Using an Availability Service for Transparent Service Migration

B.Schulze E.R.M.Madeira

Relatório Técnico IC - 97 - 11

Agosto de 1997

Using an Availability Service for Transparent Service Migration in Mobile Computing

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

Abstract. This paper details an availability service in a service-oriented platform based on OMG/CORBA, for transparent migrating of services, i.e., components or agents. Migration of services is used to move components executing on a mobile host to another host, in case of shutdown or disconnection. A mobile host is treated as any other host going down or failing in an environment where the availability and functionality of the services and tasks wants to be sustained.Transparent location of a new destination host is simplified to the search and contracting of available resources from other hosts.

Keywords. mobility, service-oriented architecture, availability service.

1. Introduction

A mobile host may need to disconnect from the stationary network, for example, because of changing communication base station, power saving, etc. 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.

This work proposes the migration of tasks from mobile hosts to any other host in the environment, whenever a mobile host disconnects. This migration is handled transparently over a DPE (Distributed Processing Environment) platform [11] with the usage of an availability service to transparently locate resources on other possible destination hosts. After location, a selection function is needed to efficiently choose the new destination host where to send code and status of the related components.

Brief description of the following sections:
Section 2, shortly describes the service-oriented architecture. Section 3, describes a proposed *availability* service, while in Section 4 this availability service is used for transparent mobility, i.e., migration of components from/back_to a mobile host. Section 5, presents some implementation details and Section 6 briefly remarks some related work. Section 7 contains concluding remarks.

2. Service Oriented Architecture

The service-oriented architecture is based on distributed active objects using OMG/ CORBA(Common Object Request Broker Architecture) as a broker for object components [12] either in a stationary phase or in a mobile phase [1]. The general concept of service is associated to active objects or agents as services already available or in the process of being put available available somewhere.

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

Transparent location of components by an application, 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 transparent code distribution at application start-up.

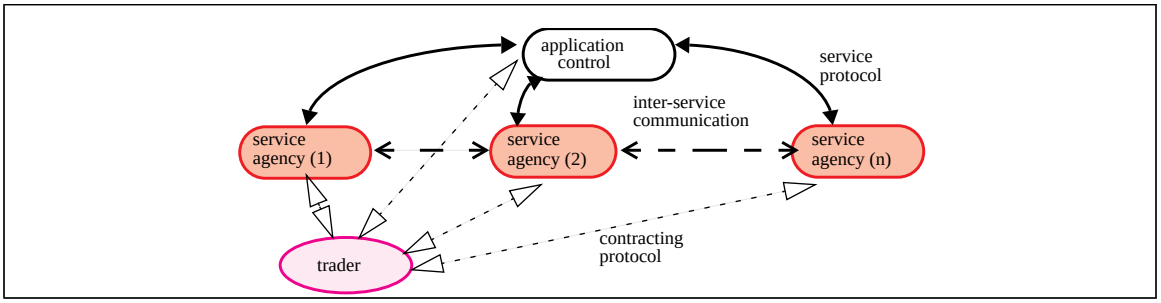

Fig. 1. Service-Oriented Application.

The diagram of Fig. 1 illustrates a service-oriented application and some basic blocks:
- *agents* are all kind of services used by an application
  ...*available* services are offered by an agency;
  ...*non-available* services are customized by the application at some site and afterwards treated as an available services;
- *agency* is a basic component able to offer services to an application;

- *negotiation* of services is handled by a *trader* [9,6];
- *trader* is yet another service for locating other services in a pool of contracted agencies.

2.1. Services

Computing with services allows a higher level of abstraction in implementing any application 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*.
Available services:
Can be co-processors, databases, data crunching, specialized processing, etc.
Non-available services:
Of the same kind as above, but for some reason it is just not available. Non-availability may come from:

- service access non-authorized;
- service is not where needed;
- service is temporarily disconnected;
- service is a too specific computation and has to be customized by the application.

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

2.2. 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.

Up to here services can be identified in different phases during its life-cycle:
Start-up:
It involves contracting and distribution like considered for 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. One can think of these services as stationary most of the time as long there is no major problem with the network or host on which these services are running. But thinking of services as *always available* demands a natural need to make smooth moves in case of some failure in

the environment.
Migration:
It 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 status, i.e., before moving the agent has to save the variables defining its status and persistently store them. Both, status and code, are moved as sequences and 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 instantiated, it reads back its status into the original variables.

