2013 IEEE International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing

Towards Efficient Deployment of Cloud Applications Through Dynamic Reverse Proxy Optimization

Wei Yuan, Hailong Sun, Xu Wang, Xudong Liu
School of Computer Science and Engineering
Beihang University
Beijing, China
{yuanwei, sunhl, wangxu, liuxd}@act.buaa.edu.cn

***Abstract*—With the increase of users and the deployment requests, the issue of dynamic deployment in PaaS becomes prominent. Different approaches of application deployment have been deeply discussed, but the issues like fast response to a large number of concurrent deployment requests are rarely focused. In this work, we extend Nginx as a dynamic reverse proxy to support dynamically remote configuration for better elasticity of cloud applications in PaaS, and then further optimize it for improving performance under a large number of concurrent configuration requests. Three optimization approaches are proposed: Batch Request Committing (BRC), Batch File Processing (BFP) and In Memory Configuration (IMC). We give a detailed implementation of each method, and a qualitative analysis on three optimization approaches has been made. Finally, a series of experiments are presented to validate the optimization effect. The experiment results show that the maximum throughput per second has increased significantly, and the average response time of each request has decreased dramatically.**

***Keywords*—PaaS; reverse proxy; elasticity; load balancing;**

I. Introduction

As one of the three service models for cloud computing (Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS)) defined by the U.S. National Institute of Standards and Technology (NIST) [1], PaaS plays a significant role in cloud computing. In PaaS White Paper [2] published by NJVC and Virtual Global, it makes the importance of PaaS explicit that PaaS potentially offers the greatest impact over any other aspect of cloud computing. PaaS usually provides consumers with hardware and software infrastructure to facilitate the designment, development, deployment, and hosting of web applications.

Public PaaS like Google App Engine (GAE) [3] and Microsoft's Windows Azure [4] often face the challenges of high scalability and rapid elasticity due to the huge amount of hosting cloud applications and more efficient resource utilization. High scalability requires better load balancers to dispatch requests among application replicas, and rapid elasticity needs to dynamically scale sizes of application replicas. The core component of PaaS to deal with the two challenges is called *reverse proxy*. For cloud applications hosted in PaaS, all requests firstly reach the reverse proxy server by DNS (Domain Name Server), then the reverse proxy reads the pre-defined redirection configuration for requests and dispatches them to all application replicas for load balancing. In addition, when applications are deployed or the sizes of application replicas scale in and out, the reverse proxy will be notified to write (or modify) the redirection configuration and then make the rapid elasticity effective.

The renowned reverse proxy solutions Nginx [5] and HAProxy [6] have dealt with the challenge of high scalability by load balancing efficiently, but they have done little on the rapid elasticity for cloud applications in PaaS since they usually require users to modify the configuration manually. Generally, when a web application is ready to be deployed in PaaS, it is usually copied to multiple remote nodes with dynamically provisioned resources (e.g. virtual machines, operation systems and basic software), and PaaS can get the access information of all application replicas. Then the access information will be sent to the reverse proxy for writing new redirection configuration. Only the new redirection configuration of the reverse proxy works, requests to the web application can be correctly received and parsed and the process of deployment finishes. Therefore, the reverse proxy plays an important role in the deployment of cloud applications, and the efficient reverse proxy in deployment will lead to a prominent improvement of cloud application deployment. Similarly, the reverse proxy will modify the involved redirection configuration for auto scaling of cloud applications. In this work, we focus on the the efficiency of reverse proxy for improving the elasticity of cloud applications.

Typical examples of PaaS providers include Google App Engine (GAE) [3], Microsoft's Windows Azure [4], and Salesforce's Force.com [7]. Windows Azure supports auto scaling, but it is based on application roles and a configuration file specified by users [8]. Users's participation in configuration is not a satisfying way allowing for convenience and reliability. GAE also supports auto scaling, and this process is transparent to users. Some of the implementation details in PaaS is unreachable for us, and we try to get some similar information from open source systems. AppScale [9] is an open source cloud platform and it implements Google App Engine APIs. One of the primary components of AppScale is AppLoadBalancer [10]. AppLoadBalancer uses Nginx [5] to serve static content, and service copies are load balanced by HAProxy [6]. The reverse proxy servers Nginx and HAProxy are widely used in PaaS, and they are also used to provide a better access for SLD (second-level domain) [11]. However, the support for dynamic configuration is not sufficient in both Nginx and HAProxy. For example, the configuration of Nginx is based on a static configuration file and a reload command is inevitable to make the modification effective. Each time on modifying the configuration file of Nginx, the operation

978-0-7695-5088-6/13 $31.00 © 2013 IEEE
DOI 10.1109/HPCC.and.EUC.2013.97

should be finished manually. A relevant third-party module in Nginx is called ngx_xconf [12]. According to its description, it will make it possible for Nginx to use remote configuration file. But regrettably this module is under active development and is not production ready yet. When it comes to HAProxy, some additional tools are also needed to support dynamic configuration.

