2022 IEEE 28th International Conference on Parallel and Distributed Systems (ICPADS) | 978-1-6654-7315-6/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICPADS56603.2022.00065

Dependency-Aware Traffic Management for Configuring On-demand in Service Meshes

Lin Wang*, Xin Li*†, Ning Wang*, Hao Li*, Xiaolin Qin*, and Jie Wu‡
*CCST/AI, Nanjing University of Aeronautics and Astronautics, Nanjing, China
†State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
‡ Center for Networked Computing, Temple University, Philadelphia, USA

***Abstract*—Service mesh is a promising micro-services architecture due to its excellent governance capabilities. Unlike traditional service invocation, configurations for governance need to be issued in the service mesh. However, we find that the control-plane traffic of governance is distributed in full by default, i.e., each service in the data plane receives all configurations. The vast majority of the configurations are redundant for a specific service. Hence, it is important and challenging to make the control plane aware of the calling relationships between services. In this paper, we propose a traffic management mechanism named DATM. Using this mechanism, the entire cluster can be dynamically controlled and services can be configured on demand. It is implemented through a dependency-aware controller and monitors. The controller first processes the information listened to by the monitors and then analyzes the connection between the metrics and the service requests through intelligent algorithms. Finally, the control traffic for regulating the control plane is generated. Our proposed mechanism is experimentally compared with the default strategy and existing work across a wide set of load scenarios in a testbed based on Istio service mesh and Kubernetes. Experimental results demonstrate that our mechanism can save the storage resources of a single agent by 40% to 60%, and the number of cluster updates can be greatly reduced. From the perspective of the whole cluster, the optimization results are even better.**

***Index Terms*—Service Mesh, Traffic Management, Configure On-demand, Cost Reduction**

I. Introduction

Service mesh [1] has emerged to deal with the problem of service-to-service communications in distributed systems or micro-services [2]. Lightweight proxies, living alongside the applications fetch configured rules from the control plane and perform governance logic when intercepting inbound and outbound traffic. In this way, the service mesh takes over the traffic of the cluster, providing higher-order capabilities like network resilience, security, traffic management [3], etc.

Cost reduction and efficiency improvement have undoubtedly become the trend in the industry. To improve the utilization of existing resources [4], various technologies have been proposed, such as (1) Resource scheduling and business deployment, (2) Resource utilization estimation, (3) Application expansion and configuration, and (4) Workload management [5]. Consider resource utilization from the perspective of traffic management, i.e., to improve system resource utilization by controlling the direction and amount of traffic through appropriate policies. Istio [6] is a powerful service mesh implementation that makes it easier to operate cloud-native services architectures in a hybrid environment. Traffic in the Istio is architecturally divided into data-plane traffic which refers to the traffic invoked for services between services, and control-plane traffic which refers to the configuration between components of the control plane and the traffic to manage the proxies. Existing research mostly focuses on the data-plane traffic, with control-plane traffic receiving less attention. The native mode of control-plane traffic is to configure fully, which wastes memory and blocks threads. For example, in Fig. 1, service *A* calls service *B* in the data plane cluster, meaning that in ideal mode, *A* only requires the configuration of *B* and some basic configuration. However, for the native mechanism, when the request arrives to trigger the control plane component, it delivers according to the service in the cluster rather than the configuration that *A* needs. Hence, *A* will receive DiscoveryResponse with xDS (x Discovery Service) related to *C* and *D* at the same time. *B*, *C*, and *D* will also receive the same configuration, even if they do not have any calling services. In a mesh with 325 clusters and 175 listeners, one proxy occupies about 100M of memory; with 466 instances in the mesh, all proxies use 466*100M=46.6G. This consumes a significant amount of storage space in data-plane proxies and causes the thread block to process the redundant configuration.

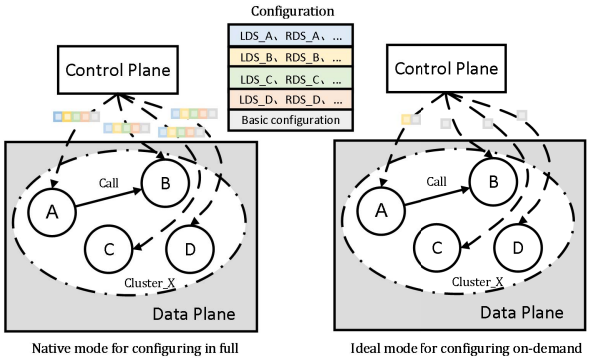


Fig. 1. Two modes of configuring.

The performance issue caused by fully configuring is the main issue at present. To direct traffic between meshes, the control plane needs to know where the endpoints (Pods in Kubernetes) are, and the dependencies between services [7].

