SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •
Progress of Projects Supported by NSFC

December 2012 Vol. 55 No. 12: 2708–2722
doi: 10.1007/s11432-012-4738-9

Towards a degradation-based mechanism for adaptive overload control

WANG ZiYou[1,2], ZHOU MingHui[1,2]* & MEI Hong[1,2]

[1] *Software Institute, School of Electronics Engineering and Computer Science, Peking University, Beijing* 100871*, China;*
[2] *Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, Beijing* 100871*, China*

Received June 1, 2012; accepted November 5, 2012

Abstract Uncontrolled overload is one of the major causes of the system's decline in reliability and performance for web applications. It also harms the construction of trustworthy software. This paper presents a degradation-based mechanism to adaptively control overload in complex, dynamic web applications. In our mechanism, the bottlenecks in the application's performance are determined by internally monitoring the performance and resource utilization state of the application, which is decomposed into a set of services. After locating the bottlenecks, two decision algorithms are proposed to dynamically and adaptively generate proper degradation plans without delay. By degrading the service which consumes critical resources and has a low priority from the perspective of business logic, the application can keep providing key services even when overload occurs. We implement a prototype and conduct a case study on a wiki application. We also evaluate our approach and demonstrate its effectiveness in simulated overload cases. Through handling overload, we make a breakthrough in building trustworthy software in an open, dynamic and rapidly changing environment.

Keywords overload, service degradation, resource monitoring, performance management

Citation Wang Z Y, Zhou M H, Mei H. Towards a degradation-based mechanism for adaptive overload control. Sci China Inf Sci, 2012, 55: 2708–2722, doi: 10.1007/s11432-012-4738-9

1 Introduction

With the development of application mode toward an "Internet-centered environment and service-oriented architecture", web-based software system nowadays is running on an open, dynamic and rapidly changing environment [1]. But software system cannot always work as expected or trusted because of the problems it has on correctness, reliability, security and performance. For instance, the occurrence of various types of faults and errors will call a stop to its proper operation and cause damage to users and society [2].

Overload is one of the major causes of the system's rapid decline in reliability and performance for web applications. It is defined as the point when the demand on at least one of the server's resources exceeds the capacity of that resource [3]. When overload happens, the response time of the software will quickly grow to an unacceptable level; meanwhile, due to the exhaustion of computing resources, the behavior of software may also be uncontrollable, causing even downtime and denial of services and

---

*Corresponding author (email: zhmh@pku.edu.cn) other serious errors. In e-commerce applications, such a serious error could translate into sizable revenue losses. For example, due to overload, most of the well-known U.S. online shopping sites have experienced service outages during the holiday seasons of 2000, among which Amazon stopped services for almost 30 minutes and Best Buy stopped services for about 4 days. Although Wal-Mart did not stop its service during overload, it took an average time of 260 seconds for a client to open a page. These abnormal system behaviors all caused great losses to the shopping sites mentioned above. Therefore, solution to server overload that provides stable server performance is critical for a system to remain operational in the presence of overload even when the incoming request rate is several times higher than the system's capacity.

Traditional software theory understands software change after deployment from a static point of view, under which software maintenance is always a passive response to problems emerged. For trustworthy software, a change from breakdown maintenance to prevention design and proactive monitoring is needed to form a control method for trustworthiness during its dynamic evolution [1]. Therefore, the construction of a reasonable and effective overload management mechanism from a dynamic point of view becomes an important research field in the construction of trustworthy software.

Service degradation1) is one of the solutions to server overload [4]. It avoids rejecting clients (as admission control does) by reducing the level of service under overload conditions. Although service degradation cannot handle all overload cases (say, when the load is too high to support even the lowest quality setting), using it can save bottleneck resources in order to admit a larger number of requests for key services. For example, rejecting extra requests to an e-commerce site when overload occurs may seem a reasonable choice, but the site will probably lose thousands of potential transactions worth millions of dollars. Using service degradation can avoid the situation by both admitting more requests and keeping the key services going.

However, some degradation-based overload control approaches used in nowadays business sites have several limitations on flexibility and adaptability. Firstly, some manual degradation-based approaches depend largely on the administrators' experiences. If the administrators need a long time to find out the bottleneck and make the right decision, the web site may experience a long time overload and suffer huge losses. For example, before the administrators of TaoBao2) degraded the product recommendation service on April 21, 2010, the TaoBao system suffered from overload for about 40 minutes and lost thousands of potential transactions worth millions of CNY. Secondly, some built-in degradation-based mechanisms are considered as inherently static, which means that their degradation policies are pre-defined, without considering the different causes of overload. Thus, these "automatic" approaches are inflexible in a way and cannot react adaptively and effectively to some overload situations. For example, some online forums, e.g. the BBS system of Peking University3) and the Shooter forum4), directly shut down their search service without considering what might happen before and during overload. That is because online forum sites' search service usually consumes CPU and memory resources. But when the overload is caused by insufficient bandwidth, just turning off the search service cannot solve the overload problem.

In this paper, we propose a degradation-based mechanism for helping web applications adaptively degrade the services which have low business priority and consume bottleneck resources. Since the degradation plan is made based on both fine-grained resource monitoring and business logic, it captures the relevant factors that overload a system. Thus it can accurately locate the bottleneck and generate a proper solution to make sure that the application can still provide its key services for clients when the requests rate is already higher than the system's capacity. Due to the instantaneous overload detection and subsequent treatment which will be triggered once the degradation plan is made, manual intervention

---

1) Software aging also refers to progressive performance degradation or a sudden hang/crash of a software system due to exhaustion of operating system resources, fragmentation and accumulation of errors. However, different from the service degradation (or performance degradation, as it is more often called) caused by software aging, which is a kind of passive result, the service degradation technology discussed in our paper is an active solution we proposed to handle server overload through adaptively degrading the service which have a lower business priority

2) http://www.taobao.com

3) http://bbs.pku.edu.cn

4) http://www.shooter.cn

is reduced when overload happens. As a case study, we apply our approach to a wiki system and provide evaluations to demonstrate the effectiveness of the approach.

This paper makes the following contributions:

- First, we provide a degradation-based mechanism to control overload. This mechanism takes business logic and the resource-consumption patterns of the application at service-level into consideration when generating a degradation plan, thus making effective and adaptive responses to overload.
- Second, we have designed two algorithms to dynamically and automatically generate degradation plans for different overload situations without delay.
- Third, we implement a prototype of our mechanism and demonstrate our approach through a case study with evaluation results.

