TCLBM: A Task Chain-Based Load Balancing Algorithm for Microservices

You Liang  
School of Computer Science and Engineering, Beihang University, Beijing 100191, China

Yuqing Lan  
School of Computer Science and Engineering, Beihang University, Beijing 100191, China

Follow this and additional works at: https://tsinghuauniversitypress.researchcommons.org/tsinghua-science-and-technology

Part of the Computer Sciences Commons, and the Electrical and Computer Engineering Commons

Recommended Citation
TCLBM: A Task Chain-Based Load Balancing Algorithm for Microservices

You Liang and Yuqing Lan*

Abstract: The microservices architecture has been proposed to overcome the drawbacks of the traditional monolithic architecture. Scalability is one of the most attractive features of microservices. Scaling in the microservices architecture requires the scaling of specified services only, rather than the entire application. Scaling services can be achieved by deploying the same service multiple times on different physical machines. However, problems with load balancing may arise. Most existing solutions of microservices load balancing focus on individual tasks and ignore dependencies between these tasks. In the present paper, we propose TCLBM, a task chain-based load balancing algorithm for microservices. When an Application Programming Interface (API) request is received, TCLBM chooses target services for all tasks of this API call and achieves load balancing by evaluating the system resource usage of each service instance. TCLBM reduces the API response time by reducing data transmissions between physical machines. We use three heuristic algorithms, namely, Particle Swarm Optimization (PSO), Simulated Annealing (SA), and Genetic Algorithm (GA), to implement TCLBM, and comparison results reveal that GA performs best. Our findings show that TCLBM achieves load balancing among service instances and reduces API response times by up to 10% compared with existing methods.

Key words: load balancing; microservices; task chain

1 Introduction

Microservices are a popular architectural style wherein an application is structured as a collection of services[1], thus overcoming some drawbacks of the traditional monolithic architecture, particularly its lack of scalability[2]. As suites of loosely coupled and independently deployable services, microservices-based applications are structured in a modular manner[3]. Compared with that in the monolithic architecture, scaling in the microservices architecture requires the scaling of specified services only, rather than the entire application[4]. Deploying instances of the same service on multiple physical machines is a common way to achieve scaling[5]. However, scaling services could encounter complex load balancing problems due to duplicate service instances.

Microservices-based back-end applications typically provide Application Programming Interfaces (APIs) for clients, such as server-side web applications and mobile clients. API requests are delivered to the API gateway[6], which calls different services based on API requests. One API call usually consists of multiple service calls executed on multiple physical machines. In this paper, service calls for an API request are called a task chain, and service instances for these service calls are called a service instance chain. Tasks with direct dependencies may transfer data between physical machines. Since any service can be deployed multiple times, more than one candidate exists for the target service of a service call[7]. Inappropriate target service instances lead to load imbalance and superfluous data transmissions, thereby increasing the API response time.

* You Liang and Yuqing Lan are with the School of Computer Science and Engineering, Beihang University, Beijing 100191, China. E-mail: [liangyou, lanyuqing]@buaa.edu.cn.
* To whom correspondence should be addressed.
Manuscript received: 2019-07-17; accepted: 2019-07-25
To summarize, this work makes the following contributions.

- We take full advantage of task dependencies by analyzing the entire API call to reduce data transmissions between physical machines.
- We propose TCLBM, a task chain-based load balancing algorithm for microservices, to select service instances for each task at runtime. TCLBM takes both system resource usages and the number of data transmissions between physical machines into account.
- We implement TCLBM by using three heuristic algorithms, namely, Particle Swarm Optimization (PSO), Simulated Annealing (SA), and Genetic Algorithm (GA), and devise a series of experiments to evaluate TCLBM; results demonstrate that GA performs best among these algorithms. Overall, TCLBM reduces API response times by up to 10% compared with other related algorithms.

