Overload Control for Scaling WeChat Microservices

(Updated on 18 December 2018)*

Hao Zhou
Tencent Inc.
China
harveyzhou@tencent.com

Ming Chen
Tencent Inc.
China
mingchen@tencent.com

Qian Lin
National University of Singapore
Singapore
linqian@comp.nus.edu.sg

Yong Wang
Tencent Inc.
China
darwinwang@tencent.com

Xiaobin She
Tencent Inc.
China
stevenshe@tencent.com

Sifan Liu
Tencent Inc.
China
stephenliu@tencent.com

Rui Gu
Columbia University
New York, USA
ruigu@cs.columbia.edu

Beng Chin Ooi
National University of Singapore
Singapore
ooibc@comp.nus.edu.sg

Junfeng Yang
Columbia University
New York, USA
junfeng@cs.columbia.edu

ABSTRACT

Effective overload control for large-scale online service system is crucial for protecting the system backend from overload. Conventionally, the design of overload control is ad-hoc for individual service. However, service-specific overload control could be detrimental to the overall system due to intricate service dependencies or flawed implementation of service. Service developers usually have difficulty to accurately estimate the dynamics of actual workload during the development of service. Therefore, it is essential to decouple the overload control from service logic. In this paper, we propose DAGOR, an overload control scheme designed for the account-oriented microservice architecture. DAGOR is service agnostic and system-centric. It manages overload at the microservice granule such that each microservice monitors its load status in real time and triggers load shedding in a collaborative manner among its relevant services when overload is detected. DAGOR has been used in the WeChat backend for five years. Experimental results show that DAGOR can benefit high success rate of service even when the system is experiencing overload, while ensuring fairness in the overload control.

KEYWORDS

overload control, service admission control, microservice architecture, WeChat

*Merged the errata published in https://arxiv.org/abs/1806.04075v2.

SoCC '18, October 11–13, 2018, Carlsbad, CA, USA

ACM ISBN 978-1-4503-6011-1/18/10…$15.00
https://doi.org/10.1145/3267809.3267823

1 INTRODUCTION

Overload control aims to mitigate service irresponsiveness when system is experiencing overload. This is essential for large-scale online applications that needs to enforce 24×7 service availability, despite any unpredictable load surge. Although cloud computing facilitates on-demand provisioning, it still cannot solve the problem of overload—service providers are restricted by the computing resources they can afford from the cloud providers, and therefore cloud providers need overload control for the cloud services they provide.

Traditional overload control for simple service architecture presumes a small number of service components with trivial dependencies. For a stand-alone service, overload control is primarily targeted at the operating system, service runtime and applications [2, 24, 29]. For simple multi-tier services, a gateway at the service entry point monitors the load status of the whole system and rejects client requests when necessary to prevent overloading, i.e., load shedding [5, 7, 23].

However, modern online services become increasingly complex in the architecture and dependency, far beyond what traditional overload control was designed for. Modern online services usually adopt the *service-oriented architecture* (SOA) [12], which divides the conventional monolithic service architecture into sub-services connected via network protocols. *Microservice* architecture, as a specialization of SOA, often comprises hundreds to thousands of sub-services, namely microservices, to support sophisticated applications [9, 21]. Each microservice runs with a set of processes on one or multiple machines, and communicates with other microservices through message passing. By decoupling the implementation and deployment of different microservices, the microservice architecture facilitates independent development and update for each microservice, regardless of the underlying programming language and framework. This yields the flexibility and productivity for cross-team development of complex online applications.

Overload control for large-scale microservice system must cope with the complexity and high dynamics of the system, which could

be very challenging in real practice. First, all the microservices must be monitored. If any microservice is out of the scope of monitoring, potential overload may emerge at that spot and further ripple through the related upstream microservices. As a consequence, system may suffer from cascading overload and eventually get hung, leading to high delay of the affected services. Nevertheless, it is extremely difficult to rely on some designated microservices or machine quorum to monitor load status, since no microservice or machine owns the view of the fast evolving service deployment.

Second, it can be problematic to let microservices handle overload independently, due to the complexity of service dependency. For example, suppose the processing of a client request relies on K microservices, but all the required microservices are currently overloaded and each of them rejects incoming requests independently with a probability p. The expectation of the complete processing of a client request is $(1-p)^K$. If p is close to 1 or K is large, the system throughput tends to vanish under such circumstance. However, system overloading is not mitigated by the shed workload, since partial processing of the failed requests still consumes the computational resources. This causes the system to transit into a non-progressive status, which situation is hard to escape.

Third, overload control needs to adapt to the service changes, workload dynamics and external environments. If each microservice enforces a service-level agreement (SLA) for its upstream services, it would drastically slow down the update progress of this microservice as well as its downstream services, defeating the key advantage of the microservice architecture. Similarly, if the microservices have to exchange tons of messages to manage overload in a cooperative manner, they may not be able to adapt to the load surge, while the overload control messages may get discarded due to system overload and even further deteriorate the system overload.

To address the above challenges, we propose an overload control scheme, called DAGOR, for a large-scale, account-oriented microservice architecture. The overall mechanism of DAGOR works as follows. When a client request arrives at an entry service, it is assigned with a business priority and a user priority such that all its subsequent triggered microservice requests are enforced to be consistently admitted or rejected with respect to the same priorities. Each microservice maintains its own priority thresholds for admitting requests, and monitors its own load status by checking the system-level resource indicator such as the average waiting time of requests in the pending queue. Once overload is detected in a microservice, the microservice adjusts its load shedding thresholds using an adaptive algorithm that attempts to shed half of the load. Meanwhile, the microservice also informs its immediate upstream microservices about the threshold changes so that client requests can be rejected in the early stage of the microservice pipeline.

整体工作机制

DAGOR overload control employs only a small set of thresholds and marginal coordination among microservices. Such lightweight mechanism contributes to the effectiveness and efficiency of overload handling. DAGOR is also service agnostic since it does not require any service-specific information to conduct overload control. For instance, DAGOR has been deployed in the WeChat business system to cater overload control for all microservices, in spite of the diversity of business logic. Moreover, DAGOR is adaptive with respect to service changes, workload dynamics and external environments, making it friendly to the fast evolving microservice system.

While the problem of shedding load inside a network path has been widely studied in literature [8, 10, 15], this paper more focuses on how to build a practical solution of overload control for an operational microservice system. The main contributions of this paper are to (1) present the design of DAGOR, (2) share experiences of operating overload control in the WeChat business system, and (3) demonstrate the capability of DAGOR through experimental evaluation.

The rest of the paper is organized as follows. §2 introduces the overall service architecture of WeChat backend as well as workload dynamics that it usually faces. §3 describes the overload scenarios under WeChat's microservice architecture. §4 presents the design of DAGOR overload control and its adoption in WeChat. We conduct experiments in §5 to evaluate DAGOR, review related work in §6, and finally conclude the paper in §7.

2 BACKGROUND

As a background, we introduce the service architecture of WeChat backend which is supporting more than 3000 mobile services, including instant messaging, social networking, mobile payment and third-party authorization.

2.1 Service Architecture of WeChat

The WeChat backend is constructed based on the microservice architecture, in which common services recursively compose into complex services with a wide range of functionality. The interrelation among different services in WeChat can be modeled as a directed acyclic graph (DAG), where a vertex represents a distinct service and an edge indicates the call path between two services. Specifically, we classify services into two categories: *basic service* and *leap service*. The out-degree (i.e., number of outbound edges) of a basic service in the service DAG is zero, whereas that of a leap service is non-zero. In other words, a leap service can invoke other services, either basic or leap, to trigger a downstream service task, and any task is eventually sunk to a basic service which will not further invoke any other services. Specially, a leap service with in-degree (i.e., number of inbound edges) being zero in the service DAG is referred as an *entry service*. The processing of any service request raised by user always starts with an entry service and ends with a basic service.