In this work, we extend Nginx as a dynamic reverse proxy to support dynamic remote configuration for better elasticity of cloud applications in PaaS, and then further optimize it for improving performance under a large number of concurrent configuration requests. The work is applied to Service4All [13] which is a cloud computing platform for service oriented software developers. To simplify the discussion, we choose the deployment of cloud applications to produce configuration requests, and the auto-scaling scenarios have similar behaviors. When a new service is deployed by the deployment module, a component called Configuration Request Controller (CRC) embeded in deployment module sends a configuration request to Configuration Request Receiver (CRR) in Nginx. The request contains information about the new deployed service and replica locations. Then CRR parses the information and creates a new configuration file. Finally the reload command is necessary to make the new redirection configuration effective. What's more, three optimization approaches are also proposed to improve the performance: (a) Batch Request Committing (BRC), (b) Batch File Processing (BFP), (c) In Memory Configuration (IMC). These methods are from different points of view. BRC is in consideration of the costs of network transmission, BFP is to reduce the times of file operations, and IMC is based on the fact that memory is much faster than disk. The main contributions of this paper include:

(1) We implement the dynamic configuration of Nginx which is employed as a reverse proxy and load balancer.
(2) We build a dynamic reverse proxy framework based on Nginx, and we propose three optimization approaches to improve the performance under a large number of concurrent configuration requests.
(3) We present the experiment results and analysis to validate the optimization methods.

In the sections that follow, we present related work in Section II, and then describe the dynamic reverse proxy framework based on Nginx in Section III. Section IV provides the implementation of three optimization approaches. A qualitative analysis on three optimization approaches has been made in Section V. Section VI shows the experiment results. Finally, we conclude this work in Section VII.

II. Related Work

In this section, on improving the elasticity of cloud applications, we provide some related work from two aspects: reverse proxy and load balancing.

A. Reverse Proxy

In computer networks, reverse proxy can be treated as a special proxy server. As shown in Figure 1, the reverse proxy accepts requests from clients and chooses a server from web

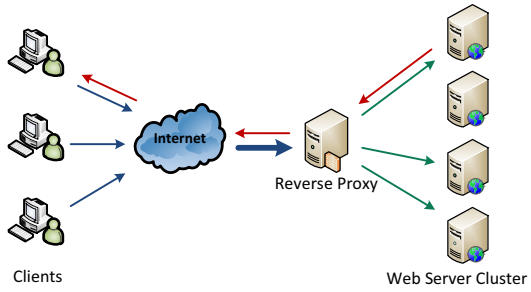

Fig. 1: Requests Processing of Reverse Proxy

server cluster according to a certain strategy. The final content will be returned by reverse proxy as if it were the origin server. Besides, reverse proxy is usually deployed near the origin content instead of near clients.

In the architecture of some existing PaaS, such as Heroku [14], Sina App Engine [15], the outermost layer is called Reverse Proxy Layer. It is not only an efficient method to extend service ability, but also plays a role in many other aspects.

1) A large number of concurrent requests (or a large amount of data traffic) can be dispatched to different servers, which will decrease the response time evidently and deliver a better experience to clients. It is also known as load balancing.
2) The reverse proxy makes back-end servers invisible to clients. The web servers in cluster cannot be accessed directly, and reverse proxy may act as a firewall to protect the data stored in back-end servers. Some reverse proxies also support SSL (Secure Sockets Layer).
3) It is more flexible to change back-end nodes or improve scalability, by modifying the mapping relationship between reverse proxy and web servers without any setting by clients.
4) Some of the servers with caching ability used as reverse proxies may reduce the load on back-end servers by caching static content. The reverse proxy will check its cache to see if the requested item is contained. If not, it forwards the request to the real web server and copies the content to its cache.

Different approaches of service deployment, such as script-based, language-based and model-based deployment approaches [16], have been deeply discussed by researchers. Most of the researches on service deployment emphasize service deployment process itself, but time and resource costs are sometimes neglected. When the service replicas are copied to the corresponding nodes, whether the new replicas can be invoked successfully and immediately is a serious problem, but it is rarely focused in recent researches [17]. This problem is determined by reverse proxy to a great extent, since the configuration requests are accepted by reverse proxy. Nginx is used in reverse proxy layer of Heroku, but the dynamic implementation and capability in handling a great number of concurrent requests is not clear yet.

B. Load Balancing

With the fast increasing users and growing traffic, a single server cannot work effectively, and the performance is the major issue in a concurrent circumstance. Fortunately, load balancing can solve this problem elegantly. It is also one of the hot topics in cloud computing [18]. Load balancing is concerned with improving scalability, availability, and reliability of web servers or web caches [19]. Based on a group of web servers, an incoming request will be assigned to one of the back-end nodes, and the server responses the request independently. The key point of load balancing is the strategy in requests dispatching. When the reverse proxy acts as a load balancer, it involves distributing requests according to some load balancing rules, such as round-robin and least connections. The server cluster of high reliability usually achieves load balancing with a layer-4 or layer-7 (in OSI model) load balancer.

