Decentralized Service Deployment for Collaborative Environments

Daniel Lázaro Iglesias
Open University of Catalonia
dlazaroi@uoc.edu

Joan Manuel Marquès i Puig
Open University of Catalonia
jmarquesp@uoc.edu

Josep Jorba Esteve
Open University of Catalonia
jjorbae@uoc.edu

Abstract

In this paper we present the design of a system which allows service deployment in a small-sized group of computers distributed through the Internet. These groups are formed by users who share a common interest, and voluntarily yield their own resources for the achievement of the collaborative activities of the group. Having enough resources contributed by the members of the group, our system guarantees service availability and the fact that the deployment and execution of the services is carried out using only the resources of the group. This management is done in a completely decentralized manner, and the system is self-organized in the presence of component connections, disconnections and failures. We demonstrate its validity through the implementation of a prototype and the execution of some tests, which allow us to guarantee that the system is self-organized, always reaching a consistent state.

1. Introduction

When a group of people formed rather spontaneously (informal school associations, people with similar interests, campaigns for social and political activists) uses internet to carry out collaborative activities, usually they won't have supporting entities that automatically and transparently guarantee the necessary resources. These group members must thus collaborate using applications that only partially meet their needs (such as email), by having a few members manage resources for the whole group, or by paying for third-party resources or accepting advertising.

In this kind of environments, it would be desirable that the members of the group could collaborate sharing their resources. Each user should be able to provide resources so that they can be used for the benefit of the group. In this type of group the members share some common motivations and want their collaborative activities to be achieved, so they'll be willing to provide their own resources. On the other hand, as they're individual users with limited resources, variations of the capacity and availability of these resources over time can be expected, either because users disconnect, either because of components' failures, etc. Therefore, a system for managing the use of own resources in a collective way is required.

LaCOLLA [1, 2] is a middleware that allows the members of a small-sized group this kind of collaboration. Following the peer-to-peer paradigm, the group works in a decentralized manner, coordinating the resources and the applications contributed to the group by its members to allow collaboration. It basically offers general purpose functionalities that user applications can use to carry on their collaborative activities. Specifically, the middleware offers presence information, location transparency, object storage and communication between members of a group by disseminating events, which inform all the members of an action that occurred in the group, and messages, sent to a subset of participants.

When someone wants to offer a service (i.e. any application that receives a query and returns an answer) using the resources provided by the participants, whether it's an internal service for the use of the own system or one that a user wants to offer to the rest of the members of the group, its implementation should deal with all the difficulties derived from decentralized systems. A possible way to avoid this would be to extend the system with the ability to deploy a service within a group. The middleware would guarantee that the service is always active and reachable, using only resources provided by group members. This way, the service could be designed and implemented without having to manage its own decentralization, and would work in our decentralized environment. This would allow users to deploy services they want to offer but lack the resources to do it (for example, a web server), as well as provide an internal service that can be used by applications.

This paper extends the LaCOLLA middleware available at http://lacolla.uoc.edu/lacolla, and presents a system that allows the deployment of stateless services in a group of computers scattered across the Internet, in a decentralized manner, so that these services are always available as long as the resources provided to the group permit it.

2. Related work

There are systems which share some characteristics with the one presented in this paper, either in their objectives or in the environment they work in. No information has been found, though, about any system offering the same functionality of deployment of services in a network in a decentralized way, in an environment similar to the one considered in this proposal.

For example, there are a certain number of distributed computing systems, which allow the users to submit a task to be executed in a cluster of machines and get the result. Most of them act in a centralized way, with a component in charge of distributing tasks and a set of machines that can execute these tasks. This structure can be made more complex, adding hierarchies to distribute tasks among several sets of computing nodes. Examples of these systems are Condor [3], Torque [12] and Sun Grid Engine [9].

We can also find decentralized versions of this kind of systems. For example, JNGI [4] uses JXTA[5] to offer distributed computation in a decentralized manner by replicating the component which distributes the tasks. This resembles more our approach, as we will also need a replicated kind of component to distribute the services among the executing nodes.

Regarding the deployment of services, we must mention grid systems, which virtualize resources and offer them to the community in the form of services. However, they are complex to manage and are not usually prepared to deal with high levels of churn.

Finally, we must mention that our system makes use, when possible, of optimistic replication techniques [6], which are also used in systems like Bayou [7]. These techniques are used to share data in an efficient manner in wide area or mobile environments. Specifically, LaCOLLA uses Golding's time-stamped anti-entropy (TSAE) protocol [8].