Figure 1 demonstrates the microservice architecture of WeChat backend which is hierarchized into three layers. Hundreds of entry services stay at the top layer. All the basic services are placed at the bottom layer. The middle layer contains all the leap services other than the entry ones. Every service task is constructed and processed by going through the microservice architecture in a top-down manner. In particular, all the basic services are shared among all the leap services for invocation, and they are the end services that serve the user-level purposes[1]. Moreover, a leap service in the

[1]For example, the Account service maintains users' login names and passwords, the Profile service maintains users' nicknames and other personal information, the Contact service maintains a list of friends connected to the user, and the Message Inbox service caches users' incoming and outgoing messages.

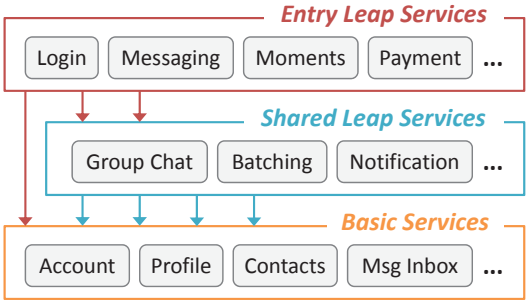


Figure 1: WeChat's microservice architecture.

middle layer is shared by all the entry services as well as other leap services. Most of the WeChat services belong to this layer.

For WeChat, the total amount of requests to the entry services is normally $10^{10} \sim 10^{11}$ on a daily basis. Each entry service request subsequently triggers more requests to the collaborative microservices to actualize the user-intended behavior. As a consequence, the WeChat backend essentially needs to handle hundreds of millions of service requests per second, and system processing data at such scale is obviously challenging.

2.2 Deployment of WeChat Services

WeChat's microservice system accommodates more than 3000 services running on over 20000 machines in the WeChat business system, and these numbers keep increasing as WeChat is becoming immensely popular. The microservice architecture allows different development teams to deploy and update their developed services independently. As WeChat is ever actively evolving, its microservice system has been undergoing fast iteration of service updates. For instance, from March to May in 2018, WeChat's microservice system experienced almost a thousand changes per day on average. According to our experience of maintaining the WeChat backend, any centralized or SLA-based overload control mechanism can hardly afford to support such rapid service changes at large scale.

2.3 Dynamic Workload

Workload handled by the WeChat backend is always varying over time, and the fluctuation pattern differs among diverse situations. Typically, the request amount during peak hours is about 3 times larger than the daily average. In occasional cases, such as during the period of Chinese Lunar New Year, the peak amount of workload can rise up to around 10 times of the daily average. It is challenging to handle such dynamic workload with a wide gap of service request amount. Although over-provisioning physical machines can afford such huge workload fluctuation, the solution is obviously uneconomic. Instead, it is advisable and more practical by carefully designing the overload control mechanism to adaptively tolerate the workload fluctuation at system runtime.

3 OVERLOAD IN WECHAT

System overload in the microservice architecture can result from various causes. The most common ones include load surge, server capacity degradation due to the change of service agreement, network outage, changes of system configuration, software bugs and hardware failures. A typical *overload scenario* involves the overloaded services and the service requests along the associated call path. In this section, we describe three basic forms of service overload that are complete and able to be used for composing other complex forms of service overload. The three basic forms are illustrated in Figure 2, in which the overloaded services are labeled by the attention sign and the associated requests along the call path are denoted by arrows.

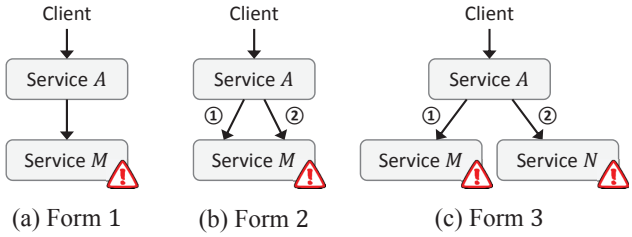


Figure 2: Common overload scenarios.

3.1 Overload Scenarios

In Form 1, as shown in Figure 2.a, overload occurs at service M. In the WeChat business system, service M usually turns out to be a basic service. This is because basic services represent the sink nodes in the service DAG of the microservice architecture, and therefore they are the most active services. In Figure 2.a, the arrows indicate a certain type of request that invokes service M through service A. When service M is overloaded, all the requests sending to service M get affected, resulting in delayed response or even request timeout. Even worse, upstream services (e.g., service A) of the overloaded service are also affected, since they are pending on the responses from the overloaded service. This leads to backward propagation of overload from the overloaded service to its upstream services.

While Form 1 is common in SOA, Form 2 and Form 3 are unique to the large-scale microservice architecture[2]. Form 2, as shown in Figure 2.b, is similar to Form 1 but involves more than one invocation from service A to service M. Such multiple invocation may be required according to the business logic. For example, in an encryption/decryption application, service A may firstly invoke service M to decrypt some data, then manipulate the plain-text data locally, and finally invoke service M again to encrypt the resulting data. We term the corresponding overload scenario as *subsequent overload*, which is formally defined as follows.

Definition 1 (Subsequent Overload). Subsequent overload refers to the overload scenario such that there exist more than one overloaded services or the single overloaded service is invoked multiple times by the associated upstream services.

后续过载的定义：调用多个过载的下游微服务或者调用一个过载下游服务多次。

In the scenario of subsequent overload, a service task instantiated by the upstream service succeeds if and only if all its issued

[2]Form 2 and Form 3 are in fact also common in GFS-like systems [13], where a big file is split into many chunks distributed over different storage servers, and all the chunks need to be retrieved to reconstruct the original file.

requests get successful responses. Otherwise, processing of a service task is considered failed if any of the service requests sent to the overloaded service is not satisfied. Obviously, both Form 2 (in Figure 2.b) and Form 3 (in Figure 2.c) belong to subsequent overload. Subsequent overload in Form 2 is due to the consecutive invocations to the single overloaded service, whereas subsequent overload in Form 3 is caused by the separate invocations to different overloaded services.

subsequent overload 过载但是请求成功率会严重下降

Subsequent overload raises challenges for effective overload control. Intuitively, performing load shedding at random when a service becomes overloaded can sustain the system with a saturated throughput. However, subsequent overload may greatly degrade system throughput out of anticipation. This is due to the service constraint of consecutive success of invoking the overloaded service. For example, in Form 2 shown in Figure 2.b, suppose service A invokes service M twice, and service M is configured with capacity C and performs load shedding at random when it is overloaded. We further suppose the current workload feed rate to service M is $2C$, and service M is only called by service A. As a consequence, service M is expected to reject half of the incoming requests, inferring the success rate of each service M invocation to be 50%. From the perspective of service A, its success rate regarding the invocations to service M is $50\% \times 50\% = 25\%$. In other words, with C service A tasks issued, $2C$ requests are sent to service M and only $0.25C$ service A tasks eventually survive. In contrast, if the workload of service A is $0.5C$, then service M is just saturated without overload and thus the amount of successful service A tasks is $0.5C$. As can be seen, subsequent overload can lead to the success rate of service A become very low if each service A task has to invoke service M many times. The same argument also applies to Form 3 shown in Figure 2.c, with the only difference that the success rate of service A relies on the product of request success rates of service M and service N.

Among the above three basic forms of service overload, Form 1 and Form 2 dominate the overload cases in the WeChat business system, whereas Form 3 appears to be relatively rare. Towards effective overload control, we emphasize that subsequent overload must be properly handled to sustain the system throughput when the runtime workload is heavy. Otherwise, simply adopting random load shedding could lead to extremely low (e.g., close to zero) request success rate at the client side when the requesting services are overloaded. Such "service hang" has been observed in our previous experience of maintaining the WeChat backend, and it motivated us to investigate the design of overload control that fits WeChat's microservice architecture.

3.2 Challenges of Overload Control at Scale

Comparing with traditional overload control for the web-oriented three-tier architecture and SOA, overload control for WeChat's microservice architecture has two unique challenges.

