MAGNET: An Architecture for Dynamic Resource Allocation

Patty Kostkova and Julie A. McCann
High-Performance Extensible Research Group (HiPeX)
City University, London, UK
{patty, jam}@soi.city.ac.uk

Abstract

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 adaptability and reconfigurability.

However, mobile and roaming users need to dynamically adapt to local system configurations to order to fully utilize resources currently available, such as a fast network connection, an available colour printer etc. In order to pro- vide support for this type of application, a dynamic resource manager supporting indirect resource requests and runtime reconfigurability is essential.

This paper proposes a new dynamic resource manage- ment architecture, MAGNET, to provide run-time adaptabil- ity for mobile applications. MAGNET enables the dynamic trading of resources which can be requested indirectly by the type of service they offer, rather than directly by their name. In addition, MAGNET enables runtime user-customized adaptation to services.

1 Introduction

In order to meet the requirements of mobile and roaming users, the role of resource management needs to be extended to enable user customization of resource allocation policies and to provide the support for runtime adaptation to changing conditions.

In open systems, there is a requirement to enable re- quests for services to be described by a type of service (e.g., a printer) , rather than directly by its name. Without this, communication between system components which did not know their identity a priori, cannot happen. In addition, dy- namic features such as the monitoring of selected resource features, and the provision of location and time-dependent information are also required.

Existing operating systems, and middleware platforms dealing with dynamic resource allocation do not provide sufficient support for mobile applications in terms of user-customization of the allocation strategies and their runtime adjustment.

This paper addresses the design of a resource manager, MAGNET, fulfilling the requirements of applications operating in frequently-changing environments. In particular, we present a framework enabling user-customized dynamic resource allocation supporting runtime adaptation.

2 The MAGNET Architecture

A resource manager which can provide dynamic resource allocation requires the following features: dynamic trading , extensibility, dynamic rebinding, resource monitoring and reconfigurability. These features and the high-level design of the MAGNET architecture are discussed in this section.

The key component of the framework is a *Trader* that collects information on services, and dynamically matches requests against demands (by type of service and not the name given to the service, e.g., printer and not 'LPR2') . In doing this, it can establish dynamic binding between components which did not need to know their identity in advance. The Trader must not constrain the format or the seman- tics of information on services and should allow the user to customize the matching process between servers' offers and applications' requests. This provides *extensibility* in terms of enabling new requests and services to be dynamically gen- erated but also in terms of customizing the matching process itself.

Further, to support runtime adaptation and system reconfiguration *dynamic rebinding* is required. That is, the old binding is dropped and a new binding is established in order to better meet application requirements. This may be as a result of client, server or a third party initiation .

For example, a mobile client currently using its lor· .l disk may wish to join a new, more stable environment in ii:, office to upload data. Therefore it will unbind from its current disk and rebind to the office disk. Information on client demands and service capabilities is maintained either manually (i.e., carried out by the components themselves) or automatically (by a monitoring process).

2.1 Using the Tuplespace Paradigm for the Trader

The Trader is the key component in the MAGNET architecture. The Trader accesses a shared data repository available to all components. We call this data structure an *information pool,* its structure is similar to the t uplespace [1] The Trader consists of three distinctive elements:

[1] T he information pool is actually a tuplespace. However, the term 'tuplespace' is often associated with the Linda distributed program-

MobiDE Seattle WA USA

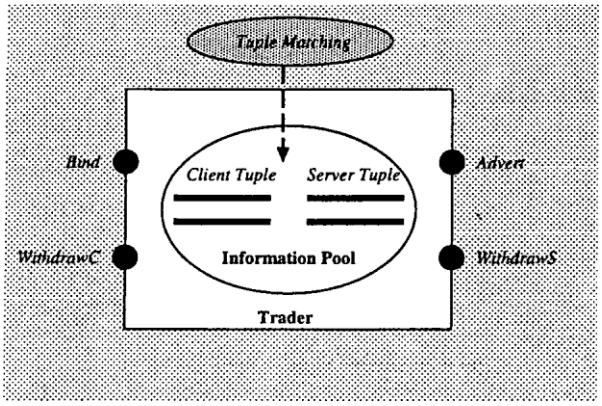

Figure 1: The Trader Structure

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

Figure 1 illustrates the structure of the Trader, and its three components. Figures are drawn in the Darwin graph- ical representation [2]. Darwin views components in terms of both services they provide (to allow other components to interact with them) and services they require (to interact with other components). A provided service is represented by a filled circle and a required service by an empty circle. Components are shown as rectangles, filleting differentiates types from instances.

