A Best-effort Mechanism for Service Deployment in Contributory Computer Systems

Daniel Lázaro[1], Joan Manuel Marquès[1,2], Josep Jorba[1*]

[1]Universitat Oberta de Catalunya
{dlazaroi,jmarquesp, jjorbae}@uoc.edu

[2]Universitat Politècnica de Catalunya
marques@ac.upc.edu

Abstract

This paper presents a simple decentralized service deployment mechanism that can be used to offer services with high availability by using contributed resources with variable, possibly very low availability. It uses little information about the state of the group, and provides best-effort results. Contributory computer systems are those where users provide their own resources to be used collectively. Contributory applications are complex to build, as they have to deal with issues like unpredictable individual resource availability, heterogeneity and ease of use, as well as the complexities derived from their own functionalities. This complexity can be alleviated by creating a middleware for the creation of contributory communities that allows the use of contributed resources in a decentralized way through service deployment.

1 Introduction

Computer supported collaboration is gaining more importance every day. People use their computers to collaborate in many ways, from providing information to direct collaboration, through local area networks or through the Internet. Many different models have been proposed for different scenarios of collaboration.

We propose the model of contributory computer systems, where users provide their own resources to be used collectively. These resources can be used in different ways, determined by the functionality and objective of the specific contributory application. There is a wide range of architectures and mechanisms that can be used to implement a contributory system. The choice highly depends on the additional requirements of each specific contributory application and the functionality that it aims to provide, as well as the kind and quantity of resources available. However, there are a set of characteristics that define this kind of environment and that must be taken into account when designing a contributory system.

The users of these systems typically will try to provide resources to the extent of their possibilities, as they wish to help the community. Users want their resources to be used by the community. However, they still want to retain control of their resources. This will result in intermittent resource availability, as users can decide whether to contribute or not at any moment. Another aspect that must be considered is that, being provided by independent users, no homogeneity can be guaranteed. Finally, if non-specialized users are to contribute their resources to the system, this process must be made as easy and intuitive as possible. No complex administration should be required.

In summary, contributory computer systems must deal with unpredictable individual resource availability, heterogeneity and ease of use. Many different models and architectures can be used to deal with these issues and build systems that can be used by contributory communities.

Contributory applications are complex to build, as they have to deal with the mentioned issues as well as the complexities derived from their own functionalities. This complexity can be alleviated by creating a middleware for the creation of contributory communities that allows the use of contributed resources in a decentralized way through service deployment. With the functionality offered by this middleware, a service can be deployed in a contributory community. The service can therefore be executed using the contributed resources, transparently to the users, who can access it independently of its actual location. This middleware should work in a completely decentralized way, so that contributory communities do not need support from any external entity.

Service deployment can be used as a building block for the construction of contributory applications. Centralized applications can be adapted to work in a decentralized contributory environment with relatively little effort, as they can be deployed in the community without decentralizing their internal behavior. Decentralized applications can also benefit of deploying some components as services in the community, as it gives them a point of centralization that can make their mechanisms simpler.

*This work has been partially supported by the "Comissionat per a Universitats i Recerca del Departament d'Innovació, Universitats i Empresa de la Generalitat de Catalunya" and by the "Fons Social Europeu" under grant FI. This work has also been supported by TIN2008-01288/TSI and TIN2007-68050-C03-01.

Previously, our research group created a middleware to ease the implementation of contributory applications in small groups. This middleware is called LaCOLLA [3] and provides applications with some basic functionalities like message passing, event dissemination or contributive storage. It also offers a basic service deployment functionality [4]. This work expands the previous research, focusing on the resource sharing through service deployment and creating a system that can scale to large communities. It is also compatible with DyMRA [5], a system that allows to exchange resources among different contributory communities through economic-based mechanisms, giving a wider range of possibilities to this kind of groups.

2 Related work

Contributory systems are strongly related to grid and peer-to-peer systems. However, contributory systems have different characteristics, as well as different purposes, from the typical representatives of both peer-to-peer and grid systems. However, there are individual mechanisms that can be adapted and used in our environment, from overlay [6] networks used in peer-to-peer systems to many of the standards developed for grid systems.

