Execution Environments for Distributed Computation Issues

Sergi Baila Vicenç Beltran Julita Corbalan

Toni Cortes Marta García Íñigo Goiri
Jordi Guitart Ferran Julià Jonathan Martí
Jesús Malo Ramon Nou

December 18, 2007

Abstract

This report is the result of a survey done by master students on different areas. Taking into account the interest of this issues, it has been decided to publish it as a Research Report.

Chapter 1 is focused on replica management in Grid environments. The chapter is divided in sections corresponding to key aspects of the state of the field of the topic. Firstly, strategies to select and locate file replicas among the Grid nodes are commented. Secondly, some techniques to create and delete replicas are described. Then, it is defined the problem of handling consistency policies among these replicas and some solutions for data grids. Finally, it is introduced a section to discuss about other concepts such as the selection of the optimal placement for file replicas, or how the replication mechanisms influence the job scheduler.

Chapter 2 focuses in a special execution environment: virtualization. This method allows abstracting overlaying resources creating a very useful and innovative way to work in computing. Different virtualization techniques and some products that implement these are classified and described. It also talks about some implementation details including new technologies such as Intel VT-x. In the last part, virtualization real applications are described and its advantages respect traditional environments.

Chapter 3 the increasing complexity, heterogeneity and scale of systems has forced to emerge new techniques to help system managers. This has been achieved through autonomic computing, a set of self-* techniques (self-healing, self-managing, self-configuring, etc.) that enable systems and applications to manage themselves following a high-level guidance. This chapter is centered in the self-management

capability of autonomic systems, it pretends to give an overview of the three most popular mechanisms used to achieve self-management, action policies, goal policies and utility function policies. This chapter presents a summary of autonomic systems' architecture and an extended view of the different policy mechanisms analysing the usefulness of each one.

Chapter 4 introduces a new technique for streaming server side events on a web application. Being an extension of the recent AJAX model for web applications Comet is another step towards the confluence between desktop and web applications. The scalability issues that this model present are examined, along with the Bayeux protocol and a glimpse on the current libraries implementing Comet and the servers which have solved the scalability using by means of asynchronous request processing based on non blocking I/O.

Chapter 5 currently distributed applications rely on several layers of abstraction. These layers of abstraction play a critical role in the whole distributed application, thus they are in part responsible for an important part of the whole performance of applications. We will focus specifically on those technologies commonly used in distributed environments like CORBA, RMI, Web Services and their impact in Grid architectures.

Contents

Chapter 1

Replica Management in the Grid

Toni Cortes, Jesús Malo and Jonathan Martí

Abstract

In order to achieve the computational and data requirements for current scientific applications, grids have appeared. Data grids provide geographically distributed resources for large-scale data-intensive applications that generate large data sets while computational grids are focused on cpu-intensive applications.

Grid environments are very dynamic and unpredictable, full of possible outage, disconnections, network splits, high latencies, bandwidth variations and data corruption. Low latency, fast access, fault-tolerance and high availability are goals that data grids must achieve when accessing to such huge and widely distributed data is required. For these reasons, data grids are very complex systems with an enormous set of possibilities and strategies.

This chapter will show the state of the art in replica management. Issues like identification of replicas, propagation of updates, consistency and coherence among replicas have to be taken into account by the replica management system. They must be solved to reach the

best advantages of data replication. Existing approaches dealing with those challenging issues will also be remarked on.

The different schemes for replication and algorithms for replica placement are keys to gain the best performance. In this case, issues like unbalanced distribution of replicas can reduce the good performance of data grids. For these reasons, replica creation, selection and migration are also important. They allow to define suitable schemes of replicas depending of access and age of data, available resources, costs and computational power. Existing approaches are both static and dynamic ones getting a fixed or an autonomic behaviour to data grids. Regarding to dynamic replication, economic models, prediction of use, game theory and bio-inspired algorithms are the most promising approaches.

1.1 Introduction

Over the last few years Computational Science has been evolving to include information management. Scientists are faced with huge amounts of data that stem from four trends: the flood of data from new scientific instruments driven by Moore's Law - doubling their data output every year or so; the flood of data from simulations; the ability to economically store petabytes of data online; and the Internet and computational Grids that makes all these archives accessible to anyone anywhere, allowing the replication, creation, and recreation of more data. Therefore, the term Data Grid has been appeared in the literature to refer to this trend change.

Data Grids provide geographically distributed resources for large-scale data-intensive applications that generate large data sets. However, ensuring efficient and fast access to such huge and widely distributed data is hindered by the high latencies of the Internet. Many studies have been done to address these problems, e.g. parallel access to data, file replication, multicast at application level, etc.

This chapter is focused on ***replication management mechanisms*** in Data Grids.

Mainly, these replication management strategies look forward to achieve the following goals (or a set of them):

- Offer high data availability: data is accessible to anyone from

anywhere.

- Exploit data locality: data is near to where it is needed, so bandwidth consumption is decreased because it is not necessary to get data whenever is needed, and therefore data access latency is improved too.
- Load balance: handling replicas of a file in different nodes helps on improve the overall load balance and also overcomes potential bottlenecks.
- Increase fault tolerance: since data is replicated in several different nodes around the world, it is more likely to be able to recover it.

Therefore, some issues must be taken into account to build replication management mechanisms and to deal with their consequences. Specifically, this chapter will be focused on:

- Replica selection and location among Grid nodes.
- Replica creation strategies (how and where).
- Replica removal strategies (which ones and when).
- Consistency and coordination among replicas.
- Other issues
  - Data replication complexity in Data Grids
  - Replica placement algorithms
  - Scheduling and data replication

Firstly, some strategies are introduced regarding replica selection and location among Grid nodes. The key idea is to introduce the main mechanisms enabling the replication management system to select and localize the best replica given a certain file request.

Secondly, regarding replica creation, the most representative strategies for replication are introduced. Mainly, we are going to focus on approaches based on:

- Predictive algorithms that use data mining over file access histories, so demand of files can be predicted and replication could be started before a file is really requested.
- Economy-based models based on auction protocols, which are used to manage the migration and creation of replicas in order to optimize the overall performance of the system.

Thirdly, we are going to expound replica removal strategies, which are necessary in order to maintain the overall scalability of the system.

Finally, we are going to talk about the consistency and coordination mechanisms. These mechanisms are applied in order to ensure that replicas of the same file are up-to-date and coordinated according different kind of consistency and coherency policies.

1.2 Selection and location of replicas

1.2.1 Replica selection

Replica selection is a challenging issue of current data grids in order to achieve best performance. Choosing the best replica usually means selecting replicas placed in hosts with the fastest links or with lowest latency, but it can also have to deal with user's requirements or others metrics.

Selecting the best replica is a key issue for grids and the chosen strategy modifies the behaviour of schedulers, replicators, the way in what jobs are submitted and the achievable performance. Schedulers have to take account of replica locations when they decide where to place new jobs. Replicators change their decisions about new replica placement for getting a better improvement. Jobs are submitted with different stage in or stage out parameters or, indeed, with parameters using several locations. Of course, using fastest replicas improve the overall system because waits of jobs for data will be reduced. As you can see, selecting good data sources is a key stone in high-performance grids.

Current approaches take care of users' requirements and *QoS* for job submissions and access to data. They are able to do it taking account of different metrics, such as round trip time, bottleneck bandwidth, server request latency, available bandwidth, network proximity,

server load or response time. These metrics allow schedulers and other smart services select the best replica.

In [49], authors introduce a mechanism for replica selection based on given information from *users' preferences* and replica location. It is implemented as a high level service where the main component is the Storage Broker. Storage Broker is a component integrated in every client and is able to process users' requirements by means of Condor's ClassAds [33] specifications and *matchmaking mechanisms* [40]. Storage brokers firstly query to the replica catalog for retrieving the whole set of locations. For each location, they ask servers for their characteristics. Finally the process retrieves data and matches them with the given requirements.

Other more flexible approach is the based on *contracts* [20]. In this approach, users specify requirements with *QoS* binding contracts specified in XML. The system performs an heuristic search taking these contracts into account. Used metrics can be RTT, bottleneck and available bandwidth, system computational load, replica load or host availability. Replicas are organized in replication domains depending of network proximity, which can be calculated by means of the topological distance or the geographical one. Besides replication domains, there are logical domains containing the set of clients accessing to a replication domain, a replication domain, a RLS and a Metrology Server, which analyzes and aggregates a history of metrics collected in a given time-period. Selection is performed in two steps. In the first one, search is done over the logical domain of the client and every non-overloaded replica is selected. Second step consists in filtering *QoS* user's restrictions. If no replica is selected, then a replica with a tolerable *QoS* is selected.

As you can see, selection mechanisms are defined in the core of the replica system. This can be moved to an encapsulated module, such as the Optor component of [29]. Also, a more flexible approach can be taken, allowing users specify their own selection algorithms, as Pegasus does [18]. Pegasus is able to deal with different replica selection algorithms, like random, round-robin or min-min ones. Replica selection is automatized but can be configured to delay it until job submission or taking account of different users' criteria. Decisions are finally adopted following the existing information in metadata servers and replica location services.

Regarding how measures of dynamism of grids are done, in order to get adaptive schemes for replica selection, NWS [50] is commonly used. NWS is a measuring system with prediction capacities over network bandwidths. In [2], replica selection is done based on received information from NWS, i.e., network bandwidth, latencies and prediction. The request manager selects the replica with highest bandwidth among source and target hosts.

Finally, it is remarkable the work presented in [19]. In this case, the proposed approach is avoiding replica selection in favour of taking advantage of parallel transfering from all replicas hosts. This scheme is shown to be efficient because it avoids discrimination among replicas with little differences. Transfers are done from locations given by a RLS and predictions of NWS. In order to get the best profits of parallel transfers, several algorithms are used to adapt size of requested data to dynamism of available transfer rates.

1.2.2 Replica location

Replication of data is a very effective strategy for achieving a good performance in access to remote data. This approach generates several copies of the same data distributed among different hosts. Obviously, benefits of replication can only be achieved if replicas are available, i.e. they can be accessed.

Every new replica requires a name to be accessed. This name can be an URI, a filename, a system unique number, etc. In short, an identifier able to identify the replica inside the whole set of existing replicas in the system. This identifier is the key to locate the replica, so without it replicas are useless because they can't be handled.

Identifiers must fulfill a set of characteristics in order to be effective: they must assure the uniqueness of themselves, avoiding ambiguity in the resolution of replicas location, and they must be resolvable in an efficient way, otherwise they could become a system bottleneck.

Uniqueness and resolution are the reasons because a replica location service or a replica catalog are needed in grids. These services are specialized databases storing locations and names of replicas. They usually allow to query for replicas fitting some characteristics, based on attributes of data.

Without this kind of services, replicas should be accessed or explicitly specified by users or grid applications or, what it's worst, they should assign directly globally unique names to replicated files. Although this task is easy to do in small systems, like a non-distributed file system, when we are talking of hundreds or thousands of hosts with millions of files distributed all over the world, it's a challenging task. Besides the magnitude of the problem, users usually want an easy-to-remember name for their data, so the necessity of services solving aliases to locations is obvious.

One of the most successful approaches dealing with the resolution and location of replicas is the introduced one at [13]. It presents a replica location service (RLS) that maintains and provides access to information about the physical location of copies. This service maps logical file names (LFN) to physical file names (PFN). A logical file name is an unique logical identifier representing the data, while a physical file name is the real name of a replica in a system, such as an URI. The service is formed by two components: the local replica catalog (LRC) and the replica location index (RLI), as shown in figure 1.1. A local replica catalog has the information of replicas at a single site while the replica location index stores and aggregates the information it receives from a set of LRCs. RLI is also responsible for answering clients' requests about LFNs and PFNs, typically of the kind *"given a LFN get a set of PFNs"* or *"given a PFN get the LFN related to"*.

An implementation of the described RLS is presented and analyzed in [14]. Besides the mentioned general structure, this implementation provides soft state updating mechanisms to maintain RLI state and optional compression of soft state updates. Soft state is required to allow RLIs recover from failures without handling consistency issues of persistent states. Local replica catalogs send periodic messages of their state to RLIs, i.e. a list of all logical names for which mappings are stored in an LRC. Soft state information expires after a period of time and it should be refreshed. Because there is a constant consignment of updates to RLIs, it's important to reduce the size of updates packages. For this reason, a compression algorithm was introduced based on Bloom filters [8].

Although the previously described architecture has been implemented in Globus [21] and used succesfully by several scientific projects, such as Earth System Grid [24] and the Laser Interferometer Gravita-

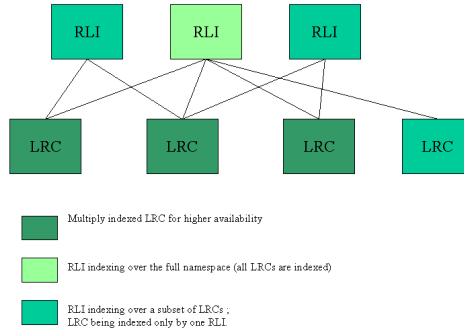

Figure 1.1: Globus Replica Location Service scheme

tional Wave Observatory [36], it has some issues, being the main one that the scheme is too static. Changes on the distribution of servers have to be made with administrator tools and, in case of failures, the system is not smart enough for self-managing. For these reasons, a modification was proposed in [10] (Figure 1.2). In this project, LRCs were kept intact but RLIs were modified, being called now P-RLIs. P-RLIs are able to contact themselves using Chord [46] for message-passing. They conform an overlay network with fault-tolerant characteristics and self-managing capable. The scalability of the system is also increased. With this approach, LRCs only have to communicate with a P-RLI instead of a set of static RLIs in order to achieve a good fault-tolerant system.

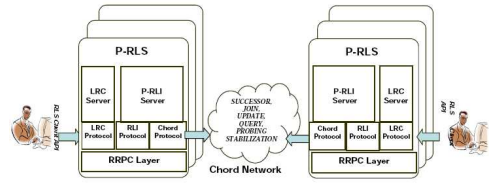

Figure 1.2: Peer-to-Peer Replica Location Service based on Chord

In [41], a different distributed RLS is shown. In this case, the authors do not use Chord but their own mechanism for distribution of indexes. This system is organized as a flat overlay network which also uses Bloom filter based compression protocols for messaging. The adaptive nature of the system comes from the use of soft-states mechanisms. As P-RLIs do, the used overlay network include the indexer services of the whole system and they distribute digests of their soft states among themselves.

A totally different approach for replica location is OceanStore's one [28]. This system assigns a globally unique identifier (GUID) to every object that it stores. A GUID is a secure hash based on owner's key and a name assigned by the owner of the object. GUIDs identifies same content, so any replica of an object has assigned the same GUID, doing useless LFN-to-PFN maps used by previously explained RLSs. Location of hosts storing replicas is done with a probabilistic algorithm and a deterministic one too. First of all, when a query for an object is received, it's routed to the closest node which probably could have a replica of the requested object. This is done using the local *Bloom filter* of the node. If any replica is not found in the target node, then a slower deterministic algorithm is used to locate the replica. Besides GUIDs for replicas, each node in the system has assigned a random unique node-ID identifying it inside the system.

A more complex scheme is shown in [44]. Besides LFNs and PFNs, site URLs, transfer URLs and source URLs are introduced. Essentially, a LFN is equivalent to a source URL. Transfer URLs are URLs containing enough information for getting the real data of the replica and can be sent to any storage resource manager (SRM). Site URLs are URLs sent to a SRM which can be managing a set of multiple physical resources. When a client sent a site URL to a SRM, it will answer with a transfer URL, allowing the client to access data.

As you can see, there are several approaches for dealing with the replica location problem. They have different APIs and semantics, making difficult to users of different schemes to use them. Large scientific project have often used different approaches for the different parts of themselves. This context is what has motivated the creation of an abstraction over the different implementations. The Replica Registration Service (RSS) [3] is an implementation covering the whole set of replica location services existing nowadays. This new service pro-

vides an uniform API to access different RLSs and replica catalogs. RSS introduces a different approach to deal with replica location. It uses two different maps for LFNs and PFNs with GUIDs: a LFN-to-GUID map and a GUID-to-PFN one, as Figure 1.3 shows. The reason for breaking the LFN-to-PFN map in two pieces is to handle aliases more efficiently. This way, a set of replicas is globally unique identified with the GUID, which can be system-assigned, and also it can have more human-readable aliases to access data. PFNs are really used as site URLs, in order to profit and cover the possibilities of storage resource managers (SRM). With this approach, attributes are assigned to GUIDs and LFNs can be changed without greater modifications. Of course, replicated data can be queried by means of LFNs and GUIDs.

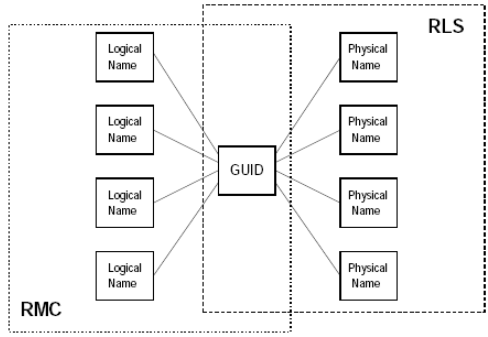

Figure 1.3: LFN, GUID and PFNs

1.3 Replica creation

As demand for information increases, centralized servers become a bottleneck. Content providers cope by distributing replicas of their files to servers scattered throughout the network. Replicas then respond to local client requests, reducing the load on the central server. Replica Management refers to the problem of deciding what files should be replicated, how many replicas of each file to distribute, and where to place them.

In a perfect system, replicas are placed near the clients that access them in order to exploit data locality. Shrinking network distance decreases access latency and sensitivity to congestion and outages.

Furthermore, exactly enough replicas should exist to handle the cumulative demand for each file. With too few replicas, servers become overloaded, and clients see reduced performance. Conversely, extra replicas waste bandwidth and storage that could be reassigned to other files, as well as the money spent to rent, power, and cool the host machine.

When one starts thinking about creating replicas, the first idea that comes into mind is to create ***replicas on demand***. In Grid terms, it means that once the job scheduler resolves a job submission request, the files required by this job are copied to the node where it has been submitted. This is the basic way of creating file replicas, and every Data Grid should implement this essential mechanism.

However, the lack of such approaches is that jobs cannot begin the execution until data has been transferred to the target node, i.e. all the input data required by the scheduled job is already there.

Therefore, new strategies appeared in the literature to optimize replica creation in Data Grids. The idea behind these techniques is creating file replicas in advance, i.e. transfer input data before it is requested. Then, replicating in advance enables jobs to start their execution faster. But, if the chosen mechanism is not good enough, it will be causing useless bandwidth consumption and storage capacity. So it is not just about replicating in advance but furthermore taking care about what, where, and how is better to replicate.

Next, in this section, we are going to talk about approaches based on two of the most significant strategies to create ***replicas in advance***. This strategies make decisions by means of analyzing prior events (file access history) and economic models in order to carry out a distributed and scalable dynamic replication mechanism.

1.3.1 Dynamic replication: Prediction based on prior events

Replication strategies based on predictive algorithms use data mining techniques to look for file access patterns from a file access history. The idea of looking for file access patterns comes from far away in

the past. For instance in [26] it is presented an approach to predict file-system actions from prior events. The main goal of the authors is to optimize an LRU cache by means of prefetching data that is likely to be accessed in the near future. The authors introduce the idea of using *tries* to analyze the filesystem access patterns.

A ***trie***, also known as prefix tree, is an ordered tree data structure that is used to store an associative array where the keys are usually strings. Unlike a binary search tree, no node in the tree stores the key associated with that node; instead, its position in the tree shows what key it is associated with. All the descendants of any one node have a common prefix of the string associated with that node, and the root is associated with the empty string. Values are normally not associated with every node, only with leaves and some inner nodes that happen to correspond to keys of interest.

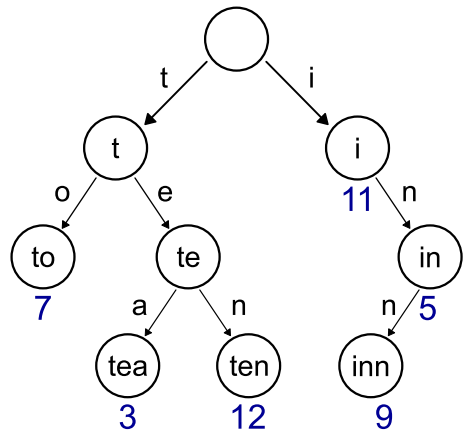

Figure 1.4: Trie sample

In the example shown (Figure 1.4), keys are listed in the nodes and values below them. Each complete English word has an integer value associated with it. A *trie* can be seen as a deterministic finite automaton, although the symbol on each edge is often implicit in the order of the branches.

Some years after, this kind of ideas were introduced to improve file-system caches, the authors of [27] presented the Thomas M. Kroger's algorithm to optimize the way to use *tries*, and furthermore they detailed how to prune them to provide scalability in terms of memory usage.

Firstly, a method called *Partitioned Context Modeling* is introduced, with this algorithm it is possible to manage a *trie* using independent partitions, or *subtries*.

Secondly, it is introduced the ***Extended Partition Context Modeling*** that establishes a threshold to determine the probability above which it makes sense to replicate. The future file access events are predicted as long as the access probability does not become lower than the threshold.

In 2006 the authors of [42] presented another mechanism to create replicas using prediction of future events based on the knowledge about which users/clients accessed which files in the past. In order to carry out this prediction, it is handled a **trie per user**. Moreover, the paper also present some experiments to determine the threshold values used to restrict:

- The tries maximum depth
- The minimum probability from which it is decided whether to replicate a file or not
- And the sequence length to be taken into account to predict the future file access.

Besides, it is also remarked how to prune the tries dynamically in order to avoid scalability problems, although the scalability problem regarding number of users (i.e. number of *tries*) is pointed to be addressed as future work.

In [52] it is presented the implementation of a file access predictor based on the knowledge about the file access history too. But, in this case, both applications and users are taken into account. Furthermore, the mechanism proposed do not use a *trie*, but a knowledge database where records about the accessed files also have the information about:

- the program

- the user
- and the sequence of the accessed files (also called successors).

In short, the algorithm takes account about what files will be accessed from a certain program executed by a certain user after a certain sequence of files have been accessed. From the evaluation made by the authors, it seems that saving three possible successors is enough. Moreover, assuming authors premises to be right, the database does not present scalability problems because *files are just accessed from five different programs at the most* (which sounds reasonable).

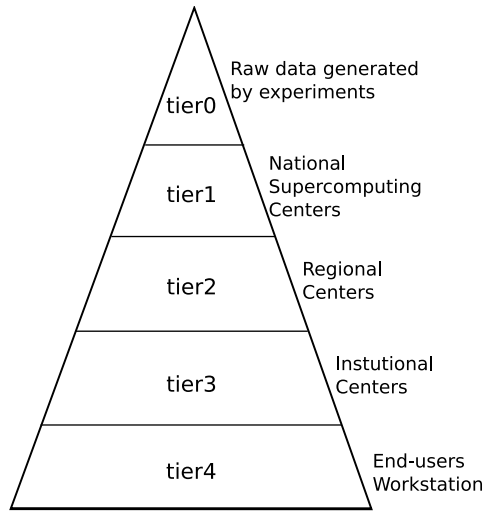

Figure 1.5: Multi-tier grid computing

On the other hand, another kind of algorithms based on file access patterns were proposed to be applied in a ***multi-tier Data Grid topology*** (Figure 1.5). The multi-tier Data Grid concept, which was first proposed by the MONARC project, aims to model the global distributed computing for the experiments on the Large Hadrond Collider (LHC) particle accelerator. These experiments are collaborations of over a thousand physics from many universities and institutes

that produce Petabytes of data per year. The raw data generated by the experiments is stored in tier-0 (e.g. at CERN), meanwhile the data analysis is carried out by several national centers (tier-1), many regional centers (tier-2), institutional centers in tier-3, and end-user workstations in tier-4. Therefore, the key idea is that data flows from the upper tiers to the lower ones, by means of a hierarchical topology that provides an efficient and cost-effective method for sharing data, computational and network resources.

The *multi-tier Data Grid* has multiple advantages:

1. It allows thousands of scientists everywhere to access the resources in a common and efficient way

2. The datasets can be distributed to appropriate resources and accessed by multiple sites.

3. The network bandwidth is used efficiently because most of the data transfers only use local or national network resources, hence alleviating the workload of international network links.

4. With the support of the Grid middlewares, the resources located in different centers and even end-users can be utilized to support data-intensive computing.

5. The multi-tier structure enables the flexible and scalable management for data-sets and users.

In 2004 the authors of [48] introduced two dynamic replication algorithms, called Simple Bottom-Up (SBU) and Aggregate Bottom-Up (ABU), that were put forward for a *multi-tier Data Grid*. These algorithms determine when to perform replication, which file should be replicated, and where to place the new replica.

In the paper, it is remarked the goal to increase the data read performance from the perspective of the clients. Then, and assuming that data access pattern changes from time to time, the authors proposed their algorithms keep track of the system situation to guide the proper replication by means of looking for the potential popular files. Therefore, the key idea is once more, analyzing current and past client access patterns to determine what files will be requested in the near future.

The basic idea of SBU algorithm is to create the replicas as close as possible to the clients that requested the data files with high rates exceeding a pre-defined threshold (that is used to distinguish the popular files). But the SBU algorithm processes the records in the access history individually and does not study the relations among these records.

On the other hand, ABU is introduced to improve SBU, because owing to the characteristics of the *multi-tier Data Grid*, i.e. every node only accesses the replicas that are in its ancestor nodes, the locations of the replica servers and the clients should be carefully considered when carrying out the replication to fully exploit the abilities of the replication resources. So the key idea of ABU is to aggregate the history records to the upper tier step by step till it reaches the root, and at the end, use this aggregation to predict which files should be predicted.

1.3.2 Replication as a game: Using economic models

In [22], authors introduced the idea of basing large-scale replica management solutions on an ***economic model***. In these economic systems, individual machines are autonomous-free to choose which replicas they host. They could make such decisions using simple on-demand algorithms or more complex predictive methods. In fact, each could use a different algorithm.

An economic approach defines the importance of a request as the amount the requester is willing to pay. A client provides useful feedback about its priorities by offering to pay servers more for certain replicas.

The economic model also helps a replica system cope with fluctuating demand. As *hot spots* appear, such as when important news breaks or a popular web site links to a normally-low-traffic page, the high demand increases the cost that servers can charge for access to replicas of the hot content. This increase encourages other servers to host a replica, distributing the load and sharing the profit.

Similarly, an economy can adapt to the addition or deletion of machines without intervention from human administrators.

Also, an economy provides an easy way to decide when to add

new servers to a system. System administrators, like capitalist entrepreneurs, can monitor price fluctuations for areas with consistently high prices, which suggests that client demand exceeds replica supply.

Scalability Replica Management Economies (RMEs) also share the scalability benefits of cooperative *P2P*[P2P survey] alternatives:

- Their use of local, greedy control algorithms avoids the computation and bandwidth bottlenecks that may appear if storage allocation, network monitoring, and failure detection are performed by a central authority.

