGNET

5.3.6 Third-Party Renegotiated Third-Party Rebinding

In this case, the Updater performs a rebinding of a client to a server that meets the client's requirements more accurately (informed via a binding between `new` and `rebindC`). This feature is highly desirable in environments with systems whose characteristics frequently change. In this situation, phases 'destroying a binding' and 'renegotiating service definitions' are performed in the opposite order, as it better suits the character of the described situation.

Figure 5.5 illustrates a simple example.

5.3.7 No-Renegotiation Third-Party Rebinding

In the final situation, a third-party, Administrator, performs a server replacement. Clients are informed about their new peer in the form of the `upgradeC` interface service. No renegotiation is required, as the third-party has external knowledge about system charge (such as a system administrator). Figure 5.6 illustrates Server1 being upgraded to Server2.

5.3.8 Other Issues

In this section we briefly describe three important issues related to rebinding — consistency, protection and buffering.

Consistency

As MAGNET treats components as black boxes it cannot be responsible for maintaining consistency, as we assume throughout this thesis. This concerns both their

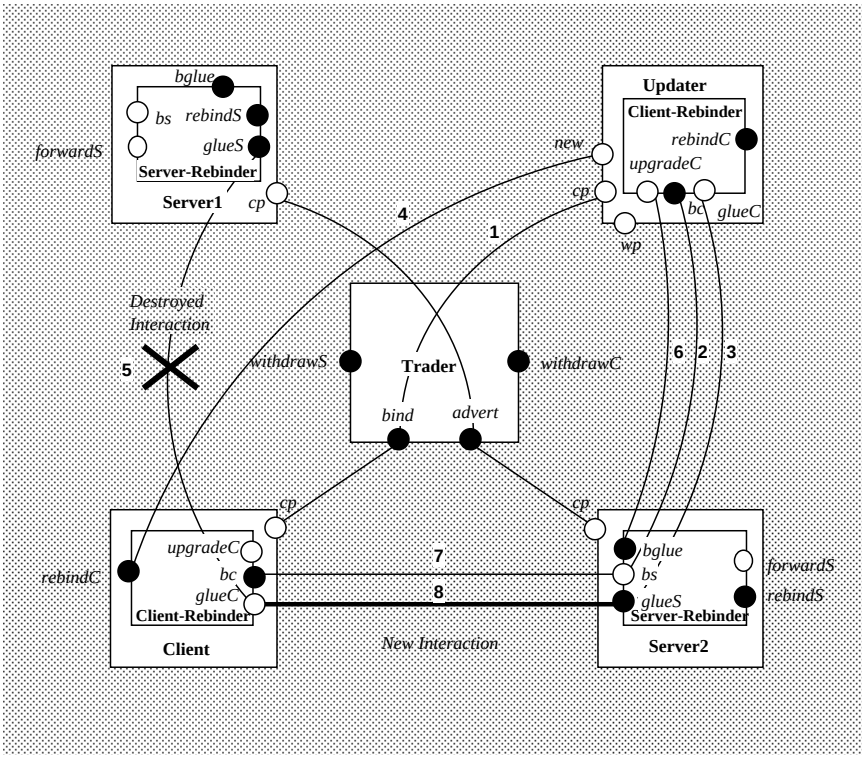

Figure 5.5: Third-Party Renegotiated Third-Party Rebinding in MAGNET

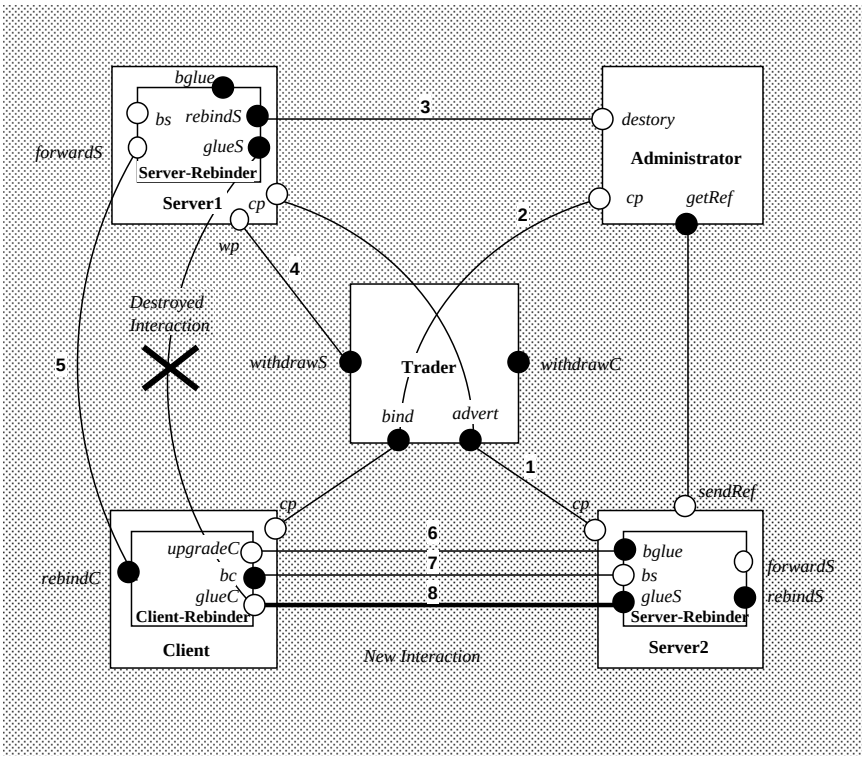

Figure 5.6: No-Renegotiation Third-Party Rebinding

bindings and updating the tuples kept in the information pool.

In addition, client components are expected to inform servers about the termination of the binding (by the appropriated operation built-in the communication channel) in order to allow servers to keep the list of current clients updated for rebinding purposes.

Protection

According to our assumptions, the framework cannot ensure the validity of information placed into the pool. However, at the component level, no binding can be established without a Rebinder-to-Rebinder protocol performing the required protection checks. Therefore, the protection of rebinding operations is ensured in the same way as the protection of the binding process, discussed in chapter 4.

The only exception can be no-renegotiation third-party rebinding. Specific applications can decide to skip the admission protocol, as the client has already been checked by the original server. If this option is supported, the new server's Server-Rebinder must provide an additional service interface for upgrading clients, with simplified semantics concerning the establishment of a communication channel. Although this operation adds complexity to the server and disables the transparency of the reestablishment phase, it might be desirable for particular applications as it improves the efficiency of the rebinding process.

Buffering

Client-server binding having a character of a 'batch processing' (as opposed to interactive processing) can be performed even when the requested resource is not available at the moment of the request. Instead of leaving the client waiting until the server is available, its requests are 'buffered' in a *buffer-server* (special Updater component) which allows the client to continue its operation. The buffer-servers, substituting the real servers while they are unavailable, ensure that the operation is performed when the requested server is connected. This can be performed without interference from the originating client, even after its completion.

A typical example of such an application is a mobile user 'printing' from a portable while on move. The jobs are buffered allowing the application which originated the operation to finish. All buffered tasks are printed out transparently when the portable is connected to a network with a printer. When the real resource becomes available (e.g., the portable is connected to the network), the buffer-server establishes an binding with the real server and the service can be performed.

5.4 Scalability

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. Therefore MAGNET, designed to support the topology of existing computing systems, consists of connected domain-like units — *federations.* In this section, we define the federations, describe their *dynamic reconfiguration* enabling computers to join and leave the system at runtime, and discuss issues concerning the *scalability* of the architecture.

5.4.1 Federations

MAGNET's primarily goal — dynamic resource management — is targeted to be small (for example an application requesting to print is usually not interested in a printer on another continent). Consequently, MAGNET's 'local-scale domains' —

federations (illustrated in Figure 3.2) — are the basic entity of the architecture, because it reflects the topology of typical resource allocation problem (e.g., most of resources are allocated in a local Internet domain). However, the architecture must also support scaling in order to enable a larger physical area to be covered. Achieving this by extending a single federation would result in maintaining and seaching a world-scale global information pool. This would be not only very inefficient, but also, for reasons of locality of resource management, also pointless.

Therefore, the framework supports the interconnection of Traders enabling inter-federation communication within the computing environment. All system features discussed so far assumed the existence of a single federation. As MAGNET's typical computing environment will use the Internet, the federation size is derived from the size of Internet subdomains. The algorithm for the configuration of a federation (distribution of Tree components over available processors) is discussed in chapter 6.

5.4.2 Dynamically Reconfigurable Domains

Before we describe scaling of the architecture (in the next section), we discuss the situation of mobile users temporarily joining 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 use the 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 provisions 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.

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'. 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), not using MAGNET.

Operation JOIN

A portable client requesting a service which might be fulfilled by office-based site servers when the portable is temporarily connected, inserts *three* bind tuples into the portable pool — the classical bind tuple $C1$ defining the request, the second tuple of the form: $C2 = (n, 1, join, C1)$ encapsulating the actual tuple $C1$ tuple, and the third one $C3 = (n, 1, leave, C1)$ which will be used for disconnection.

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, a matching between $T1$ and $C2$ can be achieved.

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 Glue-Factory 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 $C1$ 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 5.7 illustrates operation JOIN, for reasons of simplicity, we omit the Tree components. However, the binding is established in cooperation with Trees, as defined in chapter 4.

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: $T2 = (2, 1, leave, bglue2)$.

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

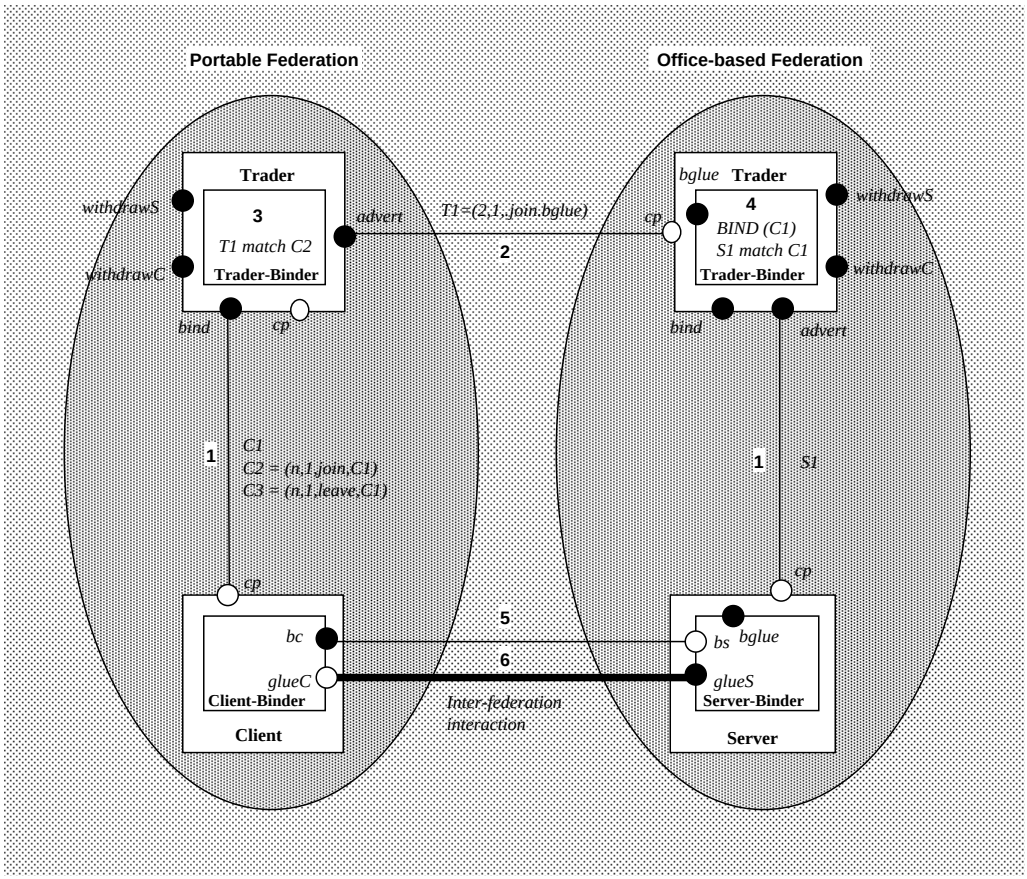

Figure 5.7: Operation JOIN

$C3$, and operation WITHDRAWC on $C1$ is performed. If it does not succeed — it means the client is already being served by an office-based server. However, all clients must be informed 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 5.8 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 $X3$.

Discussion

In order to enable portable servers to offer their provisions to other federations, the Trader-Binder must be equipped with four service interfaces: `bglue` (performing BIND), `bglue2` (performing WITHDRAWC), `bglue3` (performing ADVERT), and `bglue4` (performing WITHDRAWS). Also, two 'join tuples' would have to be inserted into the portable pool:

$T1 = (2, 1, join, bglue)$

$T3 = (2, 1, joinS, bglue3)$,

and two more tuples to disconnect the Traders:

$T2 = (2, 1, leave, bglue2)$

$T4 = (2, 1, leaveS, bglue4)$,

and all server would call operation BIND. However for reasons listed above, it is not supported by the architecture primarily, but can be added to the framework, as outlined in this section.

5.4.3 Scaling the Architecture

In order to enable the architecture to scale, the Traders must be 'identifiable'. Therefore, a method of unambiguous Trader addressing must be enforced to operate at a world level. However, any application using MAGNET will be running in a particular computing environment which, if scalable, must have some unambiguous CPU naming scheme incorporated in order to identify its processors. Running on top of an existing computing environment, MAGNET's Traders can use the existing naming scheme for their identification. Therefore, we can claim that the actual design and assignment of addresses is irrelevant to the inter-Trader communication.

However, in order to design Trader-to-Trader binding, there are two features we must assume about the CPU naming scheme that will be used for the Trader identification:

- names are unambiguous in the bounds of the computing environment
- names are constructed by a way that they can form a hierarchical tree structure with a single root, and a unambiguous path in the tree (at the naming

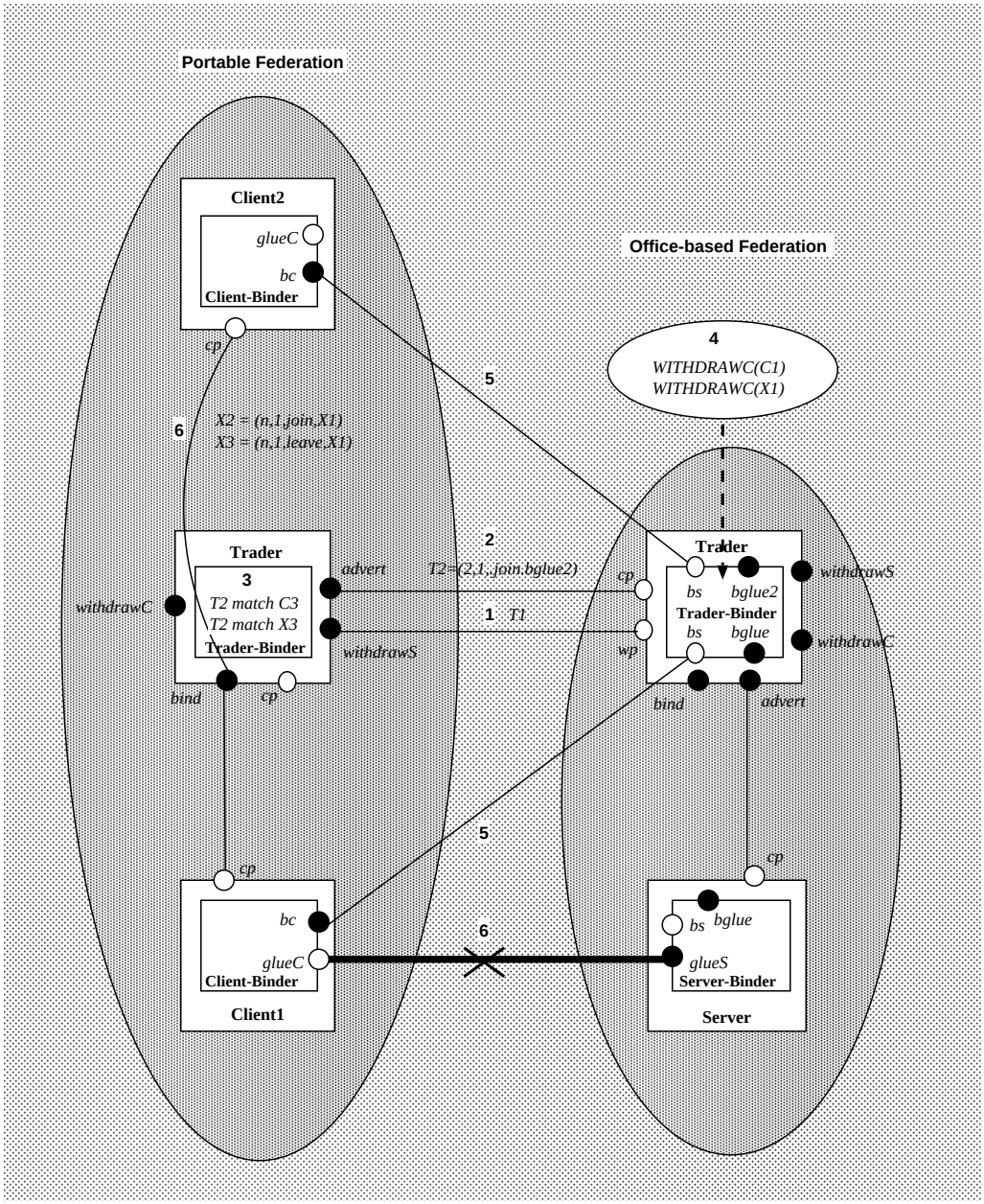

Figure 5.8: Operation LEAVE

level) between two Traders can be determined (however, nothing about the network topology at the implementation level is assumed.)

