Ripple: Large-Scale Service and Configuration Management in the Cloud

Shuping Ji
Institute of Software, Chinese Academy of Sciences
Beijing, China

Zhen Tang*†‡
Institute of Software, Chinese Academy of Sciences
Beijing, China

Wei Wang†§
Institute of Software, Chinese Academy of Sciences
Beijing, China

Hui Li†
Institute of Software, Chinese Academy of Sciences
Beijing, China

Jianguo Yao
Shanghai Jiao Tong University and Jinan Inspur Data Technology Co., Ltd
Shanghai, China

Hans-Arno Jacobsen
University of Toronto
Toronto, Canada

Abstract

Microservice architectures backed by container technology have been widely used in many real-world cloud-native applications. By enabling customers to manage their services and configurations in the cloud in a centralized, externalized, and dynamic manner, efficient service and configuration management plays a fundamental role in building cloud-native service-centric applications. The number of containers in cloud data centers continues to increase. For example, in the Alibaba Cloud, the number of containers reached hundreds of thousands by 2023 and is expected to reach several million soon. At this scale, existing service and configuration management solutions have limited efficiency, scalability and robustness. Other related approaches, such as message bus systems and publish/subscribe (pub/sub for short) systems, also do not work well for large-scale service and configuration management in the cloud, as their designs are more general purpose directed. To overcome these limitations, we design a system, called Ripple, that uniquely combines several existing and some novel features such as consistent hashing-based workload distribution, dynamic destination list-based and client-assisted message delivery, incremental update, and adaptive load balancing. Approaches exhibiting these features have not been well investigated in the domain of service and configuration management. We compare our proposed solution with existing academic and industrial approaches. The experiments show that our solution greatly outperforms its counterparts. For example, for the same workload, when Ripple is used, the average message delivery latency and network bandwidth consumption can be reduced by up to 77% and 93%, respectively.

*Zhen Tang (tangzhen12@otcaix.iscas.ac.cn) is the corresponding author.
†Affiliated with Key Laboratory of System Software at CAS, State Key Laboratory of Computer Science at Institute of Software at CAS, and University of CAS, Beijing, China. CAS is the abbreviation of Chinese Academy of Sciences.
‡Affiliated with University of CAS Nanjing College, Nanjing, China.
§Affiliated with Nanjing Institute of Software Technology, Nanjing, China.

MIDDLEWARE '24, December 2–6, 2024, Hong Kong, Hong Kong

ACM ISBN 979-8-4007-0623-3/24/12
https://doi.org/10.1145/3652892.3700777

CCS Concepts

• **Software and its engineering** → **Cloud computing**.

Keywords

service and configuration management, message delivery, publish subscribe, efficiency, scalability, robustness, dynamic destination list signature, incremental update, adaptive load balancing, microservice in the cloud

ACM Reference Format:
Shuping Ji, Zhen Tang, Wei Wang, Hui Li, Jianguo Yao, and Hans-Arno Jacobsen. 2024. Ripple: Large-Scale Service and Configuration Management in the Cloud. In *25th International Middleware Conference (MIDDLEWARE '24), December 2–6, 2024, Hong Kong, Hong Kong*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3652892.3700777

1 Introduction

Cloud computing [28, 43], microservice architectures [29] and container technology [4, 40] have been widely used in many real-world applications. Moreover, a new trend is to combine these technologies, i.e., to deploy many containers in the cloud to build numerous distributed microservice applications. This new deployment model has become very popular. For example, in the Alibaba Cloud, the number of deployed containers reached hundreds of thousands by 2023[1] and is expected to reach millions soon.

To manage this large number of containers, an efficient, scalable and robust service and configuration management system is needed, which would enable customers to manage their services and configurations in the cloud in a logically centralized, externalized, and dynamic manner. Typical service management tasks include service registration, discovery, and health checks. Configuration management permits administrators to dynamically change the configurations of their services online and helps quickly deliver the latest configurations to the corresponding containers. Different services can have different configurations, such as different cache sizes, database endpoints, and client white lists.