One of the most known and used contributory systems for distributed computing is currently BOINC [1]. It allows a centralized server to deploy an application in a set of contributed nodes, sending different sets of data to each of them and retrieving the results. Our system is designed for a more general scenario than BOINC, where the task to be done with the contributed resources can be of a more diverse nature, and the resources are used in a decentralized way.

Other example of contributory systems would be contributory storage. Systems like TFS [10] allow a set of computers to contribute their storage to be used collectively. Instead, our purpose is to contribute resources for general tasks, not exclusively for storage. However, these systems might serve of inspiration when managing storage in our contributory middleware.

A system with similar functionality is Snap [11]. It allows users to deploy a web-based service in a set of nodes united by a DHT. Service replicas are created on demand, and are hosted by those who will use it. Xenoservers [12] also has the objective of deploying services in a group of computers that can scale up to millions of servers. However, its nature is far away from contributory systems, as users can buy access to resources to use them individually, while our system uses resources collectively in a community.

3 System Model

Our system is modeled like a contributory community, where users contribute resources in the measure they desire. Resources are contributed by members organized in nodes. Each user contributes one or more nodes. In each of these nodes, the user can specify the resources it contains, that is, the portion of its capacity that will be contributed for storage or execution purposes, as well as other additional capacities the node may have and contribute to the community. The nodes are connected through the Internet, and can be located all around the world. However, they are also united by a structured overlay network [6].

The purpose of the system is to offer services to its users. The middleware offers an interface to applications that can be used to deploy services and to access them. With these tools, contributory applications can be built on the middleware, leveraging the advantages and hiding many of the disadvantages of decentralized contributory environments.

The services deployed in the community can be of varied nature. For example, some services can be meant for the direct use of the members, like web publishing, wikis, forums, etc. Other possibility is that the service deployed is a component of an application that the members of the community want to use, and that acts as a centralized or semi-centralized component supporting a client that users must also execute in their computers. Examples of this could be executing a BOINC server, that would in turn distribute tasks among its clients, or an online videogame server, which users would contact to play an online multiplayer match of a game also installed in their computer.

4 Architecture

In this section we will briefly introduce the architecture for the proposed middleware. A more detailed description can be seen at [7]. This architecture, presented in figure 1, is divided in four layers. The three lower ones can be built with existing mechanisms and technologies, requiring little or no adaptations. The upper layer, instead, is the one that requires the development of novel mechanisms to collectively use the resources contributed by members.

- *Fabric layer*: This layer provides access to the local resources of a node. The most common are local execution and local storage capabilities, but special capabilities (like a specific peripheral connected to the node) would also be controlled by this layer. Security and isolation is managed inside this layer.

- *Connectivity layer*: This layer provides access to the nodes that form the community. It forms an overlay network in order to be able to route messages to all nodes in a scalable way. It also provides some additional capabilities like multicasting and broadcasting.

- *Resource layer*: This layer provides mechanisms for remote usage of resources. This usage includes remote execution control and monitoring.

- *Collective layer*: This layer is the one that uses the resources of the users in a collective way, thus providing

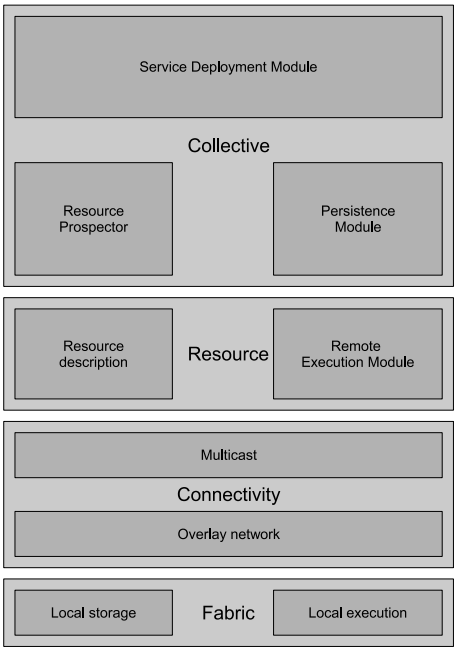

Figure 1: Architecture of the middleware.

the specific functionalities of the contributory middleware. It has a few components that carry out its main functionalities:

- Resource prospector: is in charge of finding resources available to the community that satisfy a certain set of requirements.
- Persistence module: is in charge of collectively managing the storage resources of the community.
- Service deployment module: is in charge of deploying services in the community and make them available to users through the use of the contributed resources.