In all situations, after the agent´s arrival, it is instantiated by the *agent support.* This is done in order to recover from the status file the memory on what the agent has to do. If it has just to do nothing and go idle that is coded in the status.
Removal:
It follows shutdown or migration of a service. In this phase, there is the possibility of using a migratory agent passed as a token in order to handle any application termination and proper shutdown. A token agent is composed of code and data.
Mobility:
Mobility service supports the reception of an agent, its persistent storage and the registration of its interface on the ORB.
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, another service is included in the agency itself: an *availability service* which informs the level of availability of the agency.

The availability service evaluates the loading of an agency using the performance metrics included in the instrumentation facility [13]: *response time*, *throughput*, and *utilization*. Utilization allows different parameters to evaluate loading in terms of: CPU, memory, disk, networking activity, 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.*

One can think of an evaluation process or daemon just being started when there is an availability request, 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 [6,9], 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.

Unavailable services distribution:

Independently of the kind of implementation repository, the new services are redistributed according to the application's demand on load balancing and / or inverse caching. This means that if distribution is not demanded by the application in the current environment circumstances, then it runs just locally. Another possibility is that the application has to wait in the start-up queue for sufficient resources.

Fig. 2 illustrates the multiware platform used in this work.

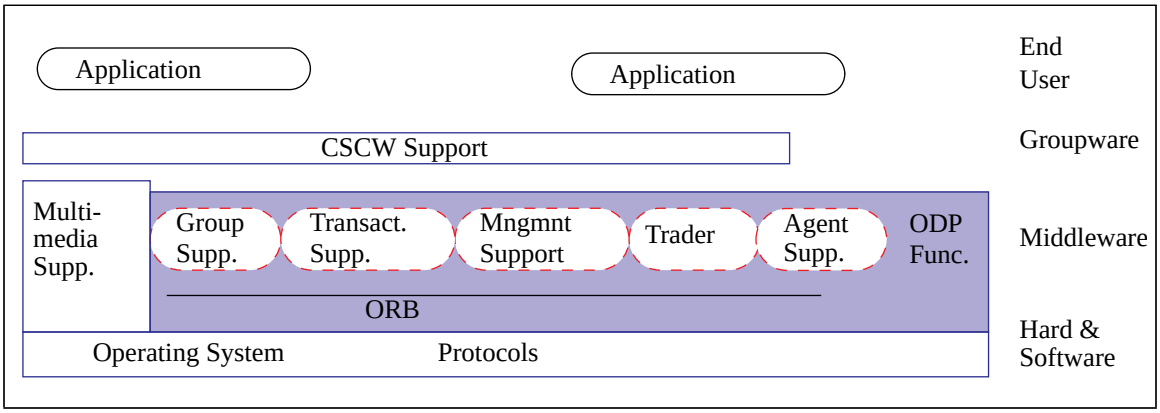

Fig. 2. Multiware Platform.

3. The Availability Service

In a previous [1] work, the concept of an availability service altogether with a code mobility support is presented in order to transparently move agents or services according to load-balancing and inverse caching [3] criterias. Refer also to Fig. 2.

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 [3] 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 going to be setup at some other site or agency. In order to identify an agency open to receive a new service, the *availability service* informs the level of availability of each agency. The availability level of each agency is published so that it can be obtained from a querying to the agency or via a *trader.*

Availability should consider a certain backtracking in time, reflecting the time the application will execute. This demands a daemon logging the loading history on every host where resources are put available.

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 specific types like management, security, etc..., according to Table.1. Specialized types inherit the basic ones.

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

Table 1. Availability service type offers.

Availability allows different parameters to evaluate, for instance, loading in terms of: CPU, memory, disk, networking activity, number of users / processes, etc. The final selection function may either compute these numbers with a *specmark* of the particular host to take a decision or an implicit computation this numbers can be used. Implicit computation means an activation function like, where a response to a query may imply in availability.

3.2. Availability Querying

Availability querying via trader [6,9] 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, Fig. 4. 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.