The rest of this paper is organized as follows. Section 2 discusses related works. Then, we detail the methodology of TCLBM in Section 3. Results and a comparison are illustrated in Section 4. Finally, Section 5 outlines the conclusion of this paper.

2 Related Work

Optimizing container scheduling is a way to achieve load balancing in microservices. Guerrero et al. proposed a scalable routing mechanism for private or enterprise users. Kehrer and Blochinger proposed a model-based approach to assist software developers in building microservices as self-configuring containers without being bound to operational technologies; this approach provides developers with a simple configuration model to specify the configuration operations of containers and automatically generate a self-configuring microservices tailored for the targeted runtime environment. Yi al. presented a dynamic weight load balancing algorithm for microservices routing; this work is the most similar to ours. However, while this algorithm only selects objectives reflecting the load condition of the server as parameters for load evaluation, our method takes both system resource usages and the number of data transmissions between physical machines into account.

3 TCLBM Methodology

In the microservices architecture, one service can usually be deployed multiple times on different physical machines to increase the application scale. One task can be executed on different service instances from different physical machines. Choosing an inappropriate service instance leads to load imbalance and increased API response times. In this section, we propose a load balancing algorithm, TCLBM, that focuses on the task chain of the API call and takes system resource usages and data transmissions between physical machines into consideration.

Let a microservices-based application be deployed on n physical machines \(\{p_1, p_2, \ldots, p_n\}\). The application consists of m service instances \(\{s_1, s_2, \ldots, s_m\}\). The task chain of an API call contains q tasks \(\{t_1, t_2, \ldots, t_q\}\). \(s_i = S(t_i)\) is the algorithm to choose service instance \(s_i\) for task \(t_i\). In the rest of this section, we explain the detailed design of TCLBM in three aspects.

3.1 Objective analysis

TCLBM takes into account system resource usages, such as CPU usage, memory usage, and bandwidth usage. Load balancing is achieved by keeping the system usages of the service instances of each service type roughly the same. We use the equations below
to represent the load balance degree among service instances.

\[ B(t_i) = \frac{U^R(t_i) - U^R_{av}(t_i)}{U^R_{av}(t_i)} \]  
\[ E_1 = \sum_R \left( \frac{\sum_{i=1}^q B(t_i)}{q} \right) \]  

In Eq. (1), \( B(t_i) \) represents the relative difference between the usage of one system resource on one service instance and the average value of the service instance type. In Eq. (2), \( E_1 \) is the load balance degree among service instances and calculated as the sum of the average relative differences between each system resource on each service instance.

TCLBM also takes the number of data transmissions between physical machines during the entire API call into account. Because application developers know the operations and purposes of API requests, the task chain for a specified API request is predictable. The execution time for a specific task on each service instance is roughly the same if the configuration of each service instance is the same and they are not working at full capacity. Therefore, when an API request is received, target service instances can be chosen for all tasks in the task chain. TCLBM prefers to service instances in the same physical machine to reduce the number of data transmissions between machines and, finally, reduces the API response time.

\[ D(t_i, t_j) = \begin{cases} 
1, & t_i \text{ calls } t_j; \\
2, & t_i \text{ requires the callback from } t_j; \\
0, & \text{otherwise}
\end{cases} \]  

Three types of dependency relationships exist between two tasks. One is task \( t_i \) calls task \( t_j \), which causes one data transmission, i.e., only the request. Another is task \( t_i \) requires callback from task \( t_j \). In this relationship, tasks transfer data twice, i.e., once for the request and once for the response. The third relationship is a no-dependency relationship. As shown in Eq. (3), \( D(t_i, t_j) \) is the number of data transmissions between \( t_i \) and \( t_j \).

\[ P(s_i, s_j) = \begin{cases} 
1, & s_i \text{ and } s_j \text{ are not in the same physical machine;} \\
0, & \text{otherwise}
\end{cases} \]  
\[ Q(t_i, t_j) = D(t_i, t_j) \times P(S(t_i), S(t_j)) \]  