- Hardware load balancer in layer 4/7. Some typical hardware equipment such as F5 BIG-IP, Cisco CSS, Radware AppDirector. They are installed between servers and outside network without dependence on operating system, and suitable for large-scale access.
- Software load balancer in layer 4. The open source project Linux Virtual Server (LVS) [20] is the representative work of this layer. For transparency, scalability, availability and manageability of the system, LVS is usually based on three-tier architecture. LVS is in charge of requests distributing without producing traffic in layer 4.
- Software load balancer in layer 7. It is mostly based upon HTTP reverse proxy such as Nginx and HAProxy. As load balancers, both of Nginx and HAProxy can bear large concurrent connections. On account of low cost and good performance, layer-7 software load balancing is widely used in recent years.

The hardware load balancers can promote the overall performance of the system remarkably, but they cost too much. LVS performs outstandingly among load balancing software. However, there is no support for regular expression, and it is unable to separate dynamic and static content. In large website applications, it involves complicated configuration. Compared with the other solutions, Nginx is more flexible in configuration. And in this paper, our work is based on it.

III. Problem Statement And Framework

A. Dynamic Reverse Proxy

PaaS is faced with an open Internet environment, and the application load is of high variability. Application requests are accepted by the reverse proxy server, then the reverse proxy reads the pre-defined redirection configuration for requests and dispatches them to all application replicas for load balancing. Moreover, when applications are deployed or the sizes of application replicas scale in and out, the reverse proxy will be notified to write (or modify) the redirection configuration and then make the rapid elasticity effective.

Service deployment strategy is an important issue, and it has drawn lots of attention from researchers. However, issues like fast response to concurrent configuration requests of deployed services are rarely focused. With the promising development of PaaS and evolution of service deployment algorithms, the frequency of deployment requests and access will be rising. Dynamic reverse proxy is of great significance since requests from users are accepted by reverse proxy, and its configuration must be modified and effective immediately. The configuration can be treated as a mapping in runtime. Therefore, the total response time is mostly determined by the dynamic reverse proxy.

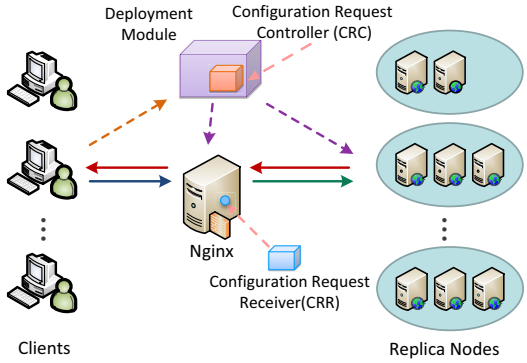

Fig. 2: Framework of Dynamic Reverse Proxy

B. Overall Framework

The dynamic reverse proxy framework and request processing are shown in Figure 2, and Nginx is chosen as a reverse proxy. The characteristics and superiority of Nginx will be shown later. A component called Configuration Request Controller (CRC) is in the deployment module. We extend Nginx as a dynamic reverse proxy. A component called Configuration Request Receiver (CRR) is embeded in Nginx. When deploying a service in PaaS, a deployment request is sent to the deployment module. The deployment module will get some certain nodes after resource allocation. An IP address with a port number can be seen as a node. Suppose that we need to create three replicas on different web servers. The deployment module puts each replica on a designated server. In the meantime, CRC sends a message to Nginx, which contains information about the new deployed service and replica locations. CRR receives and parses the message to form a configuration information segment. Then a new configuration file based on this segment will be created and reloaded.

After successful reload, the mapping relationship between reverse proxy Nginx and back-end web servers has been established. If users invoke this service, Nginx accepts the request, and one of the back-end server will be selected based on a certain load balancing strategy. The response will also transmitted by Nginx. A simple built-in upstream module in Nginx is used for load balancing, which dispatches the requests to back-end servers according to a designated algorithm. Nginx supports the following load balancing algorithms.

- Round-robin. It is also the default algorithm. Each request is distributed among back-end nodes in chronological order. Health check mechanism is realized in

Nginx. If there is a failure with one of the web servers, Nginx will discard it from cluster automatically. Besides, each server can be assigned a weight in accordance with its performance.

- IP_hash. Each request is dispatched upon the hash result of source IP address. It means that request from the same client will be sent to the same web server, and the session will be kept.
- URL_hash. It needs a third-party module. Each request is dispatched upon the hash result of destination URL, and the same URL is sent to the same web server. It is often used when back-end nodes are cache servers.
- Fair. Requests are distributed based on response time of each server, and the server with short response time will get request preferentially.