There are many examples of existing communication infrastructures based on naming fulfilling these requirements, such as the Internet based on IP addresses (each byte represents a tree naming layer, nothing about physical topology is assumed), the world telephone network (country code, city code, etc. represent tree layers), etc. It is irrelevant whether the MAGNET trading system is constructed from the 'top' (the 'root' Traders is created first, other Traders can be immediately connected to the tree, therefore the trading system is never disjoint), or whether it evolves from the 'bottom' by joining local Traders together (in this case, the trading system is disjoint, and connects on demand).

Having discussed our presumption on the Trader naming system, we will define Trader-to-Trader communication enabling bindings between components in different federations.

The Support for Scalability

For operations JOIN and LEAVE the actual identity of the local site Trader was irrelevant. However, for components requiring a server from a *particular federation*, support for addressing and locating is essential. We assume that Traders reside on unambiguous addresses forming a tree hierarchy which is derived from the naming scheme used in the computing environment where MAGNET operates. The Traders are equipped with dedicated location components, *Locators* which are attached to all Traders and form a tree hierarchy according to Traders' addresses. Figure 5.9 illustrates a fraction of a global trading system consisting of federations using IP addresses as their naming scheme. Locators reside at addresses forming the tree hierarchy which subtracted from IP addresses as illustrated in Figure 5.9.

Communication between Federations

Clients which require a tuple $C1$ to be inserted into an information pool on a particular address have to incorporate it into a new tuple of the format $C = (n, 1, up, address, operation, C1)$. These tuple elements have following meaning: 'up' is a key word — the only matching tuple element, 'address' is the address of the Trader where tuple $C1$ is to be served, and 'operation' is one of the four Trader operations. Then, tuple C is inserted into the pool by operation BIND. It matches with special server tuple $T1 = (n, 1, up, locglue)$ (described below) inserted into the pool by the Locators. As usual, the client matching tuple is passed over to the server. However in this case, instead of the Server-Binder, it is processed by the Locater. Their algorithm, described bellow, ensures that the tuple subsequently reaches the Trader on the 'address'. Here, the tuple $C1$ is extracted and inserted into the pool by the 'operation', which is also extracted from the tuple $C1$. If there is a matching tuple available, an end-to-end binding between federations can been established.

The remaining problem is to locate the Trader on a particular address — the task of Locators.

Locators

The Locators are attached to all Traders and perform two following actions: form the tree by establishing bindings between appropriate Traders, and passing over tuples addressed for other federations.

Firstly, by inserting a server-tuple $T1 = (n, 1, up, locglue)$ to every information pool in the tree layer below, binding with all Traders on that level is established.

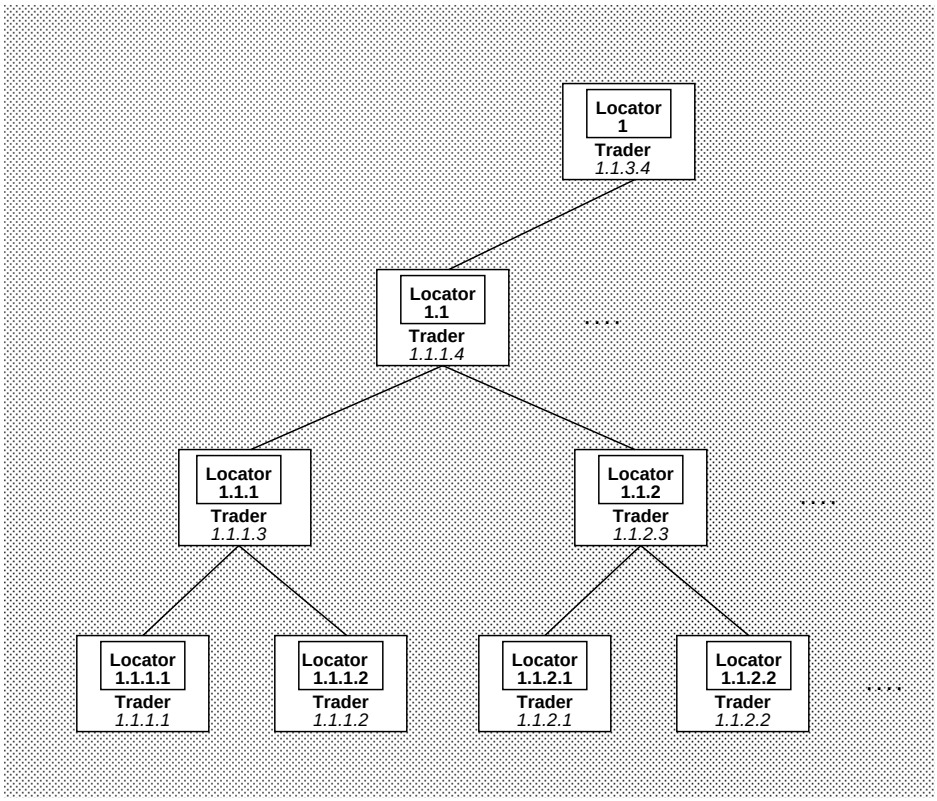

Figure 5.9: Trading scheme based on IP addresses

Consequently, a tuple of this format is inserted into every information pool by the Locator on the layer above. The Locator in tree leaves (defined by the tree hierarchical structure) Traders skip this step.

In addition, every Locator (including those in leaves) insert a server-tuple $T2 = (n, 1, down, locglue)$ to the information pool of their Trader. The service interface `locglue` represents the service of the Locater which inserted the tuple. This task is performed by the MAGNET administrator configuring the system.

Secondly, when a client tuple $C = (n, 1, up, address, operation, C1)$ is inserted and matches with $T1$, it is sent to the Locator on the layer above for further processing. The Locator algorithm is recursive and is repeated by all Locators handing over the tuple. One step leads into three cases investigated in this order:

1. the 'address' in the obtained tuple is the one of the attached Trader — the tuple reached its final destination. The actual tuple $C1$ is extracted from the tuple C, and inserted into the local information pool by extracted 'operation' (the third tuple element).

2. the component with the 'address' belongs to the subtree connected to this Locator. Then, the tuple can be forwarded 'down' towards its destination. Firstly, matching field 'up' is replaced with 'down' :
   $C = (n, 1, down, address, operation, C1)$, secondly, operation BIND on the new tuple C is called. In the Trader below it is inserted into the pool, matches with tuple $T2$, is received by the Locator, and the algorithm performs another recursion.

3. the tuple belongs to another subtree; it is inserted unchanged into the local information pool by the operation BIND where it matches the tuple $T1 = (n, 1, up, locglue)$ from the Locator above, is forwarded to to the Locator above, and the algorithm performs another recursion.

Figure 5.10 illustrates an establishment of binding between remote federations using Locators.

5.5 Chapter Summary

In this chapter we have described MAGNET's advanced featres — information monitoring, quality of service management, rebinding, and scalability. This chapter together with chapter 4 covers the design of all features provided within the MAGNET framework.

Firstly, we have discussed information monitoring as an essential system feature allowing data in the shared information pool to be kept up-to-date. There are two dedicated components for monitoring — the *Monitor* responsible for monitoring server provisions, and the *Updater* informing clients about changes in the computing environment.

Quality of service management is another indispensable feature of computing environments supporting users working with resources with frequently changing characteristics. Following our model of QoS Management defined in chapter 3, there are three tasks to be performed: QoS Definition, QoS Negotiation, and QoS Maintenance.

QoS Definition is considered at four levels. Firstly, at the characteristics level, components define values of their service characteristics, equipped with *QoS operators* (such as —,¬,|,* specifying operations *interval*, *negation*, *or*, and *all*, in this order, in addition user-defined operators can be added). Secondly, at the service level, *QoS-match* is defined to enable matching of tuples extended of the QoS operators. Thirdly, tuples might represent combinations of components to prevent

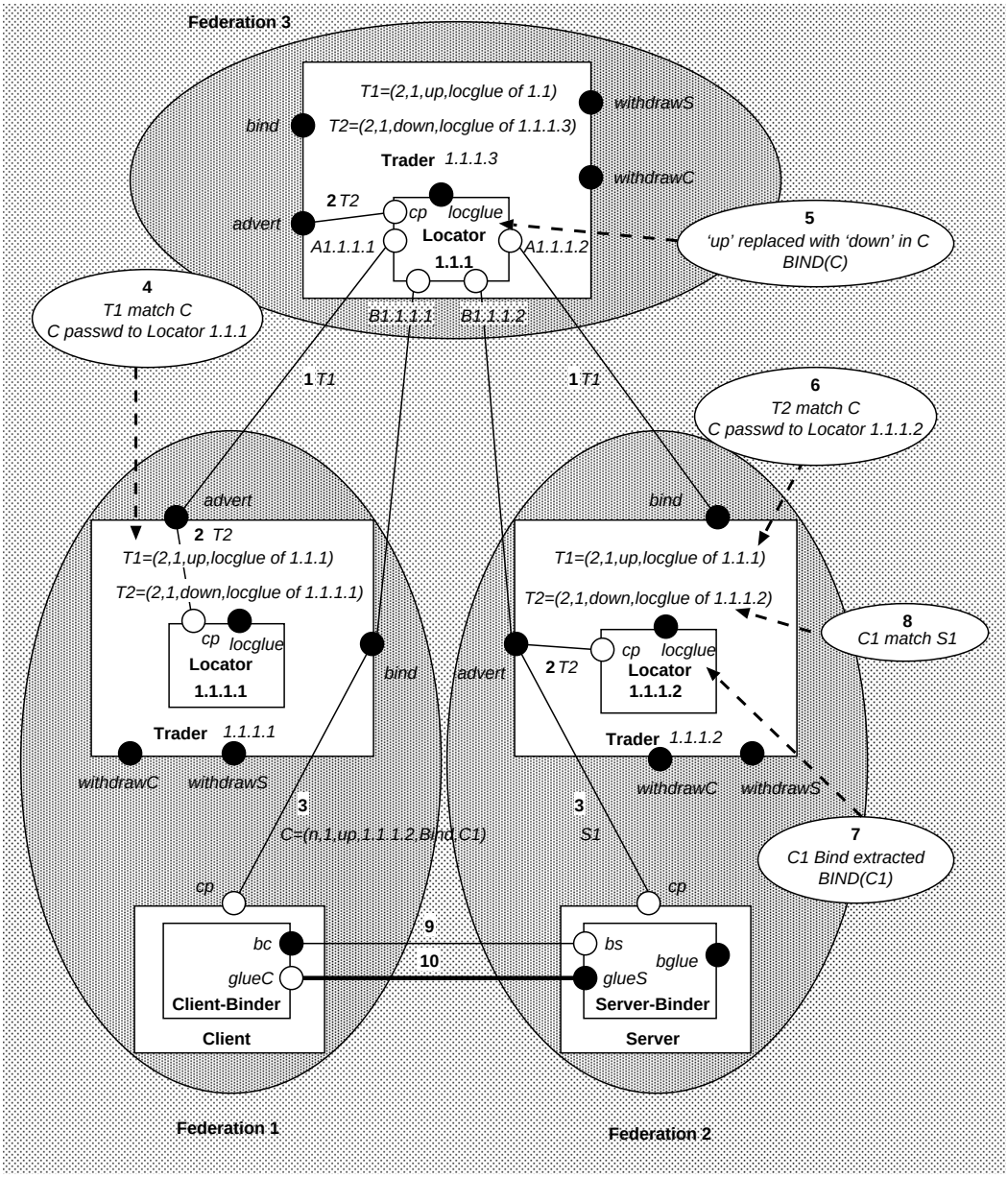

Figure 5.10: Communication between Federations

non-nondetachable resources from being allocated separately (e.g., a processor and attached devices). And finally, tuples might contain time and location dependent information to take into account a user's physical location.

QoS Negotiation introduces the *QoS-rating operators*, and *QoS rating match* operation allowing clients to define the requirements with further flexibility based on the evaluation of each tuple element, and to select those above a threshold limit.

QoS Maintenance comprises QoS Monitoring of QoS-based resource descriptions, and adaptation to change covering two strategies: resource management and application adaptation.

Rebinding is an essential feature of any dynamic system enabling operations like run-time server upgrades, or client adaptation to changed system configuration. The rebinding process comprises at four phases: exporting service definitions, renegotiating service definitions, destroying a binding and reestablishing a communication channel. MAGNET's component enabling the rebindsing operations are called the *Rebinders*, the *Administrator* (attached to a server) and the *Updater* (attached to a client). According to the component initiating the operating, we distinguish between *first-party rebinding*, and *third-party rebinding*. In addition, components might be left unbound to renegotiate a new peer themselves (*first-party renegotiation*), or are presented with a replacement (*third-party renegotiation*, or *no-renegotiation*). Four rebinding situations outlined in chapter 3 have been discussed.

Scalability is another indispensable requirement of mobile users. We have defined 'local' units comprising one Trader component — *federations*. The framework provides *dynamic reconfiguration* of federations by operations JOIN and LEAVE in order to support mobile users requiring to join and leave a local site which dynamically. As Traders follow the naming scheme of the computing environment where MAGNET operates, scaling the architecture is feasible. Special dedicated components, *Locators*, attached to all Traders, enable a tuple to be passed over to a particular Trader on a component request. If the computing environment enables a long-distance communication channels to be established, distributed MAGNET may assits in establishing bindings in world-scale physical distances.

Last two chapter have covered MAGNET's design. The next chapter focuses in its implementation issues.

Chapter 6

Implementation Experience

In this chapter we discuss implementation issues of the MAGNET architecture. *Regis*, an environment for constructing distributed systems [42], was chosen as a base for the implementation of a prototype of MAGNET because it offers an infrastructure well-suited to MAGNET's purposes.

In section 6.1, we start by a brief overview of Regis' key features in order to introduce the computing environment. Then we present how Regis was adapted for MAGNET. In section 6.2, we discuss Regis-dependent implementation features — the components of the framework and the support for the binding process.

