Dynamic Priority based Weighted Scheduling Algorithm in Microservice System

Yan Xu, Yanlei Shang*
Beijing University of Posts and Telecommunications, Beijing, China

*Corresponding author e-mail: shangyl@bupt.edu.cn

Abstract. With the development of cloud computing and container virtualization, the microservice architecture has received more and more attention. While microservices bring many benefits, they also bring some challenges. A user's request may require multiple microservices to collaborate, and chained invocations between microservices result in higher latency than a monolithic architecture. In this paper, we propose a dynamic priority based weighted scheduling algorithm for the problem that the latency of chained-service requests is too long in the case of high concurrency. Experiments show that the algorithm can reduce the latency of chained-service requests effectively in the case of little impact on the latency of leaf-service requests.

1. Introduction

With the development of computer technology and network technology, people need to analyze and design more and more efficient and reliable large-scale software systems, such as real-time social software for global users, Twitter, facebook, wechat, etc., and online e-commerce platform Amazon, Alibaba (Taobao) and so on. In the design phase, good architecture design is the premise and basis for large software systems to function properly[1].

In recent years, with the development of cloud computing and container virtualization, the microservice architecture has received more and more attention. The basic idea is to split the traditional monolithic application into a series of microservices that can be developed, managed and scaled independently[2]. Compared with the traditional monolithic architecture, the microservice architecture has the following advantages: First, high cohesion and low coupling between microservices, each microservice has a clear functional boundary, focusing only on completing one task; second, the microservices in the microservice system can be deployed independently to achieve sustainable integration and delivery; Third, each microservice can adopt different technical solutions; more importantly, the microservice architecture is more scalable. When the system load is too large, it has better horizontal scalability.

However, while microservices bring the above benefits, they also bring some disadvantages. Since the microservice application system is composed of multiple microservices, the user's request may require multiple microservices to cooperate to complete the response. Compared with the monolithic architecture, the chained invocation between the microservices leads to higher latency.

In this paper, we propose a Dynamic Priority based Weighted Scheduling algorithm(DPWS), which can reduce the latency of chained-services in microservice systems by reducing queuing latency. We divide the services in the microservices system into two categories: leaf-services and chained-services. A leaf-service is a service that a single microservice can complete the response. A chained-service is a service that requires multiple microservices to invoke each other to complete the response. When an
HTTP request arrives, we can determine the type of service by the url of the request, put it into a different request queue according to the service type, and then fetch the request completion response from the request queue according to the DPWS algorithm. Experiments show that this algorithm can effectively reduce the latency of chained-services in the case of less impact on leaf-services request latency.

2. Related Work
In order to optimize the application system of cloud services, many researchers are working to reduce the response latency of user requests. In a cloud-based application system, the time from the request to the receipt of the response can be divided into three parts. The first part is the network latency, the second part is the service time, and the third part is the queuing latency[3]. The network latency is the latency of an HTTP request transmitted in the network. In [4], Webb et al. proposed the recent server allocation to reduce client-server latency, some authors introduced game theory into this topic to optimize network latency. The service time is the time a request is served at a microservice. Some research is focused on reducing service latency, such as increasing caching, SQL optimization and so on. The queuing latency refers to the queuing time of requests to wait for services at various services. As described in [7], some studies have begun to pay attention to interaction latency.

![Figure 1. processes of Chained-service request and Leaf-service request](image)

Take the following scenario as an example. As shown in Figure 1, Service A, Service B, and Service C are three different microservices, all of which can provide leaf-services. The requests of the user1, the user2 and the user3 are all requests of the leaf-service. Therefore, only one micro service is needed to complete the response, and these requests only need to be queued at a single service. However, the request of the user4 is a request of the chained-service, and the services A, B, and C need to cooperate to complete the response. When the request d of the user 4 arrives, the request first needs to be queued at the service A, waiting for the processing of the service A. Service A needs the data of service B in the process of processing the request, so it needs to send the request d’ to service B. While waiting for the response of the request d’, the service A will hang up the request e of the user4. Likewise, the request d’ arrives at the service B, and will be queued at the service B, waiting for the response of the service B. Service B needs the data of service C in the process of processing the request, so it needs to send the request d” to service C. While waiting for the response of the request d”, the service B will hang up the request d’. In the same way, Request d”’ arrives at service C, and needs to be queued at service C. Service C completes processing and returns response to service B. After receiving the data of the service C, the service B completes the processing procedure and returns
the response to the service A. In the same way, service A returns the response to user4. In such an application scenario, the user’s leaf-service request only needs to be queued at a single service, while the request of chained-service needs to be queued multiple times at multiple services, which undoubtedly causes the latency of chained-service request to be too long.