C. Nginx

Nginx is the pivotal part in this framework. It is a lightweight, high-performance, open source HTTP web server and reverse proxy. As known to all, Apache is the most popular web server currently. It is known for its stability and reliability. According to the survey results published by Netcraft [21], as of June 2013, Nginx is the third-most-popular web server with a 14.56% market share, and Apache remains the dominant force, commanding a 53.34% share. Nginx shows a significant growth referring to a 6.55% share in June 2011.

Both Apache and Nginx offer the basic features such as SSL, virtual host, load balancing, FastCGI, and more. However, Nginx is more outstanding in reverse proxy and load balancing than Apache. It has lower CPU and memory cost but handles more connections per second. Unlike Apache, which is a processed-based web server, Nginx is an asynchronous web server [22]. Nginx supports concurrent requests with spawning very few or no threads, while Apache requires a new thread for each concurrent request. It is the reason that accounts for Nginx's low use of RAM under heavy traffic loads.

Furthermore, Nginx supports hot deployment, and it is the feature that makes frequent configuration modification possible. Allowing for the characteristics and superiority, we use Nginx as a reverse proxy and load balancer in our framework.

IV. Optimization Approaches

Based on the above framework, the configuration file of Nginx can be modified and reloaded dynamically. But the performance achieved by the dynamic reverse proxy is not favorable. When a great number of configuration requests come, they produce the modification and reload operations with the configuration file, and the static configuration file will be a competitive resource. In this circumstance, Nginx becomes the bottleneck of the entire system. Therefore, the service deployment messages are expected to be transmitted to reverse proxy as soon as possible.

Regardless of the service deployment time, we are mainly concerning the time consumed by the dynamic reverse proxy Nginx with concurrent requests. Frequent requests take up too much time in modifying and reloading configuration file.

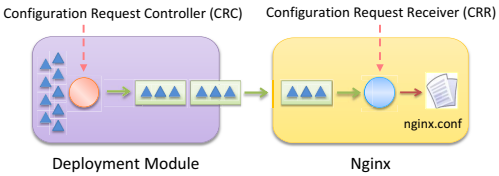

Fig. 3: BRC

A mapping between reverse proxy and web server cluster is required for each request. When the new configuration segment is created, the mapping is established. And then reloading configuration file is necessary to make the new mapping effective.

In this work, we propose three optimization approaches to improve the performance of Nginx: Batch Request Committing (BRC), Batch File Processing (BFP) and In Memory Configuration (IMC). And we are interested in two items with the optimization results: maximum throughput per second and the response time per request. The implementation details of each method is described as follows.

A. Batch Request Committing (BRC)

A single configuration request produces writing and reloading configuration file in Nginx. If the concurrent requests are intensive, Nginx's configuration file will be a critical resource, and most requests will be handled sequentially. No matter what the volume of data is, a connection between CRC and CRR should be established before transmission. Correspondingly, a disconnection is required after transmission. Due to a large number of requests, the accumulated costs on connection cannot be neglected. And we want to reduce the time spent on network transmission. Allowing for the amount and concurrency, the configuration requests can be submitted to CRR in batch. It means that several configuration information segments are packed and transmitted together, and the connection is needed only once for a batch.

When a service is deployed successfully, CRC gets the service replica information in deployment module. Then CRC creates a configuration information segment based on the replica information and the address for reverse proxy. The information segment stands for a configuration request, and it is temporarily stored in a buffer, as shown in Figure 3. We use the parameter α to denote the number of configuration requests in a batch. The buffer does not flush until the the number of requests equals the specified batch size α. Before transmission, CRC initiates a connection to CRR. CRR accepts these modification requests and parses the messages, which contain information on new deployed service replicas. CRR integrates all the configuration information which is useful at present, and writes it into the file stored on disk. At last, Nginx executes the reload command to make the new file effective.

The variable s is the time that all the configuration requests are responded successfully. It should be noted that the value of α is not the bigger the better. With n concurrent requests, the optimal s achieves when α equals an appropriate value, which is called threshold. If α is much smaller than the threshold, the optimization is not obvious. Particularly, when $\alpha = 1$,

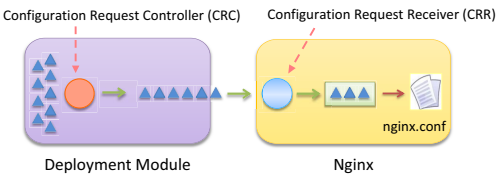

Fig. 4: BFP

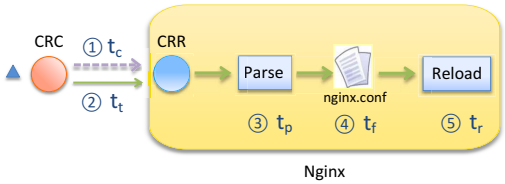

Fig. 5: Main Parts of The Response Time

it is equivalent to sequential processing. In batch scheduling problems, a response time occurs when a new batch starts. Requests are finished when all requests in this batch have been responded. Clearly, there is a trade-off between minimizing the total response time and minimizing the request waiting time within a batch [23]. It is also equivalent to the trade-off between increasing or decreasing the size of batch.

