Service Migration and a Transparency Service

B.Schulze, E.R.M.Madeira, F.Assis Silva
S.Choy, I.Busse, S.Covaci

Relatório Técnico IC - 98 - 28

Agosto de 1998

Service Migration and a Transparency Service

B.Schulze[1,2][schulze@dcc.unicamp.br], E.R.M.Madeira[1]
F.Assis Silva[3], S.Choy[4]
I.Busse[5], S.Covaci[5]

1. IC / UNICAMP, PO Box 6176, 13083-970, Campinas, SP, Brazil
2. CBPF / CNPq, R.Xavier Sigaud 150, 22290-180, Rio de Janeiro, Brazil
3. TU Berlin/FOKUS, Kaiserin-Augusta-Allee 31, D-10589 Berlin, Germany
4. IKV++, Kurfürstendamm 173-174 , D-10707 Berlin, Germany
5. GMD FOKUS, Kaiserin-Augusta-Allee 31, D-10589 Berlin, Germany

Abstract. Location and migration distribution transparencies are considered in the context of multi agent systems. The extension of a mobile agent facility architecture is proposed with the addition of an availability service and a transparency service. Transparent mobility of services is explored in applications using overall load-balancing of resources. The search for a new destination agencies follows a transparent location and selection of available resources. Transparency is trimmed by a parameter defining the goal of the application.

Keywords. distribution transparencies, mobility, availability, trading, agent facility.

1. Introduction

We concentrate on architectural aspects in the context of Distributed Object Computing (DOC). Service agents in a service-oriented architecture based on OMG/CORBA are related to some distribution transparency [RM-ODP] aspects, for instance, location and migration under the usage of a mobility service, an availability service and a transparency service. An *availability* service was previously introduced and an additional *transparency* interface is proposed in order to allow an easy interfacing to availability.

The availability service is intended for transparent location and selection of available resources. The relationships to a *Trader* service and other CORBA object services are also mentioned. The proposed availability service works closely related to a mobility service in order to transparently migrate services. Transparency is trimmed by a parameter defining the goal of the application. The approach should be specially suited for applications using load-balancing and inverse caching, i.e., code being moved close to data. The goal is to sustain the functionality of a DOC environment by migrating components over a CORBA based middleware.

The functionality of an availability service should reflect categories of applications. The approach is to add a parameter to the availability interface representing the goal for transparency, like for instance: performance, fault-tolerance, security, multimedia, and so on. Each goal may impose a slight different strategy to the availability processing and final selection. For instance in load-balancing, the availability service has to be lightweight and quick. For this reason supportive testing is based on some intrinsic evaluation of load, based on the response time of the hosts attending each kind of service.

An additional parameter should be present in the above interface to define a level of recursion in the process of recovering / replacing services, i.e., the depth of the search process allowed to step

into the services and the respective resource allocation. A sketch of the services and the recursion process is presented in Figure 1. Normally, there would be some default value for this parameter but of course it has to accommodate other values either in a flexible way reflecting the environment or interactively with the application/user. A better alternative would be to submit a value to the application/user in order to be allowed and trusted. Trusted values go into a repository for autonomous decisions.

Some simulation results [Schulze97] point out that the mean execution time of a test application could be sustained with a migration strategy based on transparent resource allocation, specially with a scale-up in the number of authorized hosts. The inclusion of a QoS range should help in the selection. A side contribution is that component migration and distribution are handled the same way.

No major point is made on the cost of replication since it consists only of a new registration of the service object interface. The registration of a service is based on a proxy like, containing information on activation mode and pointing to a general implementation repository where to find state and code classes. The implementation classes might be local or on a remote URL address [Visibroker97]. It should be remembered that the final activation (and garbage collection) of the service objects are handled by a *daemon* which is part of the CORBA environment. Any major cost is at the activation level when code is bootstraped, which happens at a *bind* or *omg.org.object_to_string* call level.

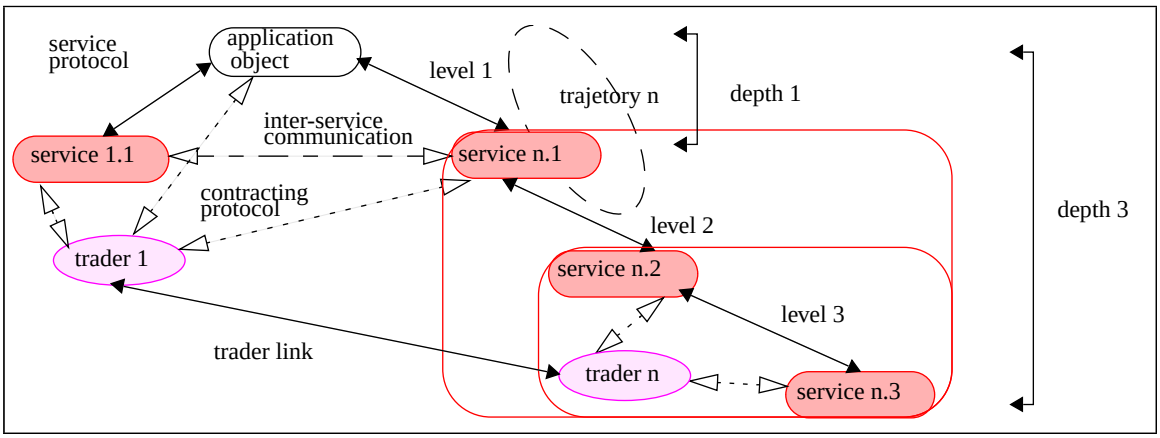

Figure 1. Recursion levels in a service-oriented architecture.

Extensions to existing DOC environments are based on a minimum of add-on by essentially rearranging pieces and the supportive approach of some simulation and testing. The basic approach is to extend existing concepts using insights from different methodologies and verifying (in)consistencies. In this sense, the presented results / implementation are not necessarily conclusive but rather supportive to the approach.