978-1-6654-7315-6/23/$31.00 ©2023 IEEE
DOI 10.1109/ICPADS56603.2022.00065

Therefore, a good control mechanism is needed to minimize the amount of configuration receiving services to improve the utilization of existing resources. We intend to investigate the traffic management of the control plane under micro-services. The contributions of this paper can be summarized as follows:

- We design the **D**ependency-**A**ware **T**raffic **M**anagement mechanism (DATM), combining monitors and a controller, a service mesh oriented mechanism for configuring on demand. The mechanism is application-agnostic, non-intrusive, and does not require any source or business logic code changes.
- We implement the controller of the control-plane traffic in the form of plugins to obtain the service's dependencies. The configuration can be distributed on demand.
- We conduct extensive experiments to evaluate these solutions.

The rest of this paper is organized as follows. We review the related work in Section II and give the problem statement of traffic management in Section III. Section IV presents our proposed solution framework. Then, we conduct some experiments and evaluate the efficiency of the controller in Section V. Finally, we conclude the paper in Section VI.

II. Related Works

Extensive research on micro-services, dynamic performance modeling, and traffic management exists currently. Most of them focus on the intersection of these areas. Some particularly notable individual studies are highlighted below.

The most typical micro-services architecture, Spring Cloud [8], builds the Software Development Kit (SDK) in the application. The service governance capability is not suitable for the integration of heterogeneous systems. The service mesh [9] makes business processes more focused on business logic. Various functions of Istio service mesh have been used to extend the high-order capabilities in the cloud-native system as originally envisioned. Cilium [10] adds network security filtering to Linux container systems such as Docker and Kubernetes, which use eBPF to enforce network and application-layer security policies on containers and pods.

Dynamic performance management for clouds has attracted considerable attention. One core issue that is gaining notable reviews is extensive resource estimation [11]. Maximum traffic is frequently used to predict the resources required by the service during deployment, resulting in severe resource waste. Since micro-services have expensive performance overhead, [12] devised a method for performance modeling and task scheduling. Suresh et al. proposed a service-oriented architecture based on rate restrictions and per-service scheduling to optimize the overall ability to meet deadlines [13]. Within the industry, many companies have started providing control solutions based on Sidecar [14]. Slime [15] has a modular architecture internally, based on the k8s-operator implementation of Lazyload, which acts as a CRD manager for Istio. It loads configuration and service discovery information, adjusts current restriction policies with monitoring information, and maintains new plugins. The TCM team designed the lazyXds scheme [16].

TABLE I
Prometheus Metric

	Filter Properties			Other Properties	
s_1	$status_1$	$mode_1$	$destination_1$	$value_1$	$resultType_1$
s_2	$status_2$	$mode_2$	$destination_2$	$value_2$	$resultType_2$
...	...	...	...	...	...
s_i	$status_i$	$mode_i$	$destination_i$	$value_i$	$resultType_i$

Some aspects of traffic management in need of further research have been highlighted in [17]. Work [18] proposes a cache-based circuit-breaker strategy, argue for the possible request failure problem in a distributed system, and quickly make the request fail and then give the user a result by returning a cache. Maggio [19] and Xu [20] targeted the problem of how algorithms and controllers can be used to optimize service delivery and cloud infrastructure power consumption. Their approach hinges on using a quality reduction to increase efficiency. Traffic management under multi-cloud is studied frequently. [21] improved the performance of load balance by a decentralized algorithm in a centralized system.

In contrast to the works mentioned above, this paper proposes a controller solution for dependency-aware adaptive management of control-plane traffic, which is a new attempt of traffic management. At the same time, We measure the performance of the controller by the memory before and after algorithm improvement.

III. Problem Analysis

This section presents the configuration optimization based on the service mesh, which includes service description, problem scenario, memory usage, and expected improvement.

A. Service Description

Our research is based on a number of micro-services in distributed clusters. Services in a cluster may contain many types. However, not all services are needed because services of the control component are not related to control-plane traffic. In order to distinguish between different types of services, we have selected the state and characteristics shown in Table I. This is reasonable enough, because the state of service may change over time. To clearly describe the micro-service, we use a tuple to characterize it as $\gamma = \langle \chi, \psi, \xi, \sigma \rangle$, where

– χ indicates whether the service is up;

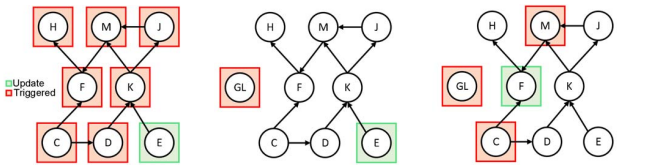
– ψ is used to identify whether the service is a business service or not;

– ξ indicates which namespace the service belongs to;

– σ is the all related destinations of this service.

For the service, we have

$$\psi\left(\gamma_j\right) = \begin{cases} 1, & \gamma_j \text{ is a business service;} \\ 0, & otherwise. \end{cases} \tag{1}$$



(a) Default mode. (b) An edge service. (c) Service with a calling relationship.

Fig. 2. Examples of triggering updates.

$$\chi\left(\gamma_j\right) = \begin{cases} 1, & \gamma_j \text{ is a running service;} \\ 0, & otherwise. \end{cases} \tag{2}$$

B. Problem Scenario

Service meshes of agents for each service can intercept all inter-service calls in an application-independent manner, and provide more advanced communication capabilities, such as traffic management. For the native mechanism, the control plane must distribute and maintain all information in the mesh. Each service can access any service in the mesh and its agent hold the full amount of data, including service discovery information, network routing rules, and network security rules. However, the vast majority of the data are not available.

