Distributed Workload and Response Time Management for Web Applications

Shengzhi Zhang*, Haishan Wu†, Wenjie Wang‡, Bo Yang†, Peng Liu*, Athanasios V. Vasilakos§
*The Penn State University, University Park, PA, USA
†IBM Research - China, Beijing, China
‡Shanghai Synacast Media Tech. (PPLive) Inc. China
§National Technical University of Athens, Greece
Email: suz116@psu.edu, wuhais@cn.ibm.com, wenjiewang@pplive.com,
boyang@cn.ibm.com, pliu@ist.psu.edu, vasilako@ath.forthnet.gr

***Abstract*—Managing workload for large scale web applications is a fundamental task for satisfactory quality of service, low management and operation cost. In this paper, we present *SCOPS*, a system of distributed workload management to achieve service differentiation and overload protection in such large scale deployment. Our system splits the workload management logic into distributed components on each back-end server and front-end proxy. The control solution is designed to protect the back-end server from overloading and to achieve both efficient usage of system resource and service differentiation by employing a unique optimization target. The control components are automatically organized based on the flow of workloads, such that management overhead is minimized. *SCOPS* is extremely flexible because it requires no source code changes to host OS, application servers, or web applications. Additionally, the distributed design makes it scalable and robust for cloud scale server deployment. Experiments with our implementation confirm *SCOPS*'s performance with dynamic heavy workload, incurring neglectable runtime overhead. More importantly, *SCOPS* also ensures fault-tolerance and fast convergence to system failures.**

***Keywords*-Admission control; service class differentiation; distributed computing; cloud**

I. Introduction

With the rapid expansion of cloud offerings, more applications are hosted in clouds for the benefit of scalability and cost saving. Thus, managing workload with satisfactory quality of service for large scale server deployment becomes a fundamental task to reduce management and operation cost. Large scale server farm usually consists of multiple tiers of servers, e.g., HTTP servers in the front tier, proxy servers in the middle tier, and application servers (usually with database servers) in the back-end tier. The HTTP server tier filters out invalid/malicious requests and forwards legitimate ones to the proxy tier, which in turn routes these requests to the corresponding application servers. In this paper, we focus on the workload management between the proxy tier and back-end server tier.

Workload management has been widely studied in the literature of application servers load balancing, where centralized controllers have been used to manipulate the traffic going through proxies [8], [15], [22], [11], and [5]. A central "hub" collects and audits the workload arrival rate and processing time, then regulates the workload queuing time and forwarding path accordingly. In some cases, the management decision is made in a centralized fashion, but executed distributively by a number of proxy-tier servers. A common practice to improve the scalability of such centralized solutions is to group proxies and application servers into small clusters, which unfortunately increases management overhead and causes resource under-utilization in case of load imbalance among clusters.

In this paper, we propose a fully decentralized workload management system, named *SCOPS* (Sub-Controller On Proxies and Servers), to achieve load balancing, overload protection, and QoS differentiation in cloud-based server farms. *SCOPS* splits the controlling functionality into two sets of components, and distributes them among all involved servers. Specifically, a sub-controller on each back-end server monitors local resource usage and applies an application-characteristics-free approach to estimate the maximum request rate that this server may handle. Then it dynamically allocates the maximum rate among control components located on the proxy tier (named *proxy dispatcher*) as *quotas*. Based on the limited quotas, each proxy dispatcher accordingly controls workload of different priorities using a carefully-designed optimization target that concerns both resource utilization maximization and service differentiation.

In summary, we make the following contributions:

1) An effective distributed controlling framework. The proposed design is based on quite some realistic deployment scenarios. It is simple and effective to meet the scalability and robustness requirements of large scale server deployment, e.g., clouds.

2) Light-weight and adaptive controllers. Our solution involves no changes to the source code of web applications, server software, nor operating systems, thus easing deployment effort. Our approach avoids accurate modelling or deep instrumentation of the controlled system, and produces a very adaptive solution. Furthermore, our evaluation shows that the management overhead is negligible for large scale server deployment.

3) Implementation and evaluation of the system. We have implemented the whole *SCOPS* solution and evaluated it in

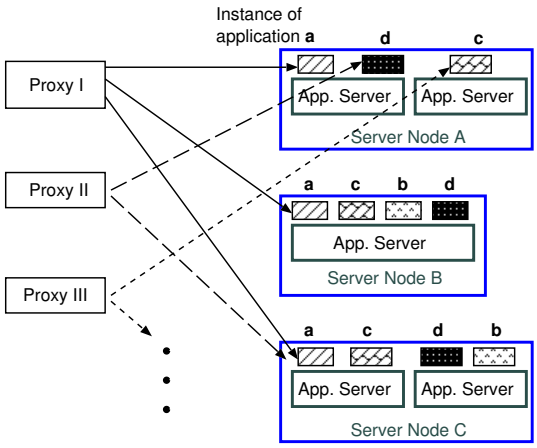

Fig. 1. An example of Workload Distribution and Resource Sharing in Cloud

a heterogeneous server environment. We confirm that overload protection and service classification can be achieved effectively with our distributed design. The results show that *SCOPS* is robust and reliable, which can tolerate system component failures very well.

The rest of this paper is organized as follows. We introduce the server deployment scenario and overview *SCOPS* system in Section II. The details of our control components are presented in Section III and Section IV, respectively. We show our performance evaluation in Section V and discuss the related work in Section VI. Section VII concludes the paper.

