Intelligent flow control algorithm for microservice system

Yudong Li¹ | Yuqing Zhang¹ | Zhangbing Zhou¹ | LinLin Shen²

¹School of Information Engineering, China University of Geosciences Beijing, Haidian, Beijing, China
²College of Computer Science and Software Engineering, Shenzhen University, Shenzhen, Guangdong, China

Correspondence
Yuqing Zhang, School of Information Engineering, China University of Geosciences Beijing, Haidian, Beijing, China.
Email: yqzhang@cugb.edu.cn

Funding information
National Key Research and Development Programme of China, Grant/Award Number: 2018YFB1402803

1 | INTRODUCTION

With increasing complexity of computer software, the performance requirements of programs have exceeded the limits of a single computer and a distribution-friendly architecture is urgently needed. Like microservice architecture [1], programs designed with microservice architecture are usually divided into multiple functional modules. While each module is implemented as a small and independent system, communication between modules usually relies on technology-agnostic protocols such as HTTP [2].

Microservices architecture brings many benefits to the software industry, which makes applications easier to understand, develop and maintain, and brings faster delivery, improved scalability and greater autonomy [3]. However, microservices approach is also subject to criticism for a number of issues, especially its stability. Distributed microservice architecture introduces more servers, and unstable infrastructures such that network problems may easily cause service failures [4]. Owing to the complex calling paths between services, the failure of a service on the path may be passed down and cause the entire system to crash. When facing high concurrent service requests, sudden large traffic may cause the system CPU usage and load to soar. As a result, requests cannot be processed in a timely manner. A series of measures to address the issue and ensure service availability have been proposed.

Fault-tolerant mechanisms, such as flow limiting and circuit breaking, are popular methods to maintain the availability of microservice systems [5]. Flow limiting controls the number of requests for services per unit time and prevents sudden request flood from exceeding the system’s processing capacity. Circuit breaking is used to detect failures and encapsulate the logic of preventing failure from constantly reoccurring during maintenance, temporary external system failure or unexpected system difficulties.

While flow control libraries such as Hystrix [6] and Sentinel [7] can help developers quickly implement flow limiting and circuit breaking, and it is still challenging to figure out the capacity of a large dynamic distributed system where flow and latency characteristics are constantly changing. Historically, flow control mechanisms are manually configured by fixed limit rules with the help of performance testing and profiling, which is a complicated and arduous process. It is difficult for even experienced engineers to accurately fit the rules to the system performance. On the other hand, fixed static rules cannot adapt to changing flow conditions and sometimes limit the throughput. Some researchers have...
proposed to simplify the testing process through automated testing [8,9].

In our work, we try to simplify testing process through dynamic flow control to improve system performance. To achieve this goal, we introduce an adaptive and auto-scaling solution to improve system resource utilization.

2 | MODELING MICROSERVICE FLOW

Setting flow threshold is a resource allocation problem, the key is to allocate appropriate amount of resources to a service that is frequently called.

2.1 | Flow characteristics in microservices

In a microservice system, modules are deployed and run independently, and call each other through network interfaces. We consider each module that runs separately as the minimum unit to calculate the flow-limit threshold.

Inside the module, we divide services into downstream services and upstream services, where upstream services need to call their downstream services when they are called. Therefore, the call path of this module can be represented as a directed acyclic graph (DAG), \( G = (V, E) \). The node set \( V \) represents the service flow entry, and the edge set \( E \) represents the service calling relationship. Weight of the node \( x \), the real-time requesting flow of each service, is calculated as the sum of the weights of all the previous nodes. We use \( x^+ \) to represent flow from external calls to the module. An example of such a DAG is shown in Figure 1.

2.2 | Optimization model

As can be seen from above figure, the module’s throughput is determined by the flow of externally accessible upstream service. Therefore, the goal of the flow limit rules is to maximize the sum of \( x^+ \_i \) on the premise that resources are available. Let the service's flow-limit threshold be \( y_i \), the actual passing flow of the node is \( \min(x_i, y_i) \). The total throughput can be expressed as:

\[
T = \sum_{i=1}^{n} \min(x_i, y_i)
\]  

The flow \( X = x_1, x_2, ..., x_i \) need to be fitted to \( Y = y_1, y_2, ..., y_j \) to obtain higher throughput. Mean-square distance can be used to assess the fitness:

\[
MD = \sum_{i=1}^{n} \sqrt{(x_i - y_i)^2}
\]

The flow-limiting rules \( Y \) should obey the following linear constraints:

- **System resource constraints**

The system resources required to process the current number of requests should not be greater than the sum of the node resources:

\[
c_1y_1 + c_2y_2 + ... + c_ny_n \leq m
\]

\( c \) is the load factor of the service, which is depended on the complexity of the task. \( m \) is the total amount of node resources, which can be smaller than the actual value in order to play a protective role.