The increasing numbers of services and containers, as well as the diversity of configurations, pose new challenges for the underlying service and configuration management system. For example, in the

[1]https://www.alibabacloud.com/blog/4-step-method-for-large-scale-container-deployment_596928

Alibaba Cloud, the Nacos system [17] is used as the underlying service and configuration management system. During "Double 11" day[2], because of the high request load, even when the number of servers in Nacos is increased to several hundred, the system can still become overloaded, and fail to meet the message delivery QoS requirements. Moreover, the system cannot easily support more containers through the simple addition of more servers, which means that its scalability is limited. In addition, as we show in the following sections, other existing solutions also have different types of limitations in meeting the efficiency, scalability and robustness requirements for this use case. As a result, a new approach is needed.

Before presenting our solution, we carefully review the typical workflow of service and configuration management, determine the detailed technical requirements, and analyze the limitations of directly applying existing methods to solve this problem.

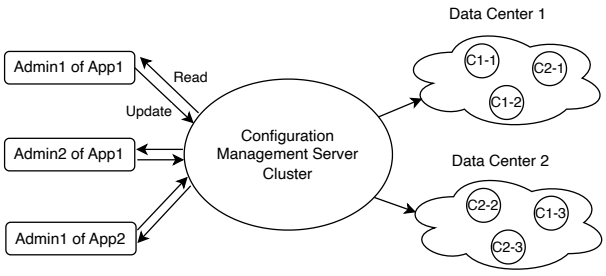

Figure 1: Configuration Management Example

Fig. 1 shows an example of configuration management. In this example, two applications are deployed in the cloud. The first application has three containers: *c1-1*, *c1-2*, and *c1-3*. The second application also has three containers: *c2-1*, *c2-2*, and *c2-3*. These containers are distributed in two data centers. There are three administrative clients for these two applications. An administrative client can query a configuration and make updates to the configuration. The server cluster is in charge of delivering the latest configurations to the interested containers. Containers of one application may be interested in the configurations of another application, such as its available servers' IP list. This usually occurs when two different applications have a dependency relationship.

The service and configuration management solutions need to meet the following requirements:

- **Data Consistency:** The system needs to provide both distributed storage and distributed message delivery functions. Moreover, the system needs to ensure that servers and clients always maintain the same configuration version and that configuration update conflicts can be correctly handled.
- **High Throughput:** The system should be able to support hundreds of thousands of containers and process thousands of updates per second. Moreover, by adding more servers, the system should be able to support more services and containers easily.
- **Low Latency:** The latest configurations need to be delivered to interested containers with low latency. For example, some applications have the service level agreement (SLA) requirement that a new version of the configuration be delivered to all interested containers within 3 seconds.
- **Fault Tolerance:** The system should be robust and needs to ensure that a new version of the configuration can always be successfully delivered to all its interested containers even in cases of failure.

Existing solutions in both industrial and academic communities have limitations in meeting these requirements. For example, for the current Nacos system [17] used in the Alibaba Cloud, a configuration update needs to be broadcast to all the servers in the system, and the servers need to directly deliver a message to each interested client. As a result, servers easily become bottlenecks and the system is not scalable. The Google Thialfi system [2] sends only a version number to clients and coalesces many updates into the most recent update. Then, the clients, which belong to different applications, connect to their corresponding servers to fetch the updates. This mechanism has high latency. Additionally, it is not appropriate for our scenario since the message delivery tasks cannot be offloaded to other servers. Another approaches are to directly utilize existing pub/sub systems. Although they often show good flexibility, they do not work well for our problem. They mainly provide best-effort, at-most-once delivery guarantee, which means additionally mechanism is needed to guarantee eventually or sequential data consistency. Also, they cannot meet all requirements for configuration management in large-scale clusters.

To meet all the above requirements and overcome the limitations of existing solutions, we first analyze the characteristics of service and configuration management. Then, on the basis of this analysis, we propose our solution called Ripple, which includes several new features and optimizations to improve performance, scalability and robustness. The key technical design components of Ripple are summarized as follows:

- **Workload Distribution:** According to the independence of applications, the range of update delivery will usually be limited inside an application, so we can use a small set of servers to handle requests for separate applications. We use a consistent hashing-based mechanism to distribute the workload to different servers. In this way, a single server needs to process only a portion of the messages in the system.
- **Message Delivery:** To achieve high flexibility, we use a dynamic destination list to route messages. On the basis of the relative stability of containers' interests, we reduce the cost of destination list management by using a destination list signature and a caching mechanism. Moreover, according to whether the containers belong to the same application and data center, we propose a client-assisted message delivery mechanism to reduce the workload on servers and improve the performance of message delivery.
- **Incremental Update:** The entire configuration of cloud services could be large, up to hundreds of megabytes. However, most updates change only a small part of the configuration. This means that bandwidth is often wasted in delivering the unchanged part of the configuration. On the basis of this

[2]From 2009, Alibaba began to use the "Double 11" day (November 11th) as a shopping festival lasting for 24 hours.

observation, we reduce the bandwidth consumption via an incremental update mechanism. This design also helps us achieve sequential consistency for incremental updates.
- **Adaptive Load Balancing:** We design a decentralized algorithm for dynamically balancing the workload on the servers and containers. The algorithm achieves low cost and high flexibility by utilizing our proposed workload distribution and message delivery mechanism.

The main contributions of this paper are the characteristic analysis of service and configuration management and the integration of different new features and optimizations of existing techniques to construct an overall efficient solution. The remainder of this paper is organized as follows. We first review related work in Section 2. In Section 3, we apply the pub/sub paradigm to the problem of service and configuration management. We then describe the design of Ripple. In Section 4, we describe Ripple's implementation in detail. We evaluate the performance of Ripple in Section 5 and conclude the paper in Section 6.

2 Related Work

In this section, we first review existing service and configuration management systems. Then, we discuss related distributed storage and message bus systems. Finally, we review existing pub/sub systems and consider why they do not work well if they are used directly for our problem.

2.1 Existing Service and Configuration Management Systems

There are several existing service and configuration management systems, such as Thialfi [2], Wormhole [39], Consul [20], etcd [16], Eureka [30] and Nacos [17]. However, these systems are either designed for specific business requirements or have scalability limitations for large-scale applications in the cloud. For example, Consul and etcd are based on the Raft [31] protocol, and their recommended number of servers is only three to seven. As a result, they do not work well for our large-scale scenario. Here, we select a few examples for illustration.

Thialfi [2] is Google's cloud notification service that enables applications to register their interest in a set of shared objects and receive notifications when those objects change. Thialfi is geographically distributed and highly reliable, even in cases of long disconnections. However, its clients are applications running in browsers or on end-users' mobile phones, laptops, and desktops, not applications within Google itself. Thialfi sends only the version number of the data to subscribers, and coalesces many updates into the most recent update. Then, the browser clients connect to the servers of different applications to fetch the updates. This means that Thialfi requires each application to have its own servers to store the latest configurations. This mechanism does not work for our scenario.

Wormhole [39] is Facebook's notification service developed for the scenario of geographically replicated datacenters. It is used to identify updates in different storage systems and transmit notifications to interested applications. Wormhole does not maintain brokers for buffering updates. Instead, it relies on data stores to provide reliable logs in the form of transaction logs. This is not appropriate for our use case, where the configuration updates are mainly from administrators. Moreover, Wormhole may suffer from high delivery latencies, such as latencies of several minutes. This is unacceptable for the scenario of service and configuration management.

Nacos [17] is Alibaba's service and configuration management system, which is specifically designed for container-based applications in the Alibaba Cloud. It has supported hundreds of thousands of containers. However, when the system scale increases, Nacos has limitations. First, every configuration update message needs to be broadcast to all the servers in the system. Second, the servers need to deliver each message directly to each interested client. As a result, servers easily become the bottleneck of the system.

2.2 Distributed Storage and Message Bus Systems

Distributed storage and message delivery are the two core functions for large-scale service and configuration management. In these two areas, there are many related works, such as Chord [41], DynamoDB [14], Pastry [37], Tapestry [44], and CAN [35] for distributed storage and TIBCO Rendezvous [21], AWS SQS [38], RabbitMQ [34], and Kafka [26] for message buses. Owing to space limitations, we select Chord, DynamoDB and Kafka to as representative illustrations.