II. System and Design Overview

In this section, we start with the description of general application server deployment scenario. Then we present the overview of our decentralized design, with control functionality spreading among sub-controllers and proxies. Last, we describe the extendibility of *SCOPS*.

A. Scenarios and Requirements

Resource sharing is a common practice for efficient usage of cloud facilities. We recognize three levels of resource sharing in general application server deployment: *application instance sharing, application server sharing*, and *server node sharing*, sorted from the top application level to the bottom physical level. That is, multiple application servers (e.g., JVM inside one virtual machine) could run on a physical server (server node sharing), while different application instances[1] are running in an application server (application server sharing). The sharing of upper level resource also indicates the sharing of lower level resource.

Fig. 1 shows a typical example of the recognized resource sharing in general application server deployment, which we determine as our design scenario. The proxy dispatcher serves as a router, forwarding service requests[2] to the corresponding application instances. An application instance can be "shared" by several proxy dispatchers, for instance, both Proxy I and Proxy II can route workload to the application instance a on server node C. Additionally, Proxy I and Proxy II share the same application server on server node A, and proxy II and III share the same physical server A.

There is resource competition even within the same resource sharing level. For example, the requests of the same application may have different quality of service (QoS) requirements, usually stated in the Service Level Agreement (SLA). In this paper, QoS requirements are defined to be the combination of preferential fairness and average response time. The shared resources are more likely to be given to workload with high priority.

The combination of preferential fairness and average response time reflects the management principles of QoS assurance in practice, particularly for the enterprise private cloud. Instead of dedicating abundant amount of resources to the high priority workload and starving the low priority one, certain degree of degradation in response time can be tolerated in exchange for continuous service among all workload. The service providers configure the degree of tolerance by means of business importance for different workload.

B. Design Overview

We propose a decentralized workload management solution named *SCOPS* with three key functionalities: load balancing, overload protection, and QoS differentiation. For both scalability and fault tolerance, *SCOPS* spreads the resource management and traffic control functionalities among a set of control components located on each system component. The connection and collaboration among these control components are designed to be simple and "soft", i.e., without tight integration or dependency, such that failure of one component does not severely affect the others. We define two control roles in our system: *sub-controller* attached to each back-end server and *proxy dispatcher* attached to each front-end proxy.

1) Sub-controller: The design of sub-controller follows a few principles, locality and externality. The sub-controller only measures local resource consumption of the back-end server. It is dedicated to protect local server from overloading solely based on local information. This locality policy avoids the pitfalls of scalability and efficiency problems that may be introduced by remote communication and external dependency.

The sub-controller runs externally outside the application server, either in virtual machines or physical servers. No integration with legacy applications greatly improves the flexibility of the controller and eases its deployment. In our implementation, we prototype the sub-controller as a standalone, server startup script. It automatically joins the server farm and starts managing the resources of the local servers. The sub-controller employs an improved Proportional-Integral controller, to periodically calculate and adjust the maximum allowed request rate that a server can support. With this adaptive method that the maximum rate is adjusted based on internal feedback loops, the sub-controller does not have

[1]We refer to the presence of an application on an application server as an application instance.

[2]We use the term *workload* and *request* interchangeably based on context.

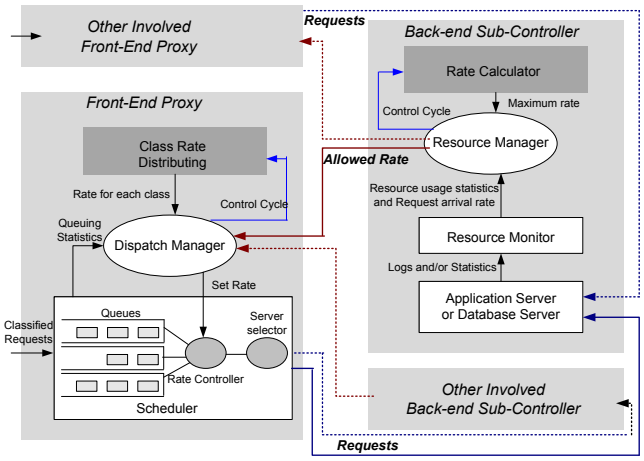

Fig. 2. System Diagram

to spend time learning the exact load characteristics of the workload in an accurate model.

2) Proxy dispatcher: The proxy dispatcher regulates the requests to each back-end server based on the resource quota allocated from the corresponding sub-controller. In our design, the proxy dispatcher only receives resource quotas from the sub-controllers that it directly forwards workload to. This greatly reduces the communication traffic among our system. The dispatcher also makes the rate regulation decision among different service classes only based on the resource quota received from these limited number of controllers. This makes the system quite robust and extremely resilient to component failures.

In order to assure service differentiation and QoS, the proxy dispatcher consists of two major components, a token-bucket-based queuing model and a system optimization model with a penalty-based utility function. The utility function allows the proxy dispatcher to effectively map the importances of service classes to their dispatch rate. The queuing model associates the dispatch rate of different service classes to their QoS in terms of average response time.

3) Control Component Communication: Fig. 2 illustrates the collaboration between sub-controllers and proxies to achieve admission control and service classification at runtime. The sub-controllers dynamically distribute the available capacity based on local information and the request profiles from the proxies. Meanwhile, the proxies that share resources on this particular server node (by means of forwarding requests to the application instances running on this server) send back their own request arrival rates and queuing profiles at the end of each control cycle. The sub-controller automatically learns the existence of proxies and distributes the rate quota among them. The proxies respect the rate quota received from each sub-controller, thus achieving the overall traffic control during overload in the cloud.