- **Call path constraints**

When there is a call path between services, the service, flow limit should be greater than the sum of its upstream flow. For example,

\[
y_3 \geq y_2 + y_3
\]

\( service_2 \) and \( service_3 \) are downstream services of \( service_1 \).

- **Maximum-minimum constraints**

In order to reduce the MD, the maximum constraints prevent the threshold from being loosened too much. Since the calculation of the flow limit always delays the actual flow, adding a relaxation factor \( a \in [1, +\infty] \) can prevent jitter. To prevent low-priority services from being overcrowded during burst flow peaks, a minimum value of a constant \( N_i \) is set for flow limit rules.

\[
N_i \leq y_i \leq ax_i
\]
FLOW CONTROL RULES

3.1 Real-time monitoring

In order to adaptively estimate the parameters of the above model and auto-scaling microservice system, real-time metrics need to be collected. Monitoring includes service monitoring and system monitoring. Service monitoring mainly monitors the description of service quality, including response time and queries-per-second. System monitoring includes the operating attributes of interest in real-time monitoring, thread queues and device physical attributes. Table 1 shows the description of service quality, including response time and required to perform a service. In a unit time (1 s), the time required to process the given requests is

\[ t = \sum_{i=1}^{n} \frac{x_i \times \text{minRT}_i}{\text{CpuNum}} \]  

(6)

LoadAvg is the queue length waiting for CPU (the ready queue or run queue), which presents the average system load over a period of time. When LoadAvg > CpuNum, the process queue length is greater than the number of cores. In this case, the system is busy, and service flow should be limited.

To exclude the effects of processes that are not related to the service module running in the system, Loadlevel is defined to indicate the relationship between current flow and system capacity:

\[ \text{Loadlevel} = \sum_{i=1}^{n} \frac{x_i \times \text{minRT}_i}{\text{CpuNum}} \]  

(7)

3.2 Flow-limit threshold

The flow-limit threshold is obtained by solving the optimization model. The input of the model is the current microservice

### Table 1: Monitoring properties

| Property            | Mark       |
|---------------------|------------|
| Minimum response time | minRT     |
| Average response time  | avgRT     |
| Queries-per-second   | QPS(x)     |
| Average CPU usage    | CpuUse     |
| CPU queue length     | LoadAvg    |
| Number of CPU processors | CpuNum |

system topology and real-time monitoring data, while the output is the flow-limit threshold \( Y \) for the next moment.

This problem can be solved using heuristic algorithms or machine learning methods such as ant colony algorithms and recurrent neural networks. These methods can automatically learn parameters and capture non-linear relationships. As this task is sensitive to delay, we prefer to choose a lightweight method.

Since the constraints are all linear, linear optimization algorithm could be used to solve the problem. Since the optimization goal of the model is to maximize system throughput, sum of downstream service flow is used as the objective.

For various business systems, the priority of each service would be different. For example, requests for important services such as ordering and payment should be processed first. Adding the benefit coefficient \( k \) enables the model to respond preferentially to requests for important services. The objective function is:

\[ \text{maxZ} = k_1y_1 + k_2y_2 + \ldots + k_ny_n \]  

(8)

And the problem can be expressed as:

\[ c_1y_1 + c_2y_2 + \ldots + c_my_n \leq m \]

\[ y_i \geq y_j + y_k (j, k \text{ is upstream}) \]

\[ y_i \leq ax_i \]

\[ y_i \geq N_i \]  

(9)

There are multiple methods for solving linear programming problems, such as simplex method, ellipsoid algorithm, which can obtain the optimal solution in a polynomial time.

3.3 Circuit breaking

Circuit breaking degrades a service whose response times out, to avoid the upstream service failures. The response time of different services is different, and a mechanism is needed to determine whether the response time of the service is abnormal. We can simply set a timeout threshold to blow the service, but we tend to use more intelligent methods to achieve better results.

