An Architecture for Decentralized Service Deployment

Daniel Lázaro, Joan Manuel Marquès and Josep Jorba
Universitat Oberta de Catalunya
{dlazaroi,jmarquesp,jjorbae}@uoc.edu

Abstract

In this paper we present a proposal of the architecture for a system which allows the deployment of services in a group of computers, connected in a peer-to-peer fashion. This architecture is divided in layers, and each of them contains some components which offer specific functions. By putting them together, we obtain a system with desirable characteristics such as scalability, decentralization, ability to deal with heterogeneity, fault tolerance, load-balancing, and self-* properties.

1 Introduction

Nowadays it is usual for groups of people who work on a common task (whether it's business-related or of another kind, like social or political activism, or tasks done as a hobby) to coordinate their efforts using the Internet. This proliferation or virtual collaboration fosters the creation of spontaneous groups who want to collaborate, but may lack the necessary resources to adopt usual solutions (dedicated servers, grid infrastructure, clusters of machines, etc.), or need more flexible approaches. We try to provide a solution for this kind of situations, centering our efforts in the following scenario.

The kind of users we want to address is that of communities formed by individuals who share a common interest and want to offer services to the members of the community and to the world. These communities can be formed by a large quantity of people throughout the world. As stated, they will share a common interest, whether it is a hobby, a business objective, or anything else. For example, they can form a community dedicated to free software, or to promote ecologist activities, etc. Some of these communities don't have a financial source that provides them with resources, or at least not a sufficient amount of them for the purpose of the community. Hence, these will need to be contributed by the own members of the community. Even if they do have

external contributors, their resource needs may not be satisfied by a single contributor, and the community would need a flexible, spontaneous way to aggregate this resources, so that they can use them to accomplish their objectives without needing great management efforts.

The members can contribute to the activities of the community in two ways: by offering their time and their work, or by providing resources. Obviously, a member can contribute to her community in both ways. The first one would consist on generating contents for the community. The latter one would imply handing over a part of her computational and network resources to allocate the services the group wants to offer. A fundamental requirement for the correct behavior of the community is that there must always be enough available resources to allocate the services that are to be offered. In the case that this requirement is not met, the services can not be offered. While counting on having enough total resources, however, no assumption can be made about the availability of a specific resource, as, being provided by the members of the community, resources can be withdrawn or fail at any moment. Moreover, not being provided by a common source but by individual users, a potentially high level of heterogeneity must also be expected.

As stated earlier, the purpose of these communities is to offer services. These can be whether exclusive for the members of the community (e.g. management information about the communities activities) or open for all internet users (e.g. publishing information about the community and its activities). These services can be, for example, web publishing, content management systems (CMS), wikis, forums, blogs, video and audio streaming, a shared agenda... For these services to be accessible, the system uses the resources of the community to keep them activated. These ability to keep services available inside the community can not only be used to offer these user-oriented services, but also to manage some aspects which are transparent to the users, but which can also leverage the possibility of deploying services to facilitate its implementation.

Although this scenario can seem quite specific at first sight, the system presented could, at least partially, be appli-

cable to other, more general, situations. Deploying services can be used as a means to share computational resources between members of a group, as well as being a good and simple way to introduce centralized or semi-centralized applications in a decentralized environment. Also, a system that offers this functionalities in a completely decentralized, peer-to-peer environment will also use mechanisms that could be used in other distributed architectures, such as grid systems.

2 Requirements

As stated earlier, we want to build a system which allows the deployment of services in a large group of distributed computers. Users must be able to deploy services in the community and manage them, monitoring their performance and starting or stopping them. All users must also be able to discover services, according to a description or to certain characteristics, and access them, whether it is directly or through the system. This functionality, however, must be delivered achieving certain conditions. The main requirements of our system are the following:

- *Community self-sufficiency:* a community should not depend on external resources. All the functionalities should be performed using only the resources contributed to the community by its members. This places some restrictions in the availability of services, as they can only be deployed when the members of the community provide enough resources for it.