First, there is no single entry point for service requests sent to the WeChat backend. This invalidates the conventional approach of centralized load monitoring at a global entry point (e.g., the gateway). Moreover, a request may invoke many services through a complex call path. Even for the same type of requests, the corresponding call paths could be quite different, depending on the request-specific data and service parameters. As a consequence, when a particular service becomes overloaded, it is impossible to precisely determine what kind of requests should be dropped in order to mitigate the overload situation.

Second, even with decentralized overload control in a distributed environment, excessive request aborts could not only waste the computational resources but also affect user experience due to the high latency of service response. Especially, the situation becomes severe when subsequent overload happens. This calls for some kind of coordination to manage load shedding properly, regarding the request type, priority, call path and service properties. However, given the service DAG of the microservice architecture being extremely complex and continuously evolving on the fly, the maintenance cost as well as system overhead for effective coordination with respect to the overloaded services are considered too expensive.

4 DAGOR OVERLOAD CONTROL

The overload control scheme designed for WeChat's microservice architecture is called DAGOR. Its design aims to meet the following requirements.

- **Service Agnostic**. DAGOR needs to be applicable to all kinds of services in WeChat's microservice architecture, including internal and external third-party services. To this end, DAGOR should not rely on any service-specific information to perform overload control. Such design consideration of being service agnostic has two advantages. First, the overload control mechanism can be highly scalable to support large amount of services in the system, and meanwhile adapt to the dynamics of service deployment. This is essential for the ever evolving WeChat business, as diverse services deployed in the WeChat business system are frequently updated, e.g., new services going online, upgrading existing services and adjusting service configurations. Second, the semantics of overload control can be decoupled from the business logic of services. As a consequence, improper configuration of service does not affect the effectiveness of overload control. Conversely, overload detection can help find the implicit flaw of service configuration which causes service overload at runtime. This not only benefits service development and debugging/tuning, but also improves system availability as well as robustness.
- **Independent but Collaborative**. In the microservice architecture, a service is usually deployed over a set of physical machines in order to achieve scalability and fault tolerance. In practice, workload distribution over the machines is hardly balanced, and the load status of each machine may fluctuate dramatically and frequently, with few common patterns shared among different machines. Therefore, overload control should run on the granule of individual machine rather than at the global scale. On the other hand, collaborative inter-machine overload control is also considered necessary for handling subsequent overload. The collaboration between different machines needs to be service-oriented so that the success rate of the upstream service can match the response rate of the overloaded service, in spite of the occurrence of subsequent overload.

- **Efficient and Fair**. DAGOR should be able to sustain the best-effort success rate of service when load shedding becomes inevitable due to overload. This infers that the computational resources (i.e., CPU and I/O) wasted on the failed service tasks are minimized. Note that those immediately aborted tasks cost little computation and consequentially yield the resource for other useful processing; in contrast, tasks that are partially processed but eventually get aborted waste the computation spent on them. Therefore, the efficiency of overload control refers to how the mechanism can minimize the waste of computational resources spent on the partial processing of service tasks. Moreover, when a service gets overloaded, its upstream services should be able to sustain roughly the same saturated throughput despite how many invocations an upstream service task makes to the overloaded service. This reflects the fairness of the overload control.

4.1 Overload Detection

DAGOR adopts decentralized overload control at the server granule and thus each server monitors its load status to detect potential overload in time. For load monitoring towards overload detection in a single server, a wide range of performance metrics have been studied in literature, including throughput, latency, CPU utilization, packet rate, number of pending requests, request processing time, etc. [20, 25, 31]. DAGOR by design uses the average waiting time of requests in the pending queue (or *queuing time* for short) to to profile the load status of a server. The queuing time of a request is measured by the time difference between the request arrival and its processing being started at the server.

The rationale of monitoring the queuing time for overload detection is not so obvious at the first sight. One immediate question is: Why not consider using the response time[3] instead? We argue that the queuing time can be more accurate to reflect the load status than the response time. Comparing with the queuing time, the response time additionally counts the request processing time. In particular, the time for processing a basic service request is purely determined by the local processing, whereas the processing time for a leap service request further involves the cost of processing the downstream service requests. This results in the measurement of response time being recursive along the service call path, making the metric failed to individually reflect the load status of a service or a server. In contrast, the queuing time is only affected by the capability of local processing of a server. When a server becomes overloaded due to resource exhaustion, the queuing time rises proportional to the excess workload. On the other hand, the queuing time would stay at a low level if the server has abundant resources to consume the incoming requests. Even if the downstream server may respond slowly, queuing time of the upstream server is not affected as long as it has sufficient resource to accommodate the pending service tasks, though its response time does rise according to the slow response of the downstream services. In other words, the response time of a server increases whenever the response time of its downstream servers increases, even though the server itself is not overloaded. This provides a strong evidence that the queuing time can reflect the actual load status of a server, whereas the response time is prone to false positives of overload.

Another question is: Why is not CPU utilization used as an overload indicator? It is true that high CPU utilization in a server indicates that the server is handling high load. However, high load does not necessarily infer overload. As long as the server can handle requests in a timely manner (e.g., as reflected by the low queuing time), it is not considered to be overloaded, even if its CPU utilization stays high.

Load monitoring of DAGOR is window-based, and the window constraint is compounded of a fixed time interval and a maximum number of requests within the time interval. In the WeChat business system, each server refreshes its monitoring status of the average request queuing time every second or every 2000 requests, whenever either criteria is met. Such compounded constraint ensures that the monitoring can immediately catch up with the load changes in spite of the workload dynamics. For overload detection, given the default timeout of each service task being 500 ms in WeChat, the threshold of the average request queuing time to indicate server overload is set to 20 ms. Such empirical configurations have been applied in the WeChat business system for more than five years with its effectiveness proven by the system robustness with respect to WeChat business activities.

4.2 Service Admission Control

Once overload is detected, the corresponding overload control is based on service admission control. DAGOR contains a bundle of service admission control strategies. We first introduce two types of the priority-based admission control adopted in DAGOR overload control, and then extend the technique to further support adaptive and collaborative admission control.

4.2.1 Business-oriented Admission Control. WeChat services are internally prioritized based on their business significance and impact on user experience, so are the corresponding service requests. For example, the Login request is of the highest priority, because user cannot interact with other services until he/she completes a successful login. Another example is that the WeChat Pay request has higher priority than the Instant Messaging request. This is because users tend to be sensitive to their money-related interactions such as mobile payment, while they are usually able to accept a certain degree of delay or inconsistency in the messaging service. The operation log of WeChat shows that when WeChat Pay and Instant Messaging experience a similar period of service unavailability, user's complaint against the WeChat Pay service is 100 times more than that against the Instant Messaging service. Similar situation also applies to Instant Messaging versus Moments, as user expects more timely delivery of content with Instant Messaging than with Moments.

Business-oriented admission control in DAGOR is to assign a business priority to each user request and enforce all its subsequent requests inherit the same business priority. When a service becomes overloaded, its load shedding routine will give priority to discarding low-priority requests, yielding resources for high-priority requests. The business priority of a user request as well as its subsequent requests along the call path is determined by the

[3]Response time is defined as the time difference between the request arriving at the server and the corresponding response sent out from the server.

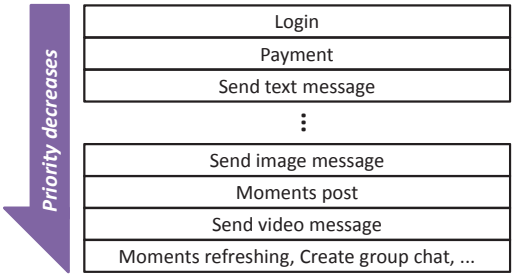


Figure 3: Hash table storing the business priorities of actions to perform in the WeChat entry services.

type of action to perform in the entry service. As there exist hundreds of entry services in WeChat's microservice architecture, the number of different actions to perform in the entry services is of hundreds. The business priorities are predefined and stored in a hash table, which is replicated to all the WeChat backend servers that host the entry services. An item in the hash table records the mapping from an action ID (representing a distinct type of action) to a priority value. Figure 3 illustrates the logical structure of the action-priority hash table. Note that the hash table does not contain all types of actions. By default, action types that are missing in the hash table correspond to the lowest priority. Only those intentionally prioritized action types are recorded in the hash table, with smaller priority value indicating higher priority of the action. This results in the hash table containing only a few tens of entries. Since the set of prioritized actions to perform in the entry service are empirically stable, the hash table remains compact with rare changes despite the rapid evolution of WeChat business.