4) Extendibility: There are two key advantages of *SCOPS* framework in its extendibility. One is that it can be easily applied to any scenario with notions of workload management. For example, it can be used to perform workload management among the application server tier and the database tier in the server farm. The other advantage of *SCOPS* framework is that the detailed design of control components can be replaced and extended easily. The "soft" dependency among system components allows them to be individually redesigned or upgraded. In this paper, we choose CPU as our focus for overload protection, but the sub-controller can be easily extended to monitor and manage memory, I/O, bandwidth and other critical resources on the back-end servers.

III. Sub-controller Design

Fig. 2 presents system diagram of *SCOPS*, including control flow of the sub-controller. During each control cycle, the resource monitor unit of sub-controller collects the resource consumption and request arrival information locally, which is then fed into the rate calculator module. The rate calculator determines the maximum allowed request rate for the local server node, which is then further distributed by the resource manager based on the requests profiles received from each proxy in the current control cycle. Below we describe the detailed functionalities of the above components in the sub-controller.

A. Resource Monitor and Data Collection

One common practice in resource monitoring is to periodically sample the usage statistics of the bottleneck resources, e.g., CPU capacity, memory, bandwidth, etc. We adopt two levels of control cycles to obtain reliable resource usage information. One is the external control cycle where the statistics is summarized and delivered to the resource manager shown in Fig. 2. The other is an internal cycle of finer granularity to collect several samples of resource usage status within one external control cycle. When monitoring the resource usage of server component externally, there may be some delay introduced by data caching inside the monitored components. Collecting multiple samples allows us to filter out outlier and conduct aggregations, thus obtaining more accurate results.

In this paper, we assume the system's bottleneck is investigated by mechanisms like [17] and specified by the cloud provider, thus the workload is managed towards a known bottleneck. Considering CPU capacity as the example, the OS API (e.g., system binary *top*) can be utilized to obtain the CPU usage, and the application server or network sniffing logs can be analyzed to obtain the request arrival rate on the server. We can also ensure the data collection itself does not affect the obtained results. For example, the actual CPU usage of sub-controller can be optimized to less than 0.3% on average, to ensure the accuracy of our measurement. It is also feasible to extend the sub-controller to monitor and manage a combination of system resources or even detect the bottleneck resource automatically, but these are out of scope of this paper.

B. Adaptive Rate Calculator Design

The rate calculator is designed based on the feedback Proportional-Integral (PI) controller due to its high adaptiveness to the variance of the workload. Fig. 3 illustrates

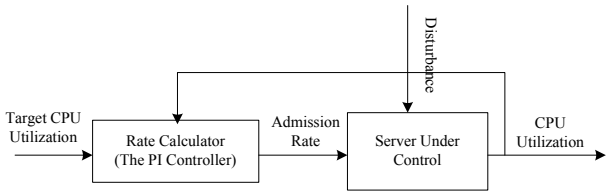

Fig. 3. Control System Diagram

the control system diagram. The "Target CPU Utilization" ($target$), configured by the system administrator, is the resource usage upper bound that the controller operates with. The "Admission Rate", the control variable, indicates the maximum admission rate in correspondence with the CPU target. The "CPU Utilization", the system output, reveals the actual CPU consumption.

The design of our PI controller involves two main steps: constructing statistical model by system identification; choosing the controller gain using pole placement technique [9].

1) System Identification: System identification is to figure out the relationship between control inputs and system outputs. We choose the CPU consumption as our system output, and the request arrival rate as our control input since it is dynamically changeable and has great impact on the CPU consumption [12]. We start from a linear model due to its simplicity as follows:

$$y(i) = A * y(i-1) + B * r(i-1) \tag{1}$$

where $y(i)$ is the CPU utilization and $r(i)$ stands for the request arrival rate during the ith control cycle. We conducted rounds of empirical experiments and verified that Equation (1) is able to adequately capture the relationship between control input and system output. Using linear regression, we estimate the time invariant unknown parameters A and B in Equation (1), which are 0.9883 and 0.0259 respectively. The linear regression fit results in the value of $R^2 = 0.93$, which indicates a good fit of the model.

2) Controller Design: The incremental form of the PI controller takes the following format:

$$r(i) = r(i-1) + (K_p + K_i) * e(i) - K_p * e(i-1) \tag{2}$$

where $e(i) = target - y(i)$ calculates the control error regarding the CPU utilization and $r(i)$, the request rate, represents the controller input. We use pole placement to calculate the controller gains K_p and K_i:

$$K_p = \frac{A - p_1 p_2}{b} \tag{3}$$

$$K_i = \frac{1 - p_1 - p_2 + p_1 p_2}{B} \tag{4}$$

where the model parameters A and B are derived from system identification. The pole locations p_1 and p_2 are obtained from the desired control performance quantified by the control settling time and the maximum overshoot. Here, we apply $K_p = 2.809$ and $K_i = 38.897$ respectively.

