Self-Optimization of Internet Services with Dynamic Resource Provisioning

Christophe Taton, Sara Bouchenak, Noël de Palma, Daniel Hagimont

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Self-Optimization of Internet Services with Dynamic Resource Provisioning

Christophe Taton — Sara Bouchenak — Noël de Palma — Daniel Hagimont

N° 6575

July 2008

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--6575--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
GRENOBLE – RHÔNE-ALPES

Self-Optimization of Internet Services with Dynamic Resource Provisioning

Christophe Taton [*] [†], Sara Bouchenak [‡*] , Noël de Palma [† *] ,
Daniel Hagimont [§*]

Thème COM — Systèmes communicants
Équipes-Projets Sardes

Rapport de recherche n° 6575 — July 2008 — 29 pages

Abstract: Self-optimization through dynamic resource provisioning is an appealing approach to tackle load variation in Internet services. It allows to assign or release resources to/from Internet services according to the varying load. However, dynamic resource provisioning raises several challenges among which: (i) How to plan a good capacity of an Internet service, i.e. a necessary and sufficient amount of resource to handle the Internet service workload, (ii) How to manage both gradual load variation and load peaks in Internet services, (iii) How to prevent system oscillations in presence of potentially concurrent dynamic resource provisioning, and (iv) How to provide generic self-optimization that applies to different Internet services such as e-mail services, streaming servers or e-commerce web systems.

This paper precisely answers these questions. It presents the design principles and implementation details of a self-optimization autonomic manager. It describes the results of an experimental evaluation of the self-optimization manager with a realistic e-commerce multi-tier web application running in a Linux cluster of computers. The experimental results show the usefulness of self-optimization in terms of end-user's perceived performance and system's operational costs, with a negligible overhead.

Key-words: Internet services, Dynamic resource provisioning, Self-optimization, Autonomic computing

[*] Christophe.Taton@inria.fr, Sara.Bouchenak@inria.fr, Noel.Depalma@inria.fr, Daniel.Hagimont@enseeiht.fr
[†] Institut National Polytechnique de Grenoble, France
[‡] University of Grenoble I, Department of Computer Science, Grenoble, France
[§] Institut National Polytechnique de Toulouse, France

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

1 INTRODUCTION

1.1 Context and challenges

A large variety of Internet services exists, ranging from web servers to email servers, streaming media services, enterprise servers, and database systems. These services are usually based on the client-server architecture, where a server provides some online service (such as reading web pages, sending emails or buying the content of a shopping cart), and clients concurrently access that service. Internet services may face a large amount of load in terms of number of clients that concurrently request the service at the same time. To tackle this load, classical techniques such as replication and partitioning are applied to allow the service to scale with its load [39]. For example, in a replicated service, the load is balanced among several replicas of the same service, allowing the service to globally process more client requests than with a single-instance service. In such systems, the amount of resources (i.e. computers) needed by an Internet service depends on the current load the service needs to process. Ideally an Internet service should be assigned the necessary *and* sufficient amount of resources to handle its current load. The necessary amount of resources would allow the Internet service to absorb the load while guaranteeing acceptable performance and Service Level Agreement (SLA) requirements to end-users. On the other hand, restricting Internet service resources to the minimal and sufficient amount allows a saving of resources, and thus reduces powering and cooling costs of the Internet service. The former objective is usually presented as the end-user's point of view, while the latter objective is seen as the system administrator's point of view. Combining these two antagonist criteria in order to plan the ideal capacity of an Internet service is challenging. This trade-off is particularly important in the context of physical platforms consisting of a set of resources (e.g. clusters or grids) that are shared by several Internet services such as in case of ASPs (Application Service Providers).

Moreover since the load of an Internet service may vary over time, a challenging task for an Internet service administrator is to well provision the service with the good (i.e. necessary and sufficient) amount of resources. A first approach, the pessimistic approach, assigns to the service a fixed amount of resources which corresponds to the resources needed by the service in its worst case (i.e. in its highest load scenario). This approach clearly results in resource wasting, because most of the time, the service is provisioned with much more resources than its instantaneous needs. Another approach is based on resource overbooking and may result in situations where the Internet service does not have its necessary amount of resources [59]. Thus, dynamic resource provisioning is an appealing approach to face load variations in Internet services. However it raises several open issues among which dynamic load variation, system oscillations and genericity that we detail in the following.

Load variation. The workload of an Internet service usually varies over time from a light workload to a heavier workload and vice versa. Roughly speaking, in contrast to a light workload, a heavy workload involves a large amount of concurrent client requests and/or requests that require long processing times. This workload variation in an Internet service reflects different client usages at different times. For instance, an e-mail service is likely to face a heavier workload in the morning than in the rest of the day, since people usually consult

their e-mails when arriving at work. Moreover the load of an Internet service may vary at different speeds, in the sense that it may vary gradually or it may have the form of load spikes that occur suddenly. While in the former case the dynamic addition and removal of a single resource at a time may be sufficient to absorb load variations, in the latter case a challenging issue is to determine the amount of resources to be (un-)provisioned before actually performing the (un-)provisioning in a single step. Thus, a challenging issue in Internet service resource provisioning is to efficiently tackle both situations of gradual load variations and load peaks.

System oscillations. If not addressed carefully, dynamic resource provisioning of an Internet service may induce multiple concurrent provisioning operations that are actually not necessary, but only triggered because the system is in a temporarily instable state. This would, for instance, result in first adding more resources than strictly necessary, and then later on removing the unnecessary resources. As a result these oscillations would hurt the overall Internet service performance. This problem is made harder by the fact that two concurrent provisioning operations do not always act on the same parts of an Internet service but may apply to *different* parts of the *same* Internet service. For instance, in a three-tier e-commerce Internet service consisting of a front-end tier of replicated web servers, a middle tier of replicated application servers, and a back-end tier of replicated database servers; the back-end tier may become the bottleneck which results in an under-load of the front-end tier (which simply waits for responses from the back-end tier). In such a situation, a provisioning operation may be triggered on the set of replicas of the back-end tier (because of its over-load) while an un-provisioning operation may be triggered on the set of replicas of the front-end tier. Obviously, the latter un-provisioning operation on the front-end tier is not necessary, and is only triggered because the back-end tier of the Internet service is in an instable state. Thus, one of the issues of Internet service resource provisioning is to prevent system oscillations due to unnecessary (un-)provisioning operations.

Genericity. Dynamic resource provisioning solutions may be tied to a particular Internet service (such as a database server [46]). This is the case when the underlying provisioning mechanism is specific to a particular service and implemented as part of that service. Such an approach makes the provisioning system hard to be directly applied to other Internet services (e.g. web servers or streaming media servers). Thus, generic resource provisioning is an appealing approach to handle the broad diversity of Internet services. However a generic approach to resource provisioning needs to be careful about providing a too general solution that does finally not apply to any specific and realistic Internet service. Thus an important question to address here is the following: is a generic approach to resource provisioning able to capture the specificities of individual Internet services in an efficient way?

1.2 Research contributions

This paper describes our experience in designing, implementing and evaluating self-optimizing Internet services through dynamic resource provisioning. The proposed self-optimization solution aims at allocating *at least* the necessary amount of resources in order for end-users to obtain good performance, and *at most* the sufficient amount of resources to reduce the operating costs of the

system. Thus, to plan the capacity of an Internet service, both criteria (provide the minimal and maximal resource amount) need to be met. We translate these criteria to an objective function based on minimum and maximum thresholds. This function aims at keeping the Internet service between acceptable minimum and maximum resource usage that respectively represent the limits under which the system is sub-optimal and over which the system is over-loaded. The design principles of the proposed dynamic resource provisioning solution are precisely motivated by the above-mentioned issues, that are addressed as follows.

Load variation. In order to face dynamic variation of the load of an Internet service, we propose a self-optimization system that continuously monitors the Internet service resource usage, recalculates its capacity planning and performs dynamic resource (un-)provisioning accordingly. Moreover, in order to tackle both situations of gradual load variations and load spikes, we enriched the capacity planning policy with heuristics that allow to determine the amount of resources that need to be (un-)provisioned according to the Internet service load.

