Achieving Load Balancing in Structured Peer-to-Peer Grids

Carles Pairot¹, Pedro García¹, Antonio F. Gómez Skarmeta², and Rubén Mondéjar¹

¹ Department of Computer Science and Mathematics, Universitat Rovira i Virgili
Avinguda dels Països Catalans 26, 43007 Tarragona, Spain
{cpairot, pgarcia}@etse.urv.es

² Department of Computer Engineering, Universidad de Murcia
Apartado 4021, 30001 Murcia, Spain
skarmeta@fcu.um.es

Abstract. In this paper, we introduce the concept of a structured peer-to-peer grid and present our contribution to this new world by means of DERMI, a wide-area decentralized object middleware. In addition, we focus on the design and implementation of a load balancing facility by using the functionalities provided by this middleware. We present two different approaches to achieve load balancing in our system: a completely decentralized solution by using the *anycall* abstraction, and a distributed interceptor-based one. Each of them can be used in a wide variety of scenarios, depending on our needs.

1 Introduction

Grid and Peer-to-Peer (P2P) Networks are two concepts which may seem different at first sight. There exist no exact definitions that allow us to compare their similarities and differences in an unambiguous way. Nevertheless, the general idea seems to split both worlds into those cases where infrastructure is used to allow seamless access to supercomputers and their datasets (Grids), and those which enable ad hoc communities of low-end clients to advertise and access the files on the communal computers (P2P).

Trying to merge the best of both worlds, the concept of a P2P Grid was devised [1]. A P2P Grid contains a set of services including the services of Grids and P2P networks and support naturally environments having features of both limiting cases. In P2P Grids' architecture, Web services play a very important role. In addition, there is an event service which links these Web services and other resources together. In P2P Grids, everything is a resource, and they are exposed directly to users and to other services. An important feature is that these entities are built as distributed objects which are constructed as services allowing their properties and methods to be accessed by a message-based protocol.

Normally, these P2P Grids have been built on top of unstructured P2P networks. This kind of networks at the extreme case are exemplified by the Gnutella-like systems, where all participants join the network in an anarchic way, and messages are flooded all over the network (normally messages are only sent a few hops away from

the origin, thus being extremely inefficient when searching for a precise resource). Other alternative techniques to flooding include the use of Random Walkers, or expanding ring techniques. However, they do not elegantly solve the problem of resource location in a deterministic time slot, being it impossible to know how long it will take to find a concrete resource, if it is found.

Many research efforts have lately been directed towards building structured P2P networks, also called overlay routing networks, which are the base for constructing distributed hash tables (DHTs). These systems provide a self-organizing substrate ideally suited for large-scale P2P applications. A wide variety of decentralized services can be built on top of these overlays. Examples include application-level multicast, network storage, and content distribution. Being structured overlays, applications can locate any resource in a small number of network hops (typically $O(\log n)$), and they require small-sized per-node routing tables.

Relating P2P Grids with structured peer-to-peer overlay networks, we have developed a structured P2P Grid middleware platform called DERMI [4]. DERMI is a distributed event-based object middleware built on top of a DHT-based P2P overlay network. It is a first step towards building a complete P2P Grid middleware platform, providing many services. DERMI provides a solid building block for the development of wide-area distributed applications. DERMI implements many of the functionalities required for a P2P Grid, including an event service which glues the distributed objects representing resources.

In this paper we introduce the term of structured P2P Grid, which we define as a P2P Grid whose network substrate is based on a structured DHT approach. This means that the P2P Grid itself is built on top of a Key-Based Routing (KBR) infrastructure. The main advantage of this approach is that these kinds of Grids can benefit from better message routing and its inherent locality properties.

The main contribution of this paper is describing two load balancing mechanisms, which we have designed by using the different services provided by DERMI.

The paper is structured as follows: we start in Section 2 with a brief description of DERMI and the main services it provides. Section 3 describes the mechanisms we have designed to achieve load-balancing using DERMI. Next, we discuss the contributions of DERMI to the P2P Grid world and continue exploring existent work related to our system. Finally, we present a summary and an outline of future research.

2 DERMI Overview

DERMI is a distributed event-based object middleware built on top of a decentralized DHT-based P2P overlay network. It models method calls as events and subscriptions under an underlying publish/subscribe layer also built on top of the overlay. We have built a prototype implementation using Pastry [5] as the routing P2P substrate and Scribe [2] as its message oriented middleware (MOM) infrastructure. However, any other DHT-based overlay platform like Tapestry, Chord or Kademlia could have also been chosen to achieve our objective.