Two improvements are adopted while implementing the PI controller. First, the "windup" situation similar to [10] is also found in our design. To tackle this issue, we replace $r(i-1)$ in Equation (2) by the actual admission rate measured by the resource monitor ($r'(i-1)$). Second, it is observed that sometimes the actual admission rate profiled is a little bit different from the max admission rate regulated by the controller due to the profiling variance. Our solution is to replace $r(i-1)$ in Equation (2) by the minimum of $r(i-1)$ and the actual rate $r'(i-1)$ when detecting that the controller is trying to back off due to overload. Hence, the final PI-controller is:

$$\begin{aligned} r(i) =& Min(r(i-1), r'(i-1)) \\ &+ (K_p + K_i) * e(i) - K_p * e(i-1) \end{aligned} \tag{5}$$

C. Resource Manager and Control Cycle Synchronization

At each regular control cycle, the rate calculator returns the maximum request admission rate for the local server node. The resource manager then allocates the rate as quotas to the related front-end proxies as shown in Fig. 2.

For Server i, we assume there are J proxies dispatching workload to it, and each of the proxy receives μ_{ij} out of the maximum rate r_i returned by rate calculator on Server i. Thus, $\sum_{j=1}^{J} \mu_{ij} = r_i$. Consequently, for proxy j, if it manages workloads for applications on I servers, it will receive I rate quotas from these servers, such that the total rate it can send back is $\sum_{i=1}^{I} \mu_{ij} = \mu_j$. With the above allocation and aggregation done at the sub-controller and the proxy tier respectively, we have to deal with time synchronization issue among them. Otherwise, the allocation and aggregation may happen at random moment on different servers, causing inaccuracy in rate calculation and management.

An internal NTP service is used to roughly synchronize the clock of sub-controllers and proxy dispatchers. Resource manager at sub-controller sets its control cycle apart from the control cycle at the proxy tier by a few seconds. Thus the control cycle of sub-controller can be roughly synchronized to be a few seconds behind the control cycle in the proxy tier. This is to ensure that the resource manager in sub-controller can receive most recent information from the proxies. For each proxy dispatcher, as discussed later, it will adjust the rate whenever it receives the updated quota. This is because the proxies care more about the resource sharing among different service classes instead of the absolute rate being shared. The adaptiveness of our overall framework provides the flexibility that strict clock synchronization among all components is not required.

IV. Proxy Dispatcher Design

Fig. 2 also shows the work flow of the proxy dispatcher. The incoming requests are classified into different weighted queues based on their importance. The rate quota received from resource manager (in back-end sub-controller) is distributed by dispatch manager among weighted queues to achieve service classification. In this section, we present the design of proxy in terms of service classification and admission control.

A. Service Class Differentiation

Service class differentiation is designed to ensure that the performance goals of different workload can be configured appropriately for business and economic reasons. These goals may be specified as part of the SLA between customers and the service provider. As mentioned earlier, the configuration is usually stated as the target average response time of different workload and the business importance of meeting the target.

In order to achieve service classification, each proxy manages multiple waiting queues with different weights for requests of different classes, e.g., gold, silver, and bronze, as demonstrated in Fig. 2. The incoming requests are classified into their corresponding service classes, and mapped to the corresponding weighted queues during the dispatching process. Each queue controls the queuing time of its queued requests based on the rate quota it receives from dispatch manager.

Our approach to achieving service differentiation consists of two components: the queuing model and the system optimization built upon a utility function. The latter functions when the system overloads, i.e., the request arrival rate is higher than the maximum rate quota that a proxy manages. In such scenario, we have to control the queuing time of each service class (may lead to certain degree of violation in SLA, e.g., longer delay in average response time of each service class), such that the penalty of SLA violation is minimized.

1) Utility Function: In order to reflect the penalty of failing to meet the target response time of each service, we design a utility function U_{jk} for proxy dispatcher j, which represents the penalty for service class k based on its current response time T_{jk} and the business importance I_{jk}:

$$U_{jk}(T_{jk}) = \frac{I_{jk}}{2}\{(T_{jk}-d_{jk})+\sqrt{((T_{jk}-d_{jk})^2)+0.5}\}, \quad (6)$$

where d_{jk} is the target response time for service class k in the proxy dispatcher j.

Under the definition of our utility function, the optimization goal is to minimized the total utility $U_j = \sum_{k=1}^{n} U_{jk}(T_{jk})$ during runtime, where n is the number of the service classes. That is, the proxy needs to determine the dispatching rate μ_{jk} for each service class according to the following constrained optimization problem: $\min_{\{\mu_{jk}\}} \Sigma_{k=1}^{n} U_{jk}(T_{jk})$, under the constraint of $\sum_{k=1}^{n} \mu_{jk} = \mu_j$.

2) Queuing Model: We now associate service class dispatch rate with its responding time. In other words, in order to determine μ_{jk} for each service class, we need to obtain the relationship between μ_{jk} and T_{jk} to translate the constrained optimization problem into $\min_{\{\mu_{jk}\}} \Sigma_{k=1}^{n} U_{jk}(T_{jk}(\mu_{jk}))$, under the constraint of $\sum_{k=1}^{n} \mu_{jk} = \mu_j$. We employ a queuing model, named **Generalized Processor Sharing (GPS) Queue** [6], to captures the transient behavior of various workload. We adopt such model not only because it is applicable to dynamic resource allocation in shared servers but also because it does not rely on the steady-state assumptions.