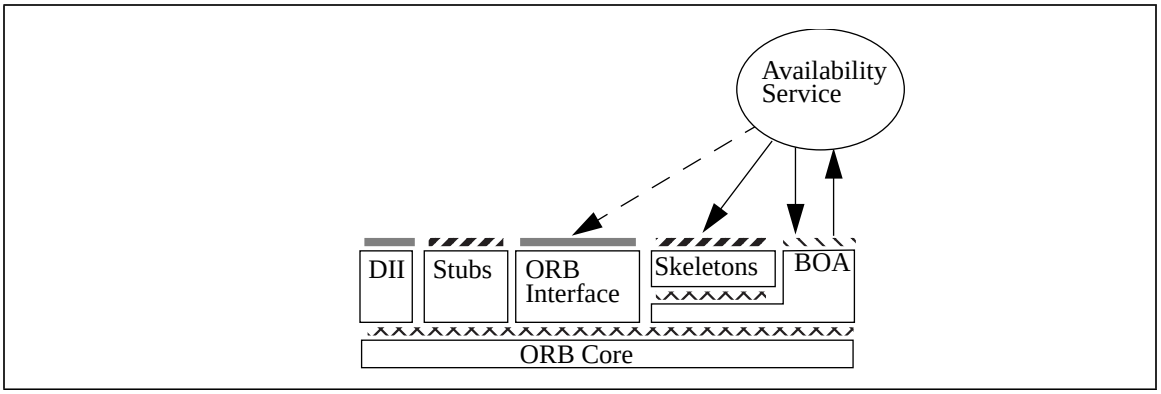

Fig. 3. Availability Service on top of CORBA.

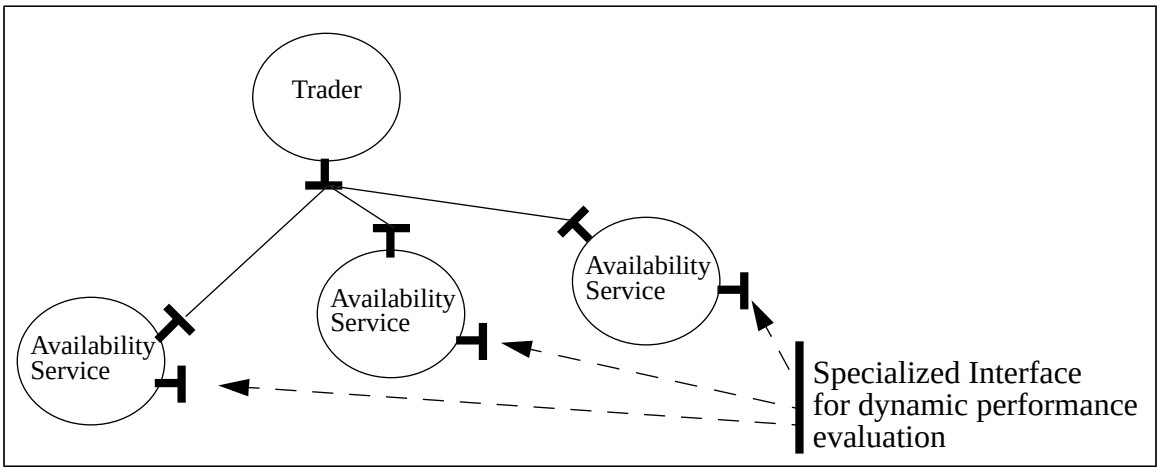

Fig. 4. Availability Service interfaces.

4. Component migration from a mobile host

A mobile host disconnecting from the static cluster of hosts is quite similar to a host going unavailable, i.e., either going down, loosing performance or failing. In these situations the client has to migrate to another available host on the network.

In the situations above, the case of a real failing host is more constraining because of no pre-announcement and a solution to this situation is explored as a general solution to all of them.

4.1. Algorithm

The approach is to keep a mirror of the client's: status, data, implementation repository and interfaces repository. Since it is an emergency situation of failure there is a two phase algorithm with an emergency phase and a recovering phase.

Emergency Phase:

1. The failing client issues an instantiation of a replicated client on the last server it has been in touch.
2. This instantiation can also be issued by this server in case of a long time waiting for a reply from the client.
3. To certificate this instantiation, the server issues a cancelation request from the client.
4. If a cancelation does not come, the replicated client is instantiated locally at the current server.
5. The replicated client immediately starts to attend to any forthcoming request and enters the recovering phase.