The rest of the chapter covers the following issues: tuple implementation and the matching process (section 6.3), the data structure of the Tree components (section 6.4), the Trader and the distribution of Trees (section 6.5), QoS Management issues (section 6.6), discussion on limitations and usability of the architecture, sections 6.7 and 6.8, respectably.

6.1 Regis Distributed Environment

MAGNET, as a resource trading architecture, relies on the computing environment where it operates. We have chosen Regis, an environment for constructing distributed systems, for building MAGNET's prototype. Although other platforms could have been used (we discuss this issue in section 6.8), Regis provides a very suitable framework matching our notion of components and services. Defining resources as Darwin components in terms of services they offer or provide (from which Regis code can be generated), simplifies the implementation effort. In addition, Regis, enabling communication between local or remote components which need not be aware of their location, also supports run-time binding establishment necessary for MAGNET architecture.

In this section, we give an overview of Regis, and discuss how it was adapted to better suit MAGNET's purposes.

6.1.1 Overview of Regis

Regis [42], developed at the Department of Computing at Imperial College in London, is a computing environment for building complex distributed systems. It is based on a model defining programs in terms of *components* (acting as black-boxes), interconnected by typed *interaction styles* which represent primarily one-way communication between server and client.

The components are defined by service interfaces, either provision or requirement, which is described by an *interface reference*, and by its type — an interaction

style. Each component can be either primitive or composed of other components.

Communication is achieved by binding components together either *statically* using Darwin, a structure configuration language [43], or *dynamically* at runtime. As open distributed systems enable components to join and leave the system at runtime, support for dynamic binding is essential. Regis supports this feature by enabling an interface reference to be passed from server to client over an existing binding, and by providing a *binding operation* (`bind`) which, called by the client, establishes a new communication channel.

Regis also enables physical distribution of its components over existing processors, a set of typed interaction styles from which complex bindings can be built. In addition, Regis supports four types of binders providing both fundamental binding operations: *first-party binding*, and *third-party binding*, and derivative operations — *import* and *export* binding — used for dynamic offers and requests for service interfaces.

In addition, Regis' interface reference naming is unambiguous and ensures scaling of the architecture [15]. Communication channels may be composed from a stack of protocols which can be loaded on-demand in order to enable adaptation to new system conditions. Consequently, the framework enables run-time program management featuring initial configuration, programmed evolution, and runtime reconfiguration. Additional features and details of the architecture can be found in [15]. For our prototype we used Regis version 0.5.8 ported to RedHat Linux 5.0. Both Regis and MAGNET are primarily implemented in the C++ programming language.

6.1.2 Adaptation of Regis

In order to use Regis to build a prototype of MAGNET, the system needed to be extended to enable the interaction required by the resource management architecture.

Firstly, as the communication between the Trader, Trees and components is based on exchanging information in the form of tuples, support for an interaction style type tuple was necessary (enabling, for example types `Port<Tuple>` sending a tuple over a basic unsynchronized interaction style `Port`, etc.) A tuple is a C++ data structure, its implementation is discussed in section 6.3.

Secondly, a new interaction style `Glue<operation, type>` was added to the set of Regis original interaction classes to enable pairs of operands to be sent over a single communication channel. The semantics of the `Glue` interaction style is derived from `Port`, and allows both operands to be sent separately or together (such as, in the `Trader.ipool` — `Tree.all` interaction, discussed in chapter 4). This characteristic further extends the flexibility of interaction typing.

6.2 MAGNET Implementation in Regis

As an in-depth description of every aspect of the implementation is not essential, we have focussed on the parts of the implementation that are significant to the framework's overall functionality. In this section we describe the implementation of MAGNET's components and interactions used by the binding process.

6.2.1 System Components

All MAGNET's components, described in chapters 4 and 5, are implemented as Darwin components. As their static services are described using Darwin, the bindings can be established in advance.

The Darwin textual notion represents the bindings in the same way as the graphical notion used throughout the thesis. Components (represented by rectangles in the graphical notion) are defined by a keyword `component` followed by the name of the component. Service provisions (represented by a black circle), is defined by a keyword `provide`. Service requirement (represented by an empty circle) is defined by a keyword `require`. The second argument stands for the interaction style (which might be predefined using `typedef`), and the third argument represents the service interface.

As an illustrating example, we define the Trader, server and client components

The Trader Component

```
typedef Glue<Oper,Tuple> glueIP;
typedef Port<Tuple> portT;

component trader {
   provide portT _bind;
   provide portT _advert;
   provide portT _withdrawS;
   provide portT _withdrawC;

   require glueIP ipool[TupleMatchSize * TElmTypeNo];
}
```

Each of the four Trader functions is declared as provisions (`_bind`, `_advert`, `_withdrawS`, `_withdrawC`). The array `ipool` represents requirements of service from distributed Tree components. This feature (including the meaning of constants defining the array boundary `TupleMatchSize` and `TElmTypeNo`) is described in section 6.5.

A Server Component

```
typedef Port<Tuple> portT;

component server {
   require portT cp;
}
```

A Client Component

```
typedef Port<Tuple> portT;

component client {
   require portT cp;
}
```

Both components define a requirement `cp`, in accordance with the graphical representation of clients and servers used throughout the thesis. The framework can be initialized by a main component declaring these three components and establishing the first static bindings between server, client and the Trader. In Darwin, components are declared by a keyword `inst`, followed by a component name and an instance name, while a static binding is defined by a keyword `bind` and two dashes connecting appropriate service interfaces.

The Main Component

```
component main {
   inst trader t;
   inst server s;
   inst client c;
   bind c.cp -- t._bind;
   bind s.cp -- t._advert;
}
```

The main component definition is only illustrative, to give the reader not familiar with Darwin an idea about the language. In our prototype, in order to enable new components to join the system at runtime, component-Trader bindings are established using the Regis nameserver — service provisions are exported into the nameserver, while server requirements are imported, and the binding is established at runtime. However, we do not provide a full discussion of the syntax of these operations for reasons of readability.

A Regis code is generated from the Darwin definition, and the remaining task is to define the actual component functionality in C++ as a `body()` function. Any further communication can be performed by statically established bindings (in Darwin), or by bindings established at runtime. In order to to so, dynamically created services must be defined in the components' `body()` functions, and their interface references passed over an existing binding at runtime.

Implementation of the MAGNET Architecture

For reasons of simplicity, we focus on the core elements of the architecture — those involved in the binding process. As all other system components (the Monitor, the Updater, etc) are implemented in a similar way, we would not gain any benefits from presenting complete code listings and binding descriptions.

In Figure 6.1, we illustrate four Darwin components: the Trader, Trees, client and server and their bindings. The subcomponents, the GlueFactory and the Binders, are implemented as C++ objects declared within a particular Darwin component. Again, bindings are numbered to illustrate the order in which they take place. In addition to the interaction styles, the Figure also shows how public service interfaces (`bglue` and `bc`) are advertised. As can be seen in the Figure 6.1, interaction styles used for the following bindings are:

- the Trader — component: interaction style `Port<Tuple>`
- the Trader — Tree: interaction style `Glue<Oper,Tuple>`
- GlueFactory — Server-Binder: interaction style `Port<Tuple>`
- Server-Binder — Client-Binder: interaction style `Port<G::Reference>` where `G` is the final client-server communication protocol interaction style
- Server-Binder — Client-Binder using the Admission protocol: interaction style `Glue<Entry<S,boolean>::Reference,G::Reference>` where `S` is the security protocol (UID, PGP, etc.) and `G` is the final client-server communication protocol interaction style
- Client — Server: application-defined interaction style `G`

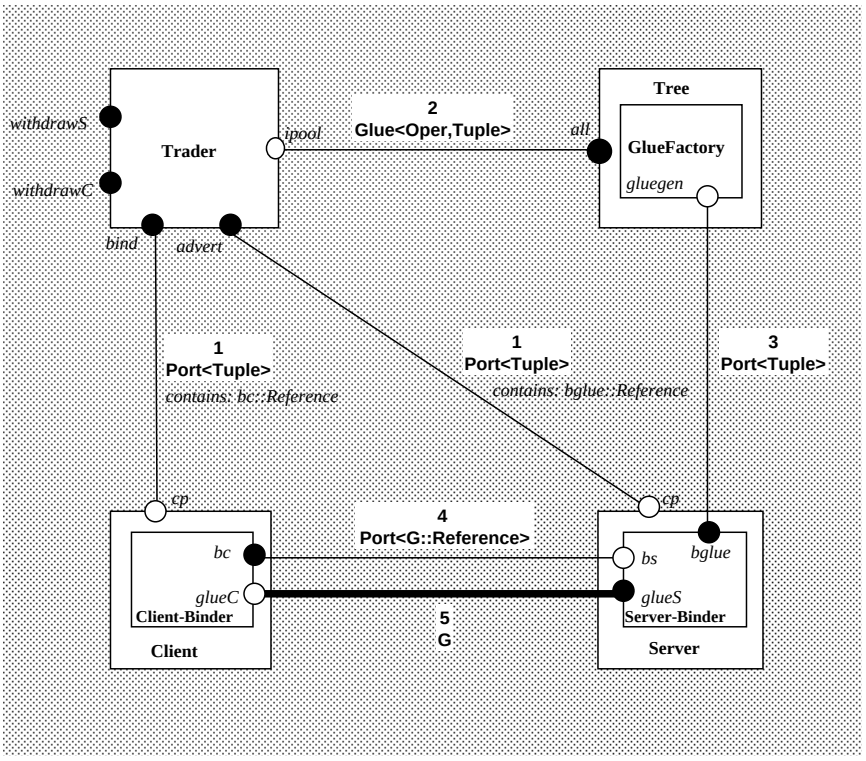

Figure 6.1: Regis bindings used in the MAGNET architecture

6.3 Tuples

A required feature of the architecture is to provide extensibility of existing services and data formats. Therefore, the implementation of the tuple format and the matching process must feature flexibility enabling this requirement to be fulfilled. Firstly, the tuple format must allow new data types to be added dynamically to represent additional resource features and QoS description. Secondly, the matching process must support user-customization enabling these new data types to be incorporated into the matching process.

Tuple Format

The tuple, at MAGNET's level (Def. 1), consists of tuple-elements which encapsulate the real data types used by components to express their features. In C++, this is implemented as a high-level base-class (`Tuple`) comprising the tuple size, the tuple matching size, and encapsulating tuple-elements. All standard and user-defined tuple-element classes are inherited from a base tuple-element class `TElm`. Figure 6.2 represents the inner structure of the tuple class.

The Matching Process

The matching function is implemented as an overloaded member function of tuple-element classes inherited from the base class `TElm`. A tuple-element type matches only the same type, and the 'equality' of values can be re-defined according to the type. The implementation of the QoS-based matching process is discussed in section 6.6.

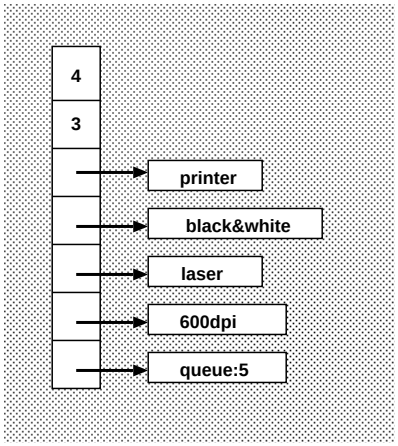

Figure 6.2: Tuple representation

6.4 The Tree

Once a tuple is received by the Trader, it is forwarded to a particular Tree component for processing. In this section we focus on the data structure of the Tree components, the implementation of the Trader operations performed by the Trees (ADVERT, BIND, WITHDRAWC, WITHDRAWS), and we will compute the complexity of these operations.

6.4.1 The Tree Data Structure

In order to design a suitable data structure for representation of tuples in the information pool, it is necessary to consider the character of operations it will be particularly used for. Fundamentally, data structures are considered *static* or *dynamic*.

Static data structures can be efficiently queried, but operations insert and delete are expensive. However, once they are built, the structure is not expected to change much, therefore, update performance is not a problem. For example, heap-type structures, such as binomial heaps [77], are designed to support the search of the minimum value (which can be performed in constant time), however, operations insert and delete are expensive. In contrast, *dynamic data structures* [26] support frequent changes (operations insert and delete) as well as 'querying operations'. In this case, the purpose of the structure and the probability of each operation determine whether insert and delete operations should be favored to search operations or vice versa.

As components might insert and withdraw a tuple at runtime, the data structure of the Tree component must be *dynamic* — it supports both insert and delete operations, and the search operation — the matching function. As every insert (ADVERT, BIND) and delete (WITHDRAWC, WITHDRAWS, or as a result of ADVERT when client-tuples are satisfied) operations follow a sequence of matching operations, the structure is designed to favour the matching to inserts and deletes. Here we describe the structure; its complexity is calculated in section 6.4.3.

The Design of the Tree Data Structure

In order to support the matching operations, the data structure of the information pool is derived from a tree[1] data structure. Edges of the tree are implemented as simple pointers, while vertexes are more complex and do not share the same structure. We distinguish between three types of vertexes in the tree: the *root*, the *inner-vertexes* and the *leaves*.

As one of the tuple-matching requirements is the equality of tuple 'matching-size' (parameter m, see Def. 1), tuples of different 'matching-sizes' can be placed in different subtrees. This feature is implemented by the *root* — it contains an array pointing to subtrees of different tuple 'matching-size'. To improve efficiency, all *inner-vertexes* are two layered: the first layer determined by the tuple-element type and the second layer by the tuple-element value. Each pair of valid elements connects this inner-vertex with an inner-vertex at the next layer, or with a leaf. Unlike the inner-vertexes which are supportive, the *leaves* contain the actual tuples. The design of the data structure implies that the leaves may contain only identical tuples (parameters in the 'matching part' of a tuple).

The distinguished roles of client and server, required by the framework, are implemented by separate trees for client-tuples (*client-tree*) and server-tuples (*server-tree*). The Tree data structure representing the server-tree is illustrated in Figure 6.3 (client-tree is structurally identical). There are five hypothetical tuples inserted into the tree: $T1 = (2, 1, printer, bglue)$, $T2 = (1, 1, printer)$, $T3 = (3, 1, printer, a, b)$, $T4 = (2, 1, 55, bglue)$ and $T5 = (2, 2, 123, memory)$. These tuples illustrate the usage of the data structure, rather than claim to represent any meaningful resources.

6.4.2 Implementation of the Trader Operations

As a tree is a recursive data structure, all operations are based on backtracking, tracing the particular tree top-down, performing an overloaded matching function on each tuple element, until either the matching fails (a tuple being searched for is not present), or a tree leaf is reached. According to the definition of each particular operation, one of the following actions takes place:

- operation WITHDRAWC searches only the client-tree, while WITHDRAWS searches only the server-tree, looking for an exact match (the operation is not restricted to the matching tuple elements, defined by tuple parameter m, but is performed on all tuple elements n).
- operation ADVERT searches the client-tree using backtracking to find all waiting client-tuples. The server-tuple is inserted into the server-tree, regardless of the result of the search.
- operation BIND searches the server-tree to find the requested server-tuple. The backtracking is interrupted when the first server-tuple is found. The client-tuple is inserted into the client-tree only if the server-tuple has not been found.

6.4.3 The Complexity of the Trader Operations

In this section we discuss the complexity of tree operations from the data structure point of view. We do not consider the complexity of the implementation (which includes processor and compiler dependencies such as memory allocation and deallocation time, efficiency of a function call, etc.) They are assumed to be performed in constant time $O(1)$.

[1] We are using the term tree (lower case t) for the tree-like data structure, in contrast to the component called Tree (upper case T).