DERMI mimics Java RMI programming paradigm and provides a *dermic* tool to generate both stubs and skeletons for our remote objects, which transparently manage object publications/subscriptions and its inherent notifications. Indeed, it provides

many features already found in Java RMI such as remote exception handling, pass by value and by reference, and dynamic class loading.

Nevertheless, DERMI is quite different from RMI. Naturally, our solution is aimed at wide-area environments, whereas RMI can suffer from bottlenecks due to its centralized client/server nature. Besides, another existent difference is in the communication layer located between stubs and skeletons. While in conventional RMI, a TCP socket is established between the caller (stub) and the callee (skeleton), DERMI stubs and skeletons both use the event service by making subscriptions and sending notifications in order to communicate the method calls and their results.

DERMI provides a set of features, including object mobility, replication, caching, and discovery services. Further, it provides the typical synchronous and asynchronous remote procedure calls. However, two new invocation abstractions are provided, namely *anycall* and *manycall*.

The *anycall* abstraction benefits from the *anycast* primitive provided by the event service (Scribe). Therefore, an *anycall* means sending an *anycast* notification to the group, which will make the sender's closest member in the network to answer, as long as it satisfies a condition. This feature can be very useful for having object replicas of any service. If a client wants to have access to this service, it will *anycall* to the object replicas' group and one of them (the client does not care which one) will answer.

The *manycall* abstraction is a variation of the *anycall*. It basically works by sending a *multicast* message to the members of a group, i.e. the message is sent to several group members. It is routed further until enough satisfying members are found to satisfy a global condition.

Moreover, DERMI implements a decentralized object location service which allows us to locate object references in order to be able to execute remote procedure calls on them. Object handles are inserted in the DHT using as their key a hash on their object identifier. To look up an object we simply hash its object UID and subsequently obtain its handle. At this point we can already call its methods.

Finally, DERMI provides distributed interception facilities. Distributed interception allows us to apply connection-oriented programming concepts in a distributed setting. With this service, it is possible to reconnect and locate type-compatible interceptors at runtime in a distributed application. In order not to change the subscriptions of both interceptor skeletons and intercepted remote objects each time a new interceptor is added or removed, we extended our event service's classes (Scribe) so as to natively support this feature. This service is very useful for very dynamic Aspect Oriented Programming (AOP) environments.

For more information on DERMI's design and implementation of these services, please refer to [4].

3 DERMI's Load Balancing Mechanisms

Load balancing can be defined as *distributing processing and communication activity evenly across a computer network so that no single device is overwhelmed*. Load balancing is especially important for networks where it is difficult to predict the number of requests that will be issued to a server. For instance, busy web sites typically employ two or more web servers in a load balancing scheme. If one server starts to get swamped, requests are forwarded to another server with more capacity.

In distributed objects technology, the problem is the same: a resource which provides a service can become overwhelmed by many clients. In such case, this resource is clearly a centralized hot spot, becoming a single point of failure. To avoid such hot spots, replication mechanisms are taken into account by making the object server replicated. Nevertheless, this is not the solution, if there is no awareness of which replica to choose depending on its current load. Therefore, apart from having multiple replicas of the same service, we need a mechanism to choose one or another according to certain parameters which may include server load or network bandwidth, to name the most common ones.

As an example, we could imagine a service like a video broadcast. If we expect our video server to be joined by an important amount of clients, we should consider replicating it. However, this does not solve the problem yet. We require some mechanism to decide which replica is to be assigned to each client that connects to our video broadcast service, depending on certain criteria (in this case, network bandwidth, or server's node stress). Naturally, this assignment process must be made transparent to the client.

Having described the problem, and keeping in mind the services DERMI offers, we present the load balancing facilities we have built on top of our middleware as follows.

In DERMI we can use two different methodologies to achieve load balancing. Before describing them both, we outline the common background they share. Firstly we must have our object server replicated among other nodes. Notice that DERMI provides us with mechanisms to do this, which are transparent to the developer. By doing so we ensure that we have all the needed replicas joined into a multicast group, and that any changes within the object server will be easily propagated to their replicas by sending event messages to the group (which will be received by all members, in this case all replicas). Once we have the object server's replicas joined into a multicast group, we can opt for two load balancing options, which are described as follows.

3.1 Decentralized Load Balancing Management

Decentralized load balancing management refers to using the *anycall* primitive to achieve load balancing purposes. The working principle for this methodology is totally decentralized, meaning that we do not have any centralized authority that knows the load of each replica.

