Transparent Service Mobility Using CORBA

B.Schulze, E.R.M.Madeira

Relatório Técnico IC - 98 - 27

Agosto de 1998

Transparent Service Mobility Using CORBA

B.Schulze[1,2] **schulze@[dcc.unicamp.br | vxcern.cern.ch]**
E.R.M.Madeira[1] **edmundo@dcc.unicamp.br**

1 Institute of Computing - IC / Unicamp
PO Box 6176
13083-970 Cidade Universitária
Campinas, SP - Brazil
fax: (+55)(19)239-7470

2 Brazilian Centre for Physics Research - CBPF / CNPq
R.Dr.Xavier Sigaud 150
22290-180 Urca
Rio de Janeiro, RJ - Brazil
fax: (+55)(21)541-2047

Abstract. Distributed Object Computing platforms like CORBA allow for object programming separate from configuration and distribution, and the addition of a *mobility* support extends distribution aspects. Reviewing distributed object computing, we present a service-oriented architecture based on a OMG/CORBA platform. The addition of an availability service associated to a mobility service allows for transparent migration of components, i.e., services or agents. The current work presents more the structure behind the availability service and less the transparency interface to it. Transparent mobility of services is interesting in applications using load-balancing mixing strategies of normal caching and inverse caching, i.e., code moved close to data. The search for a new destination host follows a transparent location and selection of resources available on other hosts. Supportive testing is presented on an example of component migration in mobile computing.

Keywords. distributed object computing, service-oriented architecture, mobility, availability, trading.

1. Introduction

Despite of the benefits provided by distributed computing, the development of distributed applications still requires good skills on various topics not necessarily inside the scope of the application object itself. This has pushed the development of distributed platforms offering a set of tools, an execution infrastructure and an appropriated paradigm. The developer should concentrate on the application itself while the platform takes care of distribution and communication.

Software development [Rumbaugh91, Booch94] changed a lot with object-orientation and distributed objects programming. Distributed Object Computing (DOC) merges the usual Client/Server approach and the object-orientation paradigm. The encapsulation of data and methods (operations) and the corresponding interfaces define services as objects which can be executed in different platforms. In some cases these services may manage themselves as well as the associated resources, allowing more flexible Client/Server applications. Object-orientation and distributed computing [Schmidt95] motivated Distributed Processing Environments (DPEs) based on objects or object-oriented, like ANSA [APM93] and OMG/CORBA [CORBA95], correspondingly. The Client/Server model is used as the communication mechanism among application objects. Servers make their services available by registering the name and type on a name server. Clients query the name server for a reference to a particular service in order to use it.

We propose a service-oriented architecture based on OMG/CORBA and the addition of an availability support and mobility support for the sake of transparent migration of services. The availability support is in charge of locating the resources needed by a service to run. The resources can be either hardware or software whereas other services like a Trader [TOS96, Trader95, Bearman96] participate in the location process.

This work also presents some supportive testing based on the migration of tasks from a mobile host to any other host in the environment, whenever the mobile host disconnects. This migration is handled transparently over a DOC platform [CORBA95] with the usage of an availability service to transparently locate resources on other possible destination hosts. After location, a selection function efficiently chooses the new destination host where to send code and state of the related components.

Organization of the paper. Section 2 is a survey of some basic concepts related to distributed object computing. Section 3 introduces the proposed service-oriented architecture. Section 4 presents some results on migration of components and implementation details. Section 5 contains related work and techniques and Section 6 in this section we summarize the paper and describe some further work.

2. Basic Concepts

Describing the Client/Server model using object orientation simplifies the modelling of distributed applications, although there are some constraints when demanding many-to-many objects interconnection. Allowing a server to be a client of other servers creates some hard interconnection processes. The solution to this is a mechanism which allows these interconnections in a more flexible way, i.e., facilitating associations between client and server objects. This software engineering approach is based on the Configuration paradigm to build a system structure separate from the components [Kramer92, Sloman89, Magee90].

The components explicitly declare the services they provide as well as the services they need, Figure 1. The term *component* is defined as a module having independence of context, i.e., it has no knowledge of its binding to the external environment. In terms of the Client/Server model the required services are references to other components. A Configuration Language is used defining the application structure in terms of instantiation of components and the bindings among required and provided services.

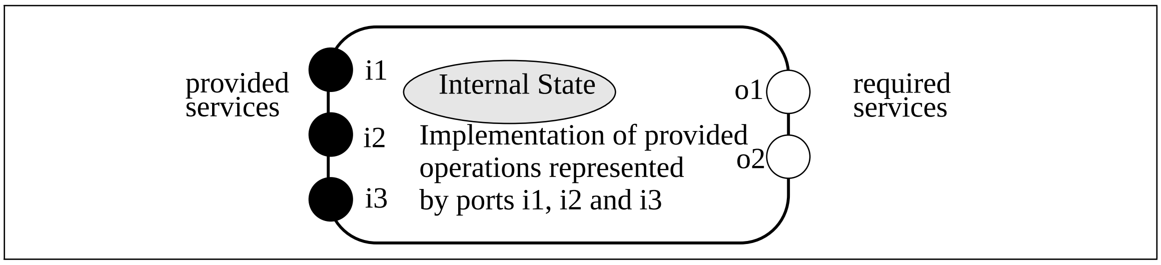
provided services
i1
i2
i3
Internal State
Implementation of provided operations represented by ports i1, i2 and i3
o1
o2
required services

Figure 1. Component independent of context.

2.1. The Configuration Paradigm

Two basic approaches may be considered when building distributed applications. In the first approach, the system is implemented all at once, compiled and executed. This is not interesting from the point of view of robustness which is a basic request of some distributed applications. In the second approach, the system is split into modules in charge of well defined tasks, so that the application may be build up of interacting modules. Each module is compiled separately and interconnected with a Configuration Language.

The Configuration paradigm assumes [Kramer90]:

- a application structure programming separate from the modules programming.
- individual modules with well defined interfaces and self-contained in a context independent way.
- more complex modules build up based on the simpler modules.
- changes in the application structure based on the replacement of modules and interconnections.

Separating the modules programming from the Configuration makes it easier to understand and to manipulate the structure of the system. The Configuration declaratives have to be concise and easy to manipulate, possibly taking into account only the description of the structure of the system. Independence of context [Kramer85] means that all references made by a module have to be exclusively to local entities. This facilitates software reusability, in the sense of modules being integrated in any compatible context without demanding any internal modifications. Access to external entities has to be done through indirect calls to other modules.

2.2. Open Distributed Processing - ODP

The Reference Model of ODP [RM-ODP] standardizes a framework for an architecture with the integrated support of distribution, interworking and portability. The Object Management Group (OMG) has a published architecture explaining how to support distributed systems. The RM-ODP adds value to this by addressing federation, transparency and system management, and by defining a fine grained framework of reference points to support the integration of functions from different sources. The following well defined inherent characteristics are normally exhibited: remoteness, concurrency, lack of global state, partial failure, and asynchronism. The absence of any central control adds: heterogeneity, autonomy, evolution, mobility, openness, integration, flexibility, modularity, federation, manageability, provision of Quality of Service (QoS), security and transparency.

ODP standardization recognizes that there cannot be an ubiquitous infrastructure that meets all heterogeneity and transparency requirements. There is a need for a framework describing infrastructure components and showing how they fit together. This enables the development of standards specifying components that are mutually consistent, and that can be combined to build infrastructures matched to user requirements.