System oscillations. Internet services are usually built as distributed systems that consist of several parts. In this context, load variations and system instabilities in some parts of the Internet service may have temporary side-effects on other parts of the system. Thus, dynamic resource provisioning on different parts of an Internet service should be conducted carefully to prevent system oscillations. For that purpose, we follow an architecture-based approach for self-optimizing Internet services. More precisely, the different parts composing a distributed Internet service are materialized in a view of the system, and the direct and indirect communication and cooperation dependencies between these parts are exhibited. Then, based on this knowledge of the system architecture and in order to prevent system oscillations, concurrent dynamic resource provisioning operations are inhibited if they occur on inter-dependent parts of the same Internet service. For instance, in case of a multi-tier e-commerce Internet service, the front-end tier and the back-end tier are automatically identified as inter-dependent, and thus concurrent provisioning operations on these two parts are automatically inhibited to prevent system oscillations.

Genericity. As discussed earlier, the resource provisioning policy that underlies the proposed self-optimization system is built upon an objective function that is based on minimum and maximum thresholds for resource usage. A resource, as a computer, can be seen as a coarse-grain resource which consists in several finer-grain hardware resources such as CPU, memory, disk or network bandwidth. The proposed resource provisioning policy makes use of this view of finer-grain hardware resources to determine if a coarse grain resource is under-utilized or over-loaded in order to (un-)provision resources. Thus, the proposed policy is based on general hardware resource usage which makes it generic and applicable to any Internet service.

Finally, we implemented a self-optimization autonomic manager as a dynamic resource provisioning system that follows the design principles introduced earlier ; we integrated this self-optimization manager to the Jade autonomic management framework [8]. This helps building self-optimizing Internet services. In this paper, we illustrate the feasibility of the proposed policy and describe its appliance in different use cases ranging from e-mail servers to streaming services and e-commerce web systems. Furthermore, we experimentally evaluated the self-optimization system in a realistic e-commerce multi-tier

web application running in a Linux cluster of computers. The results of the experimental evaluation show the usefulness of self-optimization in terms of end-user's perceived performance and system's operational costs with a negligible overhead.

1.3 Paper roadmap

The remainder of the paper is organized as follows. Section 2 describes the considered system model. Section 3 illustrates the appliance of dynamic resource provisioning to different Internet services. Section 4 describes the design principles and implementation details of the proposed self-optimization autonomic manager through dynamic resource provisioning, and section 5 presents the results of its experimental evaluation. Finally, the related work is discussed in section 6, and section 7 draws our conclusions.

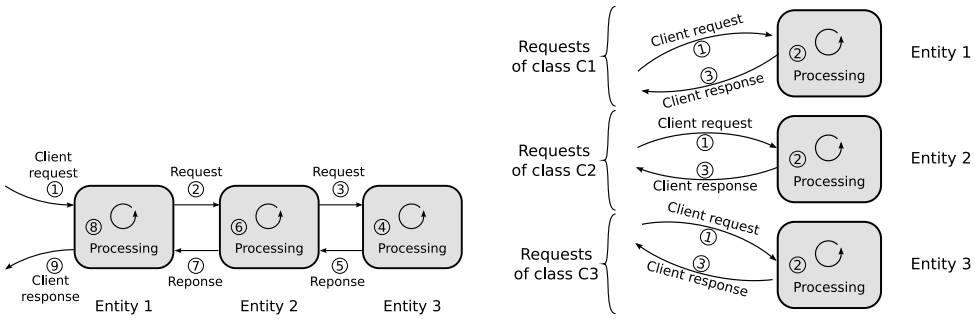
2 SYSTEM MODEL

The term *resource* is used throughout this paper to designate a node, i.e. a computer, a machine. We consider a collection of resources interconnected through a local area network. The resources are homogeneous in the sense that they have the same physical architecture and operating system, as it is typically the case in clusters of computers. A distributed application is a software computing system that runs on a set of resources. An application may dynamically acquire or release resources from a global set of resources (i.e. a cluster): a resource is either free or exclusively used by an application. In other words two applications do not share the same resource at the same time. We consider, in particular, the case of client-server Internet applications where concurrent clients connect to a server which provides them with some online service, e.g. streaming videos in a video-on-demand service, reading e-mails in an e-mail service, etc.

Moreover, the application may be seen as a monolithic entity or as a set of several entities. For instance, an e-mail service may be seen as a monolithic entity represented by the e-mail server. While a three-tier e-commerce application may be seen as a collection of three entities: the web server entity, the application server entity and the database server entity. Here, an application entity is any part of the distributed application that can be hosted by a distinct resource.

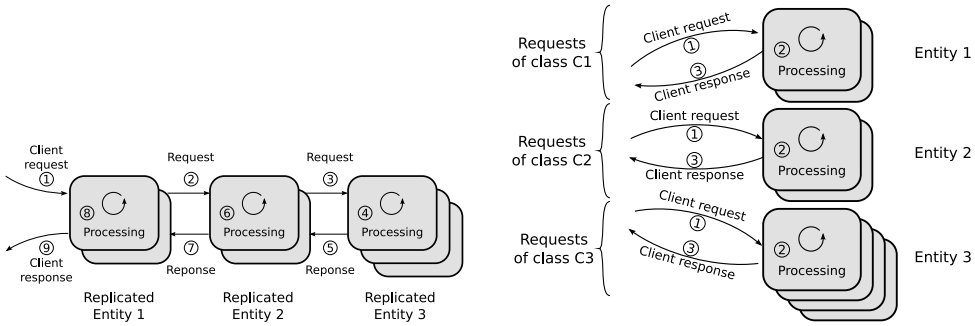
Furthermore, the architecture of a distributed application, i.e. the way its entities are organized may have different forms. We distinguish between pipelined systems and partitioned systems. In a pipelined system, the entities composing the system are organized in series (see Figure 1-a) and each entity may take part to the building of the response to the client. For instance, in a multi-tier web application consisting of three entities (the database server, the application server and the web server), the three entities cooperate in order to produce the overall web client response: the database server performs queries on the database, the enterprise server uses these results to compute the web application logic whose results are then used by the web server in order to be formatted in HTML pages. On the contrary, in a partitioned system, the entities composing the system are organized in parallel (see Figure 1-b) and do not interact with each other for client requests processing. For instance, in case of an application which consists of N different entities (i.e. servers), each one being responsible of processing requests of a particular class C_i ($1 \leqslant i \leqslant N$), client requests are spread among the different entities depending on their classes, and each client request is processed by a unique entity. Internet services may be a combination of several pipelined and partitioned systems.

Finally for scalability purposes, in order to process more client requests, application entities may be replicated on several resource instances (see Figure 2). This is coupled with a load balancer which distributes the load among replicas. Different load balancing algorithms may be used (see [37, 16, 10]). In the following we consider a load balancer that equally balances the load (in terms of resource consumption) between the replicas.



(a) Pipelined systems (b) Partitioned systems

Figure 1: Application organization – pipelined versus partitioned systems



(a) Pipelined systems with replication (b) Partitioned systems with replication

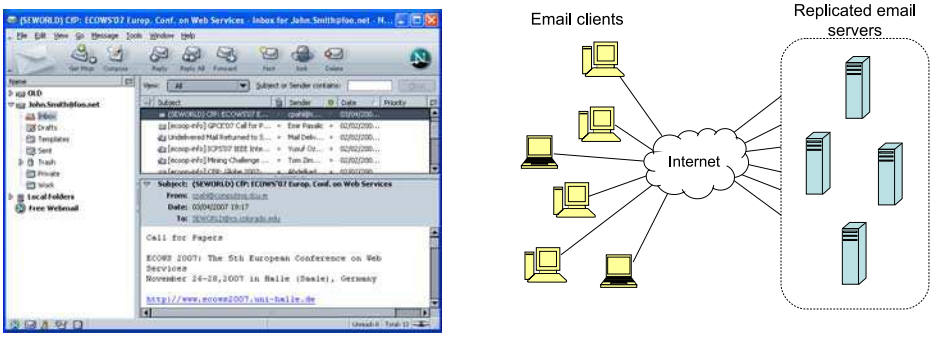
Figure 2: Application organization with replication

3 USE CASES

3.1 E-mail server

An e-mail service is an online service that is used for sending and receiving digital posts via the Internet. An e-mail server follows the classical client-server architecture where clients send requests to the server, which processes them before sending back a response. Figure 3-a illustrates an e-mail client interface, that allows a client to access its e-mails. The Simple Mail Transfer Protocol (SMTP) usually underlies the sending of e-mails to an e-mail server [22]. While the Internet Message Access Protocol (IMAP) or the Post Office Protocol Vesrion 3 (POP3) are used to retrieve e-mails from an e-mail server [23, 19]. For instance, when a client wants to send an e-mail, it uses several SMTP commands such as the *RCPT* command that allows the client to specify the recipient of the e-mail to send, and the *DATA* command that sends the e-mail content to the server. When a client wants to read its e-mails, it may use IMAP or POP3 commands such as the *LIST* command that asks the server to list the received e-mails, and the POP3 *RETR* command that asks the server to retrieve a given e-mail.