2.1.1 The Information Pool and the Matching Function

The information pool is a distributed data structure accessible by all components using MAGNET. Tuples can be inserted in, or withdrawn from, the tuplespace by a set of clearly defined operations. Tuples describing requirements and provisions for resource management often contain additional information, such as interface references for accessing offered services, or requirements on the establishment of the communication channel. These are all expressed as tuple elements.

Therefore, the tuple distinguishes between the number of all tuple elements *n* and the number of matching elements m. This extension, which we have incorporated into traditional tuple matching, enables the restriction of the matching process to matching the first *m* elements. By *matching* we mean an equality of tuple elements, or a user-defined 'match' enabling quality of service (QoS) to be taken into account. (However, QoS is beyond the scope of this paper further details can be found in [7].) We define a tuple follows:

A tuple Tis anordered (n+2)-tuple $T = (n, m, p1, p2, \ldots, Pn)$, $n > m$ where $n \in N$ represents the number of tuple elements and $m \in N$ is the number of matchable tuple el- ements . *p;* E *P;* are the values of tuple elements i.e., the actual parameters.

ming language [5}; therefore, we decided to call our data structure 'information pool' to a,·oid con fusion .

For example, to describe a Pentium processor (A) which is running at 200 MHz with 32 MB RAM memory we use the following server tuple:

A= (6, 5, *CPU, Pentium,* 200, *memory,*32, *ref)*

That is 6 tuple elements, 5 of which can be matched (CPU, Pentium, 200, memory, 32). Ref is an interface refer- enc: to the service inter ace d scribed in the tuple. Naming for mterface references 1s derived from the naming scheme used in the computing environment.

An equivalent client tuple matching the tuple A would be:

B = (6, 5, *CPU, Pentium,•, memory,*16 - 128, *ref B)*

. requesting a processor Pentium with any speed, equipped with a memory of the size between 16 to 128 MB. This tuple definition incorporates advanced operators, such as * and '-'. These are defined in detail in [7].

2.1.2 The Trader Operations

The information pool has operations defined for tuple manipulation, such as insert and delete. MAGNET's Trader in- c udes the operations: BIND, ADVERT (implementing a ser- 1ce expor), an WITHDRAWC, WITHDRAWS (implement- mg a service withdrawal). These are described below in more detail.

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

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

Op ration **ithdrawC (T),** where Tis a client-tuple , results m removing tuple T from the information pool· while opera ion **Wi hdrawS (T),** where T is a server-tu;le, re- sults m removmg tuple T from the information pool.

2.1.3 Components for the MAGNET Architecture

MAGNET'S information pool has been designed to scale by using *federations* . Figure 2 illustrates the structure of the MAGNET architecture distributed within 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 per federation, in contrast to multiple instances of Client, Server and Tree. The federation communication is done through the federation 's Trader. In addition there are two types of subcomponent performing dedicated func- tions: these are a pair of *Binders* (the *Cli,;nt-Binder* and the *Server-Binder)* present in all Clients and Servers; and the *GlueFact_ory* included in all Trees. The GlueFactory hands over a client tuple to the Server to initialize the establish- ment of the binding carried out by the Binders. There fore , Binders in cooperation with the GlueFactory , establish the resultant client-server binding.

2.1.4 Information Monitoring

Service definitions placed in the Trader must be kept up-to-date by monitoring the resources. Consequently two addi-