3. Requirements

When defining the requirements for the design of service deployment, we must consider both the general requirements previously defined for collaboration and the concrete objectives of service deployment. Therefore, the requirements for LaCOLLA [2] also apply to this proposal. Specifically, we must consider that the system is addressed to small groups, which may be typically composed of 10 or 20 members. The main requirements considered here are the following:

- *Group self-sufficiency:* A group should not depend on external resources. Both the execution of services and the deployment management should be performed using only the resources contributed to the group by its members.
- *Decentralization and self-organization:* In case of connections, disconnections and failures, the system should keep functioning (it shouldn't have a single point of failure) and should reorganize without requiring any external intervention, getting to a consistent state as soon as the available resources and group stability allow it.
- *Individual autonomy:* The group's members should be free to decide which actions to carry out, what resources and services to provide, and when to connect or disconnect.

The requirements list is extended with the following:

- *Service availability:* A service activated by a user should always be available as long as there are enough resources to execute it in the group. In order to achieve this, given that a service has been activated, we must consider the following aspects:
  - *Replication:* A service might be replicated to improve availability. The desired number of replicas for a service will be included in its specification. The system will try to keep that exact number of replicas, as long as there are enough available resources.
  - *Location transparency:* The system resolves service locations transparently, and applications access them using a location-independent identifier.

4. Architecture

The architecture of LaCOLLA [2] consists of five kinds of component. Each one assumes a responsibility in service deployment. Users can instantiate the components they want to, and this determines which resources they provide to the group. This decision will be based on their degree of involvement in the group as well as their computers' capacity and availability.

- *User Agent* (UA): Allows the interaction between applications and the system, and represents the users in the group. It allows users to create, activate, stop and access services.
- *Repository Agent* (RA): Is in charge of storing the necessary files for service execution (executables, configuration files, etc.).
- *Group Administration and Presence Agent* (GAPA): Controls the access of members to the group, and authenticates users. It acts as entry point to the system.
- *Task Dispatcher Agent* (TDA): Stores information about the services of the group. It is in charge of keeping them active and assign them to executing nodes.

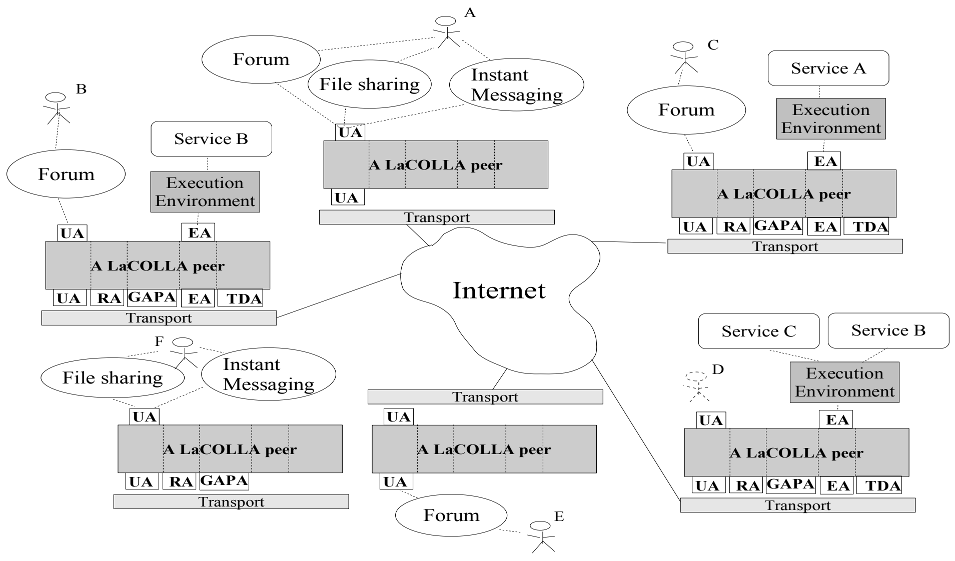


Figure 1. Snapshot of a collaborative group using services deployed within the group.

- *Executor Agent* (EA): Is in charge of the execution of tasks and services.

We decided to separate the execution environments (where the services will run) from the middleware (Fig. 1). The EA is in charge of communicating LaCOLLA and the execution environment, and provides an interface between them. This way, several environments with different characteristics can be connected to LaCOLLA without requiring any modification to the system. This allows services written in different languages and for different operating systems.

Table 1 shows the part of the API related to service deployment. Deletion and modification of services and exception passing have not been implemented because they aren't part of the minimum subset essential to prove the viability of the proposal. Once demonstrated the feasibility of our design, it will be improved with the implementation of these functionalities.