- *Individual autonomy:* the members of the community should be free to decide which actions to carry out, what resources and services to provide, and when to connect or disconnect. Resources must be used at the level their owners allow them to be used, without overloading them and always allowing user control.

- *Decentralization:* any responsibility might be assumed by any node.

- *Scalability:* the system should be highly scalable, achieving good efficiency for groups formed of a great number of machines, while also leveraging the advantages of small groups when so is the case. It should also scale well geographically, allowing users to be scattered around the world and connected through the Internet.

- *Heterogeneity:* as resources can be provided by individual users as volunteer contributions, it is impossible to expect any level of homogeneity. Hence, the system must be able to handle highly heterogeneous systems.

- *Fault-tolerance:* the system must tolerate the failure of a high number of nodes, and keep all functionalities available. Services must be kept available in presence of failures, as long as there are enough resources in the community.

- *Location transparency:* users must have access to the deployed services independently of both user's and service's current location.

- *Self-management:* our system needs to minimize the necessity of external management to achieve better ease of use for potential collaborative uses, usually carried out by non-specialized users. While not requiring human intervention, it must be able to use the available resources in, if not the optimal, a suitable way. For example, the system must return, without external intervention, to legitimate configurations when nodes fail, join or leave, as well as be aware of its performance and execute actions to improve it.

3 Architecture

We want to design a system with the characteristics mentioned above. The functionalities and requirements of our system overlap partially those of grid systems [2], as both try to allow resource sharing, while allowing heterogeneity, scalability and other characteristics. Because of this similarities, we take their layered architecture [3] in consideration and try to divide our intended functionalities into the existing layers. Although we don't compromise with complying the grid standards, we can adapt this architecture and use it to build our system. Many of the functionalities covered by our layers have been extensively studied and researched, so when presenting them we will also mention some related work, and discuss if it fulfills our requirements or not.

3.1 Fabric

In this layer we find the individual capacities of each node. We need local storage and execution capacity for those nodes that offer such resources to the group. These functionalities, however, must be offered in the proportion that the user decides to give to the group. Hence, a system to control and limit the use of the local resources is needed. Common interfaces will be needed to access this capacities from the upper layers disregarding heterogeneity.

3.2 Connectivity

We will create an overlay network which will be used to route messages inside the group through internal identifiers. This field has been extensively studied, and many

overlay networks have been proposed, mainly in the form of distributed hash tables (DHT). We can find a very interesting survey at [5]. More recent than this survey, there is also another DHT called DKS [6] which introduces some useful features, like consistent lookup results in presence of joins and leaves, efficient broadcast, bulk operations and symmetric replication, a way to place replicas which enables parallel recursive lookups. Because of these features, we will use DKS as our overlay network for the connectivity layer of our system, although the algorithms included in it could be adapted to many other DHTs. Therefore, our system could theoretically be ported to any other DHT with equivalent properties.

As well as internal routing, this overlay network will provide mechanisms for diverse types of multicast messaging (e.g. reliable, best-effort, anycast, etc) inside the group. How to implement an efficient multicast has been a subject of study for a long time, from IP-multicast to most recent application-level multicast algorithms [7]. A system that does so in an overlay network is Scribe [14]. It assigns a node to each group, that will act as rendez-vous point. Subscription messages are routed to this node through the overlay. When these messages are forwarded, a path is created, that will form part of the group's multicast tree. Whenever a publisher wants to send a multicast message, it sends it to the root of the group's tree, the rendez-vous node. As can be easily seen, this approach can introduce bottlenecks in the rendez-vous nodes, and also places forwarding and state-keeping responsibilities in nodes outside the multicast group. A different approach [15] is to create an overlay network for each multicast group. Like in the first approach, a node in the overlay network is held responsible for each group, but in this case it merely stores information about the group-specific overlay network. Hence, it is much less likely that it will become a bottleneck, as nodes must only access it to join the group or first sending a message to it. Once the message reaches the group's overlay network, it must be broadcasted. This way, only the nodes that are part of the multicast group must keep multicast-related state and forward multicast messages. As mentioned above, efficient broadcast messaging is also included in the DKS overlay network, hence allowing it to implement this multicasting method in an efficient way.