B. Batch File Processing (BFP)

In the general case, when a configuration request is accepted by CRR, CRR parses it and creates the new configuration content, which contains the parsing result and other configuration command for Nginx. Before writing the new content into the configuration file, the file open operation is inevitable, and the close operation is also needed after writing. When the reload command is executing, Nginx parses the configuration file and calls the corresponding commands. It will bring file open, read and close operations as well. If one configuration request is being processed, the following requests will wait in a queue. And the concurrent requests are processed sequentially since the static configuration file is a competitive resource. This pattern is of low concurrency, and a certain amount of time is taken up by file operations.

The optimization method is implemented as shown in Figure 4. CRR parses each request, and the parsing result segment is kept in a buffer. In order to enhance the concurrency, and decrease the time of file operations, several result segments can be organized in batch, and they will be written into the file together. Parameter β denotes the number of information segments in a batch. If the number of result segments in this batch equals β, these segments will be added to a specialized registry. Since the information of existing services should also be contained in the new configuration content. The registry is used for recording all the valid configuration information at present, which is an in-memory buffer. Similarly, when a service is undeployed, the corresponding item will be removed from the registry. Then CRR creates the new configuration file based on the updated registry.

The motivation of this method is to reduce the times of file operations. The moment to create a new configuration file depends on parameter β. However, the value of β is not the bigger the better. It means that the value of s will not always decrease with β gradually increasing. The value of β is the key to get the best response time, and the threshold is defined as the optimal value of β. If the value assigned to parameter β is much smaller than the threshold, it is hard to show the advantages brought by BFP. Particularly, when $\beta = 1$, it is equivalent to sequential processing. If the value of β is much bigger than the threshold, the information segment which arrives at CRR early will be waiting for the other segments, and the waiting time will bring a negative impact.

C. In Memory Configuration (IMC)

Nginx supports hot deployment. If the content of the configuration file is modified, a reload command will make it effective. On receiving a service configuration request, a new mapping is to be established between the reverse proxy and the back-end servers. If a service undeployment request comes, the corresponding module will remove the service replicas, and the mapping will be deleted from configuration file. Notice that the reload command produces file read operations on disk. With intensive requests, the frequency of modifying and reloading rises, and it will take much time.

In view of the difference of speed between disk and memory, we decide to deal with the data in memory instead of disk, and we call this method In Memory Configuration (IMC). When CRR creates the new configuration content based on the registry, it puts the content in a shared memory in place of writing it to disk. If Nginx needs to execute the reload command, the content it parses will come from the shared memory. Under a large number of configuration requests, this method may bring noticeable effect in reducing time of reading file, since memory is much faster than disk.

V. Request Processing Analysis

Based on the dynamic reverse proxy framework we have built, the main parts of response time during request processing are shown in Figure 5. Variable t_c denotes the time of connection and disconnection between CRC and CRR. t_t denotes the time of request transmission. t_p stands for the time of message parsing. t_f is the time of file operations (open, write, and close), and t_r denotes the time of executing reload command. We define the total number of concurrent requests per second as n, and the total response time of n requests is represented by s.

When the configuration requests are processed sequentially, the response time of each request t_{req} can be denoted as equation (1). Therefore, the total response time s is denoted as equation (2).

$$t_{req} = t_c + t_t + t_p + t_f + t_r \tag{1}$$

$$s = \sum_{i=1}^{n} (t_{req})_i \tag{2}$$

A. Time With BRC

We analyse the time when optimization approach BRC is applied in the preceding framework. BRC acts on the first three parts in Figure 5. Parameter α denotes the batch size, and the number of batches is $\lceil n/\alpha \rceil$. The time spent on the first three parts with one batch is denoted by t_α:

$$t_\alpha = \Delta t_w + t_c + t_t + t_p \tag{3}$$

Here Δt_w stands for the total waiting time of requests in this batch. Due to batch committing, the request which comes early will wait for the other requests in this batch. and the effective time of each request is delayed. Since the concurrent requests are intensive per second, the waiting time Δt_w is such a small value that it can be ignored. With n concurrent requests, the time s_1 spent on the first three parts is represented as equation (4).

$$s_1 = \sum_{i=1}^{\lceil n/\alpha \rceil} (t_\alpha)_i \tag{4}$$

When a proper value called threshold is assigned to α, although BRC brings Δt_w, the total response time is still much smaller than that of the original case. If α is much smaller than the threshold, it will result in a sequential processing approximately. If α is much bigger than the threshold, the delay resulted from each batch is prominent.