The remainder of this paper is organized as follows. Section 2 presents the overview of our approach, and Section 3 introduces decision algorithms. A case study and a prototype implementation are described in Section 4, and we demonstrate, using evaluation results, the effectiveness of our approach in Section 5. Section 6 summarizes the related work and Section 7 concludes this paper and discusses further work.

2 Approach overview

The architecture of web applications is becoming more and more sophisticated nowadays. Taking TaoBao as an example, it has more than 200 sub-applications and thousands of components. However, many overload management approaches focus mainly on the application-level monitoring and adjustment. Such approaches use no knowledge of the actual service-level resource-consumption patterns of an application. For example, a memory-intensive service (or a service with a memory leak) can lead to VM (Virtual Memory) thrashing even with a very low request-admission rate. In many cases, they also do not distinguish which services the requests are sent to and what is wrong within the application when overload happens. For example, some admission control approaches will reject incoming requests when the performance of an online retail site exceeds the threshold and lose some potential transactions [5,6].

Different from these approaches, the degradation object we target in this study is service rather than the whole application. We argue that controlling overload at service-level has a dramatic advantage in that applications can still provide key services when they are faced with overload5).

In service degradation, the immediate question we need to answer is which service is to be degraded? As we mentioned before, most web applications are constructed by many sub-applications and components. Different components are responsible for providing different services to clients, and different services correspond to different priorities from the perspective of business logic. Taking online retail sites for example, the search service is more important for the clients than the product recommendation service. While in the case of an online forum site, reading and writing new messages are more important for the clients than searching for outdated messages when overload occurs. Thus, one of the key problems in service degradation is how to introduce the application-specific business logic into the adaptive system.

In our approach, we mainly consider two aspects, business logic and fine-grained resource utilization monitoring, to automatically generate the degradation plan without delay. The approach architecture is shown in Figure 1.

2.1 Overload detection

Overload detection module is responsible for signaling the occurrence of unstable status of the applications. Through monitoring the key performance metrics of the applications, this module can notify overload controller to make proper degradation actions when the performance metrics exceed the alert value. For different web applications, the definition of performance metrics may be varied.

---

5) Wang Z Y, Zhou M H, Mei H. Towards an adaptive service degration approach for handling server overload. In: The 19th Asia-Pacific Software Engineering Conference (APSEC 2012), Hong Kong, 2012

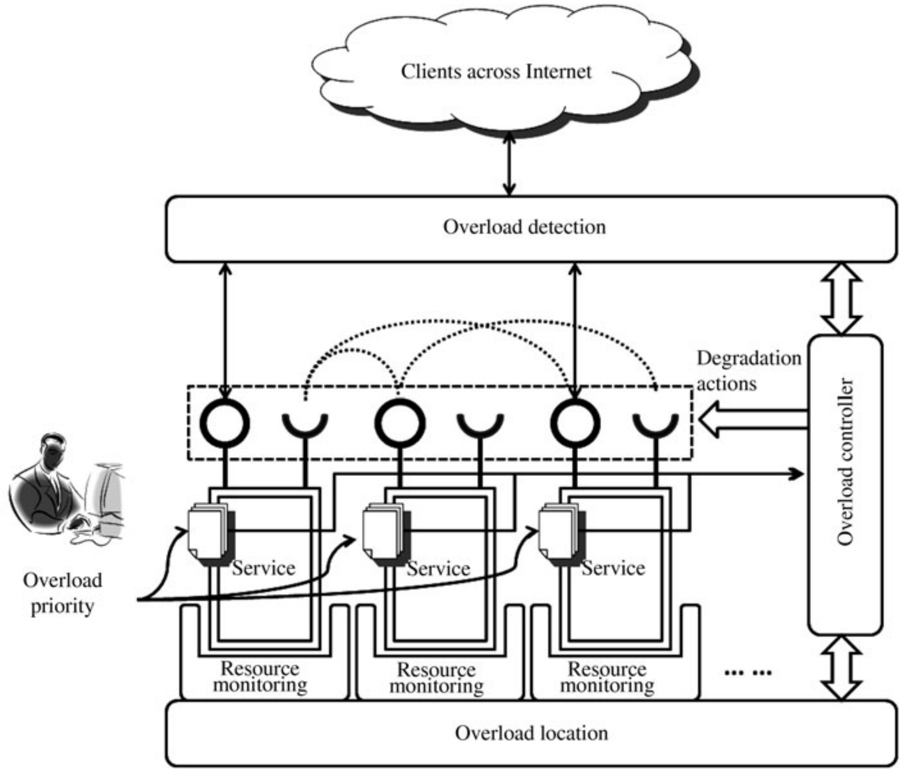


Figure 1 Approach architecture.

Many performance metrics have been studied in the context of overload management, including throughput and response time targets [7,8], CPU utilization [9], and differentiated service metrics, such as the fraction of users in each class that meets a given performance target [10].

Although some researches [11] have shown that the maximum response time is often inevitable, the average response time is used as the most important performance metric in both academic researches and business web sites. We find that the maximum response time cannot represent the real state of the system as it would be unexpectedly large even when the system is in a stable state. The occurrences of the maximum response time also disrupt the calculation of the average response time because it is much larger than the others. Therefore, in this paper, we exclude 10% maximum ones and only focus on 90th-percentile response time as a measure of client-perceived system performance. But we want to note that our approach does not set a limit on the performance metrics to be used to detect overload in other cases.

2.2 Overload location

In our approach, overload location module is triggered to locate the bottleneck of resources and the resource utilization state inside the applications. As shown in Figure 1, web applications are usually composed of many different services. But the most popular web application monitoring tools which are widely used in business web sites today, e.g. JavaMelody6) and JeeObserver7), can only provide the resource utilization state of the whole application. Although these monitoring tools can effectively record most of the running data of the application and notify the administrator when some data increase beyond a pre-computed threshold, they cannot give the administrator a fine-grained view to help him/her find the bottleneck inside the application. In other words, these monitoring tools can only view the application as a black box and show the external state of the application. In that case, when overload occurs, we can roughly know that the application suffers from the shortage of memory or bandwidth, but have no idea about why.

---

6) http://code.google.com/p/javamelody

7) http://www.jeeovserver.com

We propose a fine-grained resource monitoring approach to monitor the resource utilization state of every single service in the application. Actually, it can be more "fine-grained" as it can even monitor the resource utilization state of every single request. It tracks and records the state of most kinds of computing resources not only for an application but also for a service or a request. We also provide a fine-grained monitoring tool based on this approach, which can monitor all kinds of web applications deployed on Tomcat server. Because of page limit, we will not describe the monitoring tool in detail in this paper.