Whenever a service request triggers a subsequent request to the downstream service, the business priority value is copied to the downstream request. By recursion, service requests belonging to the same call path share an identical business priority. This is based on the presumption that the success of any service request relies on the conjunctive success of all its subsequent requests to the downstream services. As the business priority is independent to the business logic of any service, DAGOR's service admission control based on the business priority is service agnostic. Moreover, the above business-oriented admission control is easy to maintain, especially for the complex and highly dynamic microservice architecture such as the WeChat backend. On the one hand, the assignment of business priority is done in the entry services by referring to the action-priority hash table, which is seldom changed over time[4]. This makes the strategy of business priority assignment relatively stable. On the other hand, the dynamics of WeChat's microservice architecture are generally reflected in the changes of basic services and leap services other than the entry services. Since the business priorities of requests to these frequently changing services are inherited from the upstream service requests, developers of these services can simply apply the functionality of business-oriented admission control as a black box without concerning the setting of business priority[5].

4.2.2 User-oriented Admission Control. The aforementioned strategy of business-oriented admission control constrains the decision of dropping a request to be determined by the business priority of the request. In other words, for load shedding upon service overload, the business-oriented admission control presumes requests with the same business priority are either all discarded or all consumed by the service. However, partially discarding requests with respect to the same business priority in an overloaded service is sometimes inevitable. Such inevitability emerges when the admission level of business priority of the overloaded service is fluctuating around its "ideal optimality". To elaborate, let us consider the following scenario where load shedding in an overloaded service is solely based on the business-oriented admission control. Suppose the current admission level of business priority is τ but the service is still overloaded. Then the admission level is adjusted to $\tau - 1$, i.e., all requests with business priority value greater than or equal to τ are discarded by the service. However, system soon detects that the service is underloaded with such admission level. As a consequence, the admission level is set back to τ, and then the service quickly becomes overloaded again. The above scenario continues to repeat. As a result, the related requests with business priority equal to τ are in fact partially discarded by the service in the above scenario.

Partially discarding requests with the same business priority could bring on issue caused by subsequent overload, because these requests are actually discarded at random under such situation. To tackle this issue, we propose the user-oriented admission control as a compensation for the business-oriented admission control. User-oriented admission control in DAGOR is based on the user priority. The user priority is dynamically generated by the entry service through a hash function that takes the user ID as an argument. Each entry service changes its hash function every hour. As a consequence, requests from the same user are likely to be assigned to the same user priority within one hour, but different user priorities across hours. The rationality for the above strategy of user priority generation is twofold. On the one hand, the one-hour period of hash function alternation allows user to obtain a relatively consistent quality of service for a long period of time. On the other hand, the alternation of hash function takes into account the fairness among users, as high priorities are granted to different users over hours of the day. Like the business priority, the user priority is also bound to all the service requests belonging to the same call path.

The strategy of user-oriented admission control cooperates with the business-oriented admission control. For requests with business priority equal to the admission level of business priority of the overloaded service, the corresponding load shedding operation gives priority to the ones with high user priority. By doing so, once a request from service A to the overloaded service M gets

[4]The action-priority hash table may be modified on the fly for performance tuning or ad-hoc service support. But this happens very rarely in the WeChat business system, e.g., once or twice per year.

[5]We used to additionally provide APIs for service developer to adjust the business priority of request specifically for the service. However, the solution turned out to be not only extremely difficult to manage among different development teams, but also error-prone with respect to the overload control. Consequently, we deprecated the APIs for setting service-specific business priority in the WeChat business system.

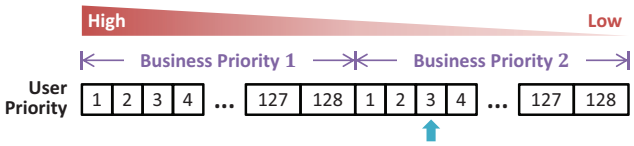


Figure 4: The compound admission level.

a successful response, the subsequent request from service A to service M is very likely to also get a successful response. This resolves the issue caused by subsequent overload of Form 2 as shown in Figure 2.b. Moreover, subsequent overload of Form 3 as shown in Figure 2.c can also be properly handled in a similar way. Suppose the upstream service A invokes two overloaded dependent services, i.e., service M and service N, in order. If the admission levels regarding the business and user priorities of service M are more restricted than that of service N, then subsequent overload can be eliminated in Form 3. This is because a request being admitted by service M implies the admission of the subsequent request to service N, due to the relaxed admission levels. Such condition of admission levels between the dependent services in Form 3 usually holds in the WeChat business system, as the preceding service is prone to more severe overload.

Session-oriented Admission Control. Other than the user-oriented admission control, we have ever proposed the session-oriented admission control to address the same issue caused by solely applying the business-oriented admission control. The session-oriented admission control is based on the *session priority*, whose generation as well as functionality are similar to that of the user priority as described before. The only difference is that the hash function of generating the session priority alternatively takes the session ID as an argument. A session ID is assigned to a user upon his/her successful login, which indicates the start of a user session. A session generally ends with a confirmed logout performed by the user. When the same user performs another login after his/her prior logout, another session is created with a different session ID and thus a new session priority is generated accordingly, even though the hash function remains unchanged. In terms of overload control in DAGOR, the session-oriented admission control is as effective as the user-oriented admission control. But our operational experience with the WeChat business system shows that the session-oriented admission control tends to degrade user experience. This is due to the natural user behavior where WeChat users often prefer to logout and immediately login again whenever they encounter service unavailability in the app. The same phenomenon also frequently arises in other mobile apps. Through the logout and immediate login, user obtains a refreshed session ID. As a consequence, the session-oriented admission control assigns the user a new session priority, which could be high enough to grant his/her service requests in the overloaded service backend. Gradually, the user may figure out the "trick" that enables him/her to escape from service unavailability via re-login. When a trick is repeatedly validated to be effective, it tends to become a user habit. However, such trick does not help mitigate the actual service overload occurring at the system backend. Moreover, it would introduce extra user requests due to the misleading logout and login, further deteriorating the overload situation and thus eventually affecting the user experience of the majority of users. In contrast, user's instant re-login does not affect his/her user priority in the user-oriented admission control. Hence, we prefer the user-oriented admission control over the session-oriented one in DAGOR overload control.

4.2.3 Adaptive Admission Control. Load status in the microservice system is always dynamically changing. A service becomes sensitive to the change of its load status when it is overloaded, since the corresponding load shedding strategy is dependent on the volume of the excess workload. Therefore, the priority-based admission levels should be able to adapt to the load status towards effective load shedding with minimized impact on the quality of the overall service. When the overload situation is severe, the admission levels should be restricted to reject more incoming requests; on the contrary, the admission levels should be relaxed when the overload situation becomes mild. In the complex microservice architecture, the adjustment of admission levels of the overloaded services needs to be automatic and adaptive. This calls for the necessity of adaptive admission control in the overload control of DAGOR.

DAGOR adjusts the admission levels of the overloaded services with respect to their individual load status. As illustrated in Figure 4, DAGOR uses the *compound admission level* which is composed of the business and user priorities. Each admission level of business priority is attached with 128 admission levels of user priority. Let $\mathcal{B}$ and $\mathcal{U}$ denote the business priority and the user priority respectively, and the compound admission level is denoted by $(\mathcal{B}, \mathcal{U})$. A cursor, denoted by an arrow in Figure 4, indicates the current admission level[6] to be $(2, 3)$, which is interpreted as all the requests with $\mathcal{B} > 2$ and the requests with $\mathcal{B} = 2$ but $\mathcal{U} > 3$ will be shed by the overloaded service. Moving the cursor leftwards represents raising the admission level, i.e., restricting the business and user priorities.