The concept of common services is an essential element of ODP system concepts. This assumes that it is not sufficient to standardize separately, for example, the representation form of data, the transport protocol for that data, the location algorithm of a service provider, and so on. By grouping together the properties of different services and gaining an agreement on these, the amount of heterogeneity can be structured in a controllable way. This model of capturing features of distributed systems in common services shows itself on multiple levels of abstraction in the RM-ODP approach to the system structure.

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 and Transaction.

Mobility and CORBA. The *move* method is present under the Life Cycle object service but OMG/CORBA [CORBA95] does not yet have a *mobility* service. There is an open request for proposal where several contributions have been presented up to now [MASIF97]. Mobility support can be considered under CORBAfacilities as well as under CORBAservices. A good example of a mobility support is present in Aglets [Lange95].

2.3. Software Mobility in Distributed Processing

The traditional model for distributed processing involves stationary processes transferring data and/or commands. Data and commands form the mobile part of a computation whereas program execution is static. There is a growing number of network computing scenarios which cannot be effectively addressed by such static interaction paradigms.

A suggested paradigm is mobile agents, where a mobile agent is an uniquely identified program that can migrate from machine to machine in a heterogeneous network. A kind of microkernel exists at every destination machine, acting like a pier for receiving/executing the agent. The performance advantage is expected to be greater in low bandwidth and high latency networks. This trend should persist since the expected performance increase of microprocessors should continue at a higher rate and lower cost than network bandwidth availability.

Inverse Caching [Goldszmidt96] is an interesting solution to the latency problem, i.e., to move the applications closer to where the resources are located. Data caching, in contrast, is used to keep data closer to where it is used, being most effective when there is a high degree of locality of use and a high degree of *hits*. Caching, however, cannot properly support highly volatile distributed data, which is quickly out of date, and makes the cache inconsistent. For example, in measurements from remote monitoring instruments, if they change frequently, it is more efficient getting code closer to the data. Obviously, inverse caching is only an advantage as long as the penalty of moving code is smaller than the gain obtained by shifting a function closer to its data.

Mobile agents are a convenient paradigm for distributed computing. First, they allow the implicit transfer of information, carrying all of its internal state with it, which eliminates the need for separate communication steps. Second, tasks like network management, information retrieval and workflow, fit naturally to this jump-execute-jump of mobile agents. An agent migrates to a machine, performs a task, migrates to another machine, performs another task that might be dependent on the output of the previous one, and so on. Finally, an application can dynamically distribute its components when it starts execution [Lange95].

Mobile agents are an interesting extension of the traditional Client/Server model, where client and server can reprogram each other, allowing the extension of the functionality that an application may provide.

Sequential and Parallel. In distributed processing, mobile agents are considered to migrate from node to node, like executing a sequential task, as represented in Figure 2. While concurrence contributes to execution speed up, on the other hand, a sequential execution may reduce communication overhead. This last may be the main contribution of mobile agents, which may move to where data is.

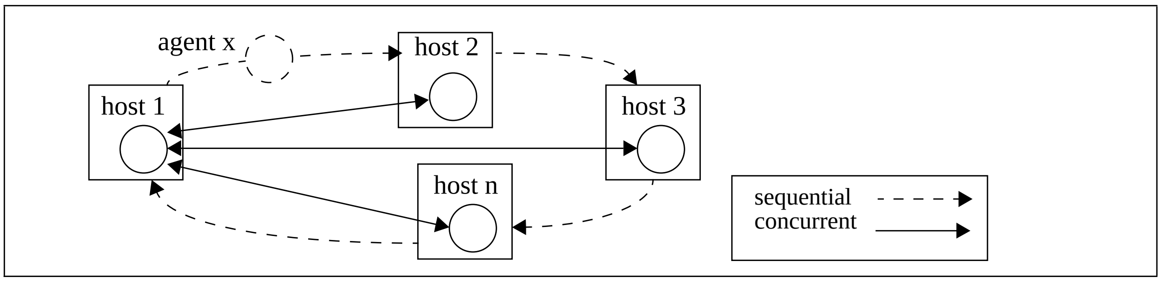

Figure 2. Distributed processing with concurrent/sequential execution.

2.3.1 Benefiting from Mobility

Client/Server applications often make implicit requirements on resource-rich, powerful computing environments. Many popular distributed applications are optimized to execute on high-performance workstations and communicate over high bandwidth links. Distributed applications which are popular in resource-rich workstations are often rendered almost useless in resource-constrained environments, such as mobile devices. Such devices are intermittently connected to wireless networks, and have low-bandwidth connections, particularly outside office buildings. Mobile devices have to constrain computing power, in part due to battery operation.

Implementing an application based on mobile components / agents can facilitate its distribution, i.e., only a small core of the process needs to be distributed while it will dynamically retrieve additional functions as needed. Distributed applications can store temporary data at intermediate proxy servers and use mobile agents to page them.

Mobile agents need a mobility infrastructure to dispatch an agent to a remote host, to invoke it and control its execution. Applications where mobile agents are useful have in common the following characteristics: (1) they are long-lived, (2) they execute over distributed heterogeneous environments, (3) they must adapt to changes in the environment, (4) they have real-time constraints and (5) they must execute over hosts with limited computing resources or over low bandwidth networks.

Although an alternative based on existing technology can always be proposed, in certain cases mobile agents have advantages over conventional approaches, at the design, implementation and execution stages. Following there is a list of some proposed advantages of mobile agent technology: efficiency, reduction of network traffic, asynchronous autonomous interaction, interaction with real-time entities, dynamic adaptation, dealing with large volumes of data, robustness and fault tolerance, support for heterogeneous environments, personalized server behavior and convenient development paradigm.

2.3.2 Attractive Applications

Functionality Dynamic Allocation. Mobile agents allow dynamic distribution of software to devices, thus allowing dynamic extension of functionality in conformance to changes in the requirements, adapting to changes of network and computing resources, as well as taking advantage of newly available resources. Network devices, for instance, are increasingly being equipped with computational power, including faster CPUs and larger memories. It is cost effective to dynamically move functionality to less expensive devices.

Autonomy. Assuming that a network is going to be partitioned for some time, agents could be moved to execute critical management at the devices while there is no connectivity to the central platform.

Software Upgrading. Long lived distributed applications cannot foresee all the functional upgrades and customization that will be required after being installed at each site. Upgrading software typically requires shutting down processes and replacing them with newly compiled versions. In many cases, the expenses of bringing the system down are substantial. Mobile agents allow substitution of components separately without the need for a global shutdown.

Software Customization. Many devices can be dynamically customized using mobile code. For instance, the programming of an interactive device where the application could dynamically replace, add, or remove specific algorithms. Such an agent could be developed based on the experiences of other users/situations, after the original application has been deployed.

Software Monitoring and Debugging. Another use of mobile agents is to augment a distributed application with monitoring capabilities for debugging. Certain events on distributed hosts can be monitored and traced. For instance to discover patterns of data access, to collect and correlate event traces, and so on.

Application Interoperability. Many distributed applications must support an increasing number of protocols to provide interoperability with different applications, for example, different data representation protocols or an application gateway to support application interoperability in heterogeneous distributed systems.

Providing fixed support for all possible combination of protocols and data formats results in large additions of code, whether used or not. For example, a laptop client could dynamically retrieve the decrypting code only when needed, thus saving resources, i.e, dynamic adaptation to resources availability.

Distributed, Collaboratory Experiment Environments (DCEE). The word *collaboratory* has been created to define integrated, tool-oriented computing and communication systems to support the collaboration of large experimental scientific facilities. A major step toward realizing that potential comes from combining the interest of the scientific community at large with those of the computer science and engineering community [DCEE97, Kouzes96, Schulze95].