Since it is impossible to predict the dependencies of service in the mesh, Pilot, the core component of the control plane, will be unable to deliver accurate data according to the requirements but provides what data are available based on the existing cluster. When a change is detected, Pilot sends a DiscoveryResponse (complete configuration) to Envoy [22], the data plane's fundamental component, to update the configuration, which will cause adverse effects on both the data plane and control plane traffic. In a large cluster, the configuration update frequency is higher and the number of instances is larger. If one of them is updated, the full configuration will be pushed to all Envoys. Fig. 2(a) shows that when an edge service E is updated, it triggers the control plane to distribute the configuration to all service proxies in the cluster in default mode. In fact, in the default mode, any service update in the cluster will trigger the update of all services and all updates are full configuration delivery.

Full configuration delivery of the control plane will result in an increase in memory during a Pilot push, which might easily lead to OOM. For the data plane, Envoy will receive a lot of redundant configuration, so that the memory overhead increases. The memory occupied by Envoy and the number of its configured endpoints is linear. The more endpoints, the more memory that Envoy occupies. Increased memory use is a significant drain. Our ideal control effect is shown in Fig. 2(b) and Fig. 2(c). When an edge service is updated, if there is no service it depends on, it only updates itself without triggering other service updates. For the updated information, additional service can be added to record all the updated information on the entire cluster for thorough processing. When a service with dependencies is updated, we need to trigger the update of the services that the service is connected to at the same time, and the rest of the services are not processed.

C. Memory Usage

There are n services $S = \{s_1, s_2, ..., s_n\}$ in working cluster. Let $s_i = \left\{s_i^1, s_i^2, ..., s_i^m\right\}, i \in [1, n]$ represents the instance of this service. I_i is the number of instances of micro-service s_i. We optimized memory to minimize storage by considering the amount of configuration.

As the size of the service within the cluster increases, the amount of storage needed by the agent and the quantity of updates both considerably rise. The relationship between memory usage U and load L:

$$U \propto k_1 L \ (k_1 > 1) \tag{3}$$

The main configuration size of the service P is determined by bootstrap, listeners, clusters, routes and a basic file in json format. The relationship between load L and configuration size of service P :

$$L \propto k_2 P \ (k_2 > 1) \tag{4}$$

Most of the composition of memory usage U is the accumulation of the configuration of all instances. Since the configuration is delivered in full, the configuration is the same size regardless of the instance, so they are all represented by P. The relationship between memory usage U and configuration size of service P:

$$U \propto (\sum_{i=1}^{n} I_i) P \tag{5}$$

D. Expected Improvement

Storage upgrade for a single service *Minimize* can be regarded as omitting the namespace's necessary configuration and default settings, such as node and software version information, admin address configuration, and trace Zipkin cluster address reference, from the entire configuration.

$$Minimize = n * P - \sigma\left(\gamma_j\right) - \omega\left[\xi\left(\gamma_j\right)\right] \tag{6}$$

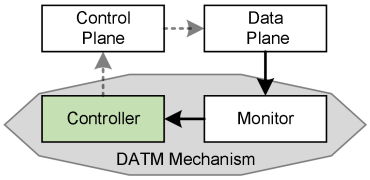
Expected improvement EI represents the storage space saved for the whole mesh. It can be defined as:

$$EI = U - \sigma\left(\gamma_j\right) * \sum_{j=1}^{n} I_j - W, \tag{7}$$

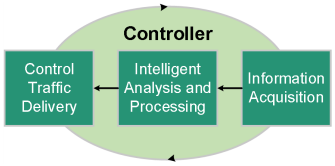
Here, $W \in \left\{\xi\left(\gamma_j\right)\right\}, j \in [1, n]$. The EI contains configuration that is not needed for this service, including configuration information that is not relevant to this service call.

IV. DTMA Mechanism

The whole DATM mechanism consists of a cloud environment, monitor, and controller as illustrated in Fig. 3(a).



(a) Traffic management infrastructure architecture overview showing the control traffic as a dotted gray line and the data traffic through Cache and Estimator components in black.



(b) Sketch of the execution logic of the aware-dependency controller.

Fig. 3. The logic structure of the DATM mechanism

The basic idea of the controller consists of four parts: real-time connection and monitoring, information acquisition, intelligent analysis and processing, and control of information distribution as illustrated in Fig. 3(b). We design the controller that uses Sidecars to limit the reachability of proxy. Sidecar can explicitly define dependencies, or visibility relationships, between services. For example, Reviews service only relies on Ratings service. So after configuration, Istiod will only distribute Ratings information to Reviews. The controller considers both the namespace and the monitoring data, thus improving the real-time performance of the traffic management, and ultimately each agent will only get its required configuration, with optimal performance. Based on the above studies, We here propose a controller component that can be easily deployed in a service mesh. The controller's component is not deployed directly in the cluster. Rather, it configures Kubernetes API resources to govern cluster traffic behavior.