As mentioned in §4.2.1, DAGOR maintains tens of distinct admission levels of business priority. With each admission level of business priority attached with 128 admission levels of user priority, the resulting amount of the compound admission levels is tens of thousands. Adjustment of the compound admission level is at the granule of user priority. To search for the appropriate admission level according to the load status, a naive approach is by trying each admission level one by one starting from the lowest, i.e., moving the cursor from the far right to the left in Figure 4. Note that for each setting of the admission level, server has to take a while to validate its effectiveness, since the load status is aggregated within a certain time interval. As the incoming requests are unlikely to be distributed evenly over the admission levels, such naive approach tends to be awkward to find the right setting. This is because every adjustment of admission level in the naive approach, i.e., moving the cursor leftwards by one user priority, exerts marginal impact on the overload situation but takes time to validate its sufficiency. Therefore, by scanning the admission levels in the naive way, the adjustment of admission level can hardly meet the real-time requirement for the adaptive admission control. An immediate improvement based on the above naive approach is to perform binary

[6]If not specified, we refer the admission level to the compound admission level in the rest of the paper

Algorithm 1: Adaptive admission control.

```
Global: Range of admission levels of business priority B_L ... B_H
Global: Range of admission levels of user priority U_L ... U_H
Global: Incoming request counters C[B_L ... B_H][U_L ... U_H]
Global: Incoming request counter N
Global: Admitted request counter N_adm
Global: Compound admission level (B*, U*)

Procedure ResetHistogram():          /* at the beginning of period */
    N, N_adm ← 0
    foreach c ∈ C do c ← 0

Input: Service request r
Procedure UpdateHistogram(r):
    N ← N + 1
    C[r.B][r.U] ← C[r.B][r.U] + 1
    if r.B < B* or ( r.B = B* and r.U ≤ U* ) then
        N_adm ← N_adm + 1

Input: Boolean flag f_ol indicating overload
Procedure UpdateAdmitLevel(f_ol):    /* at the end of period */
    N_prefix ← N_adm
    if f_ol = true then
        N_exp ← (1 − α) · N_adm
        while N_prefix > N_exp and (B*, U*) > (B_L, U_L) do
            U* ← U* − 1
            if U* < U_L then
                B* ← B* − 1
                U* ← U_H
            N_prefix ← N_prefix − C[B*][U*]
    else
        N_exp ← N_adm + β · N
        while N_prefix < N_exp and (B*, U*) < (B_H, U_H) do
            U* ← U* + 1
            if U* > U_H then
                B* ← B* + 1
                U* ← U_L
            N_prefix ← N_prefix + C[B*][U*]
```

search instead of linear scan. This makes the search complexity reduces from $O(n)$ to $O(\log n)$ where n represents the number of compound admission levels in total. Regarding the actual amount of admission levels in WeChat is at the scale of 10^4, the $O(\log n)$ search involves about a dozen trials of the adjustment of admission level, which is still considered away from efficiency.

Towards adaptive admission control with efficiency, DAGOR exploits a histogram of requests to quickly figure out the appropriate setting of admission level with respect to the load status. The histogram helps to reveal the approximate distribution of requests over the admission priorities. In particular, each server maintains an array of counters, each of which corresponds to a compound admission level indexed by $(\mathcal{B}, \mathcal{U})$. Each counter counts the number of the incoming requests associated with the corresponding business and user priorities. DAGOR periodically adjusts the admission level of load shedding as well as resets the counters, and the period is consistent to the window size for overload detection as described in §4.1, e.g., every second or every 2000 requests in the WeChat business system. For each period, if an overload is detected, the server sets the expected amount of admitting requests in the next period to be α (in percentage) less than that in the current period; otherwise, the expectation of request amount in the subsequent period is increased by β (in percentage) of the incoming requests in the current period. Empirically, we set $\alpha = 5\%$ and $\beta = 1\%$ in the WeChat business system. Given the expected amount of admitting requests, the admission level is calculated with respect to the maximum prefix sum in the histogram near that amount, and is adjusted based on its current state and whether or not an overload is detected. Let $\mathcal{B}^*$ and $\mathcal{U}^*$ be the optimal settings of the admission levels of business priority and user priority respectively. The optimal setting of the compound admission level $(\mathcal{B}^*, \mathcal{U}^*)$ is determined by the constraint such that the sum of counters with $(\mathcal{B}, \mathcal{U}) \le (\mathcal{B}^*, \mathcal{U}^*)$[7] is just below (resp. exceeding) the expected amount when overload is indicated (resp. suppressed). Algorithm 1 summaries the procedures of adaptive admission control in DAGOR. Note that even if the system is not overloaded at some point, some requests may still be discarded due to the current settings of admission levels, especially when the system is undergoing recovery from a previous overload.

Obviously, the above approach only involves a single trial of validation per adjustment of the admission level. Hence, it is much more efficient than the aforementioned naive approaches, and can satisfy the real-time requirement for the adaptive admission control.

4.2.4 Collaborative Admission Control. DAGOR enforces the overloaded server to perform load shedding based on the priority-based admission control. Regarding message passing, a request that is destined to be shed by the overloaded downstream server still has to be sent from the upstream server, and the downstream server subsequently sends the corresponding response back to the upstream server to inform the rejection of the request. Such round-trip of message passing for unsuccessful request processing not only wastes the network bandwidth but also consumes the tense resource of the overloaded server, e.g., for serializing/deserializing network messages. To save network bandwidth and reduce the burden on the overloaded server to handle excessive requests, it is advised to reject the requests that are destined to be shed early at the upstream server. To this end, DAGOR enables collaborative admission control between the overloaded server and its upstream servers. In particular, a server piggybacks its current admission level $(\mathcal{B}, \mathcal{U})$ to each response message that it sends to the upstream server. When the upstream server receives the response, it learns the latest admission level of the downstream server. By doing so, whenever the upstream server intends to send request to the downstream server, it performs a local admission control on the request according to the stored admission level of the downstream server. As a consequence, requests destined to be rejected by the downstream server tends to be shed early at the upstream server, and requests actually sent out from the upstream server tends to be admitted by the downstream server. Therefore, while the strategy of admission control in a server is independently determined

[7] For two admission levels $(\mathcal{B}_1, \mathcal{U}_1)$ and $(\mathcal{B}_2, \mathcal{U}_2)$, we have $(\mathcal{B}_1, \mathcal{U}_1) < (\mathcal{B}_2, \mathcal{U}_2)$ if $\mathcal{B}_1 < \mathcal{B}_2$, or $\mathcal{B}_1 = \mathcal{B}_2$ but $\mathcal{U}_1 < \mathcal{U}_2$.

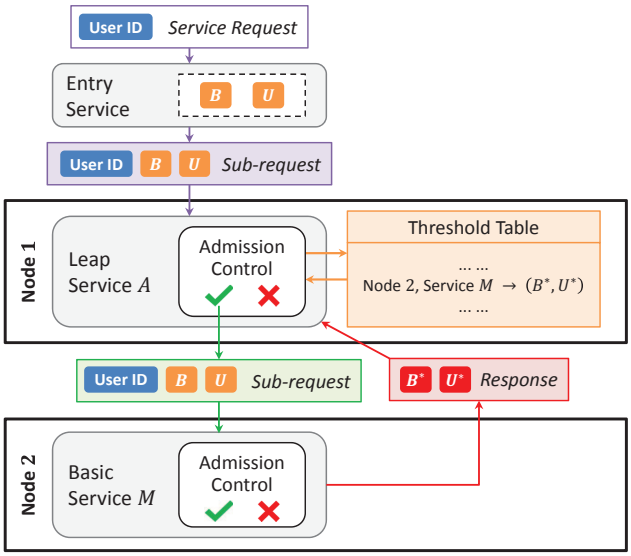


Figure 5: Workflow of DAGOR overload control.

by the server itself, the actual load shedding is performed by its related upstream servers. Such collaboration between the upstream and downstream servers greatly benefits the improved efficiency of overload control in the microservice architecture.