Sendmail, Microsoft Exchange Server, Postfix and Qmail are examples of e-mail servers [42, 49, 60, 4]. Instances of e-mail client software are Microsoft Outlook and Mozilla Thunderbird [28, 33]. Some e-mail services provide the users a web interface to access their e-mails, such as Gmail, Yahoo and Hotmail [17, 65, 29].



(a) E-mail application (b) Replicated e-mail servers

Figure 3: E-mail application and replicated servers

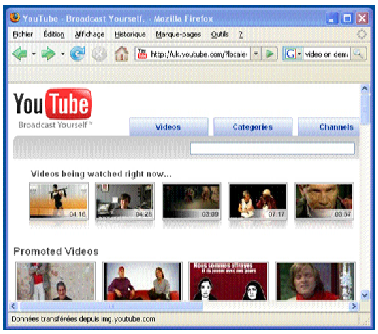
For scalability purposes, an e-mail server may be replicated as a set of multiple entities with appropriate consistency management policies [5, 41, 62]. Figure 3-b illustrates the architecture of a replicated e-mail service, in which client requests are spread among the different replicas of the service. In this context, and since the e-mail service may be more or less stressed over time depending on the varying client workload, dynamic resource provisioning would allow the system to provide good performance to end-users while requiring minimum resources.

3.2 Video-on-demand service

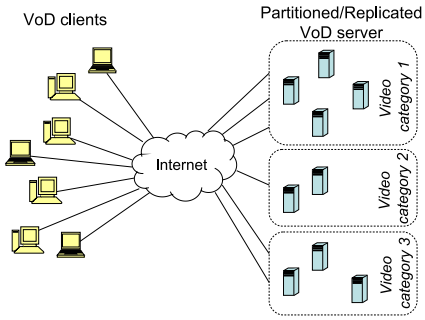
A video-on-demand (VoD) service allows users to interactively select and watch video data over the network. This provides users with streaming multimedia capabilities and, for instance, the ability to start, stop or rewind the streamed video. A video-on-demand system follows the classical client-server architecture where clients send requests to the server, which processes them before sending back a response. The Real Time Streaming Protocol (RTSP) usually underlies streaming media systems [20]. It controls on-demand delivery of real-time data such as audio and video. For instance, clients may send *RTSP PLAY* requests to play the accessed media stream or *RTSP PAUSE* requests to temporarily interrupt the media stream.

Apple QuickTime Streaming Server, Microsoft Windows Media Services and Alcatel/Lucent's pvServer Streaming module are examples of RTSP-based VoD server software [25, 27, 30]. Apple's QuickTime, RealPlayer and VLC media player are examples of VoD client software [24, 61, 40]. Instances of streaming media services are YouTube and Dailymotion [66, 13] (see Figure 4-a).

In this context, partitioning and replication are classical techniques used to build scalable VoD services [26, 11, 14, 68] (see Figure 4-b). Here, partitioning may be mapped to categories of videos provided by the VoD server. Client



(a) Video streaming application [66]



(b) Partitioning and replication in VoD servers

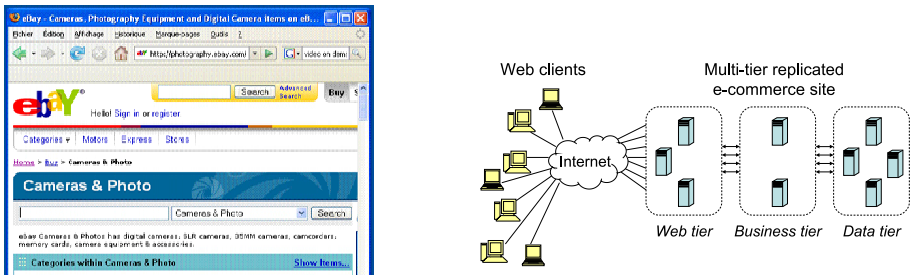
Figure 4: Video streaming application and VoD servers organization

requests are directed to the appropriate VoD service partition depending on the category of the video requested by the client. Client requests are then balanced between the different replicas of the served video for scalability purposes. Here, dynamic resource provisioning would allow to tackle the varying workload by dynamically allocating the necessary and sufficient resources to the VoD service.

3.3 E-commerce web server

A plenty of e-commerce web servers exists, such as online shopping sites (e.g. the Amazon.com shopping site [1], the Ebay.com auction site depicted in Figure 5-a [15]), online banking services (e.g. PayPal [36], Moneybookers [32]) and online booking sites (e.g. the Booking.com hotel booking site [7], the Momondo.com plane ticket reservation site [31]). More generally, an e-commerce web server provides users with several features, such as the ability to consult the provided products and services, the ability to perform online transactions to buy a product or a service, etc. E-commerce sites are based on the client-server architecture and make use of the HyperText Transfer Protocol (HTTP) [21]. HTTP is a communication protocol that allows web clients to interact with web servers and exchange hypermedia information. For instance, an HTTP GET request, the most frequently used web request type, allows a client to access a web resource available on the server. While an HTTP PUT request uploads the specified web resource on the server.

For scalability issues of e-commerce sites, the classical simple client-server architecture where the server consists of a single entity was extended to a pipelined system also known as the multi-tier architecture [48]. In a multi-tier architecture, the server is organized as a series of tiers. Such systems start with requests from web clients that flow through a front-end web server and provider of static content, then to a middle-tier enterprise server to execute the business logic of the application and generate web pages on-the-fly, and finally to a back-end database that stores non-ephemeral data. However, the complexity of multi-tier architectures and their low rate for delivering dynamic web documents (often one or two orders of magnitudes slower than static documents) place a significant burden on servers [18]. To face high loads and provide higher service scalability, a commonly used approach is to replicate servers (see Figure 5-b).



(a) E-commerce application [15]

(b) Multi-tier replicated e-commerce sites

Figure 5: E-commerce application and server organization

Replication-based clustering solutions are responsible of dynamically balancing the load among replicas, and managing replica consistency [51, 9, 10]. However, determining the right amount of resources that are needed by each tier of a multi-tier e-commerce service to handle its variable load is a challenging task. Dynamic resource provisioning makes all sense in the context of these complex distributed systems.

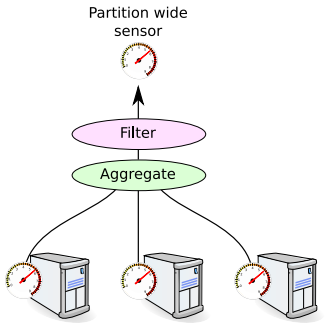
4 SELF-OPTIMIZATION OF INTERNET SERVICES

4.1 Architecture and Design principles

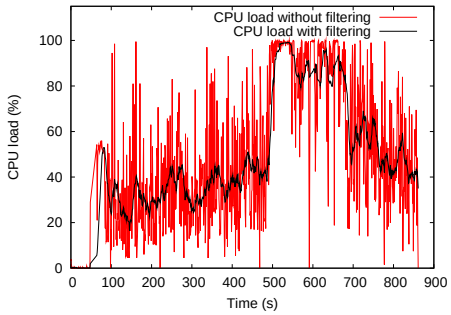
In the following, we describe the main architecture and design principles of self-optimization of Internet services. A self-optimization manager is responsible of applying a given self-optimization policy on an Internet service. The self-optimization policy described here is based on dynamic resource provisioning, i.e. on-line addition and removal of resources to and from an Internet service. A self-optimization manager is associated with each set of replicated entities of an Internet service. For instance in the Internet service described in Figure 3 a self-optimization manager will be associated with the set of replicated e-mail servers. In the Internet service described in Figure 4, a self-optimization manager will be associated with each partition of the VoD service. And in the Internet service described in Figure 5, a self-optimization manager will be associated with each tier of the multi-tier e-commerce web application. A self-optimization manager is organized as follows. It observes the behavior of a set of replicated entities and triggers resource provisioning or un-provisioning according to its observations.