Chord is a structured P2P system. It uses a distributed hash table (DHT) to construct the overlay network and provides efficient key-to-node lookups. Consistent hashing is utilized for the mapping. In an N-node system, each node maintains information about only $O(\log N)$ other nodes and resolves all lookups via $O(\log N)$ messages to other nodes. As a result, Chord is efficient and can support frequent node arrivals and departures. DynamoDB is Amazon's highly available and scalable distributed key-value data store. In DynamoDB, data are partitioned and replicated by consistent hashing, and consistency is facilitated by object versioning. DynamoDB employs a gossip-based distributed failure detection and membership protocol. Both Chord and DynamoDB are efficient distributed storage systems. Our proposed solution shares a similar design in terms of workload distribution. The difference is that our proposed solution focuses more on efficient message delivery. For example, we rely on clients to know all the servers and resolve lookups via a single message.

Kafka [26] is LinkedIn's message bus. It is an open-source system maintained by Apache. It is specifically designed for collecting and delivering high volumes of log data with low latency. It persists log data on disks and uses ZooKeeper [22] to keep track of the numbers of log messages that particular subscribers have consumed. Kafka can lose messages in the case of broker failure. Thus, additional fault tolerant mechanism is needed to guarantee data consistency when broker fails. Moreover, Kafka utilizes a *pull* model rather than a *push* model. As a result, messages may not be delivered to clients in time, which is not suitable for our scenario. Finally, Kafka maintains a message stream for each client without aggregation, which is inefficient for our scenario with a very large number of clients, i.e., containers.

2.3 Pub/Sub Systems

Pub/sub is a popular data delivery paradigm that has been widely used in many applications [18, 23]. Existing pub/sub solutions could be applied to solve the problem of efficient message delivery for service and configuration management. However, as we will present later, this method does not work well because of various limitations.

Fig. 2 shows the typical architecture of the pub/sub paradigm, which is widely adopted by pub/sub systems, such as SIENA [8], Gryphon [3], JEDI [13], Herald [5], and PADRES [24]. As shown in this figure, the pub/sub paradigm has three roles: *broker*, *subscriber* and *publisher*. Numerous brokers comprise a broker network to provide message delivery services. A subscriber can issue a *subscription* message to the broker cluster through an access point to express interest. When a publisher issues an *event*, the broker cluster delivers that event to all the interested subscribers.

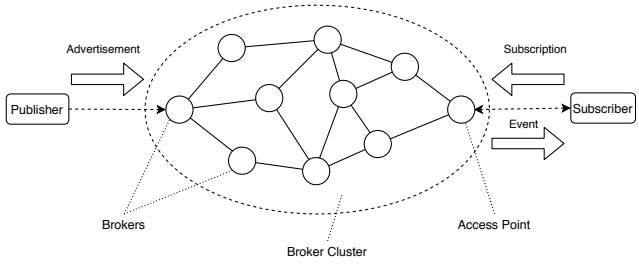

Figure 2: Pub/Sub Paradigm

The existing message routing protocols in pub/sub can be categorized into five classes: event flooding (EF) [11, 12], multicast-based routing (MBR) [1, 15, 32, 36, 42], filter-based routing (FBR) [8, 13, 24, 27], destination-based routing (DBR) [3, 6, 7, 25], and DHT-based routing [9, 10, 19, 33]. As we show individually, these message delivery methods do not work directly for the scenario of large-scale service and configuration management in the cloud owing to various limitations.

Event Flooding: In EF, each event originating at a publisher is first routed to all brokers and then delivered to interested subscribers. This routing protocol is simple and stateless. However, it has the shortcoming that a broker receives all the incoming events. The throughput of the whole system is limited by the power of a single machine and is thus not scalable. The current Nacos system [17] utilizes this type of routing mechanism.

Multicast-Based Event Routing: In MBR, the event space is partitioned into many disjoint multicast groups, for each of which a multicast tree is built. When an event is issued, it is first mapped to an appropriate group and then multicast on the corresponding spanning tree. MBR does not require a broker to receive all the events, and the routing accuracy is improved. However, this solution requires many groups in the scenario of large-scale service and configuration management in the cloud. The group maintenance cost could be too high.