Thus, we model our queuing system as follows:

$$\begin{aligned} T_{jk}(i) =& \frac{W_{jk}(i)}{W\mu_{jk}(i)}(q_{jk}(i) + \frac{W_{jk}(i)}{2}(\lambda_{jk}(i) \\ & - \mu_{jk}(i))) + \frac{1}{\mu_{jk}(i)} \end{aligned} \quad (7)$$

Here, $W_{jk}(i)$ is the non-empty time of queue k during the ith control cycle. W denotes the time duration of one control cycle and $\mu_{jk}(i)$ represents the dispatching rate. $\lambda_{jk}(i)$ denotes the request arrival rate of queue k during the ith control cycle. $q_{jk}(i)$ is the initial queue length of queue k at the beginning of the ith control cycle.

The value of $q_{jk}(i)$ can be easily measured at the beginning of each control cycle. Both $W_{jk}(i)$ and $\lambda_{jk}(i)$ should be estimated since neither of them can be measured until the end of each control cycle. A straightforward way to estimated $W_{jk}(i)$ and $\lambda_{jk}(i)$ is to approximate them to the same statistics measured from the last control cycle. Although some precision is lost by this approximation, it is validated to be effective and acceptable in Section V. As a result, Equation (7) can be translated into

$$\begin{aligned} T_{jk}(i) =& \frac{W_{jk}(i-1)}{W\mu_{jk}(i)}(q_{jk}(i) + \frac{W_{jk}(i-1)}{2}(\lambda_{jk}(i-1) \\ & - \mu_{jk}(i))) + \frac{1}{\mu_{jk}(i)}. \end{aligned} \quad (8)$$

By applying the Lagrange multiplier to Equation (8) [4], the optimization problem with constraints can be reduced to unconstrained optimization problem as follows:

$$L_k(\mu_{jk}, \alpha) = \Sigma_{k=1}^{n} U_{jk}(T_{jk}(\mu_{jk})) - \alpha(\Sigma_{k=1}^{n}\mu_{jk} - \mu_j)$$

Thus, the optimization is translated into the minimization of function L_k, that is, to solve $\nabla_{\mu_{jk},\alpha} L_k(\mu_{jk}, \alpha) = 0$ for the best allocation of μ_{jk}.

B. Rate Control and Load Balancing

At the beginning of each control cycle, the dispatch manager obtains the calculated weight of each queue from the utility function described above. The weight is the percentage of overall rate allocated to one queue. Then the dispatch manager distributes the rate quota received from the sub-controller to each queue based on its weight. The rate of each queue is updated whenever the proxy receives rate update from the back-end sub-controllers, but the weight of each queue is only recalculated once per control cycle as discussed above.

We implement the rate control utilizing the token bucket algorithm. On each queue of one proxy dispatcher, we maintain one token bucket policer for each back-end sub-controller the proxy forwards workload to. The rate of the token bucket is set to the corresponding divided rate quota from each sub-controller based on the weight of this queue. The burst size is set based on the capacity profiling of each back-end server. For a request belonging to a queue, by examining the queue's available token for each sub-controller, the proxy can determine whether the request can be admitted into the back-end system, and which server the request should be forwarded

TABLE I
LOAD DYNAMICS DURING THE EXPERIMENTS

Stage	Control Cycle ID	Duration	Concurrency	Think Time	Request CPU Consumption	*Stage Description*
0	0	*400 sec*	1	$100ms$	low	Warm up
1.a	20	*360 sec*	1	$50ms$	low	Light Load: adjust request arrival interval
1.b	38	*340 sec*	2	$50ms$	low	Light Load: adjust the number of clients
1.c	55	*400 sec*	2	$50ms$	high	Light Load: adjust request CPU requirement
2.a	75	*580 sec*	10	$50ms$	high	Overload: transit from heavy load
2.b	104	*620 sec*	12	$50ms$	high	Overload: adjust the number of client
2.c	135	*540 sec*	12	$50ms$	low	Overload: adjust request CPU requirement
3.a	162	*360 sec*	2	$50ms$	low	Fluctuation: overload to light load
3.b	180	*360 sec*	12	$50ms$	low	Fluctuation: light load to overload
4	198	*260 sec*	1	$50ms$	low	Ramp down

to. The tokens can be examined in a round robin fashion to ensure fairness and load balancing.

V. PERFORMANCE EVALUATION

A set of experiments are conducted to demonstrate the performance of managing server CPU consumption while maintaining the predefined QoS requirement under dynamic workload.

A. Experimental Environment

In order to verify the adaptability of our design, we intentionally choose three heterogeneous computers as our application server platforms, in particular with different processors: Intel Xeon X3430 quad-core, Intel Core 2 Duo E6400 dual-core, and Intel Core 2 Duo T9300 dual-core respectively. We have three proxies and three clients running on a separated machine with two Intel Xeon E5405 processors. All the machines are connected by Gigabit Ethernet. We use Apache Tomcat 6.0.29 as the application server, and Tinyproxy [1], a light weight HTTP proxy daemon, as our proxy.

A synthetic workload generator is used to generate http requests, with inter-arrival time (think time) following a truncated negative exponential distribution. The workload can be adjusted dynamically at runtime by changing the mean and the bias of the chosen think time distribution or the number of concurrent requests. By mapping the URL, the workload generator produces requests with three service classes: $Gold$, $Silver$, and $Bronze$. All of them share the same average response time target ($200ms$), but with different importance marked as I_G, I_S and I_B respectively. To emulate CPU consumption in the application server, we deploy a CPU-intensive application on all servers, which computes the sum of N integers every time. Hence, the CPU consumed per request can be adjusted by different value of N supplied through the workload generator.