We collect \( RT \) monitor data in a sliding time window \( m \) seconds, and calculate the gradient at each moment:

\[ \text{gradient} = \frac{1}{m} \sum_{i=2}^{m} \frac{RT_i + \beta}{RT_{i-1} + \beta} \]  

(10)

where \( \beta \) is a constant used to prevent gradient explosions caused by small denominators. When the gradient exceeds a certain value, it means that the \( RT \) has increased sharply over a period of time. It can be considered that the service is in an abnormal state and should be downgraded. In this case, the \( y \) corresponding to the service is moved out of the linear optimization goal, and its flow would be automatically set to 0.
A countdown will be started at the same time, and after countdown the optimization goal will be resumed.

Algorithm 1 Dynamic flow control algorithm

Input: The set of current flow of each service $x_i$; Real-time monitoring data $r$; The set of service load factor $c_n$, The set of benefit coefficient $k_n$, Circuit breaking gradient threshold $a$;
Output: A set of flow control rules for next time window $y_n$
1: for each $i \in n$ do
2: calculate gradient $\nabla_i$;
3: if $\nabla_i > a$ then
4: $k_i = 0$;
5: end if
6: end for
7: calculate LoadLevel, $b = \text{LoadLevel}$;
8: if $b < 1$ then
9: $y_i = x_i / b$;
10: else
11: Model linear programming model with monitoring data and $c_n, k_n$;
12: Solve the model to calculate $y_n$;
13: end if
14: Return $y_n$

Algorithm 1 presents the algorithm of our dynamic flow control.

The time complexity of Algorithm 1 is $O(N)$, where $N$ is the number of services. When calculating the threshold through the linear programming model, the main computational costs are from the two-stage simplex method, which in the worst case has exponential complexity. However, manual rules can be introduced to reduce the average number of iterations for simplex to $2n + 1$.

4 | FINE-GRAINED FLOW CONTROL

4.1 | Leaky bucket algorithm

It takes a numerical time window to count the number of requests. For example, QPS refers to the number of requests within $1$ s. This time window is not fixed, sometimes longer or shorter. Algorithm 1 calculates the load based on the average processing capacity. In a long time window, if requests accumulate in a short period of time, and other times are idle, flow control may not be able to protect the system. Ideally, no matter how long the time window is, we hope that the request can arrive at a fixed frequency.

We use the leaky bucket algorithm to shape the flow so that requests can pass at a uniform rate. The leaky bucket algorithm maintains a bucket with fixed capacity. When a request arrives, it is put into the bucket first. The requests in the bucket leak to the services at a fixed rate. The maximum request volume can be controlled by setting the capacity of the bucket.

The leaky bucket is essentially a simple FIFO buffer to remove burstiness or jitter, which was previously used in the packet-switched networks [10] and asynchronous transfer mode (ATM) networks [11]. Here, we regard the service as a control unit, use it in the microservice system to ensure availability, and improve the stability of flow control.

The setting of the leaky bucket corresponds to Algorithm 1 presented in the following form:

$$\text{capacity}_n = \max RT \times y_n$$  \hspace{1cm} (11)

where max $RT$ is usually set by the user and refers to the highest tolerance for $RT$. The request is passed form the bucket every $1/y_n$ second.

4.2 | Adaptive time interval

Although Algorithm 1 has a polynomial time complexity, we still hope to reduce its computational resource overhead as much as possible. In most cases, the flow would not change drastically, there is no need to frequently change the flow control rules. The frequency of executing Algorithm 1 should be adjusted appropriately, and the adjustment of the frequency is related to the trend of the flow. When the flow is stable, the flow limit rules can be updated at a longer interval, but when the flow rate changes greatly, the frequency of adjustment needs to be increased to avoid a decrease in throughput.

Slow-start is a congestion control strategy with an additive increase/multiplicative decrease (AIMD) scheme, which has been used in TCP transport protocol [12]. Based on slow start, we proposed adaptive time interval algorithm to determine the frequency of updating flow control rules.

Use a variable $T$ to indicate the time to wait for the next execution of Algorithm 1, Algorithm 2 presents the process of calculating $T$.

Algorithm 2 Adaptive time interval

Input: The current flow control rules $y_n$; The previous flow control rules $y_{n-1}$; The time elapsed since the last calculation $T$ (seconds); The global variable $k$ is used to record the maximum value of $T$ at the last reset; Update frequency factor $s$;
Output: The time interval for the next calculation $T'$
1: flag ← 0;
2: for each $i \in n$ do
3: if $|y_n - y_{n-1}| / y_n > s$ then
4: flag ← 1;
5: end if
According to the characteristics of the microservice scenario, the increase of interval time will be first multiplicative and then additive, until the flow jitters. In this case, the interval time will be reset to the minimum.