Mobile Computing. A mobile host may need to disconnect from the stationary network, for example, due to changing communication base station, power saving, and so on. This is very similar to the shutdown or failure of a host in an environment where the availability and functionality of the services want to be sustained. Provided services and active objects executing on the mobile host have to be migrated to some other host in the environment.

Support for Electronic Commerce. Combining asynchronous operation with control flows using transactional semantics suggests that agent-based transactions introduces base concepts for reliable processing in future distributed systems, like virtual enterprises and electronic markets [Assis97].

3. Service Oriented Architecture

Our proposed service-oriented architecture is based on distributed active objects using OMG/ CORBA (Common Object Request Broker Architecture) as a broker for object components [Orfali96] either in a stationary phase or in a mobile phase [Schulze97]. The general concept of service is associated to active objects or agents as services already available or in the process of being put available somewhere.

An application based on this architecture uses, as much as possible, services which are available whereas for services not available anywhere it can either: wait, abort or customize a new service. Customization involves mobility of components in order to optimize the performance of the whole execution.

Transparent location of components, in the CORBA model, is extended to transparency in the mobility of components. Hosts' resources become a service and their availability is published by an *availability service*, so that an application contracts specific resources from a host for a particular component to be downloaded.

Using performance metrics altogether with an availability service and a mobility service allows also for transparent code distribution at application start-up. An availability service is a big part of what is missing in Aglets [Lange95] and most mobile agents platforms.

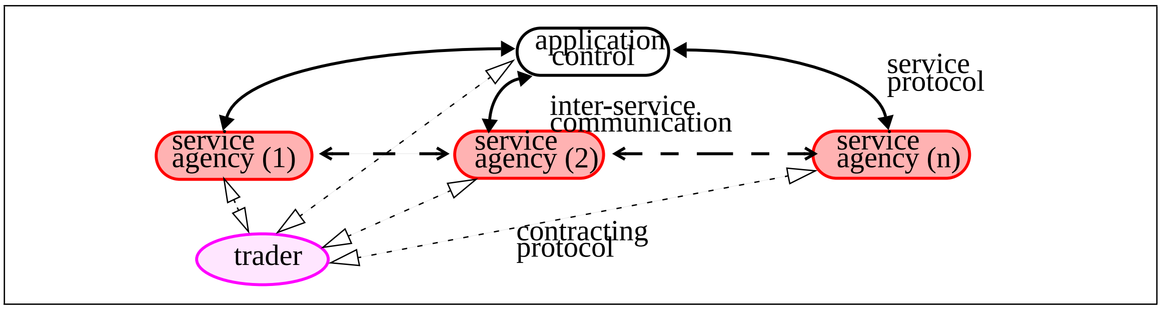

Figure 3. Service-Oriented Application.

The diagram of Figure 3 illustrates a service-oriented application and some basic blocks:
- *agents* include all kind of services used by an application. Agents in a first approach are autonomous in the sense of deciding where and when to move, but not yet flexible in the sense of changing their strategies.
- *agency* is a basic component able to receive and offer services to an application. From a generic point of view this ability is a service itself and the structure of an agency is similar to an object encapsulating all the other service objects.
- *trader* [ODP95, Lima95] is yet another service for locating / negotiating other services in a pool of contracted agencies.

A requested services may be either *available* or *non-available*. Being available means that it is ready to be used at some agency. If it is non-available means it needs to be customized. The application may decide if the customization of a new service should take place. If affirmative, a similar procedure followed by the application up to this point will be used to customize the new service. This should happen recursively every time a service is not available with an intensive location of resources. In order to improve the transparent location of resources, quality of service (QoS) metrics may be given, like: trustability, fault-tolerance, hardware dependencies, execution time, among others.

In Figure 4 services are represented as agents distributed over the domain either stationary and attached to the local ORB infrastructure or on the move to a new host. On each host there are one or more agencies which act like a shell that encapsulates local agents. Since the approach of services build up of other services is recursive, an agency itself is seen as a service with a particular interface to the operating system of the hosts. Its functionality is to support agents which do not have this functionality.

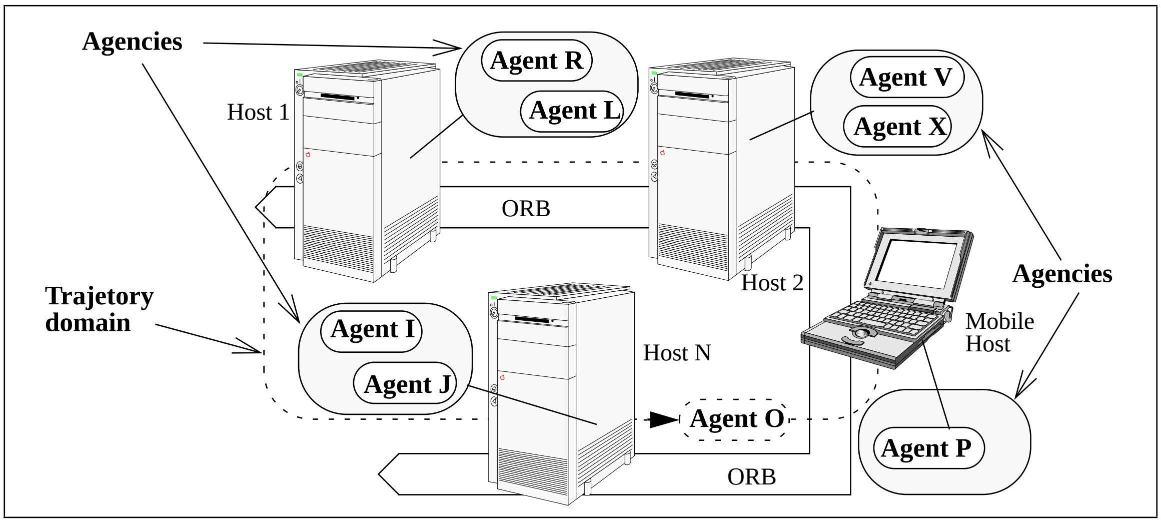

Figure 4. Stationary and moving services, i.e., agents.

3.1. Service Agents

Computing with services allows a higher level of abstraction in implementing any application today reducing the development effort to specific objects not available anywhere and to the interconnection of all the active objects regarding the application. The interconnection of these objects deals with: *contracting*, *locating*, *requesting* and *replying*.

Service Available. Should be any facility ready to be used somewhere, like for instance: any software object as well as objects on firmware, specialized processing, co-processors, databases, data crunching and so on.

Service Not Available. This means that a requested resource is not available for at least one of the following reasons: access is not authorized; not where needed; temporarily disconnected; a too specific computation that has to be customized by the application.

The application has to handle unavailability of a service accordingly and customize a new service. Service customization deals with: resource allocation for execution, naming, registry and code transportation. After customization, the new service is seen just as any already available service. Any service may use other remote services establishing an inter-service communication.

3.2. Service-Oriented Agency

The agency architecture is composed of an object broker and a collection of agent services, which may include or not an *agent mobility service* and an *availability service*. An agency with agent mobility and availability services is able to run new services loaded by the application itself, i.e., the agency is open to new services or agents to be loaded by an application demanding this kind of service. In Figure 5 the availability and the mobility services could be either under CORBAfacilities or even CORBAservices.