- Guarantees through mechanism design. One sub-field of Game Theory, called Mechanism Design, studies techniques for setting system rules (algorithms, prices, etc.) in order to induce outcomes with certain desired properties. These properties may include cooperation, a balanced budget, and various definitions of "fairness".

  As a simple example, we could define an economy in which clients and servers interact using a Second-Price Auction. Each client submits a bid for replica access; the server then awards access to the highest bidder but charges the amount bid by the runner-up. This method guarantees that "rational" clients will bid honestly.

  Many generalizations of this simple second-price auction have been proposed which may prove useful in replica management economies.

- Benefits in a federated environment. A network of machines is said to be federated if the machines operate in separate administrative domains. They may cooperate to attain a common good, but each is autonomous and primarily concerned with its own success and profitability.

  RMEs fit naturally in this type of environment, which motivates most of Microeconomics and Game Theory. RMEs explicitly deal with real trust and administrative boundaries, as well as real money. They assume that machines may often reject requests, will not always volunteer truthful information, and demand payment proportionate to the work they expend. These concepts usually must be grafted onto other systems before they can be deployed in a federated environment.

- Benefits in a trusted infrastructure. On the opposite end of the spectrum, one could imagine an environment containing a single administrative domain. All machines cooperate fully, accepting external storage and retrieval requests for the common good.Despite their apparent differences, both content distribution networks and pure, cooperative P2P systems assume this environment. The former tend to employ a more global allocation algorithm and possibly restrict the set of machines that initiate the storage requests, but both approaches rely on the same inter-machine cooperation. In contrast, machines in a replica management economy accept external requests only when paid enough to make the action worthwhile. There is no need in this environment for machines to maintain individual profitability; however, this restriction on cooperation can improve system robustness. Unbounded cooperation, although conceptually simple and morally pleasing, allows a single machine to reduce the availability of many others. Poorly configured or broken machines may accidentally flood the system with unnecessary storage requests. Compromised machines may launch Denial of Service attacks. Or, perhaps more likely, greedy users will consume more resources than they should.

  In an RME, faulty or malicious machines must pay for service, and their funds are finite. Overloaded machines can raise their prices until demand drops or the failed machines run out of money. Thus, unlike more trusting models, an RME bounds the impact of failure or active attack. One could impose a similar bound on any replica management system; however, fixed bounds can be overly restrictive. They limit the flexibility of machines that are functioning perfectly yet require a great deal of resources. In an RME, the limit is soft; a machine can always acquire access to a replica if is willing and able to pay enough. In Game Theory, this property is called consumer sovereignty.

- Benefits in the Internet. The Internet is arguably the most important environment to consider when designing a large-scale replica system. Like many networks, it is neither fully cooperative nor fully federated; it contains many competitive domains, each containing machines that cooperate more or less

> completely. One could treat domains as opaque units and only impose a replica management economy among them. This approach would allow competitors to share resources safely. One could also expose the machines in each domain and extend the economy to handle intra-domain interactions as well. As shown above, the economic model provides interesting benefits even within trusted domains. Machines could still be programmed to favor others from their own domain. The RME does not prevent such coalitional activity; however, increasing the dependencies between machines decreases the robustness benefits of an RME. As in the real world, tying a greater portion of one's income or output to a favored trading partner or single resource is often risky. The lessons from Economics must be considered when programming members of an RME.

In [12] authors proposed an approach based on economic model for optimising file replication in a Data Grid. In short, in the model there are two main classes of actors. Computing Elements (CEs) have the goal of making data file available for access to Grid jobs on the site where they are executing. CEs try to purchase the cheapest replicas of needed files by interacting, via an auction protocol, with Storage Brokers (SBs) located in the Grid sites. SBs have the goal of maximising revenues they can obtain by selling files to CEs or other SBs. In the economic model, the authors make the assumption that the usefulness of a file is proportional to the revenue a SB can obtain from it. SBs have to decide whether replicating a file to the local site is a worthwhile investment. Since Grid sites have finite storage space, this could also result in deleting other files. In order to make a replication decision, SBs may use various strategies. Specifically, authors propose the use of a prediction function that estimates the future revenue of a stored file based on the past revenue of this file and of files with similar contents.

In [11] authors, also related with the previous ones, defined two types of such prediction functions and presented some experimental evaluation that they have performed on them. Both functions use for their prediction logs of file requests, that jobs have submitted to the site, but assume different statistical models for the historic data (either using a binomial distribution or a normal one).

Finally, in [6] the same authors evaluate their approach with a well-known Data Grid simulator. Both, in the case of *on-demand replication* and *in-advance replication*, the wished file is requested by means of using the auction protocol to bid for the cheapest replica. The file costs are proportional to the time needed to retrieve them, which depends on available network bandwidth between Grid sites. Furthermore, authors also mentioned a new improvement: replication can be triggered to third party sites (sites where the file was not initially needed) by means of nested auctions. This way the data migrate towards the areas the data is most likely to be requested and also reducing the bottleneck caused by only considering the requester site for replication (the nearest SB to the CE).

1.4 Replica removal

The replication of data allows to reduce the access time to data and their availability. However, replication contains an intrinsic issue: the greater the level of replication, the bigger the occupied storage space is, and in consequence, less available space for other new replicas is left. There is also a second hidden issue, consequence of the first one: once a host is out of space, is not able to run any job, if it requires access to data non-located in the host and these data cannot be stored on it.

The mentioned problem has a lot of relevance, because replica placement is directly related to the execution time required for applications, as you could see before.

This problem owns a huge difficulty to be solved, since information about why the replica was created and limitations avoiding deletion of specifics replicas must be had in mind.

The traditional approach have been to assign the responsibility of replica removal to system administrators. These people are able to determine what replicas can be deleted and how many of them are going to be deleted in order to get the required space. As you can see, this approach is not scalable, neither admissible in current grid systems, where replication of data can involve thousands of files distributed among different organizations over the world.

The complexity of replica deletion has motivated the development

of different autonomous mechanisms. These mechanisms are able to select useless replicas and to free the space they take, taking different information into account, like time to access, number of accesses, value of the replica or recoverable storage space.

One of the most commonly used techniques is the setting up of a threshold. Thresholding can be used for automatic deletion and replication as well. [45] shows a system which generates replicas automatically and also allows the user to create replicas manually. When the system detects that replication threshold was exceeded, the deletion is performed. In this case, the system always keeps the manually-generated replicas and only the automatically-created replicas are removed, until the the number of replicas is below the established threshold. The decision about what replica will be erased is based on the number of accesses. When the number of accesses in a latest period of time is below other threshold, then the replica is erasable. As you can see, this system uses the classical LRU approach.

In [38], other system managing the deletion of replicas with thresholding is shown, as well. In this case, as it was mentioned before, the threshold technique is used for both deletion and replication. Regarding the deletion, every replica has an access counter and an affinity value. When the number of accesses falls below the threshold, the replica is deleted. In order to avoid the deletion of every replica, the system uses a protocol assuring that the last replica won't be erased. Without the latter mechanism, automatic elimination of replicas could cause data loss if data is not recently used.

Other approach, different to thresholding, for replica removal is the based on economic models. In this case, replicas have an economic value, real or fictitious, which have to be optimized. Systems using this technique contain a set of algorithms taken from the stock exchange. In [12], an economic-based system is presented. The system takes account of costs of replication and its maintenance and reachable profits of replica suppression, using an historical log of accesses. Once the system has decided if deletion is beneficial for it, less significant replicas are deleted.

Also related with replica deletion, although more oriented to resolved consistency of replicas, there is the system shown in [4]. In this case, there is not any implicit mechanism for erasing data, but there is an implicit deletion of replicas when a replica is modified, which

will be the closest to the client. The system performs a deletion of every outdated replica when a client writes. Implicitly, this behaviour is favouring the liveliness of the replicas with more accesses, since once a replica is required it will be created again.

Besides LRU and economic approaches, replica deletion taking account of the age of them is often used. This approach only cares of how old a replica is and prioritizes the deletion of newest ones. In [5], a comparison among the three former strategies is shown, although it is evaluated in a simulator of grid environments. The economic one is based in bids for the replica selection, where the price is the transfer cost. Results show that the three policies get a similar improvement in the overall system though the economic one could be parametrized and adapted dynamically to the observed distribution of requests.

In [16] it is proposed the use of the longest time unused (LTU) policy and another novel one based on event occurrence prediction (EOP). The latter one takes account of the relations among different events existing in the history of the whole replica system to make the decision of removal. However, results of the experiments show that LRU strategy is the best one in general cases.

Other totally different approach is the shown in [34]. Instead of assigning specific characteristics to replicas, only the overall performance is considered. In this case, the system computes the best placement for replicas using a genetic algorithm. If the replication scheme is changed, actions for getting the new one are performed, i.e. new replicas are done and old ones are transferred or removed.

1.5 Consistency and coordination

Data Grids are currently proposed solutions to large scale data management problems including efficient file transfer and replication. Large amounts of data and the world-wide distribution of data stores contribute to the complexity of the data management challenge. Many architecture proposals and prototypes deal with replication of read-only files because it is known that in most of the cases the replication is used for read-only data. However, in some scenarios could be useful to address the replica synchronisation between replicas to manage replicas of writable files.

In principle, two mainly different replication approaches are known: *synchronous* and *asynchronous* replication.

Whereas synchronous replication aims for keeping all the replicas permanently synchronized, asynchronous replication allows for a certain delay in updating replicas.

Based on the relative slow performance of write operations in a synchronously replicated environment (due to elapsed time updating replicas), the database research community is looking for efficient protocols for asynchronous replication accepting lower consistency.

Several replication use cases are possible and the amount of read and write access to data influences the replication policy. It is very likely that various boundary conditions will affect the replication and allow for simplifications.

- read-only data: The simplest case is if data is read-only,where data may be copied at any point in time from any replica to any other place. This requires no locking nor any other coupling (except for the replica catalogue) of replicas. Note it is probably very hard to ever remove the *readonly* property from a file in a running system without risking to compromise readers. Therefore, applications would be required to insure that data will never need any change.

- writable data: Once we allow write access to the data, it is important to have a clear policy that defines who is allowed to write/change data. If ownership is assigned to files (replicas), one policy can be that only the owner is allowed to modify the original version of a file (master copy). For a data item which can be updated (writable) we distinguish between permanent and varying ownership.

  - well defined file ownership ("master-slave case"): Only one well defined entity in the system is allowed to modify a particular piece of data (e.g. a file). As a result, the replication is not symmetric any more between all replicas in the system. The process of determining which is the most up-to-date version in the system is not required. Only the information "who is the owner" needs to be propagated to

all slave replicas. In case of data access, only one well defined node needs to be contacted to obtain the most recent version of the data. This is only true for write operations. For a read access, any replica can be selected since the master-slave approach guarantees that all copies are up-to-date. In detail, all write and update requests are forwarded to the master which in turn is responsible for synchronising all the slaves. Read requests can be served by any replica.

- varying writers (no central control of replicas): This is the most general and complex case. Several update operations need global agreement between all replicas and will also try to contact all replicas to obtain a quorum. Quorum systems are commonly used as a mechanism to get the right, for example, to update a replica. The current distributed database research proposes several solutions to this problem.

In [17] is presented a new Grid service, called Grid Consistency Service (GCS), that sits on top of existing Data Grid services and allows for replica update synchronisation and consistency maintenance. The paper presents some models for different levels of consistency provided to the Grid user using as data sources both *databases* and *filesystems*:

- Consistency Level -1 (Possibly inconsistent copy): The file replica is created using a trivial file copy concurrently with ongoing write operations. The resulting file does not necessarily correspond to a state of the original file at any point in time and internal data structures may be inconsistent. There are several well known ways to tackle this problem:
  - standard locking: obtain a file write lock - perform the file copy - release the lock.
  - optimistic locking: In case of low probability of lock contention, one could copy without getting a lock and test the modification date of the file after the copy. In case of conflict, one gets a lock and retries.
  - snapshots: One could use the database or file-system services to produce a consistent file snapshot (i.e. keep an

old version of the file until the copy process is finished, but allow writers already to modify).

- Consistency Level 0 (Consistent File Copy): At this consistency level, the data within a given file corresponds to a snapshot of the original file at some point in time. Again, we have the dirty read problem. In this case, it is still unclear if a file copy in this intermediate state would be usable by a remote Grid user.

  There are again several mechanisms to obtain such a replica:

  - locks: one obtains a read lock to exclude other writers.
  - snapshot: a consistent snapshot is maintained for the duration of the copy. This would allow a concurrent writer to continue its work.

- Consistency Level 1 (Consistent Transactional Copy): Each replica has been produced at a time when no write transactions were active and can be used by other clients without internal consistency problems.

- Consistency Level 2 (Consistent Set of Transactional Copies): If the replicas have been produced as part of a single transaction, the main consistency problem left is that replicated data might not be up to date, once the remote node starts working on it. Replica and original could diverge. This in particular poses problems if it is required to merge the data changes from different sites to the same data.

- Consistency Level 3 (Consistent Set of up-to-date Transactional Copies): This is basically what is called a "replicated federation" in Objectivity/DB where a replica stays under the control of the database system and depending on the database implementation, read/write locks may have to be negotiated.

  In a Grid system, such a complex replication environment can only be attained if all data access operations use a common interface and do not allow non-Grid access like local *fseek* on files. This vision would mean that the Grid is a distributed database management system on its own but it may not be feasible for most of the Data Grid applications.

> Read or write access to replicas is always atomic with a conventional database transaction. This is a very strict model and known as synchronous replication which might be useful for some meta data but also may impose severe performance and usability constraints.

Besides previously explained consistency mechanisms, there are schemes with specific levels of consistency, like the used on by Google file system [37]. In this case, all the consistency control is handled by a master, which also maintains all the metadata related to. Replicas are read-only but consistency about locations must be guaranteed.

Other different mechanism for replica consistency is the deletion of every existing replica when a client writes on one. This mechanism is pretty naive but it can be found in [4]. Since there won't be any replica to update after a write operation, the system will always be in a consistent state. After that operation, replicas will be generated on-demand to client requests. The written replica will be the closer one to writer client.

One commonly found approach is the based on version numbers one. It is used in [28] for instance. The mechanism assigns a version number to every replica and each update operation will increase this number. With this technique, greater version number is, more updated replicas are. Quorum can be used for coordination of versions numbers but it is not mandatory if replicas can be outdated.

This last scheme is related to optimistic replication [43], where every operation is allowed and consistency issues are resolved only when conflicts are detected. This approach have been proved to be very efficient in systems where concurrent write operations are unusual, as most typical data grids. In this case, conflicts can often be solved automatically. The improvement is due to the characteristics of locking protocols. As [9] shows, this kind of protocols adds a considerable overhead that can be avoided with lazy propagation of updates.

1.6 Other issues

1.6.1 Data replication complexity in Data Grids

In [15], authors show that data replication on data grids is a NP-hard and non-approximable optimization problem. Authors focus the analysis on approach to data replication whereby files are replicated in advance in order to make all sites as suitable for job executions as possible, but there is no replication on-demand (so this simplify the model, since it is not needed to simulate the job scheduling).

Firstly, the authors built a mathematical model of a grid and then formally define GDR, the optimization problem of data replication on grids with the explored approach. GDR can also be used as a formal framework for analysing the computational complexity of static data replication, and a starting point for the design of new algorithms for solving it.

Secondly, they studied some related problems such as:

- the file allocation problem: concerned with replicating a single file on a given network in order to improve read requests to the file. Since there are also write requests to the file, its replicas have to be updated in order to maintain data consistency. The problem is to find an optimal replica allocation, which minimizes the time for maintaining read and write requests. The difference between this problem and the GDR is:
  - in the abscence of write requests
  - the possibility of multiple objects (e.g. files)
  - in capacity constraints of sites
- the web server replica placement: where k server replicas have to be placed on a given network so that network communication load is minimized. This problem also manages only one object (a server in this case); however, no updating is needed now. Notice that a fixed number of replicas (i.e. mirrors) have to be allocated, while in GDR the number of replicas is not fixed. The web server replica placement problem can be stated as a facility location problem, in particular the k-median problem.

- the page migration/replication problem:This problem is also concerned with a single object (i.e. the page) and is not limited by capacity constraints (e.g. memory sizes) as is the GDR.

Then the authors described the goal of the GDR as distributing the replicas of objects (i.e. files) across the Grid, in such a way that every site will offer fast access (i.e. low transfer cost) to at least one replica of each object. Since by assumption all accesses are read requests, fast access is needed to at least one replica of each object. In this way, many different applications can be hosted on each site. Hence, the goal is looking for a function that assigns to each object a subset of sites. Feasible functions must take account of the storage capacities of sites, and the transfer costs in terms of bandwidth between source and target, and the size of the file. Finally, authors make a reduction to an integer programming and also carry out some simplifications, but anyhow, the problem is demonstrated to be:

- NP-hard: since it is unlikely that an exact polynomial time algorithm will be found to solve the problem.
- Non-approximable: which means that for large instances the only reasonable approach is the development of good heuristic methods.

1.6.2 Optimal Placement of Replicas

It has been shown that file replication can improve the overall Data Grid performance, but although there is a fair amount of work on file replication in Grid environments, most of this work is focused on creating the underlying infrastructure for replication and mechanisms for creating/deleting replicas. Therefore, in order to obtain more gains from replication, works on strategic placement of the file replicas appeared.

In 2004 the author of [1] studied the problem of replica placement in a Data Grid. They proposed a replica placement service called Proportional Share Replication (PSR) and he evaluated it simulating a *multi-tier Data Grid* environment. The key idea of the PSR is that each file replica should service approximately equal number of request rates in the system. The goal is to place the replicas on a set of sites

systematically selected such that files access parallelism is increased while the access costs are decreased. In the paper, an evaluation is presented by means of comparing the PSR algorithm with a classical replica location policy based on affinity, i.e. data would be replicated on or near the client machines where the file is mostly accessed.

In 2006 the authors of [32] pointed that the PSR algorithm did not guarantee to find the optimal solution. Therefore, they proposed a new algorithm to address the replica placement problem given the traffic pattern and locality requirements. This algorithm finds the optimal locations for the replicas so that the workload among these replicas is balanced. They also present another algorithm to decide the minimum number of replicas required when the maximum workload capacity of each replica server is known.

To implement both algorithms, authors took the following issues into account:

- The replicas should be placed in proper server locations so that the workload on each server is balanced. A naive placement strategy may cause "hot spot" servers that are overloaded, while other servers are under-utilized.

- The optimal number of replicas should be chosen. The denser the distribution of replicas is, the shorter the distance a client site needs to travel to access a data copy. However, maintaining multiple copies of data in Grid systems is expensive, and therefore, the number of replicas should be bounded.

  Clearly, optimizing access cost of data requests and reducing the cost of replication are two conflicting goals. Finding a good balance between them is a challenging task.

- Consideration of service locality. Each user may specify the minimum distance he can allow from him to the nearest data server. This serves as a locality assurance that users may specify, and the system must make sure that within the specified range there does exist a server to answer the request.

As you can see, these algorithms are based on the assumption of a multi-tier Data Grid environment. Although it is one of the most

common topologies used by the Grid community, studies based on other topologies are needed in order to cover another use cases.

In [39] the authors use a non-tree topology as a Grid. They examine different replica placement strategies based on the expected utility and risk. Algorithms proposed based on utility select a replica site assuming that future requests and current load will follow current loads and user requests. On the other hand, algorithms using a risk index expose sites far from all other sites and assume a worst case whereby future requests will primarily originate from there.

The authors evaluate their four algorithms by comparison with two previous replication strategies (BestClient and Cascading). Specifically, the resultant six algorithms are:

1. *MinimizeExpectedUtil*: considers each node and calculates the expected utility. The node with the lowest expected utility is selected. The replica is then placed at that node.

2. *MaximizeTimeDiffUtil*: considers each site S and determines the time based distance between the best replica site (with respect to S) and the other sites. The site with the maximum distance is the closest site for S. Thus, placing a replica at the closest site, the differential time is saved. The *MaximizeTimeDiffUtil* is calculated by multiplying the maximum time difference by the number of requests site S makes for given time period. The site with the maximum time difference utility is selected and the replica is placed on the site generating the maximum time difference.

3. *MinimizeMaxRisk*: for each site, the distance from it to other sites holding replicas is calculated so the minimum distance among them is identified. The risk index for each site is calculated by multiplying the file requests by the minimum distance. It has been shown that file replication can improve the overall Data Grid performance, but although there is a fair amount of work on file replication in Grid environments, most of this work is focused on creating the underlying infrastructure for replication and mechanisms for creating/deleting replicas. Therefore, in order to obtain more gains from replication, works on strategic placement of the file replicas appeared.

4. *MinimizeMaxAvgRisk*: calculates the average risk for each site and multiplies by the file requests. The replica is placed where it is obtained the highest average index. This algorithm is in fact a variation of the previous one.

5. *BestClient*: places the replica at the site that has maximum requests for file x (affinity).

6. Cascading: places the replica on the path of the best client.

Eventually, although the topology chosen is very simple (just eight nodes), they shown promising results for their algorithms, except for the *MaximizeTimeDiffUtil* that resulted to be a variation of Cascading that also performs well in the case that users requests contain some geographical locality. Therefore, considering them would be a good idea for current and future replica placement mechanisms.

1.6.3 Scheduling

Another issue that could be interesting regarding replica management strategies in Data Grids, is how this mechanisms affect the behaviour of the job schedulers. Usually, job schedulers tend to use data locality as a factor to take into account in their scheduling decisions, so that jobs are submitted to nodes that already have mostly of the input data these jobs need. Using this kind of policy, helps the job scheduler to minimize the bandwidth consumption by means of minimizing data transfers among the network, and also to reduce the waiting time elapsed to the time the job is able to start the execution (when the input data is available in the node).

In distributed and parallel systems, the widely used performance metrics for job scheduling include *turnaround time, throughput, utilization, makespan and slowdown.*

Turnaround time measures how long a job takes from its submission to completion. As the system utilization and throughput are largely determined by the job arrival process and the job resources requirements rather than by the scheduler, they are only suitable for closed systems in which every job is re-submitted to the system when it terminates. Makespan is used for batch mode scheduling. Slowdown is defined as the job turnaround time divided by its execution time.

In [47] the authors proposed a Data Grid architecture supporting efficient data replication and job scheduling. The computing sites are organized into individual domains according to the network connection, and a replica server is placed in each domain.

There are two centralized replication algorithms with different replica placement methods and a distributed replication algorithm are put forward.

The centralized algorithms are characterized to use a replication master running in the system which aggregates and summarizes all the collated historical information coming from specific nodes (called replica servers) about: number of accesses (NOA) per file (FID). Then, the centralized replication algorithm is invoked by the replication master which commands the replica servers to carry out the replication. A threshold for NOA is used in the algorithm to distinguish popular data files, and only the files that have been accessed more than this threshold times will be replicated. The more the NOA value is above the threshold, the more number replicas are done for the related FID. Finally, two replica placement policies are presented for the *centralized algorithms*:

- *Response-time oriented replica placement* method called RTPlace, which takes account of the CPU and the storage capacities of the nodes.
- *Server merit oriented replica placement* method, called SMPlace, which takes account of the locality of the target nodes relative to all domains.

On the other hand, in the distributed algorithm, the historical records are exchanged among all replica servers. Every replica server aggregates NOA over all domains for the same data file and creates the overall data access history of the system. At intervals, each replica server will use the replication algorithm to analyze the history and determine data replications.

Regarding scheduling, authors introduce three heuristics.

- *Shortest turnaround time.* For each incoming job, the shortest turnaround time (STT) heuristic estimates the turnaround time on every computing site and assigns the job to the site that provides the shortest turnaround time.

- *Least relative load.* Assigns the new job to the computing site that has the least relative load (i.e. the relationship between quantity of jobs and computing capability).
- *Data Present.* Takes the data location as the major factor when assigning the job. According to different situations of the data file required by the job.

Finally authors evaluate the different replication algorithms with the different job scheduling algorithms using a simulator built by them called *XDRepSim*. They conclude that centralized replication can shorten the job turnaround time greatly. In particular, the policy of STT + CDR (with RTPlace) exhibits remarkable performance under various conditions of system environment and workload.

1.7 Discussion / Conclusions

In this chapter we have presented the state of the field about data replication mechanisms in distributed environments. As you can see, we have focused on Grids since they are the current trend in the literature, but most of the described algorithms, strategies and techniques are also appliable in other distributed environments, such as clusters or P2P systems.

The discussion has been centered in the key issues that must be considered in order to achieve scalable and suitable replication management systems.

Firstly, we have presented the most important techniques and strategies to deal with location and selection of replicas that are spread among several nodes.

Secondly, we have discussed about how to create these replicas this section in two main branches.

On one hand, we have introduced predictive algorithms based on prior events. These algorithms apply data mining on file access histories in order to predict the near future files that are going to be accessed. Therefore, they proceed creating replicas taking these predictions into account.

On the other hand, we have also described the current trend on economy-based strategies that provide replication mechanims based

on economic models. The key advantage of these models is that they tend to be optimal in long-term.

Thirdly, we have addressed the issue about replica deletion, its relevance and some techniques to deal with it. The key problem is that the more quantity of replicas the system has, the more saturated storage resources are, and therefore, the system becomes unefficient since future replicas, likely to be more useful than current ones, cannot be created because of lack of space.

Fourthly, we have pointed the problem of consistency among replicas. When one decides to spread several replicas of the same file among the nodes of a distributed environment, they are likely to be accessed concurrently by different users (i.e. different applications, jobs, processes, etc.). Therefore, if these files are wished to be writable, well-known coherence troubles appear and different srategies and techniques are available in order to reach different levels of consistency according to system requirements.

Finally, we have presented a section that ponts other issues related to replication management. Specifically, we have addressed some papers that talk about the complexity of the replication problem in distributed environments, whichs tends to be NP-hard; other that describe algorithms to deal with the problem about deciding the optimal placement of replicas; and finally, the influence of file replication on job schedulers of distributed systems.

1.8 Future Trends

Replica management is a mature research topic as we showed along this chapter. However some issues are not resolved yet. Future researches in replica management systems aim at the improvement of current systems for getting a better performance and an easier maintenance.

Current trends in replica selection are based on intelligent agents and self-managing systems, taking in account the dynamism of grids and their variable characteristics. Future replica selection system will measure available bandwidth, latencies of networks and the impact of scheduling decisions in order to achieve a better performance.

Regarding replica creation and replica removal, future trends will be inspired in economics and bio-inspired algorithms, able to respond

in a fast way to the requirements of storage space, availability, fault-tolerance and QoS. Both areas are related to replica placement and selection, so improvements on these ones will influence those ones.

The main open issue in current replica management services is the consistency. Current system usually limits the replication to creation and read-only accesses. This is limiting the use of grid technologies in this always-changing field, so new systems dealing with consistency in systems with several concurrent writers and readers over the same data will be developed in the close future. Consistency issues are old-knowns since 70's and their solutions can be also applied to grid data management.

Bibliography

[1] Jemal H. Abawajy. Placement of file replicas in data grid environments. In Marian Bubak, G. Dick van Albada, Peter M. A. Sloot, and Jack Dongarra, editors, *International Conference on Computational Science, Computational Science - ICCS 2004, 4th International Conference, Kraków, Poland, June 6-9, 2004, Proceedings, Part III*, volume 3038 of *Lecture Notes in Computer Science*, pages 66–73. Springer, 2004.