\( Q(t_i, t_j) \) in Eq. (5) is the number of data transmissions between physical machines for \( t_i \) and \( t_j \).

\[ T(t_i) = \sum_{j=0}^q (Q(t_i, t_j) + Q(t_j, t_i)) \]  
\[ E_2 = \frac{\sum_{i=0}^q T(t_i)}{2} \]  

In Eq. (6), \( T(t_i) \) is the number of data transmissions between physical machines introduced by task \( t_i \). \( E_2 \) in Eq. (7) is the value of the entire API call. Because any task can be the caller or callee, the value must be divided by 2.

\[ E = kE_1 + (1-k)E_2 \]  

Both load balance degree \( E_1 \) and the number of data transmissions between physical machines \( E_2 \) play important roles in TCLBM. However, focusing on these two factors at the same time is difficult. As shown in Eq. (8), we combine these two factors as a composite index \( E \) with a weight \( k \) to balance them. This weight differs in different applications and must be estimated through experiments.

### 3.2 Heuristic algorithms

The load balancing algorithm must be executed every time an API request is received. The execution time of the load balancing algorithm is included in the API response time. As an overly complex load balancing algorithm unnecessarily increases API response times, a suboptimal solution resulting in a smaller response time is adequate.

Since heuristic algorithms are designed to find a suboptimal solution within a reasonable time, we use three heuristic algorithms to implement TCLBM, namely, PSO, SA, and GA. These heuristic algorithms are computational methods that rely on a population of candidate solutions and designed to solve multi-dimensional problems. In the service instance chain selection problem, the number of dimensions is the number of tasks in the chain, and the search space is all available service instances for the tasks. All service instances of one service type are collected in a list, numbered \( 1, 2, 3, \ldots, n \). A candidate solution is a sequence of service instance numbers.

Most problems solved by heuristic algorithms are limited in a finite search space. In PSO and GA, candidate solutions must be corrected in every iteration to fit the search space. The common operation here involves clamping solutions between the maximum and minimum. The search space of our problem is
finite but borderless. Service instances are sorted in random order, which means exchanging the first service instance and the last one in the list is harmless. For the current service instance chain selection problem, the index of the service instance is modulated by the number of server instances of the server type if the value is out of range.

Three main parameters affect the performance of PSO: inertia weight $w$, cognitive component $c1$, and social component $c2$. Inertia weight is a parameter used to balance global exploration and local exploitation\cite{14}. The larger the inertia weight, the stronger the global exploration. By contrast, the smaller the inertia weight, the stronger the local exploitation. Cognitive component indicates the tendency of the algorithm to approach the local optimum, while social component represents the tendency of the algorithm to approach the global optimum\cite{15}. At the beginning of the iterations, the search range should be as wide as possible to cover more types of solutions. Under a large inertia weight, large cognitive component, and small social component, particles are allowed to move around the search space, instead of moving toward the global optimum. Particles should then be increasingly inclined to find the best solution based on previous results instead of walking around the search space. Thus, reducing the inertia weight and cognitive component, and increasing social component during the iterations is reasonable.

In SA, the Metropolis-Hastings algorithm is used as the acceptance probability function. A fast annealing schedule instead of the traditional one is used to accelerate the cooling process\cite{16}. Since the execution time of TCLBM makes up part of the API response time, faster searching means less overhead.

GA differs from the two other heuristic algorithms in terms of how candidate solutions change along with the iterations. In PSO and GA, candidate solutions can change freely in the search space with limited speed. However, in SA, crossover and mutation operations are allowed\cite{17}, but the ability of the solution to adjust candidate solutions is limited. In TCLBM, we use two-point crossover and boundary mutation. The probabilities of crossover and mutation operations are identical at 50%. Because the scale of the population and the number of generations are limited, increasing the genetic diversity is appropriate.

### 3.3 Algorithm implementation

TCLBM consists of two main steps. The first step involves choosing a service instance chain for the task chain of an API call when the API request is received. The second step involves choosing the target service instance for a task before the task is executed.