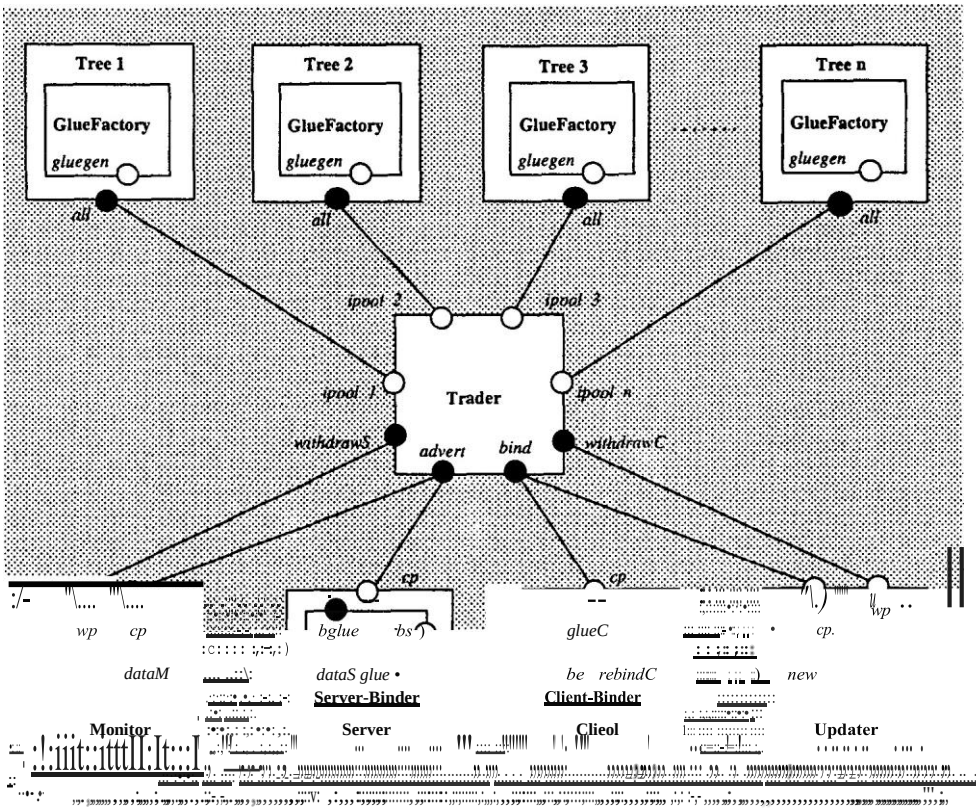

Figure 2: The MAGNET's Architecture

tional components provide monitoring : the *Monitor* (monitoring server provisions), and the *Updater* (monitoring changing client requirements).

3 Support for Adaptation

In order to present the support for adaptation provided by MAGNET, we illustrate relevant operations with an example. In this example we have a mobile client which has MAGNET on it allocating its local resources. Likewise, there is an office system which also runs MAGN ET. Essentia lly, there are two federations : the mobile and the office-b ased. To allow the mobile client to use the office-based system resources, the mobile client's Trader must communicate with the office-based Trader. This is illustrated in more detail below.

3.1 Dynamically Reconfigurable Federations

When a mobile user wishes to dock into a particular environment, the local and office-based federations must communicate to enable the user to access remote resources. MAGNET provides this support by the operations JOIN and LEAVE. \Ve assume the mobile user is only accessing those resources temporarily thus:

- operations JOIN and LEA\'E *cannot be transparent.* That is, the mobile client can decide whether it is able to accept a resource from an office-based site. (This avoids situations such as disk space being allocated in the office-based computer which is useless **when** the client is disconnected).
- consequently, clients and servers *do not act symmetrically* - mobile clients might take advantage of the office-based services but will not offer their provisions to the office-based clients.
- as client components are responsible for deciding on the usage of office-based resources, they are also re- sponsible for *maintaining a consistent state* when the mobile is disconnected from the office-based site . Con- sequently, every mobile client component using the office-based Trader, is responsible for withdrawing all inserted tuples in order to leave the office-based infor- mation pool in a consistent state .

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, as was described at the beginning of this section.

Trader Connection In order to perform the operation JOIN, the office-based Trader offers its information pool to the portable Trader by calling the operation ADVERT inserting a tuple *Tl=* (2, *1,join,bglue)* into the mobile client's information pool.

JOIN A mobile client requesting a service which can be fulfilled by the office-based site servers, inserts three bind tuples into the mobile client's pool - the classical bind tuple Cl defining the request, the second tuple of the form: C2 = *(n,* 1, *join,* Cl) encapsulating the actual Cl tuple , and the third C3 = (n , 1, *le ave* ,Cl) which will be used for disconnection .

The mobile client is connected to the office-based federation by inserting the office-based Trader tuple Tl = (2, 1, *join, bglue)* into the mobile client's information pool where a matching between Tl and C2 can be achieved. That is, the GlueFactory binds to the office-based Trader and passes the C2 tuple to the office-based Trader-Binder. This then retrieves the tuple Cl from the received tuple C2 and reinserts it into its information pool by calling operation BIND. If a matching server-tuple is available inter-federation binding is established.

Figure 3 illustrates operation JOIN, for reasons of simplicity, we omit the Tree components.