B. Time With BFP

When BFP method is adopted in the dynamic reverse proxy framework, it acts on the fourth and the fifth parts in Figure 5. Parameter β denotes the number of information segments of each batch, and the number of batches is $\lceil n/\beta \rceil$. The time spent on the fourth and the fifth parts with one batch is denoted by t_β, which is described as equation (5).

$$t_\beta = \Delta t_w + t_f + t_r \tag{5}$$

Before BFP, each configuration request is parsed separately. The batch file processing begins when the the number of information segments equals batch size β, and the the segment which comes early will wait for the other segments belonging to this batch. Δt_w denotes the waiting time of this batch, and it will produce a delay accordingly.With n concurrent requests, the time s_2 spent on the fourth and the fifth parts is represented as equation (6).

$$s_2 = \sum_{i=1}^{\lceil n/\beta \rceil} (t_\beta)_i \tag{6}$$

The time spent on the last two parts in Figure 5 will decrease due to BFP. If the value of β is equal to the threshold, the waiting time will not have a great influence on the total response time. When β is much smaller than the threshold, the times of file operations cannot drop obviously. It is important to note that the value of s_2 will not always decrease with β increasing. If β exceeds the threshold, the value of t_β will rise obviously, since the procedure executed in the batch is similar to sequential processing. Thus the optimization effect will be weakened.

C. Time With IMC

With IMC, the configuration content is stored in a shared memory, and disk operation is replaced by memory operation. If IMC is the only approach applied in the dynamic reverse proxy framework, the last two parts in Figure 5 are related to it. The parameters t_f and t_r in equation (1) are replaced by t'_f and t'_r respectively. Here the parameters t'_f and t'_r are based on the memory operations. The optimization effect is more evident with a higher frequency of file operations. So the response time of each request t'_{req} can be described as equation (7).

$$t'_{req} = t_c + t_t + t_p + t'_f + t'_r \tag{7}$$

D. Response Time With Three Optimization Approaches

If three optimization approaches are applied simultaneously, with n concurrent requests, the total response time s is denoted as equation (8).

$$s = s_1 + \sum_{i=1}^{\lceil n/\beta \rceil} (\Delta t_w + t'_f + t'_r)_i \tag{8}$$

Here s_1 is referred to equation (4), and the parameters t'_f and t'_r are based on memory operations.

VI. Experiment Results And Analysis

In order to verify the optimization effect, several experiments are presented in this section. A large number of same web pages are used as the targets to test the average response time. Each web page has three replicas, which are deployed on three back-end web nodes. We choose Apache Tomcat 7.0 as the back-end web server. The operation system runs on each computer is Ubuntu 12.04. In these experiments, we leave out the time of deployment, and we select the default algorithm round-robin as the load balancing strategy.

A. Original And Optimized Throughput

As shown in Figure 6, the optimization result is obtained with three optimization approaches. The values of α and β are based on a large number of experiment results. And the configuration content parsed by Nginx is stored in a shared memory.

Compared with the original case, the optimization result has a larger throughput per second, and the ART (Average Response Time) is far less than that of the original case. The great difference of ART between two conditions have validated the optimization effect.

B. Batch Size α With Response Time

In the description of BRC, we have mentioned that batch size α has influence on ART, and the value is not the bigger the better. Figure 7 supports this statement. For example, when 500 requests come per second, the ART drops dramatically at first, since a batch of requests saves much time. $\alpha = 64$, ART achieves the minimum. With the continual increase of α, ART goes up gradually. Without BRC ($\alpha = 1$), the ART of each request is about 46.54ms. In the optimization condition

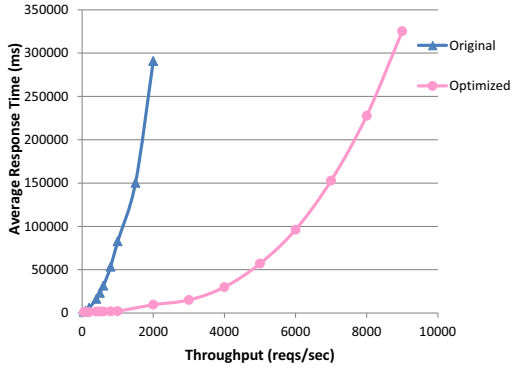

Fig. 6: Comparison of Throughput

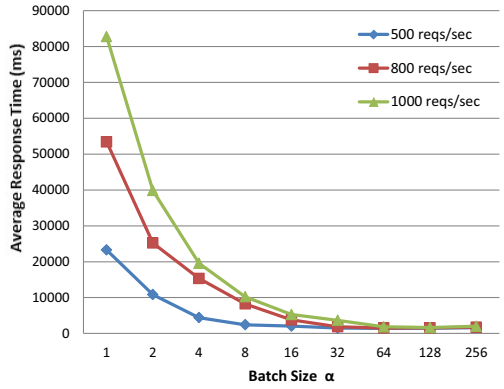

Fig. 7: ART With Batch Size α

($\alpha = 64$), the ART of each request is about 2.82ms, which has declined by 93.94%.