Recovering Phase:

1. this replicated client is responsible for migrating to a new available host;
2. the replicated client starts a search for the best host to stay;
3. the best host to stay is the original mobile host followed by any other well available one;
4. some available host is selected at random from a list of hosts fitting the constraints of quality of service;
5. the client is replicated on these selected hosts and
6. the client 1st fetching the next request survives while the others are aborted.

Crash Recover:

1. When the mobile host is not responding for recovery, a new request is issued, followed by a request for a new replicated copy to be started on another stationary host.

4.2. Host selection procedure

For the host selection, an estimated range of *quality* expected for the whole distributed execution is needed. This range should allow a more flexible choice in the final selection of the available host and reduce the chances of min./max oscillations which can appear if using a maximal availability choice. The use of a range also prunes the selection process.

For the above estimate, an execution complexity is worked out together with the percentage of availability of a host and its performance numbers. The performance numbers have to be in accordance to the usage of the different processors and co-processors.

5. Implementation Details

The current mechanism was tried out on a Orbix CORBA 2 version running under Solaris 5.5.1 on spark5 hosts. Some improvements have still to be done.

5.1. Multiprocessed Services

Some different basic approaches are possible in the configuration of the implementation repository of the ORBs:

• 1st. One common file can be used for all the ORBs, so that the whole looks like a service with a multiprocessed ORB or just as a multiprocessed service.
• 2nd. Separate implementation repositories are used for each ORB so that the whole looks like a collection of single processed ORBs.
• 3rd. A mixed approach of the two above is possible so that some services look like multiprocessed services while others only monoprocessed.

A multiprocessed 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 hosts' domains may be used as an organization of the hosts into clusters of maximum number of multiprocessed services.

5.2. Prototype Testing

The evaluation test is based on a cyclic execution of a set of commands involving communication and processing loading of the hosts. This cyclic test is composed of a general *master* server and a client, which can be either the primary client or its replica. The primary client is instantiated always on a pre-determined host, while the replica and the *master* server keep moving around for load balancing, always selecting the first prompting host as the next host to move to.

The test sequence and configuration are represented in the next 2 figures.

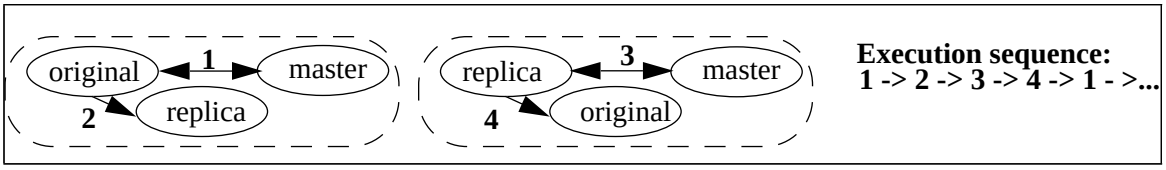

Fig. 5. Sequence of instantiation and communication of objects.

host name	Clock (MHz)	Memory (MBytes)	Comparative Clock | Mem.	3x1 Specmark* normal | w/load	Type
itapoa	2x 60	96	1.7 | 3.0	3.5 | 4.8	SPARCstation-20
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	4.1 | ----	SPARCstation-5
ilhabela	85	32	1.2 | 1.0	|	SPARCstation-5
juquei	70	32	> 1.0 | 1.0 <	5.5 | 8.7	SPARCstation-5
tutoia	40	32	0.6 | 1.0	4.7 | ----	Axil-235

***normalized #**

Table 2. Hosts' parameters.