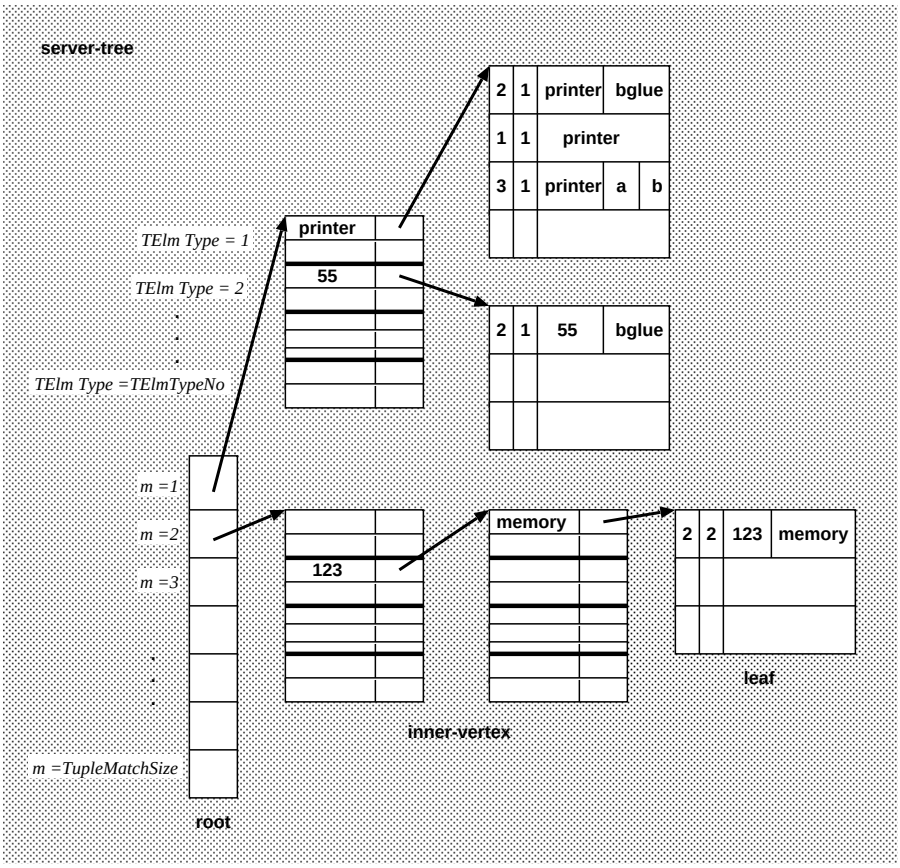

Figure 6.3: Tree data structure

Also, we do not consider the establishment of a binding performed by the GlueFactory and Binders. Firstly, it depends on the exact computing environment, and, secondly, it is beyond the Tree data structure functionality. In addition, in this section we do not compute the complexity of QoS-based operations, as they are discussed in section 6.6.

We consider the complexity of the following operations in the worst case. The following variables are used:

m	number of tuple matching elements
n	number of all tuple elements
$InnerSZ$	number of different values of a tuple type (the size of the second layer of the inner-vertex)
LSz	size of leaves
$Match$	the maximum time to perform the matching function on a tuple element of any type (as matching operation is user-customized, we consider the most 'expensive' tuple element type, for this purpose. This does not mean that the operation is time constrained by the MAGNET architecture.)

The time complexity to locate the first inner-vertex from the root pointing to it is a constant time $O(1)$. There are two unrelated variables to compute the complexity against: the length of tuples, and the time to perform the matching operation.

ADVERT and BIND

The complexity to search each inner-vertex is composed from the complexity to search the first layer, and the complexity to search the second layer:

$O(1)$	the first layer (the tuple element types), as this information is known
$O(InnerSz * Match)$	the second layer (values of tuple elements), as this must be searched

Therefore, the complexity of an inner-vertex is:

$$O(InnerSz * Match) + O(1) = O(InnerSz * Match).$$

Number of inner-vertexes is m, therefore, the complexity to search all of them is:

$$O(InnerSz * Match * m).$$

No matching functions need to be performed in the leaves, therefore, the complexity to search a leaf is: $O(LSz)$. If operation insert is required after the search has been performed, the algorithm is repeated on the complementary tree (e.g., the client-tuple searches the server-tree for binding, but it is inserted into the client-tree, etc). Therefore, the obtained complexity value has to be multiplied by two. Consequently, the overall complexity of operations ADVERT and BIND is linear in the tuple matching size:

$$2 * (O(InnerSz * Match * m) + O(ZSz)) = O(InnerSz * Match * m + ZSz).$$

For a particular tree of fixed variables $InnerSz$ and ZSz, the complexity is:

$$O(Match * m).$$

WITHDRAWC and WITHDRAWS

The complexity to search the inner-vertexes of the tree is similar to the previous case:

$O(InnerSz * Match * m)$.

However, matching functions need to be performed in leaves on the remaining tuple elements to find an exact match. Therefore, the complexity to search a leaf is:

$O(LSz * (n - m)Match) = O(LSz * n * Match)$ as $n \geq m$.

As the tree on which to perform the operation on is known from the operation itself (that is the reason why we distinguish between WITHDRAWS and WITHDRAWC), the search of the second tree can be omitted, in contrast to ADVERT and BIND.

Therefore, the overall complexity of operations WITHDRAWS and WITHDRAWC is also linear in the tuple size (for a particular size of tree):

$O(InnerSz * Match * m) + O(ZSz * n * Match) =$

$= O(Match * m) + O(n * Match) =$

$= 2 * O(n * Match) = O(n * Match)$.

6.5 The Trader

The Trader acts as a central hub and high-level interface for all components using MAGNET in a local federation. It contains distributed Tree components (semantically, not physically), and it is responsible for their distribution. In addition, the Trader's initiation procedure results in the allocation of Trees on the processors. In this section we cover these issues.

6.5.1 Tree Distribution

In accordance with our assumptions, typical federations will consist of tens of components, therefore, there is a need to support the parallelization in the matching process to prevent one matching function blocking all other components. This section discusses how the information pool (implemented as tree data structures) is 'divided' into separated Tree components which can be searched in parallel. Before we describe the approach taken in MAGNET's implementation prototype, we briefly mention other solutions to the problem of the distribution of the information pool.

For very small federations (roughly ten components), the information pool could have been centralized, and all operations (resulting in matching processes) would be done sequentially. However, for federations with tens of components this would not be feasible.

In contrast to the 'visible' distribution of the information pool implemented by MAGNET using the Tree components, an alternative approach could be to use a Distributed Shared Memory provided by MAGNET. We investigated this approach in the early stages of our research [34], and concluded that release consistency model would be adequate in terms of performance. However, as Regis was used for our prototype, the simplicity of implementation of the distribution in Tree components in Regis was the main reason for our approach. Further, Tree components are implemented in an efficient way in Regis. However, both solutions are only implementation features, not affecting the design of the MAGNET architecture.

Now we discuss our approach: the distribution of Trees follows the design of their tree data structure. There are two levels of distribution.

Firstly, the tuple *matching-size* defines the first layer of the client-tree and the server-tree, therefore it is used for distribution of subtrees into different Tree components. As tuples can only match complementary tuples of equal matching-size, the matching process can succeed only within the same subtree, hence, the same physical Tree component. Therefore, the root vertex sends tuples (according to their matching-size) to appropriate Tree components over an established binding.

Secondly, in order to further improve the parallelization, the client-tree and the server-tree are also physically distributed into different Tree components according to the *type* of their first tuple element. Consequently, the first layer of inner-vertexes will always contain only one tuple element type. This is feasible because Trees can be searched in parallel as processes (Regis components), even if there is not enough processors for physical distribution.

Therefore, the Trader is connected to a two-dimensional array of Tree components, where one dimension is defined by the maximum tuple-matching size (`TupleMatchSize`); the second dimension is defined by the number of tuple elements' types (`TElmTypeNo`). These constants are illustrated in Figure 6.3. However, as the latest version of Darwin does not support two-dimensional arrays, this was simulated using a one-dimensional array.

These constants must be configured according to the computing environment that uses MAGNET. Figure 6.4 illustrates the Trader and distributed Trees described in this section.

6.5.2 Tree Allocation on Processors

The Trader interacts with all Tree components over a binding established at configuration time. However, the level of distribution varies according to the processor configuration. MAGNET running in a distributed environment which enable full distribution of Trees can allocate all Trees to physically distributed processors. MAGNET running in a distributed system allowing only partial distribution (number of processors is smaller than the number of Trees), must allocate multiple instrances of Trees on a single processor, while equal distribution over available nodes is ensured. An extreme example of the partial distribution is a disconnected portable computer consisting of a single processor where all Tree components must run on a single CPU, therefore, no distribution is possible.

Assuming the following variables:

i is the index of the Tree component being allocated to a processor

$CpuNo$ is the number of processors

The Trader (`trader`) ensures equal distribution by allocating Trees (`treeComp`) to processors by the formula:

$i \,\%\, CpuNo$ determining the number of processor (% stands for MOD).

In Darwin, this is implemented as follows:

```
inst trader t;
forall i=0 to ((TupleMatchSize * TElmTypeNo) - 1) {
   inst treeComp ip[i] @Node (i % CpuNo);
   bind t.ipool[i] -- ip[i].all;
}
```

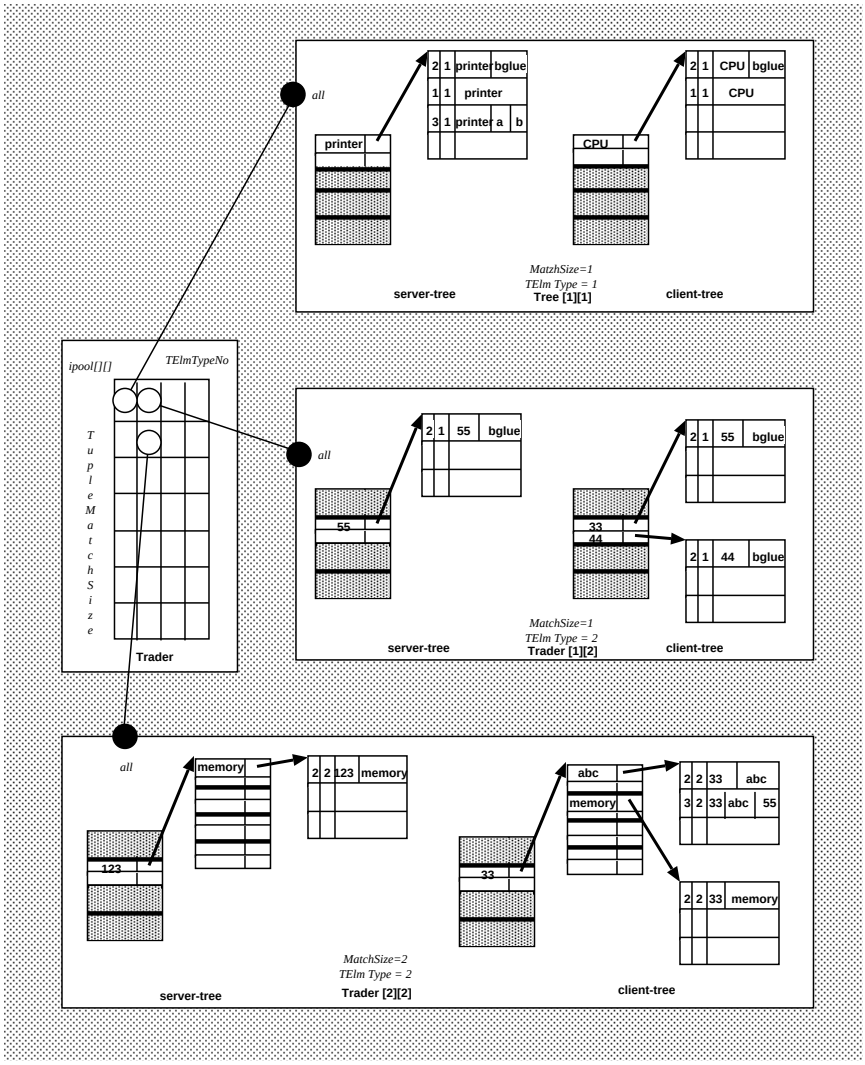

Figure 6.4: The Trader and distributed Tree components

6.6 QoS Management

In this section, we discuss interesting implementation issues concerning QoS Management. In particular, we discuss the implementation of QoS Definition (allowing the components to request tuples using QoS operators), and QoS Negotiation which covers the QoS-based matching process. We also compute the time complexity of the QoS operations ADVERT and BIND.

6.6.1 QoS Definition

In addition to actual values, QoS Definition allows tuple elements to express more complex requirements defined by a set of QoS operators (Def. 7). Consequently, QoS-match (Def. 8) allows operator-enhanced tuple elements to match, according to their definition.

Each of these operators is implemented as a derived class of the tuple element-type class which the operator is applied to. This approach allows us to implement the QoS-match simply by overloading the matching function in the derived 'operator' class. As all the operator-derived classes share the tuple element type with their base class, the tree data structure need not be modified. Inner-vertexes consists of the same number of tuple element types in their first layer, and can gain different values in the second layer, as illustrated in Figure 6.5. There is one server-tuple in the pool $T1$ and there are five client-tuples ($T2, T3, T4, T5, T6$) which QoS-match tuple $T1$.

$T1 = (3, 2, 12, a, bglue)$
$T2 = (2, 2, 12, a - d)$
$T3 = (3, 2, 12, a - d, 55)$
$T4 = (3, 2, 12, \neg w, a)$
$T5 = (2, 2, *, a|b)$
$T6 = (3, 2, *, a|b, 11)$.

All tuples are inserted into the Tree component [2][2] — holding tuples with two 'matchable' elements and whose first tuple element is of the second type according to the table of existing types (in our example, integer).

6.6.2 QoS Negotiation

The QoS Negotiation process allows the matching function to express a rating on every tuple element match (Def. 9), and a combination of these rates, QoS-rating match (Def. 8), defining component preferences.

In MAGNET, the rating is implemented as a return value of the overloaded matching functions (both original tuple element types, as well as the operator-enhanced types) gaining integer values in addition to 'true' and 'false'. The return values are summed, and a tuple-defined *threshold function* returning the threshold value X (Def. 10) is called to decide whether this combination is accepted (this implements the *select* operation).

The implementation does not require a different tuple matching process to be undertaken. That is the matching function (for the tuple class) is the same regardless of whether there are QoS attributes in the tuple or not. As was defined in section 6.3, in order to decide whether two tuples match, matching functions for every tuple element are called. However, in this case an overloaded QoS tuple element matching might be performed insted of the basic exact tuple element match. As exact matching is a special case of QoS-based matching, the default setting favours it: tuple element type functions returns 'true' or 'false', and the threshold checks for equality to m_1 by default (a number of matching elements) — a successful exact matching process returns 'true' by every tuple element, the result is always m_1).