5 Service deployment mechanism

The mechanism we have designed for service deployment uses contributed resources to offer the services to the clients in a best-effort way. Figure 2 shows a simplified diagram of the deployment process. The steps involved are: 1) user contacts the Client interface to create a service; 2) the Client contacts a Storage Node to store the files of the service; 3) the Storage node replicates the files in other Storage Nodes; 4) the Client contacts the node that will become the Service Manager to inform it of the creation of the new service; 5) the Service Manager replicates the service information in the Service Manager Replicas; 6) the Service Manager contacts the Worker Nodes to order them to execute the service; 7) the Worker Nodes get the necessary files from the Storage Nodes and start executing the service.

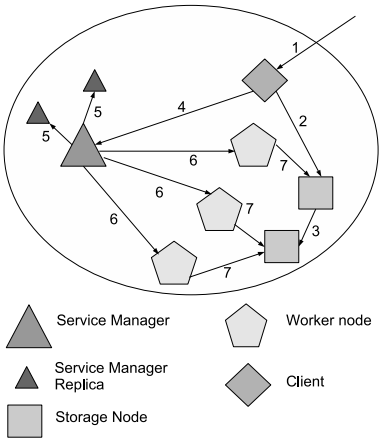

Figure 2: Interactions in a community during the process of deploying a service.

In the next subsection we will explain the mechanism in detail. Figure 3 shows the service creation and activation process over a distributed hash table (DHT).

5.1 Service creation

A user can create a service from a service deployment file. This file must contain all the necessary files for the deployment of the service. These files includes all the executable files that are needed for the program to run on a computer, as well as the data files that the service needs. For example, in the case of a web server, the service deployment file would contain the executable file for the web server and all the web pages that the server is intended to offer. Besides, the service deployment file must also contain a service descriptor. This service descriptor contains data that the Service Deployment Module (SDM) needs to deploy the service. This must include the requirements of the service for possible execution environments, as well as the execution method (the instruction or set of data that will be passed to the execution module to start the execution of the service). Other data that can be contained inside the service descriptor are the name of the service or a semantic description of the service.

In order to create a service, a user must pass the service deployment file to the SDM. The SDM extracts the execution files and stores them using the Persistence Module. Then, it creates a new entry in the index that contains the existing services in the group, and associates it with the service descriptor. After this, it sends a confirmation to the user, informing him or her that the service has been created correctly.

In order to add an entry in the service index, the local SDM, L, will use the Key-Based Routing (KBR) interface in the connectivity layer to contact the node responsible for