Filter-Based Routing: FBR requires a publisher to broadcast an *advertisement* message before issuing a stream of events. The advertisement message indicates the legal content space of the events. Subscriptions are delivered to the advertisement source broker in the reverse direction to construct a routing tree. This mechanism does not work for the scenario of service and configuration management. This is because an administrator, i.e., publisher, is usually not statically connected to a broker and often issues only a few configuration updates after it connects to the cluster. If we introduce an extra advertisement message, a high cost will be incurred because subscriptions will be routed in reverse to the publisher's connected broker. Moreover, routing information is stored by brokers. When a broker crashes, it is difficult to guarantee robust message delivery.

Destination-Based Routing: In DBR, events are first evaluated against subscriptions to obtain the list of destination subscribers and then routed on the basis of the destination lists. This method is flexible and robust. However, it introduces the extra cost of obtaining the destination list for every event. The DRP [7] and MERC [25] algorithms aim alleviate this extra cost by either using a fixed-sized bit vector or dividing the broker overlay into separate clusters. However, these methods either limit the scale of the server cluster or isolate the servers; thus, they are not appropriate for our application in the cloud scenario.

DHT-Based Routing DHT-based pub/sub systems, such as Scribe [10], SplitStream [9], Meghdoot [19] and Hermes [33], rely on existing DHT solutions, such as Chord [41], Pastry [37] and CAN [35], to map subscriptions and events to brokers. As a result, they inherit the characteristics of DHT. In the scenario of a large-scale service and configuration manager, the servers are usually stable, whereas the clients often change dynamically. Therefore, DHT-based routing is not the most suitable method for our scenario. Moreover, DHT relies on servers to send messages directly to all clients, which is also not as efficient as our designed message delivery mechanism.

3 System Design

The pub/sub paradigm has been studied for decades, but investigations in the area of service and configuration management have been limited. To exploit the techniques of pub/sub, we first map the service and configuration management application into the semantics of pub/sub. Then, we analyze the characteristics of this problem in the cloud environment. Finally, we describe our system design, which includes consistent hashing-based workload distribution, dynamic destination list-based and client-assisted message delivery, incremental update, and adaptive load balancing.

Table 1: Mapping for Configuration Management

Component	Role	Operation	Message
Container	Subscriber	Container Interest	Subscription
Administrator	Publisher	Config Update	Event
Config Server	Broker	Container Leave	Un-Sub

3.1 Semantic Mapping

There are three roles in the pub/sub paradigm: *publisher*, *subscriber* and *broker*. There are three types of messages: *subscription*, *event* and *unsubscription*. To adopt the pub/sub paradigm, we first map the components and operations in the service and configuration management system into the roles and messages of pub/sub.

Table 1 shows our designed pub/sub semantic mapping for configuration management. We map containers to subscribers. A container might be interested in different configurations. The interests of a container are mapped to subscription messages. A container can express its interests to the configuration server cluster. A configuration server assumes the broker role in the pub/sub paradigm. The server cluster provides configuration management for many applications. Each application has many containers in the cloud, which work together to provide different types of cloud services to end users or other applications. Each application often has many configurations. The configurations of an application are managed manually by administrators. Therefore, administrative clients can be mapped to publishers. A configuration update message is mapped to an event. The server cluster is in charge of delivering each message to all its interested containers in the system.

We also map service management into the semantics of the pub/sub paradigm: the service registry is mapped to publisher creation; monitoring clients are mapped to subscribers; containers are mapped to publishers; and container status updates are mapped to events. For simplicity, in the following sections, we use configuration management to explain our system design.

3.2 Characteristic Analysis

Directly applying existing pub/sub solutions to the service and configuration management problem does not work well. One important reason is that the characteristics of the service and configuration problem as well as the cloud environment are not considered. For example, a large-scale cloud application may have thousands of containers, i.e., clients. Containers of different applications often have diverse interests. Therefore, it could be inefficient to permit a client to connect with any server and then rely on the selected server to deliver all the messages that the client is interested in. Here, we analyze the characteristics of large-scale service and configuration management in the cloud environment to guide the design of our solution:

- **Inter-application Isolation:** Each application has its own containers, configurations and administrators. Unless two applications have dependency relationships, containers of different applications are often not aware of each other and do not share configuration interests.
- **Intra-application Connection:** Containers of the same application are usually deployed in a single data center or a few data centers for robustness. Containers of the same application are usually aware of each other, share similar interests and can efficiently communicate with each other.
- **Container Stability:** The distributions of containers and their interests are relatively stable. Most of the containers of an application are not dynamically added or removed. Update destinations of a configuration, i.e., containers that are interested in it, usually do not change frequently.
- **Content Duplication:** The configurations of an application usually change slightly. There could be much content duplication between two continuous configuration versions, which means that it is not only expensive but also unnecessary to deliver the entire configuration message repeatedly when the configuration is updated.