Migration is a technique well mentioned in the literature but in a world of agents and multi-agent systems, it gets some new approach and insight, specially under agent's mobility [Cardozo93, Iglesias96, Krause96, Lange95, Mendes96, Russel95, Schulze97]. An explicit approach for mobility transparency is not explored in the majority of related works. Well reported techniques may show interesting results when translated into the object-oriented model.

Section 2 discusses architectural aspects. Section 3 presents availability and Section 4 introduces transparency. Section 5 discusses service location. Section 6 resumes supportive testing. Section 7

presents related remarks work and Section 8 presents concluding remarks.

2. Architecture

A service-oriented architecture based on the OMG/CORBA model and the relationship between the added object services and the main CORBAservices involved, are presented in Figure 2 and 3 respectively.

Agency Object Model. Figure 3, shows an object model and relationship of the availability services to the mobility service and to other CORBAservices. CORBAservices [COS97] are here just represented in the corresponding box as a whole not detailed. Specific CORBAservices like: Life Cycle, Persistence and Relationships, Query, Properties and Trader are explicitly represented as well as their association to the availability and mobility services. Some custom implementation of the CORBAservices are considered rather then the full implementation. The availability service uses *querying* in the local domain and *trading* if querying does not succeed. The properties service maintains information on relevant service *properties*. It is interesting in both, querying and trading, to support *dynamic* properties [TOS96].

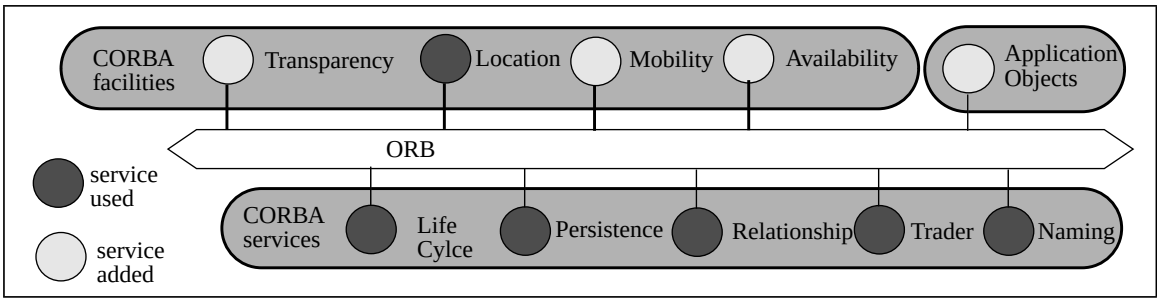

Figure 2. Adding services under CORBAfacilities.

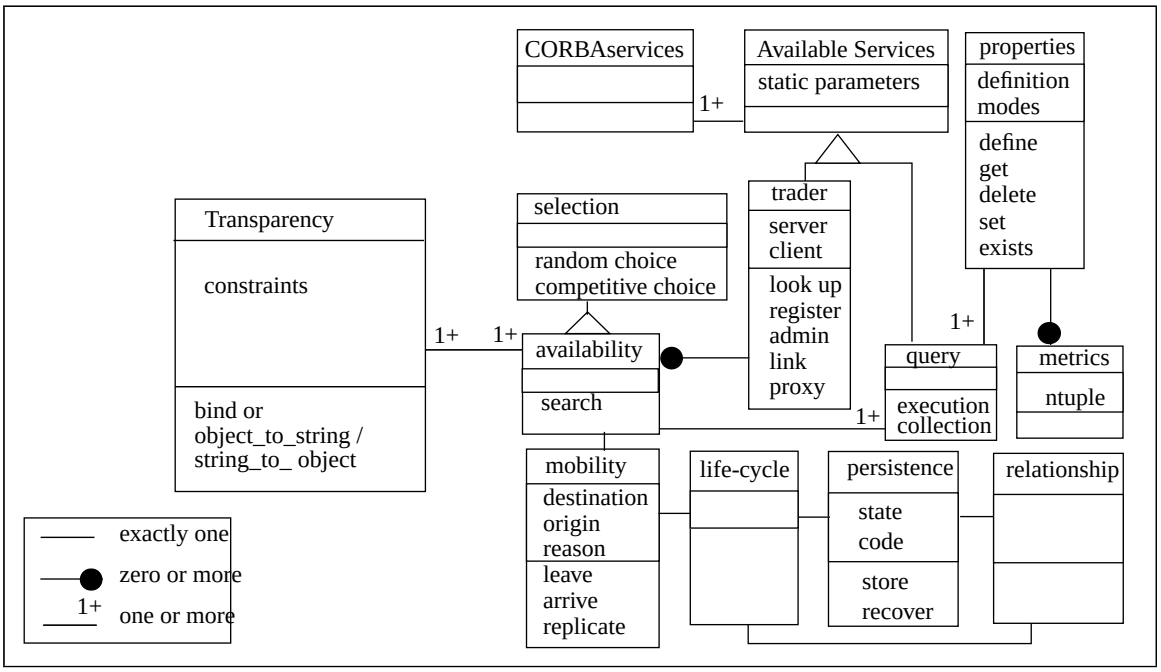

Figure 3. Object model using OMT conventions.

2.1. Transient and Persistent Agents

The transient and persistent agents (objects) are shortly discussed in the presence of an activation daemon.

Transient objects (agents) have the same lifetime as the execution of the application that creates them. By contrast, persistent objects (agents) live until they are explicitly deactivated. When the application terminates, its transient objects (agents) disappear with it and any object reference held by a client becomes invalid. If a client has a reference to a persistent object (agent), that reference can be used even if the object's (agent's) server is not running. For persistent agents, the agency activation daemon will activate the server when the agency receives an invocation on the object (agent).