For the above scenario, we design a DPWS algorithm, which can reduce the latency of chained-service requests in the case of less impact on latency of leaf-service requests. The next section will give a detailed introduction.

3. ARCHITECTURE
In this section, we will take the Web AR management platform based on the microservice architecture as an example. First, we will introduce the structure of the Web AR management platform. Second, we will introduce the system's workflow and the design of the DPWS algorithm.

3.1. Web AR management platform architecture
The architecture of the Web AR management platform is shown in Figure 2. The microservice system consists of API Gateway, Service Management Microservice (Service A), Lottery Management Microservice (Service B), and User Management Microservice (Service C). There are also a service registration discovery center and a service monitoring center. Among them, the API gateway is mainly responsible for request routing, providing a unified portal for users.

![Figure 2. Web AR management platform architecture diagram](image)

3.2. Workflow and the DPWS algorithm
Next, we will introduce the workflow of the Web AR management platform. A user’s request first arrives at the API gateway, and the API gateway forwards the request to the corresponding microservice according to the URL requested by the user. If the user's request is a leaf-service request, the API gateway forwards the request to a single microservice, and the microservice completes the request independently and returns the response to the user; if the user's request is a chained-service request (take service A as an example), the API gateway forwards the request to Service A, Service A needs the data of Service B to complete the response, so it needs to invoke Service B; likewise, Service B needs to invoke Service C. With such a chained invocation, Service A can return a response to the user.

Obviously, the request of the leaf-service only needs to be queued at one service, but the request of the chained-service needs to be queued at multiple services. In the case of high concurrency, if the queuing latency of the chained-service is too long, the latency of the chained service request may be much higher than that of the leaf-service request. Moreover, every time a request is passed from one microservice to another, the resources of the system occupied by the request will increase. If the latency of the chained-service request exceeds the user's patience, the user will give up waiting, so the system resources will be wasted. In order to reduce the queuing latency of chained-service requests...
and improve the resource utilization of the system, we designed the DPWS algorithm, which can reduce the overall latency of chained-service requests by reducing the queuing latency.

**Figure 3.** diagram of DPWS algorithm

As shown in Figure 3, each microservice maintains multiple weighted fair queues. The higher the priority of the queue, the more server resources it can occupy. When a request arrives at a microservice, the microservice will put it in different queues according to the type of request. When the request of the leaf-service comes, the microservice will put it into the queue $Q_1$ whose priority is 1. The request of the chained-service is prioritized 1 at the first microservice, and every time it is passed from one microservice to another, its priority is incremented by one. The specific priority adjustment process is as follows: When the request of the chained-service arrives at the first microservice (Service A), the request has the same priority as the leaf-service request at this time, so Service A puts it into the queue $Q_1$ whose priority is 1; when Service A needs to invoke Service B, it will send a request to Service B. When the request arrives at Service B, the priority is raised, and Service B puts it into the queue $Q_2$ whose priority is 2; when Service B needs to invoke Service C, it will send a request to Service C. When the request arrives at Service C, the priority is raised again, so Service C will put it into the queue $Q_3$ whose priority is 3; if C needs to invoke other services, and so on. The specific implementation is explained in the next section.

4. Implement

The Web AR management platform based on the microservice architecture is implemented by the Spring boot + Spring cloud framework, which integrates the Zuul gateway, the Eureka service registration discovery center, and the Zipkin error tracking mechanism. The following focuses on the implementation of the Dynamic Priority based Weighted Scheduling algorithm (DPWS). Before that, we will introduce the Generalized Processor Sharing (GPS) algorithm.

4.1. GPS algorithms

The GPS algorithm is used for congestion control in packet switching. The GPS algorithm is an ideal scheduling strategy that provides accurate and fair bandwidth allocation, proportionally allocating the egress capacity based on the minimum bandwidth required for all waiting queues. The GPS scheduling algorithm is implemented based on an ideal flow model, which assumes that the GPS scheduler can serve all waiting queues at the same time, the output link capacity can be divided into infinitesimal units and allocated to each queue. However, in an actual system, the scheduler can only serve one queue at a time, and the packets cannot be broken down into smaller units. So in the actual system, we can simulate the GPS algorithm by other algorithms.

4.2. Dynamic Priority based Weighted Schedule algorithm (DPWS)

We apply the idea of GPS to the microservice system. When the HTTP request arrives, it is put into a different queue according to the type of the requested service. Take the queue in Service B as an example. Because Service B provides two types of services, one is the leaf-service which is requested by the HTTP request from the user terminal; the other is the chained-service which is requested by the HTTP request from Service A. So there are two queues in Service B, and the priorities are 1 and 2.