5. Mechanisms

We can distinguish two kinds of mechanisms. First, the ones related to the persistent information about deployed services. These must guarantee that information about the created services is stored in a persistent and replicated way, so that it is always available and will not be lost. The second kind of mechanisms includes those related to the management of running services. The information they manage is not persistently stored, and it potentially changes quickly (e.g. location, number of replicas, etc), as they are in charge of keeping the corresponding number of active replicas of a service in the adequate nodes, ensuring its accessibility in such a dynamic environment as long as there are enough available resources.

5.1. Persistent information

The first type of mechanisms includes service creation and the propagation of this information inside the group. The components in charge of storing it persistently are the TDAs. UAs must have it cached. This requires the presence of at least one TDA connected to the group to create, activate and access services. These mechanisms use optimistic replication techniques. Propagation is accomplished in two ways:

- *Multicast:* Whenever a new information is produced (i.e., a service is created), a message is sent, using an application-layer multicast, to all the interested components (i.e., UAs and TDAs) connected, which store the information. This multicast has a moderated cost due to the small size of the groups, and allows the connected components to receive the information as soon as possible.

- *Epidemic dissemination:* Components carry out periodical anti-entropy sessions among them in order to exchange the information they know. This way, if a component misses a multicast message because of network failure, disconnection or any other reason, it will eventually receive the information through these sessions. TDAs carry out bidirectional sessions with a given number of randomly chosen TDAs. UAs, on the other hand, periodically contact a random TDA in order to update their information.

Table 1. API for service deployment

Functions that LaCOLLA user agents offer applications	
createService	Creates a service and stores its information persistently within the group.
modifyService	Modifies the information about a service previously created within the group.
deleteService	Deletes the information about a service previously created within the group.
startService	Activates a service previously created, according to its specification.
stopService	Stops a service previously created and activated.
getExistingServices	Returns a list of all existing services in the group.
getService	Returns the information of a specific service.
getServiceUbications	Returns the locations where a service is executing.
Functions that applications should offer LaCOLLA user agents	
notifyException	Allows the system to notify the application an exception about a service it activated.
Functions that LaCOLLA execution agents offer executor environments	
login	Allows an environment to connect to a group.
logout	Allows an environment to disconnect from a group.
putObject	Allows an environment to store an object (file) within the group.
getObject	Allows an environment to get an object previously stored within the group.
notifyException	Allows an environment to notify the EA a service's exception, so that it sends it to the corresponding application.
Functions that execution environments should offer LaCOLLA executor agents	
checkServiceRequirements	The environment analyzes the requirements of the services to check if it can execute it.
assignService	Allows the EA to assign a service for execution to the environment.
stopService	The environment quits executing the specified service.
serviceState	Returns the current state of the service (running, stopped).
getState	Returns the state of the execution environment (active, stopped).

5.2. Dynamic information

The second type of mechanisms includes service activation and deactivation and, whenever possible, its guarantee of availability. Specifically, this requires at least one TDA (to manage the deployment) and one EA with a connected environment which matches the service's specified requirements.

To carry out this management, a service is assigned, in the moment of activation, to a TDA which will act as master. It periodically checks that the current number of replicas is the appropiate, detects stopped service replicas, and stops or activates new replicas when needed. In addition, there is a set of secondary masters assigned to each service, chosen by the primary master, which maintain the information about the service management kept by the primary, and can take its place in case it fails. LaCOLLA provides a presence mechanism, so that TDAs don't need to obtain this information and will automatically detect other TDA's failure. It must be considered, though, that because of the optimistic approach taken in the mechanisms' design, this detection is not immediate. The master periodically multicasts the information about who the primary and the secondaries are to every connected TDA and UA. The UAs also receive the list of locations of the service.

The master of the service is in charge of choosing the nodes where the service will be executed. In order to do this, it checks the specification of each of the execution environments connected to the group, comparing them with the requirements of the service. With the ones that match the required specification, the master creates a list of candidate execution environments. Among these, it chooses the number corresponding to the desired number of replicas for the service. The TDA sends a message to every chosen environment, containing the specification of the service and telling them to execute it. On reception, each environment will get the necessary files from the storage provided by LaCOLLA (i.e., the RAs) and start execution. To ensure the TDAs know the environments connected to the group, they carry out periodical synchronization sessions with the EAs. Also, EAs use multicast messages to inform TDAs of execution environment's connections and disconnections.

Two different mechanisms have been designed to elect a master for the service. One is inspired in Distributed Hash Tables (DHT), while the other is based on CSMA/CD, the technique used in Ethernet.