Using the CORBA model, an agency can be seen as an ORB with an activation daemon and agents, mobile or not, being registered as CORBA agents on a portable object adaptor (POA) [OMG97], as shown in Figure 4. When the agent is activated, the persistent state of the agent informs if it has to run or just wait.

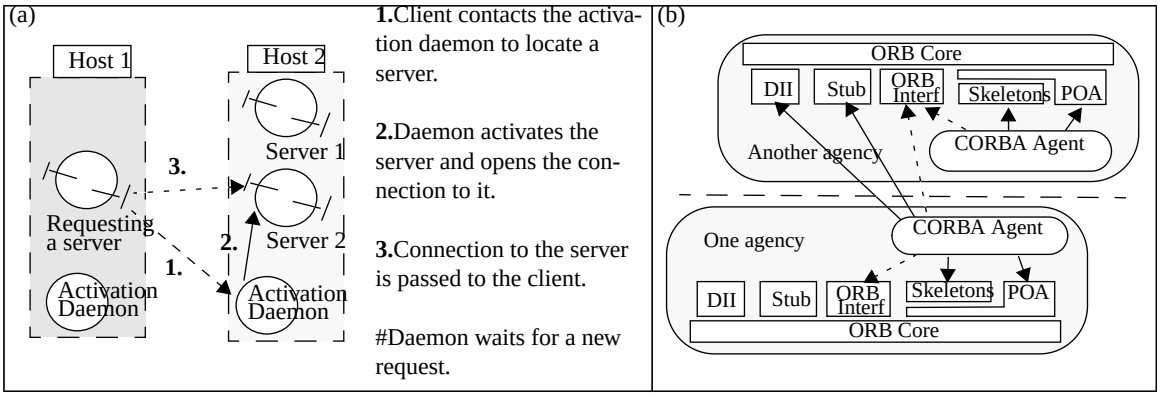

Figure 4. (a) Client / Server binding with an activation daemon. (b) CORBA Agents: communication and registration via POA (Portable Object Adaptor).

2.2. Dynamic Distributed Domains

We discuss next some aspects related to domains in open distributed system which change dynamically in time with a constant search and allocation of services.

If the search is successful, then allocation and balance of resources takes place. On the other hand, if the search or allocation ends up unsuccessful, then the search constraints are pruned for a new search. This section approaches structuring of dynamic domains using an heuristic process. A QoS value and a tolerance range are given as input to a look_up on a Trader which returns a set of guessed conditions. A balancing function is used to continuously optimize the distribution. The whole process may be started over again in order to fulfil the QoS condition within the tolerance range. The structuring of the dynamic domain is based on a Trader domain and on an Agent domain.

Trader Domain. A Trader domain is composed of the individual agencies and the individual service or agent domains which registered to it, specially the ones with dynamic properties. In Figure 5, the Traders use an information agent to get report on a set of dynamic parameters. The Trader may either dispatch this information agent to the agency or it may be part of the agency. Actually

this information agent is a customized Availability service. For example, in case of trying to connect to an agent, this information agent would be associated to a Naming service and / or an Availability service. A first attempt would try to get an agent IOR from the Naming service, if not successful a new activation of the agent is provided using the Availability service. The Trader Domain could be also called an Availability Domain.

An agency is basically composed of one or more nodes (hosts), a manager daemon for the agents [(de) activation, (de) registration, inter-agency communication] and a collection of agents (service agents, application agents) either activated or just registered.

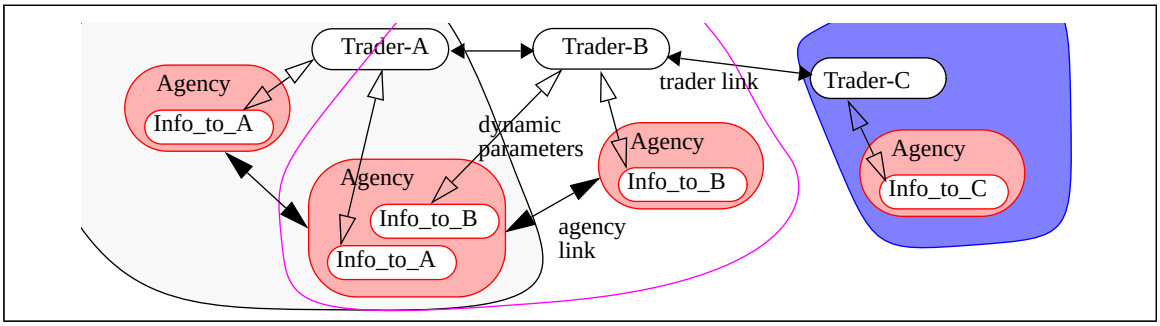

Figure 5. Each Trader has its own information domain defined by the services that registered on it.

Agent Domain. An Agent domain is built by registering the agent on a set of agencies. This set of agencies is the domain where the agent activation is authorized. The domain list is stored on the Trader in order to be retrieved by any client agent. Figure 6 shows a federation of Traders and their dynamic domains composed of a collection of agent domains.

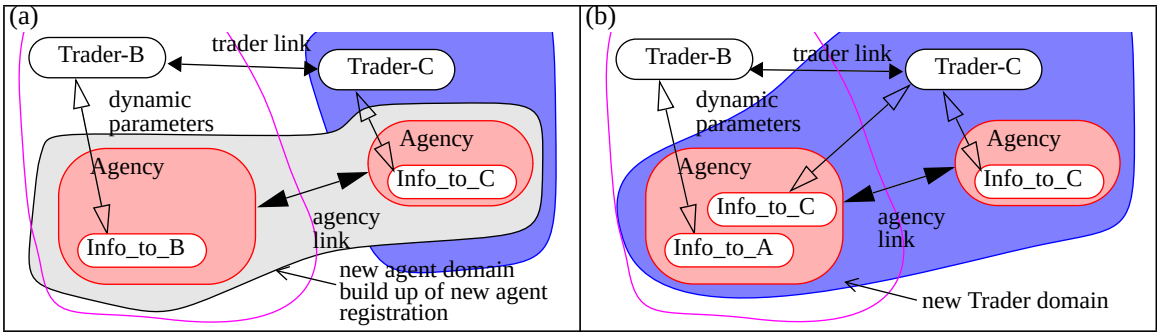

Figure 6. (a) A Trader federation is used to build up agent domains. (b) When a new agent domain is setup the agent registers on a Trader and the domain is incorporated to the current Trader domain.