4.3 Workflow of Overload Control

Based on the aforementioned strategies of service admission control, we now depict the overall workflow of DAGOR overload control as illustrated in Figure 5.

(1) When a user request arrives at the microservice system, it is routed to the related entry service. The entry service assigns the business and user priorities to the request, and all the subsequent requests to the downstream services inherit the same priorities which are encapsulated into the request header.

(2) Each service invokes one or more downstream services according to the business logic. Service requests and responses are delivered through message passing.

(3) When a service receives a request, it performs the priority-based admission control based on its current admission level. The service periodically adjusts its admission level with respect to the load status. When a service intends to send a subsequent request to a downstream service, it performs the local admission control based on the stored admission level of the downstream service. The upstream service only sends out the requests that are admitted by the local admission control.

(4) When a downstream service sends a response to an upstream service, it piggybacks its current admission level to the response message.

(5) When the upstream service received the response, it extracts the information of admission level from the message and updates the corresponding local record for the downstream service accordingly.

DAGOR satisfies all the requirements of overload control in the microservice architecture as we proposed at the beginning of this section. First, DAGOR is *service agnostic*, since the strategies of service admission control are based on the business and user priorities that are orthogonal to business logic. This makes DAGOR generally applicable to microservice systems. Second, service admission control of DAGOR is *independent but collaborative*, as the admission levels are determined by the individual services and the admission control is collaboratively performed by the related upstream services. This makes DAGOR highly scalable to be adopted for the large-scale, timely evolving microservice architecture. Third, DAGOR overload control is *efficient and fair*. It can effectively eliminates the performance degradation due to subsequent overload, since all the service requests belonging to the same call path share the identical business and user priorities. This makes the upstream service able to sustain its saturated throughput in spite of the overload situation of the downstream service.

5 EVALUATION

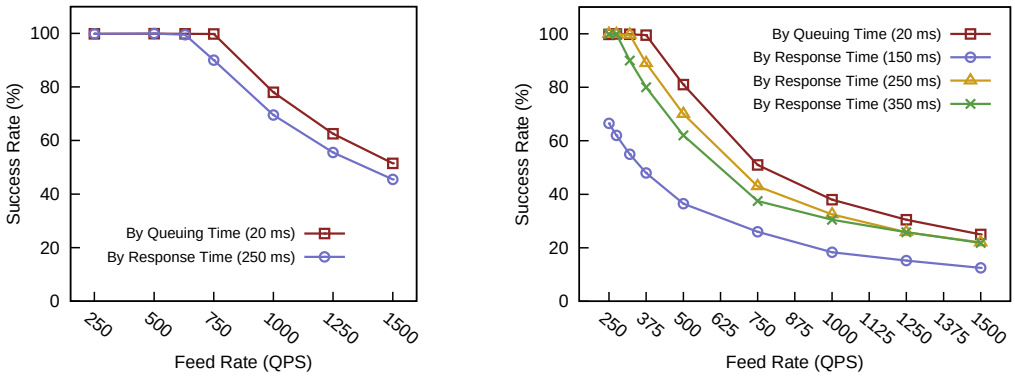
DAGOR has been fully implemented and deployed in the WeChat business system for more than five years. It has greatly enhanced the robustness of the WeChat service backend and helped WeChat survive in various situations of high load operation, including those in daily peak hours as well as the period of special events such as the eve of Chinese Lunar New Year. In this section, we conduct an experimental study to evaluate the design of DAGOR and compare its effectiveness with state-of-the-art load management techniques.

5.1 Experimental Setup

All experiments run on an in-house cluster, where each node is equipped with an Intel Xeon E5-2698 @ 2.3 GHz CPU and 64 GB DDR3 memory. All nodes in the cluster are connected via 10 Gigabit Ethernet.

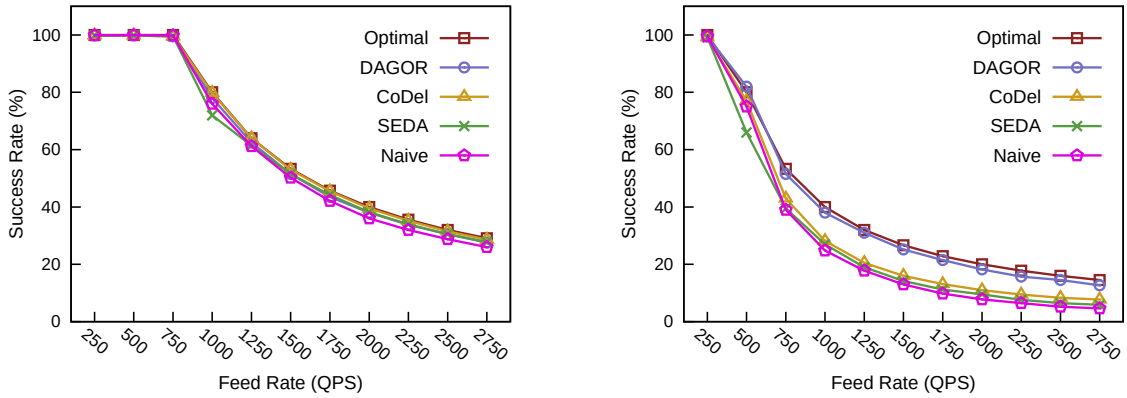
Workloads. To evaluate DAGOR independently, we implement a stress test framework that simulates the encryption service used in the WeChat business system. Specifically, an encryption service, denoted as M, is exclusively deployed over 3 servers and prone to be overloaded with the saturated throughput being around 750 queries per second (QPS). A simple messaging service, denoted as A, is deployed over another 3 servers to process the predefined service tasks by invoking service M as many times as requested by the workload. Workload is synthetically generated by 20 application servers, which are responsible for generating service tasks and never overloaded in the experiments. Each service task is programmed to invoke service M one or multiple times through service A, and the success of the task is determined by the conjunctive success of those invocations[8]. Let $\mathcal{M}^x$ denote the workload consisting of tasks with x-invocation to service M. Four types of workload, namely $\mathcal{M}^1$, $\mathcal{M}^2$, $\mathcal{M}^3$ and $\mathcal{M}^4$, are used in the experiments. Regarding the overload scenario, $\mathcal{M}^1$ corresponds to simple overload, while $\mathcal{M}^2$, $\mathcal{M}^3$ and $\mathcal{M}^4$ correspond to subsequent overload.

[8]In case of rejection, the same request of invocation will be resent up to three times.



(a) $\mathcal{M}^1$ (b) $\mathcal{M}^2$

Figure 6: Overload detection by different indicators of load profiling: queuing time vs. response time.



(a) $\mathcal{M}^1$ (b) $\mathcal{M}^2$

Figure 7: Overload control with increasing workload.

5.2 Overload Detection

We first evaluate DAGOR's overload detection, which serves as the entry point of overload control. In particular, we experimentally verify DAGOR's design choice of adopting the average request queuing time rather than response time as the indicator of load status for overload detection, as discussed in §4.1. To this end, we additionally implement a DAGOR variant whose overload detection refers to the average response time of requests over the monitoring window, i.e., every second or every 2000 requests whenever either is met. Let DAGOR$_q$ (resp. DAGOR$_r$) be the DAGOR implementations with overload detection based on the request queuing time (resp. response time). We conduct experiments by running workloads of $\mathcal{M}^1$ and $\mathcal{M}^2$ individually, varying the feed rate from 250 QPS to 1500 QPS. Figure 6 shows the comparison between DAGOR$_q$ and DAGOR$_r$. For simple overload as in Figure 6.a, we set the thresholds of average queuing time and response time to 20 ms and 250 ms respectively. As can be seen from the results, DAGOR$_r$ starts load shedding when the feed rate reaches 630 QPS, whereas DAGOR$_q$ can postpone the load shedding until 750 QPS of the input. This implies the load profiling based on the response time is prone to false positives of overload. Figure 6.b further confirms this fact with subsequent overload involved, i.e., by running the $\mathcal{M}^2$ workload. In addition to the settings as in Figure 6.a, we also measure the curves of success rate for DAGOR$_r$ with the threshold of response time set to 150 ms and 350 ms in Figure 6.b. The results show that DAGOR$_r$ exhibits best performance when the threshold of response time is set to 250 ms. However, the optimal configuration of DAGOR$_r$ is difficult to tune in practice, since the request response time contains the request processing time which is service-specific. In contrast, apart from the superior performance of DAGOR$_q$, its configuration is easy to be fine-tuned because the request queuing time is irrelevant to any service logic.