5 | EXPERIMENTS

5.1 | Generating simulated flow

Service flow is a random process and can be expressed as a function of time \( \{x(t), t \in T\} \). The previous research found that the phenomenon of self-similarity is a characteristic of the Internet traffic dominated by network technologies such as the Ethernet and the TCP/IP protocol stack [13]. Statistically the traffic autocorrelation structure is maintained for several time scales and the traffic presents a characteristic of long dependence interval or Long-Range Dependence (LRD) [14]. A self-similar process that captures the LRD-nature is characterised by a single parameter, called Hurst parameter (\( H \)). A process of LRD has \( H \) larger than 0.5, and with high degrees of self-similarity when it is close to 1 [15].

There are multiple models for self-similar traffic generation, for example the Fractional Brownian Motion (FBM), the Fractional Gaussian Noise (FGN) and the On/Off model. Here we use the fractal Gaussian noise to generate a simulated flow for its adjustable \( H \) parameter [16,17].

5.2 | Comparison with static method

We use random midpoint displacement method (RMD) to generate a 1024-length fractal Gaussian noise based simulated flow with Hurst parameter of 0.7. For better illustration, we set the average values to 50, 100 and 150, respectively, and show the flow curve in Figure 2.

The service load factors are set as 0.06, 0.04 and 0.02, respectively, and the total resource \( m \) is set as 10. At this time, the static flow-limiting threshold is the average value of the flow, which is the theoretical optimal solution of the static flow-limiting method. Figure 3 shows the actual passing traffic after applying static and dynamic flow control on the flow scenario in Figure 2.

It can be seen from Figure 3 that the actual passing flow under the dynamic flow control rule is similar to the arriving flow. When the flow peak occurs, the flow limit thresholds of all services can be dynamically adjusted to adapt to current flow scenario and maximize resource utilization. Static the flow control rules the flattened peaks, other services' flow are not saturated, resulting in lower resource utilization.

Table 2 lists the statistics of comparison experiments, where the throughput is the sum of passed QPS, the resource utilization is the product of traffic and load factor divided by the total resource amount and request arrival rate represents the ratio of total requests to actual requests. The experimental results show that when the resources conditions are the same, the performance of the dynamic flow control is about 1.5% higher than that of the static method in terms of main indicators (throughput and resource utilization), and the dynamic rules are closer to the real-time incoming flow (MD increased by about 158%).

5.3 | Influence of self-similarity on dynamic flow control

In order to compare the effect of flow self-similarity on dynamic flow control algorithm, the fractal Gaussian noise is used to generate flow with different \( H \) parameters (Figure 4). The \( H \) parameters are 0.2, 0.4, 0.6 and 0.8, respectively.

There could be random errors with Hurst parameters of flow simulated by the fractal Gaussian noise. We repeat the experiment for 100 times and take the average to reduce the effect of such error on experimental results. Table 3 presents the performance of the dynamic flow control method for different self-similar flows.

It can be seen from Table 3 that the dynamic rules' performance is better with the self-similarity flow. Considering that the Hurst parameter of microservice is usually

![Figure 2](image_url) Simulated service flow

```plaintext
6: end for
7: if flag == 0 then
8: if T < k then
9: \( T' = T \times 2 \);
10: else
11: \( T' = T + 1 \);
12: end if
13: else
14: Record k = T;
15: Reset time interval \( T' = 1 \);
16: end if
17: Return \( T' \)
```
between 0.7 and 0.9, the adaptive dynamic flow control algorithm can achieve performance improvement with micro-service flow scenarios.

5.4 Flow control on different scenarios

In various traffic scenarios, the magnitude of the traffic change will be different. We simulate this by adjusting the variance of the generated flow. Based on the standard FGN simulated flow (variance is 10, Hurst parameter is 0.7), we compared the effects of different flow control strategies when the variance is scaled. As shown in Table 4, the variance of the original flow is scaled from 50% to 130%, and we measured various indicators when static and dynamic flow control methods were applied.

Table 5 shows the improvement in various indicators of the dynamic method over the static method in different flow scenarios. Among them, MD represents the Euclidean distance between the flow control rules and the actual flow; and its value indicates the efficiency of the flow control protection. In all scenarios, the ratio of the MD value of the dynamic method to the static method exceeds 100%. In terms of resource utilization, throughput, arrival rate and other indicators, as the variance of flow decreases, the benefits of dynamic flow control also decrease accordingly. It shows that the dynamic method is more effective for the drastically changing flow scenarios.