A self-optimization manager applies a resource usage threshold-based policy to a set of managed entities (e.g. replicated entities). When the resource usage of the underlying set of replicated entities reaches a maximum threshold, that means that the system is over-loaded and thus the self-optimization manager provisions the set of managed entities with additional resources. Symmetrically, when the resource usage of the set of managed entities goes below a minimum threshold, that corresponds to an under-utilization of the system. In this case, the self-optimization manager removes resources from the set of managed entities. A self-optimization manager is organized in three main parts: (i) system observation, (ii) self-optimization policy, and (iii) system reorganization.

The first part is responsible of observing the behavior of the underlying managed system in terms of resource consumption. Here, resource consumption refers to hardware resources such as cpu, memory, disk or network. System observation may have the form of an on-line resource monitoring system that performs real-time monitoring of the system, or it may have the form of predictions of future resource usage of the system. The former is used to implement reactive self-optimization, while the latter applies in case of proactive self-optmization. On-line resource monitoring consists in resource usage indicators (i.e. sensors). Self-optimization is triggered when a sensor reports a value which violates some minimum or maximum thresholds. High-level sensors may aggregate and filter many lower-level sensors to provide meaningful resource usage indications. Aggregation allows to consolidate grouped resource usage information (e.g. partition-wide resource usage as shown in Figure 6-a), while filtering targets monitoring data quality such as stability, responsiveness. Aggregation is usually achieved through mathematical computations such as summing, averaging, minimum finding, etc; this depends on the nature of the information to measure and to report. Filtering generally targets the removal of meaningless artifacts for stability purpose through smoothing over time (e.g. raw average or EWMA), flip-flop filters, etc. Filtering effects are illustrated in Figure 6-b.



(a) Aggregation and filtering of sensors



(b) Filtering of monitoring data

Figure 6: Aggregation and filtering of sensors

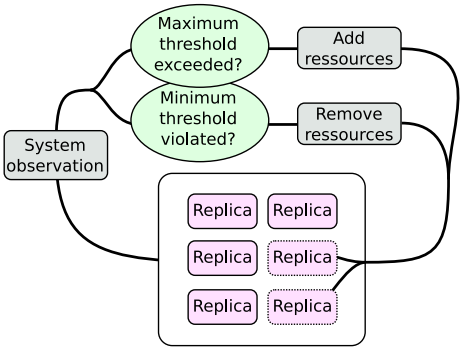


Figure 7: Dynamic provisioning control-loop and policy design

The central part of the self-optimization manager is its policy. Its general functioning is briefly described in Figure 7 and Algorithm 1. Here, the self-optimization policy is a control-loop that reacts to events received from the system observation part. Each time an event is notified, it is analyzed to check if the underlying managed system is over-loaded or under-utilized. If one of the observed resources exceeds its maximum threshold, that means that the managed system faces a bottleneck and is over-loaded. In this case the system is provisioned with additional nodes. Symmetrically, if all observed resources use less than their minimum threshold, that means that the overall system is under-utilized. Thus nodes are removed from the managed system. Addition and removal of nodes is done through system reorganization operations, the third part of the self-optimization manager.

This latter part of the self-optimization manager provides operations that actually perform the dynamic provisioning or un-provisioning of nodes to the managed system. Such operations consist in assigning new free nodes to the managed system, releasing nodes from the managed system, installing on a new node the software needed by the managed system when necessary, configuring the software and starting the software.

More generally, Figure 8 depicts a general example of how self-optimization applies in an Internet service. An Internet service may be organized as parti-

Algorithm 1 Algorithm for a dynamic provisioning policy

```
on Receive(event: MonitoringEvent):
if (event.consumption_1 > max_threshold_1)
      or (event.consumption_2 > max_threshold_2)
      or ...
      or (event.consumption_r > max_threshold_r) then
  // System is overloaded and need more resources
  AddResourcesToSystem()
else if (event.consumption_1 < min_threshold_1)
      and (event.consumption_2 < min_threshold_2)
      and ...
      and (event.consumption_r < min_threshold_r) then
  // System is underloaded and wastes resources
  RemoveResourcesFromSystem()
end if
```

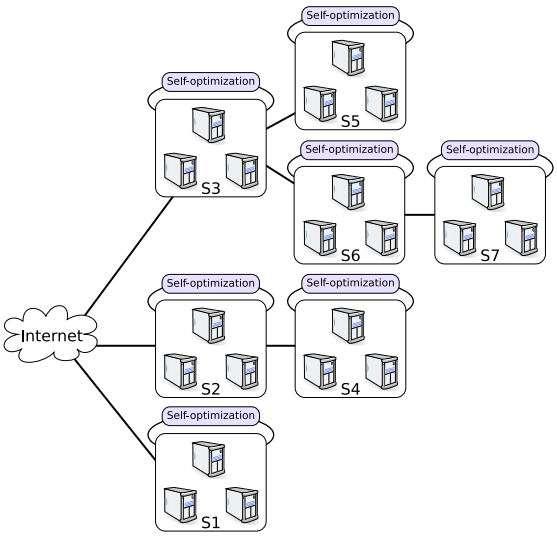
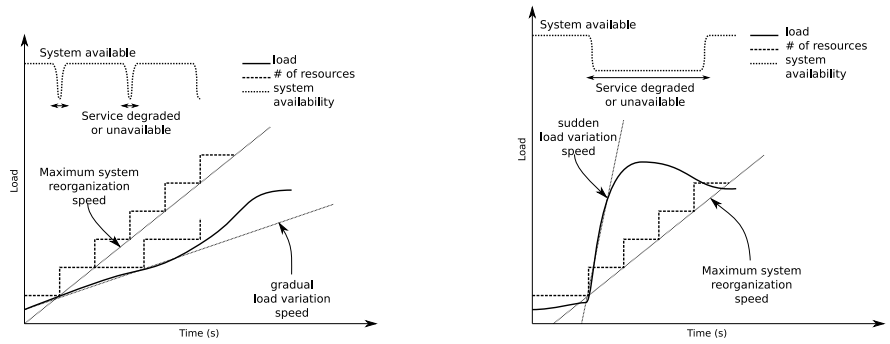


Figure 8: Architecture of self-optimized Internet services

tioned and pipelined sub-systems, where the partitioned and pipelined entities may be sets of replicated entities (S_1 to S_7 in Figure 8). In this context, a self-optimization manager is associated with each set of replicated entities. It is responsible of dynamically provisioning resources to that set of replicated entities. Moreover, the self-optimization managers of the Internet services cooperate in order to provide a global consistent behavior (e.g. preventing system oscillations as discussed in section 4.3).

In the following, we detail how the self-optimization manager tackles different types of load variation, and how it prevents system oscillations.



(a) Gradual load variation (b) Sudden load variation

Figure 9: Gradual load variation v.s. sudden load variation

4.2 Managing load variation

Load variations may happen following different schemes. A common scenario consists in a gradual change of the load which will progressively induce an underload or an overload in the system. Another common scenario often happens at the occasion of big events and consists in sudden load variations also commonly referred to as load spikes or flash crowds.

Whether a load variation is considered as gradual or sudden is related to the relative difference between the speed of load variation and the speed of system reorganization. Gradual load variation corresponds to load variation that happens slower than the system reorganization speed (see Figure 9-a). In this case, simple system reorganizations such as single resource addition or removal (i.e. at the granularity of one resource at a time) are sufficient to absorb the load variation. On the contrary, sudden load variation happens when the load variation is faster than the system reorganization speed (see Figure 9-b). In this case, fine-grain heuristics are required to determine the optimal capacity planning of the system, in order to accelerate the process of system reorganization towards an optimum state. In case of sudden load variation, it is necessary to determine the amount of resources to (un-)provision, before actually performing the (un-)provisioning in a single step. In the following, we present two mechanisms to address both types of load variations.