Instead, we will be efficiently asking for the client's nearest replica to serve our request. If the replica feels it is not overwhelmed, it will serve us. On the contrary, it will answer that our request cannot be served. In the latter case, and by following the *anycall* behaviour, another object replica will be contacted and so on.

One scenario where this load balancing scheme would fit very well is for example in a distributed simulation of some kind. Imagine we have a group of servers waiting for data chunks to analyze. These chunks could be provided to them by using the *anycall* approach: the chunk would be delivered to the closest server, which would check if it has already reached its maximum number of concurrent data chunks to analyze. If this is the case, it delegates processing to another server, and so on; else it starts the chunk's analysis. A diagram showing this process can be seen on Fig. 1.

Fig. 1. Decentralized and Interceptor-based load balancing example

Notice this load balancing system is only valid when dealing with nodes that do not suffer network congestion. If nodes themselves would become overloaded in terms of network messages, they would be saturated and even unable to respond to our anycalls. This would prevent our decentralized load balancing mechanism to work properly.

3.2 Interceptor-Based Load Balancing Management

To deal with possible network congested nodes, we have another alternative to achieve load balancing: Interceptor-based load balancing management. In this case, we make use of our distributed interception mechanism. We will have a special interceptor node, which will be aware of each server replica load. This idea is similar to the *agent* concept found in NetSolve [1]. Particularly, we add an interceptor object to the multicast group of the object server's replicas. This means that when any client wishes to call a server's method, such call will be transformed into an event that will be sent to the multicast group itself, being automatically intercepted by the distributed interceptor.

The interceptor will, in turn, be aware of the load of each replica and make a direct call to the replica whose load is adequate to solve the client's request. All object replicas will inform periodically about their possible congestion state to the interceptor, by means of sending special events to the group. These events will be captured and stored by the interceptor itself, thus making it aware of the overall replicas' status. Notice the interceptor will in fact know about each replica's capacity for serving requests, and will decide which will serve the current client's RPC. Naturally, this approach can incur in being the interceptor itself a single point of failure: if the interceptor node fails, load balancing fails as well. As explained in [4], we can use replication mechanisms for the interceptor itself, thus assuring correct load balancing even in case of failure.

Clearly, if any of the replicas becomes overloaded in terms of network messages, the interceptor would be aware of this fact, and it would thus route no more messages to that node. Notice in such case, load balancing would continue to work properly, in contrast to the problem we have explained in the decentralized alternative. However, in this case, we have an only entity which controls all the load balancing process. If we wish to distribute this task among all servers, we should use the first approach (decentralized).

One scenario where this kind of load balancing scheme proves to be useful is for the video broadcast example explained before. Clearly, the broadcast of a video stream could overwhelm a server node's network adapter, and it would thus be unable to respond to anycalls (if we used the decentralized load balancing system). Nevertheless, if we use the interceptor approach, the interceptor itself knows the state of all available servers and tells the client which the most appropriate for him to connect is. A scheme for this procedure is shown in Fig. 1.

4 DERMI and P2P Grids

In the introduction we have defined the term structured peer-to-peer grid and have stated DERMI as a middleware platform built on top of such architecture. However, does DERMI provide the majority of services that a P2P Grid should offer? The answer is mainly yes. However, it leaves out a few features that P2P Grids do clearly support. One of these is web services. Nevertheless, the implementation of web services in DERMI is relatively easy. Stub and skeleton generation can be modified so as their public interfaces are defined and described using XML; and they can, for example use SOAP as their access protocol, thus achieving the transition from pure distributed objects to web services. We consider DERMI's services (*anycall/manycall*, *decentralized object location*, and *distributed interception*) to be an interesting and innovative approach to the P2P Grid world. This kind of grids will take advantage of DHT properties such as locality awareness and a better routing infrastructure.

5 Related Work

Existent RPC-based solutions in the Grid world are typically built using the GridRPC API [6]. This API was designed to address one of the factors that hindered widespread acceptance of Grid computing – the lack of a standardized, portable and simple programming interface. Since the GridRPC interface does not dictate the implementation details of the servers that execute the procedure call, there exist different implementations of the GridRPC API, each having the ability to communicate with one or more Grid computing systems.

Two well-known implementations of GridRPC are: one which lies on top of NetSolve [1], and Ninf-G [1], which is a full reimplementaion of Ninf on top of the Globus Toolkit [8].