5.5 Time interval of the updating flow control rules

From an intuitive point of view, the higher the frequency of updating the flow-limiting rules, the better the fit with the actual flow, and the higher the utilization of resources. Updating flow control rules, especially solving linear programming problems, requires computing resources. Therefore, we need to adjust the update frequency to balance the computational costs with the benefits obtained.

In this part, we designed experiments to compare the impact of updating rules with different frequencies. It can be seen from the Table 6 that even the lowest frequency can greatly improve the resource utilization rate, which further improves as the frequency increases. The actual passed flow corresponding to different update frequencies is shown in Figure 5.

It can be seen that although various indicators have improved with the increase of update frequency, the extent of the improvement has decreased. Therefore, we hope to find an adaptive time interval that can balance the benefits of adjusting the rules and the cost of computation. In order to better show the gap between different methods, we constructed a simulated flow of 90% load, which allows more room for adjustment when the load is not full.

In Table 7, we compared fixed time interval, random time interval and adaptive time interval (Algorithm 2). We control the adaptive time interval by adjusting the value of $s$. Compared with other methods, with the same or less update times, better results are obtained. Since in actual business, the flow scenarios are different, we recommend to re-experiment according to the actual business flow to set an appropriate $s$ value.

6 IMPLEMENTATION

Sentinel is an open source framework with high availability components such as flow control, traffic shaping, circuit breaking and system adaptive protection, guaranteeing reliability and resilience for microservices. It has implemented basic functions of the microservice flow control and provides interfaces to facilitate the secondary development. We implemented an adaptive flow control on Sentinel as a function model, which enables adaptive flow control for multiple services.
6.1 | Resources and rules in Sentinel

Sentinel regards resources as the control unit, and developers only need to pay attention to the definition of resources. Various rules of accessing the resource can be added as soon as the resource is defined. Here we define services as resources to achieve the flow control of the microservice system. Rules can be dynamically changed and take effect in real-time.

The Sentinel provides pre-set rules to decide whether the incoming requests should be controlled. Developers need to manually set rules based on the characteristics of the service flow, such as limiting the QPS or concurrent number of services. Rules can also be dynamically loaded from data sources. However, setting different rules in different time periods requires manual configuration.

6.2 | Adaptive dynamic flow control

The Sentinel provides a number of infrastructures such as the resource run-time statistics and flow control, system monitoring, and establishment of the invocation chain for our algorithm implementation. The monitoring information is updated in real-time. Based on this information, we construct the linear optimization model described above. We built a AutomaticRuleManager, which build and solve model to generate rules based on monitoring information and Algorithm. The two phrase simplex method [18] is used to solve the linear programming model and we implemented the automatic construction of the simplex tableau.

Users only need to use the AutomaticSlot when accessing the Sentinel and set parameters, such as the highest CPU usage and the highest system load for the AutomaticRuleManager, which can automatically perform flow control according to the real-time flow situation and system status, without additional manual configuration. This dynamic flow control is calculated in real time. There is no need to access additional data sources, which simplifies the use process. We also provide a code example [19].
### Table 4: Flow Control on Different Scenarios

| Scenario      | MD  | Throughput | Resource Utilization | Request Arrival Rate |
|---------------|-----|------------|----------------------|----------------------|
| Dynamic-130%  | 173.1 | 303 k | 96.1% | 96.8% |
| Static-130%   | 307.8 | 289 k | 91.9% | 92.3% |
| Dynamic-110%  | 149.0 | 305 k | 97.4% | 97.2% |
| Static-110%   | 217.7 | 300 k | 96.8% | 95.4% |
| Dynamic-90%   | 190.5 | 298 k | 99.0% | 95.6% |
| Static-90%    | 252.1 | 293 k | 97.1% | 94.1% |
| Dynamic-70%   | 42.4  | 299 k | 96.2% | 99.4% |
| Static-70%    | 76.0  | 297 k | 95.5% | 98.6% |
| Dynamic-50%   | 45.2  | 299 k | 98.1% | 99.3% |
| Static-50%    | 64.1  | 297 k | 97.2% | 98.8% |

### Table 5: Improvement of the Dynamic Flow Control

| Flow Variance | MD   | Throughput | Resource Utilization | Request Arrival Rate |
|---------------|------|------------|----------------------|----------------------|
| 130%          | 177.7% | 104.8% | 104.5% | 104.8% |
| 110%          | 140.7% | 101.9% | 100.6% | 101.9% |
| 90%           | 121.8% | 101.6% | 101.4% | 101.6% |
| 70%           | 179.0% | 100.8% | 100.7% | 100.8% |
| 50%           | 141.8% | 100.5% | 100.9% | 100.5% |