Gradual Load Variations. In a system undergoing gradual load variations, the capacity planning of the system can be continuously adjusted through system reorganizations as simple as adding or removing resource units one at a time. Indeed, the gradual load variation assumption ensures that the system provisioning will be updated promptly enough to absorb and follow the load variation. We implemented a self-optimization manager able to handle gradual load variations. The manager relies on low-level resource usage system observations (such as cpu, memory, disk or network) and, based on these observations, triggers single node addition or removal to the system. We experimented and evaluated this manager on a clustered Internet service implementing a multi-tier e-commerce web application that was submitted to gradual load variation. The Internet service was able to self-optimize its behavior according to the changing workload (see section 5.2.1).

Sudden Load Variations. In a system submitted to sudden load variations, updates to the system capacity planning may require addition or removal of multiple resources at a time, so as to absorb the load peaks. In case of sudden load variation, it is preferable to factorize and parallelize system reorganization operations, thus increasing the overall speed of system reorganization. The challenge here is to determine as accurately as possible how many nodes to add or remove in one step. We implemented a self-optimization manager that copes with sudden load variations. It is based on low-level resource system observations (cpu, memory, disk and network) as well as higher application-level observations (such as the number of concurrent transactions running in a server). Application-level observations allow the construction of heuristics functions to determine the optimum capacity planning of the system. We identified a heuristics function that calculates the optimum capacity planning as being linearly proportional to concurrent transactions in the system. Then based on the result of this function, multiple nodes are assigned or released in parallel. We implemented this self-optimization policy and applied it to a cluster of Internet services that implements an e-commerce web application. In the presence of load spikes, the e-commerce web application was able to efficiently self-optimize (see section 5.2.2).

4.3 System oscillation management

Another issue of self-optimization is that it may introduce system instabilities during which sensors may report meaningless information. Thus, interpreting these signals is likely to be irrelevant and leads to erroneous decisions. Indeed, dynamic resource provisioning of an Internet service may induce multiple concurrent provisioning operations that are actually not necessary and would, as a result, hurt the overall Internet service performance. For instance, in a multi-tier Internet service composed of a front-end web server and a database back-end organized as a pipelined system, the database back-end might become a bottleneck and induce an underload on the frontend web server (which then waits for responses from the back-end tier). In such a situation, the self-optimization could trigger provisioning operations, increasing the amount of resources on the database back-end tier on one hand, while reclaiming unused resources on the front-end tier. Obviously, the latter un-provisioning on the front-end tier is a consequence of the dependency between the front-end web server and the database back-end that leads to un-necessary operations, and therefore to system oscillations. To prevent system oscillations, we introduce a technique that (i) first automatically calculates inter-dependencies between sub-parts of the system, and then (ii) automatically prevents system oscillation occurrence.

The system oscillation management relies on a description of the system that allows the manager to determine dependencies between parts of the system. More precisely, the manager is given a representation of the system in terms of pipelined and partitioned sub-systems (see Figure 1). Thanks to this knowledge, the manager infers a dependency function defined as follows: (i) a sub-system S_i depends on a sub-system S_j if S_i and S_j are parts of a pipelined system, and (ii) a sub-system S_i always depends on itself. Indeed, a pipeline materializes the dependency between sub-systems, while a partition materializes their independency.

Notice that in pipelined sub-systems, the workload of one of the sub-systems may have a side-effect on another sub-system in pipeline. This is due to the fact that client request processing may flow through all or part of the pipelined sub-systems. While in case of partitioned sub-systems, the workload of the different sub-systems are independent from each other; each partition being responsible of processing requests independently from the other partitions. Thus, based on the inter-dependency function and the knowledge of the system architecture, the self-optimization manager is able to automatically identify inter-dependent parts of the Internet service. To prevent oscillations from occurring, the self-optimization manager ensures that during a self-optimization operation on a part of the system, self-optimization is inhibited on any inter-dependent part (during a given delay). Once the inhibition delay has expired, new self-optimization operations are allowed for execution again.

We implemented the system oscillation management for a self-optimized e-commerce web application hosted by a two-tier Internet service where each tier of the Internet service was replicated and dynamically provisioned. The two tiers of the Internet service were identified as a pipelined system. Thus, all system reorganization happening on the first or the second tier of the system triggered an inhibition that blocked any new reorganization on the first and the second tier (see section 5.2).

4.4 Discussion

The proposed self-optimization approach based on resource usage observations and on simple system reorganization is attractive thanks to its simplicity. Its basic design confers the self-optimization approach a generic behavior, thus allowing its appliance to many different Internet services with minimal effort. Indeed, since self-optimization is based on low-level resource monitoring, it can apply to different Internet services. Of course, the genericity is reduced when self-optimization makes use of application-level heuristics, like it was the case for tackling sudden load variations (such observations being application-dependent).

However, a major open issue of this work concerns tuning and configuring the self-optimization manager itself. Indeed, configuration parameters may have the form of min and max thresholds used to guide dynamic resource provisioning, inhibition delays used to prevent system oscillation. The configuration of these parameters must be carried carefully since it conditions the efficiency of the self-optimization manager. In our experiments, we manually tuned these parameters of the self-optimization manager, based on an observation of the behavior of the underlying Internet service. The proposed self-optimization manager takes in charge dynamic optimization and resource provisioning in an Internet service; However, techniques that would assist, and possibly automate, the configuration of the parameters of the self-optimization manager would make the usage of self-optimization easier.

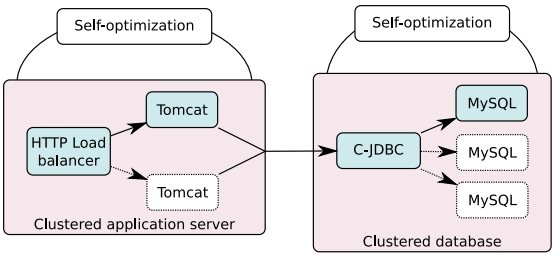


Figure 10: Experimental dynamic J2EE infrastructure

5 EXPERIMENTAL EVALUATION

5.1 Experimental environment

Hardware environment. Experimentations have been conducted on a cluster of x86-compatible machines with bi-1.8GHz Xeon CPUs and 1GB RAM, connected via a 100Mb/s Ethernet LAN.

Software environment. The cluster nodes have been installed with Linux 2.4 kernels and with the following J2EE middleware: Apache HTTPD 1.3.9 as the web server [50], Jakarta Tomcat 3.3.2 as the enterprise server [52], MySQL 4.0.17 as the database server [34], PLB 0.3 as the web server clustering solution [37], Tomcat clustering as the enterprise server clustering solution [52] and c-jdbc 2.0.2 as the database server clustering system [10].

Application. The evaluation has been realized with the Rubis multi-tier J2EE application benchmark which implements an auction site [2]. Rubis defines several web interactions (e.g registering new users, browsing, buying or selling items); and it provides a benchmarking tool that emulates web client behaviors and generates a tunable workload. Rubis comes with two mixes: a browsing mix in which clients execute 100% read-only requests and a bidding mix composed of 85% read-only interactions. This benchmarking tool gathers statistics about the application. Rubis was deployed as a cluster-based replicated multi-tier system, consisting of a cluster of replicated web/enterprise servers as a front-end, and a cluster of replicated database servers as a backend. The cluster hosting this instance of Rubis has been enhanced so as to provide dynamic provisioning abilities (see figure 10). Indeed the experimental cluster allowed us to dynamically adapt the resource provisioning of the web/enterprise servers and the resource provisioning of the replicated database, while providing meaningful resource usage sensors. We used the Rubis 1.4.2 version of the multi-tier J2EE application running the middleware platform described above.

5.2 Experimental results

We present here the main results obtained after experimenting various scenarios on the environment described previously.

Protocol. The purpose of the following experimental evaluation is to demonstrate the correctness and the effectiveness of the proposed dynamic provisioning technique. To achieve this all conducted experiments include a comparison of the dynamic provisioning algorithm behavior with the standard static provision-

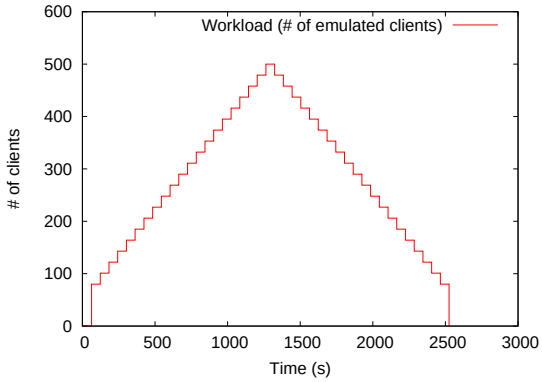


Figure 11: Generated workload to simulate gradual variations

ing practice. The reference statically provisioned infrastructure corresponds to the J2EE infrastructure presented in figure 10 with a single Tomcat server and a single MySQL server.