A. Monitor

Real-time connect and listen to the running cluster with a monitor. It is difficult to know the service status and to a priori determine call dependency before services run. During development, a software developer cannot reasonably know what time service works, unless the underlying logic is known to be stable over time. However, as our results show, it is possible for the software to check the service calling relations in real-time. Determining which actions are possible to cache and storage length can also be a challenge. There is no such definition at the protocol level and the control level, so developers and operators can judge the behavior to be performed by some attribute information defined by a monitor. The monitor server can pull monitoring indicator data from the active (up) target host regularly and save it locally as a time series. Configuring static tasks or service discovery to monitor the services in the Kubernetes cluster can collect monitoring data. The triggered alarms are delivered to the Alertmanager by configuring the alarm rules. Using the monitoring logs indicator data, obtain real-time information about the Kubernetes cluster, and store the services participating in the business and their accompanying service dependencies in the configuration filtering controller according to a predetermined format.

B. Information Acquisition

Get the data required for cluster traffic management from the monitoring system. The data are obtained after the full configuration is called, when the business service expands or decreases, or when the dependence relationship between the service change. Acquire the latest service information in the cluster, as well as the service dependency information that goes with it, and preserve it in the configuration filtering controller. The local storage is compared to the most recent configuration information, and the result is forwarded to the controller's task queue, which updates the local storage in the order that the comparison results arrive.

C. Intelligent Analysis and Processing

First of all, the acquired data are sorted out and stored in Newest Config. Then, the local existing stored data of Newest Config are passed into a Control-Loop for Diff processing, and the action that needs to be updated is analyzed. It is necessary to add a Sidecar for a new service. A Sidecar needs to be deleted or the existing Sidecar configuration needs to be updated. The generated update action and update data are passed into a work queue for execution.

Alg.1. gives the pseudocode of the configure-on-demand algorithm. The controller is configured with one parameter: the service's basic information. The result is provided to the control plane, which distributes it to each sidecar in xDS and records all activities to Etcd to provide data support for other services based on the policy's final service dependency. To function, our controller requires the metric of the recently executed service as input. We select all services with Istio's *metric_destination_service* as running services from the result array that Prometheus monitored. Then filter them as business services using *security_istio_io_tlsMode* from Istio, which correlates well with Istio's business services. We deposit the names of these services in a slice and then clean it through the map to determine the length of the business service queue, assuming that there is more than one business instance of a service, most likely multiple. To test our controller, we leverage the above-mentioned metrics and Istio's *istio_requests_total* metrics. The basic idea behind the processing algorithm is a three-step approach:

Initialization (Line 1-4). This step initializes some variables. The *GApp* is a priority queue that records all applications in the cluster. The *Qlist* is used to store the running business services of Istio. The *Cmap* is a map data structure based on Hash tables. The initialized *Cmap* value is set to nil, which is used as the call relationship between receiving services. The *Dlist* is used to store the services which have the calling relationship.

Algorithm 1: Configure On-demand Algorithm

Input: Metric M;
Output: Newest Sidecar Configuration $Cmap$;

```
1  GApp ← getGlobalApplication(M);
2  Qlist ← ∅;
3  Cmap ← ∅;
4  Dlist ← ∅;
5  for q ∈ GApp do
6      if isRunning(q) ∩ q.isContains(security_istio_io_tlsMode) then
7          if Qlist.notContains(q) then
8              Qlist.collect(q);
9          else
10             continues;
11         end
12         continues;
13     end
14 end
15 for p ∈ Qlist do
16     Cmap.keyCollect(p);
17     if Dlist == p.getProperties(metric_destionation_service) then
18         while Dlist do
19             Cmap.valueCollect(p);
20         end
21     else
22         ns = getNamespace(p);
23         Cmap = DefaultSidecarConfiguration(ns);
24     end
25     continues;
26 end
```

Generate the initial cluster relationships (Line 5-14). In this step, we calculate the initial relationships of the cluster. We first use a condition query to find all the running services. Note that we may get a lot of worthless data, such as services that have run out of time or services that are used to control scheduling. For a service with on-demand enabled, the controller stores the dependency of the service in the cache and creates a Sidecar resource for it to limit the visibility of this app and notifies Kubernetes' APIServer of updates to the service.

Determine the final results (Line 15-26). This component will check whether the *Qlist* initial relationships need to be collected in the *Cmap* key. We calculate the new service information, including Namespace and default configuration. By comparing the original call information with the new call information, the controller expresses the new dependency into the Sidecar to update the resources.

D. Control Traffic Delivery

With the increasing number of services in the distributed clusters, the configuration quantity is more and more required. Therefore, the topic of loading configuration must be explored in the micro-service environment. Pilot isolates services by namespace when creating listeners and clusters, reducing the number of listeners and clusters in the Envoy somewhat. But it's still too rough and has limited memory optimization effects. For this problem, the community provides a solution: Sidecar, the Custom Resource Define (CRD), but this requires manual operation and is cumbersome.