3.3 Resource

In this layer we find access to individual resources, defining the protocols, messages, etc, required to, for example, start a process in a remote computer. Whether routing these messages over IP or through our overlay network, we will access to resources individually. Hence, we find the need to describe both services (or jobs) requirements and node characteristics, as well as a protocol to start remote job ex-

ecution. There has been an extensive work also in this area.

Along with classic distributed computing systems, with their own formats to define the characteristics of computing nodes and the requirements of jobs, and mechanisms to assign the execution of a job to a node, we must also consider standardization efforts like GRAM (Grid Resource Allocation and Management) [13]. GRAM is a Globus service which defines mechanisms for users to locate, submit, monitor and cancel remote jobs on Grid-based compute resources. Specifically, it uses a language called RSL (Resource Specification Language) or its XML-based version JDD (Job Description Document) to specify the resources of the grid. Being a widely used and accepted standard, it is a good option for using in the implementation of our system. Another option, that is also based on standards, is OGSA's (Open Grid Services Architecture) [29] execution management, which uses the Job Submission Description Language (JSDL) [28]. Anyway, it is not our purpose to define these protocols, and the selection of one or another existing method shouldn't affect the design of the rest of the system.

3.4 Collective

This layer encapsulates all the services which will use resources collectively. Hence, we will find here the components which allow, among other functionalities, the deployment of a service in a group, or the storage of an object. As we have shown, all the functionalities required for our system in the fabric, connectivity and resource layers can be implemented using existing mechanisms. The core of our system is, thus, in this collective layer, where we must find a way to use the resources of the group to achieve the specified objectives.

We divide the collective layer of our system in many components, that will be now discussed, along with related work in the area which each of them covers.

3.4.1 Publish/subscribe

We need a mechanism to subscribe to events occurred in the group and receive notifications. These events can range from the connection of a node with certain characteristics, to the creation of a service. This service can be used by other components as well as by users. For example, a user can subscribe to be notified whenever a certain service is modified, or when a certain type of service is created within the group. This area has been extensively researched, and there are many distributed publish/subscribe systems [11], but we will focus on the ones based on DHTs.

A system that tries to leverage the capacities of DHTs to implement publish/subscribe in a more efficient way is Hermes [12], which divides the possible notifications in types,

and assigns each of them to a node in the overlay network. This node, called the rendez-vous node, will receive all the subscriptions of the assigned type, as well as the notifications, that will be forwarded to the corresponding subscribers. This is just a plain adaptation of the multicast messaging in Scribe, with few additions. They introduce the possibility of assigning attributes to each type of event, and users being able to specify values for these attributes to subscribe. In our system, this could be used, for example, to create a type of event related to resources, with attributes that define such resources, allowing service providers to subscribe to notifications related to the resources needed to offer their service.

A different approach is taken in [17], where subscriptions are content-based. Events are defined by attributes, and clients subscribe to events with a specific value for a specific attribute. This attribute-value pair is hashed, and the subscription stored in the corresponding node. When a node wants to disseminate an event, it checks the nodes corresponding to the attribute-value pairs of the event. If subscriptions are found, their conditions are checked and the message is sent to the interested clients. Another system for content-based subscriptions is presented in [18], where templates are defined to express the content of events. Clients can subscribe to certain values of the attributes in the template, which constitute a topic, and disseminated events will be checked for conformance to the existing topics. When this conformance occurs, the message is multicasted to the topic subscribers. Although these approaches have evident shortcomings in terms of efficiency and flexibility, content-based publish subscribe is an active field of research, as it can be useful in many systems, as our own, where we could use it to monitor the available resources in the group.

We must study the possible range of events that we want to cover in depth, in order to decide which system we adopt in the implementation of our system, possibly with some adaptations. For now, we want a system that offers a certain flexibility for content-based subscriptions, and that is decentralized, scalable and efficient.