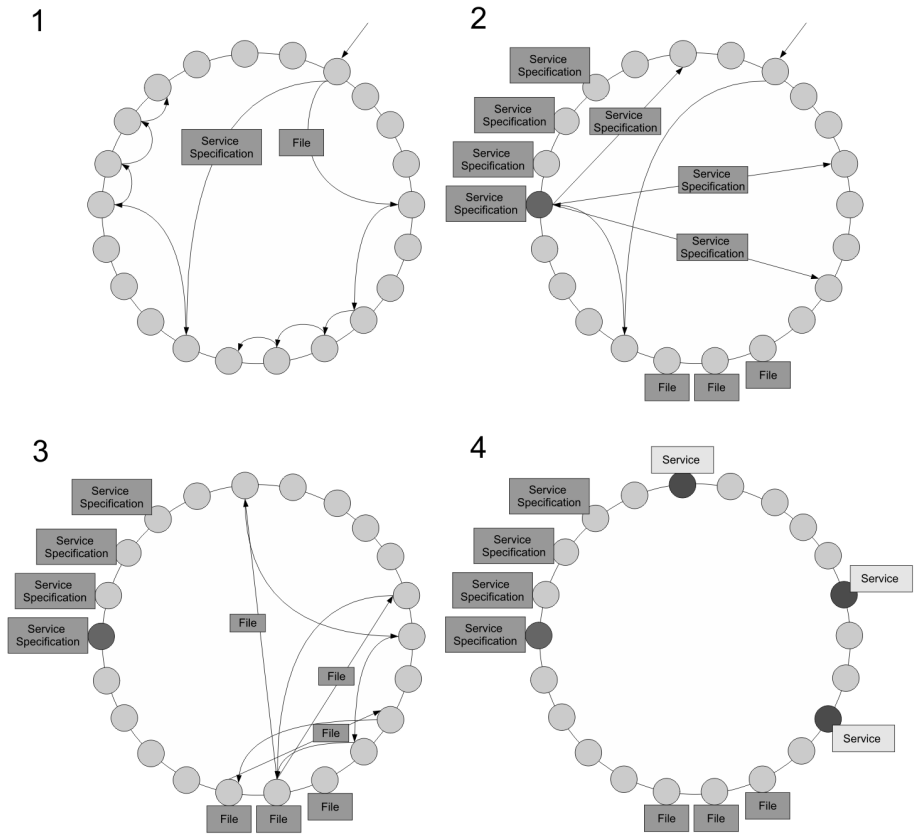

Figure 3: 1. To create a service, the Client stores the files in the DHT, and sends the service specification to the responsible node, which replicates it in a set of backup nodes. 2. To activate the service, the Client contacts the responsible node, which chooses a set of nodes to execute the service and sends them the service specification. 3. The workers get the service files from the DHT. 4. The workers execute the service.

the entry. This responsible node R is determined by a hash of the name of the service. R stores the service descriptor associated to the name of the service, which is specified in the same descriptor. R sends the descriptor to a set of nodes that must store replicas of the data. This set is decided by the replication policy of the underlying DHT. These nodes store the descriptor on reception. After sending the descriptor to the designated backup servers, R contacts L to inform it about the success of the operation. In case L does not receive a response, it will retry the operation. In case that R has failed or disconnected during the operation, the message will be directed by the DHT to the node that is currently responsible for the identifier.

Periodically, each SDM checks each of the service descriptors it stores. First, it asks the Overlay module if the key of the service is under its responsibility. If it is, then it obtains a replica set from the Overlay module, and compares it to the backup servers currently assigned to the service. If new nodes have joined since the last check and are into the replica set, or old backup servers have disconnected, and this have caused new nodes to fall into the replica set, the SDM sends the new backup servers the service descriptor. This descriptors should be stored persistently, to prevent them from being removed from the community. Although the number of backup servers should be fixed taking into account the dynamism of the community, to minimize the possibility of all the nodes containing a service descriptor being disconnected simultaneously, this possibility still exists. Therefore, storing the service descriptors persistently will allow the service to be reintroduced once one of the nodes that stored it connect again to the community.

5.2 Service activation

When a user wants to activate an existing service, he or she contacts the SDM indicating the name of the service to activate. This step can be done automatically after the creation of a service. However, existing services can be started and stopped, and as their creation is an independent event of their activation, they can be reactivated without requiring to go through the creation process again.

Using the name of the service, the SDM sends a message to the node R responsible for the service, indicating the name of the service to activate, using the underlying KBR. When R receives this message, it uses the Resource Prospector to get a list of nodes that fulfill the requirements of the service. From this list, R choses N nodes where it will try to execute the service. This selection can be random, taken from an ordered list or guided by some preferences that could not be considered by the Resource Prospector, depending on the policy determined in the service descriptor. The number of replicas N should be determined considering both the characteristics of the community, in terms of churn and resource availability, and the importance of the service. For each of these N selected nodes, the SDM sends a message to the Remote Execution Module (REM) in the Resource layer, containing the service descriptor and the identifier of the node. The local REM contacts the destination using the Resource layer protocols. In the destination node, the REM checks whether it can execute the service. If the result of this check is negative, the remote REM will inform the local REM of this result. If the result is positive, the remote REM retrieves all the necessary files from the Persistence Module and starts the execution of the service. The SDM stores the information of the node in the data structured associated to the service, so that it can contact and monitor its REM via the Resource layer.