By monitoring the fine-grained resource state, overload location module can provide most kinds of running state data without delay. These data can be used to determine the internal problems of the application when overload occurs. It means that our approach uses "internal" observations of application performance and maintains knowledge of the actual resource-consumption patterns of services in an application.

2.3 Overload priority

Overload priority module defines the priorities of different services and the degradation actions to be taken. As we know, although an application can have hundreds of services, these services are not of the same importance from the perspective of business logic. Taking TaoBao as an example, the administrators will consider two things before they choose a service to degrade: which service is not as important as the others and how many resources will be released if the service is degraded. Although our approach can automatically answer the second question, the first question must be answered by the administrators or the designers of the application.

To illustrate this in more detail, it is known that trade service is a key service in online retail sites, as it helps the customers make deals with the providers. The search service is also a key service in retail sites. Since there are thousands of products on the site, it can quickly find the right thing customers want. Therefore, when overload occurs, although trade and search services may consume more bottleneck resources, they cannot be degraded from the perspective of business logic. On the other hand, product recommendation service is a common service in most online retail sites. When a customer buys a TV, for instance, the product recommendation service may recommend a blue-ray player to this customer. Usually, the content of product recommendation is generated by mining past trading data. Shutting down this service can release some resource without affecting the key functions of the web application.

As to the degradation actions, we argue that they should be varied and service-specific. For a news service, it can adopt content adaptation which means that it can provide lower quality images or text-only messages. But content adaptation is not applicable to many other services due to their design. For example, an e-mail or chat service cannot degrade content in response to overload as "lower-quality" content e-mail and chat have no meaning. Thus, although many degradation actions, such as content degradation [12] and QoS degradation [13], are commonly used, service-specific degradation actions are also needed and welcomed in particular cases. For example, some online forums limit the range of messages to be searched when the server's load is at a high level. This method can be viewed as another choice for search service degradation.

Although service priorities and degradation actions should be provided by the administrators or the designers of the application, we argue that our approach can greatly simplify the priority determination and action allocation process. First, the service priority is defined as a single value in the interval (0, 1). Since this value is considered as an important isolating factor in our decision algorithms described later, administrators can determine these values through load simulation based on the common model of user behavior. Second, most common degradation actions are reusable. For example, rejecting new requests and redirecting requests to another alternative service are widely used by many business applications. We also implement three reusable degradation actions in our prototype described later. We find that the development work is greatly simplified as the degradation actions can be shared by many different services.

2.4 Overload controller

Finally, overload controller module will take appropriate degradation actions to degrade some unnecessary services to release idle resources to support key services. The overload controller can be viewed as an instance of the principle of feedback control. Feedback control is an important model in the engineering of adaptive software, as it defines the behavior of the interactions among the control elements over the adaptation process, to guarantee system properties at run-time. The controller is the key of our approach. It is notified by the overload detection module when overload is coming. Then it will decide what degradation actions should be taken based on the monitoring data from overload location module. We propose two decision algorithms to automatically generate the proper degradation plans. The detailed description of the algorithms is provided in the next section.

3 Decision algorithm

In our approach, decision algorithms are based on fine-grained monitoring data from overload location module and pre-set service priorities from overload priority module to generate the final degradation policies. We propose two decision algorithms to handle different situations. The first algorithm, which is named RUDA (decision algorithm based on resource utilization), mainly focuses on the real time resource utilization. When overload is detected, this algorithm calculates a utility value for every service to determine which service should be degraded. In other words, this algorithm does not care about what happened before the overload but only makes decision based on the data just monitored.

The other algorithm, which is named RPDA (decision algorithm based on request prediction), makes its decision based on a linear model to predict a “future” utility value for every service. Different from many request prediction attempts, this algorithm does not try to predict the number of requests which would arrive in the next minute, but tries to predict the resource utilization state to calculate a utility value for every service.

There are two fundamental differences between the two proposed algorithms. First, the RUDA algorithm is more like a response mechanism than RPDA. It makes a kind of passive response to overload that has emerged. In contrast, the RPDA algorithm is more like a prevention mechanism. It provides a proactive monitoring way to prevent the overload which is likely to happen in a few minutes. Second, the RUDA algorithm runs the monitoring tool only when the overload is detected. It avoids bringing extra overhead to the system when it does not suffer from overload. However, the RPDA algorithm must keep the monitoring tool always running to make sure it has enough data to make the prediction.

Through experiments, we find that both algorithms can effectively handle the pix load which is a common kind of overload. In this kind of situation, the request number increases in a very short time and decreases after a few minutes. We also find that RPDA can work more effectively when the number of requests increases in a slower and more stable way and the system stays in a high load for a longer time.

3.1 RUDA algorithm

The RUDA algorithm uses the following equation to calculate the utility value for every service:

$$f_{nk} = \frac{1}{v_n} \times \frac{r_{nk}}{\sum_{i=1}^{m} r_{ik}} \times \frac{\sum_{j=1}^{m} q_j}{q_n}. \tag{1}$$

In Eq. (1), the value m indicates the number of the services in the application and the parameter n indicates the serial number of the service which is focused on in this equation. The parameter k indicates the serial number of the resource, e.g. CPU, which is focused on in this equation. Since in most real cases, the bottleneck resource is one of all the resources during each overload, we do not consider more than one bottleneck resource in our algorithms right now.

The parameter v_n is the priority value of the service n. For all the services, the value of v should be in the interval (0, 1), and the sum of all the v should be equal to 1. As mentioned in Subsection 2.3, these values are assigned by the administrators of the application. It indicates the degree of importance of different services. A higher v value means that the service is more important.

The parameter r_{nk} is the number of resources k consumed by the service n. The parameter q_n indicates the number of requests received by the service n. Thus, the utility value of service n for resource k, f_{nk}, is composed by three parts: the inverse of v_n, the percentage of the resource k consumed by this service and the inverse of the percentage of the request received by this service. To put it another way, the service which has a higher utility value may have a lower priority, consume a higher percentage resource and receive a lower percentage request. Thus, the RUDA algorithm can simply be described by the pseudo codes below:

```
Algorithm: Decision algorithm based on resource utilization (RUDA)
Input: none
output: a service number $p$
Let $S$ be the set of services
for all $s$ belongs to $S$ do
    calculate the $f_{nk}$ for $s$
    let $p$ be the $s$ which has the biggest $f_{nk}$
end for
return $p$
```

Through calculating the f_{nk} for every service, the RUDA algorithm can choose the service which has the highest utility value to degrade. Therefore, the time complexity of this algorithm is $O(m)$ where m is the number of services. We want to emphasize that all the data used in the RUDA algorithm coming from the overload location module are just monitored in the right moment when the overload is detected. Its advantage is that it needs not to keep the monitoring tool running all the way and can avoid affecting the performance of the application which is running in the normal state.