The time of BRC depends on the last request in this batch. If the value of α exceeds the certain threshold, the waiting time of requests which arrive early will have a negative impact on ART. Based on the analysis above, it is clear that the optimization effect of BRC is closely related to the value of α.

C. BRC Results Analysis

In original condition, a configuration request is sent to CRR by CRC as soon as it generates. Then CRR creates a new configuration file and reloads it instantly. Concurrent configuration requests are handled sequentially. Figure 8 displays the ART with BRC optimization method. For each case, we obtain several results based on different values of α, and we select the smaller ART to represent the corresponding case. As the number of requests rises, the advantage of BRC becomes more evident.

The average delay of each request with BRC is shown in Table I. Compared with the average delay in original condition, average delay sharply declines with BRC optimization. Particularly, when the number of concurrent requests is 600 per second, the drop is up to 95.42%.

BRC reduces the times of message transmission and file

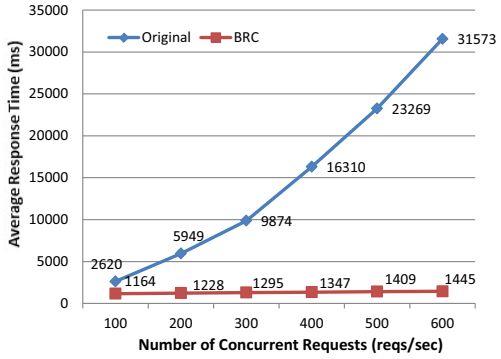

Fig. 8: ART With BRC

TABLE I: Average Delay With BRC

Number of Concurrent Requests (reqs/sec)	100	200	300	400	500	600
Original Average Delay (ms/req)	26.20	29.75	32.91	40.78	46.54	52.62
BRC Average Delay (ms/req)	11.64	6.14	4.32	3.37	2.82	2.41
Decrease By (%)	55.57	79.36	86.87	91.74	93.94	95.42

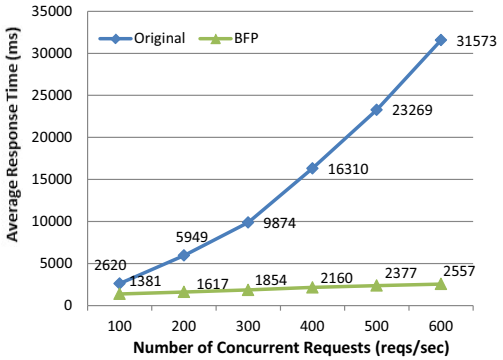

Fig. 9: ART With BFP

creation. Due to the intensive requests and appropriate value of α, the transient delay of each request can be ignored. However, the benefit it brings is significant.

D. BFP Results Analysis

As shown in Figure 9, BFP is effective in decreasing ART of different numbers of requests, especially with a great number of requests in unit time. Parameter β is defined as the number of information segments in a batch.

For a certain number of configuration requests, with the increase of β, the ART decreases gradually. When β exceeds the certain threshold, the trend will change. Since the value of β is not the bigger the better.

Table II shows the average delay of each request with BFP. With the increase of concurrent requests, the average delay of each request goes up in original condition. But it declines when it comes to BFP. It suggests that the drops brought by the optimization is gradually rising.

TABLE II: Average Delay With BFP

Number of Concurrent Requests (reqs/sec)	100	200	300	400	500	600
Original Average Delay (ms/req)	26.20	29.75	32.91	40.78	46.54	52.62
BFP Average Delay (ms/req)	13.81	8.09	6.18	5.40	4.75	4.26
Decrease By (%)	47.29	72.81	81.22	86.76	89.79	91.90

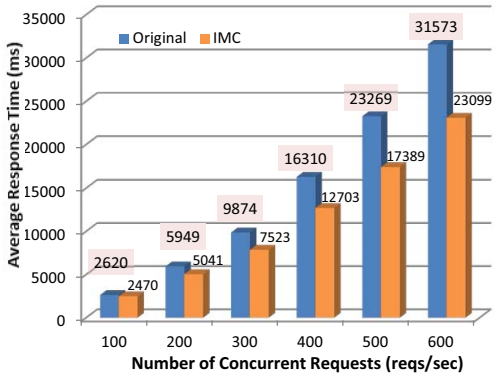

Fig. 10: ART With IMC

TABLE III: Average Delay With IMC

Number of Concurrent Requests(reqs/sec)	100	200	300	400	500	600
Original Average Delay(ms/req)	26.20	29.75	32.91	40.78	46.54	52.62
IMC Average Delay(ms/req)	24.70	25.21	26.20	31.76	34.78	38.50
Decrease By (%)	5.73	15.26	20.39	22.12	25.27	26.83

E. IMC Results Analysis

The idea of IMC comes from the fact that memory is much faster than disk. Therefore, under a large number of concurrent requests, the frequency of file read/write will be high, and the strength of IMC will be distinct. Figure 10 shows the result of ART with IMC optimization. If Nginx executes the reload command, the content to be parsed will come from a shared memory. To some extent, this improvement saves the time on file operations.