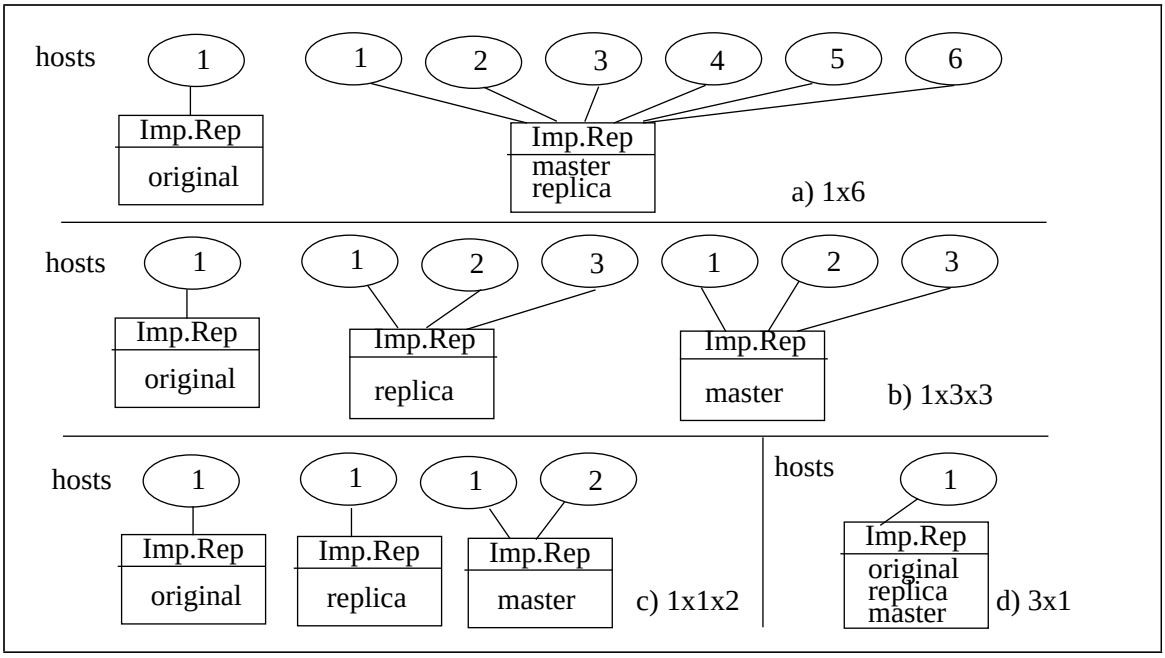

Fig. 6. Different hosts allocation: a) 1 per original client and 6 per master and replica; b) 1 per original client, 3 per replica and 3 per master; c) 1 per each one; d) all on a single host.

The execution performance is obtained with respect to a specmark based on the execution performance of all client/server processes running on the same host. This specmark is obtained for two different loading conditions, i.e, normal unloaded and explicitly loaded with a load program.

The load program demands the same resources that the performance program, so that it is seen as an actual load from the point of view of the performance program. This specmark, in Table 2, is also obtained for the higher and lower end of the host being used, such that a lower bound and an upper bound is obtained for the execution performance distributed on several hosts.

Configuration (Fig.6)	**Results normal**	**(min/cycle) w/load** half the hosts	**w/load** all the hosts	**Comments**
(a) 1 x 6	4.1 - 5.2*	5.4	10.4 - 11.5	-> *with tambau, one of the 6 host failing
(b) 1 x 3 x 3	~5.			
(c) 1 x 1 x 2	~7.			
(-) 1 x 1 x 6	~5 .			-> 1 original / 1 replica / 6 masters
(d) 3 x 1	3.6 - 5.5	--	4.8 - 8.7	-> fastest and the slowest host empiric specmark

Table 3. Execution measurement.

This 3x1 specmark is not an actual situation, and in addition to that, it allows faster inter

process communication, since communication does not go across the network. For ethernet, this difference in transmission speed is about 50% higher for local client/server [1].

The numbers in Table 3 are avarage values computed in different occasion selected at random. where a minimum and a maximum values are anotated.

These results in Table 3 suggest that there was no significant loss in performance with the use of migration. Actually, in an environment with many hosts there are always good chances of unloaded hosts being available and the average performance with migration should be higher than with stationary services.

For mobile computing, the usage of migration of components applies in a similar way and the advantage of it is the sustaining execution of tasks and the whole availability of services in the environment.

6. Related works

Related works are: migration, load-balancing, code/status mobility, distributed object computing (DOC) platforms.

Migration has already been studied and presented in many documents as well as load-balancing techniques and its performance evaluation [16,17,3,], while mobility has got some new order with recent works on agents and multi-agents [1,8,5,4,2,14] in highly reconfigurable environments as for instance in telecommunications and large scientific experimental facilities [15].

The usage of DOC platforms [9,10,11,12] opens new perspectives and results for these techniques specially if applied to the achievement of a higher level of abstraction in programming using the paradigm of service-oriented architectures.

7. Conclusion

The present work uses a DPE platform based on CORBA to implement the migration of components from a mobile host to any other host in the environment and later on it’s migration back to the mobile host, as soon as it is available again. Transparent migration is possible with the use of an availability service to identify a new destination and a code mobility support using sequence passing to dispatch the components to the destination host.