The pseudo-DHT method is based on the main idea of DHTs, assignation of the responsibility on a particular entity to a certain node based on a proximity function. We obtain keys by hashing the service identifier and each TDA identifier, using the function SHA-1 [10]. Resembling Chord [11], the TDA keys are organized in a ring, so that for each service, the closer TDA is the one with the same key or the first successor. Unlike DHTs, we don't need to have a service always assigned to the closest node, due to the master disseminating information periodically. Because of that, as long as the master for a service is active, the connection of new members will not change this assignation. This way we save ourselves to make many responsibility shifts if the level of churn in the group is high. When the master disconnects or fails, each of the secondary masters of the service will check if they're the next successor. If this is the case, that TDA will proclaim itself as primary master. In case there are no active secondary masters, the primary will be elected among the rest of the TDAs. If conflicts arise from the decisions taken locally by the TDAs due to the eventual consistency of the presence mechanism (in a given moment, a component may not know all connected members, or may think a disconnected member is still connected), they will be eventually detected (using the presence mechanism and the multicast messages sent by the master) and solved using the proximity function.

The pseudo-CSMA/CD method takes advantage of randomness to distribute the responsibilities on services among the TDAs. Firstly, when a service is activated, the UA who receives the request randomly choses a TDA, which is assigned as the master. Likewise, this one choses the secondaries randomly. The problem comes when a disconnected master must be substituted. This is where the CSMA/CD mechanism is taken as model. We used its negotiation mechanism, based on random intervals, to design a negotiation method between TDAs to decide who is responsible of a service. When a TDA detects that the master of a service is no longer available, if it was one of the secondary masters of the service, or wasn't but doesn't know of any active secondary, it can decide to substitute it. In case this decision is favorable, it informs the rest of the TDAs. After this, it waits a certain interval for any equivalent message from another TDA (which, in CSMA/CD, would mean a collision). In case it is received, the TDA will wait a random interval. If, during this wait, it receives a second message of the other TDA, it will withdraw from the negotiation and yield the responsibility to the other one. Otherwise, it will send the message again, waiting for another interval to detect messages from other TDAs. In case it doesn't receive any, it will infer that he has won the negotiation and will proclaim himself as the master, multicasting a message to all the TDAs.

Finally, when a user wants to access a service, he asks its location to the UA where he is connected and uses it to directly contact the service. In case that, for any reason, the UA doesn't have this information, it will ask the master of the service or any of the secondaries, and provide it to the user.

6. Validation

We have a prototype of the LaCOLLA middleware implemented in Java that can work either in real time or in simulation time (measured in iterations). It offers an interface for the users to access the functionalities of the middleware, and also has the ability to simulate users activity and system dynamism (connections, disconnections, failures) in order to conduct tests and validate its functioning. The basic core of the service deployment system has been implemented on this prototype. Specifically, we have implemented the mechanisms for service creation, activation, deactivation and access. Both methods for master election, pseudo-DHT and pseudo-CSMA/CD, have been implemented.

Table 2. Component disconnection and failure probability per iteration

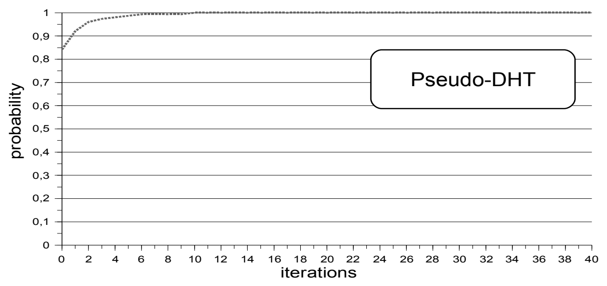
	Failure		Disconnection	
	Prob.	Duration	Prob.	Duration
UA and EA	0,0005	[15,120]	0,0025	[60,540]
RA, GAPA and TDA	0,000125	[12,60]	0,0005	[15,120]

Table 3. Activity generation

Action	Probability per iteration
Service creation	0,01
Service activation	0,007
Service deactivation	0,005

This implementation has been used to carry out tests with groups of 10 components, formed by one RA, one GAPA, two UAs, four TDAs, and two EAs. Tests have been carried out in order to verify that the system does converge, that is, it reaches a consistent state. Each experiment was divided into two phases: during the first one, we simulated users activity (service creation, activation and deactivation) and system dynamism (failure and disconnection of components); the second phase involved only internal mechanisms. We measured the number of steps it took LaCOLLA to achieve consistent information in all the components.