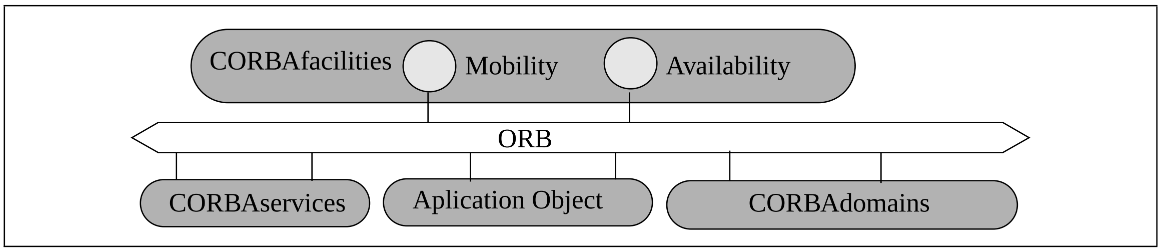

Figure 5. Adding mobility and availability support under CORBAfacilities.

3.3. An Availability Service

In a previous [Schulze97] work, the concept of an availability service altogether with a mobility support is discussed in order to transparently move agents or services. 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 these, 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 allows to 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.

3.3.1 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, according to Table.1. Specialized types inherit the basic ones.

Table 1. Examples on an Availability Service type offers.

Basic types		**Specialized types**
Static	*Dynamic*	
hosts' hardware resources	resources allocation	performance
physical location		management
communication hardware		security
protocols		
network management		
MTBF and aging		

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 an *explicit* computation or an *implicit* computation of these numbers. *Explicit* means to compute these numbers with a *specmark* of the host, whereas *implicit* means to use the number of attempts and time out as threshold levels.

3.3.2 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, Figure 6. 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.

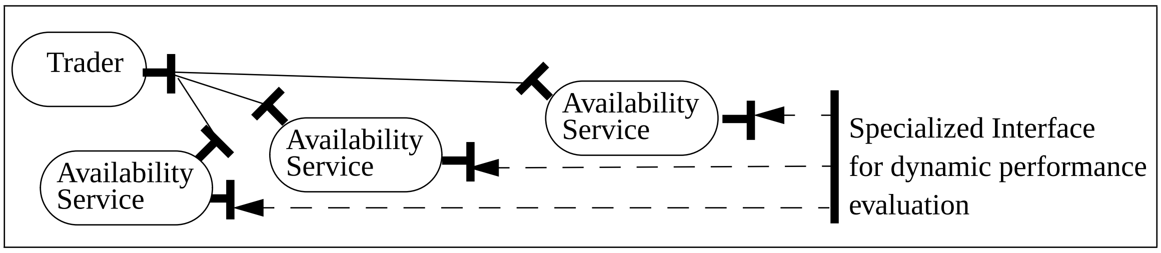

Figure 6. Interfaces on the availability service to the trader and for the competitive choice.

Availability Domain. The Availability Domain could be also called an 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. The Traders use information agents to get report on sets of dynamic parameters. The Trader may either dispatch an information agent to an agency or it may be part of the agency. Actually this information agent is closely related to the Availability service.

3.3.3 The Object Model

In Figure 7 there is an object model showing the relationship of the availability services to the mobility service and to other CORBAservices. CORBAservices are extensive and well specified [COS97] and here just represented as a whole but not detailed in the corresponding box. Specific CORBAservices like: Life Cycle, Persistence and Relationship, Query, Properties and Trader are explicitly represented as well as their association to the availability and mobility services. Not all of the CORBAservices are available and some custom implementation is needed in these cases. 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]. Each agency should have a local availability service. A transparency interface/service is sketched in the model but will not be discussed in more details here in, since the current work concentrates on the services behind the availability service and not on the interface seen by an external client.

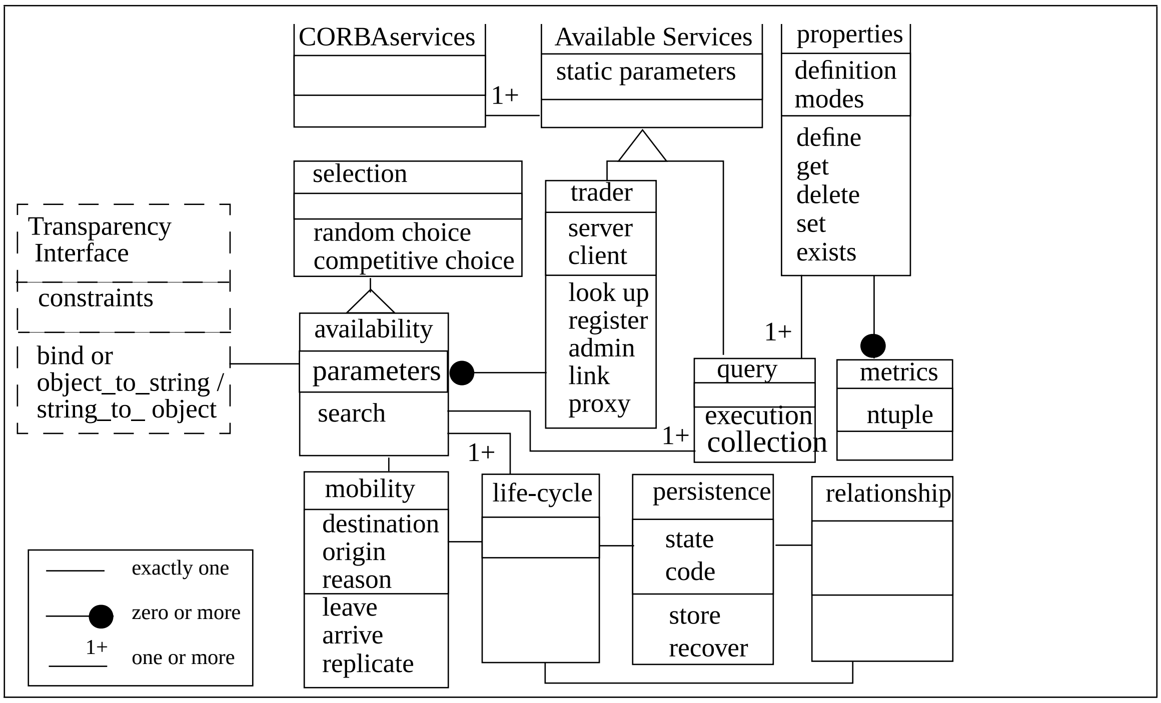

Figure 7. Availability and mobility object model based on OMT conventions [Rumbaugh91].

In this model two service were added to the CORBA facilities to provide infrastructure for transparent mobility: the Availability service and the Mobility service.

Availability: (server side)

attribute *parameters* :

description : resources to be searched

method *search* :

description: searches for the resources and binds to them.
strategy :
1.lookup Naming Service
2.lookup Intra Domain Resources
3.lookup Extra Domain Resources
4.Select
5.Bind to active instance

Mobility:

attribute *destination* :

description : new location of an agent. It may be an agency or a domain of agencies.

attribute *origin* :

description : previous location of an agent, in case of a callback. It may be an agency or a domain of agencies.

attribute *reason* :

description : a parameter indicating the kind of migration strategy. Fig 8 and 9 show two kind.

method *leave* :

description : finalizes the agent and sends it to the new location.

strategy :
Firstly,. no new request is attended.
Secondly, the agent reference is removed.
After, the reference is sent to the new destination.
Finally, the activation of the agent is requested at the new destination.

method *arrive* :

description : receives the reference and executes a constructor.
strategy :
Firstly, the agent reference is registered.
Secondly, the instance activation request is attended.
Finally, the constructor is executed to recover the state of the agent.

method *replicate* :

description : registers the agent reference on more then one destination. It can be seen as a specialization of the method leave.
strategy :
Firstly, no new request is attended.
Secondly, the agent reference is removed.
After, the reference is sent to several new destinations.
Finally, the activation of the agent is requested at one of the new destinations.