Algorithm 1 represents the selection of an appropriate service instance chain for an API request. First, a task chain is presented for the API request based on the business logic of the application. Since the task chain and all service instances are known, listing all possible service instance chains is only a combinatorial problem. The possible solutions are in the search space of heuristic algorithms. Then, multiple solutions are randomly selected as candidate solutions. In each iteration, the best solution is selected from all candidate solutions. All candidate solutions are updated at the end of each iteration as required by the heuristic algorithm. The best solution is the most suitable service instance chain for the API request.

Before executing each task, the service instance obtained from Algorithm 1 must be checked, as shown in Algorithm 2. If the service instance does not fit the task or its load is salient among all of the candidates, the task must be executed on another service instance. The selection of new service instance only takes system

---

**Algorithm 1** Choose service instance chain for API request

**Input**: an API request, $R$;  
**Output**: a map of task-service-instance pair, $M$;  
1. Set $M$ to null and $E_M$ to MAX;  
2. Get the task chain for the API request, $R$;  
3. Generate a set of feasible task-service-instance maps, $M_x$;  
4. for $i$ from 0 to max iteration generations, $G_{\text{max}}$, do  
5. for $m$ in $M_x$ do  
6. $e_m = E(m)$  
7. if $E_M > e_m$ then  
8. $E_M = e_m$  
9. $M = m$  
10. end if  
11. end for  
12. Update $M_x$;  
13. end for  
14. return $M$

**Algorithm 2** Choose service instance for task

**Input**: task, $T$; target service instance, $S_c$;  
**Output**: corrected target service instance, $S_c$;  
1. $S_c = S$  
2. if $T$ does not fit $S$ or the load of $P$ is salient then  
3. Set $S_c$ to the service instance under the lowest load  
4. end if  
5. return $S_c$
resource usages into consideration. The accuracy of task chain prediction depends on the business logic of the application. If too much time is spent on previous tasks, the load of service instances may differ from the prediction. However, because the API response time of a healthy application cannot be too long, this case is rare.

4 Experiment and Analysis

In this section, we implement TCLBM by using PSO, SA, and GA and evaluate its performance. The experiment platform is the CloudSim framework \[^{[18]}\] and the algorithms are tested on four scales of 4, 8, 16, and 32 physical machines. The configuration of each scale is shown in Table 1. The experiment host machine is configured with 4 CPU cores at 2.8 GHz and 16 GB RAM.

4.1 Objective weight

According to Eq. (8), \(k\) is the unknown weight of two objectives. Previous analysis indicates that \(k\) changes with the application and scale; thus, this weight must be estimated by experiments. We conduct nine experiments for each scale. In these experiments, the weight ranges from 0.1 to 0.9, stepped by 0.1. \(E_1\) in Eq. (2) and \(E_2\) in Eq. (7) are compared to find a suitable weight. The experiment results of the 8-physical machine scale is shown in Fig. 1.

As \(k\) increases, the load balance degree \(E_1\) clearly decreases, and the number of data transmissions between physical machines \(E_2\) increases. We put the minima and maxima of \(E_1\) and \(E_2\) on the same horizontal lines. At the intersection of two polylines, \(k\) is about 0.45. However, the scale of the two objectives is not the same and breaks the balance \(k\) has achieved. The maxima of \(E_1\) and \(E_2\) are used to measure their scales. Finally, we choose 0.96 for \(k\) on the 8-physical machine scale.

4.2 Comparison between heuristic algorithms

We investigate the search abilities of PSO, SA, and GA. Each heuristic algorithm is iterated 300 times with 20 individuals, and the objective is the composite index \(E\) proposed in Eq. (8). The result in shown in Fig. 2.