Once the N replicas have been started, R sends the data structure with the information about all the replicas to a subset of the set of backup servers that store replicas of the service descriptor. This subset will also store the information of the replicas. This is done because this information might change frequently (depending on the rate of connection and disconnection of the nodes), so increasing the number of nodes that store this information will greatly increase the communication cost. Moreover, contrarily to the service descriptor, it is not vital that the information of the current replicas is kept inside the community. If this information is lost because the responsible of a service and all the others which store it are disconnected, the node that becomes responsible of the service can simply create new replicas. Still, it is undesirable as they would need some start-up time, so there is a trade-off among the decrease of availability this could imply and the increase in communication cost that comes with storing this information in more nodes.

Periodically, as has been explained in the service creation process, each SDM checks the service descriptors it stores and decides for which ones of them it is responsible. For these, it also checks, using the Overlay module, if the nodes that store the replicas are alive. If it finds that some are not, it creates new replicas by the same process as when the service was first activated. Then it sends a keep-alive message to all the nodes that store a replica of the service. Each REM has, for each service it is executing, an associated TTL (Time-To-Live) which decrements periodically. It is reset to its original value when the REM receives a keep-alive message for the service. If the TTL reaches zero, the node stops executing the service. If a REM receives a keep-alive message for a service it is not executing, it responds with a message informing that this service is not currently in that location. When a SDM receives one such message, it deletes the corresponding location of the ones assigned to the specified service, if it was one, and proceeds to create a new replica.

6 Validation

We have tested the proposed mechanism through simulation. We have performed a number of tests for different configurations, with varying parameters like community size and number of services. For these tests, though, we only considered a single class of services, that could be executed in any node. Therefore, our Resource Prospector only needed to provide a set of random nodes from the community.

We implemented the service deployment mechanism over PlanetSim [9], a simulator for overlay network protocols and applications. It uses a general API that allows applications to be adapted to any underlying network protocol. For this simulations we used Chord, but uses generic KBR operations, so it could be adapted to work on top of any other DHT.

We conducted tests using networks of varying sizes and behaviors. We simulated user behavior with random connections and disconnections of nodes. We took user behavior data from traces of SETI@home [8], as there are some similarities in the expected behavior, both having users contributing their personal resources. From there, we assigned nodes a mean availability of 80%. To explore more extreme cases, we also simulated communities with a mean node availability of 50% and 30%. Figure 4 shows, for each of these configurations, the mean service availability obtained, for different community sizes. As expected, the configuration with 30% node availability is the one that provides a worse service availability, but it still reaches 88% for all community sizes, which is a good number considering the poor node availability. We also see that the service availability does not decrease with community size. Therefore, we can say that the service deployment mechanism scales up to communities of 5000 nodes with good results in terms of service availability. However, other factors like message overhead and latency should be considered to fully characterize the scalability of the algorithm.

We also measured the mean length of availability and unavailability intervals of services for the same configurations. The duration of simulations was of 10,000 simulation steps. We can see in figure 5 that with a mean node availability of 30%, the mean length of availability intervals is considerably lesser than with 50% and 80% of node availability. Still, considering that mean length of node availability intervals was fixed at 3,200 iterations, length of service availability intervals still surpasses it in most cases. We also see

in figure 5 that length of unavailability intervals range from 30 to 160 simulation steps.