During the experiment, the characteristics of the workload are changed dynamically. Table I summarizes the parameters for each stage and the purpose of doing such changes. Basically, the workload changes during light load (Stage 1), overload (Stage 2), and workload fluctuation (Stage 3). We choose 80% as the target CPU usage for each application server, and 20 seconds as the control cycle for both sub-controllers and the proxy dispatchers. Below, we present the evaluation results of *SCOPS* in terms of overload protection, service differentiation, and robustness against control component failure.

B. Scenario 1: Overload Protection and Service Differentiation

The CPU utilization of all the application servers is shown in Fig. 4. The workload stage is marked by the top x axis. Given the target CPU utilization at 80%, the controllers maintain a stable CPU consumption during the overload stages, regardless of the dynamics of the workload. As shown in Fig. 4, quick convergence and little oscillation are observed for all the application servers, even though they are heterogeneous in their computation capacity.

Fig. 5 illustrates the throughput for one application server (*arrival rate*), together with the *target rate* calculated by our sub-controller. Both of them experience very limited oscillation, although the control is conducted without $a\ priori$ knowledge of workload characteristics. We observe similar results for the other application servers, which are not presented here due to space limit.

Fig. 6 shows the throughput of all the service classes on Proxy III. Under the light load stage, each service class is equally treated since there is no resource contention. During the overload stage, the effect of service differentiation emerges due to the limit of CPU capacity. Service class with higher importance receives more resources, so $Gold$ class experiences limited queuing and shorter waiting time. The evaluation result confirms that $Gold$ wins with the highest throughput. Similarly, $Silver$ class outperforms the $Bronze$ class.

In order to demonstrate that our service differentiation achieves low overall QoS penalty, we repeat the same experiment by using a static weighted assignment algorithm based on service class importance. The dispatching rate μ_{jk} is calculated with the following function: $\mu_{jk} = \frac{I_{jk}}{\sum_{k=1}^{n} I_{jk}} \mu_j$. The comparison of the total utility is shown in Fig. 7. *SCOPS* shows better performance with less penalty during overload, and outperforms the static weighted assignment by 50% on average.

C. Scenario 2: Fault Tolerance against Server or Proxy Failure

In order to evaluate the resilience of *SCOPS*, the same setup as Scenario 1 is used, but with fewer stages in workload characteristic changes. In particular, we limit the experiment to stage 0, $1.b$, $2.c$, and 4. The experiment is repeated twice, with failure and restart of a back-end application server and a proxy server respectively.

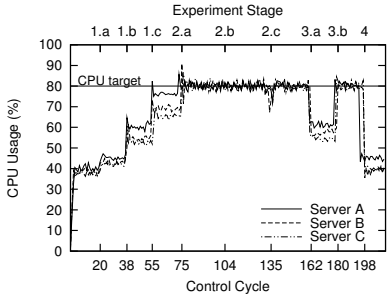

Fig. 4. CPU Load of All Three Servers during Load Dynamics

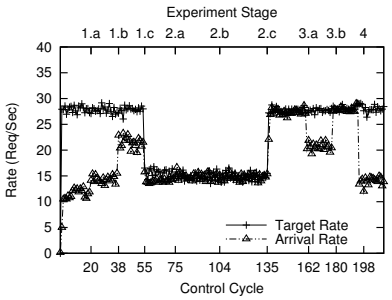

Fig. 5. Actual Request Arrival Rate versus the Maximum Allowed Rate on One Server

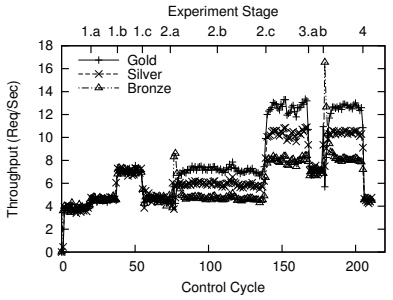

Fig. 6. Throughput Differentiation of One Proxy

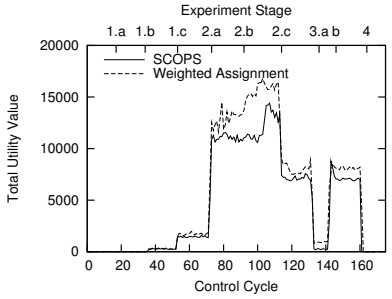

Fig. 7. Total Utility Comparison

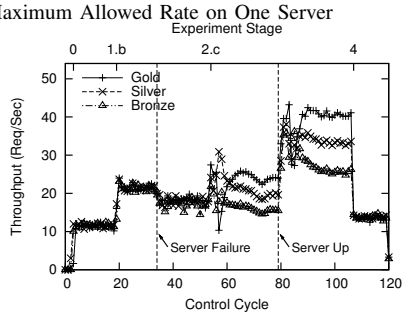

Fig. 8. Resilience to Application Server Failure and Recovery

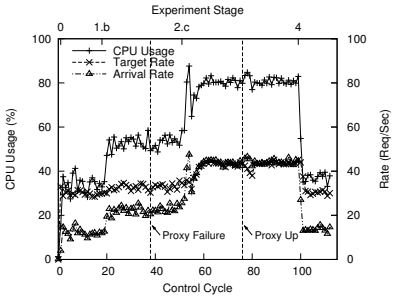

Fig. 9. Resilience to Proxy Failure and Recovery

In the first run, we turn off one application server to emulate the application server failure. Fig. 8 shows the sum of the throughput for each service class. The server failure happens in the middle of stage $1.b$, as marked in the figure. The system throughput drops a little bit due to the reduction of overall server capacity. However, the overall throughput remains stable most of the time, not affected by the server failure. In stage $2.c$, the same application server is restarted to simulate a server reboot or addition. This event is also marked in Fig. 8. The overall throughput increases quickly, and is even higher than the time before server failure. This is caused by additional load at stage $2.c$ comparing to the light load of stage $1.b$. We observe that the throughput converges quickly within 5 control cycles with little oscillation, demonstrating resilience to such server failure.