5.3 Service Admission Control

Next, we evaluate DAGOR's service admission control for load management. As described in §4.2, the basis of DAGOR's service admission control is based on priority, which is further devised to be business-oriented and user-oriented. The business-oriented admission control is commonly adopted in state-of-the-art load management techniques [25, 32]. DAGOR is novel in its additional

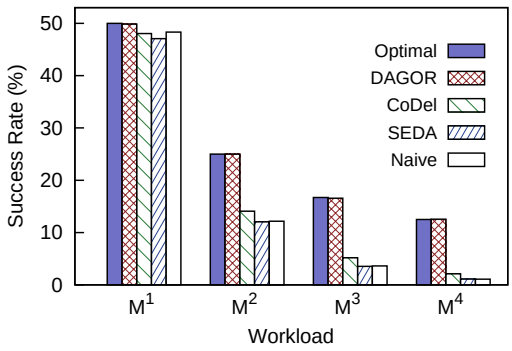


Figure 8: Overload control with different types of workload.

use of user-oriented priority for fine-grained load management towards improved end-to-end service quality. Moreover, DAGOR overload control can adaptively adjust service admission levels in a real-time manner through adaptive admission control, and optimize the load shedding runtime with the adoption of collaborative admission control. These distinct strategies distinguishes DAGOR from existing techniques of load management by mechanism. Hence, in order to verify the effectiveness of DAGOR's innovation in terms of service admission control, we compare DAGOR with state-of-the-art techniques, i.e., CoDel [25] and SEDA [32], through experiments with business priority being fixed for all the generated query requests. In addition, we also employ a naive approach of service admission control such that the overloaded service M performs load shedding at random. Such naive approach serves as the baseline in the experiments.

CoDel
SEDA

Figure 7 demonstrates the comparison results by running workloads of $\mathcal{M}^1$ and $\mathcal{M}^2$ respectively. Each figure compares the success rate of the upstream service requests with the adoption of different service admission control techniques. In addition, we also depict the theoretically optimal success rate of the upstream service when the corresponding downstream services are overloaded. The optimal success rate is calculated by f_{sat}/f, where f_{sat} is the maximum feed rate that makes the downstream service just saturated, and f refers to the actual feed rate when the downstream service is overloaded. As can be seen from Figure 7.a, all the overload control techniques perform roughly the same for simple overload (i.e., $\mathcal{M}^1$). However, when subsequent overload is involved as shown in Figure 7.b, DAGOR exhibits around 50% higher success rate than CoDel and SEDA in the workload of $\mathcal{M}^2$. Figure 8, measured by fixing the feed rate to 1500 QPS, further shows the greater advantage of DAGOR with the increment of subsequent overload in the workloads of $\mathcal{M}^3$ and $\mathcal{M}^4$. Moreover, the request success rate contributed by DAGOR is close to the optimal in all the above results. Such superior of DAGOR is due to its effective suppression of subsequent overload through the priority-based admission control, especially the user-oriented strategy.

5.4 Fairness

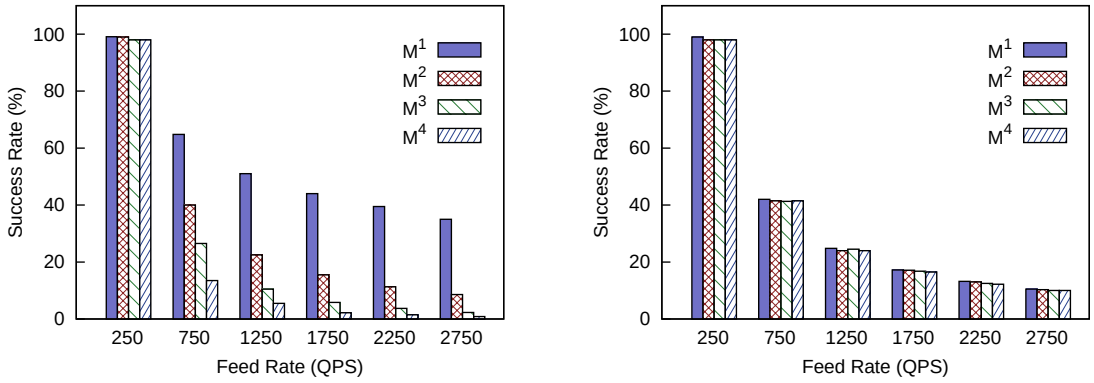
Finally, we evaluate the fairness of overload control with respect to different overload situations. The *fairness* refers to whether the overload control mechanism biased towards one or more specific overload forms. To that end, we run mixed workload comprising $\mathcal{M}^1$, $\mathcal{M}^2$, $\mathcal{M}^3$ and $\mathcal{M}^4$ requests with uniform distribution. The requests are issued concurrently by the clients with the feed rate varying from 250 to 2750 QPS. The business priority and user priority of each request are chosen at random in a fixed range. We compare the fairness of request success rates between DAGOR and CoDel, and the results shown in Figure 9. As can be seen, CoDel favors simple overload (i.e., $\mathcal{M}^1$) over subsequent overload (e.g., $\mathcal{M}^2$, $\mathcal{M}^3$ and $\mathcal{M}^4$). Such bias of CoDel renders its overload control dependent on the service workflow, where the more complex logic tends more likely to fail when the system is experiencing overload. In contrast, DAGOR manifests roughly the same success rate for different types of requests. This is because the priority-based admission control of DAGOR greatly reduces the occurrence of subsequent overload, in spite of the number of upstream invocations to the overloaded downstream services.

6 RELATED WORK

Plenty of existing research of overload control has been devoted to real-time databases [3, 14], stream processing systems [4, 19, 27, 28, 33] and sensor networks [30]. For networked services, techniques of overload control have been mainly proposed in the context of web services [11]. However, most of these techniques are designed for the traditional monolithic service architecture and they do not apply to modern large-scale online service systems that are often built based on the SOA. To the best of our knowledge, DAGOR is the first to specifically address the issue of overload control for large-scale microservice architectures. As the microservice architecture is generally deemed to belong to the family of SOA, we closely review the emerging techniques of overload control for SOA, which are built based on either admission control [5–7, 26, 31] or resource scheduling [1, 2, 18, 22, 24].

Controlling network flow by capturing complex interactions of modern cloud systems can mitigate the impact of system overload. Varys [8] employs the coflow and task abstractions to schedule network flows towards reduction of task completion time. But it relies on the prior knowledge of flow sizes and meanwhile assumes the availability of centralized flow control. This renders the method only applicable to limited size of cloud computing infrastructure. Baraat [10] alternatively adopts a FIFO-like scheduling policy to get rid of centralized control, but at the expense of performance. Comparing with these coflow-based control, DAGOR is not only service agnostic but also independent of network flow characteristics. This makes DAGOR a non-invasive overload control suitable for microservice architecture.

Cherkasova et al. [7] proposed the session-based admission control, which monitored the performance of web services by counting the completed sessions. When the web services become overloaded, the admission control rejects requests for creating new sessions. Chen et al. [5] further proposed the QoS-aware session-based admission control by exploiting the dependencies of sessions to improve the service quality during service overload. However, these techniques favor long-lived sessions, making them unsuitable for the WeChat application which incorporates tremendous short-lived and medium-lived sessions. Differently, DAGOR's admission control is user-oriented rather than session-oriented, and hence does



(a) CoDel (b) DAGOR

Figure 9: Fairness of overload control.

not bias any session-related property of the service for overload control. Moreover, the above session-based techniques rely on a centralized module for overload detection and overload control. Such centralized module tends to become the system bottleneck, resulting in limited scalability of the system. In contrast, DAGOR by design is decentralized so that it is highly scalable to support the large-scale microservice architecture.