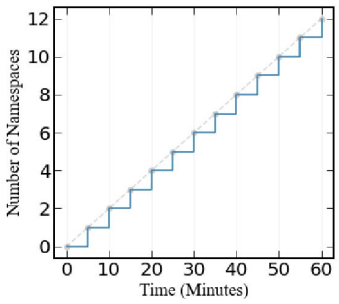


Fig. 4. The generated workload scenario.

When the resources are distributed, the traffic control resources mainly control the routing of services and the traffic of cluster entrances and exits. The final dependency of each service obtained according to the policy will be transmitted to the control plane. Meanwhile, all operations will be recorded to Etcd to provide data support for other services. The controller is implemented in the Golang language. This scheduling mechanism will follow this process in each working cycle until a manual stop or failure. Our implementation design objectives are scalable, suitable for research through instrumentation/observability, and easily integrated with the existing service mesh. The latter implies an application-independent approach such that existing services can benefit without modifying the source code.

V. Performance Evaluation

We start by describing the experimental setting, followed by a discussion of the pressure test scenario we created, and finally a comparison of different strategies. We want to count the number of update times and memory usage for different settings.

A. Experiment Setup

Experiment Setting. To evaluate our proposed method, all experiments are performed on a bare-metal machine with Hygon C86 7151 16-core CPU, and a 256 GB NVMe hard drive running Ubuntu 18.04 LTS. The cluster is set up with Docker 20.10.7, Kubernetes 1.20.0, and Istio 1.8.2. We test a mock service in three kinds of isolated environments to evaluate the performance: our proposed controller (DATM), loading configuration by default modes (Default), and a control strategy developed using scaffolding (Lazyload). In the study of configuration delivery improvements for service mesh, the Lazyload is the best seen in the industry.

Benchmarks. Bookinfo [23] is the benchmark in our experiments. It is a basic sample bookstore application made up of four services that provide a web productpage, book details,

reviews, and ratings—all of which are stateless and use Istio to demonstrate the fundamental phases of traffic management. We developed three sets of Bookinfo in different namespaces in the same mesh, one uses DATM in the namespace, the other uses Lazyload in the namespace, and the remaining one does nothing. The obtained service deployment information in the cluster is as follows:

- Four business services: productpage, details, reviews, and ratings. There are one version of productpage-v1 and details-v1 respectively. Reviews have three versions.
- A global configuration service that contains the configuration information for all of the services, e.g., service discovery information, network routing rules, and network security rules.

Workload test settings. The workload for the experiment is generated by the Isotope (the "Load Service") [24], Istio's official loading-test tool set. It is a comprehensive application with configurable topology results. The simulation services component of Isotope is a reasonably simple HTTP server that obtains Prometheus measures by following the instructions in YAML files. The GKE (Google Kubernetes Engine) cluster runs the Fortio1.1.0 client and an Isotope service, with the service machine limited to 1vCPU and 3.75GB of memory. To imitate mesh scale growth, we gradually increase the number of workload services. Each namespace contains 19 services, with each initial pod consisting of 5 services for a total of 95 pods. The workload traffic scenario, a multiple server load, is shown in Fig. 4. The experiments are each 60 minutes long (3600 seconds), constituting a complete period for the above workload.

B. Tuning the controller

Our controller can adapt to the workload, complex service dependencies, and cluster service changes. Various methods for such controller tuning have been proposed [25]. However, because the use of these methods is beyond the scope of this work, we will concentrate on comparing the proposed controller to the default mode and Lazyload. To define good measures for determining appropriate controller settings, we deployed the Bookinfo service and our controller (as specified in Algorithm 1) on our server. We use the benchmark work Isotope officially recommended by Istio as the influencing factor, the workload of a single namespace has 18 services, and the generated workload is shown in Fig. 4, adding one namespace workload every 5 minutes. Based on these experiments, we conclude that it is a smart option to include commonly used dependencies in Sidecar. This is consistent with our intuition that the controller should avoid abrupt output changes and maintain relative stability. Prometheus is widely regarded as a good monitoring architecture for gathering cluster service information and calling relationships. Hence, we use it to monitor the whole system in our proposed mechanism and deploy DATM on the Master node. The controller queries Prometheus to obtain the service information and the dependencies of services and then tunes the relationship of running the service in the Sidecar's configuration (EgressHosts).

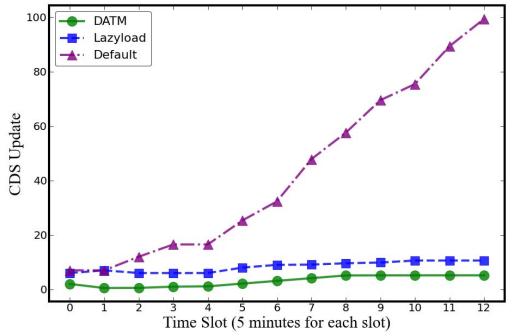


Fig. 5. The number of CDS update.