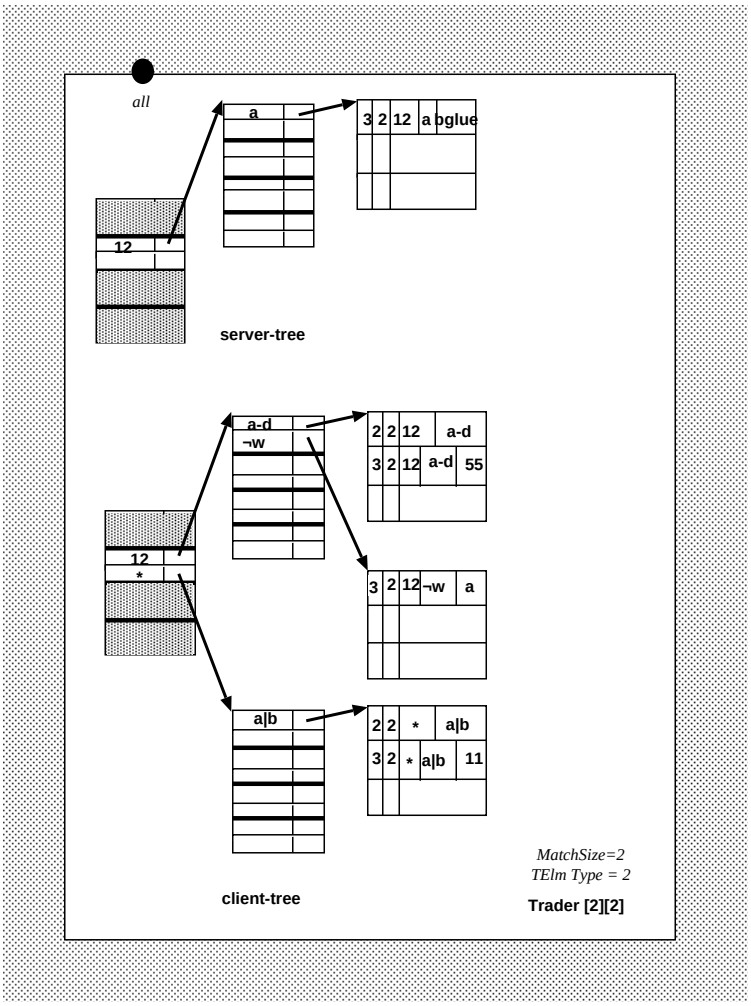

Figure 6.5: Tree data structure incorporating QoS Definition

Tuples performing exact matching, simply do not overload these functions, however components which require the extended flexibility have got the means to express complex requirements.

Although, the implementation of Trader functions need not be modified, the complexity might change significantly.

6.6.3 The Complexity of QoS-based Matching Operations

As operations WITHDRAWS and WITHDRAWC require an exact match, the number of inner-vertexes to search does not change, therefore, the complexity is equal to the exact matching case: $O(n * Match)$.

However, the complexity of operations BIND and ADVERT can be significantly worse.

ADVERT and BIND

The complexity to search each inner-vertex is the same as in the 'exact' search — the tuple-element type is known, therefore, the first layer search can be performed in $O(1)$. The second layer takes $O(InnerSz * Match)$ as it is irrelevant for the worst case if tuple elements contain QoS operator-enhanced values or only actual values. Therefore, the complexity to search an inner-vertex is:

$O(InnerSz * Match)$.

However, the number of inner-vertexes to search rapidly increases. Allowing QoS operator-enhanced values to be used as tuple elements results in the possibility of matching more than one value in each inner-vertex. That does not change the complexity of searching the actual inner-vertex, but changes the number of inner-vertexes to be searched from m to an exponential dependency:

$O(InnerSz^m)$.

The complexity to search all inner-vertexes, in the worst case, is:

$O(InnerSz * Match) * O(InnerSz^m) = O(InnerSz * Match * InnerSz^m)$.

Again, no matching functions need to be performed in leaves, therefore, the complexity of searching a leaf is: $O(LSz)$. However, in cases of operations ADVERT and unsatisfied BIND, an operation insert needs to be performed after the search — nevertheless, only one tuple in each inner-vertex can match, which leads into the same situation as the non-QoS matching; the complexity was calculated in section 6.4.3:

$O(InnerSz * Match * m + ZSz)$.

Therefore, the overall complexity of QoS-based operations ADVERT and BIND is a sum of the search and insert, which is exponential in the tuple matching size (again, the final equation takes tree variables (ZSz, $InnerSz$,) as constants):

$$O(InnerSz * Match * InnerSz^m) + O(ZSz)) + O(InnerSz * Match * m + ZSz) =$$

$$2 * (O(InnerSz * Match * (InnerSz^m + m) + ZSz)) = O(Match * InnerSz^m).$$

Application designers have to examine the tradeoff between flexibility and efficiency: linear exact matching is fast, but QoS-based matching offers advanced flexibility.

6.7 Limitations

In this section we discuss limitations of the tuplespace design. Firstly, we focus on implications for the architecture where it has to cope with a high number of components. Secondly, we discuss issues concerning data structure congestion as a result of a large number of tuples in the pool.

6.7.1 Large Number of Components

According to our assumptions, we expect a federation to consist of about tens of components handled by a single Trader. However, if this number reaches many hundreds of components, the Trader could become a bottleneck in the system. To overcome this problem, the federation would have to be divided into several, with tens of components in each, and interconnected using Locators. Therefore, matching would be performed in a fraction of the original information pool. Alternatively, clients would have to specify other Traders and use the Locators to perform the inter-federation communication. This solution brings additional complexity on clients by forcing them to call each particular Trader 'manually', using inter-Trader communication. A more appropriate solution for federations of this size would be to implement the tuplespace in distributed shared memory which is accessed by multiple Traders, as discussed in section 6.5.1.

A similar problem would appear if our assumption concerning the number of components accessing the Trader at a given time was no longer true. That is, we expected not more than ten components to access the Trader at the same time, however higher numbers of components would also cause the Trader to become a bottleneck in the system. Similar solutions as those discussed in the former case could be undertaken to improve the Trader throughput.

6.7.2 Large Number of Tuples

We assume that tens of component can generate roughly tens to hundreds tuples. Also, we expect that for this number of components, tens of types of tuple elements would be sufficient (as it defines the size of the tree data structure). If these assumptions are not the case, the information pool fills up. This results in the Trader being unable to handle new components. Therefore, they would have to try again when the pool is less full. Here we discuss the implications for the data structure and possible solutions to this problem.

The prototype of the architecture, implemented in Regis, was tested for small number of tuples (tens), for which this the performance was sufficient (matching in the Trader and binding establishment in Regis were performed in a few seconds.)

Higher numbers of tuples (hundreds) would still give good performance, if they were distributed equally according to their matching size and the type of the first tuple element (these two tuple features define the Tree component in which the tuple will be matched, as described above). Therefore, equal distribution of tuples into the Tree components, where they can be processed in parallel, would result in roughly tens of tuples being matched at the same Tree. We tested this, and the performance was also sufficient.

However, non equal distribution of tuples in the pool according to the type matching size (e.g., where some matching sizes are significantly larger than others) would result in the particular Tree component filling up. Overcoming this problem by configuring the tree data structure to keep large number of tuples (hundreds instead of tens), would probably result in non-acceptable responce time. If non-equal distribution of tuples is characteristic of a particular application running MAGNET, the distribution algorithm can be adjusted to allow more Tree components

for the larger matching sizes. This does not influence the structure of the overall framework, only the Tree distribution algorithm needs to be reconfigured.

Finally, very large number of tuples (thousands and more) would have the same affect — all Tree components fills. To solve this problem, either the federation would have to be divided, each having a separated Trader, interconnected together using Locators, or the distribution of the information pool would have to be organized differently.

6.8 Usability and Porting

In this section, we assess how easy it was to implement MAGNET in Regis, and discuss the implications when it is implemented on another platform.

6.8.1 Usability

Having implemented the prototype in Regis, we can claim that the framework is fairly usable for 'programmers'. Writing applications in MAGNET requires writing the appropriate code representing component functionality in C++ (the `body()` function). This includes declaration of tuples representing offered or required services, and calling particular Regis communication functions for sending tuples to the Trader.

However, in order to make the framework more user-friendly, a GUI could have been built to allow non-programmers to define their components. Avoiding programming the component functionality, users would have to only select and configure components from a group of pre-defined ones.

6.8.2 Porting

For reasons of flexibility, and implementation simplicity, we have chosen Regis to be the platform for our prototype. However, the framework can be ported to other existing systems supporting the notion of independent components defined by their services, providing distributed communication between entities (objects, components, etc.) and enabling runtime binding. For example, MAGNET can trade system resources in system like Exokernel or Nemesis, or it can deal with application level object, for example in CORBA or DCOM.

Porting our framework consists of two tasks:

- system component 'wrappers' (Trader, server, client) have to be defined as entities of the particular environment (we have used Darwin, from which Regis code was generated), and
- communication between system components (the Trader, clients, servers) enabling tuple transmission has to be implemented using the particular communication protocols (e.g., in Regis we have used the interaction style `Port<Tuple>`).

The remaining functionality of the architecture (tree data structure, tuple classes, etc.) could remain and be called from the ported 'wrapper' components.

6.9 Chapter Summary

In this chapter we discussed the implementation issues of the MAGNET framework. Its prototype is based on the *Regis* distributed environment which was slightly adapted to provide the bindings required by MAGNET.

Detailed discussion of every aspect of the implementation is unnecessary, and we focused on describing the system core and interesting implementation features.

MAGNET components are implemented as Regis components, defined by the Darwin configuration language. As MAGNET is designed for open dynamic systems, the dynamic binding in Regis was used extensively. Tuples are implemented as high-level containers enabling tuple elements to be user-defined and their matching operation customized.

Trees contain tree data structures supporting linear search (matching) for non-parametrized requests. The complexity of the Trader operations was calculated and was found to be *linear* in the number of tuple matching elements.

The Trader, acting as a hub for all components, is responsible for distributing the tree data structure over Tree components. Also, the algorithm of equal allocation of Tree components on the available processors in the system was described.

QoS Definition and QoS Negotiation are supported by overloading basic tuple element classes, their matching functions, and tuple rating functions. Although this feature allows components to express their requests with further flexibility, the efficiency of QoS-based ADVERT and BIND is *exponential* in the tuple matching size.

We have discussed the limitations of the architecture in terms of the number of components and the number of tuples, and briefly covered issues concerning usability and porting of the framework.

The last three chapters have presented a complete design of the MAGNET architecture. In the next chapter, based on the complete description of the MAGNET framework (provided in chapters 4, 5 and 6), we present several applications using MAGNET for allocation of their resources and provide an evaluation of the architecture.

Chapter 7

Case Studies and Evaluation

In this chapter we demonstrate the utility of MAGNET using several examples. Having implemented the prototype (in the Regis distributed environment), these examples provide a proof of concept. We will demonstrate that MAGNET meets the initial requirements (dynamic trading, extensibility, QoS Management, dynamic rebinding, information monitoring, and scalability) for several resource allocation problems. Firstly, we will simulate several system resources and user applications, in section 7.1. Based on this, in section 7.2, we will demonstrate quality of service allocation by simulating an application requesting resources described in terms of quality of service. Then, in section 7.3, we focus on dynamic issues of the architecture. We demonstrate dynamically changing network connectivity which illustrates the advanced system features such as monitoring, dynamic rebinding and scalability. Finally, in section 7.4, we evaluate of the architecture by discussing the features it provides, and the implications of the assumptions we have made by comparison with existing architectures.

7.1 System Components

In order to present examples of applications using MAGNET, we have to simulate essential system resources — CPU, memory, disk and printer. In this section, we define the functional interface of the components, offering services accessed by applications. In addition, we describe the relevant tuples representing services placed into the Trader. Figure 7.1 illustrates components described in this section connected to the Trader.

As the low-level design of the system components is beyond the scope of this thesis, we focus on presenting the functional interface — the component 'wrapper', rather than realistic hardware representation. For our examples, we assume that all components are running on processors connected to the Internet, are assigned IP addresses (forming the naming scheme), have network protocols (IP, TCP, UDP) installed and have running network daemons, such as *inetd* processing incoming packets, as usual.

7.1.1 CPU

Virtual CPU's functional interface consists of two fundamental functions:

`PID=cpu::alloc()` allocates an application to the processor by adding it to its priority-queue. Adds the application into the list of active clients kept by the Server-Rebinder, if appropriate. Returns process number `PID`.

`RET=cpu::leave(PID)` deallocates process `PID` from the processors. Deletes the record in the Server-Rebinder, if appropriate. An exit status `(RET)` is returned.

Internal schemes, such as an applied scheduling algorithm (implemented by a scheduler process), necessary network daemons (such as *inetd*) are run by privileged processes not accessible by the clients.

A typical processor, without attached devices, is described by a server-tuple: $CPU = (4, 3, CPU, manufacture, speed, ref)$, where ref is the reference to access the processor. It can be implemented, for example, as the processor's IP address and a socket number opened by the scheduler process. Other tuple parameters are self-explanatory.

7.1.2 Memory

In order to ensure that memory is allocated with its processor, they form a single component. Therefore, memory does not offer the functions `alloc()` and `leave()` as a part of the functional interface, because they are provided by the processor. However, Virtual memory's functional interface consists of two fundamental functions:

`RET=memory::write(ADD,buff)` writes a byte from a buffer `buff` to an address `ADD`. Function returns a status, `RET`.

`*buff=memory::read(ADD)` reads a byte from address `ADD` into buffer `*buff`.

Memory and attached processor is described by a server-tuple:

$CPU - MEM = (6, 5, CPU, manufacture, speed, memory, size, ref)$, where ref is the reference for accessing the processor, can be implemented as described in the previous section 7.1.1. Other tuple parameters are self-explanatory.

7.1.3 Disk

A disk should also form a single component with its processor, in order to ensure they are allocated together. For the same reason as memory, disk does not offer functions `alloc()` and `leave()` as a part of the its functional interface, because they are provided by the processor. However, its functional interface consists of two fundamental functions: `write()` and `read()`.

In order to provide enhanced flexibility for applications like DBMS, the disk function interface incorporates a paratemer (`No`) determining the number of blocks to be read from the address `ADD`. The default value is 1, however, applications can take the advantage of implementing 'grouping' write and read operations, as moving the disk arm (seek times) are expensive.

`RET=disk::write(ADD,No,block)` writes a number of blocks `No` from a buffer `buff` to an address `ADD`. A status `RET` is returned.

`*block=disk::read(ADD,No)` reads a number of blocks `No` from address `ADD` into a buffer `*block`.

A disk with attached processor, is described by a server-tuple: $CPU - DISK = (6, 5, CPU, manufacture, speed, disk, size, ref)$, where ref is the reference for accessing the processor can be implemented as described in the previous case. Again, other tuple parameters are self-explanatory.

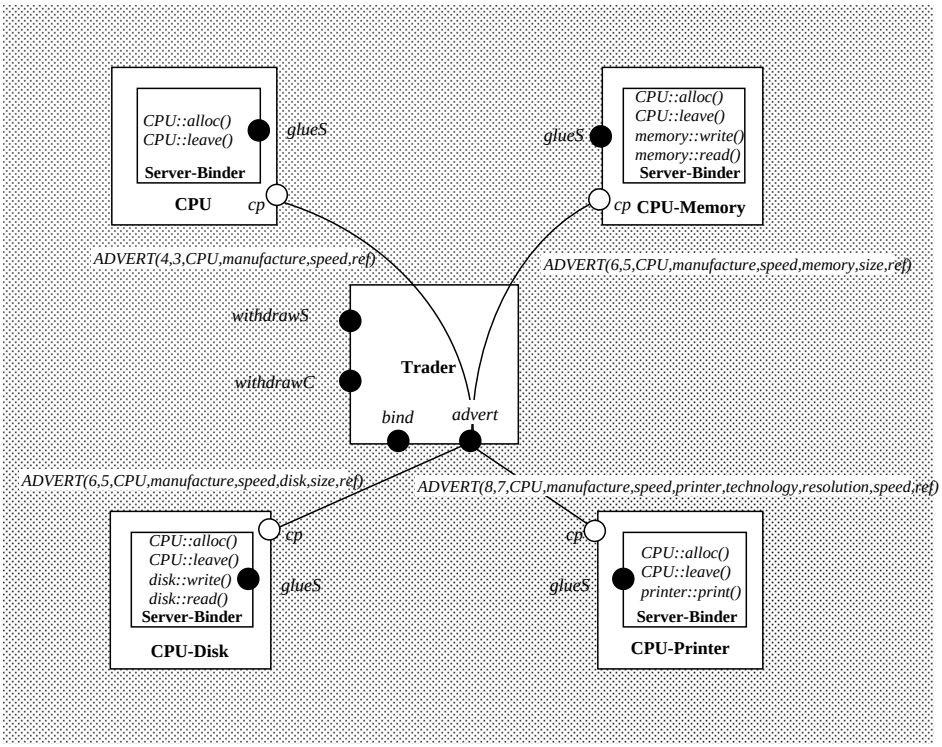

Figure 7.1: Essential system server components

7.1.4 Printer

A printer is a character device attached to a processor. For simplicity, we do not consider internal printer buffers. Like the memory and disk, a printer does not offer functions `alloc()` and `leave()`. `PID` numbers allocated by the processor function `alloc()` are used by the Virtual printer for protection of one application against another. The functional interface consists of one function: `print()`.