In the second run, a front-end proxy is shut down and then brought up to evaluate the system's performance. Fig. 9 shows the CPU utilization and the throughput of one application server (the other servers share similar qualitative results). As marked in the figure, one of the proxy is shut down during stage $1.b$. The capacity assigned to the failed proxy is redistributed to the other running proxies quickly. We observe no impact to the performance of our system. The proxy restarts during stage $2.c$. Similarly, both the CPU utilization and the throughput remain stable with very little oscillation during the change. We have also tested system failure cases with different workload profiles, and observed similar results as presented. These evaluations confirms the resilience of our decentralized system to control component failures.

D. Runtime Overhead

Our approach requires several parameters to be measured and exchanged among the proxy dispatchers and servers during runtime. We monitor the CPU overhead imposed by the sub-controller to the back-end server, and find that the CPU overhead of this online estimation is negligible, less than 0.3% across a number of experiments, even when the controller is working intensively during the control cycles.

TABLE II
OVERHEAD OF SCOPS: COMPARISON OF RESPONSE TIME

Architecture	Response Time Under Throughput		
	12.75 Req/s	*60 Req/s*	*97.5 Req/s*
Without Proxy	$133.3ms$	$149.3ms$	$205.2ms$
SCOPS	$134.3ms$	150.7	$207.4ms$
Delay Overhead	0.75%	0.94%	1.07%

The queuing operation and weight calculation of proxy impose response delay. We compare the baseline response time (requests directly go to application server bypassing the proxy) with our managed response time (requests are forwarded by our proxy dispatcher with queuing and calculation functionality). Table II shows that, as throughout increases from 12.75 requests per second to 97.5 requests per second, the delay incurred by the proxy increases slightly from 0.75% to 1.07%, about 1-2 ms. Thus, we believe that the management mechanism in proxy dispatcher will not impose serious runtime overhead to our system.

Since the statistics measured at the proxy and sub-controller need to be periodically communicated, we also analyze the network overhead of these messages. For the scenario where we run three servers with the throughput upto 140 requests per second, the network traffic created by our approach is about 24 Kbps, negligible for Gigabit network of cloud. For large scale application server deployment, since our design automatically groups proxies and servers into clusters based on their working relations, the communication overhead is very limited.

VI. Related Work

The boom of cloud computing, especially the concept of offering resource as a service, emphasizes the importance of designing a flexible control system for overload protection together with QoS assurance [16]. A number of research efforts have been done in this area. Most of them rely on one centralized unit to gather information and dispatch decisions, not quite applicable to large scale cloud environment.

Researchers have focused on the management of data centers on different resource-sharing environments for years: multiple application instances sharing the same application server [19] and [20], multiple application servers (each with exactly one application instance) sharing the same physical machine [21], [14] and [2], multiple virtual machines (each with exactly one application instance) sharing a physical machine [3] and [13], and finally, each application instance running on a dedicated physical machine[7]. We adopt the resource sharing scenario of multiple application instances running on the application server which in turn run on the physical machine, to represent general application server deployment and offer extreme flexibility and scalability.

Admission control on application servers has been widely studied in the literature. Li et al. [15] describe an algorithm allocating configurable fixed percentages of bandwidth across numerous simultaneous clients with Apache HTTP server. Cataclysm [22] is a low overhead, highly scalable admission control system, handling extreme loads in hosted Internet applications. Welsh et at. [23] present SEDA architecture, which splits applications into event-driven stages connected by explicit queues, and relies on dynamic resource controllers to keep stages within their operating regime despite large fluctuations in workload. Xiong et al. [24] propose a novel and efficient distributed flow control scheme for multirate multicast, based on Proportional Integral and Derivative (PID) controllers to achieve both intrasession and intersession fairness. Our overload protection is done through the collaboration among sub-controllers and proxies. The sub-controllers generate the max admission rate and distribute it among the proxies, while the proxy applies the token bucket policer to abide by its rate quota. Several researchers have designed various utility functions to support SLA, such as using price-based priority service to provide differentiated quality of service [18]. Our utility function aims to minimize the penalty due to the violation of the target response time. It produces the weight of each class based on online auditing of its runtime information to achieve an optimal service class differentiation.

VII. Conclusion and Future Work

In this paper, we present a decentralized approach to managing workload on the application servers in large scale deployment, e.g., cloud environment. In particular, we design and develop *SCOPS*, a low overhead, highly scalable, simple but effective workload management system for QoS assurance and overload protection. *SCOPS* is designed to collaborate with commodity host OS, legacy application servers and application instances, thus easing the field deployment. The performance evaluation demonstrates that our approach features with low runtime overhead, effective resource overload protection, and SLA-oriented service classification. As part of our future work, we plan to extend our overload protection to more general session-oriented workloads and other types of resources, e.g., memory, bandwidth, etc. In the scenario of protecting multi-type bottleneck resources, we will also consider integrating an optimal load balancing scheme to automatically detect the current bottleneck capacity of back-end servers.