Agent Registration. Related to *export* in the ODP-Trader standard [Trader95], *register* in the OMG Trader Object Service [TOS97] and *region_registration* in the Mobile Agent specification [MASIF97]. Any new service has to have an registration, like in Table 1, registered at some domain, i.e., site, group or host. This is quite similar to a cookie [Netscape] or agent credential containing: the service *authorization* parameters, the parameters that characterize the service *availability*, a pointer to a *persistent state repository*, a pointer to a *persistent implementation repository*. Migration or mobility of a portable agent is dependent on the migration of this interface and its registration as a new service/agent via a portable object adaptor (POA) [OMG97,98].

Table 1. A CORBA agent registration in the Implementation Repository. Sort of a cookie or credential.

Parameter	Value	Comments
Name	agent_name	
Domain list	hx ... hn ...	list of host addresses
Group list	gx ... gn ...	other groups of hosts
Comms protocol	xdr/tcp	
Security	ssl	a pre-define security protocol
Activation mode	shared	
Owner	owner_id	
Launch		
Invoke		
Version	x.x	
no. of servers	1 or more	single or replicated copy
server's port	0	
Marker	*	a particular method only
Launch Command	http://~AOdir.AOclass	agent class on a server over http
State	http://~AOdir.Statefile	agent state on a server over http

A remark has to be made on *state persistence* of an agent as a topic still demanding some effort. For instance, agents based on the same code (classes) might differ only on their state, and so, agents could use a state repository and be identified by their state. Not every application needs state persistence of agents as well as not every agent is allowed to be replicated, like for instance agents executing sequential tasks. In this case the state “file” has to be locked to only one agent instance at each time. State repository naming should reflect the instance naming, e.g., used by omg.org.object_to_string.

2.3. Mobility support

The Mobility service should support the reception of an agent and its persistent storage. The reception of an agent is actually the registration of its interface on the remote agency. Code/class is move effectively only when agent activation occurs and is handled by the CORBA infrastructure. Persistence of code and state, i.e., before moving the agent has to save the variables defining its state and persistently store them. Both, state and code, are moved when the agent is activated by the agency and may or not be persistently stored at the receiving site, followed by a removal at the sender after a successful completed move. At the very moment when the agent is activated by the agency, it reads back its state into the original variables. If it has to go idle that is coded in the state. Similarly, when the agent is deactivated it executes a finalize method (destructor) that updates its state repository and releases it to the next agent activation.

Migration is demanded by the environment or the service itself and in attendance for instance to load-balancing, inverse caching or redistribution due to some failure.

2.4. Searching and Balancing

Searching and balancing of resources is shortly analysed next exploring the behaviour of migration in the context of the case presented in the supportive testing. Independent of the implementation repository type, services found unavailable, are (re)distributed accordingly to demand on load balancing and / or inverse caching. This means that if distribution is not necessary in the current envi-

ronment circumstances, then it runs locally where it started. Another possibility is that applications may have to wait for resource allocation in a start-up queue.

Searching and Allocating Resources. A client agent retrieves the domain list of another agent from the Trader. With the list it tries to bind to the server agent, selecting an agency at random. The connection to the agency is via the agency manager which will look_up the local naming service for any corresponding Interoperable Object Reference (IOR). If only one activated copy is allowed the manager tries to bind to the IOR and returns the IOR. If it fails, it tries a local activation and returns the IOR. If it fails, control is returned to the client side.

Balancing Resources. Two different service / agent scenarios are possible. One scenario is based on services existing only during an application's life-time, i.e., *transitory* services. This scenario is dominated by the setup time of the individual services distribution. Another scenario consists of services / agents participating in more than one single application and persisting beyond the life-time of the individual applications, i.e., *persistent* services. Persistent is not meant to be static since agents may migrate on their own for the sake of sustained performance / functionality of the service they provide.

The *balancing* of the processing load and the communication load is based on the optimization of the computation time of each doublet, as sketched in Figure 7. The balancing of the two loadings are not untangled but treated as a whole and allowed to change dynamically as the overall conditions may change at the agencies on each side. An intermediate agency is not allowed in this balancing process since the evaluation is rather based on a doublet in order to be straight forward.

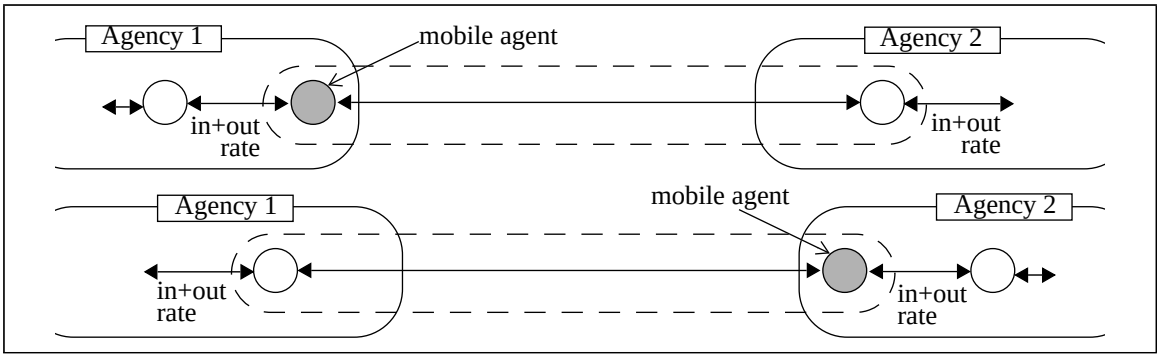

Figure 7. Agent migration is decided on each doublet for the sake of computation load-balancing.