3.2 RPDA algorithm

The RPDA algorithm is based on a linear model to predict the resource utilization state for every service. It assumes that the number of requests will linearly increase in a few minutes when overload happens. As RUDA does, it also uses an equation to calculate a "future" utility value for each service, i.e. given by

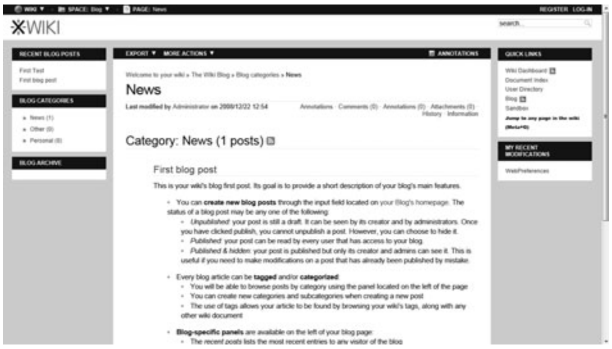
$$l_{nk} = \frac{r_{nk.t} - r_{nk.(t-1)}}{q_{n.t} - q_{n.(t-1)}} * (q_{n.(t+1)} - q_{n.t}), \tag{2}$$

$$f'_{nk} = \frac{1}{v_n} * \frac{l_{nk}}{\sum_{i=1}^{m} l_{ik}}. \tag{3}$$

The same as Eq. (1), the value m indicates the number of services in the application and the parameter n indicates the serial number of the service which is focused on in this equation. The value t presents the timestamp when the monitoring data is recorded. The value $t-1$ and $t+1$ indicate the timestamps just before t and after t respectively.

The parameter $r_{nk.t}$ represents the amount of resource k consumed by the service n at time t. The parameter $q_{n.t}$ indicates the number of requests received by the service n at time t. Since the change of requests is viewed as a linear function during a short period, $q_{n.(t+1)}$ is a predicted value which stands for the number of requests sent to service n at the next timestamp. Thus, Eq. (2) predicts the increment of the resource k to be consumed by service n at the next timestamp.

In Eq. (3), the parameter v_n has the same meaning and value as in Eq. (1). Then the utility value in Eq. (3) is composed of two parts: the inverse of v_n and the percentage of the increment of the resource k at the next timestamp. To explain this equation in another way, the service which has a higher utility value may have a lower priority and may consume a higher percentage resource in the next stage. Thus, the RPDA algorithm can simply be described by the pseudo codes below: Similar with RUDA algorithm, the services are chosen according to their utility values. The RPDA algorithm will choose the service



Welcome to your wiki » The Wiki Blog » Blog categories » **News**

News

Last modified by Administrator **on 2009/12/22 12:54**

Annotations · Comments (0) · Annotations (0) · Attachments (0) · History · Information

Category: News (1 posts)

First blog post

This is your wiki's blog first post. Its goal is to provide a short description of your blog's main features.

- You can **create new blog posts** through the input field located on your Blog's homepage. The status of a blog post may be any one of the following:
  - *Unpublished*: your post is still a draft. It can be seen by its creator and by administrators. Once you have clicked publish, you cannot unpublish a post. However, you can choose to hide it.
  - *Published*: your post can be read by every user that has access to your blog.
  - *Published & hidden*: your post is published but only its creator and admins can see it. This is useful if you need to make modifications on a post that has already been published by mistake.
- Every blog article can be **tagged** and/or **categorized**:
  - You will be able to browse posts by category using the panel located on the left of the page
  - You can create new categories and subcategories when creating a new post
  - The use of tags allows your article to be found by browsing your wiki's tags, along with any other wiki document
- **Blog-specific panels** are available on the left of your blog page:
  - The recent posts lists the most recent entries to any visitor of the blog

Figure 2 Screenshot of XWiki.

which has the highest utility value to degrade. Therefore, the time complexity of this algorithm is $O(m)$ where m is the number of services.

Since the RPDA algorithm needs the history monitoring data to make its decision, its overhead brought by monitoring is higher than the RUDA algorithm. Although the overhead is within a reasonable range in our evaluation, the performance of application which is running in the normal state is affected.

4 Prototype implementation and case study

Currently, we have implemented a proof-of-concept prototype in Java. The prototype can be deployed on the Tomcat web server and works for most applications which can be deployed on Tomcat. In this prototype, we extend the web monitoring tool JavaMelody to implement the fine-grained resource monitoring. The tool can monitor requests, CPU, memory, bandwidth and database connection at service-level, and more functions will be added in future versions.

The prototype has some internal degradation policies. The first one is blocking requests, which is often used in admission control schemes. It simply rejects all the requests to the chosen service and redirect the requests to a friendly notice page, just as the BBS system of Peking University does. The second one is redirecting requests to a pre-set link. It can be used to redirect the requests from a complex and heavyweight service to a simple and lightweight service to ease the load on the server. For example, CNN's administrators manually changed their home page to a simple and text-only page to overcome the heavy load on September 11, 2001 [14]. The third one is session-based and will keep the sessions already connected to the chosen service and reject the new coming sessions. This way avoids affecting the clients already using the service and degrades the service in a more gentle way. We find that these seemingly simple policies are widely used in most real service degradation cases. But we want to indicate that there may not exist a way suitable for every application, and the administrators may need to provide their application-specific degradation policies for different situations.

To demonstrate our approach, we choose a business application in our case study. XWiki Enterprise is a professional wiki that has powerful extensibility features such as scripting in pages, plugins and a highly modular architecture. A screenshot of the XWiki instance is shown in Figure 2.

We choose this application in our case study for two reasons. First, XWiki software is developed in Java and under the LGPL open source license. It is easy for us to get all the source code of the application to understand its service-level architecture. Second, this application has a highly modular architecture which means different services are wrapped in different modules and if we degrade one service, the other services will not be affected.

After a careful analysis, we choose five services from the XWiki platform in this case study. The most common services are view, edit and management. We also choose another two assistant services, i.e. search and export, in our case study. XWiki integrates the Lucene[8] search engine as its search service. It provides an easy way for visitors to find exactly what they want to see. The export service can be used to export the documents in many file formats, such as PDF, DOC and XML.