Acknowledgement

This work was partially supported by AFOSR FA9550-07-1-0527 (MURI), ARO W911NF-09-1-0525 (MURI), and NSF CNS-0905131.

References

[1] *Tinyproxy: light-weight HTTP proxy daemon*. https://www.banu.com/tinyproxy.

[2] Zakaria Al-qudah, Hussein Alzoubi, Mark Allman, Michael Rabinovich, and Vincenzo Liberatore. Efficient application placement in a dynamic hosting platform. In *Proceedings of the 18th international conference on World wide web*, pages 281–290, 2009.

[3] Amr Awadallah and Mendel Rosenblum. The vmatrix: A network of virtual machine monitors for dynamic content distribution. In *7th International Workshop on Web Content Caching and Distribution*, 2002.

[4] Dimitri P. Bertsekas. *Nonlinear Programming: 2nd Edition*. Athena Scientific, 1999.

[5] N. Bhatti and R. Friedrich. Web server support for tiered services. In *Proceeding of IEEE Network*, pages 64–71, 1999.

[6] Abhishek Chandra, Weibo Gong, and Prashant Shenoy. Dynamic resource allocation for shared data centers using online measurements. In *Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 381–398, 2003.

[7] Jeffrey S. Chase, David E. Irwin, Laura E. Grit, Justin D. Moore, and Sara E. Sprenkle. Dynamic virtual clusters in a grid site manager. In *Proceedings of the Twelfth International Symposium on High Performance Distributed Computing*, 2003.

[8] Ludmila Cherkasova and Peter Phaal. Session-based admission control: A mechanism for peak load management of commercial web sites. *IEEE Transactions on Computers*, 51(6):669–685, 2002.

[9] Yixin Diao, Joseph L. Hellerstein, Sujay Parekh, Rean Griffith, Gail E. Kaiser, and Dan Phung. A control theory foundation for self-managing computing systems. *IEEE Journal on Selected Areas in Communications*, pages 2213–2222, 2005.

[10] Yixin Diao, Xiaolei Hu, Asser Tantawi, and Haishan Wu. An adaptive feedback controller for sip server memory overload protection. In *Proceedings of the 6th international conference on Autonomic computing*, pages 23–32, 2009.

[11] Sameh Elnikety, Erich Nahum, John Tracey, and Willy Zwaenepoel. A method for transparent admission control and request scheduling in e-commerce web sites. In *Proceedings of the 13th international conference on World Wide Web*, pages 276–286, 2004.

[12] N. Gandhi, D. M. Tilbury, Y. Diao, J. Hellerstein, and S. Parekh. Mimo control of an apache web server: modeling and controller design. In *Proceeding of American Control Conference*, pages 200–2, 2002.

[13] Xuxian Jiang and Dongyan Xu. Soda: a service-on-demand architecture for application service hosting utility platforms. In *Proceeding of Twelfth IEEE International Symposium on High Performance Distributed Computing*, 2003.

[14] A. Karve, T. Kimbrel, G. Pacifici, M. Spreitzer, M. Steinder, M. Sviridenko, and A. Tantawi. Dynamic placement for clustered web applications. In *Proceedings of the 15th International Conference on World Wide Web*, pages 595–604, 2006.

[15] Kelvin Li and Sugih Jamin. A measurement-based admission-controlled web server. In *Proceedings of IEEE INFOCOM*, pages 651–659, 2000.

[16] Harold C. Lim, Shivnath Babu, Jeffrey S. Chase, and Sujay S. Parekh. Automated control in cloud computing: challenges and opportunities. In *Proceedings of the 1st workshop on Automated control for datacenters and clouds*, 2009.

[17] S. Malkowski, M. Hedwig, and C. Pu. Experimental evaluation of n-tier systems: Observation and analysis of multi-bottlenecks. In *IEEE International Symposium on Workload Characterization*, pages 118–127, 2009.

[18] Peter Marbach. Priority service and max-min fairness. In *IEEE/ACM Transactions on Networking*, pages 733–746, 2003.

[19] Michael Rabinovich, Zhen Xiao, and Amit Aggarwal. Computing on the edge: A platform for replicating internet applications. In *International Workshop on Web Caching and Content Distribution*, pages 57–77, 2003.

[20] S. Sivasubramanian, G. Pierre, and M. van Steen. Replicating web applications on-demand. In *Proceedings of the IEEE International Conference on Services Computing*, 2004.

[21] Chunqiang Tang, Malgorzata Steinder, Michael Spreitzer, and Giovanni Pacifici. Abstract a scalable application placement controller for enterprise data centers. In *Proceedings of the 16th international conference on World Wide Web*, pages 331–340, 2007.

[22] Bhuvan Urgaonkar and Prashant Shenoy. Cataclysm: policing extreme overloads in internet applications. In *Proceedings of the 14th international conference on World Wide Web*, pages 740–749, 2005.

[23] Matt Welsh, David Culler, and Eric Brewer. Seda: An architecture for well-conditioned, scalable internet services. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, 2001.

[24] N. Xiong, X. Jia, L. T. Yang, A. V. Vasilakos, Y. Li, and Y. Pan. A distributed efficient flow control scheme for multirate multicast networks. *IEEE Transactions on Parallel and Distributed Systems*, 21(9):1254–1266, 2010.