[2] William E. Allcock, Ian T. Foster, Veronika Nefedova, Ann L. Chervenak, Ewa Deelman, Carl Kesselman, Jason Lee, Alex Sim, Arie Shoshani, Bob Drach, and Dean Williams. High-performance remote access to climate simulation data: a challenge problem for data grid technologies. In *SC*, page 46, 2001.

[3] Alex Sim Arie Shoshani and Kurt Stockinger. Rss: Replica registration service for data grids, 2005.

[4] Awerbuch, Bartal, and Fiat. Competitive distributed file allocation. *INFCTRL: Information and Computation (formerly Information and Control)*, 185, 2003.

[5] William H. Bell, David G. Cameron, Luigi Capozza, A. Paul Millar, Kurt Stockinger, and Floriano Zini. Simulation of dynamic grid replication strategies in optorsim. In Manish Parashar, editor, *Grid Computing - GRID 2002, Third International Workshop, Baltimore, MD, USA, November 18, 2002, Proceedings*, volume 2536 of *Lecture Notes in Computer Science*, pages 46–57. Springer, 2002.

[6] William H. Bell, David G. Cameron, Ruben Carvajal-schiaffino, A. Paul Millar, and Kurt Stockinger. Evaluation of an economy-based file replication strategy for a data grid, February 13 2003.

[7] William H. Bell et al. Optorsim: A Grid simulator for studying dynamic data replication strategies. *The International Journal of High Performance Computing Applications*, 17(4):403–416, Winter 2003.

[8] Bloom. Space/time trade-offs in hash coding with allowable errors. *CACM: Communications of the ACM*, 13, 1970.

[9] Yuri Breitbart and Henry F. Korth. Replication and consistency: Being lazy helps sometimes. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 173–184, Tucson, Arizona, 12–15 May 1997.

[10] Min Cai, Ann Chervenak, and Martin R. Frank. A peer-to-peer replica location service based on a distributed hash table. In *SC*, page 56. IEEE Computer Society, 2004.

[11] L. Capozza, K. Stockinger, and F. Zini. Preliminary Evaluation of Revenue Prediction Functions for Economically-Effective File Replication. Technical Report DataGrid-02-TED-020724, CERN, Geneva, Switzerland, July 2002.

[12] Mark Carman, Floriano Zini, Luciano Serafini, and Kurt Stockinger. Towards an economy-based optimisation of file access and replication on a data grid. In *CCGRID*, pages 340–345. IEEE Computer Society, 2002.

[13] Ann L. Chervenak, Ewa Deelman, Ian Foster, Leanne Guy, Wolfgang Hoschek, Adriana Iamnitchi, Carl Kesselman, Peter Kunst, Matei Ripeanu, Bob Schwartzkopf, Heinz Stockinger, Kurt Stockinger, and Brian Tierney. Giggle: A framework for constructing scalable replica location services. In *SC'2002 Conference CD*, Baltimore, MD, nov 2002. IEEE/ACM SIGARCH.

[14] Ann L. Chervenak, Naveen Palavalli, Shishir Bharathi, Carl Kesselman, and Robert Schwartzkopf. Performance and scalability of a replica location service. In *HPDC*, pages 182–191. IEEE Computer Society, 2004.

[15] Uros Cibej, Bostjan Slivnik, and Borut Robic. The complexity of static data replication in data grids. *Parallel Computing*, 31(8-9):900–912, 2005.

[16] Hluchy L. Ciglan M. Towards scalable grid replica optimization framework. In *Parallel and Distributed Computing, 2005. ISPDC 2005.*, pages 43–50, 2005.

[17] Dirk Düllmann and Ben Segal. Models for replica synchronisation and consistency in a data grid. In *HPDC '01: Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC-10'01)*, page 67, Washington, DC, USA, 2001. IEEE Computer Society.

[18] Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G. Bruce Berriman, John Good, Anastasia C. Laity, Joseph C. Jacob, and Daniel S. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219–237, 2005.

[19] Jun Feng and Marty Humphrey. Eliminating replica selection - using multiple replicas to accelerate data transfer on grids. In *ICPADS*, pages 359–366. IEEE Computer Society, 2004.

[20] Corina Ferdean and Mesaac Makpangou. A scalable replica selection strategy based on flexible contracts. In *WIAPP '03: Proceedings of the The Third IEEE Workshop on Internet Applications*, page 95, Washington, DC, USA, 2003. IEEE Computer Society.

[21] Ian T. Foster and Carl Kesselman. The globus project: A status report. In *Heterogeneous Computing Workshop*, pages 4–18, 1998.

[22] Dennis Geels and John Kubiatowicz. Replica management should be A game, July 26 2002.

[23] Jim Gray, Pat Helland, Patrick E. O'Neil, and Dennis Shasha. The dangers of replication and a solution. In H. V. Jagadish and Inderpal Singh Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 173–182, Montreal, Quebec, Canada, jun 1996.

[24] The Earth Systems Grid.

[25] Leanne Guy, Peter Kunszt, Erwin Laure, Heinz Stockinger, and Kurt Stockinger. Replica management in data grids, jul 2002.

[26] Thomas M. Kroeger and Darrell D. E. Long. Predicting file-system actions from prior events. In *Proceedings of the USENIX Annual Technical Conference*, pages 319–328, Berkeley, January 22–26 1996. Usenix Association.

[27] Thomas M. Kroeger and Darrell D. E. Long. Design and implementation of a predictive file prefetching algorithm. In USENIX, editor, *Proceedings of the 2001 USENIX Annual Technical Conference: June 25–30, 2001, Marriott Copley Place Hotel, Boston, Massachusetts, USA*, pub-USENIX:adr, 2001. USENIX.

[28] John Kubiatowicz, David Bindel, Yan Chen, Steven E. Czerwinski, Patrick R. Eaton, Dennis Geels, Ramakrishna Gummadi, Sean C. Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Y. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *ASPLOS*, pages 190–201, 2000.

[29] Peter Z. Kunszt, Erwin Laure, Heinz Stockinger, and Kurt Stockinger. Advanced replica management with reptor. In Roman Wyrzykowski, Jack Dongarra, Marcin Paprzycki, and Jerzy Wasniewski, editors, *PPAM*, volume 3019 of *Lecture Notes in Computer Science*, pages 848–855. Springer, 2003.

[30] Houda Lamehamedi, Zujun Shentu, Boleslaw K. Szymanski, and Ewa Deelman. Simulation of dynamic data replication strategies in data grids. In *17th International Parallel and Distributed Processing Symposium (IPDPS-2003)*, pages 100–100, Los Alamitos, CA, April 22–26 2003. IEEE Computer Society.

[31] Houda Lamehamedi, Boleslaw Szymanski, Zujun Shentu, and Ewa Deelman. Data replication strategies in grid environments, June 18 2002.

[32] Yi-Fang Lin, Pangfeng Liu, and Jan-Jan Wu. Optimal placement of replicas in data grid environments with locality assurance. In *ICPADS*, pages 465–474. IEEE Computer Society, 2006.

[33] Michael J. Litzkow, Miron Livny, and Matt W. Mutka. Condor - A hunter of idle workstations. In *ICDCS*, pages 104–111, 1988.

[34] Thanasis Loukopoulos and Ishfaq Ahmad. Static and adaptive distributed data replication using genetic algorithms. *J. Parallel Distrib. Comput*, 64(11):1270–1285, 2004.

[35] Anirban Mondal and Masaru Kitsuregawa. Effective dynamic replication in wide-area network environments: A perspective. In *DEXA Workshops*, pages 287–291. IEEE Computer Society, 2005.

[36] LIGO Laser Interferometer Gravitational Wave Observatory.

[37] Sean Quinlan. The google file system. In *The Conference on High Speed Computing*, page 24, Salishan Lodge, Gleneden Beach, Oregon, April 2006. LANL/LLNL/SNL.

[38] M. Rabinovich, I. Rabinovich, and R. Rajaraman. A dynamic object replication and migration protocol for an internet hosting service. In *19th International Conference on Distributed Computing Systems (19th ICDCS'99)*, Austin, Texas, May 1999. IEEE.

[39] Rashedur M. Rahman, Ken Barker, and Reda Alhajj. Replica placement in data grid: Considering utility and risk. In *ITCC (1)*, pages 354–359. IEEE Computer Society, 2005.

[40] Rajesh Raman, Miron Livny, and Marvin H. Solomon. Matchmaking: Distributed resource management for high throughput computing. In *HPDC*, page 140, 1998.

[41] Matei Ripeanu and Ian T. Foster. A decentralized, adaptive replica location mechanism. In *HPDC*, page 24. IEEE Computer Society, 2002.

[42] Jih-Sheng Chang Ruay-Shiung Chang, Ning-Yuan Huang. A predictive algorithm for replication optimization in data grids, 2007.

[43] Saito and Shapiro. Optimistic replication. *CSURV: Computing Surveys*, 37, 2005.

[44] Arie Shoshani, Alexander Sim, and Junmin Gu. *Storage resource managers: essential components for the Grid*, pages 321–340. Kluwer Academic Publishers, Norwell, MA, USA, 2004.

[45] Renata Slota, Darin Nikolow, Lukasz Skital, and Jacek Kitowski. Implementation of replication methods in the grid environment. In *EGC*, pages 474–484, 2005.

[46] Ion Stoica, Robert Morris, David R. Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, pages 149–160, 2001.

[47] Ming Tang, Bu-Sung Lee, Xueyan Tang, and Chai Kiat Yeo. The impact of data replication on job scheduling performance in the data grid. *Future Generation Comp. Syst*, 22(3):254–268, 2006.

[48] Ming Tang, Bu-Sung Lee, Chai Kiat Yeo, and Xueyan Tang. Dynamic replication algorithms for the multi-tier data grid. *Future Generation Comp. Syst*, 21(4):775–790, 2005.

[49] Sudharshan Vazhkudai, Steven Tuecke, and Ian T. Foster. Replica selection in the globus data grid. *CoRR*, cs.DC/0104002, 2001.

[50] Rich Wolski, Neil T. Spring, and Jim Hayes. The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5–6):757–768, 1999.

[51] Rich Wolski, Neil T. Spring, and Jim Hayes. The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5–6):757–768, October 1999.

[52] Tsozen Yeh, Darrell D. E. Long, and Scott A. Brandt. Increasing predictive accuracy by prefetching multiple program and user specific files. In *HPCS*, pages 12–19. IEEE Computer Society, 2002.

Chapter 2

Towards Computing Resources Abstraction: Using Virtualization

Íñigo Goiri and Jordi Guitart

Abstract

Computing in these days is becoming more and more powerful and this can imply a resource underusage. Sharing these remaining resources between different virtual environments is an elegant solution. It can be achieved using virtualization.

Virtualization allows resource abstraction and an isolated environment for different purposes. This chapter presents virtualization alternatives and how to get this with different techniques. These techniques will be discussed according to its functionality in real environments and innovations in this research area.

Furthermore, managing resources between virtual environments in a smart way is not an easy issue. To serve this purpose monitoring domain behavior is crucial in order to achieve this smart resource sharing.

2.1 Introduction

There are many virtualization alternatives but they have a common issue: hiding technical system details. It creates an interface that hides implementation issues through encapsulation. In addition, it had introduced access multiplexing to a single machine opening new ways in research.

This chapter will make an overview about virtualization technologies and will describe which techniques exists and how these are implemented in real solutions. It will also do a complete review about one of these implementations, Xen, one of the most significant virtualization technologies in these days.

It will also have a look at how this technologies are being used on real environments and which virtualization alternatives and why are being utilized describing its advantages respect typical alternatives.

2.1.1 History

Virtualization is an old issue, it was used since 1960s as software abstraction layer that partitions a hardware platform into virtual machines that simulate the underlying physical machine that allows running unmodified software. This mechanism provide a way to multiplex application usage to users sharing processor time.

Providers started the development of hardware that supports virtualized hardware interfaces through the Virtual Machine Monitor, or VMM. In the early days of computing, the operating system was called the supervisor. With the ability to run operating systems on other operating systems, the term hypervisor resulted in the 1970s.

At that time, it was used in industry and in academic research, nevertheless, in the 80s, modern multitasking operating systems and hardware price allowed users run their applications in a single machine. It seemed to be the end of virtualization and hardware providers no longer support virtualization in their architectures. It became a historical curiosity.

However, in the 1990s Stanford University researchers found that they disposed many architectures running different operating systems made difficult to develop and port applications. Introducing a layer that makes different hardware look similar, Virtualization was the so-

lution. Furthermore, it brings new solutions like process isolation, security, mobility and efficient resource management.

In the present days, it is a real alternative and is widely extended, for instance hardware providers are taking up again virtualization support in its hardware.

2.1.2 Why is virtualization important?

There are many reasons for using virtualization such as implying saving in power, space, cooling or administration. For example, server consolidation, that means putting a number of under-utilized systems on a single server.

Another area where virtualization can mean a higher quality level is development. It can be very useful for developers having at their disposal a safe and reliable system that can be easily managed.

Virtualization is also opening new ways in research projects thanks to its resource managing capabilities that allows well known problems being easily solved.

These technologies are becoming more and more popular in these days in real environments. Some of these will be discussed in the section 2.4.

2.2 Virtualization types

There is not just one way to achieve a virtualized environment. In fact, there are several ways to achieve this with different levels of abstraction obtaining different characteristics and advantages.

Computer systems introduces a division into levels of abstraction separated by well-defined interfaces. Levels of abstraction allow implementation details at lower levels of a design to be ignored or simplified, thereby simplifying the design of components at higher levels.

The levels of abstraction are arranged in a hierarchy, with lower levels implemented in hardware and higher levels in software. Figure 2.1 shows how a typical system is separated in different layers that introduces a different abstraction degree according to the layer level.

Virtualization introduces an abstraction layer to show higher layers a different overlayed system. Virtualization can be classified according

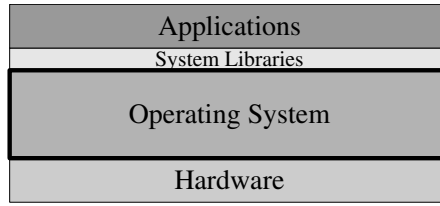

Figure 2.1: Computer systems abstraction layers

with the system layer interface that it abstracts, although, some virtualization techniques such as paravirtualization or partial virtualization combine some of these with performance purposes.

Taking into account some virtualization reviews like [1] that introduces some virtualization techniques, the majors types are: hardware emulation, full virtualization, partial virtualization, paravirtualization, operating system-level virtualization, library virtualization and application virtualization.

All these types and some particular implementations will be described in the next subsections.

2.2.1 Hardware Emulation

In this virtualization method, VM simulates a complete hardware allowing an unmodified OS to be run. Every instruction is simulated on the underlying hardware and this means a high performance lost (can achieve a 100 times slowdown). The VMM which has to translate guest platform instructions to instructions of the host platform is called emulator.

Emulator tries to execute emulated virtual machine instructions by translating them to a set of native instructions and then execute them on the available hardware. This set of instructions has to contain typical, I/O, ROM reading, rebooting, etc to allow a successful real computer emulation.

On the one hand, this method allows running an operating system without any modification. This method have an ease of implementation and this means a facility to port this to different guest platforms.

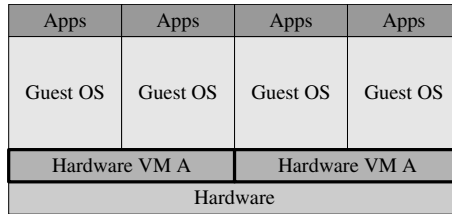

Figure 2.2: Emulation

In addition, you can even run multiple virtual machines, each simulating a different processor.

On the other hand, since every instruction needs to be interpreted in software, the performance penalty involved is significant. This lost of performance can easily mean a 100 times slowdown and it could be a 1000 times slower in a high-fidelity emulation that can include cycle accuracy, CPU pipeline, caching behavior, etc.

Many techniques are used to implement emulation. Some of the most famous examples of emulation are Bochs and QEMU.

Bochs

Bochs [2] is an emulator available in many platforms such as x86, PowerPC, Alpha, SPARC and MIPS that emulates an x86 architecture and is currently released unde LGPL license.

This emulator mostly written in C++ simulates the whole computer including peripherals, memory, display... and not just the processor. In addition, it can be configured in many modes like older Intel 386 or the newest 64-bit alternatives and can simulate new instructions like MMX processors.

Bochs runs on Linux systems, therefore, any operating system that supports x86 architecture can be run on Linux. Nowadays, this is mostly used for operating system development and it is also used to run older software.

QEMU

QEMU [3] is an open source software that can be used as a fast processor emulator that utilizes dynamic translation or as a full virtualizer by executing guest code directly on the host CPU.

Taking into account this duality, it has some differences with other emulators like Bochs. It supports two operation modes, the first one is full system emulation mode which emulates a full system with processor and peripherals. This mode emulates architectures like x86, x86_64, ARM, SPARC, PowerPC and MIPS with reasonable speed using dynamic translation.

The second mode called user mode emulation, which can only be hosted on Linux with a host driver called, KQEMU, that allows executing binaries for different architectures to be executed on Linux running on x86. Among the supported architectures in this mode, we can find ARM, SPARC, and PowerPC.

A full virtualizer called VirtualBox [4] was created taking profit of QEMU full virtualized mode. It uses a built-in dynamic recompiler based on QEMU. It runs nearly all guest code natively on the host and uses the recompiler only for special situations.

In conclusion, QEMU can be considered as an emulator with full virtualizion capabilities at the same time.

2.2.2 Full virtualization

This method, also known as native virtualization, uses a virtual machine that mediates between guest operating system and the native hardware. Is faster than emulation but slower than underlyed hardware because of the hypervisor mediation.

In this case, host operating system doesn't need to be modified. Virtual machine simulates enough hardware to allow an unmodified operating system. Certain machine instructions must be trapped and handled within the hypervisor because the underlying hardware is not owned by an operating system but is instead shared by it through the hypervisor.

One of the biggest advantages of full virtualization is that guest OS can run unmodified. Nevertheless, it must support the underlying hardware.

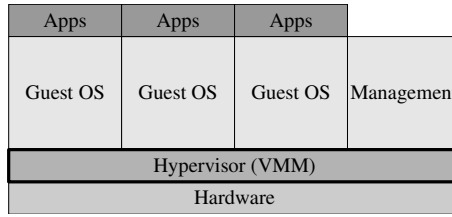

Figure 2.3: Full virtualization

There are multiple alternatives in this technique like VirtualBox. VMWare, Parallels Desktop and z/VM.

VMWare

VMWare [5] is a commercial full virtualization alternative that implements a hypervisor sat between the guest operating system and the bare hardware as a new layer. This layer abstracts any operating system from the real hardware and allows this OS running without knowledge of any other guest on the system.

VMWare also virtualizes the available I/O hardware and places critical drivers into the hypervisor increasing performance.

The virtualized environment is seen as a file that can be easily and quickly migrated to a new host.

z/VM

z/VM [6] new IBM product has a long heritage from 1960s VM developing. Its core is the Control Program (figure 2.4) which is the operating system hypervisor for the system z that provides virtualization of physical resources and allows multiple processors and resources to be virtualized to different guest operating system, like Linux or z/OS.

This is designed to allow the capability for clients to run hundreds to thousands of Linux server on a single mainframe running with other System z operating system, such as z/OS as a large-scale Linux-only enterprise server solution.

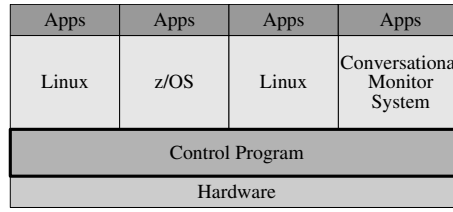

Figure 2.4: z/VM

2.2.3 Partial virtualization

This kind of virtualization only simulates some parts of an underlying hardware environment. A specific case of this method is *address space virtualization*.

Environment supports resource sharing and process isolation but does not allows separate guest operating system instances

Although not generally viewed as a virtual machine category per se, this was an important approach historically, and was used in such systems as CTSS, the experimental IBM M44/44X, and arguably such systems as OS/VS1, OS/VS2, and MVS.

2.2.4 Paravirtualization

This technique has some similarities to full virtualization. It uses a hypervisor for shared access to the underlying hardware but integrates some virtualization parts into the operating system. This approach implies that the guest system needs to be modified for the hypervisor.

This technology born with the need of increase full virtualization performance. It explores ways to provide high performance virtualization of x86 by implementing a virtual machine that differs from the raw hardware. Guest operating systems are ported to run on the resulting virtual machine.

To implement this method, hypervisor offers an API to be used by the guess OS. This call is called “hypercall”. This issue increase performance respect full virtualization.

On the one hand, guest OS needs to be modified and this can mean

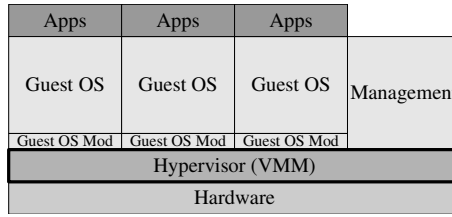

Figure 2.5: Paravirtualization

a disadvantage. On the other hands, this approach offers performance near to the unvirtualized system. In addition, it can run multiple different operating systems concurrently.

Some of the most famous examples of paravirtualization are Xen and Parallels Workstation.

Xen

Xen [7] is a free open source hypervisor that allows a high usage degree and consolidation of servers created by XenSource. It provides mechanisms to manage resources, including CPU, memory and I/O. This is the quickest and safer virtualization infrastructure in this moment. Nevertheless, paravirtualization requires introducing some changes in the virtualized operating system but resulting in near native performance.

Many distributors such as Intel, AMD, Dell, Hewlett-Packard, IBM, Novell, Red Hat or Sun Microsystems use this software. In addition, it has a GPL license and can be download freely.

In a Xen environment a virtual server is just an operating system instance (called domain in the Xen environment) and its load is being executed on top of the Xen hypervisor. This instances accesses devices through the hypervisor, which shares resources with other virtualized OS and applications.

Xen was created in the 2003 by the computation laboratory of the University of Cambridge known as the Xen Hypervisor project, leadered by Ian Pratt. In the next years, the present Xen company was created, XenSource.

The key of Xen success is paravirtualization that allows obtaining a high performance level. Xen gives to the guest operating system an idealized hardware layer. Intel has introduced some extensions in Xen to support the newest VT-X Vanderpool architecture. This technology allows running operating systems without any modification to support paravirtualization.

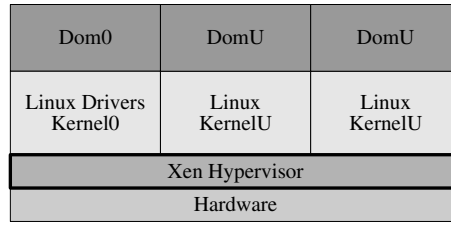

Figure 2.6: Xen

When the base system supports Intel VT or AMD Pacifica, operating systems without any modification like Windows can be ran. With this new architecture and paravirtualization allows this OS without modifications achieve virtualized Linux performance levels.

Overhead introduced by Xen hypervisor is less than 3.5%. In addition, thanks to paravirtualization I/O operations are executed out of the hypervisor and shared between domains following resource sharing policies. Nevertheless, virtualized domains are fully isolated.

Xen also offers some tools like live migration, CPU scheduling and memory management combined with open source software advantages makes Xen a great alternative that allow administrator having a full resources control.

User-mode Linux

User-mode Linux (UML) [8] allows a Linux operating system to run other Linux operating systems in user-space. Each guest Linux operating system is a process. This allows multiple Linux kernels (with their own associated user-spaces).

As of the 2.6 Linux kernel, UML resides in the main kernel tree, but it must be enabled and then recompiled. These changes provide,

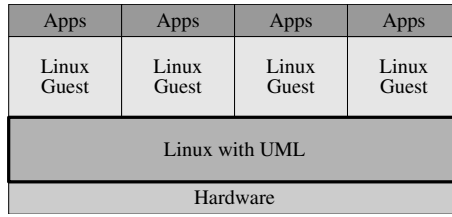

Figure 2.7: User-mode Linux

among other things, device virtualization that allows the guest operating systems to share the available physical devices, such as the block devices (floppy, CD-ROM, and file systems, for example), consoles, NIC devices, sound hardware, and others.

To run kernel in application space, they must be specially compiled for this use. UML can be nested and a guest kernel can run another guest kernel.

2.2.5 Operating system-level virtualization

This method uses a different technique to virtualize servers on top of the operating system itself. It supports a single operating system and simply isolates the independent servers from one another. The guest OS environments share the same OS as the host system and applications running in this environment view it as a stand-alone system.

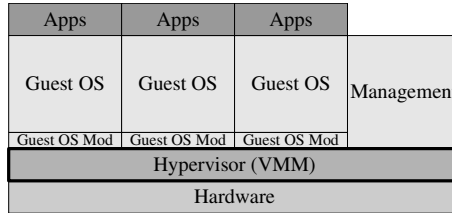

Figure 2.8: Operating System Virtualization

This method requires changes to the operating system kernel but

this implies a huge advantage, native performance. It enables multiple isolated and secure virtualized servers to run on a single physical server. Each one has its own superuser, set of users/groups, IP address, processes, files, applications, system libraries, configuration files, etc.

Whereas VMs attempt to virtualize "a complete set of hardware" a virtual OS represent a "lighter" abstraction, virtualizing instead "an operating system instance". All guests run at top of a single operating system kernel. Its mechanism multiplexes this one OS kernel to look like multiple OS (and server) instances, especially from the perspective of running applications, users, and network services.

Because they virtualize less, it imposes lower overhead than VMs. As a result, more virtual servers can be supported on a given server. Proponents occasionally claim "thousands of VPS per server" in test situations to determine the upper limits of the technology.

OpenVZ

OpenVZ [9] is an operating system-level virtualization solution that supports isolated user-spaces and virtual private server (VPS) built on Linux and available under the GNU General Public License.

It creates isolated and secure virtual environments on a single physical server enabling better server utilization and ensuring that application do not conflict. Each virtual machine can be considered as an independent machine with its own root access, users, IP addresses, memory, processes, files, applications, system libraries and configuration files. In addition, OpenVZ provides a set of management tools to easily create, list or destroy virtual environments.

OpenVZ includes a two-level CPU scheduler that first choose which virtual server has to take CPU control and then gives it to a process of this machine. In addition, it defines resource sharing between VPSs and supports migration of a VPS to a new server.

Virtuozzo

Virtuozzo [10] is a proprietary operating system virtualization product produced by SWsoft, Inc. A version that supports Linux has been available since 2001 and a version that supports Microsoft Windows became available in 2005.

It separate system in virtual environments that behaves in most respects as if it were a stand-alone server. Virtuozzo can support tens to hundreds of VEs on a single server due to its use of operating system-level virtualization. It is available for Linux and Microsoft Windows.

Virtuozzo is based on OpenVZ (SWsoft also supports it), and its concepts are similar to several other operating system-level virtualization implementations, including Solaris Containers, Linux-VServer and FreeBSD Jail.