Trader Disconnection Disconnecting the mobile client from the office-based site must return the system to a consis- tent state. Firstly, the office-based Trader's tuple Tl is withdrawn from the client's information pool by the opera- tion WITHDRAWS. Client tuples waiting to be served in the office-based information pool must be also removed. The client components themselves must perform the withdrawal as Traders cannot distinguish between the office-based client-tuples and mobile cl nt-tuples. This is done by inserting a local Trader advert-tuple into the mobile client's information pool: *T2* = (2, 1, *leave,bglue2).*

Operation LEAVE The inserted T2 tuple matches with a waiting client tuple *C3* = *(n,* 1, *leave,*Cl) which is passed to the office-based Trader-Binder over an binding established between a particular Glue Factory and the **bglue2** service interface. Tuple Cl is extracted from C3, and operation WITHDRAWC on Cl 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 be obtained from the tuple Cl) and notifies them. Incidently, bs and be are service interfaces to Client Binder (be) and Server Binder (bs) see Figure 4. Full descriptions of service interfaces be, bs, glues etc. are beyond the scope of this paper, it can be found in [7].

Figure 4 illustrates this operation. There are two clients (Client! and Client2) temporarily connected to an office- based site. Client! (described by tuples Cl, C2 and C3) is being served by an office-based server (Server), while Client2 (described by tuples Xl, X2 and X3) is still waiting. The no- tification about disconnection from the office-based Trader results in different actions: Clientl must terminate its bind- ing with Server, while Client2 just reinserts its 'connecting' tuples X2 and X3.

4 Implementation

MAGNET, has been implemented in Regis [2], an environment for constructing distributed systems. The tuple is implemented in C++ as a high-level base-class (Tuple) comprising the tuple size, the tuple matching size, and encapsulating the tuple-elements. All standard and user-defined tuple-element classes are inherited from a base tuple-element class TElm.

Trees contain tree data structures supporting the search (and matching) for non-parametrized requests. The complexity of the Trader operations was calculated and was found to be linear to the number of tuple matching elements. The Trader is responsible for the efficient distribution of tree data structures over Tree components.

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

As the focus of the architecture is to provide dynamic features, such as runtime adaptability, user-customization and flexibility, the implementation results cannot be described in terms of performance. However, critical analysis of various features of the framework can be found in [7].

5 Related Work

In recent years, dynamic issues such as providing greater flexibility, supporting dynamic runtime adaptations or designing loosely coupled communication schemes have been successfully addressed by many research projects and commercial technologies.

Dynamic service coupling is implemented in other architectures (such as the Matchmaker Component in Match- making [10] and the Aster Selector in Aster [6]) and in tuplespace-based systems. These include: Limbo [3] based on notion of multiple tuplespaces and an explicit tuple-type hierarchy, Jini [12] providing binding between clients and servers by a lookup service. However, these systems do not provide customization of the matching function. Further, Limbo and Cardelli's Ambient Calculus model [1] reference systems components by type and not service characteristics which was our goal. These architectures typically target one mobile issue, rather than providing a unified architecture. Nevertheless, MAGNET can be used with other component- based systems like Exokernel [4] and Nemesis [11]. It can also trade objects in user-level applications, for example Jav- aBeans, CORBA objects [9), and DCOM objects [8].

6 Implications of our Assumption and Future Work

The architecture described in this paper is a general framework which provides more functionality (such as Quality of Service definition and matching) than described here. Full description of the architecture can be found in [7]. The architecture was originally designed for operating system like resource allocation, therefore a number of assumptions were made:

1. **Consistency.** Applications are assumed to be well behaved and fully responsible for maintaining system consistency that includes keeping the tuples in the pool up-to-date and withdrawing all tuples when the component is finished. MAGNET does not maintain consistency nor does it carry out garbage collection. This is a feature which would aid both consistency and protection and may be added later.

2. **Protection.** Validity of tuples within the informa- tion pool is the responsibility of the components and not MAGNET. To prevent actions like binding be- tween non-exiting components, the framework can be extended to authorize components to call the Trader functions, or introduce capabilities (as tuple elements) to improve component protection.