3.4.2 Resource prospector

In order to deploy services, each with its own requirements of resources, in a group, we need to be able to find nodes in the group with certain characteristics, or certain kinds of resources. This has also been an important subject of research, with many mechanisms being proposed for efficient non-id-based searching in DHTs. One interesting way to do this is through data aggregation. Usually, aggregation allows to calculate simple functions for an attribute in a group of nodes, like MIN, MAX, AVG and COUNT. This is useful for resource searching, as e.g. with the MAX function we can easily know if we can find a node with the required
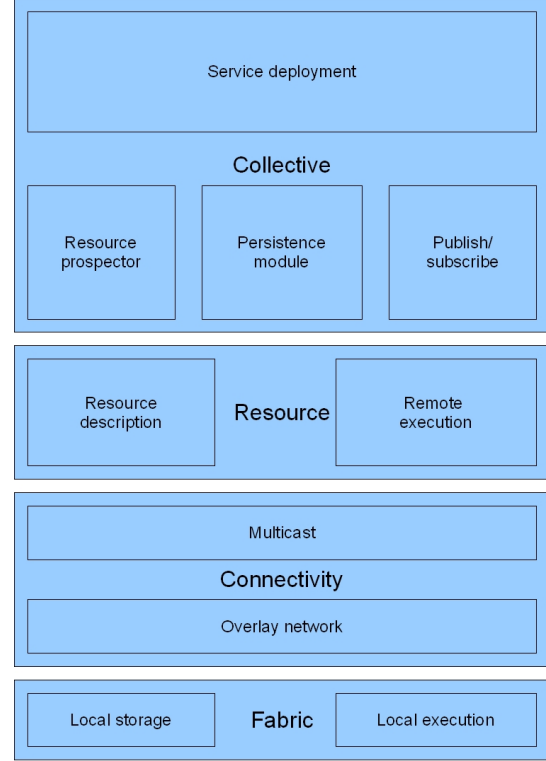
Figure 1. Layers that compose the architecture of the system. The functionalities carried out by each layer are displayed.

memory to execute a service in a subgroup of nodes.

One example is Willow [19], a system that creates a tree for both data aggregation and event notification. SDIMS [20], instead, creates an aggregation tree for each attribute, rooting it in the node responsible for the key of said attribute in a DHT. Cone [21], also working on a DHT, builds a trie for each attribute and aggregation operation. With the correct aggregation functions computed over the right attributes, these aggregation trees allow to efficiently search for nodes with the desired characteristics.

3.4.3 Persistence module

This module must allow the distributed storage of objects using the resources of the group. File storage is probably the most popular and basic application for structured overlay networks, so we could use any existing DHT to implement this part of our system.

As storage has been the main functionality of DHTs since their appearance, there have been many attempts to optimize its use of resources for that purpose. For example, many load-balancing mechanisms have been proposed. A usual method is to assign a set of virtual servers [22] to

each node, having each of these virtual servers a different identifier inside the overlay network. Hence, each node has responsibilities on more than one set of identifiers, increasing the probability of equality among nodes. If a node becomes overloaded nevertheless, it can migrate some of its virtual servers to less loaded nodes. Another possibility for load-balancing is to put aside the virtual servers and directly move nodes [23] from a region of the space to another by dynamically changing their identifier.

However, we have additional requirements that we want to address (limited volunteer resources in nodes, high dynamism, heterogeneity) and that might justify the creation of new mechanisms. Also, we would like to support the storage of mutable data. This could be useful as a way for stateful services to store their state persistently.