Virtuozzo supports servers with up to 64 x86 CPUs and 64 GB of RAM, but 1-4 CPU systems are far more common in practice.

2.2.6 Library virtualization

In almost all of the systems, applications are programmed using a set of APIs exported by a group of user-level library implementations. Such libraries are designed to hide the operating system related details to keep it simpler for normal programmers. However, this gives a new opportunity to the virtualization community.

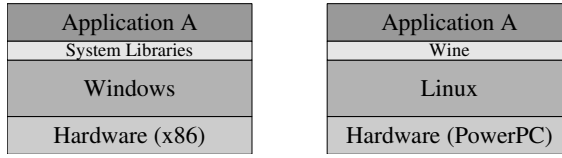

Figure 2.9: Library Virtualization

This type of virtualization is not mostly considered as a technique but it also introduces an abstraction layer (figure 2.9) between applications and underlying system. The most famous library virtualizer is Wine.

Wine

Wine [11] is an open source reimplementation of the win32 API for UNIX-like systems and it can be viewed as layer that allows compatibility for running Windows programs without any modification, for

example, it allows running windows native application to be run in Linux.

Rather than acting as a full emulator, Wine implements a compatibility layer, providing alternative implementations of the DLLs that Windows programs call, and processes to substitute for the Windows kernel.

It was primarily written for Linux, but the Mac OS X, FreeBSD and Solaris ports are currently well-maintained and thanks to this application, major part of standard Windows software doesn't need any modification to be executed in these operating systems.

2.2.7 Application Virtualization

This approach runs applications in a small virtual environment that contains components needed to execute a program such as registry entries, files, environment variables, user interface elements and global objects. This virtual environment acts as a layer between the application and the operating system (figure 2.10), and eliminates application conflicts and application-OS conflicts.

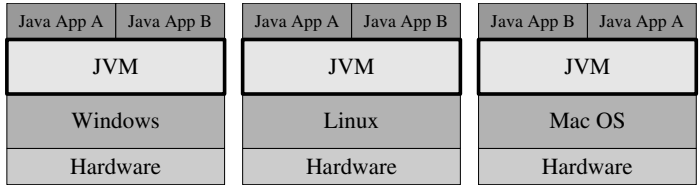

Figure 2.10: Application Virtualization

The most popular application virtualization implementation is the Java Virtual Machine provided by Sun.

JVM

Java Virtual Machine [12] is the most famous and extended application virtualization alternative. This is a software layer that introduces a virtual environment that can execute java bytecodes. It abstracts

Project	Type	Creator
Bochs	Emulation	Kevin Lawton
QEMU	Emulation/Full virtualization	Fabrice Bellard
VMWare	Full Virtualization	VMWare
z/VM	Full Virtualization	IBM
Xen	Paravirtualization	University of Cambridge
UML	Paravirtualization	Jeff Dike
OpenVZ	OS Virtualization	Community
Virtuozzo	OS Virtualization	SWsoft
JVM	Application Virtualization	Sun
Wine	Library Virtualization	Bob Amstadt

Table 2.1: Virtualization types

application from the underlying system, the same code can be executed in a x86 or in a PowerPC architecture.

Because it is available for many hardware and software platforms, Java can be both middleware and a platform in its own right

Software executed on top of the JVM must be compiled into a standardized portable binary format and then can be executed emulating the JVM instruction set by interpreting it, or using a just-in-time compiler (JIT) such as Sun's HotSpot. JIT compiling, not interpreting, is used in most JVMs today to achieve greater speed.

JVM introduces some mechanisms like garbage collecting, CPU management and an interface to access to the overlayed system without taking into account its unique characteristics.

2.2.8 Summary

Many techniques and some implementations of these have been described. These implementations can be summarized in the table 2.2.8.

All these methods are not isolated and can be easily combined if it was desired, for example figure 2.11 an extreme virtualized system is presented. This is a x86 computer that supports VT-X running a Linux that runs a Bochs and a Xen. This Bochs executes a Mac OS X running a VMWare that runs a Linux with OpenVZ. This OpenVZ

runs multiple Linux.

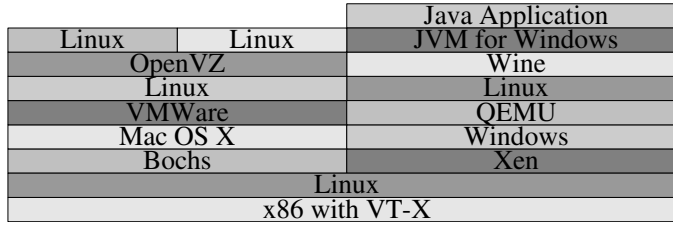

Figure 2.11: Virtualization has no limits

Executed Xen run a Windows thanks to VT-x, this Windows executes a QEMU for Windows that executes a Linux. This Linux execute a Java Virtual Machine for Windows that run a Java application thanks to Wine.

In conclusion, all these virtualization techniques can be mixed as we want with no limits, obviusly having six operating systems running on the same system implies high performance lost. Finally, virtualization is a very flexible technology.

2.3 Implementation issues

Virtual machines execute software in the same manner as the machine for which the software was developed. The virtual machine is implemented as a combination of a real machine and virtualizing software and implementations issues depends on the virtualization technique, nevertheless, the main part of them follow the same philosophy more or less.

Typically virtualization is done by a layer that manages guest petitions (processor demand, memory or input/output) and translate them into the underlying hardware (or to the underlying operating system in some cases) making them executable.

A typical implementation decision in emulation and full virtualized environment is separating executed code between privileged and non-privileged for performance reasons. This decision is based on the

principle that code is executed in different ring levels and virtual machines are tipically in the non-privileged layer and it demand an special control for the privileged instructions.

In the next subsections, some issues of virtualizing different components such as processor, memory and I/O will be discussed.

2.3.1 Processor

Emulating instructions interpreted by the underlying processor is the key feature of different virtualization implementations. The main task of the emulator is convert instructions and it could be done by interpretation or binary translation for instance. Then it executes this code in the underlying machine.

Nevertheless, current architectures like IA-32 is not efficiently virtualizable because it doesn't distinguish between privileged and non-privileged instructions. Some improvements in newest processors to avoid this problem will be discussed in next sections.

Because this limitation, current virtualization engines must identify each instruction and treat it to execute them in the right privileged level.

In addition to instruction interpretation a virtualization technique must deal with scheduling. Meanwhile, a typical operating system uses a scheduling algorithm that determines which processes will be executed in which processor and how long, in a virtualized environment, virtualization layer must take this decisions following different policies.

2.3.2 Memory

Operating system assigns memory pages among processes with a page table that assigns real memory among processes running on the system. And virtual machine monitors uses this host operating capabilities to map memory to each process.

To implement memory sharing between virtual machines there are several ways. but every method maps guest application memory into the host application address space, including the whole virtual machine memory. This mapping is managed by a process (hypervisor in Xen for instance) which. This mapping can be done in a more software way or

relying this decisions to the hardware depending on the virtualization method.

Paging requests are converted into disk read/writes by the guest OS (as they would be on a real machine) and they are translated and executed by the virtualization layer. Then requests are actually made by a single process every time. With this technique, standard memory management and replacement policies still the same than in a non-virtualized machine.

2.3.3 Input/Output

Operating system provides an interface to access I/O devices. This accesses can be seen as a service that is invoked as a system call which transfers control to the operating system. It uses an interface to a set of software routines that converts generic hardware requests into specific commands to hardware devices and this is done through device driver calls.

Implementing Input and Output typically only store the I/O operation and pass it to the overlying system and then return it to the application converting petitions to system specific formats.

2.3.4 Recent hardware support

In the beginning, x86 architecture does not support virtualization and it makes difficult to implement a virtualized environment on this architecture.

Virtualization software need to employ sophisticated mechanisms to trap and virtualize some instructions. For example, some instructions do not trap and can return different results according to the level of privilege mode. In addition, these mechanisms introduce some overhead.

In the year 1974 Popek and Goldberg [13] defined a set of conditions to define if an architecture supports virtualization efficiently.

Main chip vendors, Intel and AMD, have introduced extensions to resolve these difficulties. They have independently developed virtualization extensions to the x86 architecture that are not directly compatible with each other but serve largely the same functions. These

extensions will allow a hypervisor to run an unmodified guest operating system without introducing emulation performance penalties.

This improvements are based on the inclusion of an special mode, VMX, that supports privileged and non-privileged operations and then any instruction can be easily executed without taking into account if it s privileged or not. In addition, this improvement does not introduce an overhead respect a traditional architecture.

Intel is producing new virtualization technology know as IVT (short for Intel Virtualization Technology) that supports hypervisors for both the x86 (VT-x) and Itanium (VT-i) architectures. The VT-x supports two new forms of operation, one for the VMM (root) and one for guest operating systems (non-root). The root form is fully privileged, while the non-root form is unprivileged (even for ring 0). The architecture also supports flexibility in defining the instructions that cause a VM (guest operating system) to exit to the VMM and store off processor state. Other capabilities have been added; see the Resources section.

AMD is also producing hardware-assisted virtualization technology, AMD Virtualization, abbreviated AMD-V (code named Pacifica). Among other things, Pacifica maintains a control block for guest operating systems that are saved on execution of special instructions. The VMRUN instruction allows a virtual machine (and its associated guest operating system) to run until the VMM regains control (which is also configurable). The configurability allows the VMM to customize the privileges for each of the guests. Pacifica also amends address translation with host and guest memory management unit (MMU) tables.

These new technologies can be used by a number of virtualization techniques discussed here, including Xen, VMware, User-mode Linux, and others.

2.4 Virtualization in the real world

All these technologies can achieve different objectives and introduce improvements in different scenarios

One of the most important areas where virtualization can introduce big improvements is hosting. In this scenario, servers can be underutilized and different machines can be consolidated in a phisyc one. Some technologies like *operating system virtualization* and *par-*

avirtualization can achieve desired performance levels in a complete isolated environment. With this solution, fewer machines are needed to attend the same workload with a hardware saving (including costs and space). In addition, it reduces management and administration requirements thanks to migration and replication capabilities of this methods.

Thanks to some virtualization techniques isolation capabilities, virtualization is a great solution for *sandboxing* purposes. Virtual machines provide a secure and isolated environment (sandboxes) for running foreign or less-trusted applications. Virtualization methods that achieve a robust environment for the underlying machine are full virtualization and paravirtualization. Therefore, virtualization technology can help building a secure computing platform.

Multiple environments in a single computer is another virtualization feature. Many of the virtualization types support multiple virtual machines, nevertheless, just some of them achieve a performance level enough for being really usable in real environments, full virtualization and paravirtualization. In addition, virtualization resource managing capabilities also allow resource sharing in a managed way, taking into account virtual machine requirements and giving QoS capabilities to the system.

Last virtualization usage also allows multiple simultaneous operating systems and it allows running specific operating system applications without being necessary to reboot to other operating system. This feature open system dependent applications to every operating system and every architecture.

Thanks to virtualization, architectures or hardware that has never been implemented can be tested. Full virtualization and emulators can achieves this objective, providing new instructions or new features with developing purposes. It also allows a complete profiling that can introduce a considerable overhead, however, developing benefits are much more bigger than difficulties. In addition to architecture virtualization, non existing hardware can be used, for instance, Virtual SCSI drives, Virtual Ethernet adapters, virtual Ethernet switches and hubs, and so on.

Software developing takes great benefits of virtualization and one of the biggest is debugging. Having a complete profiled system permit a complete software debugging. In addition, it can help to debug

complicated software such as an operating system or a device driver by letting the user execute them on an emulated PC with full software controls.

Another improvement can be obtained with virtualization, migration. A application (or the complete operating system) can be migrated to another machine. This feature is one of the features of application virtualization, full virtualization, paravirtualization and library virtualization. With this techniques an application can be move to a different hardware without any modification. In addition, some of these methods allows live migration, in other word, moving an application to an other place while it is being executed.

This characteristic can be moved to a higher level, converting a whole system in a package that can be easily deployed and configured in other machines providing complete software packages.

Combining last virtualization capabilities, a complete test scenario can be easily produced. Having a great amount of machines can be impossible to obtain. Nevertheless, having complete packages that can be deployed as a whole system in a single machine, reduces hardware and deploying time.

Different real usages alternatives will be discussed in the next subsections.

2.4.1 Virtual servers

Xen is a widely extended virtualization alternative for increasing server usage and optimizing global costs and is used by application services providers and hosting companies because it provides a precise resource manager.

Hosted applications rarely makes use of all machine resources. Combining some of them with a complementary server load increases and allocate them in the same computer would increase server utilization and reduce hardware costs. Nevertheless, putting distinct type applications in the same environment without any control would interfere other applications, therefore, is needed to control and isolate them with a mechanism like virtualization.

Introducing this solution, number of used machines is reduced and then cost decrease, nevertheless, it also reduces management costs. Migrating and replicating virtual machines is easier than installing a

complete operating system or check why is it failing. So it reduces time and personal to manage systems with minimum knowledge.

In the last years, virtualization could be considered as a not too efficient solution but in these days alternatives like OpenVZ or Xen has a minimum overhead with a great performance that makes them a real choice.

Nowadays, some hosting enterprises offers virtualized servers known as VPS. Some examples are Spry and its division VPSlink that offers virtual private servers with OpenVZ, linode.com with Xen or Axarnet that uses Virtuozzo.

An important measure of web hosting quality is uptime and using virtualization and its migration characteristics provides a 100% server uptime, an impossible issue with traditional hosting. This is another great virtualized servers advantages.

2.4.2 Research projects

Virtualization management and the facility to change policies according to its needs is a great alternative for research purpose.

Virtualization open new ways in computing and one of these is resource managing. There are many research projects like Tycoon [**?**] that manages compute resources in distributed clusters like PlanetLab, the Grid, or a Data Center. This system is based on credits and users pay for resources and they can provide resources to earn credits.

It allocates resources according to automated economic mechanisms with more efficiency than manual allocation and it uses Linux and Xen as a prototype.

Another project that take profit of virtualization resource managing is an adaptive control in data centers [14] that dynamically adjusts the resource shares to individual tiers in order to meet application-level QoS goals while achieving high resource utilization in data centers.

Porting resource managing to a higher level, virtualization also allows creating and destroying new machines, duplication, migration in an easy way. This set of facilities can be used in autonomic computing allowing self-managing and reducing administration time.

Exists an IBM project [15] that takes advantage of IBM Virtualization Engine to give autonomic features to their system. With these

capabilities they achieve a high level of efficiency in system administration. This system manages servers, storage, system and network. This is a great solution to optimize any infrastructure management.

Another project that gives autonomic features to their solution is a system that implements a virtualized server with autonomic management of heterogeneous workload [16] that uses Xen management capabilities. This system innovation and the key feature is that allows virtual machine migration to achieve job machine requirements and shares resources according with specified policies taking into account each virtual machine load.

In conclusion, all these projects take advantage of virtualization to resolve well known problems and giving new solutions for this purpose.

2.4.3 Development

In the IT development, virtualization can make easier development tasks and it can be used in many areas such as software development or security issues. Working in this type of environment introduces some improvements respect traditional environments.

This computing area has been highly benefited by virtualization. This was an area that implied many time for deploying, managing and other tasks that were not strictly needed for developing, thanks to virtualization these undesired tasks have been mostly removed saving many time and it has made development easier.

Software development

Virtualized environments are used in development and software testing because of it allows developers use multiple virtual machines and check it introducing a basic issue: hardware cost reduction. In addition, tested hardware can be easily adapted to change system characteristics according with developer needs.

Another advantage is porting software from the test environment to a production one migrating this machine. This deployment time has been eliminated and applications start running instantly.

Virtualization is also used as a sandbox for critic application development. Developing a kernel or a module can crash the machine many times and introducing a minimal layer that isolates real system

from the working one to develop applications would make this task easier. Therefore, developer can work without being afraid of crashing the whole system and reducing time to reboot the whole system.

For instance, the Linux kernel occupies a single address space, which means that a failure of the kernel or any driver results in the entire operating system crashing. Applying virtualization if one operating system crashes due to a bug, the hypervisor and other operating systems continue to run. This can make debugging the kernel similar to debugging user-space applications.

Mobility scenarios

Taking into account virtualization implementation issues, it allows taking the whole virtual machine state. This allow migrating a virtual machine to another machine including its state. This feature enable developing new applications that supports execution for a large period, when the overlaying machine needs to be maintained it can be moved to another machine.

Furthermore, a virtual machine can be stored periodically to avoid systems failures due to power problems or hardware fails and restored immediately, obtaining a high availability degree.

Virtualization can also be seen as a middleware that abstracts underlying system and therefore implementing software in a virtual machine can be ported to any architecture that supports that virtualization layer without any modification.

Security

Thanks to virtualization, a system can be considered as a safe environment and protect the overlayed system and the rest of virtual machines from possible attacks or failures.

In security developing projects, virtualization has also great advantages. For instance, in virus profiling, this job can be done in a virtual environment without any risk and allowing a complete system profiling thanks to VM characteristics.

In a local area network a *honeypot* implemented on a virtual machine representing a system with some typical bugs or security weaknesses for attracting hackers that try to attack the network and dis-

tract them from the really important systems of the network. In addition, this honeypot can be highly monitored to make an early detection of possible intrusions.

From the local network security view, virtual machines can be a way to easily restore infected systems. Thanks to virtualization management capabilities, a minimal system installation or system backups can be stored in a server to restore them later if it was necessary.

Having multiple users in a single machine implies a risk, isolating each user in a restricted virtual machine reduce these risks to the minimum expression. Using a virtualization method some restrictions like preventing some instructions executions, restricting traffic network... can be specified, giving a high security level.

Finally, virtualization is a great tool for security issues that gives many facilities to security experts.

2.5 Conclusions

Virtualization is an old technology that was forgotten and nowadays is becoming one of the most used computing trends because its capabilities.

In this chapter, virtualization types and how they are implemented in real products have been explained. So many products have been presented with their own features. Deciding which alternative should be used in each environment according with its features is a key issue.

Taking into account open new ways, we can conclude that virtualization is a great solution.

Bibliography

[1] M. Tim Jones. Virtual linux. 2006. http://www-128.ibm.com/developerworks/library/l-linuxvirt/index.html.

[2] Bochs. http://bochs.sourceforge.net.

[3] Qemu. http://fabrice.bellard.free.fr/qemu.

[4] Virtualbox. http://www.virtualbox.org.

[5] Vmware. http://www.vmware.com.

[6] z/vm. http://www.vm.ibm.com.

[7] Xen. http://www.xensource.com.

[8] User-mode linux. http://user-mode-linux.sourceforge.net.

[9] Openvz. http://openvz.org.

[10] Virtuozzo. http://www.swsoft.com/en/virtuozzo.

[11] Wine. http://www.winehq.org.

[12] Jvm. http://java.sun.com.

[13] Formal requirements for virtualizable third generation architectures. 1974. http://www.cs.auc.dk/~kleist/Courses/nds-e05/papers/vmformal.pdf.

[14] Xiaoyun Zhu Mustafa Uysal Zhikui Wang Sharad Singhal Arif Merchant Kenneth Salem Pradeep Padala, Kang G. Shin. Adaptive control of virtualized resources in utility computing environments. http://www.eecs.umich.edu/~ppadala/research/dyncontrol/eurosys07.pdf.

[15] Lori Simcox. Autonomic features of the ibm virtualization engine. 2004. http://www-128.ibm.com/developerworks/linux/library/ac-ve/.

[16] Ian Whalley David Carrera Ilona Gaweda Malgorzata, Steinder and David Chess. Server virtualization in autonomic management of heterogeneous workloads.

[17] An introduction to virtualization. http://www.kernelthread.com/publications/virtualization/.

[18] Server virtualization: let battle commence. 2006. http://www.cbronline.com/article_feature.asp?guid=609D18C1-C9F9-42A5-9BE3-B5B3B781C91B.

[19] Eric Van Hensbergen. The effect of virtualization on os interference. http://research.ihost.lv/osihpa-hensbergen.pdf.

[20] Bryan Clark. A moment of xen: Virtualize linux to test your apps. 2005. http://www-128.ibm.com/developerworks/library/l-xen/.

[21] Rami Rosen. Introduction to the xen virtual machine. 2005. http://www.linuxjournal.com/article/8540.

[22] Tzi-cker Chiueh Susanta Nanda. A survey on virtualization technologies. http://www.ecsl.cs.sunysb.edu/tr/TR179.pdf.

[23] Gabriel Torres. Intel virtualization technology (vt) explained. 2005. http://www.hardwaresecrets.com/printpage/263.

[24] Intel® virtualization technology. http://developer.intel.com/technology/virtualization/index.htm.

[25] Pradeep Padala Sharad Singhal Zhikui Wang, Xiaoyun Zhu. Capacity and performance overhead in dynamic resource allocation to virtual containers. http://www.eecs.umich.edu/~ppadala/research/dyncontrol/im07.pdf.

[26] Franck Cappello Benjamin Quetier, Vincent Neri. Selecting a virtualization system for grid/p2p large scale emulation. `http://www.lri.fr/~quetier/papiers/EXPGRID.pdf`.

[27] T. Garfinkel M. Rosenblum. Virtual machine monitors: Current technology and future trends.

[28] James E. Smith and Ravi Nair. *Virtual Machines: Versatile platforms for systems and processes.*

Chapter 3

Self-managed policies, a survey

Ferran Julià and Ramon Nou

Abstract

The increasing complexity, heterogeneity and scale of systems has forced to emerge new techniques to help system managers. This has been achieved through autonomic computing, a set of self-* techniques (self-healing, self-managing, self-configuring,etc...) that enable systems and applications to manage themselves following a high-level guidance. This chapter is centered in the self-management capability of autonomic systems, it pretends to give an overview of the three most popular mechanisms used to achieve self-management, action policies, goal policies and utility function policies. We present a summary of autonomic system's architecture and an extended view of the different policy mechanisms analysing the usefulness of each one.

3.1 Motivation

The motivation of this chapter is basically to give and overview of the most current techniques used to develop the self-managed systems. This types of systems are emerging and many different articles and implementations have appeared in the recent years. As the complexity of systems reclaims to reduce the charge of systems' administrators.

We have followed the approach used by Kephard in order to classify the different autonomous systems. We think this classification adjusts very well to most of systems, and it remarks the utility function model which we thing is the most emerging and may be the most effective model.

3.2 Introduction

"Biological systems have inspired systems design in many ways:Artificial Intelligence, Artificial Neural Networks, Genetic Algorithms, Genetic Programming, and Holonic Systems to name a few. The most recent is the inspiration to create self-managing systems.", as said by R. Sterritt in [15], designers have copied the idea of human body and applied to autonomic systems, this gives systems the ability of manage themselves in an automatic way taking decisions to preserve its integrity and performance.

The major difference between this to systems is that while in nervous systems the decisions are involuntary in IT systems the decision are taken from designer's rules. Independently of where it comes, it's clear than the increasing complexity, scale, heterogeneity and dynamism implies a huge management effort, this has forced the investigators and designers to create mechanisms to reduce the management complexity of such systems. The last purpose of autonomous systems is to avoid the administrator to directly manage the system, instead of that they give systems' administrators some high-level rules that will make the system change its behavior according to this guides.

Autonomic systems are composed by autonomic elements, and this elements interact one with each other in order to follow the high-level polices. M.Parashar [14] divide the different parts of autonomic system/application that can be autonomous in eight: Self-Awareness,

Self-Configuring, Self-Optimizing, Self-Healing, Self-Protecting, Context Aware, Open, Anticipatory. The four first make reference to management or decisions aspects, and the rest are design or implementation characteristics. Due to the complexity of such systems we can consider the decision aspects as the most complex and important, the development of all these characteristics has its own algorithms and techniques, but all follow the same autonomic architecture described in the following sections.

This chapter is centered in Self-managing or Self-optimizing characteristic of autonomic systems, in order to understand the different ways to archive this in the following sections we introduce some architecture and design characteristics of autonomic systems valid for implementing any kind of autonomic element. As we discuss later on, the crucial part of this type of systems is the taking of decision (the "intelligence"), is in that part of the autonomic schema where we have centered the survey.

The chapter is divided in five main sections. After the introduction we will make an overview of autonomic architecture and give some definitions to better understand the rest of the chapter, the third section exposes the different ways to design the decision taking procedures, what we call management policies in self-managed systems and also we'll give some examples of use. In fourth we expose some conclusions and finally in five section we give some future trends.

3.3 Architecture

For the understanding of a self-* system and more in concrete the self-managed ones it's necessary to know how such types of applications are usually structured. Obviously there are much many ways to design an autonomic systems but we believe the one presented here is representative of most of them. Almost all architectures are equivalent, it depends on how you define the parts or layers involved, but the "philosophy" behind them it's always the same.

In this chapter we'll also give a view of two important concepts in autonomic world, the difference between open-loop and closed-loop systems and the typical architecture of an autonomic system. The former is a way to classify any kind of distributed systems, and in

the second we will expose some important concepts in order to better understanding the chapter.

The choice of open or closed loop when developing an autonomic system would be the first one to take because this fixes the overall architecture.

The architecture presented in this section is a closed-loop one, and it's independent of the type of self-* or the type of policy based system that implements it, the ideas described here can be applied to all autonomic closed-loop systems.

3.3.1 Open-loop vs Closed Loop

Due to its analogy, we can consider that autonomic IT systems are control systems. This systems take decisions over its behavior depending on some input information.

The terms Closed-loop and Open-loop come from electrical engineering, they are used for identify the two possible ways to design a control system. The main difference between them is that the former uses feedback information when taking decisions and the second only uses a reference input.

Open-loop (also known as feed-forward control) systems are controlled directly by an input signal without the benefit of feedback. Open-loop control is useful for well-defined systems where the relationship between input and the resultant state can be modeled by a mathematical formula. The feed-forward controller uses the input signal (and may be disturbance or noise signals) to determine the control input that will make the system target achieve the desired output. 3.2 shows the typical schema of such control system.

One problem of this approach is that to construct it we need an accurate model of our system and the mechanism must be robust enough to changes in environment. Let's imagine that we have an Apache Tomcat application server and we want to configure it to do not consume more than 75% of CPU, when can achieve that setting the max number of worker threads, as they are in charge of accept clients, deppending on how much of them we have much CPU the application will consume. Let's supose know that we want to change it to 50% which will be the correct max worker threads value? To find the correct number of threads we would need to know our system in detail. We

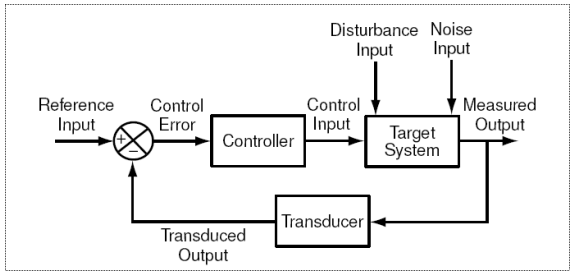

Figure 3.1: Open-loop control system bloc diagram

cannot apply lineality between worker threads and CPU consumption, the only way to find the correct realtion is by empirical experimentation. An incorrect setting would drive the system to a no desirable state unpossible to repair through that control system.