`RET=printer::print(PID,byte)` if the identification of the 'printing' application is `PID`, `byte` is printed out. Function returns `RET`, an exit status.

Advanced schemes, such as spooling are not considered in this example. Therefore, starvation cannot be avoided — applications allocated to the printer are only successful if the printer is currently idle — their call must be the first one after a previous application has detached from the component by calling `leave()`.

A printer with an attached processor, is described by a server-tuple:

$CPU - PRINTER = (8, 7, CPU, manufacture, speed, printer, technology, resolution, speed, ref)$ where *technology* defined a printing technology (such as, laser, matrix, ink-jet, etc.) and ref is the reference for accessing the processor, discussed in previous sections. Again, other tuple parameters are self-explanatory.

Figure 7.1 represents components described in this section: a processor, memory-processor, disk-processor and printer-processor. Interaction with the Trader enabling components to insert their advert-tuples into the pool is also illustrated. The reference to the Trader is obtained from a place known *a priori,* and this operation is performed by a parent process or by a human administrator inserting components into the system.

7.1.5 Discussion

As can be seen on examples discussed in this section, components representing services often rely on subcomponents offering additional 'subservices'. There are two ways for approaching this: encapsulating services, and forming service chains.

Firstly, encapsulation of services which rely on each other, is formed in advance (e.g., a printer encapsulates a processor, etc.) The composite component can be described by one tuple, and one matching process is sufficient to establish the binding with a client. However, a physical resource can form several components, each offering different functionality (e.g., the processor components in our previous example).

Secondly, services can be treated independently, each described by a tuple placed into the pool. When a client requests a service, the server generates a further client tuple requesting a particular subservice until the necessary chain of services is established, and the communication can be achieved. In this case, there are more tuples placed in the pool, and a single client-server communication might require several matching processes to be performed.

The framework supports both approaches — this is purely an administrator design decision. The former approach is less flexible, but more efficient in establishing the client-server binding, while the latter is more flexible, yet can cause additional problems with consistency or deadlock. In cases, when the chain of services cannot be established due to one server being unavailable, all components already bound are blocked. This can be prevented by introducing timeouts to tuples (this is discussed in section 7.4.2). In addition, rebinding of servers in a chain requires a third-party maintaining the overall consistency of all other relevant components in the chain.

7.2 QoS-based Allocation

This example demonstrates a quality of service based resource allocation. The situation was briefly described in chapter 5, here we will elaborate on it in greater detail. We discuss several client requests demonstrating various QoS requirements and different results of the matching process.

There are three server components in the system consisting of two subcomponents: Pentium processors, each of different speed and RAM memories of different size. They offer functions described in previous section, and are defined by following server-tuples:

ProcessorA 200 MHz with 32 MB RAM memory described by a server-tuple A:
$A = (6, 5, CPU, Pentium, 200, memory, 32, ref)$

ProcessorB 200 MHz with 16 MB RAM memory described by a server-tuple B:
$B = (6, 5, CPU, Pentium, 200, memory, 16, ref)$

ProcessorC 300 MHz with 4 MB RAM memory described by a server-tuple C:
$C = (6, 5, CPU, Pentium, 300, memory, 4, ref)$

In accordance with the tuple definition (Def. 1), definition ranges for all three tuples are $P_1 = P_2 = P_4 = N$ and $P_3 = P_5 = S$ where N is the set of natural numbers and S is the set of all words constructed from letters from English alphabet.

There are three applications in the system requesting processor and memory. An Application D expresses its request using QoS-rating operators, and the QoS-rating match, while Application E uses only QoS-rating operators to define its requirement, Finally, an Application F described its requests by the basic exact match.

Server-Tuples	Client-Tuples
$C = (6, 5, CPU, Pentium, 300, memory, 4, ref)$	
$B = (6, 5, CPU, Pentium, 200, memory, 16, ref)$	
$A = (6, 5, CPU, Pentium, 200, memory, 32, ref)$	

Table 7.1: The Information Pool containing tuples A, B, and C.

QoS-rating match	$m_1 = m_2$	$P_i = Q_i \; \& p_i \in f_i(q_i)$ $\forall i \in \{1, m_1\}$	$\sum_{i=1}^{5} k_i \geq 5$	result
tuples: C D	$5 = 5$	$for\ i = 5:$ $4 \notin 16 - 64 \Rightarrow no$		$do\ not$ $match$
tuples: B D	$5 = 5$	yes	$1+1+1+1+1 =$ $= 5 \geq 5$	$match$ $ALLOCATED$
tuples: A D				$not\ tested$

Table 7.2: QoS-based allocation — matching between tuples A,B, C and D

Table 7.1 illustrates the information pool for the described system is configuration. The table illustrates a schematic view of the information pool, rather than a realistic representation reflecting implementation issues (Trees components, etc.) Components join the system in the order of ProcessorC, ProcessorB, ProcessorA and Application D, Application E, and Application F. All server-tuples A, B, and C representing processors are inserted into the server part of the pool.

Application D

The Application D requires the fastest available Pentium processor running at least on 200MHz with at least 16MB memory. The request is defined by a tuple D:
$D = (6, 5, CPU, Pentium, 200 - 300, memory, 16 - 64, ref)$.

The rating values are:

CPU	$\Omega(CPU) = 1$ (exact match)
Pentium	$\Omega(Pentium) = 1$ (exact match)
200-300	$\Omega(-(200)) = 1$, $\Omega(-(300)) = 2$
memory	$\Omega(memory) = 1$ (exact match)
16-64	$\Omega(-(16)) = 1$, $\Omega(-(20)) = 2$, $\Omega(-(32)) = 3$, $\Omega(-(64)) = 4$
X	5 (threshhold)

When an operation BIND is called on the tuple D, a match is found, and requested binding can be established, therefore, the tuple D is *not* inserted into the information pool (in accordance with operation BIND).

Table 7.1 illustrates the matching process resulting in the application being allocated the component described by tuple B. Although tuple A would fulfill client's requirements better ($1 + 1 + 1 + 1 + 2 = 6 \geq 5$), it is not tested because tuple B was already allocated to the application. The example shows how, from the component's point of view, non-deterministic placement of tuples into the pool determines the resultant allocation.

Server-Tuples	Client-Tuples
$C = (6, 5, CPU, Pentium, 300, memory, 4, ref)$	$F = (6, 5, CPU, Pentium, 300, memory, 2, ref)$
$B = (6, 5, CPU, Pentium, 200, memory, 16, ref)$	
$A = (6, 5, CPU, Pentium, 200, memory, 32, ref)$	

Table 7.3: The Information Pool containing tuples A, B, C and F.

Application E

The Application E requires a Pentium processor running on 300MHz with any amount of memory. The request is defined by a tuple E:
$E = (6, 5, CPU, Pentium, 300, memory, *, ref)$.

When an operation BIND is called on the tuple E, it matches against the tuple C, and binding can be established. Also in this case, the client tuple E is *not* inserted into the information pool in accordance with operation BIND.

Application F

Finally, the Application F requires a Pentium processor running on 300MHz with 2MB memory. (This requests illustrates the features of the matching process, and does not claim to be a realistic resource requirement.) It is defined by a tuple F:
$F = (6, 5, CPU, Pentium, 300, memory, 2, ref)$.

An operation BIND is called on the tuple F, however, it does not match any server tuples currently present in the pool, evenhough the tuple C would provide the required resources. Therefore, without this exact match no binding can be established, the tuple F is inserted into the client part of the pool, and the Application F is waiting. Table 7.3 illustrates the information pool after the tuple F has been inserted.

This final example illustrates the drawback of using an exact matching (Application F remains waiting in spite of the fact that ProcessorC would provide adequate resources). This further highlights the advantage and flexibility of QoS definitions supported by the architecture.

7.3 Dynamic Network Connectivity

In this example we will demonstrate dynamic issues of the Magnet architecture in terms of scalability, dynamic rebinding and monitoring, in addition to dynamic binding and QoS-based matching. Dynamic network connectivity is a good example of an application requiring adaptation. That is the system must be able to adapt from disconnected operation, through weakly connected, to fully connected. We will illustrate adaptation of the system to changes in connectivity on two typical applications — a word-processor and a Web client.

7.3.1 Disconnected Case

In a 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, client-tuples (PRINT, and WebClient) are inserted into the Trader to represent these requests:

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

Table 7.4: The Portable Information Pool— the disconnected case

Printer a request for a laser printer with resolution of 600dpi, any speed, attached to any processor:
$PRINT = (8, 7, CPU, *, *, printer, laser, 600, *, ref)$

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 fullfilled, as there is no printer component nor Web server connected to the system (which is disconnected).

Table 7.4 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.

7.3.2 Weakly Connected Case

In this example, the portable is weakly connected by a modem, below we describe an establishment of a binding between the Web Client and the Web Server.

Inserting Required Information

Applications which want to take the advantage of the portable being connected to the Internet later on, have to insert the 'joining' and 'leaving' tuples into the pool. In addition to the PRINT tuple, the word processor also inserts following two tuples:

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

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

In order 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.)

The Web Client also wants to take the advantage of the portable being connected to the network, but it can 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, the Web Client can be connected to the server by any network hardware. Transmitting Web pages over a mobile line is feasible, however, printing a job at a printer in a remote office is not desired. Therefore, the printer tuple does not express options of network connections.

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 7.5: The Portable Information Pool — the weakly connected case

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 7.6: The Office-Based Information Pool

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, 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 7.5 illustrates the information pool with all the client tuples and the 'joining' tuple T1 inserted. Again, the table is schematic, implementation issues are omitted.

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.

The laser printer component with resolution 600dpi and speed of printing 40pages per minute is described by a server-tuple

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

The processor running the Web Server can communicate by 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 7.6 illustrates the configuration of the office-based information pool before the portable computer dialled in.

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.

Server-Tuples	Client-Tuples
$T1 = (3, 2, join, modem, bglue)$	$PRINT = (8, 7, CPU, *, *, printer, laser, 600, *, ref)$
$T2 = (3, 2, join, network, bglue)$	$WebClient = (3, 2, WebS, modem, ref)$
	$PRINT2 = (12, 2, join, network, 8, 7, CPU, *, *, printer, laser, 600, *, ref)$
	$PRINT3 = (12, 2, leave, network, 8, 7, CPU, *, *, printer, laser, 600, *, ref)$
	$WebClient4 = (7, 2, join, network, 3, 2, WebS, network, refU)$
	$WebClient3 = (7, 2, leave, modem, 3, 2, WebS, modem, ref)$
	$WebClient5 = (7, 2, leave, network, 3, 2, WebS, network, refU)$

Table 7.7: The Portable Information Pool — the fully connected case

7.3.3 Fully Connected Case

In this section, we illustrate an establishment of new bindings which take place as a result of the portable being plugged into the network.

Inserting the Required Information

The Web Client is currently communicating with the Web Server by a modem. If the portable is connected to a network by a LAN adaptor, 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 a third-party renegotiated third-party rebinding. The Updater inserts following two tuples into the portable 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.

Trader Connection

When the portable arrives into the office, it 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 tabale 7.7) contains two joining server-tuples, all original client-tuples except the tuple $WebClient2$ which has been removed when the Web Client was weakly connected to the Web Server. In addition, two tuples from the Updater $WebClient4$ and $WebClient5$ for 'join' and 'leave' have been inserted.

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 (as described in chapter 5). Both tuples are reinserted into the office-based information pool (see table 7.6) 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 (which is accessed by the functional interface defined in section 7.1). As a result of this, the word-processor job can be printed.

The Updater tuple case is more complex: extracted tuple $(3, 2, WebS, network, refU)$ matches waiting server-tuple $WebServer1$ (see table 7.6). However, when the binding between the Web Server (by the network port) and the Updater is established, the Updater performs all steps necessary for the third-party renegotiated third-party rebinding to take place, as described in section 5.3.6. Finally, the resultant binding between the Web Client and Web Server takes place over the network.

Discussion

As MAGNET requirements were defined in terms of requirements, we needed to demonstrate how they were met by using MAGNET in applications requiring this support. Dynamic resource allocations, discussed above, demonstrated that MAGNET achieves its goals — providing QoS-based user-customized adaptable resource management of diverse resources.

7.4 Evaluation

In this section, we will evaluate the architecture by discussing the features it provides, elaborating on the implications of the assumptions we have made, and comparing it with other trading frameworks.

7.4.1 Evaluation of Provided Features

The MAGNET architecture proposed in this thesis was specified in terms of the features it should provide in order to meet the requirements of applications in dynamic and mobile environments. These features include: dynamic trading, extensibility, QoS-based management, dynamic rebinding, information monitoring, and scalability.

There are three fundamental ways to evaluate features provided by an architecture, such as MAGNET — a theoretical proof, experimental measurements, and a case study together with an informal discussion. As these six features were not specified in terms of formal mathematical definitions, no theoretical proof could be presented to demonstrate that they have met the original requirements.

Also, the framework could not have been evaluated in terms of performance measurements for two reasons. Firstly, the architecture defined by the six features addresses flexible and dynamic issues, in contrast to performance results (e.g., claiming that the framework is extensible cannot be measured in seconds). Secondly, any performance results measuring the time required to match tuples and to establish a dynamic binding would be imprecise and non-representative as the framework relies on multi-variable computing environment which consists of the processor (its speed), network (its connectivity and current traffic), operating system (efficiency of system calls), Regis (efficiency of its implementation), etc.) For these reasons, we believe that performance measurements are not useful.

Therefore, we have to evaluate the architecture by case studies and an informal discussion. We have illustrated the usage of MAGNET on two examples: QoS-based resource allocation (section 7.2), and dynamic network connectivity (section 7.3). Here, we discuss what is supported, and what is not supported for each of the six features, with respect to specifications defined in chapter 1.

- **Dynamic Trading**. Dynamic trading was defined as a third-party matching of service requests against demands described by a type of service, not directly by a name. The Trader, based on a tuplespace paradigm, provides this functionality by matching tuple elements defining features of service provisions and requirements, and establishing a binding.

- **Extensibility**. Extensibility was defined at two levels. Firstly, existing services and data formats should be extended (new resources, services and user requests can be defined at run-time). This is enabled by deriving new tuple element classes. Secondly, the matching process performed by the Trader could also be dynamically redefined (resource allocation strategies could be user-customized). This is supported by allowing users to overload the matching function for each tuple element class. As there are no restrictions on semantics of tuple elements, the framework can be used for any applications requiring third-party trading, beyond the scope of resource management.
  However, we assume that the extending matching functions are secure in terms of returning control back while not modifying data of other tuple elements (they are expected to be 'well-behaved'). We have chosen full generality and full extensibility (in terms of user-defined matching functions) compromising security rather than providing secure, yet restricted extensions. We believe that the power of full user-customization overweighs the risk of potential problems. We further elaborate on problems caused by insecure matching functions in section 7.4.2.

- **QoS-based Management.** Extensibility and flexibility of the architecture enables QoS Management which we defined as QoS-based selection of services. This is addressed by enabling users to enhance service definitions by QoS operators (which can be user-customized), and by enabling the QoS Negotiation (by QoS-rating operators and the QoS-rating match function). Apart from QoS-based selection, QoS Maintenance, enabling two adaptation strategies (resource management and application adaptation) is also supported.
  However, we did not fully address dynamic and continuously changing features, such as network traffic, throughput etc. as the framework is not suitable for applications requiring very fine grain updates (seconds and milliseconds) due to rebinding and matching overhead.