3.4.4 Service deployment

The main part of our system, which uses all the others. It must allow the deployment of services, ensuring their availability. It uses publish/subscribe to detect events and perform self-* actions, the persistence module to store the executables and data files necessary for service execution, the resource prospector to find suitable nodes to allocate the services, as well as the lower layers. It must support stateless services, as well as stateful services with some limitations. These limitations will be linked with the resources provided to the community and the availability and efficiency required for the services. There is an obvious trade-off between the consistency of replicated state and the availability of data. For example, services can quite easily use the persistence module to store their state, and internally manage any conflicts and consistency problems that arise. If they want the system to provide strong consistency guarantees to this state, though, performance will suffer. The best choice for this trade-off highly depends on the type of service being deployed, and hence different options will be made available for users.

- *Service deployer:* It must decide the correct way to execute a service (mainly, number of replicas) in order to ensure the desired level of availability for each service, considering the resources currently available to the group as well as the consistency requirements of the service.

- *Service allocator:* It is in charge of choosing the nodes where services will be executed. It will use the resource prospector to find nodes where the service can be executed. After the nodes are chosen, the execution will be started using the protocols defined in the resource layer.

- *Service Self-healing:* It reacts to failure of nodes and reallocates services from failed nodes.

- *Service Self-tuning:* It must decide when to create or destroy replicas of a service, to maintain its individual availability as well as the performance of the cluster.

- *Service Self-configuration:* It must react to nodes joining and leaving to reallocate services in the most convenient way.

- *Service client:* It must allow users to access deployed services. The system must ensure that messages will reach the desired service. As we want to provide support to multiple patterns of communication, we must allow, as well as transparent access to the service as a whole, access to a certain replica, or to replicas which fulfill certain conditions. Moreover, users should be able to choose between reliable or best-effort message passing.

4 Related work

First of all, we must say that our general purpose of sharing the computational resources of users is one of great importance in the scientific community as well as in business. Hence, many research and implementation efforts have been done in this direction. Possibly the most important and relevant contribution in this area has been the concept of the grid [1, 2]. It has brought an standardization effort that has materialized in the Globus Toolkit [4] as its main representation. Another paradigm that has pursued sharing between users and has been taken as a model for our system is peer-to-peer, which breaks with the classic client/server model. Many research has been done about this paradigm, and, as we have discussed in the previous sections, many of the results obtained directly affect our work. As related work on specific functionalities has already been presented on the corresponding section, we will not further discuss it here.

Finally, as we aim to create a system for service deployment, we must say that there are also a few systems which do similar tasks. LaCOLLA [26] is a middleware for decentralized collaboration with the capability of service deployment [27], which is very similar to the one presented here. The main difference is the fact that LaCOLLA targets small groups, and service deployment is only one of its functionalities, not the main one. The system presented here, instead, is designed for the main purpose of service deployment, but is also thought to scale to large groups. Snap [24] deploys web services over a DHT, and creates replicas of a service on demand, stopping these replicas when demand decreases. However, it assumes all nodes are equal and able to execute any service. We dont want to make this assumption, allowing nodes to participate without offering services, and services that can only be executed in very specific nodes. Another system called Chameleon [25] deploys

services in a cluster while trying to maximize its utility (calculated from a value assigned to each service and its performance). It also assumes that any nodes can execute any service, and only one at a time, simplifying the estimation of the utility function. Although this approach is interesting, it seems fit for a cluster of computers with a single manager, who can decide the utility of each service, in contrast to our system, that aims to allow individual users to deploy their own services.

5 Conclusions and future work

We have presented an architecture for a system which allows the decentralized deployment of services in a group of computers with a high degree of individual autonomy, which results in unexpected connections and disconnections. We have modeled its architecture after the layered architecture defined for grids, and shown that the lower layers can be easily implemented with existing technologies and protocols. The upper layer, however, which comprises the main functionalities required by our system, can give place to the creation of novel mechanisms. We believe that such mechanisms, as well as giving us the necessary tools to fulfill the requirements of our scenario, could later be reused in many systems, as their encapsulation in components permits the definition of general mechanisms.

Our future work consists in better refining the requirements of our upper layer and creating suitable mechanisms to implement the corresponding components, namely the resource prospector, the persistence module and the service deployment module. After these are defined, implemented and tested, we will create a complete system by combining our novel components with the ones in the lower layers, implemented through existing protocols and technologies.

