Grid computing

Mattia BUCCARELLA

2010 February, 24th

1 Abstract.

In this document we will deal about the principles of the distributed computing. In particular, this paper wants to emphasise some concept about the grid computing technology.
In this sector, will be important being precise in terminology; that's the reason why we start to define the meaning of some words:

- *distributed* indicates something involving more than one calculator as performers of a computation. These computers are interconnected through a network that makes them communicate in order to share information they stored, useful to achieve one common goal;
- *enabling* means something has the capability of making practically and easily possible realize our ideas;
- a *platform* is the equipment needed to a particular purpose, where by "equipement" we mean all the stuff we want to combine together to achieve the result (i.e.: instruments, GPS, satellities, mobile devices, ...).

We mentioned above the expression "common goal". What does it mean? What do we want to do with all these things combined together? The main topic under consideration is finding a solution to a *large scale problem*. "Large scale" means that we are treating something that can not be solved on a single machine or on a little network of computers. Thus, in some way, we have to widen our perspective in order to consider a very extended solution, that requires a very large number of equipments. Just to make some example, we can deal with two types of situations: what happens in research and what happens in real production.

- In *research* it is possible to think about several situations whose requirements are shed along the whole world (as technological stuff, specific environments or things that cannot be reproduced, etc...) or needed for simulations of large- or world-scaled mathematical models (weather forecast, avoidance of Earth's quakes, etc...);

- on the other hand, in the *production* environment (tipically industrial or oriented to the commerce), there are problems that require a lot of calculators in order to perform operations, on-line analysis of petabytes of data or space to store huge quantities of information (as an example, just think about the quantity of 2.0-web traffic generated by a single facebook user or about the quantity of space needed to store biological information, etc...).

Thus we are dealing with numbers like 10^{19} bytes per year generated or, in the case of Google, also 10^{19} bytes per day, maintaining that the query latency must remain unchanged (about 0.1 seconds)!!!

2 An experiment.

However, even if we consider the possibility to use equipments shared from the whole world, we have to take into account that there are several difficulties bounded to the structure of the solution itself. To give a more precise idea of above, we consider the following: someone is asking us to solve the problem of printing on a paper (or on a book, in our case) the first 10^{20} prime numbers. We don't want to consider the computational aspetcs of the problem, hence we can assume that we are given a best algorithm for the primality check, a best way to parallelize the whole tasks, and so on, and so forth. We rather want to understahd how to approach the problem. Certainly we:

- agree to collaborate;
- set our computer connected on a network;
- install the needed software;
- run the program;
- wait for the results.

All these operations seem to be quite simple, but in real world, all this process has to solve some "structural" problem. For example, it may be not so simple, for an user, to agree to connect his computer on a network, because it is possible that he does not trust some user or the network itself. Similarly, he should install all the software, assuming that it is not dangerous for his system (but this is not the typical behaviour of a computer scientist). However, assuming that the user above wants to trust everything and everyone, what does it happen if after six days of computation, the software administrator discovers that more computers are needed? Or what does it happen if some IP address get lost? At this list we can add more and more questions that are difficult to answer, hence the conclusion is that all the things that people have to do are not so simple as they seemed earlier. Indeed when people think about a distributed software, they tipically have to solve problems about security, sharing of resources, dinamicity, fault tollerance, and so on and so forth.

3 Tools we need.

Before starting to talk about grids, even in this case, it is really important to precisely define our terminology. We basically have to deal with the following different things:

- resources;
- protocols;
- services;
- APIs;
- SDKs;

and we shall formally define each entry.
A *resource* is an entity that may be shared. According to this definition, we don't have to think that it must necessarely be a physical object; clearly it could be an object, as a CPU, an hard disk, but it can also be a software, a filesystem, etc. It is more important to say that a resource is something defined in terms of interfaces and capabilities. This means that the users, who want to use a particular resource R_i, only have to know what kind of job R_i can perform and what is the way of make R_i do it. As an example we can assume that R_i is a filesystem which allows users to do read-only operations. Hence, its capabilities could be:

- open a file or a directory x;
- read the content of x;
- release x (if it is a file).

An interface of above (in a one-to-one correspondence) could be the following set of identificators:

- open;
- read;
- close;

and this could conclude the description needed to use R_i.
A *protocol* is a set of rules describing a format that resources have to be compliant to in order to use messages to communicate. Tipically a protocol is a good one when it defines rules for doing one single thing: defining more than one thing can be diffficult and it can incurr in some mistake. Also a protocol has to be defined in terms of its capabilities and interfaces, hence it is easy to conclude that a protocol is one particular resource. The capabilities and the interfaces of a protocol are called *API*s.
In the end, a *service* is an implementation of a server-side protocol providing

a set of capabilities. If, for example, we consider the HTTP protocol, we have that a web server implementing it is the HTTP service.
An *API* (Application Programming Interface) defines the logic and the semantics of a set of functions or functionalities without providing an implementation of them. This latter is a duty of an *SDK* (Software Development Kit). Clearly, given an API, it is possibile to have more than one implementation, that is more than one SDK; for example we can consider the Java language, hence we have the Sun's implementation, the Blackdown's one, the IBM's one, etc. Furthermore, when we talk about API/SDK, we also have to talk about *standards*. This strange word is important because standards are capable to achieve portability of applications between systems of a different type. That does not mean that without standards it is impossible to port something to another platform, but without them it could become a very difficult process.