- **Dynamic Rebinding.** Based on the core of the framework providing dynamic trading and binding, MAGNET (in cooperation with application level components, Rebinders, Updater and Administrator) also supports all the required types of rebinding and renegotiating — first-party rebinding, third-party rebinding, first-party renegotiating, third-party renegotiating, and no-renegotiating.
  However, consistency is assumed to be maintained by the components themselves during all rebinding actions. MAGNET does not support recovery from inconsistent states, such as a component crash, rebinding components when they were not safe to do so. We further discuss issues concerning consistency in section 7.4.2.

- **Information Monitoring.** In addition to a manual update, monitoring of all service features ('classical' and QoS-based) is supported by user-level components, Updaters, and Monitors.
  As was discussed above regarding QoS, the framework is not suitable for real-time applications, or those requiring very fine grain updates (in terms of seconds and milliseconds).

- **Scalability.** All provided features are supported within a federation. Scalability of the architecture is defined at two levels: firstly, dynamic reconfiguration of domains (supported by operations JOIN and LEAVE) which do not assume the users need to know the identity of the Trader they want to join. The second case, scaling the architecture within the limits of the computing environment, faces a tradeoff between response time and 'precision' of provided information (such as, finding all matching tuples). Therefore, the framework enables a tuple to be passed over to a particular Trader for processing (using Locators). However, in this case, the user must know the remote Trader identity in advance. We supported these two cases, as they are typical in mobile and adaptive systems. Examples are, a user with a portable computer roaming around and using resources in various offices is an example of the first case. An example where a user requires to print a job at a printer in a remote office where he will travel to later, illustrates the need for an inter-Trader communication.

  However, 'world-wide' scalability (performing tuple matching in all information pools) is not feasible in MAGNET due to unacceptable response time. A different approach would have to be undertaken, in order to provide this functionality.

7.4.2 Discussion on Assumptions

In chapter 3, we have defined assumptions for the design of the MAGNET architecture. Here we summarize them, and discuss the implications for the architecture if these assumptions were not the case. In addition, we propose a solution to possible problems caused by the assumptions being invalid.

1. **Consistency**. All system components are assumed to maintain consistency. That is we assume that: rebinding can be performed only when the system is in a safe state, an unexpected component crash cannot happen, and a component when finished its operation must leave MAGNET in a consistent state (all tuples from the pool must be withdrawn, and all allocated resources must be released).
   Dealing with inconsistency caused by an unsafe rebinding or unexpected component crash from the component point of view would need a powerful fault tolerant framework (featuring transactions, replications, rollbacks, etc.) This is beyond the scope of this thesis. However, from the Trader point of view, the problem can result in out-of-date tuples being left in the pool. In order to prevent this, the Trader can periodically clean the pool performing garbage collection. However, in order to find out whether components are still alive, they would be obliged to provide a 'still alive' function which would be called by the Trader before removing the tuple. Another solution for clients would be to equip their tuples with timeouts limiting how long their tuples are to wait in the pool, before they can be garbage collected. In addition, in order to prevent components from leaving tuples in the pool when they finish the operation, a special subcomponent (present in every component) could automatically withdraw all inserted tuples. However, this solution requires cooperation with the component, in terms of initialization of the operations, so it is not fully automated.

2. **Protection**. By protection we mean that all components, servers and clients, are responsible for ensuring the validity of the tuples.
   As the architecture does not restrict the semantics of contents of the tuples,

there is no means to check the validity of the information. Placing a tuple with a non-existent request or offer in the pool can result in an attempt to establish a binding between non-existing components, or incompatible services. In order to prevent this, the framework can authorize components to call the Trader functions, or introduce capabilities (as tuple elements) to improve the component protection. In addition, garbage collection, or the 'still alive' function can be used, as discussed in the previous paragraph.

3. **Synchronization**. Components are responsible for synchronization. This includes communication with the MAGNET framework as well as component-to-component interaction.
   The architecture can provide an additional function for client components (operation BINDRET) which would perform the same matching process as operation BIND, and return 'no' instead of blocking the component if the requested server has not been found in the pool. There is no need for an equivalent operation to be provided for server components, as they offer service regardless the interest of clients. As for the component-to-component interaction, there is no reason for MAGNET to interfere in synchronization of components themselves.

4. **Security**. The user-defined matching functions are assumed to be secure in terms of returning control back to the Trader, while not altering other system data. The implementation of the matching functions as overloaded C++ functions does not allow the matching function to alter the protected data of other tuple elements, however this does not provide full hardware protection. Furthermore, the architecture cannot check if the function will return the control back to the Trader, as this is the halting problem. This might result in the Trader getting blocked by a 'non-secure' function. A solution to this problem, not restricting the extensibility we are aiming for, would be to finish any matching function by force after a timeout period.

5. **Federation Scale**. We assumed the number of components in a federation to be roughly tens, they could generate tens to hundreds tuples placed into the information pool. In addition, more than ten components accessing the Trader at the same time would result in non-acceptable responce time.
   A high number of components can result in the Trader becoming a bottleneck in the system. The same affect would be observed if more than ten components were accessing the Trader at the same time. We have dealt with this problem in section 6.7.1 discussing possible solutions, for example distributed shared memory. Problems of congestion due to a large number of tuples placed in the pool were discussed in section 6.7.2.

6. **Frequency of Change.** The framework is designed for components that will change their features with a frequency of minutes and hours, rather than seconds and milliseconds. Therefore the proposed support for monitoring and rebinding as a result of a change is adequate.
   The support for applications requiring finer grained updates (with frequency of seconds and milliseconds) would not be viable. This can be improved by enabling a direct access to the Tree components for trusted Monitors and Updaters, as suggested in section 5.1.3. However, for environments with very frequent changes, or those which rely on real-time response, our framework is not suitable.

7. **Service Characteristics.** We assumed that tuples have not more than tens of elements, and are equally distributed according to the number of elements (tuple matching size). In addition, we expected the number of types of tuple

elements to be not more than tens.
Exceeding the number of tuple elements, the number of tuple element types, or non-equal distribution of tuples according to the tuple matching size might lead to the problem of congestion of the tuplespace or a particular Tree. We have discussed implications and solutions to this problem in section 6.7.2.

8. **Naming**. Naming of the computing environment used for Trader naming is assumed to provide unambiguous names in the scale of the environment. In addition, names are constructed in a way that they can form a hierarchical tree structure with a single root, and a unambiguous path in the tree between two Traders can be determined.
If the computing environment does not provide naming which meets these requirements, there must be an additional Trader naming scheme defined. However, it can be derived from common naming schemes, such as IP addresses.

7.4.3 Comparison with Existing Architectures

In this section, we briefly compare MAGNET with other trading architectures which were introduced in chapter 2. Then, we give examples of platforms suitable for porting MAGNET to, and those which do not provide the required flexibility.

Architectures providing service matching based on a special 'matching' component, such as Matchmaking (the Matchmaker Component), or Aster (the Aster Selector) also perform dynamic service coupling. However, we believe that the 'matching' component (the Trader in MAGNET) should be universal and user-customizable. In contrast to this, the discussed platforms rely on a 'knowledgeable' component which decides on component coupling, but can perform only a non-customizable matching function.

Other tuplespace-based architectires (such as Limbo, Osprey, Jini, etc.) also do not provide extensibility in terms of matching function customization. In addition, Limbo and Osprey implement tuple-typing in contrast to the universality of our approach, which we aimed for. As for the typing, we believe that the system components are of different types, however, they can describe their service characteristics in a universal, clean format, as a collection of features — the tuple.

Further, we distinguish between two levels of approaching the problems: 'component (or object) level' where typing is desirable, and 'component description level' which can provide universality. For this reason, we did not adopt tuple typing, nor did we use other models which require typing (e.g., C++ objects, tagged trees such as XML). Another approach, Cardelli's Ambient Calculus [12], was devised to match characteristics of wide-area networks and systems composed of objects communicating among themselves through reliable and transparently accessible object interfaces. Similarly, this was not adopted as it addresses the typed 'component level' rather then the universal 'component description' level.

MAGNET can trade entities of varying granularity due to the universality of the framework. It can be used for resource allocation, at an operating system level in flexible component-based systems, such as Exokernel, Nemesis, DEIMOS, etc. On the other hand, it can also trade objects in user-level applications, for example JavaBeans, CORBA objects, DCOM objects, etc. If applications in these systems followed the assumptions on our famework (regarding the number of components, consistency, etc.), MAGNET would provide a powerful and flexible trading functionality. Unsuitable platforms for this type of trading are for example Unix, or SPIN due to their monolithic nature.

7.5 Chapter Summary

In this chapter we have demonstrated the utility of the MAGNET architecture on several examples. Based on the prototype of the MAGNET framework (described in chapter 6), we have built a simple resource allocation system demonstrating particular features of the architecture.

Firstly, we have used an example of the operating system's basic resources to show how MAGNET can be used in an extensible operating systems. We modeled a processor, memory, disk, and printer — by defining their functional interface and the tuples describing their services. We demonstrated QoS based resource allocation on a client requesting a processor and memory. Then, we focused on dealing with dynamic changes in the system — dynamic network connectivity. We described the allocation of a printer and a Web Server to clients running on a portable computer in three situations — disconnected, weakly connected and, fully connected. Advanced features, such as monitoring, rebinding, scalability were illustrated in this example. Finally, we have also evaluated the architecture by discussing features it provides, elaborating on implications of assumptions we have made, and by comparing MAGNET with other existing architectures.

This chapter concluded our study of the resources management architecture, MAGNET, by demonstrating its flexibility, universality and feasibility in particular examples and by providing evaluation of the framework.

Chapter 8

Conclusion

As a result of recent changes in computing environments, there has been an increasing need for *dynamic resource management* providing trading resources defined in terms of the type of service they offer. Additional requirements posed by users in dynamic environments include QoS-based description of resources, user-customization of allocation strategies, and runtime adaptation to changes in computing environments. This thesis has described a dynamic resource management architecture, MAGNET, meeting these requirements.

MAGNET provides component-customized QoS-based *trading* of server definitions resulting in establishment of a requested component binding — resource allocation.

This chapter reviews this thesis by summarizing the goals and achievements of the MAGNET architecture (section 8.1), presenting possible directions for future research (section 8.2), and concluding the work by final remarks (section 8.3).

8.1 Thesis Review

This thesis has argued that the role of resource management has significantly changed due to two factors: recent *technological improvements* which resulted in an increasing diversity of computing environment and the boom in mobile computing, and the *inability of traditional operating systems* to provide a flexible user-customized platform where implementation of dynamic resource allocation strategies is feasible.

In this section we recapitulate four major areas this thesis has addressed: identifying a new role of resource management, the model of dynamic third-party trading applied to MAGNET, the dynamic resource management architecture, and BITS, the component-based architecture.

8.1.1 A New Role of Resource Management

Resource managers in dynamically changing systems must fulfill requirements for user-customization, extensibility and adaptability. We have mapped the field of operating systems and resource management, and identified the new role of and requirements for resource managers.

8.1.2 A Model of Dynamic Third-party Trading

The design of MAGNET is based on a model of dynamic third-party trading of resources based on requests for the type of service. Although the model is primarily

designed for dynamic resource management, its generality makes it suitable for any system requiring trader-based dynamic binding.

The model enables MAGNET to meet the initially identified requirements for dynamic resource management:

- **Dynamic Trading.** Resource coupling by a component in a third-party role, the Trader, is based on requesting services by their types, rather than by names. Component *export services* to the Trader which performs a matching process resulting in an establishment of a dynamic binding. This *binding process* is performed through cooperation of components and the Trader, and comprises three phases: exporting service definition, negotiating service definitions, and establishing a communication channel.

- **Extensibility.** Extensibility of the architecture is provided at two levels: services definitions exported into the Trader can be user-defined, and consequently, the matching process can be user-customized.

- **QoS-based Management.** The extensibility of the model enables QoS-based management. In particular it allows the resource definitions to be parametrized by QoS-based characteristics. In addition, user-customized QoS-based matching processes can be incorporated. QoS Management, dealing with all aspects of non-functional behaviour of components, consists of three phases performed in sequence: QoS Definition, QoS Negotiation and QoS Maintenance.

- **Dynamic Rebinding.** Dynamic rebinding is defined by the *rebinding process*. Performing adaptation to changes in system conditions, it follows the semantics of the binding process. This consists of four phases: exporting service definition, renegotiating service definitions, destroying a communication channel, and reestablishing a communication channel.
  According to the role of component initiating the process, the model distinguishes between two types of rebinding: *first-party rebinding*, and *third-party rebinding*. Also, the model defines two more cases considering the component responsible for renegotiation of the replacement: *first-party renegotiation* (unbound components are left to find a new component themselves), or they are given a new peer. Semantically, this can be found in the Trader (*third-party renegotiation*), or obtained from an external third-party (*no-renegotiation* is performed).
  Components, treated as black-boxes, are responsible for maintaining system consistency, therefore the model does not handle inconsistent states.

- **Information Monitoring.** Monitoring is implemented by *monitors* ensuring service information kept in the Trader is up-to-date. It was provided by the extensibility of the model, therefore, there is no need for a dedicated support at the model level.

- **Scalability.** Scalability of the framework is supported by defining a notion of *federation*, a local scale distributed system with one Trader. Internetworking between federations provides scaling of the architecture.

8.1.3 MAGNET: A Dynamic Resource Management Architecture

Based on the framework discussed above, MAGNET provides trading of service definitions based on a tuplespace paradigm. Here, we summarize how MAGNET actually supports the initial requirements for the defined trading model.

- **Dynamic Trading**. The Trader, based on a tuplespace paradigm, consists of three components: the information pool, Trader operations on tuples, and the matching operation. Dynamic trading is performed by enabling components to define their services in terms of tuples placed into the information pool (by operations ADVERT, and BIND), performing a matching operation and establishing a resultant binding. Tuples can be withdrawn from the Trader by complementary operations, (WITHDRAWC and WITHDRAWS). Components involved in the binding process (Binders, GlueFactory, Tree) were also defined.

- **Extensibility**. By enabling users to redefine tuple formats and user-customize the matching function, MAGNET supports *extensibility*.

- **QoS-based Management.** Extensibility and flexibility of the architecture enables QoS Management. Firstly, QoS Definition introduces QoS operators (which can be user-customized). In addition, within the QoS Negotiation phase, components can formulate their preferences by *QoS-rating operators* and *QoS-rating match* in order to select the best tuple among a group of matching tuples. QoS Maintenance, based on QoS Monitoring, enables two adaptation strategies to be implemented: resource management and application adaptation.

- **Dynamic Rebinding.** Tuplespace-based implementation of four phases of the rebinding process was described. In addition, we have discussed all components involved in the process (Rebinders, Updater, Administrator) as well as different situations into which the system may transforms (as a result of different components performing the initiation and renegotiation).

- **Information Monitoring.** Components for monitoring (Updater, Monitor) of service definitions placed into the information pool (for both parties — clients and servers) ensure that component tuples are up-to-date at all times.

- **Scalability.** MAGNET supports operations JOIN and LEAVE to enable mobile users to use local resources transparently in a site where they have arrived. In addition, the architecture also supports scaling by enabling a tuple to be passed over the trading system to a particular Trader for processing using special components, Locators.

8.2 Future Work

This section identifies areas of future research, discussing problems beyond the scope of this thesis, or identifying potential alternative design decisions to those undertaken in MAGNET.

8.2.1 The Resource Management

In this section we identify alternative design approaches to those implemented in MAGNET.