References

[1] Stockinger, H.. Defining the Grid: A Snapshot on the Current View. To be published in the *Journal of Supercomputing*. Springer, 2007.

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

[3] Foster, I., Kesselman, C., Tuecke, S. The anatomy of the Grid. Enabling scalable virtual organizations. *Intl J. Supercomputer Applications*, 2001.

[4] http://www.globus.org

[5] 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, Second Quarter 2005

[6] Ali Ghodsi. Distributed k-ary System: Algorithms for Distributed Hash Tables. PhD dissertation, KTH-Royal Institute of Technology, October 2006.

[7] Yeo, C. K., Lee, B. S., and Er, M. H. A survey of application level multicast techniques. *Computer Communications*, 27(15):1547–1568, 2004.

[8] Thain, D., Tannenbaum, T., Livny, M. Distributed Computing in Practice: The Condor Experience. *Concurrency and Computation: Practice & Experience*. v. 17, Issue 2–4, pp. 323–356, February 2005.

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

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

[11] Eugster, P. et al. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, Vol. 35, Issue 2, pp 114-131, 2003.

[12] Pietzuch, P.R. and Bacon, J.M. Hermes: A distributed event-based middleware architecture. In *Proc. DEBS02*, July 2002.

[13] Feller, M., Foster, I., and Martin, S. GT4 GRAM: A Functionality and Performance Study.

[14] Castro, M. et al. Scribe: A large-scale and decentralized applicationlevel multicast infrastructure. IEEE *JSAC*, 20(8):1489–1499, October 2002.

[15] Ratnasamy, S., Handley, M., Karp, R., and Shenker, S. Application-levelMulticast using Content-Addressable Networks. In *Proc. NGC* 2001.

[16] Dabek, F. et al. Towards a common API for structured peer-to-peer overlays. In *Proc. IPTPS* 2003.

[17] Triantafillou, P., Aekaterinidis, I. Content-based Publish/Subscribe Systems over Structured P2P Networks. In *Proc. DEBS*'04, May 2004.

[18] Tam, D., Azimi, R., and Jacobsen, H.A. Building Content-Based Publish/Subscribe systems with distributed hash tables. In *DBISP2P*, Sept. 2003.

[19] Van Renesse, R. and Bozdog A. Willow: DHT, Aggregation and Publish/Subscribe in one protocol. In *Proc. IPTPS* 2004.

[20] Yalagandula, P., Dahlin, M. A scalable distributed information management system. In *SIGCOMM* '04, ACM Press 379–390, 2004.

[21] Bhagwan, R., Mahadevan, P., Varghese, G., Voelker, G.M. Cone: A Distributed Heap Approach to Resource Selection. UCSD Technical Report CS2004-0784.

[22] Rao, A. et al. Load balancing in structured P2P systems. In *Proc. of IPTPS* 2003.

[23] Karger, D. R. and Ruhl, M. Simple efficient load balancing algorithms for peer-to-peer systems. In *Proc. of IPTPS*'04, pp 131–140, February 2004.

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

[25] Adam, C. and Stadler, R. Implementation and Evaluation of a Middleware for Self-Organizing Decentralized Web Services. In *Proc. IEEE SelfMan* 2006, IEEE CS Press, 2006.

[26] Marquès, J.M., Vilajosana, X., Daradoumis, T., Navarro, L. Lacolla: Middleware for self-sufficient online collaboration. IEEE *Internet Computing* 11 (2) 56-64, 2007.

[27] Lázaro, D., Marquès, J.M., Jorba, J. Decentralized service deployment for collaborative environments. In *Proc. CISIS*07, pp. 229-234, 2007

[28] Anjomshoaa, A. et al. Job Submission Description Language (JSDL) Specification, Version 1.0, Global Grid Forum, GFD-R-P.056, November 2005.

[29] Foster, I. et al. The Open Grid Services Architecture, Version 1.5, Global Grid Forum, GWD-I.080, July 2006.