5.2.1 Handling gradual load variations

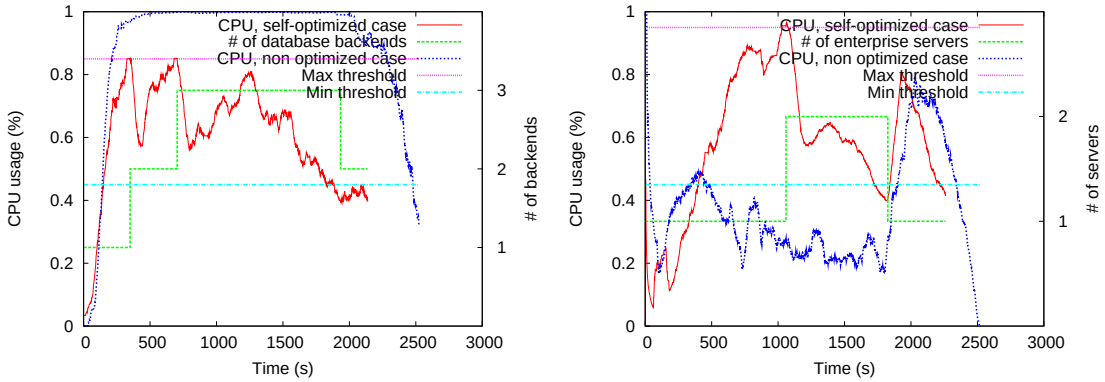
We present here an experiment demonstrating the dynamic provisioning for gradual load variations. Therefore we expose the Rubis web application hosted on the cluster described previously to a workload with gradual variations intending to demonstrate the ability of our dynamic provisioning mechanism to adapt and follow the changing workload appropriately.

The workload (see figure 11) starts lightly by simulating 80 clients and slowly increases up to 500 clients by steps of 20 new clients every minute. After this the workload symmetrically decreases from 500 clients down to 80 clients. This dynamic workload corresponds to the scenario of a normal service day where the number of clients increases in the morning up to a maximum in the afternoon and then decreases in the evening.

Figure 12-a (resp. figure 12-b) present the aggregated resource usage[1] of the database servers (resp. the application servers) hosting Rubis during the experiment. Both the behavior of the statically provisioned and the dynamically prosioned systems are represented on these figures. Moreover the figures display the minimum and maximum thresholds driving the dynamic provisioning control-loops. Finally the figures also present the current number of nodes provisioned for the specific part of the system they represent.

When configuring the hosting cluster to be statically provisioned for Rubis, we observe that the service gets quickly overloaded with its aggregated resource being saturated. In comparison when enabling dynamic provisioning on the hosting cluster, the violation of the maximum threshold by the monitored resource usage triggers the increase of the current provisioning of the system part by one node. This induces a decrease of the aggregated resource usage that

[1]In the present experiments, the CPU was the unique bottleneck resource. For simplicity and space constraints, we only present CPU usage.



(a) Database behaviour in response to gradual load variations

(b) Application server behaviour in response to gradual load variations

Figure 12: Behavior of Rubis entities in response to gradual load variations

allows the service to keep functioning while the statically provisioned system is thrashing.

The dynamic provisioning mechanism is triggered three times during the workload increase: the two first activations of the dynamic provisioning lead to two additions of one node to the clustered database, thus shifting the bottleneck on the application server which then also takes advantage of one node addition. Symmetrically as the load decreases, the aggregated resource usage drops and eventually violates the minimum threshold. The threshold violation leads to resource unprovisioning as expected.

5.2.2 Handling load spikes

This section details an experimentation which aims at demonstrating the effectiveness of our dynamic provisioning algorithm in case of sudden load variation. To this end Rubis is then exposed to a load spike that widely exceeds the current capacity of the system.

As shown in figure 13 the workload starts moderately by simulating 100 clients. After three minutes (at time 180s) the workload instantaneously jumps to 500 clients, thus generating a load spike which cannot be handled correctly by the current system configuration. Figure 13 presents the resulting behaviour of the system in response to the generated load spike, both in case of static and dynamic provisioning. The behaviour is here depicted through the response times to requests as perceived by clients. More precisely a dot on this figure represents one client request which has been submitted at the time the dot is located at (its x-axis), and which took as many seconds to be served as its y-axis location.

The initial moderate workload directed to the system is satisfied before the load spike, which translates to low request latencies. Just after the load spike happens the request response times jump very quickly. This reveals a thrashing of the underlying system which is actually unable to serve requests decently anymore. When the system is statically provisioned the thrashing is very strong so that many requests submitted after the spike are given a response after the end

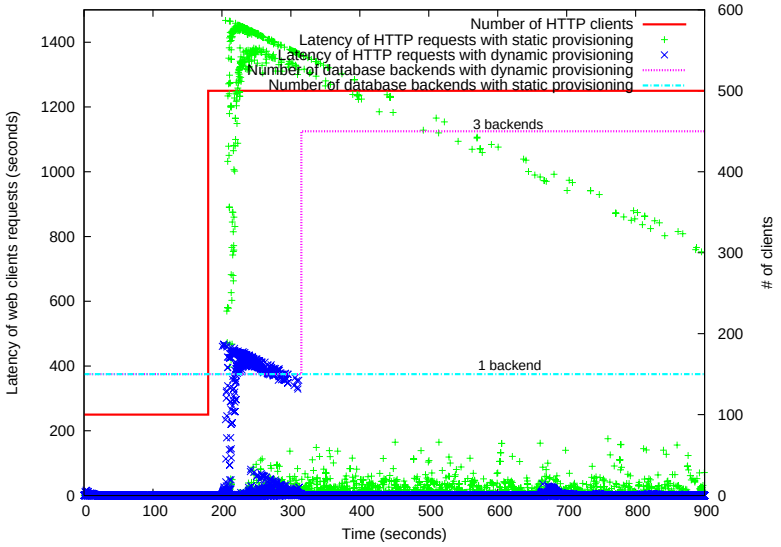


Figure 13: Sudden load variations

of the experiment. For example requests submitted at time 200 seconds after the beginning of the experiment have response times which exceeds 1400 seconds, meaning that the response have been issued at time 1600s. In comparison to a dynamically provisioned system, the load spike is quickly detected through its resulting thrashing. The control-loop handles this event and provisions the system with two more database servers. Once these servers are integrated into the running system, the client requests get served correctly as before, as figure 13 shows low latencies again after time 300s.

5.2.3 Performance overhead

In order to measure the possible performance overhead induced by the self-management framework, we compared two executions of the same multi-tier system: when it is run over Jade and when it is run without Jade. During the experiments, the managed application has been submitted to a medium workload so that its execution under the control of Jade induced no self-optimization.

	Throughput (req./s)	Resp. time (ms)	CPU usage (%)	Memory usage (%)
Self-optimized	12	89	12.74	20.1
Non optimized	12	87	12.42	17.5

Table 1: Performance overhead

The results presented in Table 5.2.3 show no significant overhead in terms of application response times and throughput. We can notice a slight memory overhead (20.1% vs. 17.5%) that can be linked with the creation of internal software components by Jade. However, Jade does not induce a perceptible

overhead on CPU usage; this is due to the fact that Jade does not intercept application communications but only configuration/management operations.

6 RELATED WORK

Past work on resource management of Internet services falls in different categories. A first category has focused on studying strategies to manage and share a given and fixed set of resources [64, 63, 38, 59, 57, 44, 43, 6].

In [64, 63], the authors experiment graceful degradation in case of overloads. They achieve this thanks to a framework that helps the design of staged event-driven software allowing a fine-grained control over the resource sharing among the various stages. This provides a good level of performance isolation between concurrent applications as well as a salient resistance to overloads as the imposed design prevents thrashing. Moreover an overload controller minimizes the amount of dropped requests as a result of admission control while providing guarantees on the associated response times. Some controllers have been developed to dynamically adapt the resource allocated to a stage in response to load changes. However this approach requires all applications to be specifically (re-)designed according to the framework.