These tests have been carried out for both the pseudo-CSMA/CD method and the pseudo-DHT method (Fig. 2). As can be seen in the graphs, in both cases the system reaches a consistent state in few iterations. Specifically in an 85% of the executions the system is consistent in the same moment the first phase ends, and in only two iterations 95% of the executions have already reached consistency. From the results we deduce that the design proposed makes the system reach a consistent state, proving that the decentralized management and self-organization work in presence of churn. With current tests, though, we can not observe important differences between the behavior of the two mechanisms implemented for master election. Therefore, we can say that both are valid for the operation of the system, although it will be necessary to carry out more ex-



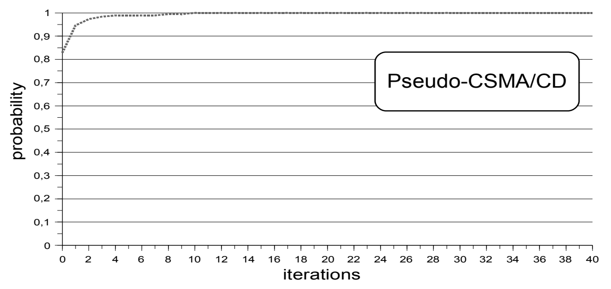


Figure 2. Cumulative probability that in N iterations the system has reached consistency using the pseudo-DHT method and the pseudo-CSMA/CD method.

haustive tests in order to identify in which situations and environments we can obtain better performance of one particular mechanism.

7 Conclusions and future work

We have presented a basic core for the deployment of stateless services in a collaborative environment. However, there are many aspects that can be further developed. Future lines of work include:

- Carry out more exhaustive tests, using a greater number of components and collecting different measurements.
- Add the functionalities for service modification and elimination, while keeping the system's self-organization capability, so that it can solve by itself the possible inconsistencies caused by concurrent updates.
- Define different replication policies, which allow the user who deploys the service to choose which option he considers more adequate. Current implementation guarantees an exact number N of instances of the service, but it would be interesting to implement different guarantees such as a minimum of N instances, a maximum of N, etc.
- Allow the deployment of stateful services.
- Allow communication between different replicas of a service, through message passing among them. This possibility would allow to offer more complex services, where the existing replicas coordinate their actions.
- Build execution environment able to execute other types of applications (current implementation can only execute Java applications), adding new considerations like security, isolation from the machine where it's running, users privacy, etc. This would allow its use in environments where a trust relation between users does not exist, and hence extend the system's usability.
- Explore possible modifications of the proposed mechanisms to improve system's scalability, so that it can be applied to large groups.

References

[1] Marquès, J.M."LaCOLLA: Una Infraestructura Autònoma i Autoorganitzada per Facilitar la ColLaboració," [LaCOLLA: An Autonomous and Self-Organized Infrastructure to Facilitate Collaboration], doctoral thesis, Dept. of Computer Architecture, Tech. Univ. of Catalonia, 2003, http://people.ac.upc.es/marques/LaCOLLA-tesiJM.pdf.

[2] Marquès, J.M., Navarro, L., Vilajosana, X., Daradoumis, A. (2007). LaCOLLA: Middleware for Self-Sufficient Online Collaboration. IEEE Internet Computing. pp. 16-24. ISSN:1089-7801. (to be published on January 2007)

[3] Thain, D., Tannenbaum, T., Livny, M. Distributed Computing in Practice: The Condor Experience. Concurrency and Computation: Practice & Experience. v. 17, Issue 2-4 (February 2005) pp. 323-356 ISSN:1532-0626

[4] Verbeke, J., Nadgir, N., Ruetsch, G., and Sharapov, I. Framework for peer-to-peer distributed computing in a heterogeneous, decentralized environment. In Proceedings of the 3rd International Workshop on Grid Computing, November 2002.

[5] Gong, L. JXTA: A network programming environment. IEEE Internet Computing, v. 5, pp. 88-95, 2001

[6] Saito,Y., Shapiro, M. (2005). Optimistic replication. ACM Computing Surveys (CSUR). v.37 n.1, p.42-81, March 2005.

[7] Terry, Douglas B., et al. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. Proc. 15 th Symposium on Operating Systems Principles, Dec 1995

[8] Golding, R.A. Weak-consistency group communication and membership, USCS-CRL-92-52, Concurrent Systems Laboratory, University of California, 1992.

[9] http://gridengine.sunsource.net

[10] NIST,"Secure hash standard", U.S. Department of Commerce, National Technical Information Service FIPS 180-1, Apr. 1995.

[11] Stoica, I., Morris, R. et al. Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications, IEEE/ACM Trans. Net., vol. 11, no. 1, 2003, pp. 1732.

[12] http://www.clusterresources.com/pages/products/torque-resource-manager.php