3. **Synchronization.** Components are responsible for synchronization. The architecture can provide an additional function for client components (e.g., operation BINDRET) which would perform the same matching process as operation BIND, and return 'no' instead of

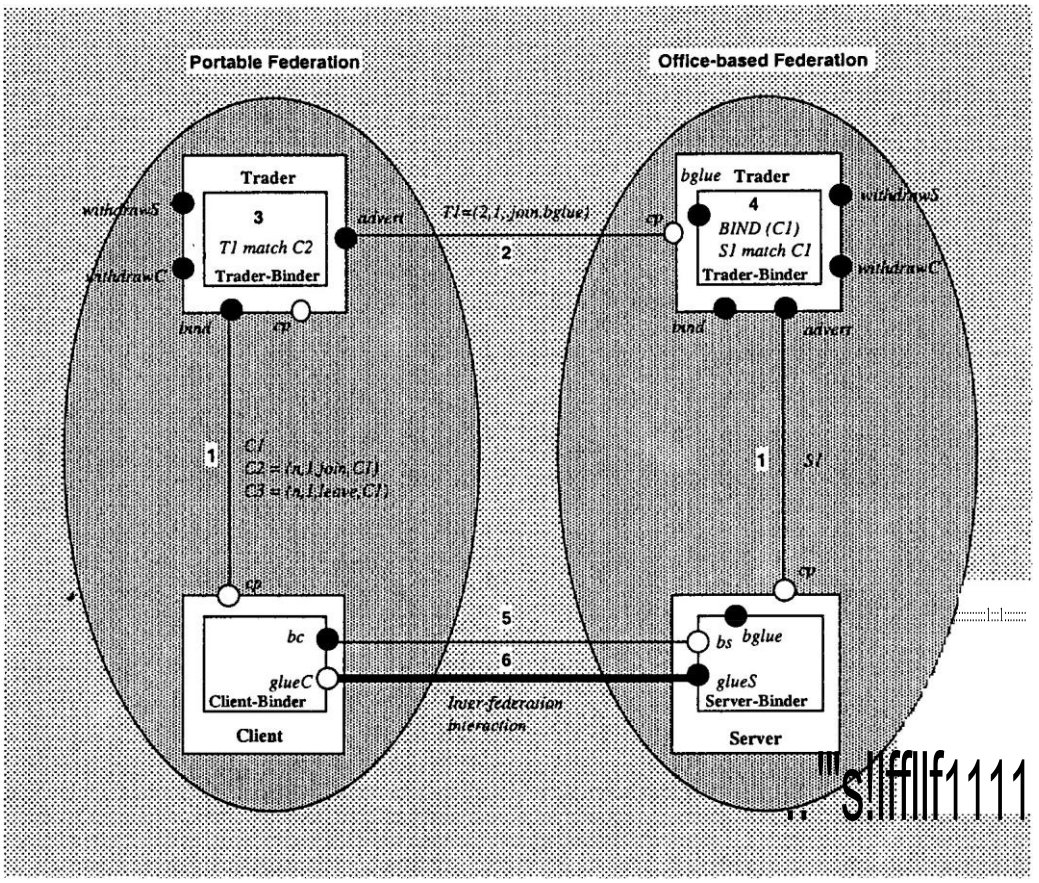

Figure 3: Operation JOIN

blocking the component if the requested service has not been found in the pool.

4. **Scale and Performance.** The estimated numbers of components in a federation are in the region of tens and they have the potential to generate tens to hun- dreds tuples placed in the Trader. Likewise, the num- ber of concurrent components accessing the Trader at one time are estimated to be in the region of tens. A higher number of components can result in the Trader becoming a bottleneck. A possible solution would be to implement the information pool in distributed shared memory.

5. **Change Frequency.** 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 a frequency of seconds and milliseconds) would not be viable. This can be improved by enabling direct access to the Tree components for trusted Monitors and Updaters.

7 Conclusion

New resource managers are required to allow dynamic resource allocation for mobile systems. The tuplespace-based MAGNET architecture, described in this paper, provides this kind of support. In particular, we focussed on MAGNET's support for dynamic trading, extensibility, dynamic rebinding, resource monitoring and reconfigurability. Our research on MAGNET has demonstrated the feasibility of dynamic resource management which further provides negotiation of services, user-customization of allocation strategies, and runtime adaptation to changes in the computing environment.

References

[1] 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.l 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.

[2] 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.