In the 4-PM scale, the three algorithms similarly find the equivalent solution. In other scales, however, the solution found by GA is better than the solutions found by PSO and SA, mainly because of the way GA changes candidate solutions along with the iterations. Unlike other heuristic algorithms, GA keeps segments between generations, thereby avoiding modification on every dimension. For the current service instance chain selection problem, the solutions are discrete, and the objective values of neighbors may be very different. However, solutions sharing the same segment have approximate objective values because the contribution of the segment to data transmissions between physical machines is the same. The objective values of adjacent solutions is different, and the possibility of moving toward worse solutions is high at the beginning of the search.

We now study the stability of GA on TCLBM. A suboptimal solution that is 30% worse than the optimal solution is considered acceptable. The experiment is repeated 1000 times on four scales and the statistical distribution of the generations to obtain the suboptimal solution is shown in Fig. 3. With increasing size of physical machines, more iterations are required to achieve the suboptimal solution.

4.3 Performance of TCLBM

First, we evaluate the overhead contributed by TCLBM to the API response time. Most of the time required by TCLBM is spent on heuristic algorithms. Thus, we use the cost of GA to represent the overhead of TCLBM. The experiment is repeated 200 times with 300 iterations, and we attempt to find the suboptimal solution on the 32-PM scale. The time cost of GA is shown in Fig. 4. Approximately 2.3 ms is required to
Next, we evaluate load balance degree $E_1$, the number of data transmissions between physical machines $E_2$, the composite index $E$, and API response time among round robin, least request, the algorithm proposed by Yi et al.\cite{13}, and TCLBM. The first two algorithms are widely used in microservices frameworks, such as Istio\cite{19} and Ribbon\cite{20}. The result is shown in Fig. 5.

Yi et al.’s algorithm achieves load balancing effectively because system resource load balancing is the only objective of this algorithm. However, system resource load balancing is not the goal of the three other algorithms. TCLBM significantly reduces the number of data transmissions between physical machines by up to 50% compared with the three other algorithms. Therefore, the composite index of TCLBM is very small. More importantly, TCLBM reduces API response time by up to 10% compared with PSO, SA, and GA.

The number of data transmissions between physical machines does not change linearly with the scale of physical machines. The probability of transferring data between physical machines is small if only a few physical machines are available. The effect of TCLBM is not significant. As the number of physical machines increases, the number of candidates for each service instance selection increases, causing the data trend to be transferred between physical machines. If the number of physical machines is much larger than the number of tasks in API calls, increasing the number of physical machines cannot create more data transmissions.

In the case of a few physical machines, the API
suitable number of physical machines, the time spent on business logic decreases and the effect of the decrease in number of data transmissions is enhanced. However, the improvements introduced by TCLBM to the 16- and 32-physical machine scales do not significantly differ, consistent with the improvement in the number of data transmissions between physical machines. Because most of the API response time is spent on the business logic, the improvement introduced by TCLBM in API response times is smaller than the improvement in the number of data transmissions between physical machines. Service instance chains are not linear but criss-cross. Thus, the time saved for API response times is smaller than the time saved for data transmissions.

5 Conclusion

We propose TCLBM, a load balancing algorithm for microservices. TCLBM selects target service instances for all tasks of an API call when the API request is received and achieves load balancing by taking system resource usages into account. Service instance chains are optimized by reducing the number of data transmissions between physical machines. TCLBM is implemented by three heuristic algorithms, namely, PSO, SA, and GA, and comparison results show GA performs best. We devise a series of experiments and demonstrate that TCLBM achieves load balancing among service instances and reduces API response times by up to 10% compared with other related algorithms.

References


Tsinghua Science and Technology, June 2021, 26(3): 251–258


You Liang received the BS degree from Huazhong University of Science and Technology, China in 2015. He is currently pursuing the MS degree in Beihang University, China. His research interest lies in the area of parallel and distributed processing.

Yuqing Lan is currently a professor in the School of Computer Science and Engineering, Beihang University, China. His research interests lie in the area of operating system, information security, software reliability, and data mining.