Figure 8 and 9 give some complementary details on the methods based on two different scenarios. The selection object either *randomly* chooses one out of a set or *competitively* chooses one out of a set of replicated services. The mobility service is related to other CORBA services like Life Cycle, Persistence and Relationship for the creation, migration, deletion, and storage (code, state and links).

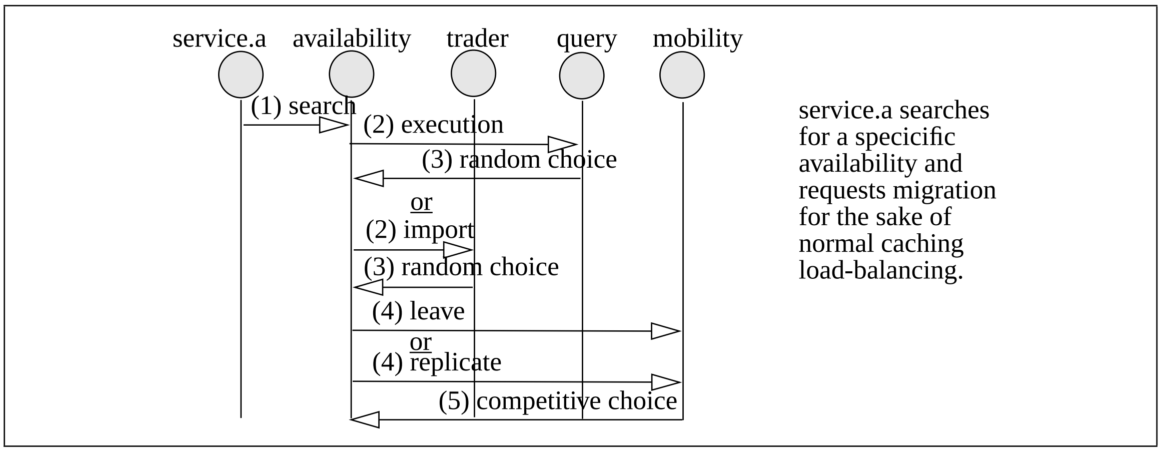

Figure 8. Event trace on a load-balancing scenario where normal caching takes place.

The scenario in Figure 8 is closely related to load-balancing where a *service.a* is searching for some specific resource(s) availability in order to request migration to where the resources are. The referred service requests the search method on the availability service which executes a query in the local domain or an import on the trader in case of not finding the resource(s). Either the query or the trader returns a list (of one or more) form which a random choice is done. If the selection involves minimizing latency then a competitive choice is allowed in case the task has the sufficient privileges. Competitive choice involves a request to the mobility support to replicate *service.a* on a set of hosts extracted from the returned list. Once *service.a* is replicated, a multicast is issued and the first prompting reply selected. The remaining replicas are either removed in case of a service single offer or maintained in case of service multi offer. Service multi offer allows for a QoS with a new call to Availability:CompetitiveChoice. When the QoS is not reached anymore then a new search of resources has to start all over again.

The scenario in Figure 9 is more related to inverse caching with *service.b* requesting to the mobility service that it wants to leave to where a certain resource is. The mobility service issues a search to the availability service to locate the resource(s). In this scenario, search and leave methods are different specializations from these methods in the previous scenario, benefiting from the polymorphism of the IDL language. In this case, there are no repeated calls to the availability service to optimize QoS.

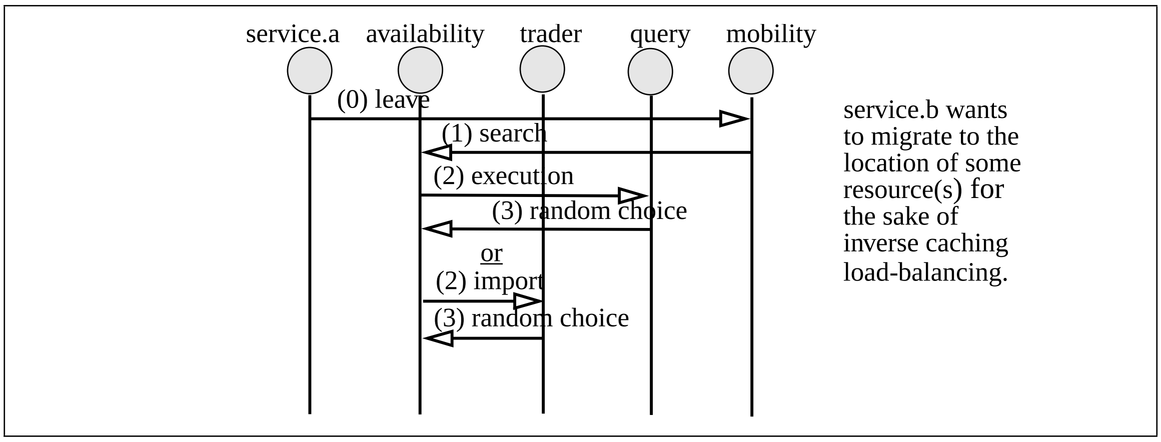

Figure 9. Event trace on a load-balancing scenario where inverse caching takes place.

Up to here, services can be identified in different phases during its life-cycle:

Start-up. This involves contracting and distribution as considered in any application.

Stationary. This phase of a service can be temporary or indefinite according to the characteristics of the service. Making services available for general usage involves management and distribution of these services in order to guarantee availability as much as possible. Assume these services as stationary as long there is no major problem with the network or the hosts on which they are running. Thinking of services as *always available* demands a natural need to make smooth moves in case of some failure in the environment.

Migration. This is demanded by the environment or the service itself and in attendance to load-balancing, inverse caching or redistribution due to some failure. Migration involves 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 sending site 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 do nothing and to go idle, this 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.

The mobility support should send/receive an agent, i.e., state and code. The protocol to send/receive an agent is actually based on forwarding its interface to the remote agencies and to register it there in order to allow further location. Code/class is moved effectively only when the agent activation occurs. Location and activation go over CORBA. No registry is actually need if the agent acts only as client.

Removal. This follows shutdown or migration of a service. A mobile agent may handle the proper termination of an application object.

Availability Evaluation. When a new service is going to be setup at some site, there is the need to locate and allocate resources on an agency. In order to identify agencies which are open to new services, the *availability service* is used.

The availability service evaluates the loading of an agency using the performance metrics included in the instrumentation facility [Queiroz97]: *response time*, *throughput*, and *utilization*. Utilization allows different parameters to evaluate loading in terms of: CPU, memory, disk, networking activity, and number of users / processes. These numbers are computed including the *specmark* of the particular host in order to allow a comparative value to other hosts. The availability level of the agency is published in order that this parameter can be obtained from a querying to the agency or via a *trader.*

An evaluation daemon being just started when there is an availability request is possible, however, availability has to consider a certain backtracking in time, reflecting the time the application will execute. Considering this approach, availability evaluation demands a continuous running daemon on every host which puts its resources available with a logging of the host loading history.

Trading Service. In case of querying via a trader [Lima95, ODP95], 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. An additional step at this point allows fine tuning, by using a customized agent to evaluate the agency more closely in case of a very sensible application. This can be added as a trading extended service at the trader side or at the application level itself.

Services distribution. Independent of the implementation repository type, services found unavailable, are (re)distributed according to demand on load balancing and / or inverse caching. This means that if distribution is not necessary in the current environment 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.

4. Results on Component Migration

In this section we explore a so called availability service as an additional support to implement transparency. The statement of a goal for transparency is expected for an optimal functionality. This goal is internalized and it participates in the process of resource allocation.