Welsh et al. [31, 32] proposed the technique of staged overload control, which partitioned the web services into stages with respect to the service semantics and performed overload control for each stage independently. Each stage is statically allocated with a resource quota for load constraint. Although such mechanism shares some similarity with DAGOR, its prerequisites of service partitioning and static resource allocation render it inapplicable to the complex, dynamic service architecture such as the WeChat backend. In contrast, DAGOR is service agnostic, making it flexible and highly adaptable to the continuously evolving microservice architecture.

Network overload control targeting at reduced reponse time has been well studied [16]. Our experience of operating WeChat business system shows that latency-oriented overload control is hardly effective when used for the large-scale microservice architecture. This motivated us to employ queuing time for overload detection in DAGOR. Moreover, techniques discussed in the THEMIS system [17] inspire us to take fairness into account in the DAGOR design.

7 CONCLUSION

This paper proposed the DAGOR overload control for the microservice architecture. DAGOR by design is service agnostic, independent but collaborative, efficient and fair. It is lightweight and generally applicable to the large-scale, timely evolving microservice systems, as well as friendly to cross-team agile development. We implemented DAGOR in the WeChat service backend and have been running it in the WeChat business system for more than five years. Apart form its effectiveness proved in the WeChat practice, we believe DAGOR and its design principles are also insightful for other microservice systems.

Lessons Learned. Having operated DAGOR as a production service in the WeChat business backend for over five years, we share lessons we have learned from our development experience as well as design principles below:

- Overload control in the large-scale microservice architecture must be decentralized and autonomous in each service, rather than counting on the centralized resource scheduling. This is essential for the overload control framework to scale with the ever evolving microservice system.
- The algorithmic design of overload control should take into account a variety of feedback mechanisms, rather than relying solely on the open-loop heuristics. A concrete example is the strategy of collaborative admission control in DAGOR.
- An effective design of overload control is always derived from the comprehensive profiling of the processing behavior in the actual workload. This is the basis of DAGOR's design choices of using the request queuing time for overload detection as well as devising the two-tier priority-based admission control to prevent subsequent overload.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable comments and constructive suggestions that helped improve the paper.

REFERENCES

[1] V. A. F. Almeida and D. A. Menasce. 2002. Capacity planning an essential tool for managing Web services. *IT Professional* 4, 4 (2002), 33–38.
[2] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. 1999. Resource Containers: A New Facility for Resource Management in Server Systems. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.
[3] Azer Bestavros and Sue Nagy. 1997. Admission Control and Overload Management for Real-Time Database. In *Real-Time Database Systems: Issues and Applications*. 193–214.
[4] Sirish Chandrasekaran and Michael Franklin. 2004. Remembrance of Streams Past: Overload-sensitive Management of Archived Streams. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*.
[5] Huamin Chen and P. Mohapatra. 2002. Session-based overload control in QoS-aware Web servers. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*.

[6] Xiangping Chen, Prasant Mohapatra, and Huamin Chen. 2001. An Admission Control Scheme for Predictable Server Response Time for Web Accesses. In *Proceedings of the International Conference on World Wide Web (WWW)*.

[7] L. Cherkasova and P. Phaal. 2002. Session-based admission control: a mechanism for peak load management of commercial Web sites. *IEEE Transactions on Computers (TC)* 51, 6 (2002), 669–685.

[8] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. 2014. Efficient Coflow Scheduling with Varys. In *Proceedings of the ACM SIGCOMM International Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*.

[9] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*.

[10] Fahad R. Dogar, Thomas Karagiannis, Hitesh Ballani, and Antony Rowstron. 2014. Decentralized Task-aware Scheduling for Data Center Networks. In *Proceedings of the ACM SIGCOMM International Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*.

[11] Sameh Elnikety, Erich Nahum, John Tracey, and Willy Zwaenepoel. 2004. A Method for Transparent Admission Control and Request Scheduling in e-Commerce Web Sites. In *Proceedings of the International Conference on World Wide Web (WWW)*.

[12] Thomas Erl. 2005. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall.

[13] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google File System. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*.

[14] Jörgen Hansson, S. F. Andler, and Sang Hyuk Son. 1999. Value-driven multi-class overload management. In *International Conference on Real-Time Computing Systems and Applications (RTCSA)*.

[15] Yuxiong He, Sameh Elnikety, James Larus, and Chenyu Yan. 2012. Zeta: Scheduling Interactive Services with Partial Execution. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*.

[16] Virajith Jalaparti, Peter Bodik, Srikanth Kandula, Ishai Menache, Mikhail Rybalkin, and Chenyu Yan. 2013. Speeding Up Distributed Request-response Workflows. In *Proceedings of the ACM SIGCOMM International Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*.

[17] Evangelia Kalyvianaki, Marco Fiscato, Theodoros Salonidis, and Peter Pietzuch. 2016. THEMIS: Fairness in Federated Stream Processing Under Overload. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*.

[18] K. Kant and Y. Won. 1999. Server capacity planning for Web traffic workload. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 11, 5 (1999), 731–747.

[19] Qian Lin, Beng Chin Ooi, Zhengkui Wang, and Cui Yu. 2015. Scalable Distributed Stream Join Processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*.

[20] Ben Maurer. 2015. Fail at Scale. *ACM Queue* 13, 8 (2015), 30:30–30:46.

[21] Tony Mauro. 2015. Adopting Microservices at Netflix: Lessons for Architectural Design. https://tinyurl.com/htfezlj.

[22] Daniel A. Menasce and Virgilio Almeida. 2001. *Capacity Planning for Web Services: Metrics, Models, and Methods*. Prentice Hall PTR.

[23] Pieter J. Meulenhoff, Dennis R. Ostendorf, Miroslav Živković, Hendrik B. Meeuwissen, and Bart M. Gijsen. 2009. Intelligent Overload Control for Composite Web Services. In *Proceedings of the International Joint Conference on Service-Oriented Computing (ICSOC-ServiceWave)*.

[24] Jeffrey C. Mogul and K. K. Ramakrishnan. 1997. Eliminating Receive Livelock in an Interrupt-driven Kernel. *ACM Transactions on Computer Systems (TOCS)* 15, 3 (1997), 217–252.

[25] Kathleen Nichols and Van Jacobson. 2012. Controlling Queue Delay. *ACM Queue* 10, 5 (2012), 20:20–20:34.

[26] Lalith Suresh, Peter Bodik, Ishai Menache, Marco Canini, and Florin Ciucu. 2017. Distributed Resource Management Across Process Boundaries. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*.

[27] Nesime Tatbul, Uğur Çetintemel, and Stan Zdonik. 2007. Staying FIT: Efficient Load Shedding Techniques for Distributed Stream Processing. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*.

[28] N. Tatbul and S. Zdonik. 2006. Dealing with Overload in Distributed Stream Processing Systems. In *22nd International Conference on Data Engineering Workshops (ICDE Workshops)*.

[29] Thiemo Voigt, Renu Tewari, Douglas Freimuth, and Ashish Mehra. 2001. Kernel Mechanisms for Service Differentiation in Overloaded Web Servers. In *Proceedings of the USENIX Annual Technical Conference (ATC)*.

[30] Chieh-Yih Wan, Shane B. Eisenman, Andrew T. Campbell, and Jon Crowcroft. 2007. Overload Traffic Management for Sensor Networks. *IEEE/ACM Transactions on Networking (ToN)* 3, 4 (2007).

[31] Matt Welsh and David Culler. 2003. Adaptive Overload Control for Busy Internet Servers. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)*.

[32] Matt Welsh, David Culler, and Eric Brewer. 2001. SEDA: An Architecture for Well-conditioned, Scalable Internet Services. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*.

[33] Ying Xing, Jeong-Hyon Hwang, Uğur Çetintemel, and Stan Zdonik. 2006. Providing Resiliency to Load Variations in Distributed Stream Processing. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*.