Unfortunately the typical ebusiness systems are much complex than a simple Tomcat, if we only introduce a new tier to the server (a data base) the system is practically impossible to control with open-loop. It's easy to see that, lets take the last example and we suppose we can control the CPU consumed by setting the number of max worker threads on Tomcat, due to all the requests made by the clients are different surely the requests to the database will be very different (in time and cost). This is very difficult to predict and could cause that worker threads stopped waiting for database response which will imply a different CPU consumption than we expected.

It's very difficult to implement an open-loop control system that manages with unpredictable changes as occurs with typical workloads of public web servers.

Closed-loop (or feedback) control systems use the measured outputs to determine the control inputs. In **??** we have a diagram that shows how such type of systems typically work. The control system adjust the values to achieve a measured output as similar as possible to the reference input with the help of the output measures obtained from the last settings. This enable the system to readjust itself even under unpredictable situations.

The design of feedback control systems it's a very complex task,

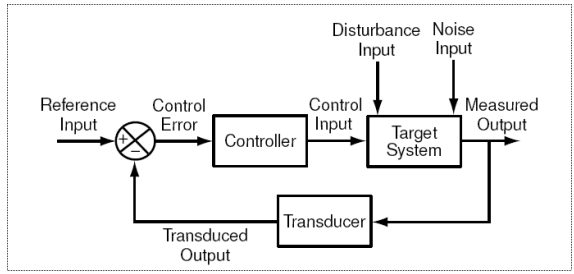

Figure 3.2: Closed-loop control system bloc diagram

there are some properties that they must have, but the most important it's stability. A system it's said to be stable it for any unbounded input the output is also bounded. There are some mathematical approaches that make a theoretical treatment to that problem [9].

Obviously this type of systems also involve knowledge of application (not as deep as open loop) but we always have to be careful of stability. Remember last example where we had Apache Tomcat with a database, if we use now a closed-loop control system to control Tomcat CPU Consumption, suppose that our design it's not good enough, we could have a situation like the following: Most of the clients are making requests that involve large database queries, so there are a lot of worker threads stopped. An improperly designed control system would, for example, increase (with inverse proportion of measured output) the number of workers, as the CPU Consumption is very low the new number of workers threads is so high than if all database request turn to be short the system would collapse.

Although open-loop systems are less difficult to design, they can only be used in very stable and controlled environments, but in computing systems the major part of environment are unpredictable, for that reason the most typical autonomous systems are designed using feedback control.

An autonomous system with fixed states that use closed-loop controllers is more easy to design, as if there are instabilities you can notice easily about them, although with this approach we can loose some of benefits some times it's worthy to have a less intelligent system

that never fails than a clever but unstable one.

3.3.2 The Autonomic cycle

There are several approaches to achieve a self-manager platform in terms of architecture, we introduce here the one we used with very successful results in the past [**?**] which is based on IBM's architecture[2]. This is not the only one but we think it's enough representative to describe the typical components of self-managed systems and we can consider the rest of them as little variations of the one described here.

The IBM's original proposal described four basic components that work together in a life-cycle to adapt and efficiently run a system in constant flux. These components combine to provide a service in accordance with the policies of the application or system and can continuously adjust themselves while conforming to dynamically changing factors through out its run time. This simple but powerful concept has attracted a lot of attention recently [8, 4, 3]] as it can provide a solution to help operators navigate and run modern day servers, which have become increasingly perplex and intricate environments over time.

The four components that are needed in an Autonomic System, as described by IBM 3.4, are a General Manager, an Autonomic Manager, Touchpoints and Managed Resources.

The General Manager decides which policies should be used to construct an overall plan. This plan is then used to guide the application or system and tell it what it has to do to reach a desired healthy state.

The Autonomic Manager is very similar to the General Manager, having an analogous life-cycle, with the goal of producing and executing a plan according to predefined policies. The Autonomic Manager component performs this at a lower level however and is therefore often considered the "core" of the autonomic system. It takes care of the self-management life-cycle whereby it reads the system and manages it according to the changes in those readings and their relation with the identified policies of the system and application. Managed Resources are those resources that the self-managed system is able to control. It could map directly to a physical resource such as a hard drive or it could be a logical resource such as a communications channel.

A Touchpoint is essentially an interface which is used to link the Autonomic Manager to the Managed Resource. The interface has two

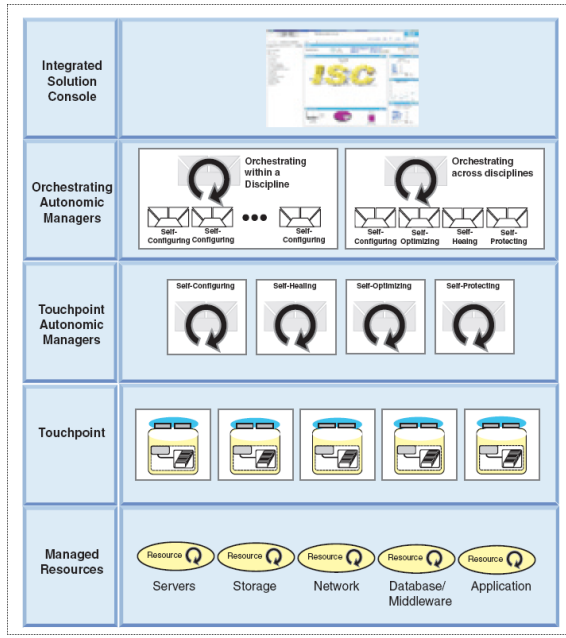

Figure 3.3: Autonomic computing layered architecture

different methods for providing interaction between these components. The first, sensors, enable us to consult and check the system's behavior second, effectors, actually let us modify the behavior that resource. We can calculate and change the system state using these.

We can find variations of this architecture, they have this layers defined in a different manner, but in essence they are all the same.

The self-managed life-cycle **??** is a general mechanism with which any application can manage itself and consists of four distinct phases; monitoring, analyzing, planning and executing. Initially it needs knowledge of the different possible states of the system and how they can be determined using the values available from the sensors. At startup the A.M. can load all of this as well as the policies which will be used to plan the running of the application. Once it is up and running,

the monitoring phase is where it calls the sensors of the resources and reads their values. Having completed monitoring, it moves on to the analyzing stage, where it compares the values obtained in the monitoring phase with the possible states to calculate the current state. The planning stage is entered next and a plan is formulated based on the current state we are in so that the system can be led to its desired state. Finally the Autonomic Manager executes this plan by making calls to the effectors of the Managed Resources. The manager repeats the entire cycle every X seconds to capture any changes and adapt its policies.

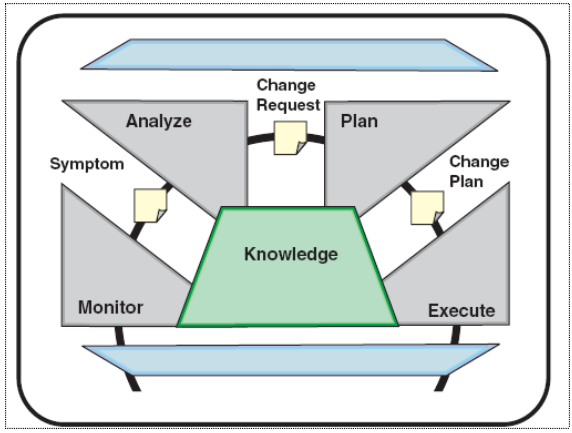

Figure 3.4: Autonomic computing life-cycle

The parts of this architecture that require more attention are the analyzing and planning states, where the system choose itself the behavior of the following step. The policy mechanisms described in the further section applies in this stages, some of them combining, in practice, both into one only stage.

3.4 Achieving Self-management

The final idea of autonomous systems is to translate high-level directives into specific actions to be taken by elements. This is achieved by the use of policies. The policy represents the desired behavior, this policy-based self-managed systems has been studied since some time ago [17], [19]. To present the broadest situations in policy-based systems we use the same approach than Kephard and Walsh in [10].

They divide possible designs in three types: action policies, are self-managed systems driven by conditions like IF (Condition) THEN (Action), e.x. IF (CPU consumption is greater than 80%) THEN (Reduce the number of worker threads). This are the most simple policy based systems. The system get the values measured by the sensors and applies the rules that satisfy the condition. The next type are goal policy based systems. This type of systems specify the desried dicisions to be attained without specifying how to attain them, e.g. CPU consumption can not exceed 66%. This approach is better that the first, because the administrator don'thave to know detail of application internals, and this facilitates the communication between different autonomous elements. The most complex design are the ones that use utility funcitions policies. The utility function specifies de desirability of alternative states. This is done assigning generical values to all the possible states. The goal of this system is to maximize the utility, this is the best approach as allways get the better solution, the one that maximizes the system's overall utility. The last ones can be viewed as a subgoup of the goal policy based ones.

The location in the autonomic cycle of the different mechanisms described in this section involve the analyze and plan stages of the cycle. All the systems presented describe the way the values measured form the system are treated and how and why decisions are taken. Although here we present this mechanism as one, is usual to divide the task, like we described in the architecture section, first analyzing the measured values and then determining the actions to take, in the plan stage.

This methodologies are often used to control and decide the allocation of resources in shared platforms [5]. They are useful because the changing resource demand of such environments needs intelligent control system to avoid wasting resources. I

3.4.1 Action policy based

Action policy based mechanism is the most direct way to implement a self-managed system, it's based on the principle of action-reaction. All the decisions taken by the manager follow the If (CONDITION) Then (ACTION) statement.

Once obtained the actual state or position of the system through the sensors, the next step is to make some *action* that directly or indirectly drives the system to another state 3.5. The idea of the action polices is that changing one or some of the effectors the system will change to the desired state.

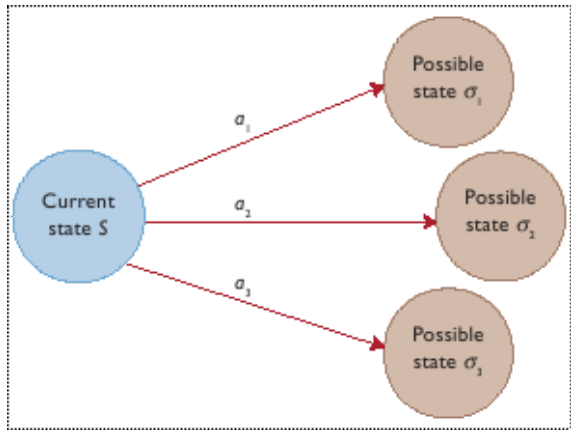

Figure 3.5: States and actions

This type of policy systems make the assumption that changing the effectors the systems is going to change as we expect with a high probability. This implies that the designer of such systems must know not only the internals of the application but it's behavior when changing some of its configurable parameters and effectors.

We can find two examples of action policies in [1], [7], although the two papers implement action policy based techniques to achieve the QoS requirements the main difference between them is that while the former uses an "atemporal" event based system the framework

developed by the second is done using a "life-cycle" which obviously involves frequency.

Efstratiou et. al.in their paper describe an interesting way to achieve the self-management porting some ideas from event calculus. The idea is take some of the reasoning of event and changes and construct an action-policy guided by events. They define the states as specific situations that have some time duration, this states are defined by the events that can initiate or terminate them. This allows to define predicates and with this predicates we can evaluate the different states of our systems. and following the **if** (condition) **Then** (action) make the system change.

As we can see the system does not depend on any time controlled daemon, the systems changes depending on the event values, the time dependency relies on how we define the states not on the frequency of taking sensor measures.

On the other hand Lutfiyya et. al. implements the typical action-policy based system. Instead of changing the system when an event occurs there is a cycle that repeats until end of application that take a decision on each turn. They use their own formalism to specify policies.

One example of action-policy algorithm would be 3.6, this is the typical example where we have a QoS we must achieve and *while* is the taking decision or plan cycle. The allocation reduction or improvement could be changed for the change of effectors depending on the system.

As we have seen the action-based policy self-managed systems do not specify the state in which the system should be, as we will see that occurs in Goal-policy based ones, the system is programmed to make **actions** depending on **conditions**, this has high reliability in the action-reaction principle.

The major inconvenient of this systems are firstly that the designer of such systems must know in detail the behavior of the application although this is common in all self-managed systems it has more relevance on this systems due to the "low level" programming of them. With low level here we mean that the designer has to set values for all effectors, may be guided by a high-level policy, but there is more manual component. The system relies on a human that explicitly gives it the rational behavior

The second and also very important issue is that this type of sensors

Given: Application p.
Initial resource allocation a.
QOS metric m.
QOS expectations e.

---

1. p is started with resource allocation a.
2. **while** (p **not done**) **do:**
3. Execute code in p.
4. $q \leftarrow$ QOS measured with m.
5. **if** ($q > e$) **then**
6. Determine allocation reduction r.
7. $a \leftarrow a$ - r.
8. **else if** ($q < e$) **then**
9. Determine allocation improvement i.
10. $a \leftarrow a + i$.
11. **endif**
12. **endwhile**

Figure 3.6: Action-policy algorithm

doesn't guarantee the stability of the systems, as typically the decision taken does not depend on past states. This implies that we could have an oscillating systems, this problem is typical in feedback control systems [9]. It's difficult to detect such type of problems in this type of systems as we do not have clear view of the system behavior under all the different possible conditions.

Note that the designer must choose the states in a way that they cover all the possible "space of states" and assure that each state is mapped with an unique action. This often drives to conflicts among actions, which might be not detectable via semantic checking and might surface only at runtime.

Another inconvenient of such systems is that the system could not react as we expect, when constructing them we assume that the actions taken by the system when it's in an A state will drive to B (more desired) state. In some situations, this could not be true, specially if

we do not control all the possible variables that influence our system, as usually happens. We can minimize this effects ensuring that we cover the whole space of possibilities when designing the states of our system.

3.4.2 Goal policy based

These type of policies don't use the If (condition) Then (action) conditional structures, the idea is to define a preferred state or group of preferred states instead of specifying exactly what to do when we are in a concrete state.

The choice for that desired states can be done in several ways, for example J. Rolya in [11] made an statistical approach in order to determine the demand of the application, with the demand of the application you can easily see when the system is overloaded and from them define the desired states for the application. May be a more typical approach is to extract demands and possible configurations from a queue model [16], which doesn't have the real component of the former empirical method but can give you more specific information as you can study concrete parts of the application.

A more complex approach is done by Chandra in [**?**] where they use modeling unitedly with online measures which becomes what we can call *online modeling*, it's a more adjusted approach as the parameters of the model are set online, and the results are recalculated every cycle-time.

All of this methods have the common property of not relying in explicit encoding, any of them fixed the actions to do when the application is in a concrete state, the self-managed system make predictions based on application's previous behavior and regulates itself in function of the measured sensor's values in order to achieve a desired state. The policy or the high-level rule here is applied at design level by taking the prediction model and configuring it corresponding to designers interest.

Another way to construct a goal-based is through control theory [23], this approach doesn't require any a priori modeling. The systems tries to adjust a fixed input value setting the effectors correctly. There is a reference input fixed, which it is not any measured sensor of the application, and the self-managed systems regulate the effectors

in order the get an output as similar as possible to the reference input. The reference input is the desired value of the system's measured output the controller adjusts the setting of effectors so that it's measured output is equal to reference input, and the transducer transforms the measured output so it can be compared with the reference input.

This type of systems are much more complicated that the action-based ones, as they can involve a more theoretical approach using control-theory.or the design of complex queuing models.

As we have seen in this models there aren't any prefixed state as in action-policy based ones, here the states are more abstract. If we think in an state as a vector of sensor values we could say that in action-policy based systems this change discreetly while in goal-policy based one they change in continuous way.

Note that this type of systems could have problems in situations where the resources are scare and the system can't satisfy all the goals, or when resources are plentiful and multiple states might satisfy the goals and the system is not able the choose the better state among the correct ones.

3.4.3 Utility function based

What we talk about when we talk of utility functions? The term *Utility function* comes from a branch of economy called *Consumer theory* [20], it is used for indicate or measure the relative happiness or satisfaction of consumers when buying goods or services. The particularity of this approach is that expresses utility in function of real goods (pe.x. kilograms, litres,....) instead of nominal goods (dollars, euros).

In economy say that there are two rules in optimizing behaviors utility maximization and profit maximization. The idea is apply the utility maximization to computer systems.

The utility is a numerical rank value assigned to each option in a choice. This rank is in a way that the most preferred is the one that have the high utility value. To qualify as a true utility scale however, the rating must be such that the utility of any uncertain prospect is equal to the expected value (the mathematical expectation) of the utilities of all its possible outcomes (which could be either "final" outcomes or uncertain prospects themselves).

The decisions taken by a rational agent can be easily mapped to a

numerical range, in a way that we can easily rate any possible outcome "simply" comparing and ranking them. For example if A is preferred to B and B is preferred to C it's clear than A would be preferred to C. Although it can seem easy, it's sometimes difficult to find a rating system that posses the above described fundamental property.

One theoretical way to do so is to compare prospects and/or final outcomes to tickets entitling the holder to a chance at winning some jackpot, which is at least as valuable as any outcome under consideration. A ticket with a face value of 75% means a chance of winning the jackpot with a probability of 0.75 and it will be assigned a utility of 0.75. Anything which is estimated to be just as valuable as such a ticket (no more, no less) will be assigned a utility of 0.75 as well.

In real life, utilities are not linearly related to money values (or else the lotteries would go out of business), which is another way to say that the mathematical expectation of a monetary gamble need not be the proper utility measure to use. The monetary expectation is only a special example of a utility, which is mathematically acceptable but not at all realistic. It is, unfortunately, given in elementary texts (which do not introduce the utility concept) as the sole basis for a rational analysis of gambling decisions.

The use of utility-based resource allocation in computer systems goes all the way back to 1968 when Sutherland [18] presented a futures market in which the users could bid for computer time based on their own utility functions. They has a server that had to be shared among students and faculty members, in order to give priorities they distributed some virtual currency in function of projects importance. They could reserve the computer paying with the virtual money and it was returned once the users had used their reserved time. A user could not bid more than he could afford so the users with the most yen had the advantage.

Since there there have been many applications of utility function based model to computer systems, most of them to autonomous systems. We can find several recent examples of self-management (or in this case better said self-optimizing) systems in [6], [22], [12], [13], [21].

The utility function based model for self-managed systems can be viewed as an extension of the goal policy based model, rather than performing a binary classification in desirable or non desirable states, they assign a real-valued desirability to each state. The author no

specifies a preset desired state, instead the system tries to achieve the state that has the higher value of utility function.

A utility function is written as U = f(x1, x2, x3,...xn) where xi are "real goods" that contribute to utility, in a computer system xn could be for example resources allocated, demand space, etc... thinks that we can obtain from our system but by themselves doesn't attain knowledge.

The use of linear utility functions is disallowed, because proper utility function must be bounded when the stakes are potentially unbounded, this makes more typical the use of exponential functions instead.

For better understanding of how utility functions are used we can take a look to [22] or [12] where this theory is applied to a Data center, in order to achieve a self-optimized application. In both systems the architecture used 3.7 has different autonomic levels, the Resource arbiter and the Application Managers, each of one is able to allocate o re deallocate resources at it's own level. The applications managers send the utility functions $U(S_iR_i)$ calculated to the resource arbiter in order to allocate the resources maximizing the global utility: $\sum U(S_iR_i)$. In these examples the variables S,R can be for example service demand, resource levels (CPU utilization), using simple functions we can obtain a numerical value for the utility function $\sum U(S_iR_i)$.

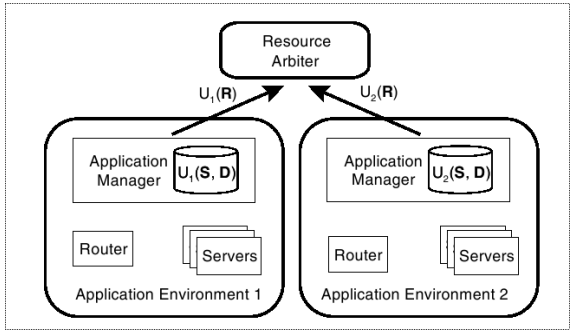

Figure 3.7: Architecture of the data center

The values used as variables in utility functions can be measures

sensors or values obtained from simulation, any kind of variable that give us information about the system.

3.5 Conclusion

This chapter pretends to give an overview of self-managed systems, more in concrete in the decision part of such systems. The chapter explains the different ways to implement policy based self-managed systems and give some examples and references that implement them.

In the first section we introduced the two main different approaches of self-controlled systems, open-loop and closed-loop. Most of self-managed systems are designed following the feedback system. It's very important to know the typical problems that this type of systems has when designing them. There are some solutions to these problems that comes from the mathematical treatment and can be applied to the IT systems.

We have exposed an architectural approach, based on IBM's one, and that we think it's the most general one. It's important to follow a well structured architecture as this simplifies the overall process of designing and also makes more easy the interaction between the different parts.

We could locate the policy system in the Autonomic or General Manager layers and in the analyse and plan stage of the that Managers. The classification can be divided, basically in two great blocks ,the action based and the goal based, we have also considered a very large and important subgroup of goal systems the utility function based systems. The action policy based systems are the most simple when implementing but is the one that requires much low level knowledge of the system. The goal policy based is more complex but more effective method and the utility function based systems are the most effective and the most emerging ones.

As we have seen the self-managed systems comprises a large set of disciplines and can be applied to very different types pf systems. The approaches used are different in every system but the most of them follow the classification proposed. This indicates that although we can classify them there is a lack of standardization in this area, due to the different requirements of the studied systems.

The utility function self-managed systems seems to be, by th moment, the best approach to this type of autonomous systems.

3.6 Future Trends

It's difficult to say what will be the standard or the facto of autonomous systems in the following years, because their application is wide and depending on the area where it applies.

One of the future trend, in our opinion will appear, is the adding of simulation in the taking decision stage. This technique consists in make some simple and quick simulations, using different configuration parameters, the choice of this parameters can be done through several ways ex: using genetic algorithms. The simulator give the system which is the most appropriate configuration for the next interval. With the help of the simulator we could predict the behavior of the application with high accuracy.

Autonomic systems are a mix of knowledge and techniques from very different areas, in our opinion is in that mixing of specialities where the most effort has to be put. Which a which probability techniques developed for artificial intelligence will be more involved in the self-managed management. One of the goals of the architecture described in the previous section is that enables and facilitates the interaction of such different contributions.

What is clear is that the future trend is to isolate from the low level and apply techniques as utility functions or discontent functions to negotiate SLA, this implies that the self-managed systems have to treat with them.

Bibliography

[1] N. Davies K. Chevers C. Efstratiou, A. Friday. Utilising the event calculus for policy driven adaptation on mobile systems.

[2] IBM co. *An architectural blueprint for autonomic computing.* www.ibm.com, 2004.

[3] M. Spreitzer M. Steinder D. M. Chess, G. Pacifici and A. Tantawi. Experience with collaborating managers: Node group manager and provisioning manager. In *Second International Conference on Autonomic Computing*, pages 39–50, 2005.

[4] M. Bennani D. Menasce and H. Ruan. On the use of online analytic performance models in self-managing and self-organizing computer systems. pages 128–142, 2005.

[5] M. Bennani D. Menasce and H. Ruan. Dynamic resource provisionign for self-adaptative heterogeneous workloads in smp hosting platforms. 2007.

[6] William E. Walsh Jeffrey O. Kephart Gerald Tesauro, Rajarshi Das. Utility-function-driven resource allocation in autonomic systems. 2005.

[7] M. Katchabaw M. Bauer H.Lutfiyya, G. Molenkamp. Issues in managing soft qos requirements in distributed systems using policy-based framework.

[8] D. Ardagna C. Francalanci J. Almeida, V.Almeida and M. Trubian. Resource management in the autonomic service-oriented architecture. pages 84–92, 2006.

[9] S. Parekh D. Tilbury J. Hellerstein, Y. Diao. *Feedback Control of Computing Systems*. John Wiley and Sons, 2004.

[10] W. Walsh J. Kephart. An artificial intelligence perspective on autonomic computing policies. In *Fifth IEEE International Workshop on Policies for Distributed Systems and Networks*, volume 0, page 3, 2004.

[11] Martin Arlitt Artur Andrzejak Jerry Rolia, Xiaoyun Zhu. Statistical service assurances for applications in utility grid environments.

[12] Marko Kankaanniemi. Self-optimization in autonomic.

[13] Terence Kelly. Utility allocation directed. In *First Workshop on Algorithms and Architectures for Self-Managing Systems*, pages 2003–2115, 2003.

[14] M. Parashar and S. Hariri. Autonomic computing: An overview.

[15] H. Tianfield R. Sterrit, M. Parashar and R. Unland. A concise introduction to autonomic computing.

[16] Omer M. Asad Wei Jin Amin M. Vahdat Ronald P. Doyle, Jeffrey S. Chase. Model-based resource provisioning in aweb service utility.

[17] M. Sloman. Policy driven management for distributed systems. In *Journal of Network and Systems Management*, volume 2, 1994.

[18] I. Sutherland. A futures market in computer time. In *Communications of the ACM*, volume 11, pages 449–451, 1968.

[19] WebPage. Policy workshop: International workshop on policies for distributed systems and networks. http://www.policy-workshop.org/.

[20] WebPage. Econ model. http://www.econmodel.com/classic/terms/utility_fun 2007.

[21] Baochun Li Weihong Wang. Market-based self-optimization for autonomic service overlay networks. In *Selected Areas in Communications, IEEE Journal*, volume 23, pages 2320–2332, 2005.

[22] Jeffrey O. Kephart Rajarshi Das William E. Walsh, Gerald Tesauro. Utility functions in autonomic systems. 2004.

[23] Sujay Parek Rean Griffith Gail E. Kaiser Dan Phung Yixin Diao, Joseph L. Hellerstein. A control theory foundation for self-managing computing systems. 2005.

Chapter 4

Comet architecture for web applications

Sergi Baila and Vicenç Beltran

Abstract

The last two years have seen a revolution on the way web applications are developed. The popularization of new techniques under the acronym AJAX (Asynchronous Javascript and XML) has made web applications a lot more interactive and closer to desktop applications. At the core of this new approach is the ability of a web page script to send requests to the server without user prior action. This breaks one of the limitations of web applications and the HTTP protocol, and now a web application can trigger an asynchronous partial page update which makes applications a lot more responsive and interactive, and also hides latency effects.

This technology has evolved and has become quickly a new foundation for developing web applications which are closer to desktop applications. Web mail, calendars, instant messaging... Also, common bussines software is starting to be developed as a web application, even on intranet scenarios. However, AJAX is still limited by the underlying HTTP protocol and it's request/response cycle. On this known

client-server architecture the browser is the one which always initiates actions (send requests). Desktop application frameworks, based mainly on the MVC software pattern, implement GUIs which are based on an event-response model. Events can be fired on the client side but also on the server side. Web applications face a significant problem here. Perhaps even bigger than common desktop applications which tend to be single user whereas web applications are starting to be designed from the beginning as multi user applications. So the need for a server propagated event model is more necessary.

Comet is a new approach which uses an open idle connection, mainly unused, until there's a need for the server to push information to the client. This allows the push of events from the server to the client, so the gap between desktop applications and web applications is further reduced. But keeping an open connection per client breaks classic servers' scalability where you have one thread per connection. New server implementations based on asynchronous I/O are already available, which can handle thousands of connections with just a pool of threads.