The implemented application treats 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 case of a actual failing host is more constraining because no pre-announcement is possible and a solution to this is explored as a general solution.

The strategy for an algorithm is based on the migration of the components on a mobile host to somewhere else on the network. It is assumed also the existence on the network of a (mirror) repository of the client's: state, implementation, interfaces and data. A mobile host disconnecting and reconnecting can be seen as failure recovery and a lot in this subject can be found under fault-tolerance[Cristian94-93-91]. The main issue of this work is on reallocation of resources and re-instantiation of services rather then on state persistence and the mirroring a consistent copy of it.

Under this reallocation of resources and re-instantiation of services, two phases are suggested for migrating a mobile host's components: an emergency phase and a recovery phase.

Emergency Phase.

1. The client issues an instantiation of a replica on the last server with which it has been in touch.
2. This instantiation may also be issued by this server in case of a long waiting for an acknowledge from the client.
3. To certify this instantiation, the server issues a cancellation request from the client.
4. If a cancellation does not come, the replica is instantiated locally on the current server.
5. The replica immediately starts attending any forthcoming request and enters the recovery phase.

Recovery Phase.

1. This replica is responsible for migrating to a new available host.
2. The replica searches for the original mobile host for reactivating the original client.
3. If the original mobile host is not responding, then an available host is selected at random from a set of hosts fitting the constraints on QoS.
4. The client is replicated on the first fetching the request.

Client/Server connection. A CORBA client uses a *_bind* call to connect to a CORBA server. Associated to the _bind call all the previous resource selection/allocation should happen. Several strategies are possible for the client retrying to migrate back to the mobile host. In this work the approach is to use every _bind as a checkpoint for a new tentative. The same checkpoint is used to try out for a new server.

Host selection procedure. Some QoS parameters like a range of expected execution cost may be useful to the host selection. The use of ranges should prune the selection of available host and reduce the chances of min./max. oscillations which could appear in case of a selection based on maximal availability.

An estimated range of execution cost can be translated into an availability range expressed in terms of the host's performance numbers (specmarks).

4.1. Implementation Details

The implementation results presented next are based on a Orbix CORBA 2 version running under SunOs 5.5.1 on sparc-5 hosts.Garbage collection is made by the CORBA environment.

Host Selection. The test results presented in this example are based on a simplified version of the *host selection procedure*, in the sense that an actual CORBA::*trader object* was not involved in the location process, but instead a list containing a separate set of possible hosts per service. When a _bind is issued to a service, a host is selected at random from the corresponding set. If the interrogated host does not respond, another host is selected at random out of the remaining subset. The protocol of the _bind call uses parameters as number of tentative and time out. Adjusting these two parameters is equivalent to adjusting the discrimination level of the interrogated host occupancy, i.e., availability.

Test Cycle. 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 10. The client can be either the *original* client or a *replica*.

The *original* client is instantiated 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.

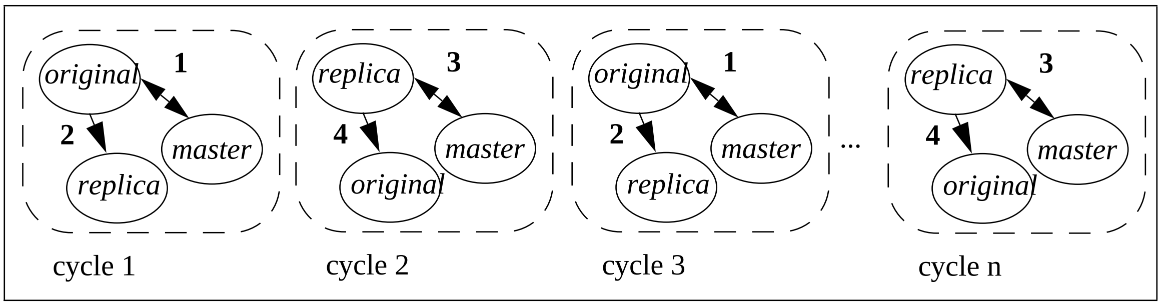

Figure 10. Sequence of instantiation and communication of test objects: 1, 2, 3, 4, 1, 2 and so on.

Interfaces. Following there is a description of the interfaces of the objects in Figure 10, in an object-oriented pseudo-code which is actually similar to IDL.

```
// Master objects'
        interface MasterObject_1 {
           oneway void any_method_a ();
           oneway void any_method_b ();
        }

        interface MasterObject_2 {
           long void any_method_aa ( in typedef var1aa, inout typedef var2aa, out typedef var3aa );
        }

// Original Client
        interface ClientO {
          oneway void main_method ();
          oneway void any_other_entry_point_method ();
        }

// Replica Client
        interface ClientR {
          oneway void main_method ();
          oneway void any_other_entry_point_method ();
        }
```

Implementation Repository. At Orbix, this is a directory where a service registration is stored. Three different approaches for the implementation repository are considered: Firstly, a single common place is used for all the hosts, so that the whole looks like a set of services with a multiprocessed ORB, i.e., multi-host service. Secondly, a separate implementation repositories is used for each host, so that the whole looks like a collection of single-host ORBs. Finally, a mixed approach of the other two, where different number of hosts are used per service. By a multi-host service we mean that several hosts are able to attend to a request of a specific service. The first host sufficiently unloaded will attend the request allowing an intrinsic load balance in the execution. The host configuration per service is represented in Figure 11. Domains may be used as an organization of the hosts into clusters of maximum number of multi-host services.

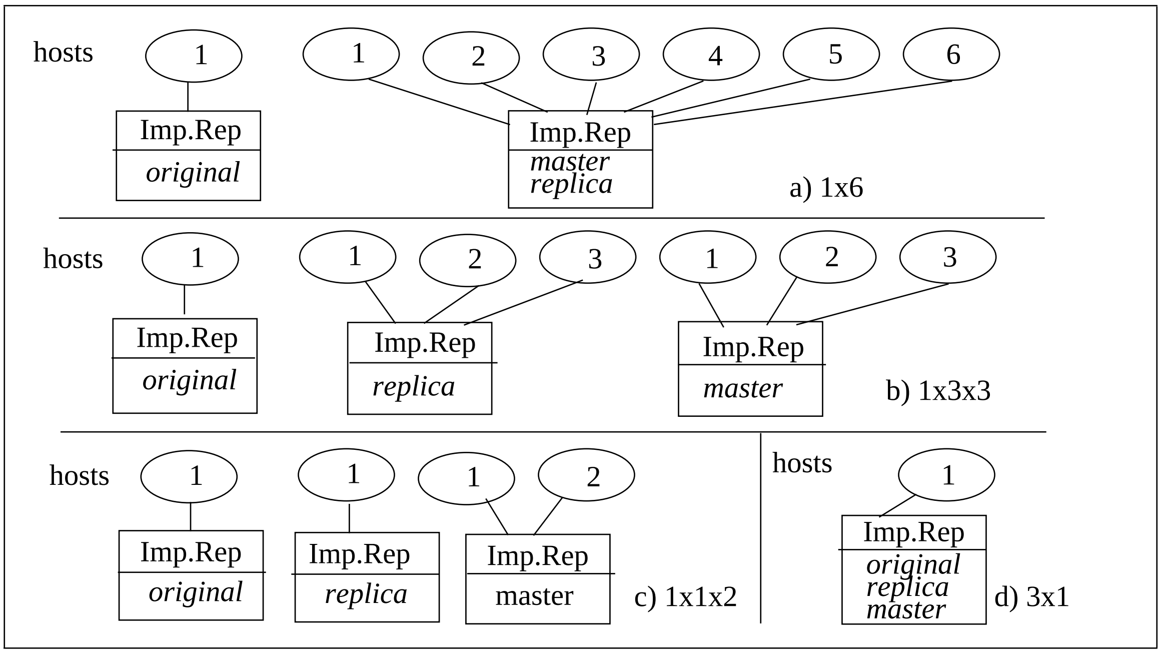

Figure 11. Different hosts allocation: a) 1 per original client and 6 per master and replica; b) 1 per original client, 3 per replica and 3 per master; c) 1 per each one; d) all on a single host. Configuration d) only as a comparative performance specmark between different machines.