### Figure 5: Passed flow with different update time interval

- (a) Static rule
- (b) 256 time interval
- (c) 64 time interval
- (d) two time interval
7 | CONCLUSION

We propose a flow-control algorithm to dynamically calculate the flow-limiting threshold based on the real-time monitoring data. The method in this article has been experimentally verified to have a significant performance improvement over the traditional flow control method and implemented an open source within the Sentinel. The algorithm can reduce manual intervention during the microservice system deployment, improve system resource utilization, and has great application potentials for the current distributed microservice architecture.

In future, we will study the flow control problem under the condition of non-linear resource consumption to improve the robustness of the algorithm.

ACKNOWLEDGEMENT

This work was supported by the National Key Research and Development Program of China under Grant 2018YFB1402803. We are also grateful to Alibaba Summer of Code, Sentinel Community and Zhao Yihao the PMC of Sentinel for their technical support and valuable feedback.

ORCID

Yudong Li  https://orcid.org/0000-0001-6779-8836

REFERENCES

1. Jamshidi, P., et al.: Microservices: the journey so far and challenges ahead. IEEE Softwar. 35(3), 24–35 (2018)
2. Pahl, C., Jamshidi, P.: Microservices: a systematic mapping study. in CLOSER(1), 137–146 (2016)
3. Dragini, N., et al.: Microservices: yesterday, today, and tomorrow. In: Present and Ulterior Software Engineering, pp. 195–216. Springer (2017)
4. Cheraghliou, M.N., Khadem-Zadeh, A., Haghparast, M.: A survey of fault tolerance architecture in cloud computing. J. New. Comput. Appl. 61, 81–92 (2016)
5. Wu, Y., et al.: A model-based fault tolerance mechanism development approach for cloud computing. J. Comp. Res. Develop. 53(1), 138–154 (2016)
6. Netflix: Hystrix: Latency and fault tolerance for distributed systems (2020) https://github.com/Netflix/Hystrix
7. Alibaba: Sentinel: the sentinel of your microservices. https://github.com/alibaba/Sentinel (2020)
8. von Kistowski, J., et al.: A microservice reference application for benchmarking, modeling and resource management research. In: 2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), pp. 223–236. IEEE (2018)
9. de Camargo, A., et al.: An architecture to automate performance tests on microservices. In: Proceedings of the 18th International Conference on Information Integration and Web-based Applications and Services, pp. 422–429 (2016)
10. Turner, J.: New directions in communications (or which way to the information age?). IEEE Commun. Mag. 24(10), 8–15 (1986)
11. Niestegge, G.: The leaky bucket policing method in the ATM (asynchronous transfer mode) network. Int. J. Digital Analog Commun. Syst. 3(2), 187–197 (1990)
12. Ha, S., Rhee, I., Xu, L.: Cubic: a new TCP-friendly high-speed TCP variant. ACM SIGOPS – Oper. Syst. Rev. 42(5), 64–74 (2008)
13. Melo, E.F., de Oliveira, H.d.M.: An overview of self-similar traffic: its implications in the network design (2020). arXiv preprint arXiv: 2005.02888
14. Hosking, J.: Fractional differencing. Biometrika 68 165–176, Mathemat. Rev. (MathSciNet): MR614953 Zentralblatt MATH. 464 (1981)
15. Luo, S., Marin, G.A.: Realistic internet traffic simulation through mixture modeling and a case study. In: Proceedings of the Winter Simulation Conference, pp. 9–pp. IEEE (2005)

16. Paxson, V.: Fast, approximate synthesis of fractional Gaussian noise for generating self-similar network traffic. Comput. Commun. Rev. 27(5), 5–18 (1997)

17. Ryu, B., Lowen, S.: Fractal traffic models for internet simulation. In: Proceedings ISCC 2000. Fifth IEEE Symposium on Computers and Communications, pp. 200–206. IEEE (2000)

18. Sedgewick, R., Wayne, K.: Two phase simplex (2016) https://algs4.cs.princeton.edu/65reductions/

19. Yudong, L.: Automaticqpsdemo (2020) https://github.com/alibaba/Sentinel/pull/1003/files

How to cite this article: Li Y, Zhang Y, Zhou Z, Shen L. Intelligent flow control algorithm for microservice system. Cogn. Comput. Syst. 2021;1–10. https://doi.org/10.1049/ccs2.12013