Each agent may have more then one connection thread and we propose each one to be balanced individually as doublets, in order to avoid a global optimization structure which may introduce too much overhead. We also consider the general case where an agent is not only a client but also a server to another agent and consequently it has an interface. A doublet has to be at least a client agent and a server agent with one interface. The more general case is the one with two server agents, each one with one or more interfaces. The interfaces establish the frontiers of an agent migration cell.

3. Availability Service

The present work concentrates on an availability service [Schulze97,98] in order to support transparent migration of agents or services according to load-balancing and inverse caching [Godlszmidt96] criterias.

As seen above, the mobility of services in DOC (Distributed Object Computing) architectures like CORBA can be handled in a transparent way if the need for mobility is included in the goal of a specific service or distributed computation. For instance, performance metrics like load-balancing and inverse caching [Godlszmidt96] may be included in the goal, and to achieve this metrics an availability service is introduced.

The *availability service* is important for offering resources to services setup at some other site or agency. In order to identify an agency ready to receive a new service, the *availability service* publishes the level of availability of each agency. This information can be obtained from a querying to the agency or via a *trader.*

Availability requires a history to reflect the time the application will execute. This demands a daemon logging the loading history on every host where resources are available.

Availability Offering. An organization of this metrics is proposed into: common basic metrics, application specific metrics, and specialized metrics. The availability service offers are organized into basic types and specialized types like management and security. Specialized types inherit the basic ones.

Availability allows for instance the evaluation of loading in terms of: CPU, memory, disk, networking activity, number of users / processes. The final selection function may decide based on either a *explicit* computation or an *implicit* computation of these numbers. *Explicit* means to compute these numbers with a *specmark* of the host, while *implicit* means to use number of attempts and time out as threshold levels.

Availability Querying. Availability querying via trader [Lima95, ODP95] is used to select a range of availability of a specific kind of resource. The reply returns a list or simply the most available host. The selection phase can include a direct interrogation before contracting of the new host. A customized agent can be used to evaluate a host more closely, as a trading extended service.

Thus, the availability service allows an interface to a trader as well as some specialized interfaces for decision functions based on thresholds and activation levels. The interface to the trader is a general first step in any availability identification where a larger number of hosts exists. From this interrogation, a list of good candidates is generated, and even some dynamic parameters can be obtained. A specialized interface is important if the final selection can take advantage of thresholds and activation functions like performance evaluation and optimization. The call to the *locator* service should happen transparently either attached to the _bind or as a library function or as a CORBA object.

4. Transparency

Distribution Transparency according to RM-ODP is *the property of hiding from a particular user the potential behaviour of some parts of a distributed system*. Users may include for instance end-users, application developers and function implementors. Conceptually it defines transparency for: Access, Failure, Location, Migration, Relocation, Replication, Persistence, Transaction.

The Reference Model includes rules for selecting and combining distribution transparencies in ODP systems. It defines for each one a schema for expressing requirements and a refinement process for transforming a specification which contains requirements. In some cases, e.g. *access*, the schema is null whereas in others, e.g. *transaction*, the schema contains one or more parameters dictating the precise form of transparency required. The refinement process typically involves introducing additional behaviour, including the use of one or more ODP functions into the specifi-

cation.

Interfacing transparency requires a main goal and some constraints to be given to the search procedure:

```
interface transparency extends availability {
        // transparency specfic constraints
              parameter transparency_type;
              parameter QoS;
        // protocol constraints
              parameter recursion_depth;
              parameter time_out ;
        // performance constraints
              parameter MIPS;
              parameter execution time;

         Object agent_bind (agent_name);
       }
```

Building a Connection. A strategy followed by an agent in order to build a connection to another agent can be roughly described as follows:

```
1  try { find on naming service }
2  catch any { try {
3                    the location service local table
4                    activate server
5                    store the address in the naming service }
6              catch any { try {
7                                  submit request with constraints to a Trader
8                                  build a local table }
9                          catch any { try {
10                                        refine constraints }
11                                 catch any {exit }
```

To enable a client agent to make a remote method invocation on another agent's method, a previous binding to the remote agent is needed.

5. Discovering Services

CORBA Naming Service. The Naming Service [Vogel97] locates object implementations and thus it is a fundamental service for distributed object systems. It provides an extra layer of abstraction for the identification of objects, allowing readable object identifiers and persistent identification mechanism, i.e., objects can bind themselves under the same name regardless of their object reference.

Trading Service. In Figure 8 there is a representation of the full-service trader and its interfaces [Bearman96, TOS97, TOI]. Traders are repositories of object references that are described by an interface type and a set of property values. Interfaces definitions: Type Repository - Trader Components - Look_up - Iterators - Register - Link - Admin - Proxy - Dynamic Properties. In case of querying via a trader [Trader95, Lima95], the query includes a range of availability of a specific kind of resource. The trader replies returning a list or simply the most available agency. The selection phase can include a direct interrogation before contracting for the loading of the new service by the application.

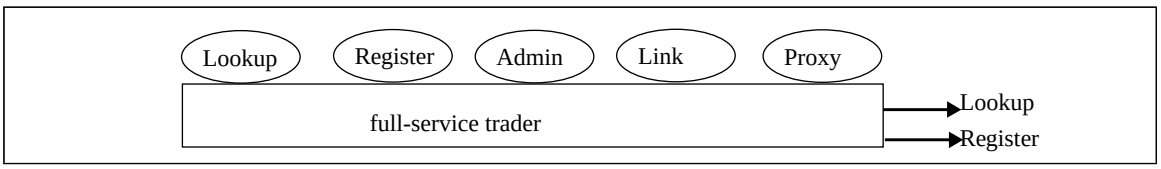

Figure 8. Full-service trader.