- **Distribution of the Information Pool.** In contrast to the 'visible' distribution of the information pool implemented by MAGNET using the Tree components, an alternative approach uses a Distributed Shared Memory, as discussed in chapter 6. We investigated this approach in the early stages of our research [34].

- **Protection of the Information Pool.** MAGNET assumes that all tuples placed into the pool represent the services of existing components. There is no notion of protecting applications against misleading components inserting tuples which represent non-existing services. An additional protection scheme might be implemented, such as an authorization of components to call the Trader operations.

- **Consistency of the Information Pool.** According to our assumption, MAGNET cannot be responsible for consistency of the information pool. An alternative approach, MAGNET's responsibility over tuples in the pool, leads to investigation of garbage collection of out-of-date tuples left in the pool, as discussed in chapter 7.
  Within the black-box approach, the only way to find out whether the component which inserted the tuple into the pool still exists is to call a dedicated operation 'still alive' provided by the component itself, or by introducing timeouts. Implications for system performance are worth investigation, in particular in the case of operations JOIN and LEAVE.

- **Diverse Applications of the Framework.** The utility of the MAGNET framework was illustrated in several operating system-based examples, in chapter 7. As the architecture was designed to be suitable for any applications requiring trader-based dynamic binding and rebinding, there are more diverse applications that can use the potential of the architecture. An example of dynamic taxi-controlling system, based on MAGNET, was investigated during our research [35].

8.3 Summary

This chapter has presented general conclusions and also areas of possible future research. We have summarized this work by recapitulating initial requirements (dynamic trading, extensibility, QoS-based management, dynamic rebinding, information monitoring, and scalability) and presenting how they were met, at the model level and at the MAGNET architecture level.

Our research on MAGNET has demonstrated the feasibility of dynamic resource management which provides flexible QoS-based description, negotiation of services, user-customization of allocation strategies, and runtime adaptation to changes in computing environment.

Bibliography

[1] A.P.M. Ltd. *The ANSA Reference Manual Release 01.00.* APM Cambridge Limited, UK, March 1989.

[2] C. Aurrecoechea, A. Campbell, L. Hauw. *A Review of Quality of Service Architectures.* ACM Multimedia Systems Journal, Internal report number MPG-95-10, November 1995.

[3] M. J. Bach. *The Design of the Unix Operating System.* Prentice Hall International, Inc., 1986.

[4] A. Benerjea, B. Mah. *The Real-Time Channel Administration Protocol.* In Proceedings of the Second International Workshop on Network and Operating System Support for Digital Audio and Video, IBM ENC, Heidelberg, Germany, 1991.

[5] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, S. Eggers. *Extensibility, Safety and Performance in the SPIN Operating System.* In Proceedings of the 15th ACM Symposium on Operating Systems Principles, pages 267-284, Colorado, USA, December 1995.

[6] L. Besse, L. Dairaine, L. Fedacui, W. Tawbi, K. Thai. *Towards an Architecture for Distributed Multimedia Application Support.* In Proceedings of the International Conference on Multimedia Computing and Systems, Boston, USA, May 1994.

[7] A. Birrell and B. Nelson. *Implementing Remote Procedure Calls.* ACM Transactions on Computer Systems, 27(4), pages 349-350, April 1984.

[8] G. S. Blair, N. Davies, A. Friday and S. P. Wade. *Quality of service support in mobile environments: an approach based on tuple spaces.* In Proceedings of the 5th IFIP International Workshop on Quality of Service, New York, USA, May 1997.

[9] G. S. Blair, G. Coulson, N. Davies, P. Robin, T. Fitzpatrick. *Adaptive Middleware for Mobile Multimedia Applications.* In Proceedings of the 7th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV '97), St. Louis, MI, USA, May 1997.

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

[11] A. Campbell, G. Coulson, D. Hutchison. *A Quality of Service Architecture.* Computer Communication Review, 1(2), pages 6-27, April 1994.

[12] L. Cardelli. *Foundations for Wide-Area Systems.*Paolo Ciancarini, Alessandro Fantechi and Roberto Gorrieri, Editors. Formal Methods for Open Object-Based

Distributed Systems, IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS), February 15-18, 1999, Florence, Italy. pages 349-349, Kluwer Academic Publishers, 1999.

[13] W. H. Cheung, A. H. Loong. *Exploring Issues of Operating Systems Structuring: from Microkernel to Extensible Systems.* Operating Systems Review, 29(4), pages 4-16, October 1995.

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

[15] J. S. Crane. *Dynamic Binding for Distributed Systems.* PhD thesis, University of London, Department of Computing, Imperial College of Science, Technology and Medicine, 180 Queen's Gate, London SW7 2BZ, UK, 1997.

[16] N. Davies, G. S. Blair, K. Cheverst and A. Friday. *Supporting Adaptive Services in a Heterogeneous Mobile Environment.* In Proceedings of the 1st Workshop on Mobile Computing Systems and Applications, Santa Cruz, CA, USA, December 1994.

[17] 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

[18] S. M. Dorward, R. Pike, D. L. Presotto, D. M. Ritchie, H. W. Trickey, P. Winterbottom. *The Inferno Operating System.* Bell Labs Technical Journal, 2(1), pages 5-18, Winter 1997.

[19] D. R. Engler, M. F. Kaashoek, J. W. O'Toole 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.

[20] T. Fitzpatrick, G. S. Blair, G. Coulson, N. Davies, P. Robin. *Supporting Adaptive Multimedia Applications through Open Bindings.* In Proceedings of the 4th International Conference on Configurable Distributed Systems, pages 128-135, Annapolis, Maryland, USA, May 1998.

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

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

[23] A. S. Grimshaw, W. A. Wulf. *The Legion Vision of a Worldwide Virtual Computer.* Communications of the ACM, 40(1), January 1997.

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

[25] D. B. Hehmann, R. G. Herrtwich, W. Schultz, T. Schuett, R. Steinmetz. *Implementing HeiTS: Architecture and Implementation strategy of the Heidelberg High Speed Transport System.* In Proceedings of the Second International Workshop on Network and Operating System Support for Digital Audio and Video, IBM ENC, Heidelberg, Germany, 1991.

[26] M. Henzinger, V. King. *Fully Dynamic 2-edge Connectivity Algorithm in Polylogaritmic Time per Operation.* Technical Note 1997-004, Digital Equipment Corporation, Systems Research Center, Palo Alto, CA, USA, June 1997.

[27] D. Hildebrand. *QNX: Microkernel Technology for Open Systems Handheld Computing.* The Pen & Portable Computing Conference and Exposition, Boston, USA, May 1994.

[28] M. A. Hiltunen, R. D. Schlichting. *Fine-Grain QoS customization in Distributed Middleware Services.* Department of Computer Science, University of Arizona, Tuscon, AZ, USA. Electronic document available at ftp://ftp.cs.arizona.edu/ftol/papers/iwqos.ps

[29] N. Hutchinson, L. Peterson. *The x-kernel: An Architecture for Implementing Network Protocols.* IEEE Transactions on Software Engineering, 17(1), pages 64-76, January 1991.

[30] V. Issarny, C. Bidan, T. Saridakis. *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.

[31] M. F. Kaashoek, D. R. Engler, G. R. Ganger. *Application Performance and Flexibility on Exokernel Systems.* In Proceedings of the 16th ACM Symposium on Operating Systems Principles, pages 52-65, Saint-Malo, France, October 1997.

[32] P. Kostkova, K. Murray, T. Wilkinson. *Component Based Operating System.* Second Symposium on Operating Systems Design and Implementation, WIP session, Seattle, USA, October 1996. Electronic document available at http://www.usenix.org/publications/library/proceedings/osdi96/wip.html

[33] P. Kostkova, J. S. Crane, J. A. McCann, T. Wilkinson. *MAGNET: QoS-based Dynamic Adaptation in a Changing Environment.* Internal TR, HiPeX, Department of Computer Science, City University, London, UK, February 1998. Electronic document available at ftp://ftp.cs.city.ac.uk/users/patty/qos-abstract.html

[34] P. Kostkova, T. Wilkinson: *MAGNET: A Virtual Shared Tuplespace Resource Manager.* International Journal on Parallel and Distributed Computing, Special Issue on Parallel and Distributed Computing Practices, Ed. M. Paprzycki, NOVA Science Books, Commack, New York, 1(3), September 1998.

[35] P. Kostkova, J. S. Crane, T. Wilkinson: *MAGNET: A Dynamic Information Broker for Mobile Environments.* Internal TR, HiPeX, Department of Computer Science, City University, London, UK, March 1998. Electronic document available at ftp://ftp.cs.city.ac.uk/users/patty/broker-abstract.html

[36] J. Kramer, J. Magee, A. Young. *Towards Unifying Fault and Change Management.* In Proceedings of IEEE International Workshop on Distributed Computing Systems in the 90', pages 57-63, Cairo, Egypt, 1990.

[37] B. W. Lampson. *Designing a Global Name Service.* In Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing (PODC 86'), Calgary, Canada, August 1986. ACM New York, pages 1-10, 1986.

[38] A. A. Lazar. *Challenges in Multimedia Networking.* In Proceedings of the International Hi-Tech Forum, Osaka, Japan, February 1994.

[39] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, E. Hyden. *The Design and Implementation of an Operating System to Support Distributed Multimedia Applications.* University of Cambridge, Computer Laboratory. Cambridge, UK, June 1997.

[40] J. Liedtke. *On micro-kernel construction.* In Proceedings of the 15th ACM Symposium on Operating Systems Principles, Copper Mountain Resort, Colorado, USA, pages 237-250, December 1995.

[41] M. J. Litzkow, M. Livny, M. W. Mutka. *Condor — A Hunter for Idle Workstations.* In Proceedings of the 8th International Conference on Distributed Computing systems, pages 104-111, 1998.

[42] J. Magee, J. Kramer, M. Sloman, N. Dulay. *A Constructive Development Environment for Parallel and Distributed Programs.* Distributed Systems Engineering Journal, Special Issue on Configurable Distributed Systems,1(5), pages 304-312, September 1994.

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

[44] K. R. Mayes, J. Bridgland. *Arena — a Run-Time Operating System for Parallel Applications.* In Proceedings of Euromicro '97, Workshop on Parallel and Distributed Processing, pages 253-258, 1997.

[45] J. A. McCann, P. Kostkova. *Advances in Operating Systems and their Implications for DBMS.* Internal TR, HiPeX, Department of Computer Science, City University, London, UK, August 1997. Electronic document available at ftp://ftp.cs.city.ac.uk/users/patty/dbms-abstract.html

[46] J. A. McCann, J. S. Crane. *Kendra: Internet Distribution & Delivery System — an introductory paper.* In Proceedings of SCS EuroMedia Conference, Leicester, UK. Ed. Verbraeck A., Al-Akaidi M., Society for Computer Simulation International, pages 134-140, January 1998.

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

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

[49] D. Mosberger, L. L. Peterson. *Making Paths Explicit in the Scout Operating System.* In Proceedings of the Second Symposium on Operating Systems Design and Implementation, Seattle, USA, pages 153-167, October 1996.

[50] K. Nahrstedt. *Middleware Support for Quality of Service Support.* In Proceedings of the Grace Hopper Celebration for Women in Computing, San Jose, USA, pages 65-68, September 1997.

[51] K. Nahrstedt, J. Smith. *The QoS Broker.* IEEE Multimedia, 2(1), pages 53-67, Spring 1995.

[52] K. Nahrstedt, J. Smith. *Design, Implementation and Experience of the OMEGA End-Point Architecture.* IEEE Journal on Selected Areas in Communications, 14(7), pages 1263-1279, September 1996.

[53] K. Nahrstedt, R. Steinmetz. *Resource Management in Networked Multimedia Systems.* IEEE Computer 28(5), pages 52-64, May 1995.

[54] G. C. Necula, P. Lee. *Safe Kernel Extensions Without Run-Time Checking.* In Proceedings of the second Symposium on Operating Systems Design and Implementation, Seattle, USA, pages 229-243, October 1996.

[55] G. Nilisoin, F. Dupuy, Chapman. *An Overview of the Telecommunications Information Networking Architecture.* In Proceedings of TINA 95, Melbourne, Australia, February 1995.

[56] 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.

[57] B. Oki, M. Pfluegl, A. Siegel, D. Skeen. *The Information Bus.* In Proceedings of the 14th ACM Symposium on Operating Systems Principles (SIGOPS '93), Asheville, North Carolina, USA, pages 58-68, December 1993.

[58] A. Oliva, L. E. Buzato. *The Design and Implementation of Guarana'*. Submitted to the 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS '99), San Diego, USA, May 1999.

[59] P. Pardyak, B. Bershad. *Dynamic Binding for an Extensible System.* In Proceedings of the second Symposium on Operating Systems Design and Implementation, Seattle, USA, pages 201-212, October 1996.

[60] R. Pike, D. L. Presto, S. M. Doward, B. Flandrena, K. Thompson, H. W. Trickey, P. Winterbottom. *Plan 9 from Bell Labs.* Journal of Computing Systems, 8(3), pages 221-254, Summer 1995.

[61] B. Potter, J. Sinclair, D. Till. *An Introduction to Formal Specification and Z.* Prentice Hall International, second edition, 1996.

[62] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, K. Zhang. *Optimistic Incremental Specialization: Streamlining a Commercial Operating System.* In Proceedings of the 15th ACM Symposium on Operating Systems Principles, pages 314-324, Colorado, USA, December 1995.

[63] R. Raman, M. Livny, M. Solomon. *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.

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

[65] S. Savage, B. Bershad. *Some Issues in the Design of an Extensible Operating System.* In Proceedings of the First Symposium on Operating Systems Design and Implementation, Panel session, Monterey, California, USA, page 196, November 1994.

[66] Secretariat: ISO/IEC JTC1/SC33. Standards Association of Australia, PO Box 1055, Strathfield, NSW, Australia 2135. *Open Distributed Processing — Interface References and Binding.* January 1998. Document ISO/IEC JTC1/SC33 N119, ITU-T Draft Recommendation X.930 (1998).

[67] Secretariat: ISO/IEC JTC1/SC21. Standards Association of Australia, PO Box 1055, Strathfield, NSW, Australia 2135. *Information technology — Open Distributed Processing — Trading Function.* June 1995. Document ITU-T Rec.9tr ISO/IEC JTC1/SC21 DIS 13235.

[68] Secretariat: ISO/IEC JTC1/SC *Working Draft for Open Distributed Processing — Reference Model — Quality of Service.* January 1998. Document ISO/IEC JTC1/SC21 N10979 Ed 6.4.

[69] R. Staehli, J. Walpole, D. Maier. *Quality of Service Specification for Multimedia Presentations.* Multimedia Systems, 3(5/6), November 1995.

[70] H. Storner. Linux kerneld mini-HOWTO. Electronic document available at http://www.image.dk/~storner/kerneld-mini-HOWTO.html, version 1.7, July 19, 1997.

[71] M. I. Seltzer, Y. Endo, C. Small, K. A. Smith. *Dealing With Disaster: Surviving Misbehaved Kernel Extensions.* In Proceedings of the second Symposium on Operating Systems Design and Implementation, Seattle, USA, pages 213-227, October 1996.

[72] A. S. Tanenbaum. *Distributed Operating Systems.* Prentice Hall International, Inc. 1995.

[73] A. Veitch, N. Hutchinson. *Kea — A Dynamically Extensible and Configurable Operating System Kernel.* In Proceedings of the Third International Conference on Configurable Distributed Systems, pages 236-242, Annapolis, Maryland, USA, May 1996.

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

[75] J. Waldo. *Jini Architecture Overview.* Electronic document available at http://www.javasoft.com/products/jini/whitepapers/architectureoverview.pdf, Sun Microsystems, Inc., 1998.

[76] 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.

[77] Niklaus Wirth. *Algorithm + Data Structures = Programs.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1975.