[3) G. S. Blair, G. Coulson, N. Davies, P. Robin, T. Fitz- patrick. *Adaptive Middleware for Mobile Multimedia Ap- plications.* In Proceedings of the 7th International Work- shop on Network and Operating System Support for Digi-

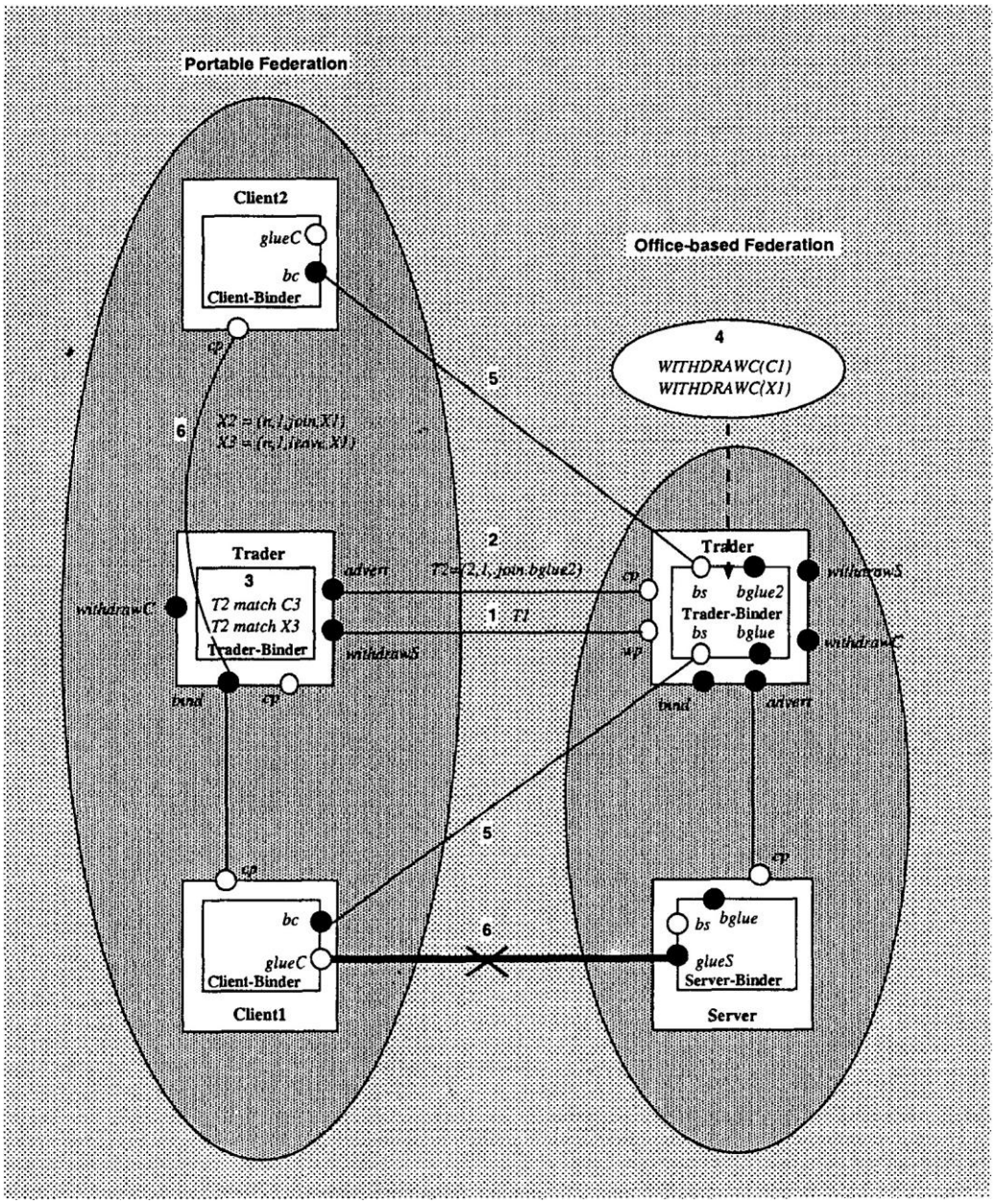

Figure 4: Operation LEAVE

ta! Audio and Video (NOSSDAV '97), St. Louis, Ml, USA, May 1997.

[4) D. R. Engler, M. F. Kaashoek, J. W. O'Toole Jr. *Exok- emel: 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.

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

[6] 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.

[7) P. Kostkova. *MAGNET: Dynamic Resource Manage- ment Architecture.* PhD Thesis. Department of Comput- ing, City University, London. March 1999.

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

[9) 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.

[10) 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 Comput- ing, Chicago, IL, USA, July 1998.

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

[12) J. Waldo. *Jini Architecture Overview.* Electronic document available at http://www.javasoft.com/products/jini/ whitepapers/architectureoverview.pdf, Sun Microsystems, Inc., 1998.