Proxy Interface. If a trader supports the proxy offer capability, then this interface provides the operations to accept, withdraw, and describe proxy offers. If an importer´s Query operation results in a match to a proxy offer, the trader holding the proxy offer performs a nested Query on a 'secondary trader' indicated by the proxy offer. The proxy offer is computationally represented by:

- Service Type Name (type)
- Look_up Interface (target)
- Service Properties (properties)
- Matching flag (if_match_all)
- Constraint Recipe (recipe)
- To be appended policies (policies_to_pass_on)

Specific Function Interfaces. If a trader supports the dynamic property capability, then the trader can be a client to the DynamicPropEval interface, which is provided by an exporter who wishes to provide dynamic property value in a service or proxy offer.

A *nil* interface value is returned from a Query to the TraderComponents interface if a Trader does not support a particular interface and a NotImplemented exception is raised if an invoked operation is from an interface not supported by a trader.

6. Supportive Testing

This section presents an overview of the general steps of the work and shortly some prototyping results. An implemented application simulates a mobile host disconnecting from the static cluster of hosts similar to a host going unavailable, i.e., either going down, losing performance or failing. In these situations, the client has to migrate to another available host on the network.

The detailing and implementation of the availability service involves definition of two main interfaces. The first deals with the trader service, the query service and properties service, and their relation to the existing definitions and implementations of these services according to OMG specifications. The second interface deals only with some specific resources where a quick final evaluation of the availability of the particular resource is needed through direct interrogation and requesting for prompting. In short, the availability service could be seen as a branch of the Trader in the domain but with down sizing functionality. It sort of intermediates the Trader access on dynamic properties.

The detailing and implementation of the trader service involves porting its defined functionalities according to OMG specifications and integrating it to the other services and environment. For the trader, a first minimum version is thought, and further a full version (TOI - GMD/Fokus).

Mobility implementation considers existing mobility platforms [Grasshopper] and their relation to CORBA. The Mobility service should support the reception of an agent and its persistent storage. The reception of an agent is actually the registration of its interface on the remote agency. Code/class is move effectively only when agent activation occurs and is handled by the CORBA infrastructure. Any new service has to be registered at some domain: site, host or group. The registra-

tion publishes: the service *authorization* parameters, the parameters that characterize the service *availability*, a pointer to a *persistent state repository*, a pointer to a *persistent implementation repository*. Migration or mobility of an agent is dependent on the service/agent registration via a portable object adaptor (POA) and its activation by the *activation* daemon (i.e., agency capsule manager daemon).

A full integration of the above service is foreseen. Testing should involve the evaluation of an application regarding performance in load balancing and inverse caching.

The evaluation test is based on a cyclic execution of a set of commands involving communication loading and processing loading of the hosts. This cyclic test is composed of a *master* object serving a *client* object, Figure 9. The client can be either the *original* client or a *replica*. The *original* client is activated always on a pre-determined host (mobile host) while the *replica* and the *master* keep moving around for load balancing, selecting the first prompting host from a set of hosts.

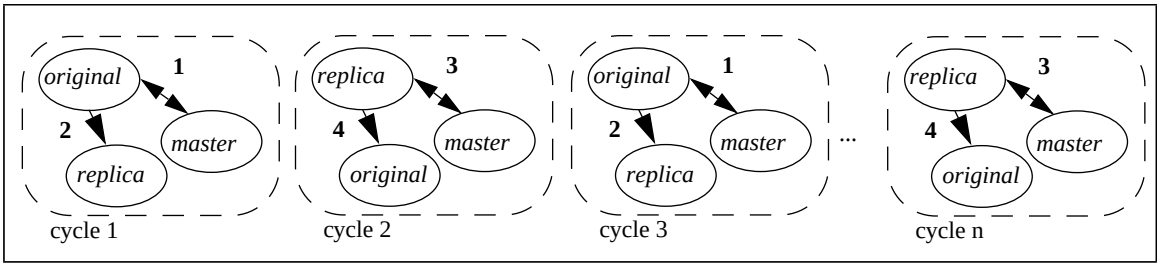

Figure 9. Sequence of activation and communication of the test objects: 1, 2, 3, 4, 1, 2, and so on.

A multi-host service means that several hosts may attend to the request of a specific service and the first host sufficiently unloaded will attend to the request allowing an intrinsic load balance in the execution. The host configuration per service is represented in Figure 10. Domains may be used as an organization of the hosts into clusters of maximum number of multi-host services.

Specmark. The execution performance is obtained with respect to a specmark based on the execution of all Client/Server processes on the same host. It is obtained for two different loading conditions, i.e, normal loaded and extra loaded with a load program. The specmark is not an actual situation [Schulze98].

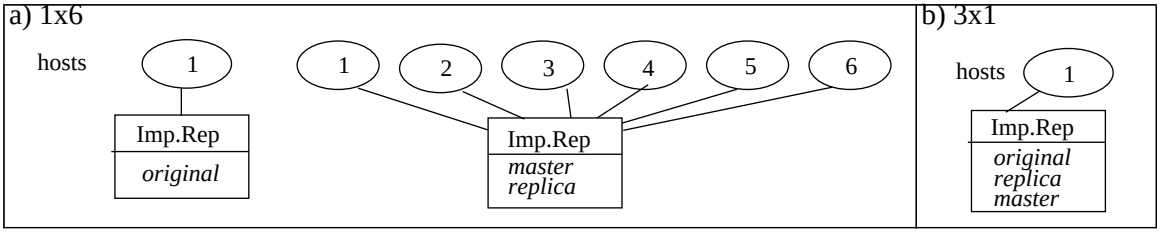

Figure 10. Hosts allocation: a) 1 per original client and 6 per master and replica; b) all on a single host.