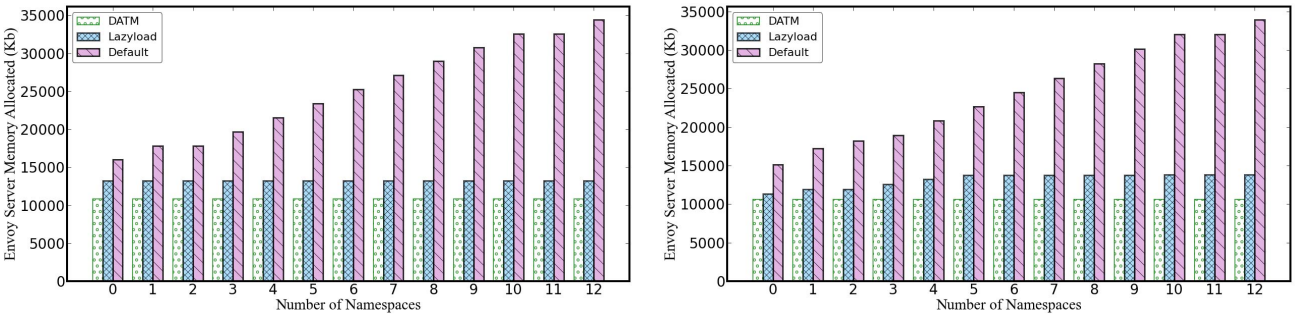
C. Comparing the number of updates

We compare the metric of updates using Prometheus' *envoy_cluster_manager_cluster_updated*. In the passive distribution mode of the Pilot and Envoy communication, the Envoy subscribes to specific resource events, generates configurations and delivers them when the resource is updated. Envoy's full update strategy provides strongly consistent configuration synchronization as a stream. As a result, any change will trigger the full configuration delivery, which imposes a high burden on the entire mesh.

Fig. 5 manifests the performance comparison of Default delivery mode, Lazyload mode and DATM mode in Istio. In contrast, the DATM mode reduces redundant service data more than the other two modes. The number of CDS updates increases dramatically as the workload increases in the default namespace. As the load within the cluster increases and the number of instances proliferates, both DATM and Lazyload scenarios do not update as drastically as in the Default mode because of the restrictions placed on configuration distribution. Lazyload uses a form of scaffolding in the form of operator-sdk to change the scope of the limit, while DATM is a more radical rewriting of the sidecar (CR). With our controller in action, the control plane delivering the xDS protocol associated with Sidecar limits the service visibility so that the envoy does not receive the full xDS updates in the same scenario. Therefore, the service is shielded from the majority of the system's unnecessary update requests.

D. Comparing Envoy Memory

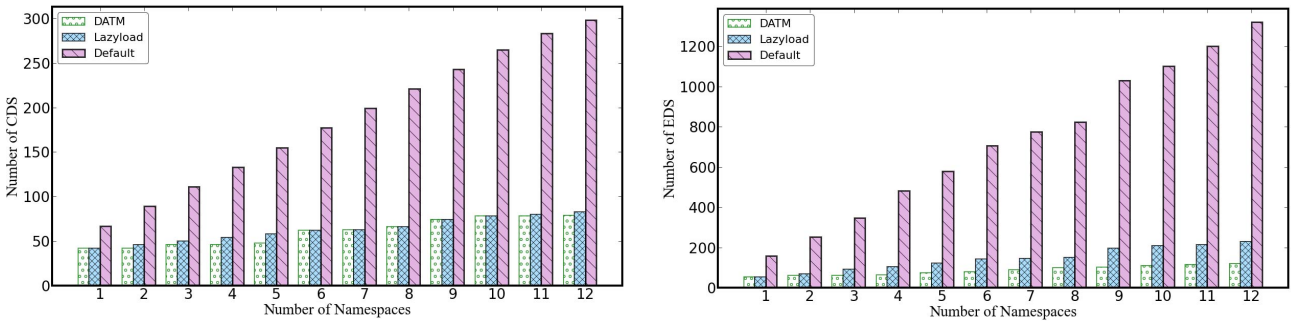
In the memory size experiment, we change the cluster load while deploying the same service. Examine the impact of adding a large number of instances to the cluster on the deployed service's proxy memory. Recalling Fig. 4, we simulated the traffic rate in each experiment. The results for memory usage in three modes of different workloads are shown in Fig. 6. The memory utilization for our controller case is 11M, with a mesh size of 360 Pods (5 namespaces). The default configuration is 25M, and Lazyload configuration is 13M. This approach reduces memory by 14M relative to the mesh size, by about 56%. As we can see in the figure, our



(a) Productpage envoy memory

(b) Ratings envoy memory

Fig. 6. Envoy server memory allocated for micro-services in different workload scenarios.



(a) CDS

(b) EDS

Fig. 7. The number of CDS/EDS in different workload scenarios.

controller results match Lazyload's and work a little better than it. While our controller is only marginally better than Lazyload, this is just for one agent in the cluster. As the number of service versions and instances grows, the overall cost savings become significant.