Specmark. The execution performance is obtained with respect to a specmark based on the performance of all Client/Server processes executing on the same host. The specmark is obtained under different loading conditions: normal unloaded and extra loaded via a loading program. Faster inter process communication does not go through the network and for *ethernet*, the difference in transmission speed is about 50% higher for local Client/Server [Schulze97]. The load program demands the same resources as the specmark, in order to be an actual load from the point of view of the specmark. The values in Table 2 are obtained for the faster and slower host in use, as an estimate of upper and lower bounds of the distributed execution over several hosts. The specmark is not supposed to be a real situation but rather an estimate.

Table 2. Hosts' parameters and an average specmark on each individual machine.

host name	**Clock** (MHz)	**Memory** (MBytes)	**Comparative** Clock | Mem.	**Type**	**Configuration*** normal | w/load
itapoa	2x 60	96	1.7 | 3.0	sparcstation-20	3.5 | 4.8
aracati	110	64	1.6 | 2.0	sparcstation-5	|
tambau	110	64	1.6 | 2.0	sparcstation-5	|
pajussara	110	32	1.6 | 1.0	sparcstation-5	4.1 | ----
ilhabela	85	32	1.2 | 1.0	sparcstation-5	|
tutoia	40	32	0.6 | 1.0	axil-235	4.7 | ----
juquei	70	32	- 1.0 | 1.0 -	sparcstation-5	5.5 | 8.7

*specmark (Figure 11d)

The numbers in Table 3 are average values computed in different occasion selected at random, where a minimum and a maximum values are annotated. We presented the results on different loading conditions and the configurations of Figure 11. A range was obtained for normal loading and extra loading conditions plus an additional measurement for the configuration of Figure 11a with half of the hosts extra loaded. This value does fit quite well between the range obtained. Actually, the more hosts exist, better the chances of migrating to unloaded hosts. The results suggest that the average performance should be better with migrating distributed services rather than with distributed stationary services. The usage of migration of components in mobile computing is proposed in a similar way for the sake of sustaining execution of tasks and sustaining the whole availability of services in the environment.

Table 3. Test execution measurements with the configuration in Figure 11.

Configuration		**Results**	**(minutes/cycle)**	**Comments**
Figure 11	**normal**	**extra load**	**extra load**	
		half the hosts	all the hosts	
(a) 1 x 6	4.1 - 5.2*	5.4	10.4 - 11.5	*while this measurement, 1 (tambau) of the 6 hosts was failing
(b) 1 x 3 x 3	~5.			
(c) 1 x 1 x 2	~7.			
(d) 3 x 1	3.6 - 5.5	--	4.8 - 8.7	fastest and slowest host empiric specmark
(-) 1 x 1 x 6	~5.			1 original / 1 replica / 6 masters

Object Classes. Below there is a basic structuring of classes for each of the objects in Figure 11 presented in a object-oriented pseudo-code which is actually Java like.

```
// Master Class
       public class master {
         _MasterObject_1Ref newobject1 = null;
         _MasterObject_2Ref newobject2 = null;
         try {
              newobject1 = new MasterObject_1Impl();
              newobject2 = new MasterObject_2Impl();
             _CORBA.Orbix.impl_is_ready("any_name_for_master");
           } catch( Exception) {
           exception_handling;
           }
       }
       // MasterObject_1 implementation
       class MasterObject_1Impl extends _some_basic_class_of_MasterObject_1 {
         // constructor classes and methods of that class have to be stated here in //
       }
       // Same for MasterObject_2 implementation

// Client Classes
       public class cliento {
         _ClientORef newobject = null;
         try {
              newobject = new ClientOImpl();
             _CORBA.Orbix.impl_is_ready("any_name_for_cliento");
           } catch( Exception ) {
           exception_handling;
           }
       }

       // ClientO implementation
```

```
class ClientOImpl extends _some_basic_class_of_ClientO {
   // constructor methods of that class //
   public ClientOImpl() throws SystemException {
       super();
   }
   public final void main_method() {
      any_particular_method ();
   }
   public final any_other_entry_point_method { };

   // A desctructor finalize method requests the instantiation of new client
   public final void finalize() {
      _ClientRRef clientR;
      try { clientR = ClientR._bind("any_name_for_clientr", ""); }
      catch ( Exception ) {
         exception_handling; return;
      }
      try { clientR.main_method(); }
      catch ( Exception ) {
         exception_handling; return;
}}}
// Similar for clientR //
```

// *Common client implementation classes*

```
public class client {
   // typedef declarations
   public static void any_particular method () {
      _MasterObject_1Ref newobject1 = null;
      _MasterObject_2Ref newobject2 = null;

      try { newobject1 = MasterObject_1._bind("any_name_for_master", ""); }
      catch ( Exception ) {
         exception_handling;
      }
      try { MasterObject_1.any_method_x(); }
      catch ( Exception ) {
         exception_handling;
         return;
      }
      try { newobject1 = MasterObject_2._bind("any_name_for_master", ""); }
      catch ( Exception ) {
         exception_handling;
      }
      try { MasterObject_2.any_method_xx(); }
      catch ( Exception ) {
         exception_handling;
         return;
      }

      // Any other request to another server / service object
}}
```

5. Related Work and Techniques

Distributed Object Computing allows for a higher level of abstraction of programming in a service-oriented environment [ODP95, COF97, COS97, CORBA95, Orfali96]. DOC platforms suggest new perspectives and exploratory results on some previously explored techniques.

Migration is a technique well mentioned in the literature but in a world of agents and multi-agent, it gets some new approach and insight under agent's mobility. An explicit approach for mobility transparency is missing in the majority of related works [Cardozo93, Iglesias96, Krause96, Lange95, Mendes96, Russel95, Schulze97].

There is a lot of work on load-balancing techniques and its performance evaluation specially regarding CPU power. But it could be better explored as transparency of resource allocation in a highly reconfigurable environment. Transparency in the sense of overall load-balancing of computational resources, regarding for instance communication load, storage load and making use of QoS parameters [Ciupke96, Golubski96, Goldszmidt96].

Distribution Transparency according to the ODP reference model [RM-ODP] conceptually defines transparency for: Access, Failure, Location, Migration, Relocation, Replication, Persistence, and Transaction and it includes rules for selecting and combining them. Transparency should use a specific goal in order to search for different QoS parameters. A good example of a goal would be to use resources with little failures. Although fault-tolerance and recovery techniques are important to any distributed computing [Cristian94-93-91], there are different levels which can be translated into different QoS.

Highly reconfigurable environments can be found for instance in very profitable areas like telecommunications and not so profitable but very constraining ones like large scientific experimental facilities [DCEE97, Kouzes96, Schulze95].

6. Summary and Further Work

We introduce an availability service for transparently locating and selecting available resources, involving the trader service. The relationship to other CORBA object services are also presented. For instance, we propose the availability service working closely related to a mobility service in order to transparently migrate services. The approach should be specially suited for applications using load-balancing and inverse caching, i.e., code is moved close to data.