This chapter introduces AJAX and Comet architectures, the new frameworks, and the servers which implements them on top of asynchronous I/O. We also analyze the new problems introduced by these technologies. AJAX relies completely on JavaScript, the DOM model, CSS... all web technologies which are now starting to see standards compliant products. Portability is one of the first problems encountered by an AJAX developer even between minor revisions of the same browser. Also, usability of web applications suffers from the AJAX approach as existing mechanisms for disable people aren't prepared yet for this new technique.

4.1 Introduction

On February 18th 2005 Jesse James Garret published a short article[1] on his company website coining a new buzzword on the internet world. No one suspected that essay would be seen later as the first milestone of a revolution on the way we understand web applications. There was no new technology, because the ingredients were present for some time, nor there was no new product. Instead he pointed to existing products

like Google Suggest or Google Maps. But that short name, AJAX, was rapidly spread among technological publications, blogs and sites.

But that was just the name. Most people, me included, had the first encounter with the new technology and a glimpse of the possibilities behind with a sub project from Google called Google Suggest (back in 2004). It was a simple product, just the google page with a twist added: as soon as you start typing the web page started to show suggestions of searches along with estimated result count. So you typed "car re" and google suggests "car rentals" but also "car reviews" and so on. Most non technical people saw it's speed and ease of use. We technical people were amazed as how it broke the classic HTTP request response and full page reload mechanism.

The magic behind relates to the XMLHttpRequest object, which was created by Microsoft (as the ActiveX object XMLHTTP) and later (2002 and beyond) was implemented on Mozilla and some other browsers. This object allows to send an HTTP request and retrieve the response as an asynchronous javascript method call. This is what Google Suggest uses to send a request to a server each time there's a keystroke and then parsing the HTTP response.

So the evolution of the usage and impact of this technology has been somehow exponential. It took nearly four years to reach a side project on google, then some other sites started using similar effects (GMail, Google Maps, Flickr, ...) and a name was adopted on 2005. That same year saw the explosion of the technology. This can be considered the first milestone of the future web applications.

There has been always a clear gap between web applications and desktop applications. Before 2005 the answer to the question "do we need a web application or a desktop application?" was easily answered because web applications were very poor and the only advantage was that they are distributed and easily available applications with a very thin and common client. Then came AJAX and applications like GMail which broke the limitation of the full page reload model based on the HTTP request response model. It was not the first step nor the last, but an important one. The HTTP protocol was not designed as a foundation of general purpose applications. Actually, it was not designed with any application on mind, even classic web applications. One of the first important limitations resolved in the past was the stateless property of the protocol. Today with the use

of cookies or URL rewriting, and with every web developer framework supporting sessions, this seems an easy task. We are here to take a look at the next step to narrow the gap between desktop applications and web applications: Comet or how can you build a event-based web application.

4.2 Background

In this section we provide the necessary background information for those not familiarized with web technologies. Some concepts of the HTTP request response model are presented. Then we introduce the JavaScript environment available on web browsers and we dive into AJAX as the precursor technology for Comet.

4.2.1 HTTP model

The Hypertext Transfer Protocol is a communications protocol designed mainly for the retrieving of HTML pages and accesory elements (CSS pages, images, etc.). It is a request/response client/server protocol. That means that the model is clearly and strictly defined[2]: the client (browser) sends a request to the server which reads the whole request, processes it and returns back a response (see figure 4.1). An HTTP client (known as the user agent) establishes a TCP connection on port 80 (the standard one, but could be any) of the web server in order to send the request and retrieve the response. The server listens to that port and can serve multiple clients simultaneously.

The HTTP model has several limitations for developing a web application. It is a stateless protocol, bonded to a strict request/response cycle. The stateless problem was solved with the use of cookies or URL rewriting to keep a session between the client and server. Until recently, that was the foundation for developing web application, and is what we call here the classic model (figure 4.1). On this classic model each time there was an action from the user the browser sent a request to the server which resulted on a new page loaded. This is what we call the full page reload model. Given current network latency, even on a local area network, is very difficult to develop a web application with the same funcionality as a desktop application. We

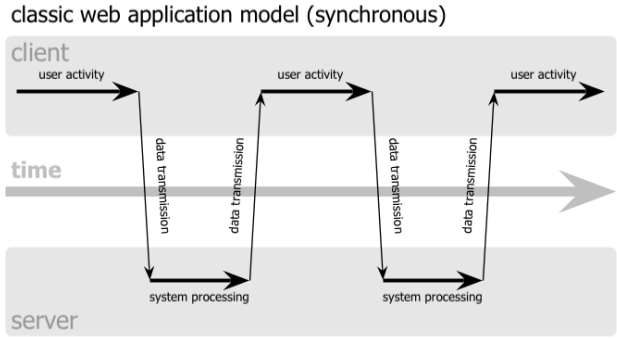

Figure 4.1: Classic HTTP model

will see how AJAX solves this and brings us the next model.

Another problem arises with a Comet architecture that we will explore on a next section and is introduced by a HTTP protocol limitation. The protocol [2] limits (by suggestion) the number of simultaneos connections from a user agent (browser) to the server to just two. Using new techniques like HTTP pipelining and a classic or AJAX model this is not of much concern. But with a Comet model using a permanent connection there's just one left.

4.2.2 JavaScript

JavaScript is nowadays a real distributed execution environment, being the standard language for script execution inside web pages. Its real name is ECMAScript[3] and its evolution is tightly close to that of the web browsers. This scripting language allows, within a browser, to manipulate most of the components of the web page (the document structure via a DOM(Document Object Model)[4] interface). It is a quite powerful language which not only can manipulate the Document Object Model but also can be used to listen on events, use it asynchronously, parse XML and even send HTTP request from within a web page without triggering a complete reload (this is the base for AJAX).

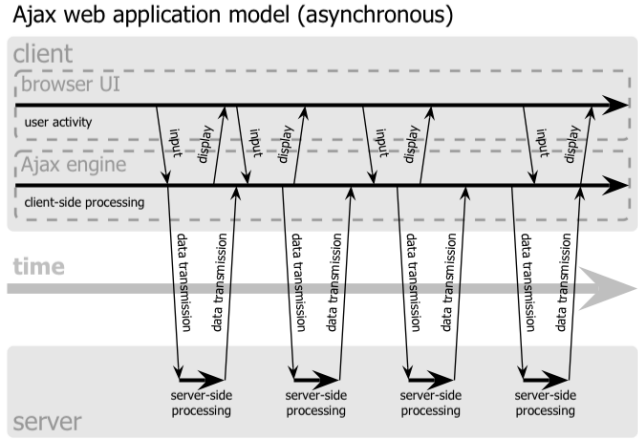

Figure 4.2: AJAX HTTP model

4.2.3 The AJAX model

For years web developers had faced the problem of having a full page reload every time they wanted to get or set new data to or from the server. But the advent of the XMLHttpRequest object and it's easy asynchronous usage led quite rapidly to a new breed of web applications. There was no more a synchronous and closed request and response cycle. We can now have a request sent on the background which response triggers a partial change (thanks to the ability of JavaScript to manipulate the page through the DOM). We can have then partial page updates, background server communication and requests made by programming logic and not subject to user interaction. We will call this the AJAX application model as seen in figure 4.2.

The AJAX acronym [1] stands for Asynchronous JavaScript And XML. The original concept was suposed to use XML as the language to encapsulate the response where the JavaScript has the ability to parse it and modify the page state and contents via the DOM interface. Some applications use that model, but most of the time developers use a simpler and stripped down model where the response is just HTML (partial page) and the action to do is just replace some part of

the page. Also a common usage is encapsulating JavaScript code on the response so the server can trigger any event on the page. All of this work has been greatly simplified with the development of JavaScript frameworks like Prototype or Dojo.

4.3 Introduction to Comet

As J.J. Garret coined the word AJAX there was also a blog post from Alex Russell [5] where he tried to follow the same path coining the Comet term to refer to the possibility of the server to send events to the client without having to wait for a request from the browser to arrive. Also as with the AJAX term, there were prior works on the area to solve the problem of a web application being unable to receive asynchronous events from the server. We're just referring to this milestone as a signal of the maturity of the idea.

The reason for this need was actually a consequence of the exit of the AJAX architecture. A great number of new highly functional web applications were developed with AJAX and both developers and users wanted to push developments further [6]. But as interactive it was an AJAX application it lacked a core mechanism from desktop application: real time updates. Developers notice that it was necessary to propagate events from the server to the client in order to have an event-driven web application.

The problem again was the HTTP protocol. It's a client-server protocol, without option to the server to contact the client. Also, given the diversity of networks and connections between browsers and servers, building any mechanism for the server to open a connection to the client is out of the equation.

So there's only one solution possible (without severe modifications of the underlying protocol). To have an open connection idle just waiting for an event on the server (any Comet client should have one then). So when the server has to send an event to one, some or all of the clients, it just uses the open connection (which is a standard HTTP connection). You can see figure 4.3 for a diagram of the Comet model with an HTTP streaming connection technique. The beauty of the solution is that it works. And works without modification of clients, servers, protocols, etc. Unfortunately, the problem introduced

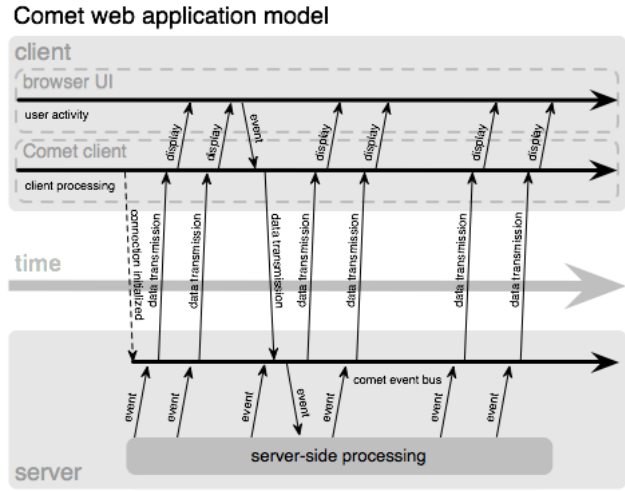

Figure 4.3: HTTP streaming Comet model

is of a different nature. The servers suffer from a scalability problem with this architecture. [5] [7] [8] [9]

4.3.1 Comet architecture

The tradicional implementation of web servers, specially an application server (like a JavaEE server) uses a pool of threads to manage incoming connections. When a new connection arrives it's established and a thread from the pool is assigned to the connection. Request is read, code is executed and a response is generated and sent. Then, the thread returns to the pool. This is designed under the assumption that requests are short in duration but intensive in computing resources. But a Comet connection is established and is expected to be long in duration (can be several minutes) and very low CPU or memory intensive. The connection is only required for sending events to the client and just keep the connection open.

A classic web application can handle easily on the order of tens of thousands of simultaneous users, because users are not sending re-

quests all the time. So the number of active connections on the server is always a fraction of the users of the application at any given time. With a Comet architecture each user on the system is an open connection on the server and a thread (with a classic model) which is locked to the opened connection. Even if it's not doing something, any server has problems managing tens of thousands of threads.

The servers need to be redesigned around this new problem. The solution comes from a know mechanism, asynchronous I/O, which has existed in modern operating systems for a log time. C programmers know it as the select() or poll() system call. Java, for example, has support for it since version 1.4 with the introduction of the java.nio packages. [10] [11]

The new design decouples the one to one relationship between connection and thread. There's also a thread pool, but threads are also used to process active connections, not connections which are not handling data. Of course, developers of server side components need to do some modifications, but they're only needed on the comet handlers.

Besides scalability on the server the Comet architecture introduces another subtle problem on the client side. The HTTP protocol [2] limits on 2 the number of simultaneous connections to a server. Using at least one for a Comet connection leaves the whole page with just one connection. As the page is probably using the AJAX model it's obvious that a complex or simply slow response would block all the other connection and leave the page unable to send any other request as we have the two connections busy: one for the comet connection and another waiting for the slow reponse to an AJAX call. So this is nearly impossible to circumvent but it can be alleviated. One necessary step is to stream all Comet communication to the same and only connection as no page can afford to have the two connections busy on different components.

We've seen that the Comet architecture posses a series of challenges both on the server and the client. We are now presenting the internal details and work done on the model.

4.3.2 Bayeux protocol

As a non standarized architecture Comet faces significant interoperability problems. Actually there are as protocols as implementations.

Some of the major names behind certain libraries and servers are pushing for a standard protocol of communication between a Comet client (JavaScript library) and a Comet server component. The result of this is the Bayeux protocol [12] with the Dojo Foundation behind it. There's also work in progress from the authoritative source W3C for HTML 5 server sent event listeners [13] but without any real work impact yet.

The lead person behind Bayeux is Alex Russell from Dojo which guarantees a certain level of notoriety for the protocol. As he states in his first post [**?**] about Bayeux: "One of the biggest problems facing the adoption of Comet is that it's, by definition, not as simple. It's usually not possible to take ye-old-RESTian HTTP endpoint and suddenly imbue your app with realtime event delivery using it unless you want your servers to fall over. The thread and process pooling models common to most web serving environments usually guarantees this will be true. Add to that the complexity of figuring out what browsers will support what kinds of janky hacks to make event delivery work and you've got a recipe for abysmal adoption. That complexity is why we started work on Bayeux."

As they define it: "Bayeux is a protocol for transporting asynchronous messages over HTTP. The messages are routed via named channels and can be delivered: server to client, client to server and client to client (via the server)". The protocol specification is in a very initial stage but has seen some support from the community which see it as a good way to push the architecture support and ease of development further.

The protocols tries to address the main problems associated with the Comet architecture. It uses JSON (JavaScript Object Notation) as the data interchange format to define the messages. Those messages are clearly defined on the specification and cover all the low level technical details needed as the handshake, connection negotiation, channel subscription, reconnection, etc. The standarization of the messages allow the development of interoperable client libraries and server components. Further, it ensures that key concepts like negotiation and reconnection are taken into account even for simple developments. The protocol also introduces a versioning system which allows to negotiate between client and server for a preferred protocol level in the same way as the HTTP negotiation works.

A key concept on the protocol is the multiplexing of different endpoints for comet components via a mechanism of channels. Each message sent with the protocol has a channel destination, which helps alleviate the two connection limit problem of HTTP. So having different server components accessed via Comet no longer wastes multiple connections but just one. It also helps server components to clean up and separate things. Another helpful introduction is the identification of each client with an autogenerated id, much the same way as an HTTP session id.

Another advantatge of a standard protocol is the support for multiple connection models and a negotiation protocol. The Comet architecture is really a hack over the limitations imposed by the HTTP protocol, so different connection methods are not only necessary but desirable to support as many clients as possible. We're going to take a look at the different Comet connection models.

4.3.3 Comet connection models

We've seen that a Comet architecture needs somehow a permanent connection to the server in order to be able to receive server generated events. But the handling of this connection can be different on the way is managed mainly in the client side but also affecting the server side. Choosing the right one is not an easy answer [9].

The first one and the first used before the advent of Comet or even AJAX is the polling connection model (figure 4.4). This can't really be considered a Comet model because it's not receiving the event when it happens but we're incluing it here as a base idea which has been used in the past and is a perfect example of an scenario where Comet can really help.

Of course this model has a clear scalability problem. As it has not the problem of keeping an idle connection, the number of polling request received on the server can be extremely high with a high frequency value which will be desirable in order to make the application responsive. This model can work with a small number of users even with an update every 2 or 3 seconds. This model also has other drawbacks. There is an overhead of a new request and response. Also, probably the main problem, depending on the application usage some or many of the connections could be empty, just the client asking the

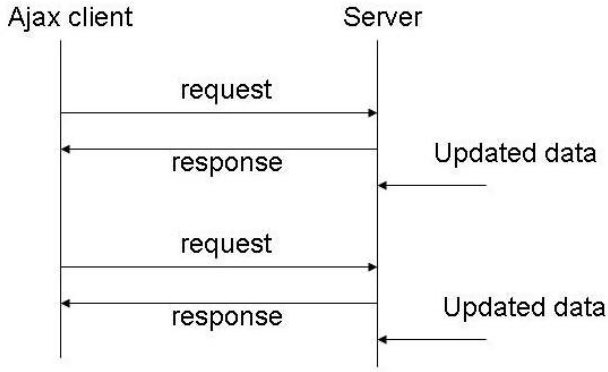

Figure 4.4: Polling connection model

server for events and getting a negative response. This overloads the server and the network for nothing.

Long polling is an evolution of this technique which solves the problem of the void requests because the server only responds when there are data to. Meanwhile, the connection is just waiting. You can see on figure 4.5 that the model just sends a request waiting for data on the server which also waits until there's some event. So the requests are not returning void never, but you have on average as many connections as clients on that page.

As this model solves the void response problem it may introduce an scalability problem on those servers which block on the request and have a classic 1:1 mapping between threads and connections. That's because if you want your application to be able to scale to tens of thousands (or more) simultaneous users you will have at least as many connections on the web application. So, for example, having 10.000 users on your application means you will have 10.000 AJAX connections because of your long polling model. That on a classic server translates to 10.000 threads just waiting (sleeping) on each connection doing nothing but wasting resources. Even if your OS, TCP/IP stack and so supports that it's unlikely your application server do. Fortunately, new servers (Grizzly [14]) and revisions on old ones (Jetty [8],

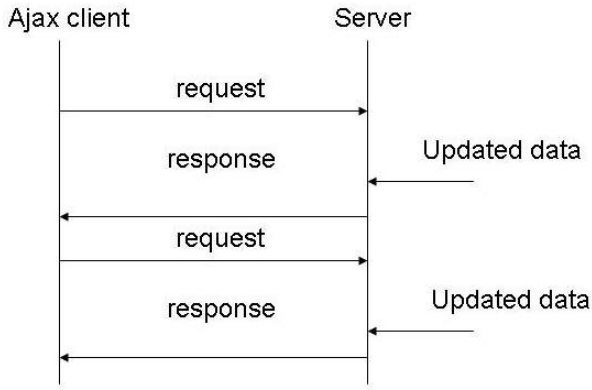

Figure 4.5: Long-polling connection model

Tomcat, Apache) implement what's called Asynchronous Request Processing [14] which is based on non blocking I/O, a mechanism found on recent revisions of OS and libraries [15] [10].

Of course this isn't the perfect solution. If the server is pushing events fast enought you will find yourself in a similar scenario as with the polling model where you have several connections and a big overhead for each request/response cycle. Also both models suffer from the network latency specially as they need to send a new request for each response (event) received. There's also some bandwith wasted on the multiple requests.

This leads us to the third model called HTTP streaming [16], a model similar to long polling but without closing the connection even after getting a response (see figure 4.6). The trick here is to use a transfer mechanism from HTTP [2] called chunked transfer encoding, which allows to send a response build up of blocks of data (chunks) without knowing the amount of data and lenght of each chunk in advance. This fits exactly to a series of events on the server which need to be propagated to the client without knowing in advance the number of events, the lenght or most important when they will happen. This model greatly helps leveraging the network usage as eliminates the overhead of multiple requests and reduces the latency because the

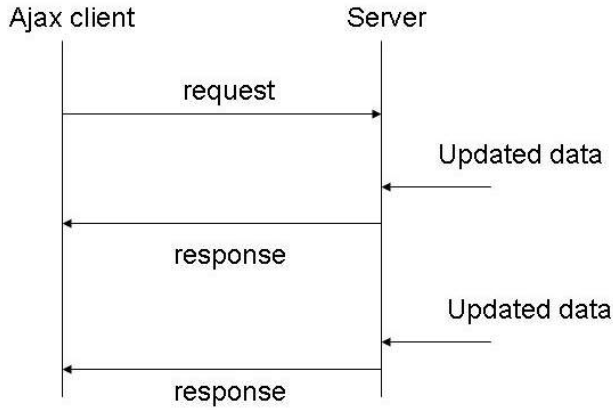

Figure 4.6: HTTP streaming connection model

response can be sent without waiting for a request to end.

Even HTTP streaming is of course not exempt from caveats. Not only the server should support thousands of connections with a limited number of threads on the pool, on big scenarios you can find yourself with too many events which can't be correctly propagated to the clients because of network congestion. So some kind of event throttling should be considered.

Surely there are also some challenges that need to be addressed. For example even with the HTTP streaming connection model there's no way that the client can send events to the server on the opened channel. A bit ironic that the standard way of communication doesn't work, but in this case is more a limitation of the XMLHttpRequest JavaScript construction that needs a complete request before starting the transaction.

4.4 Scalability issues

There's no extensive work on the AJAX and Comet impact on performance in web environments, and existing work has very preliminary results [17]. Even without extensive experimental evidences the one

to one mapping between connections and threads doesn't seem the best idea. And not only Comet HTTP streaming or long polling connection models benefit from it, some testing indicates that certainly most kinds of web application can benefit from it. J.F. Arcand, one of the engineers behind Sun's Grizzly server, has done [18] some syntethic test and real benchmarks over Grizzly asynchronous request processing module based on Java no blocking library java.nio. The throughtput of static files and simple JSP and servlets is quite the same (see figures 4.7, 4.8 and 4.9) but a classic connector (Tomcat Catalina) needs a 500 threads pool to match a 10 threads pool on a ARP connector. Testing the maximum number of users that a website can handle (figure 4.10) with a maximum response time of 2 seconds on 90% of request and an average think time of 8 seconds show a clear winner of the non blocking model because there is less context switching and more available memory with the far lesser number of threads of the second model.

4.5 Comet frameworks

As the AJAX and Comet technologies evolve and popularize we see an increasing number of frameworks appearing. Actually, the number of AJAX frameworks is growing very quickly [19], perhaps because it's quite new technology and the market hasn't done the natural cleaning for the best ones. Anyway just a very small subset of this frameworks support Comet so we're centering on the most popular ones. We will first take a look at some of the developer libraries to implement Comet solutions. We will introduce Pushlets, a combined library which uses a client JavaScript component and a Java servlet for the other side. As a different example, Dojo is a more general purpose framework written completely in JavaScript without server side components. The Comet part is solved implementing the Bayeux protocol.

We will then introduce three of the server which implement some kind of asynchronous request processing using non blocking I/O. Grizzly from Sun is the web container for their JavaEE server GlassFish and is built from the ground thinking on asynchronous request processing. Jetty is a very popular servlet and JSP container which was one of the first (if not the first) to implement a solution with its Continuations mechanism. The newest Apache Tomcat version 6 includes

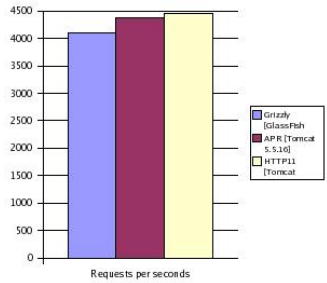

Figure 4.7: ARP 2k file static performance

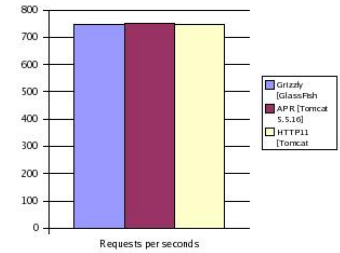

Figure 4.8: ARP 14k file static performance

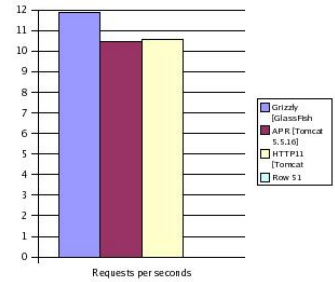

Figure 4.9: ARP 954k file static performance

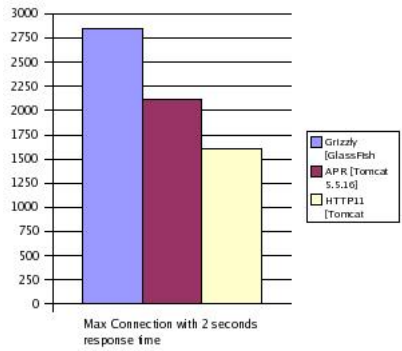

Figure 4.10: ARP maximum number of simultaneous connections with 2s response time

an ARP connector.

4.5.1 Client libraries

"Pushlets are a servlet-based mechanism where data is pushed directly from server-side Java objects to (Dynamic) HTML pages within a client-browser without using Java applets or plug-ins. This allows a web page to be periodically updated by the server. The browser client uses JavaScript/Dynamic HTML features available in type 4+ browsers like NS and MSIE. The underlying mechanism uses a servlet HTTP connection over which JavaScript code is pushed to the browser. Through a single generic servlet (the Pushlet), browser clients can subscribe to subjects from which they like to receive events. Whenever the server pushes an event, the clients subscribed to the related subject are notified. Event objects can be sent as either JavaScript (DHTML clients), serialized Java objects (Java clients), or as XML (DHTML or Java Clients)." [20]

The Dojo toolkit is a modular open source JavaScript toolkit (or library), designed to ease the rapid development of JavaScript- or Ajax-based applications and web sites. It was started by Alex Russell in 2004 and is dual-licensed under the BSD License and the Academic Free License. The Dojo Foundation is a non-profit organization de-

signed to promote the adoption of the toolkit. [21]. Alex Rusell is the responsible for the word Comet [5] and one of the people behind the Bayeux Protocol [22] [12]

4.5.2 Server solutions

Grizzly is the HTTP server component for the new reference JavaEE server Glassfish from Sun. A description of Grizzly from one of his creators: "Grizzly has been designed to work on top of the Apache Tomcat Coyote HTTP Connector. The Coyote Connector is used in Tomcat 3/4/5 and has proven to be a highly performant HTTP Connector when it is time to measure raw throughput. But as other Java based HTTP Connector, scalability is always limited to the number of available threads, and when keep-alive is required, suffer the one thread per connection paradigm. Because of this, scalability is most of the time limited by the platform's maximum thread number. To solve this problem, people usually put Apache in front of Java, or use a cluster to distribute requests among multiple Java server. Grizzly differ from Coyote in two areas. First, Grizzly allow the pluggability of any kind of thread pool (three are currently available in the workspace). Second, Grizzly supports two modes: traditional IO and non blocking IO." [15]

Jetty is a 100% pure Java based HTTP Server and Servlet Container. Jetty is released as an open source project under the Apache 2.0 License. Jetty is used by several other popular projects including the JBoss and Geronimo Application Servers. This server was probably the first breaking the one thread per request mapping with it's Continuations [8] and provide a sort of Comet server framework before even the concept was clear.

Apache Tomcat is a web container developed at the Apache Software Foundation (ASF). Tomcat implements the servlet and the Java Server Pages (JSP) specifications from Sun Microsystems, providing an environment for Java code to run in cooperation with a web server. It adds tools for configuration and management but can also be configured by editing configuration files that are normally XML-formatted. Tomcat includes its own internal HTTP server. Since version 6, Tomcat supports a NIO HTTP Connector and has native Comet support.

4.6 Conclusions

The Comet architecture allows to develop web applications based on server sent events. Because of the nature of the HTTP specification the only way to really have near real time event propagation from client to server is keeping an open connection. We've seen this introduces serious scalability problems on servers but they can be and are being adressed using new models for processing requests based on non blocking I/O systems.