In [38], the authors explore an approach specifically targeting Web servers to provide adaptation to changing workloads. In this approach the Web server concurrently processes many QoS classes and adjusts their respective resource share through a two-level control-loop: system-wide adaptations determine an operating mode and derive classes resource shares, while local adaptations trade resources between classes to enforce the QoS requirements. To keep the system stable and prevent oscillations local adaptations are committed gradually.

In [59], the authors describe a provisioning technique based on a controlled resource overbooking that yields improved resource usages while guaranteeing a statistically controlled resource overload level. Hosted applications are being profiled to derive their respective resource requirements. These information drive an application placement algorithm which implements the overbooking. Applications are then isolated through low-level QoS kernel mechanisms. Sharc extends this preliminary study and provides dynamic provisioning through adaptations of the reservations allocated to applications parts on nodes [57]. Interesting features of Sharc and Resource Overbooking are their ability to manage multiple resources simultaneously (CPU and Network bandwidth) and their genericity as demonstrated through their experiments on applications with different profiles. However resources are being wasted in case of general underload.

Neptune achieves dynamic provisioning thanks to an elaborated two-level load-balancer and class-based differentiation [44, 43]. The load-balancing first applies at the cluster level and spread the load equally between the nodes, and then applies at the node level to select the *best* request to serve next. Thus load variability is handled by an indirect admission control mechanism which discriminates requests based on their estimated revenue. While Neptune requires application to be (re-)designed according to their given framework, Quorum essentially reimplements Neptune and removes this last constraint [6].

Another category of work on resource management of Internet services has considered the management of a dynamically extensible set of resources, where the infrastructure can dynamically grow or shrink [3, 35, 45, 46, 55, 58].

Ocano provides an adaptive hosting environment with a dynamic partitioning of the resources among the running applications [3]. This dynamism allows the system to react to load peaks by increasing the partition size of the concerned application and to shift unused resources from under-loaded applications

to the others. The main issue in this work seems to be the node allocation delay. That explains why the platform assumes that some application parts cannot be dynamically and are thus statically allocated and configured (e.g. the database tier). OnCall is similar to Ocano but specifically targets fast handling of load spikes thanks to an approach based on virtual machines which can be promptly activated when required [35]. In case of load spikes extra nodes are allocated to applications willing to pay more, based on a free market of nodes. Contrary to Ocano this project does not assume any statically allocated resources and looks more generic with respect to the managed applications though this aspect has not been demonstrated. Finally none of these projects takes system oscillations of the dynamic provisioning into account.

In [45, 46], the authors propose a self-optimized dynamic provisioning algorithm that specifically targets a cluster of databases. Regarding load spikes the system always provisions a set of unused nodes with database instances kept within a given range of freshness with respect to the active database instances. This contributes to improve the latency of provisioning operations. Furthermore oscillations are explicitly prevented as a result of a delay-aware allocation mechanism of database replica.

Cataclysm is a hosting platform for Internet service which features dynamic provisioning through a dynamic partitioning of nodes between the running applications and a adaptive size-based admission control mechanism which takes advantage of a request classifier to optimally degrades the service quality in case of overloads [55, 58]. The provisioning algorithm is based on a basic model of clustered network services. Cataclysm has been specially designed to absorb extreme overloads: the size-based admission controller prevents the system from thrashing as a result of accepting too many requests, additionally taking advantage of a request classifier to maximize the revenue during overloads, while the dynamic provisioning algorithm adds extra resources in case of overloads. The provisioning algorithm relies on a coarse-grained modeling of simple Internet services. The strength of Cataclysm is the cooperation of admission control and dynamic provisioning as components of an integrated resource management system. It assumes simple Internet services structures where the database back-end is statically provisioned.

Finally, besides the above-mentioned heuristics-based approaches, another category of work on resource management of Internet services has studied mathematical characterization and analytical modeling of the systems [56, 53, 12, 67, 47].

For instance, in [53, 54], authors propose a model for multi-tier Internet applications. This model captures the structure and the behavior of Internet applications built as cooperative entities (i.e. entities in series) thanks to a network of queues. Transitions between queues standing for two connected tiers are probabilistic. Indeed this allows the model to capture requests processing paths (including caching mechanisms) through appropriate values for these transition probabilities. Replication and load-balancing, concurrency limits and requests classification and differentiation are taken into account as enhancements over the baseline model. The effectiveness of the model to achieve accurate capacity planning is demonstrated in a dynamic provisioning scenario in which parameters of the model are determined by mean-value analysis.

7 CONCLUSION

Internet services, such as e-mail services, streaming servers and multi-tier e-commerce web services, have to deal with a varying workload, ranging from stable workload to a large amount of concurrent users accessing the service. Such services usually need to deal with two antagonist objectives: providing good performance to end-users while minimizing operational costs of the Internet services.

In this paper, we presented a self-optimization system for Internet services. We described the design principles and implementation details of a self-optimization manager that we integrated to the Jade autonomic management framework. This self-optimization manager adapts the Internet service by dynamically (un-)provisioning resources to the service, according to its changing workload. The main results of the self-optimization manager are three-fold: (i) its ability to tackle both situations of gradual load variations and load spikes, (ii) its ability to automatically prevent system oscillations that may result from dynamic resource provisioning, and (iii) its genericity and applicability to different types of Internet services.

We applied self-optimization to a realistic e-commerce web system, consisting of web and application servers and database servers, running in a Linux cluster. The experimental results show that the Internet service is able to dynamically and successfully adapt the amount of resources it uses to its varying workload, and to provide good performance to end-users while minimizing its operational cost.

References

[1] Amazon.com Inc. Amazon site, 2007. http://www.amazon.com/.

[2] C. Amza, E. Cecchet, A. Chanda, A. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel. Specification and Implementation of Dynamic Web Site Benchmarks. In *IEEE 5th Annual Workshop on Workload Characterization (WWC-5)*, Austin, TX, November 2002.

[3] Karen Appleby, Sameh A. Fakhouri, Liana L. Fong, Germn S. Goldszmidt, Michael H. Kalantar, S. Krishnakumar, Donald P. Pazel, John A. Pershing, and B. Rochwerger. Ocano-SLA based management of a computing utility. In *Proceedings of Integrated Network Management*, pages 855–868, 2001.

[4] D. J. Bernstein. qmail, 2007. http://cr.yp.to/qmail.html/.

[5] A. D. Birrell, R. Levin, M. D. Schroeder, and R. M. Needham. Grapevine: an exercise in distributed computing. *Communications of the ACM*, 25(4):260–274, April 1982.

[6] Josep M. Blanquer, Antoni Batchelli, Klaus Schauser, and Rich Wolski. Quorum: Flexible quality of service for internet services. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 159–174, 2005.

[7] Booking.com Inc. Booking.com site, 2007. http://www.booking.com/.

[8] Sara Bouchenak, Noël de Palma, and Sacha Krakowiak. Tolérance aux fautes dans les grappes d'applications Internet. In *4ème Conférence Française sur les Systèmes d'Exploitation (CFSE 2005)*, Le Croisic, France, April 2005.

[9] B. Burke and S. Labourey. Clustering With JBoss 3.0. October 2002. http://www.onjava.com/pub/a/onjava/2002/07/10/jboss.html.

[10] E. Cecchet, J. Marguerite, and W. Zwaenepoel. C-JDBC: Flexible Database Clustering Middleware. In *USENIX Annual Technical Conference, Freenix track*, Boston, MA, June 2004. http://c-jdbc.objectweb.org/.

[11] C. K. Chan, T. P. J. To, and C. K. Li. Strategies for video file replication based on multiple video popularity models. In *Internet and Multimedia Systems and Applications (IMSA 2003)*, Honolulu, USA, August 2003.

[12] Abhishek Chandra, Weibo Gong, and Prashant Shenoy. Dynamic resource allocation for shared data centers using online measurements. In *Proceedings of the Eleventh IEEE/ACM International Workshop on Quality of Service (IWQoS 2003)*, Monterey, CA, June 2003.

[13] Dailymotion. Dailymotion, 2007. http://www.dailymotion.com/.

[14] M. Ditze, C. Loeser, P. Altenbernd, and K. H. Wan. Improving content replication and QoS in distributed peer-to-peer VoD appliances. In *International Conference on Distributed Computing Systems Workshops (ICDCS 2004 Workshops)*, Honolulu, USA, March 2004.