---

[8] http://lucene.apache.org/

Table 1 Priorities for different services

Service	View	Edit	Management	Search	Export
Priority value	0.5	0.2	0.2	0.05	0.05

Table 2 Requests probabilities

Service	View	Edit	Management	Search	Export
Probability	75%	15%	2%	7%	1%

Finally, as we mentioned in Subsection 2.3, we manually give these five services a priority value respectively according to their different importance to visitors. We divide the priorities into three levels: 0.5, 0.2 and 0.05. Among these services, we set the view service to the highest priority, i.e. 0.5. Since a wiki system is somewhat like a news site, providing useful information is viewed as the heart of the application. As the second level, edit and management services' priorities are set to 0.2 for their importance. Since the search and export services are assistant functions in a wiki system, they are set to the lowest priority, i.e. 0.05. The sum of all these priorities is equal to 1.

The degradation actions are also pre-set manually. The view, search and export services are viewed as stateless services, which means visitors do not login nor build a session with the application. The degradation action is set to block requests for these three services. For the edit and management services, visitors or administrators must log in to use their functions for security reasons. The degradation action is session-based and will keep the sessions already connected to the chosen service and reject the new coming sessions. It avoids shutting down the service for the clients who have already begun editing a page and will continually provide service until they log out. Table 1 shows the priorities for different services.

5 Evaluation

To evaluate the effectiveness of our approach, we apply the approach to the complex, realistic application XWiki under extreme load situations. The experimental platform is constructed by three PCs which are connected by 1000 Mbit Ethernet in a LAN. Every machine runs Windows 7 with a 2.8 GHz Intel CPU and 3 GB of RAM. An instance of Tomcat version 6.0.29 is running on one of them, and we also deploy our prototype and an XWiki enterprise 2.0 instance on the Tomcat instance. Another PC is running an instance of Mysql server 5.1 for providing database access needed by XWiki. Finally, we deploy the application-specific client load simulation tool on the last PC.

The client load simulation tool used in our experiments emulates a number of simultaneous clients, each accessing a single service at each time. Simulated clients access the services based on a simple model of user behavior. In this model, the clients will randomly choose a service from the five services to visit, and the probabilities of choosing each service are listed in Table 2. This model of user behavior is derived from traces of the wiki system used in our research team. The inter-request time is aggressively set to 100 ms. Since we are only interested in the overload behavior of the server, WAN network effects are not incorporated into our evaluation.

5.1 RUDA algorithm with increased user load

Figure 3 shows the 90th-percentile response time for the XWiki system with and without the RUDA degradation enabled. The 90th-percentile response time threshold is set to 2 seconds. For each data point, the corresponding number of simulated clients load the system for about 15 minutes, and response-time distributions are collected after an initial warm-up period of about 1 minute. As the figure shows, the RUDA degradation mechanism is effective at meeting the response time target despite a many-fold increase in load.

The internal mechanism of the degradation is that when clients increase in number, the CPU utilization rate increases more quickly than the other resources. Since the 90th-percentile response time also increases for lack of CPU time, the RUDA degradation mechanism is triggered. Based on the RUDA algorithm,

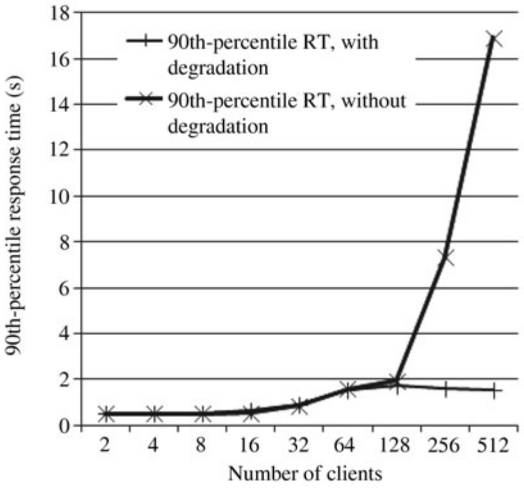


Figure 3 RUDA degradation in XWiki system.

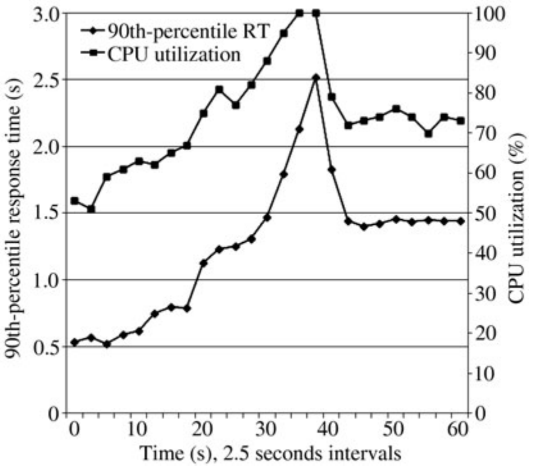


Figure 4 RUDA degradation under an increased load.

the search service is chosen to be degraded when the number of clients is around 128. After the Search service stop serving new requests, the trend of average response time becomes more stable.

To demonstrate the effectiveness of the RUDA degradation mechanism more intuitively, Figure 4 shows the 90th-percentile response time and CPU utilization state for the XWiki system with the RUDA degradation enabled. This figure shows the operation of the RUDA degradation in the XWiki during a linear load increase. The number of simulated clients increases by 4 in every second. The response time threshold is set to 2 seconds. The CPU utilization comes to 100% around $t = 36$, while the 90th-percentile response time also exceeds 2 seconds and reaches around 2.5 seconds. The search service is degraded. Notice that the overload lasts for about 6 seconds. As we mentioned in Subsection 3.1, the RUDA algorithm only needs the real-time data and avoids continually monitoring the system state as the RPDA algorithm does, and thus it takes some time to wake the monitoring tool and wait for the monitoring data.

5.2 RUDA algorithm under a massive load spike

The previous section evaluated the RUDA algorithm under a steadily increasing user load, representing a slow increase in user population over time. We are also interested in evaluating the effectiveness of this method under a sudden load spike. In this scenario, we start with a base load of 20 clients accessing the XWiki system, and suddenly increase the load to 200 clients. This is meant to model a “flash crowd” in which a large number of clients access the service all at once.