The availability service has a straight relation to the concepts present in the Trader [TOS96], the MOF [MOF97] and Naming Service [COS97]. It is a structuring and encapsulation of this services and concepts. In addition it establishes a connection to the desired service via a multicast to a cluster of hosts defined for the service. The communication considered in this work is asynchronous connection oriented.

The present work uses a DPE platform based on CORBA to simulate the migration of objects executing on a mobile host to any other host in the environment and later back to the mobile host when it is there again. The contribution here is the ability to handle a mobile host as if it were an ordinary host going down and where the functionality of the whole environment is maintained by migrating a mobile host's components. This migration is done transparently with specific mechanisms on top of a middleware using CORBA.

Different approaches to an availability service may be possible according to a particular goal of an application. We believe that the best approach is to add functionality to the availability interface where the goal for transparency is given, like for instance QoS in: performance, fault-tolerance, security, and image / sound resolution and definition. These parameters may determine also different strategies in the availability processing and final selection. For instance in load-balancing, the availability service has to be lightweight and quick. For this reason the tests above were made using some intrinsic evaluation of load, using the time response of the hosts attending each kind of services.

Another parameter has to be present in the above interface limiting the level of recursion in the process of recovering / replacing services which were not available. In other words, how many levels deep may the search process be allowed to. There would be some default value for this parameter but of course it has to accommodate some changing either in a flexible way reflecting the environment or interacting with the application/user, or even better the mixing of both. This mixing means that some values would be submitted to the application/user in order to be allowed and trusted. Trusted values go in to a repository for autonomous decision.

The presented simulation shows as a side result that a mean execution time can be sustained with the implemented strategy specially with a scale-up in the number of authorized hosts. The inclusion of a QoS range in the selection should help pruning this result. Another side contribution is that component migration and distribution are handled the same way.

No major point is made on cost of replication since any replication that might be needed consists only of a new registry of the service object interface. This registry is like a proxy containing information on activation mode and pointing to a general implementation repository where to find state and code (classes). This implementation repository address might be local or 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. Activation happens at a *_bind* call level.

There are some remarks on agent state persistence. Agents based on the same code (classes) might differ only on their state, and using a state repository they could be identified by their state. When replicating the reference of an agent, the state “file” has to be locked to only one agent active instance at each time. Some application might avoid persistent storage of agents’ state in order to improve performance.

A future work is the addition of a transparency interface/service to the local availability service which should be present at each agency. Location and migration transparency should be possible during a client call to a service via this interface.

Acknowledgments. This work is partially funded by: FAPESP, CAPES and CNPq.

References:

[APM93]. APM . *An Overview ANSAware 4.1*. Architecture Projects Management Ltd., Cambridge UK, 1993.
[Assis97]. F. M. de Assis Silva, S. Krause . *A Distributed Transaction Model Based on Mobile Agents*, Mobile Agents - MA97, LNCS #1219, Springer-Verlag, pp. 198-209, 1997.
[Bearman96]. M. Bearman . *Tutorial on ODP Trading Function*, DSTC, Faculty of Information Sciences & Engineering, University of Camberra, Australia, revised june 1996.
[Booch94] . G. Booch . *Object-Oriented Analysis and Design with Applications*. The Benjamin/ Cummins Publishing Company, Inc., 1994.
[Cardozo93]. E. Cardozo, J. S. Sichman, 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, www.omg.org.

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

[COS97]. OMG . *Corba Services*, 1997, www.omg.org.

[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.

[DCEE97]. W. E. Johnston, S. Sachs . *Distributed, Collaboratory Experiment Environments (DCEE) Program1: Overview and Final Report*, February 1997, www-itg.lbl.gov/DCEEpage/DCEE_Overview.html#1021081.

[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.

[Iglesias96]. C. Iglesias, J. C. Gonzalez, J. R. Velasco . *MIX: A General Purpose Multiagent Architecture*, LNAI #1037 Springer-Verlag, pp. 251-266, 1996.

[Kouzes96]. R. T. Kouzes, J. Myers, W. A. Wulf . *Collaboratories: Doing Science On The Internet*, IEEE Computer, Vol.29 #8, August 1996.

[Kramer85]. J. Kramer, J. Magee . *Dynamic configuration for distributed systems*, IEEE Transactions on Software Engineering, SE-11 (4), April 1985.

[Kramer90]. J. Kramer . *Configuration Programming - A framework for the development of Distributed Systems*, Proceedings of the IEEE COMPEURO´90, Tel-Aviv, Israel, pp. 374-384, May 1990.

[Kramer92]. J. Kramer, J. Magee, M. Sloman. *Configuring Distributed Systems*, Proc.5th ACM SIGOPS Workshop on Models and Paradigms for Distributed Systems Structuring, September 1992.

[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.

[Loyolla94]. W. P. C. Loyolla, E. R. M. Madeira, M. J.Mendes, E. Cardozo, M. F. Magalhães . *Multiware Platform: An Open Distributed Environment for Multimedia Cooperative Applications*, IEEE COMPSAC'94, Taipei, Taiwan, November, 1994.

[MASIF97]. GMD FOKUS, IBM Corporation, Mobile Agent System Interoperability Facilities Specification, OMG TC orbos/97-10-05, November 10, 1997.

[Mendes96]. M. J. Mendes et alli . *Agents Skills and their roles in mobile computing and personal communications*, IFIP 14th World Computer Congress, World Conf. on Mobile Communications, Canberra, Australia, September, 1996.

[MOF97]. DSTC . *CORBA Component Model: Initial Submission*, OMG Document orbos/97-11-07, 10 November 1997, www.omg.org.

[ODP95]. ODP . *Trading Functions*, ISO/IEC JTC1/SC 21, June 1995, ftp.dstc.edu.au/pub/arch/RM-ODP.

[Orbix96]. IONA Technologies, Ltd. . *OrbixTalk: Management Overview*, April 96, www.iona.com/Orbix/Talk/MO/CorbaIntro.html.

[Orfali96]. R. Orfali, Dan Harkey, J. Edwards . *The Essential Distributed Objects Survival Guide*, John Wiley & Sons, 1996.

[Queiroz97] . A. Queiroz, E. R. M. Madeira . *Management of CORBA objects monitoring for the Multiware platform*, Proc.ICODP'97, Toronto, Canada, pp. 122-133, May 1997.

[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.

[Rumbaugh91] . J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen . *Object-Oriented Modeling and Design*, Prentice Hall, 1991.

[Russel95]. S. Russel, P. Norvig . *Artificial Intelligence, A Modern Approach*, Prentice Hall Series in Artificial Intelligence, New Jersey, pp. 33, USA, 1995.

[Schmidt95]. D. C. Schmidt, S. Vinoski . *Object Interconnections: Introduction to Distributed Object Computing (Column1)*, C++ Report Magazine, January 1995.

[Schulze95]. B. Schulze et alli (DELPHI Trigger Group) . *Architecture and performance of the DELPHI trigger system*, Nuclear Instruments and Methods in Physics Research A 362, pp. 361-385, 1995.

[Schulze97]. B. Schulze, E. R. M. Madeira . *Contracting and Moving Agents in Distributed Applications Based on a Service-Oriented Architecture*, Mobile Agents, 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.

[Sloman89]. M. Sloman, J. Kramer, J. Magee. *Configuration Support for System Description, Construction and Evolution*, Proc.5th Int.Workshop on Software Specification and Design, Pittsburg, May 1989.

[TOS96]. OMG . *Trader Object Service*, RFP5 Submission, May 10, 1996.

[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.