In NetSolve, there exist 3 main entities: the *client*, the *server* and the *agent*. Clients and servers model the typical behaviour in RPC systems, whereas agents maintain a list of all available servers and perform resource selection for all client requests as well as ensuring load balancing of the servers. Notice the interceptor approach used in DERMI to solve load-balancing issues resembles of a NetSolve agent. Further, fault detection and recovery is managed in a way that is transparent to the user. The agent keeps track of the status of all available servers so that in case of a problem, the agent can choose a new server to handle the problem. In DERMI, if there exist several replicas of an object server. We do not need to have an agent object which monitors

the status of these servers (in case we use decentralized load balancing management). If a server goes down, the others will be unaffected and so will be the clients' requests. Our object replicas belong to a multicast group and clients *anycall* to that group, meaning that any (or the most adequate, depending on certain criteria) of the remaining object servers will answer.

Ninf-G is a GridRPC system built on top of the Globus Toolkit. Globus provides Grid's lower-level features such as authentication, authorization, secure communication, directory services, and many others. Nevertheless, the Globus Toolkit alone is insufficient for programming the Grid at higher-level layers. Ninf-G is a full reimplementation of Ninf. This system provides a mechanism to *gridify* an application or a library by using the Ninf IDL language to generate the necessary stubs. The executable is registered into Globus' MDS, and Globus-I/O is used for communication between client and server.

In such GridRPC systems, no efficient one-to-many asynchronous calls can be made. They can surely be emulated by means of n sequential calls incurring in a penalty overhead. In DERMI we have the notion of a communication bus that allows us to have efficient one-to-many asynchronous calls by using the underlying event service.

Both DERMI and Ninf-G have a directory service. DERMI's directory service is totally decentralized; whereas in Ninf-G MDS is used. MDS consists of two layers of LDAP servers and Grid Index Information Service (GIIS), which manages project-wide information; and Grid Resource Information Service (GRIS) which is responsible for site local information.

The latest version of the Globus Toolkit (3.0) introduced the concept of *Grid Services*. There is a clear resemblance between these Grid Services and Web Services in P2P Grids. However, by definition a service is a *Grid Service* if and only if it adheres to the OGSI specification. In DERMI we could transform our pure distributed objects into web services, in the way we have described in section 4. In addition, we believe that the OGSI specification and the Grid community could benefit from the DHT properties such as locality and even the services DERMI offers such as *anycall* and the *manycall* abstractions.

6 Conclusions and Future Work

Peer-to-peer Grids have until now been constructed on top of unstructured P2P networks. This kind of grids presents very interesting features including web services and an event service, among others. However, we believe that they can benefit from the efficient routing primitives available in structured P2P networks built on DHT-based architectures.

We have defined the term of structured P2P Grid term as a P2P Grid whose P2P network substrate is based on a DHT topology. We have presented a structured P2P Grid in the form of DERMI, an event-based object middleware built on top of a DHT. We have also introduced two mechanisms to achieve load balancing in our architecture, which we believe that can be a solid building block for the development of future applications on top of P2P Grids.

As future work, we are on the way to compare both described load balancing approaches by means of simulation results. By doing so we expect to provide exact

data on when to use the decentralized method, or the interceptor-based one. Additionally, we are currently researching on adding support for shared sessions and artifacts, which mean the first step in creating a collaboration framework to provide wide-area CSCW services.

Acknowledgements. This work has been partially funded by the Spanish Ministry of Science and Technology through project TIC-2003-09288-C02-00.

References

1. Berman, F., Fox, G.C., and Hey, A.J.G., *Grid Computing: Making the Global Infrastructure a Reality*, John Wiley & Sons, Ltd, England, 2003.
2. Castro, M., Druschel, P., et al, "Scalable Application-level Anycast for Highly Dynamic Groups", *Proc. of NGC'03*, September 2003.
3. DERMI Website, <http://ants.etse.urv.es/DERMI>
4. Pairot, C., García, P., Gómez Skarmeta, A.F., "DERMI: A Decentralized Peer-to-Peer Event-Based Object Middleware", to appear in the *Proceedings of ICDCS'04*, Tokyo, Japan.
5. Rowstron, A., and Druschel, P., "Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems", *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pp. 329-350, November 2001.
6. Seymour, K., Nakada, H., et al, "GridRPC: A Remote Procedure Call API for Grid Computing", *Proc of GRID 2002*, Springer, Baltimore, MD, USA, pp. 274-278.
7. Terpstra, W., Behnel, S., et al, "A Peer-to-peer Approach to Content-Based Publish/Subscribe", *Proc. of DEBS'03*, 2003.
8. The Globus Toolkit, <http://www.globus.org>.