Figure 5 shows the performance of the RUDA service degradation in this situation. The load spike comes when the time is around 20, and ends when the time is around 40. Without service degradation, there is an enormous increase in response times during the load spike, making the service practically unusable for all clients. With the RUDA service degradation and a 90th-percentile response time target of 2 seconds, most of the response time for all requests is kept around 1.5 seconds after the Search service is degraded. Notice the response time with degradation exceeds the threshold for about 5 seconds, during which time the degradation system wakes the monitoring tool and makes the degradation actions.

5.3 RPDA algorithm with increased user load

To demonstrate the effective of the RPDA degradation mechanism, Figure 6 shows the 90th-percentile response time and CPU utilization state for the XWiki system with the RPDA degradation enabled. This figure shows the operation of the RPDA degradation during a linear load increase. The number of simulated clients increases by 4 in every second. The response time threshold is set to 2 seconds. Notice that the 90th-percentile response time never exceeds the threshold, because the RPDA algorithm can predict the resource utilization state for all the services, it begins to degrade the Search service just be-

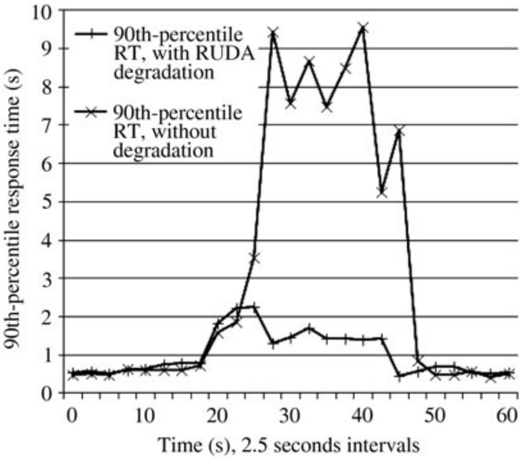


Figure 5 RUDA degradation under a massive load spike.

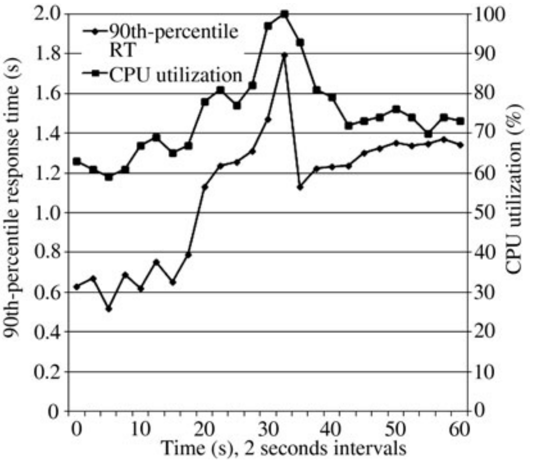


Figure 6 RPDA degradation under an increased load.

fore the overload really happens. Although the CPU utilization comes to 100% around $t = 30$, the 90th-percentil response time reaches 1.8 and quickly decreases in the next timestamp. Comparing with Subsection 5.1, RPDA algorithm avoids the application's experience of overload in the similar scenario. But the RPDA algorithm needs to continually monitor the system state when the system is running. Thus it brings more overhead for the system, as we will see in the Subsection 5.6.

5.4 RPDA algorithm under a massive load spike

To evaluate the effectiveness of the RPDA method under a sudden load spike, we repeat the scenario in Subsection 5.2. Again, we start with a base load of 20 clients, and suddenly increase the load to 200 clients. Figure 7 shows the 90th-percentile response time experienced by clients using the XWiki system under a massive load spike. Without degradation, response times grow without bound; with RPDA degradation (using a 90th-percentile response time target of 2 seconds), response time quickly becomes stable after a small increase during overload.

The load spike comes when the time is around 20, and ends when the time is around 40. When the system is running without overload, the response times with RPDA degradation are slightly higher than the response times without degradation. It is because that the running of the monitoring tool influences the system's performance. This influence is also showed in response times. Although the RPDA algorithm can predict the linear load increase, it still needs time to react to a sudden increase of clients. The response time with degradation exceeds the threshold for about 3 seconds before the Search service is degraded.

5.5 RUDA & RPDA with a different user behavior model

As we mentioned before, web applications nowadays are running on an open, dynamic and rapidly changing environment. User behaviors are one of the key changing components of the environment that are difficult to predict. This is an important reason why our approach focuses on flexibility and adaptability, trying to handle overloads caused by different problems. Therefore, we are also interested in evaluating the effectiveness of our approach under a different load model.

In this scenario, simulated clients access the services based on a different model of user behavior from the previous one. The new model is deliberately captured from traces of the wiki system used in our research team in a short period. The clients will randomly choose a service from the five services to visit, and the probabilities of choosing each service are listed in Table 3. The inter-request time is also set to 100 ms. The number of simulated clients is increased by 3 in every second. The response threshold is set to 2 seconds.

Figure 8 shows the 90th-percentile response time and memory state for the XWiki system with the RUDA degradation enabled. We find that the system experiences a long time overload. Although the

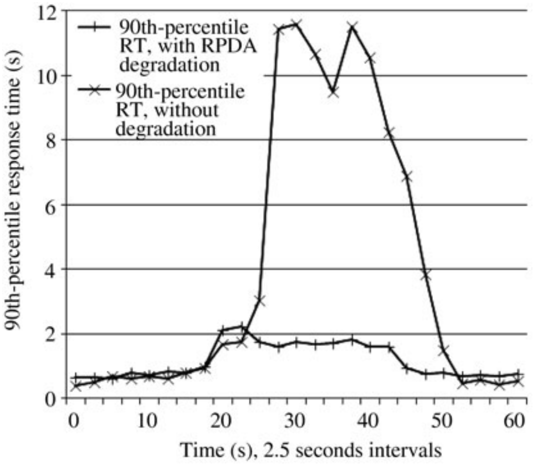


Figure 7 RPDA degradation under a massive load spike.

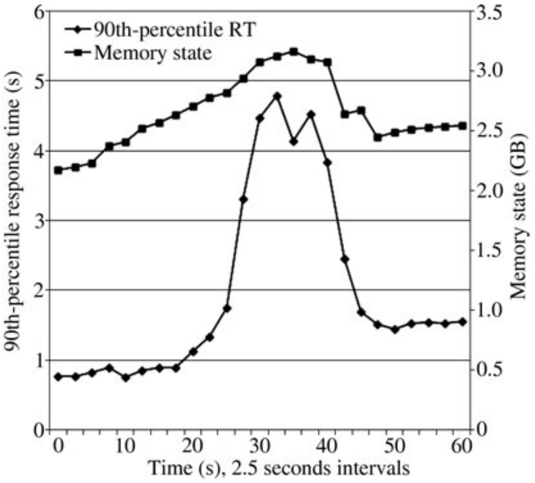


Figure 8 RUDA degradation under an increased load.