In short, the amount of statically configured agent memory varies substantially depending on the load size, whereas the proposed controller keeps agent memory minimal. When the system is loaded, both dynamic and static configurations have no effect on the standard application services. These results indicate that our controller is able to maintain a reasonable capacity use and application capabilities.

E. Comparing the number of CDS and EDS

Comparing CDS and EDS, each set of data shown in Fig. 7 represents an increase in load service namespace, with three values in each data set: Fig. 7(a) is the number of CDS detected in the DATM, Lazyload and Default namespace respectively, and Fig. 7(b) is the number of EDS detected in the DATM, Lazyload and Default namespace respectively. CDS dynamically obtains cluster information and is the xDS protocol with the most changes. Envoy cluster views all upstream clusters in it. An Envoy typically abstracts an upstream cluster from a Listener (for TCP) or Route (for HTTP) as a traffic forwarding target. As the number of namespaces increases, it represents an increase in the number of services within the entire cluster and a heavier cluster load. The advantage of DATM is greater in larger clusters because the larger the cluster, the greater the percentage of individual services that do not want to close with other services, the more stored information is intercepted, and therefore the less CDS and EDS are stored relatively. We demonstrate traffic management on an official micro-service case, which has relatively little impact on user traffic, and user traffic will not block. The performance loss is also relatively small.

F. Comparing the different service

To determine the impact of the controller on services with different business relationships, we deployed the same service scenario and stress test to evaluate the memory consumption of Productpage with complex invocation relationships and Ratings with invoked relationships only. Fig. 6(a) and Fig. 6(b) respectively show memory usage of a proxy, which also prove that the configuration in DATM generally maintains the Envoy memory below 13M and 11M independently of the service. Because no calling service can be written into the *workloadSelector* field for Ratings as the called party, we examined the service's memory use in both controlled and uncontrolled scenarios to see if our controller needs to act on it. From Fig. 8, we observe that although Ratings do not require a dedicated Sidecar, it appears that configuring a sidecar without any *workloadSelector* field to apply to all

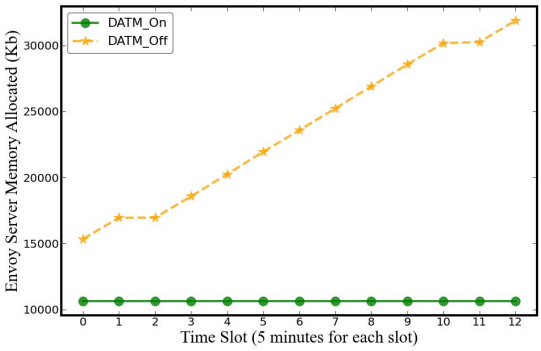


Fig. 8. The impact of DATM on Ratings in the same namespace.

workloads in a particular namespace is preferred in order to only reach services running in the same namespace and the Istio control plane (e.g. required by Istio's egress).

VI. Conclusion

In this paper, we creatively propose a DATM mechanism that combines an adaptive controller and monitors, avoiding the issues that come with fully configuring by default. The controller uses control theory of traffic management architecture. This dynamic approach enables the control plane to be aware of dependencies between services, saving storage usage and reducing update times. We test it in a wide number of scenarios. The experimental results demonstrate that our controller can save the storage space of a single agent by 40% to 60% and the number of cluster updates can be greatly reduced. The optimization results are even better when viewed as an entire cluster. And the results show that our proposed approach can be applied to black-box services and effectively reflects the dependencies between services in the entire cluster for configuration.

Acknowledgment

This work is supported in part by the National Key R&D Program of China under Grant 2019YFB2102002, and in part by the Collaborative Innovation Center of Novel Software Technology and Industrialization.

References

[1] J. Lewis and M. Fowler, "Microservices," *library Catalog: martinfowler.com*, 2014. [Online]. Available: https://martinfowler.com/articles/

[2] S. Wang, Y. Guo, N. Zhang, P. Yang, A. Zhou, and X. Shen, "Delay-aware microservice coordination in mobile edge computing: A reinforcement learning approach," *IEEE Trans. Mob. Comput.*, vol. 20, no. 3, pp. 939–951, 2021. [Online]. Available: https://doi.org/10.1109/TMC.2019.2957804

[3] W. Li, Y. Lemieux, J. Gao, Z. Zhao, and Y. Han, "Service mesh: Challenges, state of the art, and future research opportunities," in *13th IEEE International Conference on Service-Oriented System Engineering, April, 2019*. IEEE, 2019. [Online]. Available: https://doi.org/10.1109/SOSE.2019.00026

[4] J. Wang, J. Cao, S. Wang, Z. Yao, and W. Li, "IRDA: incremental reinforcement learning for dynamic resource allocation," *IEEE Trans. Big Data*, vol. 8, no. 3, pp. 770–783, 2022. [Online]. Available: https://doi.org/10.1109/TBDATA.2020.2988273

[5] B. G. S. Costa, J. B. Jr., L. R. de Carvalho, and A. P. F. Araújo, "Orchestration in fog computing: A comprehensive survey," *ACM Comput. Surv.*, vol. 55, no. 2, pp. 29:1–29:34, 2023. [Online]. Available: https://doi.org/10.1145/3486221

[6] "Istio - connect, secure, control, and observe services." [EB/OL], https://istio.io/ Accessed 2022.