There are currently production ready servers with support for asynchronous request processing. There are multiple libraries supporting Comet models and even a standard protocol (Bayeux) with some support behind it. So it's safe to say Comet is ready for production and actually it's being actually used in several public web applications.

The Comet architecture represents another step into the evolution of web application like AJAX has been on the last two years. In the following years we will see a proliferation of AJAX and Comet enabled web applications that will implement funcionality only available to desktop applications today.

4.7 Future Trends

The gap between desktop applications and web applications is getting small. Not also because there are the technical mechanism available but also because people has started to think about web applications and browser as the ultimate application framework. Is not unlikely a future were most of the applications are web based and built upon web standards [23]. Probably not the ones the current ones but an evolution. That road would bring several challenges which will need to be addressed.

In the middle of the nineties there was a boom coming from the hardware and software major vendors about the thin clients, net clients or NetPCs. It was a vision of things to come, but as many vision it was too much ahead of time. Nowadays we can start talking about the WebOS again, and think of the true mobility where you will have all your desktop computing environment anywere there's a Internet connection. Most of us have already a web based email system which

we can read from anywhere in the world (who hasn't read email on holiday on a very far and remote computer?). Google is one of the pioneering companies behind products like GMail, Google Calendar and Google Docs. Today you can have on the web the email, a calendar, a word processor, a spreadsheet, an instant messenger, a music player, a company files repository... all of the applications most company computers execute at the end of the day. Mobility is a demanded requirement today as sales for laptop systems exced desktop systems. The next step could be simplifying the laptops, making it smaller, more durable, more usable and rely on the network for bringing the applications.

This of course is still years ahead but there's an important wind of change on the industry and companies like Microsoft and Apple who mostly rely on selling an operating system should start thinking in other terms. The software bussines is also changing, and subscription models are starting to become interesting on a world where someone cares about the software, updates, storage of data, etc. Will the WindowsOS be hosted on Microsoft server and billed for usage or monthly rates?

What is clear is that the revolution is starting at the web application level and the Comet architecture is just a single step on that direction.

4.8 References / Further Reading

We're listing some references with some examples and further readings work which could be useful to complement this chapter. On [24] AJAX is applied at the middleware level. Mesbah and Deurse [25] define an architectural style for a single page AJAX model while Khare and Taylor [26] propose an extension to the REST architectural style for decentralized systems. Jacobi and Fallows [27] explore on a single article the Comet architecture and Bayeux protocol.

Bibliography

[1] Jesse James Garrett. Ajax: A new approach to web applications, 2005. http://www.adaptivepath.com/publications/essays/archives/000385.php.

[2] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – http/1.1. *Internet RFCs*, 1999. http://tools.ietf.org/html/rfc2616.

[3] E.L. Specification. Standard ecma-262. *ECMA Standardizing Information and Communication Systems*, 3, 1999.

[4] A. Le Hors, P. Le Hegaret, G. Nicol, J. Robie, M. Champion, and S. Byrne. Document object model (dom) level 2 core specification version 1.0. *W3C Recommendation*, 13, 2000.

[5] Alex Russell. Comet: low latency data for the browser, 2006. http://alex.dojotoolkit.org/?p=545.

[6] Rohit Khare. Beyond ajax: Accelerating web applications with real-time event notification, 8 2005. http://www.knownow.com/products/docs/whitepapers/KN-Beyond-AJAX.pdf.

[7] Wikipedia page for comet. http://en.wikipedia.org/wiki/Comet_%28programming%29.

[8] Greg Wilkins. Jetty 6.0 continuations - ajax ready!, 2005. http://web.archive.org/web/20060425031613/http://www.mortbay.com/MB/log/gregw/?permalink=Jetty6Continuations.html.

[9] Jean-Francois Arcand. New adventures in comet: polling, long polling or http streaming with ajax. which one to choose?, 2007. http://weblogs.java.net/blog/jfarcand/archive/2007/05/new_adventures.html.

[10] Giuseppe Naccarato. Introducing nonblocking sockets, 2002. http://www.onjava.com/pub/a/onjava/2002/09/04/nio.html.

[11] Nuno Santos. Building highly scalable servers with java nio, 2004. http://www.onjava.com/pub/a/onjava/2004/09/01/nio.html.

[12] Greg Wilkins Alex Russel, David Davis and Mark Nesbitt. Bayeux: A json protocol for publish/subscribe event delivery, 2007. http://svn.xantus.org/shortbus/trunk/bayeux/bayeux.html.

[13] Web apps 1 / html 5, 2007. Server sent events specification http://www.whatwg.org/specs/web-apps/current-work/#server-sent-events.

[14] Jean-Francois Arcand. Grizzly part iii: Asynchronous request processing (arp), 2006. http://weblogs.java.net/blog/jfarcand/archive/2006/02/grizzly_part_ii.html.

[15] Jean-Francois Arcand. Grizzly: An http listener using java technology nio, 2005. http://weblogs.java.net/blog/jfarcand/archive/2005/06/grizzly_an_http.html.

[16] Http streaming. http://ajaxpatterns.org/HTTP_Streaming.

[17] Youri op't Roodt. The effect of ajax on performance and usability in web environments, 8 2006. http://homepages.cwi.nl/~paulk/thesesMasterSoftwareEngineering/2006/YouriOpTRoodt.pdf.

[18] Jean-Francois Arcand. Can a grizzly run faster than a coyote?, 2006. http://weblogs.java.net/blog/jfarcand/archive/2006/03/can_a_grizzly_r.html.

[19] Michael Mahemoff. 210 ajax frameworks and counting. *ajaxian.com*, 2007. http://ajaxian.com/archives/210-ajax-frameworks-and-counting.

[20] Just van den Broecke. Pushlets - whitepaper, 8 2002. `http://www.pushlets.com/doc/whitepaper-all.html`.

[21] Dojo toolkit. `http://en.wikipedia.org/wiki/Dojo_Toolkit`.

[22] Alex Russell. Cometd, bayeux, and why they're different, 2006. `http://alex.dojotoolkit.org/?p=573`.

[23] Aaron Weiss. Webos: say goodbye to desktop applications, networker 9, 4 (dec. 2005). *netWorker*, 9(4):18–26, 2005.

[24] John Stamey and Trent Richardson. Middleware development with ajax. *J. Comput. Small Coll.*, 22(2):281–287, 2006.

[25] Ali Mesbah and Arie van Deursen. An architectural style for ajax. *wicsa*, 0:9, 2007.

[26] R. Khare and RN Taylor. Extending the representational state transfer (rest) architectural style for decentralized systems. *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, pages 428–437, 2004.

[27] Jonas Jacobi and John Fallows. Enterprise comet: Awaken the grizzly!, 2006. `http://java.sys-con.com/read/327914_1.htm`.

Chapter 5

Job Self-Management in Grid

Marta Garcia Gasulla and Julita Corbalan

Abstract

Grid computing is growing as a competitive distributed computing environment. In this chapter we want to focus on a specific topic of Grid computing, the Job management, and specially in environments that provide self-management of jobs.

The interest and importance of self-management is growing as an alternative to centralized schemes. Its point is that it offers a solution to the scalability problems inherent to Grid computing. Opposed it raises other problems such as, how to provide fault tolerance or how to manage the lack of a centralized control. The purpose of this chapter is to discuss the solutions to these problems that have been proposed from different perspectives.

With the first section about user level APIs we want to present the standardization efforts that Grid-related communities are doing. Besides, we aim to provide the reader a general view of the concepts and most usual requirements related to job management in Grid computing.

In the next sections we are going to describe the different architectures that are being proposed to provide job self-management.

We are going to devote a section to Service Level Agreements (SLA) since this is evolving as the future trend to guarantee a Quality of Service in Grid environments. Under this section we will explain mostly the SLA standardized by the OGF and other specific examples where SLAs are used.

5.1 Introduction

Grid computing was born due to the conjunction of several facts: the existence of computing centers with resources distributed around the world, the proliferation of personal computers with a high percentage of idle time that were wasting computing cycles, and the sharing philosophy in the research world, predisposed to share knowledge that now wanted to share not only huge amounts of data but also computing resources.

Since then, Grid computing has been evolving as a promising distributed computing environment. Those behind this flourishing are the organizations that are trying to define standards (like the Open Grid Forum), the academic world addressing its research towards this field and the business that have developed or adapted commercial applications for the Grid.

Because of the importance of the standardization efforts in the evolution and adoption of Grid computing, we will devote a section of this chapter to explain and compare the several projects that try or tried to define standards for Grid, specially for the user level API's and Job Management.

The field of Grid computing has opened a lot of research areas that go from network topics, to security issues, including job scheduling or distributed data management and many others. In this chapter we will approach only one of these topics, the problem of Job Management under Grid environments.

The matter of Job Management is not a topic that was born with Grid computing, its origins come from cluster computing where jobs need to be scheduled or monitored when running in the clusters. But while in cluster computing the research efforts were mainly dedicated

to design scheduling algorithms to obtain the most profit from the computing power of clusters, within Grid computing Job Management is a topic that has gained a lot of importance. The increase in the relevance of Job Management can be explained by the increase in the number of functions it is responsible of.

The responsibilities of a Job Manager can be summarized in one sentence: To accompany the job since it is created until it dies. To be a bit more explicit, its main functions include: to schedule, monitor, migrate and control the job during its life cycle. If we consider that the number of jobs running on a Grid can not be limited, that means we have to approximate it to infinite because if we try to limit the number of jobs that can exist in a Grid environment sooner or later this limit will be overstepped and the Job Manager will get outdated. Therefor, the Job Manager is not a trivial service and the details of its design, architecture, implementation and other challenges need to be studied carefully.

The most evident properties that are desirable for a Job Manager are scalability, transparency, good performance and last but not least not to overload the system, since we want to spend the computational resources on executing a lot of jobs and not wasting them into managing a few of them. To achieve this becomes very important to choose a suitable architecture when designing a Job Manager for a Grid environment.

If there is something that the community working in distributed computing has for certain is that centralized services are not scalable. From this knowledge the autonomic computing approach is gaining ground, and why not to apply it to the Job Management field too? From the sum of these two concepts we get the Job self-management.

The main distinctive idea of the Job Self-Management is that its goal will be always to obtain the best for the Job, unlike other managers that are designed to obtain the most of the system, or what is the same to use as much efficiently as possible the computing resources available but sometimes at the expense of the performance of some Jobs.

An other concept that was not present in cluster computing and raises with Grid computing is the Quality of Service when executing a Job, how to ensure it, and how to evaluate if it is achieved or not. The answer to these questions is three letters: SLA, or what is the same

Service Level Agreements.

Service Level Agreements have produced a lot of literature the last years, the reason of its popularity can be explained by the change in the way computer resources are accessed. In the past, computer resources where owned by a company, a research center or a private person. Nowadays with the expansion of Grid, computing resources are owned by big companies, or computing centers, and anybody can hire computing services from them, what is more, most of the time paying or with some kind of previous agreement. In this scenario is easy to see the importance of the existence of a formal agreement, standardized, flexible and robust to be used by the parts that are negotiating with computing services.

5.2 User Level API's and its Standardization efforts

If we look for a unified definition of Grid computing probably we will not find one in which everybody agrees, but there is a concept that appears in all the definitions proposed: *heterogeneous resources*. This is the most attractive attribute of Grid computing and at the same time when trying to enable Grid computing for everyone it is the most important problem, or shall we say the most interesting challenge?

This is the reason why so many efforts have been dedicated to define an Application Programming Interface (API) for Grid environments. In Figure 5.1 a very general Grid architecture is shown so that the API and the other elements can be easily situated and identified.

There is a large number of projects and products that try to solve the problem of providing a user friendly API that enables jobs to easily access the Grid. From these projects one can find a wide variety regarding the field, the development stage or even the purpose.

A lot of these projects originate from the Open Grid Forum [17], a community that includes professionals of both industry and research areas, whose main goal is to enable Grid technology for research and business environments. They focus their effort mainly in developing open standards and specifications for Grid Computing. OGF comes from the fusion of two older Grid-related organizations: the Global Grid Forum (GGF) and the Enterprise Grid Alliance (EGA).

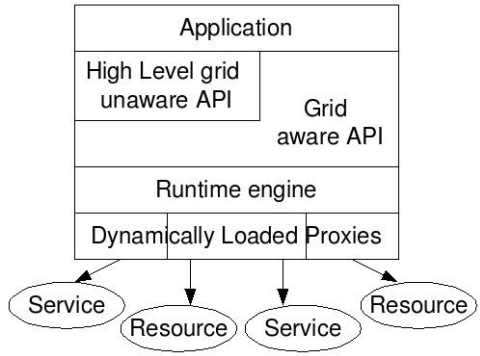

Figure 5.1: General Grid Architecture

The OGF deals with a huge number of topics, and is organized in research or working groups, two of these groups are of interest in this section: the Distributed Resource Management Application API Work Group (DRMAA-WG) [23] [28] and the Simple API for Grid Applications Research Group (SAGA-RG)[22].

The main goal of DRMAA-WG is to develop the specification of an API to enable job submission and job monitoring in a distributed resource management (DRM) environment. The extent of the specification is the high level functionality necessary for an application or user to manage jobs submitted to a DRM. They offer a very basic set of operations to create, monitor, control (start, stop, restart, kill) and retrieve the status of a job.

The SAGA group is close to DRMAA one, in the sense that they share the same objective, but differ in the way to achieve it. The SAGA objective is more ambitious and somehow it is built on the DRMAA experience. They aim to develop a much more flexible and complex API than DRMAA. Besides this, the SAGA specification is still being discussed, and the DRMAA's was finished in 2004.

Although the SAGA specification is still being discussed, there are several projects that support it, the Grid Application Toolkit (GAT) [2] is probably the most important one. GAT is a layer from the GridLab [3] [30] project. GridLab is an European project that has

developed an environment to enable developers to exploit all the possibilities and power of the Grid. It is divided in different layers. Among them the Grid Application Toolkit (GAT) is the layer that provides access to the different Grid Services, as can be seen in Figure 5.2. The main properties of the GAT API are the ease of use, and its malleability, since it supports different programming languages and Grid middlewares. It also enables the same application to run in a variety of systems (from a laptop to a HPC resource).

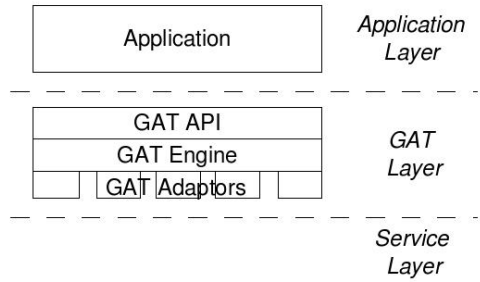

Figure 5.2: Layers that form the GAT API

A part from GAT there is an other project that has supported SAGA, the Commodity Grid Toolkit (CoG) [34] that is also part of the Globus Alliance. The goal of this project is to enable commodity technologies for the Grid so that applications can have both the advantages of being developed in a commodity environment and Grid services, as shown in Figure 5.3. Nowadays you can find the Java CoG Kit, Python CoG Kit and Perl CoG Kit [25] in the web site of the Globus Alliance.

All the projects we have been talking until now provide API's for Grid-aware applications, which means that the applications must be modified or already implemented to be executed on the Grid. But there are other levels of API's that aim to provide access to the Grid for Grid-unaware applications, that is that unmodified applications can be executed on the Grid. In the schema shown in Figure 5.4 is represented the difference between the two kinds of API's for Grid.

One of the projects that offer an API for Grid-unaware applications

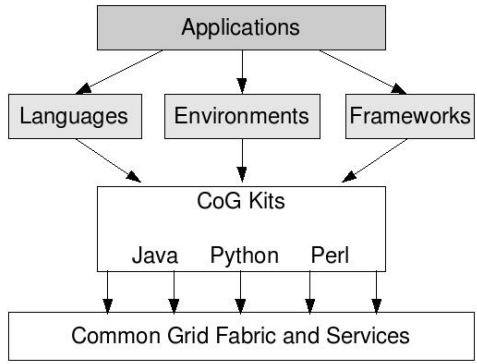

Figure 5.3: Overview of CoG Kits situation in the Grid

is Ibis [33], a Grid programming environment based in Java, and more specifically one of the components of Ibis: the Ibis Portability Layer (IPL). Ibis provides portability among a wide range of Grid platforms thanks to java's main property; furthermore it defines a high level communication API that hides Grid properties from the application and at the same time fit's in the java's object model.

In this category of projects, whose effort is to enable Grid-unaware applications to run on the Grid, can be found also Grid Superscalar

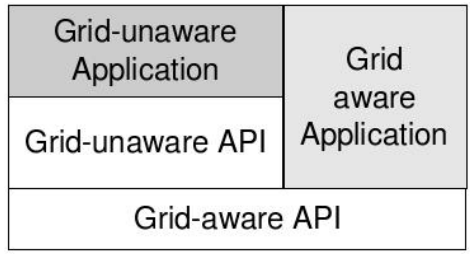

Figure 5.4: Grid-aware and Grid-unaware APIs

[7] and ProActive [8]. Grid Superscalar is more a programming environment to enable easily Grid-unaware applications to run in the Grid than a simple API. Proactive is a Java library that provides a simple API to isolate the underlying framework, which can be a distributed computing on a LAN, on a cluster of PCs, or on Internet Grids.

To end with there are two names that must appear when talking about Grid at any level, Globus [4] [18] and Condor [12], they are not literally a user-level API but an environment that embraces all the layers of the Grid picture.

Globus has been the reference product when talking about solutions for the Grid. It is a complete open source software that covers all the needs when working in a Grid environment. Because of its early appearance it has emerged as the standard de facto. The Globus Toolkit consists of a number of components that can be used together or separately combined with others to provide solutions to a wide range of contexts. The main potential of the Globus Toolkit is that its components are decoupled enough to offer their functionalities individually but at the same time Globus itself offers a complete independent product. As we will explain in Section 5.3 Globus presents a layered architecture hence each layer provides it's own API, as can be seen in Figure 5.5.

Condor....

In this sections we have presented different API's that are applied at different levels, an important concept to have in mind is that as each one is addressed to a different degree they are not incompatible. Moreover is usual to find several of this paradigms working together, as an example we can find that Grid Superscalar works on top of SAGA API, and at the same time SAGA is implemented by GAT [31].

5.3 Job Management Architectures

There are several characteristics of Grid Computing that have to be taken into account when developing any kind of software tailored to this environment such as heterogeneity, transparency, scalability, security.

To ensure all these characteristics, or at least to try to achieve the most of them it is very important to define the architecture that will

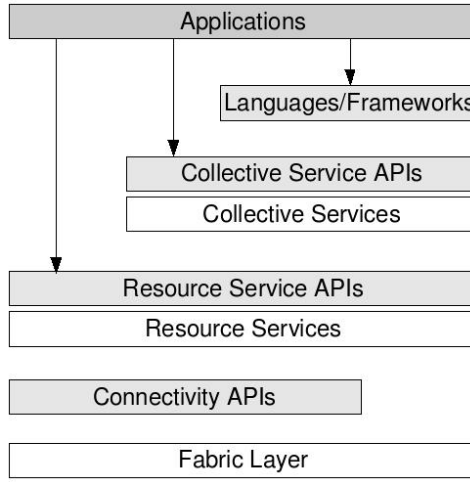

Figure 5.5: API's in the Globus Layered model

be used from the beginning, as on this decision depends the success in the achievement of the desired properties.

The first as can be no other is Globus [18] a complete Grid product that covers all the areas of a Grid environment. It has a modular architecture that lets use its different components separately combined with other pieces of software or together to obtain a complete framework for Grid environments.

The components of the Globus Toolkit are organized following the Layered Grid Architecture, in Figure 5.6 is shown an overview of the Globus layered architecture with an analogy to the Internet Architecture, followed by a brief explanation of each layer.

The Fabric Layer comprises the resources that are administrated by the Grid, computing, network or storage resources.

The connectivity Layer provides the communication protocols and handles the security issues.

The Resource layer is build on top of the connectivity layer and

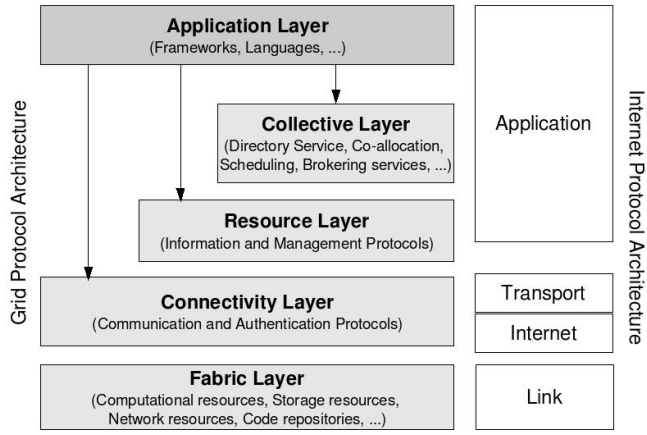

Figure 5.6: Architecture of the Globus Toolkit Layered model

defines the protocols and APIs to access the resources.

The Collective layer manages the services that are not associated to a single resource but are global, distributed or collective.

The Applications layer lies on top of all the others it comprises the user applications; these applications call the services of the other layers through the defined APIs.

Each of these layers is formed by three elements, the API and SDK, the protocols and the implementation and follow the principles of the *Hourglass Model* [19]. The hourglass model is represented in Figure 5.7 and is based in the IP hourglass model that represents a variety of applications (on top), a single protocol (IP, in the middle) and a wide range of platforms (on the bottom).

Grid Resources Allocation and Management (GRAM) is the component of Globus that belongs to the Resource Layer and provides an interface to submit, monitor and cancel jobs. It is not a scheduler but an interface to provide access to a different range of schedulers such as: PBS, Condor, LSF or LoadLeveler. The internal architecture of GRAM is shown in Figure 5.8, it is formed by three tiers; the client

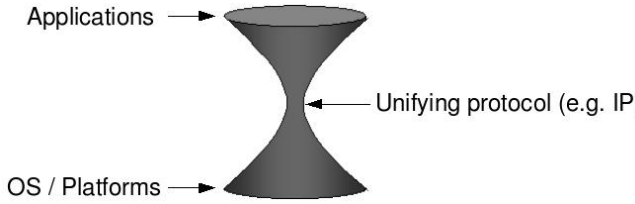

Figure 5.7: The Hourglass Model

tier is from where a client can submit a job to GRAM and check its status.

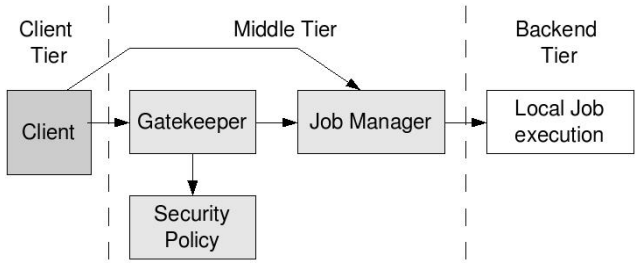

Figure 5.8: Internal Architecture of GRAM

Internally, GRAM consists of a gatekeeper and a job manger. The gatekeeper is responsible for authentication with the client. After this initial security check, it starts up a job manager that interacts thereafter with the client based on the GRAM protocol. Each job submitted by a client to the same GRAM will start its own job manager. Once the job manager is activated, it handles the communication between the client and the back-end system on which the job is executed.

The second in order of importance is Condor [32] a project that aims to develop a management system to support high-throughput Computing (HTC) in distributed environments. Users submit their serial or parallel jobs to Condor, Condor places them into a queue, chooses when and where to run the jobs based upon a policy, care-

fully monitors their progress, and ultimately informs the user upon completion.

From the union of these two giants appears Condor-G [21] a distributed computing framework. It gets the Gird experience of Globus, and the management of distributed computing from Condor. In Figure 5.9 is shown the architecture of Condor-G, The submission machine is where the user submits the Job, here the Condor-G scheduler sends the petition to the Grid Manager that uses GASS (Global Access to Secondary Storage) component of Globus, that provides a transparent remote execution of jobs without the user taking care of redirecting the I/O and copying the executable.

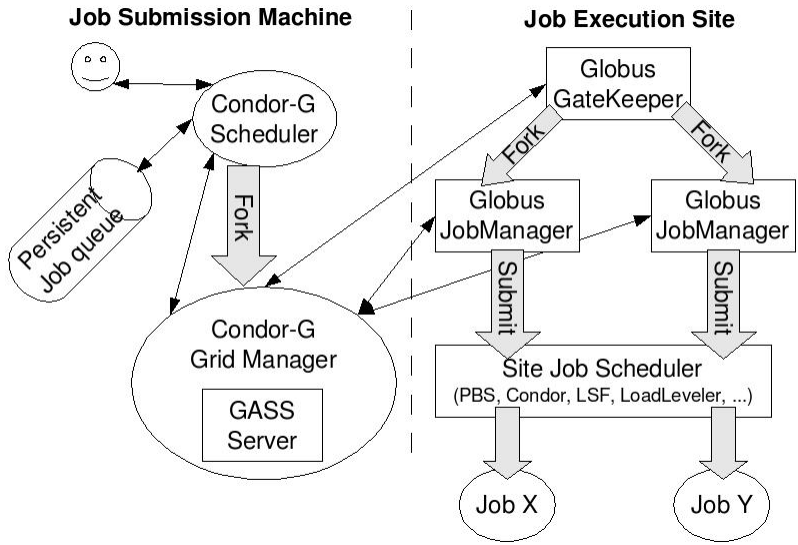

Figure 5.9: Condor-G Architecture

When the scheduling decision is taken the job is submitted to the execution site using GRAM that will be in charge of monitoring and possible needed recoveries of the job. All the security issues are handled by GSI (Grid Security Infrastructure of Globus)

With the Portable Batch Scheduler Professional Edition (PBS Pro)

[27] we can find also a fully featured software for job management in Grid. PBS includes novel approaches to resource management, such as the extraction of scheduling policy into a single separable, completely customizable module. The PBS allows the implementation of policies for the resources sites, such as what types of resources to use or how much a resource can be used by a job. It also provides advanced reservations at user level, which means that a user can request a reservation for a specific start time and duration. The interaction between the components of the PBS is a client-server model.

Application Level Scheduler better know by AppLeS [9] is a project that develops a methodology for adaptive application scheduling. This scheduler is targeted to multi-user distributed heterogeneous environments (like a Grid). Each application is scheduled looking for its better performance. In Figure 5.10 is shown the methodology followed by AppLeS to schedule a job.

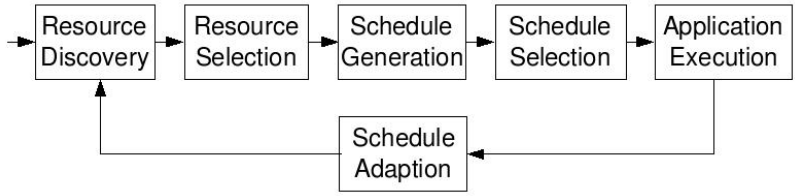

Figure 5.10: Scheduling process in AppLeS

An AppLeS agent is organized in terms of four subsystems and a single active agent called the Coordinator. The four subsystems are the Resource Selector, which chooses and filters different resource combinations for the application's execution, the Planner, which generates a resource-dependent schedule for a given resource combination, the Performance Estimator, which generates a performance estimate for candidate schedules according to the user's performance metric, and the Actuator, which implements the best schedule on the target resource management system(s).