We use the concept of virtual time to simulate the time when the packet leaves in the GPS algorithm by virtual finish time. The virtual time has two parts: 1) virtual system time $V(t)$; 2) virtual start
service time and virtual finish service time. After each request (the kth request of the i-type request) arrives, it is respectively enqueued, and we calculate the virtual finish service time $F_{i,k}$ corresponding to the request. $L_i$ is calculated as the average service time of the type request corresponding to the request; $P_i$ is the priority of this queue; $a_{i,k}$ is the time when the request arrives. $A$ represents the set of all queues; $B(t)$ represents the set of all blocking queues in the system at time $t$.

The virtual system time $V(t)$ is the "coordinate time" shared by all queues (in any busy period, the virtual time starts from 0), the update method is:

$$V(0) = 0$$
$$V(t + \tau) = V(t - \tau) + \sum_{i \in A} \sum_{j \in B(i)} \frac{P_i}{P_j} \text{ for } \tau \leq t - t_j - 1, j = 2, 3, \ldots$$

The request’s virtual start service time $S_{i,k}$ and virtual finish service time $F_{i,k}$ are calculated by the following formula:

$$S_{i,k} = \max\{F_{i,k-1}, V(a_k)\}$$
$$F_{i,k} = S_{i,k} + \frac{L_{i,k}}{P_i}$$

The request of each queue head are sorted according to virtual finish service time, and the request with the smallest virtual finish service time accepts service firstly. The pseudo code is as follows:

Requests arrive in different queues
For each queue
    queue[i].finishTime = queue[i].preFinishTime + $L_i/P_i$
Request Extract() for all queues
    min.finishTime = Min(queue[i].finishTime)
for(i=0;i<num.queues;i++)
    handleRequest(queue[i]) = Min(queue[i].finishTime)

5. Experiment and Evaluations

We deployed three microservices, API gateways, service registration discovery centers, and service monitoring centers in different docker containers. The physical machines used Ubuntu and the network cards used 100Mb/s.

To verify the effectiveness of the DPWS algorithm, we will do the following experiments:
We use JMeter persistence to alternately send HTTP requests for leaf-services and chained-service to Service A, Service B, Service C, and then observe the average latency of leaf-services and chained-service requests from the Service Monitoring Center.

The experimental results are shown in Figure 4, where (a) indicates the request latency of the two type of services without the DPWS algorithm, and (b) indicates the request latency of the two type of services after using the DPWS algorithm. We did ten experiments separately. The experimental results show that the DPWS algorithm can effectively reduce the latency of chained-service requests, and has little impact on the latency of leaf-service requests.
6. Conclusion
Aiming at the problem of excessive latency of chained-service response in microservice system under high concurrency, this paper proposes the DPWS algorithm, which uses weighted fair queues to improve the priority of chained-service request dynamically. Experiments show that the algorithm can effectively reduce the request latency of the chained-service in the case of little impact on the latency of leaf-service requests, and achieve the state of latency balance between chained-service and leaf-service.

Acknowledgments
This work was supported by the National Natural Science Foundation of China under Grants No. 61671081 and No. 61720106007, the Beijing Natural Science Foundation under Grant No. 4172042, and the 111 project (B18008), the Fundamental Research Funds for the Central Universities under Grant No. 2018XKJC01.

References
[1] van Vliet H, Tang A. Decision making in software architecture[J]. Journal of Systems and Software, 2016, 117: 638-644.
[2] Marcelo Amaral et al., “Performance Evaluation of Microservices Architectures Using Containers”, Proc. 2015 14th International Symposium on Network Computing and Applications, IEEE, 2015, pp. 27-34.
[3] Yuchao Zhang et al., “Going Fast and Fair: Latency Optimization for Cloud-Based Service Chains,” Proc. IEEE Network, 2018, 32:138 - 143.
[4] S. D. Webb, S. Soh, and W. Lau, “Enhanced Mirrored Servers for Network Games,” Proc. 6th ACM SIGCOMM workshop on Network and System Support for Games, ACM, 2007, pp. 117–22.
[5] J. Guo et al., “A Cooperative Game Based Allocation for Sharing Data Center Networks, Proc. IEEE 2013 INFOCOM, IEEE, 2013, pp. 2139–47.
[6] K. Xu et al., “Online Combinatorial Double Auction for Mobile Cloud Computing Markets,” Proc. 2017 IEEE Int’l. Performance Computing and Communications Conf. (IPCCC), IEEE, 2014, pp. 1–8.
[7] H. Wang et al., “On Design and Performance of Cloud-Based Distributed Interactive Applications,” Proc. 2014 22nd IEEE Int’l. Conf. Network Protocols (ICNP), IEEE, 2014, pp. 37–46.