[15] eBay Inc. eBay site, 2007. http://www.ebay.com/.

[16] The Apache Software Foundation. http://tomcat.apache.org/connectors-doc/. http://tomcat.apache.org/connectors-doc/.

[17] Google. Gmail, 2007. http://www.gmail.com/.

[18] X. He and O. Yang. Performance Evaluation of Distributed Web Servers under Commercial Workload. In *Embedded Internet Conference 2000*, San Jose, CA, September 2000.

[19] IETF. Post Office Protocol - Version 3, 1996. http://tools.ietf.org/html/rfc1939.

[20] IETF. Real Time Streaming Protocol (RTSP), 1998. http://tools.ietf.org/html/rfc2326.

[21] IETF. Hypertext Transfer Protocol - HTTP/1.1, 1999. http://tools.ietf.org/html/rfc2616.

[22] IETF. Simple Mail Transfer Protocol, 2001. http://tools.ietf.org/html/rfc2821.

[23] IETF. Internet Message Access Protocol- Version 4rev1, 2003. http://tools.ietf.org/html/rfc3501.

[24] Apple Inc. QuickTime, 2007. http://www.apple.com/quicktime/.

[25] Apple Inc. QuickTime Streaming Server, 2007. http://www.apple.com/quicktime/streamingserver/.

[26] Infosys. Distributed Video-on-Demand - A grid-based VoD solution, April 2006. White paper. http://www.infosys.com/.

[27] Alcatel Lucent. pvserver, 2007. http://www.pvnetsolutions.com/products/pvserver.html.

[28] Microsoft. Outlook, 2007. http://office.microsoft.com/outlook/.

[29] Microsoft. Windows Live Hotmail, 2007. http://www.hotmail.com/.

[30] Microsoft. Windows Media Services, 2007. http://www.microsoft.com/windowsmedia/forpros/server/server.aspx.

[31] Momondo.com. Momondo.com site, 2007. http://www.momondo.com/.

[32] Moneybookers Ltd. Moneybookers site, 2007. http://www.moneybookers.com/.

[33] Mozilla. Thunderbird, 2007. http://www.mozilla.org/products/thunderbird/.

[34] MySQL. MySQL Web Site. http://www.mysql.com/.

[35] James Norris, Keith Coleman, Armando Fox, and George Candea. OnCall: Defeating spikes with a free-market application cluster. In *1st International Conference on Autonomic Computing (ICAC'04)*, pages 198–205, New York, NY, USA, May 2004.

[36] PayPal. PayPal site, 2007. http://www.paypal.com/.

[37] PLB. PLB - A free high-performance load balancer for Unix. http://plb.sunsite.dk/.

[38] Prashant Pradhan, Renu Tewari, Sambit Sahu, Abhishek Chandra, and Prashant Shenoy. An observation-based approach towards self-managing web servers. In *IWQoS'02: Tenth IEEE International Workshop on Quality of Service*, pages 13–22, 2002.

[39] M. Rabinovich and O. Spatscheck. *Web Caching and Replication*. 2001.

[40] RealNetworks. Realplayer, 2007. http://www.real.com/player/.

[41] Y. Saito, B. N. Bershad, and H. M. Levy. Manageability, availability and performance in Porcupine: a highly scalable, cluster-based mail service. In *The seventeenth ACM Symposium on Operating Systems Principles (SOSP'17)*, Kiawah Island Resort, SC, USA, December 1999.

[42] Sendmail.org. Sendmail, 2007. http://www.sendmail.org/.

[43] Kai Shen, Hong Tang, Tao Yang, and Lingkun Chu. Integrated resource management for cluster-based internet services. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI'02)*, pages 225–238, Boston, Massachusetts, USA, December 2002.

[44] Kai Shen, Tao Yang, Lingkun Chu, Joanne Holliday, Douglas A. Kuschner, and Huican Zhu. Neptune: Scalable replication management and programming support for cluster-based network services. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS'01)*, pages 197–208, San Francisco CA, March 2001.

[45] Gokul Soundararajan and Cristiana Amza. Autonomic provisioning of backend databases in dynamic content web servers. Technical report, Department of Electrical and Computer Engineering, University of Toronto, 2005.

[46] Gokul Soundararajan, Cristiana Amza, and Ashvin Goel. Database replication policies for dynamic content applications. In *First EuroSys Conference (EuroSys 2006)*, Leuven, Belgium, April 2006.

[47] Christopher Stewart and Kai Shen. Performance modeling and system management for multi-component online services. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 71–84, 2005.

[48] Sun Microsystems. Java 2 Platform Enterprise Edition (J2EE). http://java.sun.com/j2ee/.

[49] Microsoft TechNet. Exchange server, 2007. http://technet.microsoft.com/en-us/exchange/.

[50] The Apache Software Foundation. Apache HTTP Web Server. http://httpd.apache.org/.

[51] The Apache Software Foundation. Apache Tomcat. http://tomcat.apache.org/.

[52] The Apache Software Foundation. Apache Tomcat. http://tomcat.apache.org/.

[53] Bhuvan Urgaonkar, Giovanni Pacifici, Prashant Shenoy, Mike Spreitzer, and Asser Tantawi. Analytic modeling of multitier internet applications. *ACM Transaction on the Web*, 1(1):2, 2007.

[54] Bhuvan Urgaonkar, Giovanni Pacifici, Prashant J. Shenoy, Mike Spreitzer, and Asser N. Tantawi. An analytical model for multi-tier internet services and its applications. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS'05)*, pages 291–302, Banff, Alberta, Canada, June 2005.

[55] Bhuvan Urgaonkar and Prashant Shenoy. Cataclysm: Handling extreme overloads in internet services. Technical report, Department of Computer Science, University of Massachusetts, November 2004.

[56] Bhuvan Urgaonkar, Prashant Shenoy, Abhishek Chandra, and Pawan Goyal. Dynamic provisioning of multi-tier internet applications. In *Proceedings of the 2nd IEEE International Conference on Autonomic Computing (ICAC'05)*, Seattle, June 2005.

[57] Bhuvan Urgaonkar and Prashant J. Shenoy. Sharc: Managing CPU and network bandwidth in shared clusters. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 15(1):2–17, January 2004.

[58] Bhuvan Urgaonkar and Prashant J. Shenoy. Cataclysm: policing extreme overloads in internet applications. In *Proceedings of the 14th international conference on World Wide Web, (WWW'05)*, pages 740–749, Chiba, Japan, May 2005.

[59] Bhuvan Urgaonkar, Prashant J. Shenoy, and Timothy Roscoe. Resource overbooking and application profiling in shared hosting platforms. In *Proceedings of the 5th Symposium on Operating System Design and Implementation (OSDI'02)*, Boston, Massachusetts, USA, December 2002.

[60] W. Venema. The postfix home page, 2007. http://www.postfix.org/.

[61] VideoLAN. Vlc media player, 2007. http://www.videolan.org/vlc/.

[62] J. R. von Behren, Steven E. Czerwinski, A. D. Joseph, E. A. Brewer, and J. Kubiatowicz. NinjaMail: The Design of a High-Performance Clustered, Distributed E-Mail System. In *International Workshop on Parallel Processing (ICPP Workshops 2000)*, Toronto, Canada, August 2000.

[63] Matt Welsh and David Culler. Adaptive overload control for busy internet servers. In *Proceedings of the 4th USENIX Conference on Internet Technologies and Systems (USITS'03)*, March 2003.

[64] Matt Welsh, David E. Culler, and Eric A. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP'01)*, pages 230–243, Banff, Alberta, Canada, October 2001.

[65] Yahoo Inc. Yahoo! Mail, 2007. http://www.yahoomail.com/.

[66] YouTube. Video streaming server, 2007. http://www.youtube.com/.

[67] Qi Zhang, Ludmila Cherkasova, and Evgenia Smirni. A regression-based analytic model for dynamic resource provisioning of multi-tier applications. In *ICAC '07: Proceedings of the Fourth International Conference on Autonomic Computing*, page 27, Jacksonville, Florida, USA, June 2007.

[68] X. Zhou and C. Z. Xu. Optimal video replication and placement on a cluster of video-on-demand servers. In *International Conference on DParallel Processing (ICPP 2002)*, Vancouver, Canada, August 2004.

Contents

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399