The average delay with this optimization approach is shown in Table III. with the growth of concurrent requests, the number of file read/wite operations also goes up, and the optimization effect becomes evident gradually.

VII. Conclusion

In this paper, we concentrate on the concurrent configuration requests processing based on the reverse proxy Nginx. In order to improve the performance, we extend Nginx as a dynamic reverse proxy. In addition, based on the dynamic reverse proxy framework, three optimization approaches are proposed to get a better elasticity. Three methods including Batch Request Committing (BRC), Batch File Processing (BFP) and In Memory Configuration (IMC), and they are from different points of view. We compare the maximum throughput between original and optimized cases, and discuss the relationship between the batch size and the response time. Finally, some experiments have been presented to validate the optimization effect.

The current appropriate values of batch size and the number of group members are obtained by a large number of experiment results. In real application the values are needed to be trained or estimated several times, and a proper method is expected in future.

Acknowledgment

This work was supported by China 863 program (No. 2012AA011203, No. 2011AA01A202), National Natural Science Foundation of China (No. 61103031, No. 61370057), A Foundation for the Author of National Excellent Doctoral Dissertation of PR China, Beijing Nova Program, Specialized Research Fund for the Doctoral Program of Higher Education (No. 20111102120016), the State Key Lab for Software Development Environment (No. SKLSDE-2012ZX-12).

References

[1] P. Mell and T. Grance, "The nist definition of cloud computing (draft)," *NIST special publication*, vol. 800, p. 145, 2011.
[2] "Platform as a service (paas): What is it? why is it so important?" 2012, http://www.njvc.com/resource-center/white-papers-and-case-studies/platform-service.
[3] Google App Engine, http://developers.google.com/appengine/.
[4] Windows Azure, http://www.windowsazure.com/.
[5] Nginx, http://nginx.org/.
[6] HAProxy, http://haproxy.1wt.eu/.
[7] Force.com, http://www.force.com/.
[8] Q. Zhang, L. Cheng, and R. Boutaba, "Cloud computing: state-of-the-art and research challenges," *Journal of Internet Services and Applications*, vol. 1, no. 1, pp. 7–18, 2010.
[9] AppScale, http://www.appscale.com/.
[10] C. Bunch, N. Chohan, C. Krintz, J. Chohan, J. Kupferman, P. Lakhina, Y. Li, and Y. Nomura, "An evaluation of distributed datastores using the appscale cloud platform," in *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*. IEEE, 2010, pp. 305–312.
[11] Second-level domain, http://en.wikipedia.org/wiki/Second-level_domain.
[12] xconf-nginx-module, https://github.com/kindy/xconf-nginx-module.
[13] Service4All, http://www.service4all.org.cn/.
[14] Heroku, http://www.heroku.com/.
[15] Sina App Engine, http://sae.sina.com.cn/.
[16] V. Talwar, D. Milojicic, Q. Wu, C. Pu, W. Yan, and G. Jung, "Approaches for service deployment," *Internet Computing, IEEE*, vol. 9, no. 2, pp. 70–80, 2005.
[17] T. Liu, T. Lu, and Z. Liu, "An optimization approach for service deployment in service oriented clouds," in *Computer Supported Cooperative Work in Design (CSCWD), 2010 14th International Conference on*. IEEE, 2010, pp. 390–395.
[18] N. J. Kansal and I. Chana, "Cloud load balancing techniques: A step towards green computing," *IJCSI International Journal of Computer Science*, no. 9, p. 1, 2012.
[19] G. Barish and K. Obraczke, "World wide web caching: Trends and techniques," *Communications Magazine, IEEE*, vol. 38, no. 5, pp. 178–184, 2000.
[20] Linux Virtual Server, http://www.linuxvirtualserver.org/.
[21] "June 2013 web server survey," http://news.netcraft.com/archives/2013/06/06/june-2013-web-server-survey-3.html.
[22] A. Padilla and T. Hawkins, "Chapter 6 choosing the right web server," in *Pro PHP Application Performance*. Apress, 2011, pp. 131–163. [Online]. Available: http://dx.doi.org/10.1007/978-1-4302-2899-8_6
[23] G. Mosheiov and D. Oron, "A single machine batch scheduling problem with bounded batch size," *European Journal of Operational Research*, vol. 187, no. 3, pp. 1069–1079, 2008.