The numbers in Table 2 are average values computed in different occasions selected at random, where a minimum and a maximum values are annotated. On a previous work [Schulze97-2] we presented the results on different loading conditions and the configurations of Figure 13a and b. A range was obtained for normal loading and extra loading and an additional measurement for the configuration of Figure 13a with half of the hosts extra loaded. This value did fit quite well between the range obtained. Actually, the more hosts exist better chances there are of migrating to

unloaded hosts. The average performance should be better with migrating distributed services rather than with distributed stationary services.

The C++ test was repeated for a comparison to the Java result. The old test used a bind twice to the master server, passing 10 times a sequence size of 100000. The new test uses the same twice bind to the master server but with a sequence size of 100 passed 10000 times. The reason for that change in the test program is that the Java version was generating a too big server code and not being able to activate it successfully. The result for Java could be over 2x faster with *just in time* compilers (JIT) or over 10x faster on Java chips [Javachip].

Table 2. Porting of the test to Java and a different set of SunOS 5.5 machines.

Host Type	**Clock** (MHz)	**Memory** (MBytes)	**Comparative Clock | Mem.**	**Java/OrbixWeb Specmark (3x1 Fig. 13a)**	**C++/Orbix2.1** (min/cycle)
SparcUltra	143	64	2.0 | 1.0	8.4[2]	3.8[2] | 3.3[1]
sparcstation20	2x 60	96	1.7 | 3.0		3.5[1] | 4.8[1a]
sparcstation20	2x 50	128	1.4 | 4.0	-	4.8[2]
sparcstation5	110	64	1.6 | 2.0	9.4[2]	
sparcstation5	70	82	1.0 | 2.5	22.0[2]	6.8[2] | 4.3[1]
sparcstation5	70	32	- 1.0 | 1.0 -		5.5[1] | 8.7[1a]
				Results (1x6 Fig.13b)	(min/cycle)
all the hosts	**normal**	**loaded**		13.2[2]	6.2[2]
half the hosts	**extra**	**loaded**			5.4[1a]
all the hosts	**extra**	**loaded**			10.4[1a] - 11.5[1a]
			1.old test / normal loading	1a.old test / extra loading	2.new test / normal loading

7. Related Work

Distribution transparencies are important in agent system as they are to any distributed computing [RM-ODP, Cristian94-93-91]. If we concentrate on Distributed Object Computing and extend the benefits to agent systems we should expect at least a similar level of abstraction in programming, reusability, portability and reliability [ODP95, COF97, COS97, CORBA95, Orfali96]. These should be specially considered in applications where the agent based model suits well and is suppose to cover some lack in formalism and/or easy description based on the traditional client/server model.

Some techniques, like for instance load-balancing and its performance evaluation could be exercised on agent systems and extended to transparent allocation of overall computational resources in highly reconfigurable environments. Several QoS parameters could be worked out, e.g.: communication channels, storage devices and so on[Ciupke96, Golubski96, Goldszmidt96].

Continuous effort on DOC platforms and a world of agents and multi-agents systems {MASIF97] suggest more exploratory work on techniques previously reported on non OO environments. For instance, management of open distributed systems is an application we have been working out and which seems to profit very well from multi-agent systems characteristics [Ropelatto98].

Distributed Problem Solving (DPS) with Multi-Agent Systems (MAS) involves research in coordination, negotiation and communication. Agents must *communicate* to *coordinate* their activities and *negotiate* their conflicts. Conflicts may result from limited resource contention to more com-

plex issue-based computations where agents disagree between their domains of expertise.

Interactive knowledge media or collaborative problem solving environments rather then traditional autonomous intelligent thinking machines was proposed by Fischer. Users and system share the problem solving and decision making whereas different role distributions are chosen according to the user's goal/knowledge and the task domain. Semi-formal system architectures are appropriate. Partial understanding and knowledge is acceptable. Breakdowns are not detrimental if there is support to deal with exceptions. Agents may explore asymmetry and achieve more than one.

Asynchronous Teams of agents execute their tasks without any explicit control but in a collaborative way, each one knowing precisely its function and actuation time, asynchronously activated without scheduling. It is specially suited for multiple complementary strategies in solving the same problem [Talukdar].

8. Conclusion

In this paper we extended architectural aspects of agent systems based on distributed computing concepts. Focusing on distributed object computing, most aspects translate very well specially if we work with agents as active objects [Cardozo93] with special properties like [Franklin96]: reactive, autonomous, goal-oriented, temporally continuous, communicative, learning, mobile, flexible (not scripted), character (personality).

The mobility support needed in addition to the existing CORBA infrastructure is simplified to the registration of the agent on the remote agency. Transparent mobility is handled by the CORBA bind (object_to_string) call and the location mechanisms.

A previously discussed Availability service was extended with the Transparency service/interface. Transparency may also be implemented as an additional interface to the availability service instead of a full object. The corresponding methods should be incorporated by the availability service.

Acknowledgments. This work is partially supported by FAPESP, CAPES and CNPq (Brazil) and in collaboration with IMA-CC GMD Fokus (Berlin/Germany).

References:

[Bearman96] M. Bearman . *Tutorial on ODP Trading Function*, DSTC, Faculty of Information Sciences & Engineering, University of Camberra, Australia, revised june 1996.

[Cardozo93] E. Cardozo, J. S. Sichman and Y. Demazeau . *Using the Active Object Model to Implement Multi-Agents Systems*, Proceedings of the 5th IEEE Conference on Tools with Artificial Intelligence, Boston, USA, pp. 70-77, November 1993.

[Ciupke96] O. Ciupke, D. A. Kottmann, H-D. Walter . *Object Migration in Non-Monolithic Distributed Applications*, Int.Conf.Distributed Computing and Systems, Hong Kong, pg.529-536, May 27-30, 1996.