The main contribution is the handling of a mobile host as a usual host going down and where the functionality of the whole environment is kept by migrating mobile host’s components. This migration is done transparently with specific mechanisms added to middleware using CORBA.

Different approaches for an availability service are possible, according to every particular application, specially if performance is being taken into account. An extension of this work is the classification of general purpose availability interfaces and methods as well as specialized ones with particular interest in system management of open distributed processing.

For load-balancing, the availability service has to be quick and light processing. For this reason the tests above where made using some intrinsic evaluation of load, using the time response of the hosts attending each kind of services.

The results show that a mean execution time can be sustained and with a scale-up in the number of hosts. A preliminary selection is needed based on a query to the availability service where a search is made for host available inside a certain range. With this smaller list it is again possible to use a selection based on the previous approach of intrinsic availability evaluation.

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

References:

1. B.Schulze and E.R.M.Madeira, *Contracting and Moving Agents in Distributed Applications Based on a Service-Oriented Architecture*, to be published on the proceedings of the 1st Workshop on Mobile Agents - MA97, Berlin, Germany, 7-8 April 97.
2. E. Cardozo, J.S. Sichman and Y. Demazeau. *Using the Active Object Model to Implement Multi-Agents Systems*, Proceeding of the 5th IEEE Conference on Tools with Artificial Intelligence, Boston, USA, pp 70-77, November 93.
3. G.S. Goldszmidt. *Distributed Management by Delegation*, Ph.D. Thesis, Graduate School of Arts and Sciences, Columbia University, US, 96.
4. C.Iglesias, J.C.Gonzalez and J.R.Velasco, *MIX: A General Purpose Multiagent Architecture*, LNAI #1037 Springer-Verlag, pp 251-266, 96.
5. S.Krause and T. Magedanz. *Mobile Service Agents enabling "Intelligence on Demand" in Telecommunications*, IEEE GLOBECOM'96, London, UK, pp 78-84, November 96.
6. L.A.P. Lima Jr. and E.R.M. Madeira. *A Model for a Federative Trader*, Open Distributed Processing: Experiences with Distributed Environment, pp.173-184, Chapman&Hall, 95.
7. W.P.C. Loyolla, E.R.M. Madeira, M.J Mendes, E. Cardozo and M.F. Magalhães. *Multiware Platform: An Open Distributed Environment for Multimedia Cooperative Applications*, IEEE COMPSAC'94, Taipei, Taiwan, November, 94.
8. M.J Mendes et al. *Agents Skills and their roles in mobile computing and personal communications*, IFIP 14th World Computer Congress, World Conference on Mobile Communications, Canberra, Australia, September, 96.
9. ODP. *Trading Functions*, ISO/IEC JTC1/SC 21, June, 95, ftp.dstc.edu.au/pub/arch/RM-ODP.

10. OMG. *Common Facilities Architecture, Rev. 4.0*, OMG Document # 95-1-2, January, 95.
11. OMG. *The Common Object Request Broker: Architecture and Specification*, rev 2.0, July, 95.
12. R.Orfali, Dan Harkey and J.Edwards. *The Essential Distributed Objects Survival Guide*, John Wiley & Sons, 96.
13. A. Queiroz and E.R.M. Madeira. *Management of CORBA objects monitoring for the Multiware platform*, ICODP'97, Toronto, Canada, May97, accepted for publication.
14. S. Russel, P. Norvig. *Artificial Intelligence, A Modern Approach*, Prentice Hall Series in Artificial Intelligence, New Jersey, pp 33, USA, 95.
15. DELPHI Trigger Group. *Architecture and performance of the DELPHI trigger system*, Nuclear Instruments and Methods in Physics Research A 362, pp 361-385, 95.
16. O. Ciupke, D. A. Kottmann and H-D. Walter. *Object Migration in Non-Monolithic Distributed Applications*, proceedings of the 16th Int. Conf. on Distributed Computing and Systems, Hong Kong, May 27-30, 96, pg.529-536.
17. W. Golubski, D. Lammers and W-M. Lippe. *Theoretical and Empirical Results on Dynamic Load Balancing in an Object-Based Distributed Environment*, proceedings of the 16th Int. Conf. on Distributed Computing and Systems, Hong Kong, May 27-30, 96., pg.537-544.