[7] Jingchao Song, et al, *Istio Handbook - advanced practice of Istio Service Mesh [M]*. Beijing: Electronic Industry Press, 202008:10-18.

[8] Binildas Christudas, *Practical Microservices Architectural Patterns[M]*. Berkeley, CA: Apress, 2019, no. 4.

[9] M. Delavergne, R. Cherrueau, and A. Lebre, "A service mesh for collaboration between geo-distributed services: The replication case," in *Agile Processes in Software Engineering and Extreme Programming - Workshops - XP 2021 Workshops, Virtual Event, June 14-18, 2021*. Springer, 2021. [Online]. Available: https://doi.org/10.1007/978-3-030-88583-0_17

[10] A. V. Kuznetsov, "Protein transport in the connecting cilium of a photoreceptor cell: Modeling the effects of bidirectional protein transitions between the diffusion-driven and motor-driven kinetic states," *Comput. Biol. Medicine*, pp. 758–764, 2013. [Online]. Available: https://doi.org/10.1016/j.compbiomed.2013.03.009

[11] B. Jennings and R. Stadler, "Resource management in clouds: Survey and research challenges," *J. Netw. Syst. Manag.*, vol. 23, no. 3, pp. 567–619, 2015. [Online]. Available: https://doi.org/10.1007/s10922-014-9307-7

[12] L. Bao, C. Q. Wu, X. Bu, N. Ren, and M. Shen, "Performance modeling and workflow scheduling of microservice-based applications in clouds," *IEEE Trans. Parallel Distributed Syst.*, vol. 30, no. 9, pp. 2101–2116, 2019. [Online]. Available: https://doi.org/10.1109/TPDS.2019.2901467

[13] L. Suresh, P. Bodík, I. Menache, M. Canini, and F. Ciucu, "Distributed resource management across process boundaries," in *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24-27, 2017*. ACM, 2017, pp. 611–623. [Online]. Available: https://doi.org/10.1145/3127479.3132020

[14] R. Bhattacharya, "Smart proxying for microservices." in *In Proceedings of the 20th International Middleware Conference Doctoral Symposium*. Association for Computing Machinery, 2019, p. 31–33. [Online]. Available: https://doi.org/10.1109/CCGrid49817.2020.00009

[15] "Github - aeraki-framework/aeraki." [EB/OL], https://github.com/aeraki-framework/aeraki Accessed 2022.

[16] "Github - slime-io/slime." [EB/OL], https://github.com/slime-io/slime Accessed 2022.

[17] X. Xie and S. S. Govardhan, "A service mesh-based load balancing and task scheduling system for deep learning applications," in *20th International Symposium on Cluster, Melbourne, Australia, May, 2020*. IEEE, 2020. [Online]. Available: https://doi.org/10.1109/CCGrid49817.2020.00009

[18] L. Larsson, W. Tärneberg, C. Klein, M. Kihl, and E. Elmroth, "Towards soft circuit breaking in service meshes via application-agnostic caching," *CoRR*, vol. abs/2104.02463, 2021. [Online]. Available: https://arxiv.org/abs/2104.02463

[19] J. Auriol, I. Boussaada, R. J. Shor, H. Mounier, and S. Niculescu, "Comparing advanced control strategies to eliminate stick-slip oscillations in drillstrings," *IEEE Access*, vol. 10 949–10 969, 2022. [Online]. Available: https://doi.org/10.1109/ACCESS.2022.3144644

[20] M. Xu and R. Buyya, "Brownout approach for adaptive management of resources and applications in cloud computing systems: A taxonomy and future directions," *ACM Comput. Surv.*, vol. 52, no. 1, pp. 8:1–8:27, 2019. [Online]. Available: https://doi.org/10.1145/3234151

[21] M. Rusek and J. Landmesser, "Time complexity of an distributed algorithm for load balancing of microservice-oriented applications in the cloud," *ITM Web of Conferences*, vol. 21, p. 00018 (8 pp.), 2018.

[22] "Envoyproxy. 2022. envoy - adaptive concurrency filter." [EB/OL], https://www.envoyproxy.io/docs/envoy/v1.17.1/configuration/ Accessed 2022.

[23] "Github - bookinfo sample. 2022." [EB/OL], https://github.com/istio/istio/tree/master/samples/bookinfo Accessed 2022.

[24] "Github - isotope." [EB/OL], https://github.com/istio/tools/tree/master/perf/load Accessed 2022.

[25] A. Leva and M. Maggio, "The pi+p controller structure and its tuning," *Journal of Process Control*, vol. 19, no. 9, pp. 1451–1457–8:27, 2009. [Online]. Available: https://doi.org/10.1145/3234151