Figure 5.11 depicts the Coordinator and these four subsystems. The information Pool contains Application-specific, system-specific,

and dynamic information used by the subsystems to take decisions.

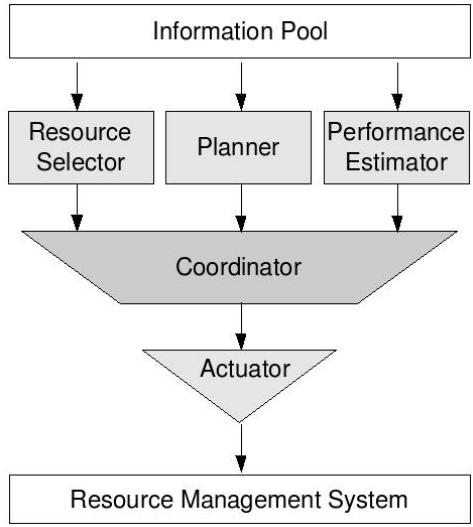

Figure 5.11: AppLeS Agent Architecture

Apples Parameter Sweep Template (APST) [11] is targeted to Parameter Sweep Applications [1], that are an ideal class applications for the Grid.

Parameter Sweep Applications are independent but usually they share big input Files and produce big output files; these are the main characteristics to be taken into account when scheduling these kind of applications. The user/application provides the information relative to the job to the AppLeS agent, from the combination of this information and the current system state the job will be scheduled.

The architecture of APST is shown in Figure 5.12 at the bottom of the picture are the resources available on the Grid, that can be accessed

[1]Parameter sweep applications are a class of application in which the same code is run multiple times using unique sets of input parameter values. This includes varying one parameter over a range of values or varying multiple parameters over a large multidimensional space

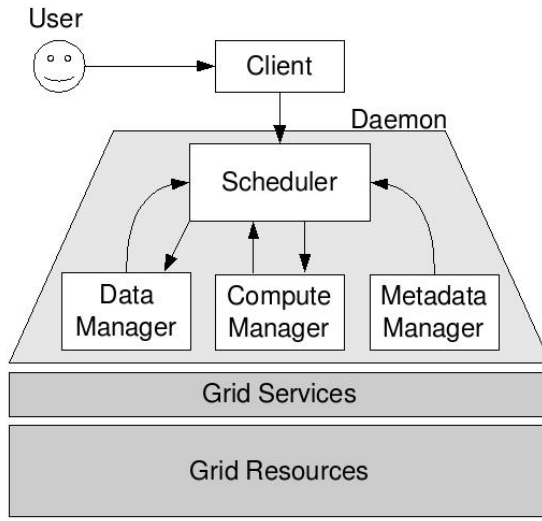

Figure 5.12: APST Architecture [10]

via the Grid services. The scheduler is the central component which takes all the decisions of resource allocation, the data manager and the compute manager help the scheduler by providing information that they obtain from the Grid services. The metadata manager facilitates the scheduler published information about available resources. The scheduler also has a predictor that compiles information from those three sources and computes forecasts.

Under the GRIA project [16] an architecture for Grid have been proposed [26], the differentiation of this architecture is that they aim to provide QoS, to enable a commercial Grid. The architecture proposed can be seen in Figure 5.13, it extends the Globus architecture to provide QoS aspects in the resource management model.

GridLab is an European project whose primary aim is to provide users and application developers with a simple and robust environment enabling them to produce applications that can exploit the full power and possibilities of the Grid [3]. As can be seen in Figure 5.14

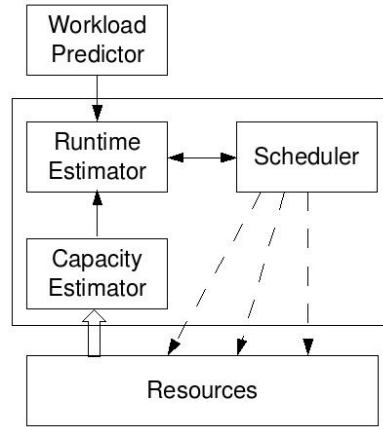

Figure 5.13: GRIA project architecture for QoS

GridLab components can be divided in layers, on the highest layer (called User Space) there is GAT (Grid Application Toolkit), GAT is a high level API for Grid Environments. The middleware layer is called Capability Space and contains the Service layer where are located the Grid services such as GRMS (Grid Resource Management Service)[24], Data Access and Management (Grid Services for data management and access), GAS (Grid Authorization Service), iGrid (GridLab Information Services), Delphoi (Grid Network Monitoring and Performance Prediction Service), Visualization (Grid Data and Visualization Services).

5.4 Service Level Agreements (SLA)

A common requirement in distributed computing systems such as Grids is to negotiate access to, and manage, resources that exist within different administrative domains than the requester. Acquiring access to these remote resources is complicated by the competing needs of the client and the resource owner.

The clients want to know what is happening in the resources and

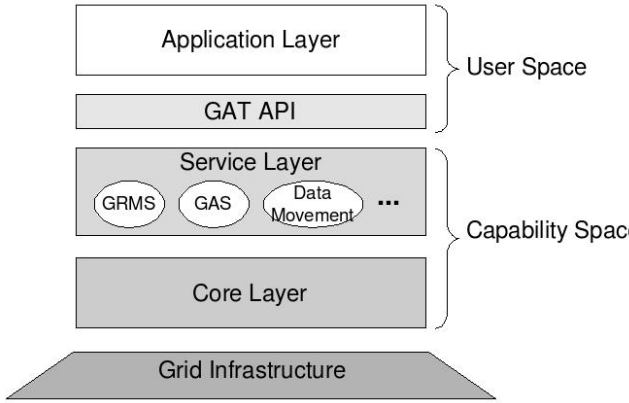

Figure 5.14: GridLab project architecture

to be sure of the level and type of service offered by the resource. The owner wants to keep the control of the resource and its usage.

From these requirements the Service Level Agreements (SLAs) are emerging as the standard concept by which work on the Grid can be arranged and Quality of Service ensured.

The following examples capture some of the diverse resource management situations that can arise where a SLA would be needed:

Task submission in which the resource accepts responsibility to perform a specified task, for example, execute a program, move a file, or perform a database lookup. This is the most basic type of resource management agreement, in which the provider simply commits to perform the agreed-upon function without necessarily committing to when the task will start and finish, how many additional tasks the resource would be able to take on for the user, how many other tasks it might take on in the future, and so forth.

Workload management in which the task submission scenario described above is extended by provisioning tasks to provide a specified level of capability, such as processors on a computer,

threads or memory in a server, bandwidth on a network, or disk space on a storage system. This extension enables the application manager to control not only what task will be done but also aspects of how tasks are performed. Levels of capability might be expressed as maximum task turnaround time, average turnaround time, task throughput, and so forth. This case can relate one manager interaction to a set of application or resource interactions.

Advanced reservations in which resource capability is made available at a specified point in time, and for a specified duration. This type of resource management can be particularly important for so called on line applications, such as teleoperation.

Coscheduling in which a set of resources is made available simultaneously by coordinating advanced reservation agreements across the required resources.

Resource brokering scenarios in which a broker service acts as an intermediary to a set of resource capabilities and directs tasks to appropriate resources based on broker specific policy. One such policy is to maximize total job throughput.

Grid-based resource management systems generally cannot create quality of-service agreements without cooperation from the resource being managed. The reason is that a resource is typically not dedicated to a specific user community, or virtual organization (VO).

Based on the Service Negotiation and Access Protocol (SNAP) [15] the Service Level Agreements can be classified in three categories:

Task service-level agreements (TSLAs) that represent a commitment to perform an activity or task with embedded resource requirements. For example, a TSLA is created by submitting a job description to a queuing system. The TSLA characterizes a task in terms of its service steps and resource requirements.

Resource service-level agreements (RSLAs) that represent a commitment to provide a resource when claimed by a subsequent SLA. An RSLA might be negotiated without specifying the activity for which the resource will be used. For example, an advance

reservation takes the form of an RSLA. The RSLA characterizes a resource in terms of its abstract service capabilities.

Binding service-level agreements (BSLAs) that represent a commitment to apply a resource to an existing task. For example, an RSLA promising network bandwidth might be applied to a particular TCP socket, or a RSLA promising parallel computer nodes might be applied to a particular job task. The BSLA associates a task, defined by its TSLA with the RSLA and the resource capabilities that should be met by exploiting the RSLA.

5.4.1 Standardization

The power of agreement based resource management lies in its ability to enable resources and users from different administrative domains to combine resources in such a way as to provide well defined behaviors. As such, agreements are fundamental to the Grid vision of delivering virtualized services to a collaboration that spans organizational boundaries. However, without well-defined, interoperable protocols for negotiating, establishing, and managing agreements, the ability to visualize resources across organizations will be greatly impeded. To this end, the Grid Resource Allocation and Agreement Protocol Working Group (GRAAP-WG) in the Global Grid Forum (GGF) [5] is defining a standard set of agreement protocols.

The GGF structure considers a three-layered agreement model consisting of the following.

- A **service layer** that provides domain-dependent interfaces to that actual function of the service. The service layer has the implementation-specific mechanisms for enforcing the terms of an agreement.
- An **agreement layer** that provides management (i.e. creation and destruction) and status monitoring of agreements.
- A **negotiation layer** which implements a term negotiation protocol that provides for the exchange of agreement terms.

The GRAAP-WG of the OGF is working on the definition of a standard Web Services Agreement (WS-Agreement) [6], to address

the agreement layer. The first version was presented in 2004. A WS-Agreement is a XML-based document containing descriptions of the functional and non-functional properties of a service oriented application. Its structure can be seen in Figure 5.15. Specifications for other layers will be defined in the future.

Figure 5.15: Web Service Agreement (ws-agreement) Structure

According to the first specification of the WS-Agreement the states of an agreement can be *Pending*, *Pending And Terminating*, *Observed*, *Observed And Terminating*, *Rejected*, *Complete* and *Terminated* and the transitions between them are shown in Figure 5.16.

There have been proposed extensions to the ws-Agreement, like the one proposed from the university of Trento [1]. Their proposal can be divided in two. The first idea consists in anticipating violations, while the second is devoted to the run-time renegotiation. They base their study in a formal analysis of the Agreements by a finite state automata, and they provide a set of rules that tie together the agreement terms and the life-cycle of an agreement.

Their proposal consists in adding some terms to the WS-Agreement that contains the renegotiation possibilities. Providing renegotiation permits that in case an Agreement has not been accomplished instead

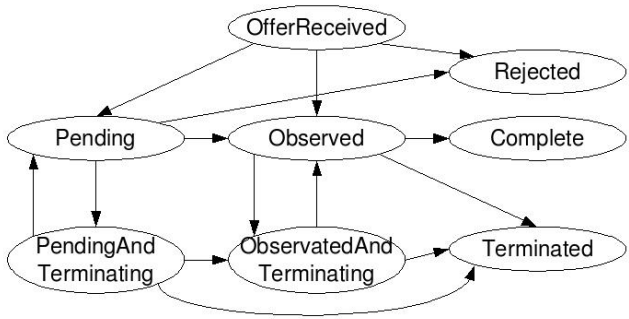

Figure 5.16: Web Service Agreement (ws-agreement) States

of aborting it, and the need of creating a new one, the Agreement can be renegotiated.

In a later article [20] they expand their proposal to anticipate violations, and for this they add a new state of *Warning* when an agreement is close to being violated, then renegotiation can be done.

5.4.2 SLA's in use

Service Negotiation and Acquisition Protocol (SNAP) [15] is a model and protocol for managing the process of negotiating access to, and use of, resources in a distributed system. Defines a general framework within which reservation, acquisition, task submission, and binding of tasks to resources can be expressed for any resource in a uniform fashion it is not focused in a particular type of resource (e.g. CPU's, Networks).

The states of an Agreement in the SNAP protocol are shown in Figure 5.17. Mainly there are four states the initial one (S1) when the resources are being requested and the Agreement negotiated. When the task has been assigned to a resource it changes to the scheduled state (S2). As soon as the resource is being utilized the state changes to S3, and finally when the resource is released either by successful termination or any other reason, the state is S4.

The EPSRC project Service Level Agreement Based Scheduling

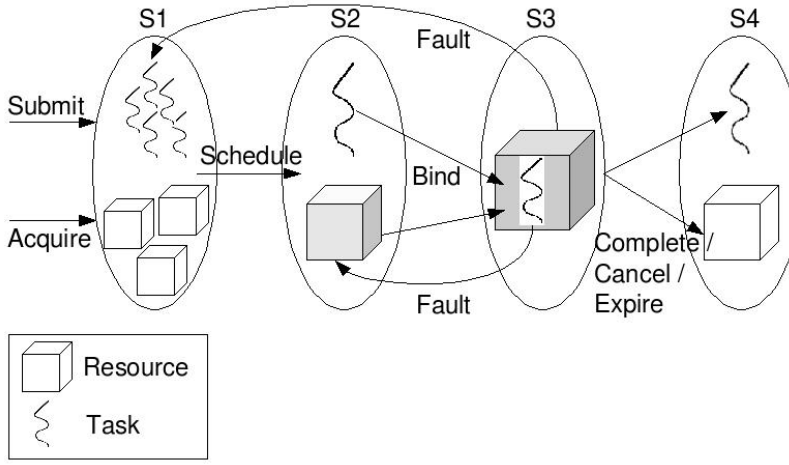

Figure 5.17: States of an Agreement in the SNAP protocol

Heuristics have a wide work on the field, they have proposed an extension to WS-Agreements to include functions in the terms of the Agreement rather than constant values or ranges to provide more flexibility, that will result in less violated Agreements and a better performance of the negotiation process [29].

The research they are working on aims to provide a flexible and efficient scheduling infrastructure based in Service Level Agreements, they position their work opposite the advanced reservations because these ones are too rigid. The architecture they proposed is formed by a centralized coordinator that receives all the petitions and negotiate the SLA with the resources[35] .

In the University College London they propose SLAng, A Language for Defining Service Level Agreements. SLAng is a language for defining QoS properties in XML. It is not Grid tailored and is based in the definition of two levels of Agreements, horizontal and vertical. Horizontal Agreements are those binding different parties providing the same kind of service and Vertical Agreements are contracted between parties that ones is above the other infrastructure.

The requirements they aim to achieve by the definition of SLAng are the following:

Parametrization Each SLA includes a set of parameters; these parameters describe quantitatively a service that have been stated previously. A set of parameters of a particular kind of SLA provides a qualitative description of a service.

Compositionality Service providers should be able to compose SLAs in order to issue offers of services that can be aggregated or cascaded. An SLA language has to enable composition of services.

Validation An SLA must be validated before being accepted in terms of its syntax and consistency. Furthermore, validity should be verified as a result of a composition.

Monitoring Ideally, parties should be able to automatically monitor the terms of the Agreement, SLAs should therefore provide the basis for the derivation and installation of automated monitors that report extents with which service levels are being met.

Enforcement Once service levels are agreed, network routers, database management systems, middleware and web servers can be extended to enforce service levels in an automated manner by using techniques such as caching, replication, clustering and farming.

SLAng defines that an SLA to be legally binding has to be embedded in a SLA contract. A SLA contract is formed by one or more SLAs plus the names of the two juridical persons contracting the agreement, and additionally a third trusted party, with all the digital signatures. An SLA is formed by three parts, the Endpoint description contains the information of the service providers and consumers, the contractual statements that are not the terms referents to the requested service but to the agreement itself and finally in the SLS can be found the Terms of the Agreement. The structure of a SLA embedded in a SLA contract is shown in Figure 5.18.

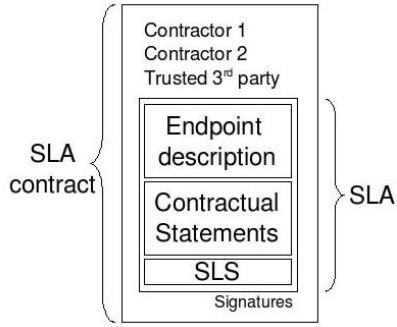

Figure 5.18: A SLAng contract Structure

5.5 Conclusions and Future Trends

As we have seen in the previous sections, there are a lot of projects that put research efforts in the Job Management for Grid Computing. The level of maturity they have is very heterogeneous, some are a finished product and other ones are just being specified. Therefor we can say that Grid is not the future nor the present of Distributed computing, it is both things at the same time the present and the future.

To understand the point where Grid computing is now, we can look for the similarity with the Internet: it was born in a lab, later on it was expanded all over the world and anyone could have access to it just with a modem and a personal computer, but it took some years since it was accessible until it became popular and almost anyone access it every day. Grid Computing is nowadays a reality, anyone can have access to it but it is mostly known in the academic domain. In the next years Grid computing will gain popularity, and if the trend continues not in a far future Grid computing will be as popular as Internet is these days.

But for the Grid Computing to achieve this level of maturity there is still work to do. Corresponds to the business world to enable applications for the Grid and give it a chance by supporting its expansion. On the other hand there is a lot of research to be done in the Grid

Computing area, the academic world is responsible for enabling the Grid to be accessible to everybody, to become reliable, and of course rentable in terms of performance.

But among all these, maybe the most important task that should be done is standardization. The standardization process is not always done by dedicated organizations that impose a standard. Sometimes is simply that everybody comes together to use the same, when this happens is because whatever it is, it has the characteristics that were demanded in that moment. In this case it is still the responsibility of the standardizing organizations to formalize it and spread it use.

In any of these cases the standardization organizations have an important role in the expansion of Grid computing and its popularity.

Focusing in Job Management in Grid Computing, the topic of this chapter, we have devoted the different sections to the areas that in our opinion are decisive for the evolution of Grid computing from the research labs to the public domain. First the Application Programming Interfaces at user level, because as soon as there is a strong, reliable and standardized API, more applications could be enabled for the Grid and it will be easier for programmers to develop Grid-aware applications. As we have seen a lot of effort has been put in this direction, but the proposed APIs are not enough wide to cover all the needs and are still not enough flexible, it is the standardization organizations duty to propose an API with all these properties. But not only to propose it, furthermore they have to listen to the community, and adapt the proposal to their needs and suggestions. Beyond this only time can settle down and positively test a standard.

Of course the architecture of Grid environments is decisive to develop a reliable, profitable and flexible Grid framework. Unlike it can seem, the goal here is not to achieve a standard and all the solutions to converge into the same approach. But to offer a variety of solutions that can cover all the different needs. Although at first one can think that everywhere the requirements of a Grid framework are the same, depending on the environment one or another gain more importance. The academic world is the responsible for researching in the different options and evaluate for which environment is better each one. Offering a wide range of approaches so that all the different needs can be satisfied.

In order to obtain a place in the business world is essential for

the Grid to ensure a Quality of Service. Furthermore empowering the business applications for the Grid is the only way to give it significance and force. The QoS in Grid computing is translated to SLA, as this is the instrument to represent the requirements and agreements about the services provided. Even though there is a lot of literature about SLA written already, a further effort have to be made to converge to a unique model. Again the standardization process is needed here. But not only is necessary a definition of the contents of a SLA also to specify a protocol of negotiation. As explained in Section 5.3 and Figure 5.7 the hourglass model would be the ideal to achieve here.

Concluding, Grid Computing is already a reality and anyone can access it with the products that exist nowadays. Despite this a lot of work should be done by the researching community and the business parties to make it become a common environment for the general public. In some areas the need is to converge to a unique solution and in some others to offer a wide range of possibilities. But the key aspect is that the interest in Grid is not lost, and looking at the amount of literature that is being produced about it there is no danger at all. Beyond this the natural process of research and evolution will drive to the maturity of Grid Computing.

Bibliography

[1] Marco Aiello, Ganna Frankova, and Daniela Malfatti. What's in an agreement? an analysis and an extension of ws-agreement. In *International Conference on Service Oriented Computing (ICSOC)*, pages 424–436, 2005.

[2] Allen, G., K. Davis, T. Goodale, A. Hutanu, H. Kaiser, T. Kielmann, A. Merzky, R. van Nieuwpoort, A. Reinefeld, F. Schintke, T. Schott, E. Seidel, and B. Ullmer. The grid application toolkit: toward generic and easy application programming interfaces for the grid. *Proceedings of the IEEE*, Vol. 93(Issue 3):pages: 534–550, March 2005.

[3] Gabrielle Allen, Kelly Davis, Konstantinos N. Dolkas, Nikolaos D. Doulamis, Tom Goodale, Thilo Kielmann, André Merzky, Jarek Nabrzyski, Juliusz Pukacki, Thomas Radke, Michael Russell, Ed Seidel, John Shalf, and Ian Taylor. Enabling applications on the grid: A gridlab overview. *International Journal of High Performance Computing Applications: Special Issue on Grid Computing: Infrastructure and Applications*, Vol. 17(Issue 4):pages: 449–466, November 2003.

[4] The Globus Alliance. http://www.globus.org/.

[5] The Grid Resource Allocation and Agreement Protocol Working Group (GRAAP-WG). Global Grid Forum. https://forge.gridforum.org/projects/graap-wg.

[6] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu. Web services agree-

ment specification (ws-agreement). Technical report, Global Grid Forum, May 2004.

[7] R. M. Badia, JesÃºs Labarta, RaÃºl Sirvent, J. M. Cela, and Rogeli Grima. Grid superscalar: a programming paradigm for grid applications. In *Workshop on Grid Applications and Programming Tools (GGF8)*, June 2003.

[8] F. Baude, D. Caromel, F. Huet, L. Mestre, and J. Vayssiere. Interactive and descriptor-based deployment of object-oriented grid applications. *In The Eleventh IEEE International Symposium on High Performance Distributed Computing (HPDC-11)*, pages pages: 93–102, July 2002.

[9] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, S. Spring, A. Su, and D. Zagorodnov. Adaptive computing on the grid using apples, 2003.

[10] H. Casanova and F. Berman. *Parameter Sweeps on the Grid with APST*. Grid Computing: Making the Global Infrastructure a Reality. John Wiley & Sons, April 2003.

[11] Henri Casanova, Francine Berman, Graziano Obertelli, and Richard Wolski. The apples parameter sweep template: User-level middleware for the grid.

[12] Condor Project High Throughput Computing.

[13] K. Czajkowski, I. Foster, and C. Kesselman. Agreement-based resource management. In *Proceedings of the IEEE*, volume Vol.93, pages Pages: 631– 643, March 2005.

[14] Karl Czajkowski, Ian Foster, Carl Kesselman, and Steven Tuecke. Grid service level agreements: Grid resource management with intermediaries. pages 119–134, 2004.

[15] Karl Czajkowski, Ian T. Foster, Carl Kesselman, Volker Sander, and Steven Tuecke. Snap: A protocol for negotiating service level agreements and coordinating resource management in distributed systems. In *Job Scheduling Strategies for Parallel Processing (JSSPP)*, pages 153–183, 2002.

[16] Service Oriented Collaborations for Industry and Commerce. http://www.gria.org/.

[17] The Open Grid Forum. http://www.ogf.org/.

[18] Ian Foster. Globus toolkit version 4: Software for service-oriented systems. *IFIP International Conference on Network and Parallel Computing*, Springer-Verlag LNCS 3779:pages 2–13, 2006.

[19] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the Grid: Enabling scalable virtual organizations. *Lecture Notes in Computer Science*, 2150, 2001.

[20] Ganna Frankova, Daniela Malfatti, and Marco Aiello. Semantics and extensions of ws-agreement. *Journal of Software*, pages pages: 34–42, 2006.

[21] James Frey, Todd Tannenbaum, Ian Foster, Miron Livny, and Steve Tuecke. Condor-G: A computation management agent for multi-institutional grids. In *Proceedings of the Tenth IEEE Symposium on High Performance Distributed Computing (HPDC)*, pages pages: 7–9, San Francisco, California, August 2001.

[22] T. Goodale, S. Jha, H. Kaiser, T. Kielmann, P. Kleijer, G. von Laszewski, C. Lee, A. Merzky, H. Rajic, and J. Shalf. Saga: A simple api for grid applications - high-level application programming on the grid. *Computational Methods in Science and Technology: special issue "Grid Applications: New Challenges for Computational Methods"*, 2005.

[23] Distributed Resource Management Application API Working Group. http://drmaa.org/wiki.

[24] http://www.gridlab.org/WorkPackages/wp 9/. Grid(lab) resource management.

[25] Globus Toolkit CoG Kits. http://www.globus.org/toolkit/cog.html.

[26] Antonios Litke, Athanasios Panagakis, Anastasios D. Doulamis, Nikolaos D. Doulamis, Theodora A. Varvarigou, and Emmanouel A. Varvarigos. An advanced architecture for a commercial grid infrastructure. In *European Across Grids Conference*, pages 32–41, 2004.

[27] Bill Nitzberg, Jennifer M. Schopf, and James Patton Jones. Pbs pro: Grid computing and scheduling attributes. *Grid resource management: state of the art and future trends*, pages pages: 183–190, 2004.

[28] H. Rajic, R. Brobst, W. Chan, F. Ferstl, J. Gardiner, J.P. Robarts, A. Haas, B. Nitzberg, and J. Tollefsrud. Distributed resource management application api specification 1.0. Technical report, DRMAA Working Group-The Global Grid Forum, 2004.

[29] Rizos Sakellariou and Viktor Yarmolenko. On the flexibility of ws-agreement for job submission. In *MGC '05: Proceedings of the 3rd international workshop on Middleware for grid computing*, pages 1–6, New York, NY, USA, 2005. ACM Press.

[30] Ed Seidel, Gabrielle Allen, Andre Merzky, and Jarek Nabrzyski. Gridlab–a grid application toolkit and testbed. *Future Generation Computer Systems*, Vol. 18(Issue 8):pages: 1143–1153, October 2002.

[31] Raul Sirvent, Andre Merzky, Rosa Badia, and Thilo Kielmann. Grid superscalar and saga: forming a high-level and platform-independent grid programming environment. In *CoreGRID integration workshop. Integrated Research in Grid Computing*, November 2005.

[32] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience*, Vol. 17(Issue 2-4):323–356, 2005.

[33] R. van Nieuwpoort, J. Maassen, G. Wrzesinska, R. Hofman, C. Jacobs, T. Kielmann, and H. Bal. Ibis: a flexible and efficient java-based grid programming environment. *Concurrency and Computation: Practice and Experience*, Vol. 17:pages: 1079–1107, June-July 2005.

[34] Gregor von Laszewski, Ian Foster, and Jarek Gawor. Cog kits: a bridge between commodity distributed computing and high-performance grids. In *JAVA '00: Proceedings of the ACM 2000 conference on Java Grande*, pages pages: 97–106, New York, NY, USA, 2000. ACM Press.

[35] Viktor Yarmolenko, Rizos Sakellariou, Djamila Ouelhadj, and Jonathan M. Garibaldi. Sla based job scheduling: A case study on policies for negotiation with resources. In *Proceedings of the UK e-Science All Hands Meeting (AHM'2005)*, September 2005.

List of Figures