Table 3 Changed requests probabilities

Service	View	Edit	Management	Search	Export
Probability	68%	20%	2%	2%	8%

system log shows that the degradation mechanism degrades the export service which consumes much more memory than the other services do around $t = 30$, the system still suffers from the VM (virtual memory) thrashing problem. Besides the fact that the system needs more time to handle requests under processing, we find that the garbage collection mechanism of the JVM (Java virtual machine) is another important reason why releasing the idle memory is delayed.

Figure 9 shows the 90th-percentile response time and memory state for the XWiki system with the RPDA degradation enabled. Although the RPDA degradation performs better than the RUDA degradation in the same situation as it can make the degradation actions a little ealier, the response time still exceeds the threshold and even reaches 4.5 seconds. Since the delayed effects of garbage collection mechanism makes it hard to immediately recover the system from overload to a stable state, how to improve the effectiveness of our approach in this situation is still a problem to be solved in the next stage of our research.

5.6 The overhead of the approach

Since our approach needs to introduce an extra mechanism into the applications, we also want to evaluate the overhead brought by this mechanism. In this scenario, we do not consider the overload situation and suppose the applications are always running in the normal state.

Figure 10 shows the 90th-percentile response time for the XWiki system with and without degradation enabled. For each data point, the corresponding number of simulated clients loads the system for about 15 minutes, and response time distributions are collected after an initial warm-up period of about 1 minute. As shown in the figure, the response time with RUDA degradation roughly coincides with the response time without any degradation mechanism. But the response time with RPDA degradation is higher than them from 0.05 second to 0.15 second in each stage. And the overhead mainly keeps stable while the number of clients increases over 4. As the RPDA algorithm keeps the monitoring tool running to collect data, it needs a little time to process the collecting work for every request and results in a longer response time.

5.7 Threat to validity

According to our experiences, the biggest threat is that some real-world applications are not designed as clear modules or components. Thus it is not easy to integrate the monitoring and controlling processes

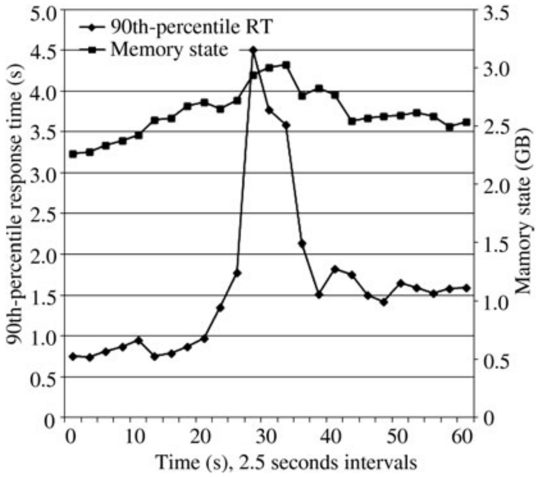


Figure 9 RPDA degradation under an increased load.

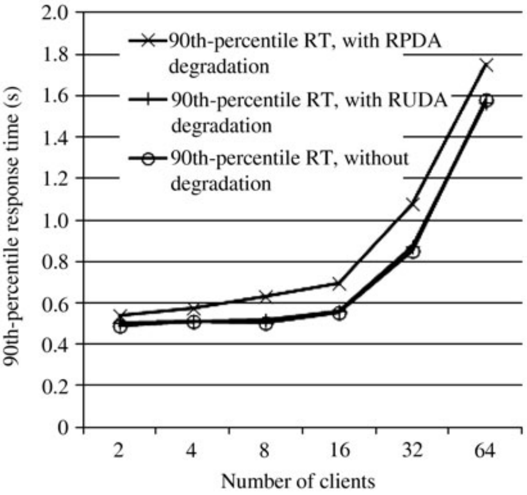


Figure 10 Approach overhead.

of the services. Great efforts are needed to overcome this. First, efforts must be maid to understand the existing system. It is necessary for developers to fully understand the existing system before integrating the extended mechanism. It could be challenging if there were few or deficient software documents to explain the structure of the system. Second, the effectiveness of the mechanism can be affected in some cases. For example, the complicated invocations between two modules will make it hard to find out where and how to control the services. Another threat is that the overload patterns in real life are varied. Although we have considered the two most common cases, massive load spike and increased user load, we can not guarantee the effectiveness of our mechanism in all overload conditions.

6 Related work

Content adaptation [12,15,16] is an initial approach for implementing service degradation. It serves to reduce the quality of static web content (e.g. lower resolution images) provided to clients during overload, consuming less system resources in this way. In particular, content adaptation mostly aims to reduce the network bandwidth consumption.

Abdelzaher et al. [12] evaluate the potential for content adaptation on the web using three techniques: image degradation by lossy compression, reduction of embedded objects per page, and reduction in local links. They also present a solution to the overload problem in web servers, which relies on adapting the delivered content. Informed transcoding techniques [15] are also used to provide differentiated service and to dynamically allocate available bandwidth among different client classes, while delivering a high degree of information content (quality factor) for all clients. This system will serve proportionately lower quality variations of images if the average consumed bandwidth for the past half hour is more than the target bandwidth.

Another example of service degradation is replacing web pages with many images and links to other expensive objects with simplified web pages. This was the approach taken by CNN.com on September 11, 2001 [14]. In response to overload, CNN replaced its front page with simple HTML page that could be transmitted in a single Ethernet packet. This technique was implemented manually, though a better approach would be to degrade service gracefully and automatically in response to overload.

But with the development of Internet technology, web applications that provide only static web content no longer play the leading role in mainstream use today. These early degradation techniques based on static web content cannot fully meet the overload challenges brought by the current web applications which provide mainly dynamic services. Compared with these techniques, our mechanism can give a better overload management assistance to applications which provide mainly dynamic services. Besides, it is not specific to a particular application or service, and can be more widely adapted to different situations. However, these application-specific and service-specific degradation techniques can still be combined with our approach to provide better degradation choices under different overload situations.

Brewer et al. [17] propose an approach to make performance tradeoffs in terms of the freshness, consistency, or completeness of data delivered to clients. This tradeoff in terms of the harvest and yield of a data operation; harvest refers to the amount of data represented in a response, while yield (closely related to availability) refers to the probability of completing a request. For example, a web search engine could reduce the amount of the web database searched when overloaded, and still produce results that are good enough such that a user may not notice any difference.