4 How to use these stuff?

We can define several roles and assign each of them to the entities componing our distributed environment. Tipically we deal with two important entities: the *service provider* and the *service requestor*. The first one may be an HTTP server as we have seen above or a different service provider. The requestor is the entity that wants to use the provided service under consideration. However, for the current state-of-the-art, it is not enough to have these two components: how can the requestor know that some service exists? Another entity, the *service registry*, answers this question by providing a list of available services to the requestor. After the requestor has discovered the service that is of its interest, it will be able to use it without any problem. This sequence of operations is performed thanks to existing web services that provide interactions between these three entities. In particular, an entity can use the following web services:

- *UDDI* to ask the registry about available services and their addresses in order to discover and use them;
- *WSDL* to retrieve a description of a particular service (compliant to a standard human-readable XML format);
- finally *SOAP* to perform an invocation on a service.

These three services communicate with each other by using XML messages and these messages are tipically transported over HTTP.
The latter point needs a bit of explainations. We are talking about one entity C (possibly located in Italy) that wants to use a service s provided by another entity S (tipically located in the USA). How does it work? Well, provided that C has already discovered s and retrieved its description through UDDI and WSDL, it's now time to perform an invocation. Either C and S must have an environment that describes them to each other. This thing is called "STUB" and it can be informally defined as the fake C or S. More precisely, C will call (locally) its STUB. This invocation generates a SOAP request that will reach

S in order to invoke the service s. Thus this invocation gets serialized into a sequence of bytes (*marshalling*) and sent to S. The thing that will receive this sequence of bytes is the server's STUB, which has the duty to deserialize that sequence (*unmarshalling*), interprete it and finally perform the real request to the real server. After the request is served, the game is repeated again in the reverse direction in order to give back to the client the answer he wanted.
The reason why these web services are used in order to perform communications is that, in general, the interlocutors may be substantially different from each other. Using a normal communication layer (as TCP) may cause different interpretation of the sent data, due to these differences. The problem could be solved by giving to each entity the specifics of all of its interlocutors, but this is a very long and complex process and it is not dynamic enough (just think of what happens if a new interlocutor joins the network under consideration). Since the SOAP messages are XML messages, they exploit their general and always-working format because XML is independent of the architecture. Thanks to this assumption we solve readly the problem.

5 Grid computing.

We mentioned a lot of stuff because they are the fundamental components to set up a grid computing. Indeed, first of all we need a set of **shared resources** (like computers, data storage, networks, etc...) and these resources are shared and then used according to a certain set of rules orchestraring negotiations, payments, etc.
Thus, we are thinking about **solving problem in a coordinated way**, and we will see that these problems will be solved in a dynamic and multi-institutional **virtual organization**. This concept needs to be discussed a bit more in detail.
A *virtual organization* is a dynamic set of individuals and/or institutions that share a goal and a set of rules. The keyword "virtual" means that there is no physical entities identifying or formalizing this concept. As a simple example, we can consider a class of students as a virtual organization. Indeed all the students have to share some didactic rules and they, for example, have to submit some work realized by collaborating with each other (i.e. they have to write LaTeX notes for a course in a distributed way). A virtual organization is also characterized by the fact that it may vary in size, scope, duration and structure (for example, think about some student, of the class above, who gives up). Furthermore, in a virtual organization, resources are highly controlled.
There is not a strict definition of "grid computing". The Wikipedia's definition of Grid computing is the following:

Definition combination of computer resources from multiple administrative domains applied to a common task, usually to a scientific, technical or business problem that requires a great number of computer processing cycles or the need to process large amounts of data.

Our definition could be:

Definition Grid computing is all about achieving performance and throughput by pooling and sharing resources on a local, national or world-wide leve.

The advantages of using a grid are:

- clearly that each user can access the resources simply by plugging his computer in the network, without having any problem (as legal problems or other constraints that people typically have when they use physical things);
- on-demand services are dinamically coordinated and combined with each other, and furthermore this dinamicity includes the capability of adding other new resources applying the minimum effort;
- components are autonomically managed, hence the complexity of the structure is completely invisible to the users that are connected to.

In order to maintain these properties, however, some characteristics are needed. The first thought goes to the security. Indeed also this kind of structure needs protection, especially intrusion detection, fault management, and so on. This mechanism means that there should be authantication systems (with authorizations, etc...) in order to access remote data, possibility to discover resources, get their characterizing description and finally use them. This is not enough: to solve a problem in a distributed way, the system needs distributed algorithms, distributed management of resources, and all these things put together make the payment system (if any) more complicated to be handled too.
The architecture of a grid can be defined independently of its services and resources. We can divide protocols in levels:

- fabric layer;
- connectivity layer;
- resource layer;
- collective layer;
- application layer.

The *fabric layer* is the set of services and capabilities and protocols that allow to control locally the resources. Since these resources need some way to communicate with each other, then we have the *connectivity layer* that is the way of making these resources interact. At this level we also have security issues in order to guarantee that the communications above are secure. When we need to do something, or use some resource, we are in the *resource layer*, and there resources are combined together with the *collective layer*. All these layers make possible have the final layer we are interested in, that is the *application layer*.
These stuff need to be implemented in some way. The *hourglass model* gives this implementation. For the current state-of-the-art, there are no standards implementing a grid computing architectures; each grid has its own implementation.

However, the Globus Tooklit (some basic information on the wikipedia's page) has emerged as a de-facto standard that can be followed to achieve several and important connectivity, resources and collective protocols.

6 Using grids.

Now the question is: how grids are used? Which kind of problems they can solve? First of all, grids are used to perform collaborative jobs. For example, in the aeroplane building problem, tipically people and different organizations work together according to the specific role each one can provide to help the others. Other examples could be given by weather simulation, mathematical computations, etc. Grids are also used to solve problems that have a really huge amount of data as input (this is the typical situation we have when we think of google, yahoo and other contexts in which petabytes of data must be stored somewhere and processed in a few fractions of second).