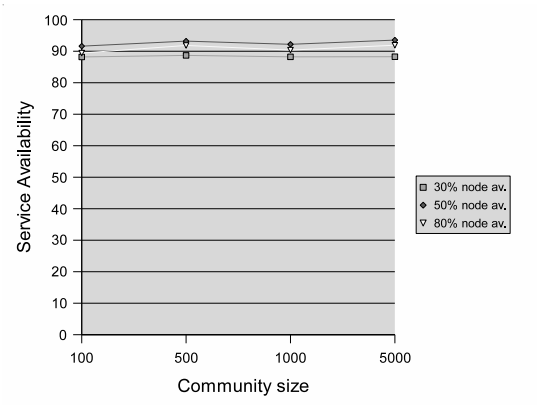

Figure 4: Mean service availability vs. community size, with different values of mean node availability.

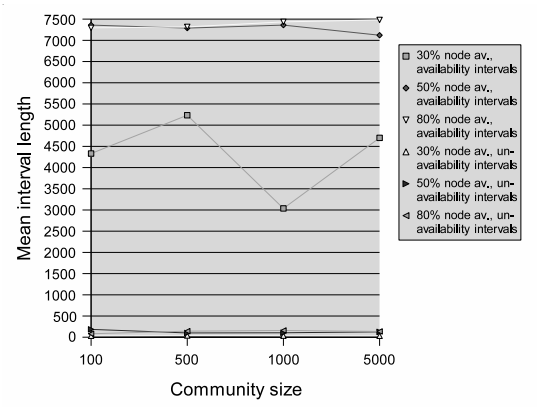

Figure 5: Mean length of service availability and unavailability intervals vs. community size, with different values of mean node availability.

7 Conclusions

We have presented a simple decentralized service deployment mechanism that can be used to offer services with high availability by using contributed resources with variable, possibly very low availability. It uses little information about the state of the group, and provides best-effort results.

Given the good results obtained with this mechanism, we plan to extend it to support services with various specific resource requirements. When implementing it to work on a real community, this will also imply the need of a scalable mechanism for resource discovery, that can work without depending on non-contributed resources. Finally, improvements can be made to the service deployment mechanisms in order to guarantee better availability for high priority services. Such mechanism could make use of a little more information about the state of the group to offer better availability guarantees. The trade-off among information overhead and availability performance should be adequate to the needs and the importance of each individual service and community.

References

[1] Anderson, D. BOINC: A System for Public-Resource Computing and Storage. 5th IEEE/ACM International Workshop on Grid Computing. 2004.

[2] Foster, I., Kesselman, C. The Grid. Blueprint for a new computing infrastructure. Morgan Kaufman, 1998.

[3] Marquès, J.M. et al. Lacolla: Middleware for self-sufficient online collaboration. IEEE Internet Computing 11 (2) 56-64, 2007.

[4] Lázaro, D., Marquès, J.M., Jorba, J. Decentralized service deployment for collaborative environments. In Proc. CISIS07, pp. 229-234, 2007

[5] Lázaro, D., Vilajosana, X., Marquès J.M. DyMRA: Dynamic Market Deployment for Decentralized Resource Allocation. LNCS. OTM Workshops (1) 2007: 53-63 (2007)

[6] Eng Keong Lua et al. A survey and comparison of peer-to-peer overlay network schemes, Communications Surveys & Tutorials, IEEE, vol.7, no.2, pp. 72–93, 2005.

[7] Lázaro, D., Marquès, J.M., Jorba, J. An Architecture for Decentralized Service Deployment. Proceedings of CISIS'08. pp.327-332. March (2008)

[8] Kondo, D., Anderson, D., McLeod Vii, J. "Performance Evaluation of Scheduling Policies for Volunteer Computing," e-science,pp.415-422, e-Science 2007.

[9] García, P. et al. PlanetSim: A New Overlay Network Simulation Framework. LNCS, Volume 3437. Software Engineering and Middleware, SEM 2004, pp. 123-137.

[10] Cipar, J., Corner, M. D., and Berger, E. D. 2007. Contributing storage using the transparent file system. Trans. Storage 3, 3 (Oct. 2007), 12.

[11] Pairot, C., García, P., Mondjar, R. Deploying Wide-Area Applications Is a Snap. IEEE *Internet Computing*, 11 (2) 72-79, 2007.

[12] Kotsovinos, E. et al. Global-scale service deployment in the XenoServer platform. In Proceedings of WORLDS '04.