Some other works [13,18] have reinterpreted the concept of service degradation. For instance, Urgaonkar et al. [13] implement what they call "QoS adaptive degradation". This approach considers that during overload conditions, the performance of admitted requests can be degraded within the limits established by the SLA. The same concept is applied in [18], but is named "QoS adaptation". Compared with our approach, these approaches are more concerned about how to degrade the quality of the service itself rather than the combination of services in an application.

Other works do not directly implement service degradation mechanisms, but rather signal overload to applications in a way that allows them to degrade if possible. For example, in SEDA [19], stages can enable or disable the stage's admission control mechanism. This allows a service to implement degradation by periodically sampling the current response time and comparing it to the target. If service degradation is ineffective, because the load is too high to support even the lowest quality setting, the stage can re-enable admission control to reject requests.

Although these approaches do not concentrate on the technique of service degradation, they combine it with other overload management techniques, such as admission control, to provide a synthetic overload management framework. Through a proper combination, the other techniques can give full play to the advantages of service degradation while complementing its disadvantages.

7 Conclusion and future work

In this paper, we present an adaptive service degradation mechanism for handling extreme overloads for complex web applications. In this approach, we propose a fine-grained resource monitoring tool to monitor the resource utilization states at service-level at runtime. Interfaces are provided for the designers and administrators of the applications to describe the priorities and the degradation actions for different services inside the applications. Based on the fine-grained monitoring data and the application-specific configuration, we propose two decision algorithms to automatically generate the service degradation plan at runtime to help applications handle overload. In our approach, the bottleneck which causes the overload will be quickly located and the corresponding solution will be adaptively changed for different overload situations. We have presented a prototype implementation of this approach and used a complex business application to evaluate it. The results have shown that our approach is effective for managing load with increasing user populations as well as under massive load spikes. This paper also provides a feasible approach to cope with the evolution and control problems of trustworthy software under overload.

We argue that service degradation is only one kind of overload management techniques. Its advantage is avoiding rejecting clients as a response to overload and still keeping key services of the applications for clients. Service degradation can be considered as a complementary technique to enhance a given performance management solution. Using it alone may delay the occurrence of overload, but it needs to be combined with other techniques to be fully effective [4]. For example, if service degradation is ineffective (say, because the load is too high to support even the lowest quality setting), the administrators can enable admission control to reject extra requests. Therefore, finding a more effective way to prevent overload for web applications through a combination of our approach and other techniques while providing performance guarantees will be one aspect of our future work.

We have dealt primarily with load management within a single node, though large-scale web applications are typically constructed with workstation clusters to achieve incremental scalability and fault tolerance. So another aspect of our future work would be to explore the approach in a clustered environment.

Acknowledgements

This work was supported by National Basic Research Program of China (973 Program) (Grant No. 2009CB320703), National Natural Science Foundation of China (Grant Nos. 91118004, 61073016) and National High-tech Research and Development Program of China (Grant No. 2012AA011202).

References

1 Liu K, Shan Z G, Wang J, et al. Overview on major research plan of trustworthy software (in Chinese). Bull Natl Nat Sci Found China, 2008, 3: 145–151

2 Wang H M, Tang Y B, Yin G, et al. The trustworthiness of the Internet software (in Chinese). Sci China Ser E-Tech Sci, 2006, 36: 1156–1169

3 Schroeder B, Harchol-Balter M. Web servers under overload: How scheduling can help. ACM Trans Internet Technol, 2006, 6: 20–52

4 Guitart J, Torres J, Ayguad E. A survey on performance management for Internet applications. Concurr Comput Pract Exp, 2010, 22: 68–106

5 Elnikety S, Nahum E, Tracey J, et al. Method for transparent admission control and request scheduling in e-commerce web sites. In: 13th International Conference on World Wide Web (WWW'04), New York, 2004. 276–286

6 Iyer R, Tewari V, Kant K. Overload control mechanisms for web servers. In: Workshop on Performance and QoS of Next Generation Networks, Nagoya, 2000. 225–244

7 Chen X, Chen H, Mohapatra P. An admission control scheme for predictable server response time for web accesses. In: Proceedings of the 10th World Wide Web Conference (WWW'01), Hong Kong, 2001. 545–554

8 Chen H, Mohapatra P. Session-based overload control in QoS-aware web servers. In: Proceedings of IEEE INFOCOM 2002, New York, 2002. 516–524

9 Diao Y, Gandhi N, Hellerstein J, et al. Using MIMO feedback control to enforce policies for interrelated metrics with application to the Apache Web server. In: Proceedings of the Network Operations and Management Symposium 2002, Florence, 2002. 219–234

10 Guitart J, Carrera D, Beltran V, et al. Designing an overload control strategy for secure e-commerce applications. Comput Netw, 2007, 51: 4492–4510

11 Welsh M, Culler D. Adaptive overload control for busy Internet servers. In: Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems–Volume 4 (USITS'03). Berkeley: USENIX Association, 2003. 43–57

12 Abdelzaher T, Bhatti N. Web content adaptation to improve server overload behavior. Comput Netw, 1999, 31: 1563–1577

13 Urgaonkar B, Shenoy P. Cataclysm: Scalable overload policing for Internet applications. J Netw Comput Appl (JNCA), 2008, 31: 891–920

14 LeFebvre W. CNN.com: Facing a world crisis. In: Invited Talk at USENIX LISA'01, 2001

15 Chandra S, Ellis C S, Vahdat A. Differentiated multimedia web services using quality aware transcoding. In: Proceedings of IEEE INFOCOM, 2000. 961–969

16 Fox A, Gribble S D, Chawathe Y, et al. Cluster-based scalable network services. In: Proceedings of the 16th ACM Symposium on Operating Systems Principles, St.-Malo, 1997. 78–91

17 Fox A, Brewer E A. Harvest, yield and scalable tolerant systems. In: Proceedings of the 1999 Workshop on Hot Topics in Operating Systems, Rio Rico, 1999. 174–178

18 Abdelzaher T, Shin K, Bhatti N. Performance guarantees for web server end-systems: A control-theoretical approach. IEEE Trans Parallel Distrib Syst, 2002, 13: 80–96

19 Welsh M, Culler D. Adaptive overload control for busy internet servers. In: 4th Symposium on Internet Technologies and Systems (USITS'03), Seattle, 2003. 43–57