[COF97] OMG . *Corba Facilities*, 1997.

[CORBA95] OMG . *The Common Object Request Broker: Architecture and Specification*, rev 2.0, July 1995.

[COS97] OMG . *Corba Services*, 1997.

[Cristian91] F. Cristian . *Understanding Fault-Tolerant Distributed Systems*, Communications of ACM, 32(2):56-78, February 1991.

[Cristian93] F. Cristian . *Automatic Reconfiguration in the Presence of Failures*, Software Engineering Journal, IEE and British Computer Society, pp. 53-60, March 1993.

[Cristian94] F. Cristian . *Abstractions for Fault-Tolerance*, 13th IFIP World Computer Congress, August 28 - September 2, Hamburg - Germany, 1994.
[Fischer95] G. Fischer. *Rethinking and Reinventing Artificial Intelligence from the Perspective of Human-Centered Computational Artifacts*, LNAI #991 Springer-Verlag, pp 1-11, 95.
[Franklin96] S. Franklin and A. Graesser. I*s it an Agent, or just a Program?: A Taxonomy for Autonomous Agents*, 3rd Int. Workshop on Agent Theories, Architectures, and Languages, 96, www.msci.memphis.edu/~franklin/AgentProg.html.
[Goldszmidt96] G. S. Goldszmidt . *Distributed Management by Delegation*, Ph.D. Thesis, Graduate School of Arts and Sciences, Columbia University, US, 1996.
[Golubski96] W. Golubski, D. Lammers, W-M. Lippe . *Theoretical and Empirical Results on Dynamic Load Balancing in an Object-Based Distributed Environment*, Int.Conf. Distributed Computing and Systems, Hong Kong, pg.537-544, May 27-30, 1996.
[Grasshopper] IKV++ GmbH . Grasshopper:An Intelligent Mobile Agent Platform,1998.
[Javachip] OSF, *Java Mobile Code Paper*, www.sun.com/microelectronics/whitepapers/wpr-0014-01/.
[Java] Sun MicroSystmes, Inc . java.sun.com/products/jdk/1.2/docs/.
[Krause96] S. Krause, T. Magedanz . *Mobile Service Agents enabling "Intelligence on Demand" in Telecommunications*, IEEE GLOBECOM'96, London, UK, pp. 78-84, November 1996.
[Lange95] D. B. Lange, D. T. Chang . .*Aglets Workbench*, IBM Corporation, August, 96, www.ibm.co.jp/trl/aglets.
[Lima95] L. A. P. Lima Jr., E. R. M. Madeira . *A Model for a Federative Trader*, Open Distributed Processing: Experiences with Distributed Environment, pp.173-184, Chapman&Hall, 1995.
[MASIF97] GMD FOKUS and IMB Corporation, Mobile Agent System Interoperability Facilities Specification, OMG TC orbos/97-10-05, November 10, 1997.
[Netscape] Netscape Communicator, Netscape Communications Corporation, http://home.netscape.com/.
[ODP95] ODP . *Trading Functions*, ISO/IEC JTC1/SC 21, June 1995, ftp.dstc.edu.au/pub/arch/RM-ODP.
[OMG97] OMG . *Minimum CORBA*, OMG Documet orbos/97-11-08, November 10, 1997.
[OMG98] OMG . *ORB Portability IDL/Java Language Mapping*, OMG TC Doc orbos/98-01-06, Jannuary 19, 1998.
[OrbixWeb] Iona Technologies, www.iona.com/
[Orfali96] R. Orfali, Dan Harkey, J. Edwards . *The Essential Distributed Objects Survival Guide*, John Wiley & Sons, 1996.
[RM-ODP] ODP . Part 1: Overview and Guide to Use; Part 2: Descriptive Model; Part 3: Prescriptive Model; Part 4: Architectural Semantics; Trader: Defines the RM-ODP Trader, www.dstc.edu.au/AU/research_news/odp/ref_model/standards.html.
[Ropelatto98] P.Ropelatto, E.R.M.Madeira, B.Schulze, F.Vasconcellos . *Distributed Objects Management in Corba Environment using Mobile Agents*, to appear on NCS98 - International Conference on Networks and Communication Systems, Pittsburgh, USA, May 13-16, 1998.
[Schulze97] B. Schulze, E. R. M. Madeira . *Contracting and Moving Agents in Distributed Applications Based on a Service-Oriented Architecture*, Mobile Agents - MA97, LNCS #1219, Springer-Verlag, pp. 74-85, 1997.
[Schulze97-2] B.Schulze, E.R.M.Madeira . *Using an Availability Service for Transparent Service Migration in Mobile Computing*, Technical Report 97-11, Institute of Computing, University of Campinas, S.Paulo, Brazil, August 1997.

[Schulze98] B.Schulze, E.R.M.Madeira . Technical Report 98-xx, Institute of Computing, University of Campinas, S.Paulo, Brazil, August 1998.
[Talukdar] S. Talukdar . *What is an Asynchronous Team of Autonomous Agents?*, www.cs.cmu.edu/afs/cs/project/edrc-22/project/ateams/WWW/.
[TOS96] OMG . *Trader Object Service*, RFP5 Submission, May 10, 1996.
[TOI] IKV++ GmbH . TOI: OMG Trading Object Service, http://www.ikv.de/products/toi.html.
[Trader95] ODP . *Trading Functions*, ISO/IEC JTC1/SC 21, June 20, 1995, ftp.dstc.edu.au/pub/arch/RM-ODP.
[Visibroker97] Visigenic Software, Inc. . *Programmer's Guide, Release 3.0*, Visibroker for Java, 1997.
[Vogel97] A. Vogel, K.Duddy. *Java Programming with CORBA*, John Wiley & Sons, 1997.