3.3 Workload Distribution

In the existing pub/sub paradigm, a client can select any broker as its access point. Then, that access broker is in charge of delivering all the events that the client is interested in. This mechanism simplifies the function of clients. However, the overhead at the broker side is high: a single broker might have to serve many clients with diverse interests, and a single event may need to be routed through many brokers. As a result, brokers can easily become bottlenecks, especially when the system's scale is large and the workload is high.

As presented in the previous section, service and configuration management features good interapplication isolation and intraapplication connectivity. This means that we can route messages for different applications relatively independently. We adopt a consistent hashing-based workload distribution mechanism.

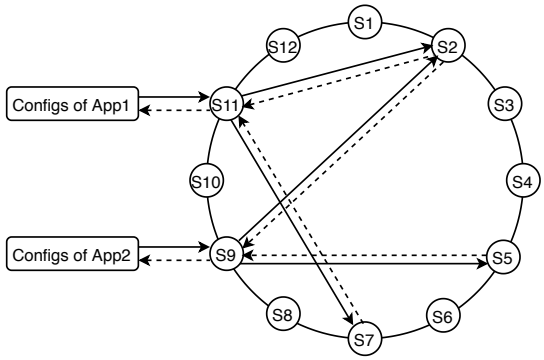

Figure 3: Configuration to Server Mapping

Here, we use a simple example to illustrate the basic ideas. As shown in Fig. 3, there are two applications, each of which has numerous configurations. Rather than allowing each administrative client and container to connect to any server, we run a consistent hashing algorithm on the administrative clients and containers to select a specific server on the basis of the corresponding application's name. In this way, we cluster the publishers and subscribers with similar interests. When an event is issued to the system, it will no longer be routed to many servers. Several methods have been proposed in pub/sub for clustering and migrating clients' interests on the broker side. These methods seem to be different from our client-based method and are not as efficient at reducing the server cost.

In a large-scale service and configuration management system in the cloud, servers may fail for different reasons. For critical cloud applications, we cannot tolerate any system failures. To improve robustness, we permit an application to configure a few backup servers. In the example shown in Fig. 3, each application has one primary server and two backup servers. A backup server receives all the update messages for its corresponding application. Only when an update message is delivered to the primary server and all the backup servers is that message considered successfully delivered to the server cluster.

3.4 Message Delivery

In the cloud, there could be many large-scale services, each of which could have thousands of containers. Directly delivering messages to such number of containers would be expensive for the servers, and this could limit the system's throughput and message delivery latency. To overcome this limitation, we propose adopting a container-assisted message delivery mechanism. Rather than relying only on servers to directly deliver messages to the containers, we use the containers as proxies to deliver messages to other containers that belong to the same application and are in the same data center. This design is based on the observation that such containers can usually communicate with each other and that containers of the same application often have many shared interests.

This client-assisted message delivery mechanism greatly helps reduce the overheads of servers. However, it also introduces extra overhead to the containers. To ensure that the overhead and effects on containers are small, we design a flexible, dynamic destination list-based message delivery method. For each message to be routed, the primary server dynamically calculates its routing paths on the basis of a load balancing algorithm and identifies a few proxy containers. The server sends the message only to the proxy containers. A destination list is attached to the message for each proxy container. A proxy container simply routes the message on the basis of the destination list and does not perform any complex computations. As a result, the extra overhead is limited. To reduce the overhead further, we design a dynamic load balancing mechanism, as presented in Section 3.6.

This dynamic destination list-based message delivery method is flexible. However, it also introduces extra network costs for the destination list. Is there a way to eliminate these extra costs? By observing container stability in service and configuration management systems, we conclude that the delivery paths of messages of the same application do not change frequently. On the basis of this characteristic, we propose generating a hashing ID for each destination list. The servers and proxy containers cache and map from the hashing IDs to the destination lists. As a result, only a short hashing ID needs to be attached.

Many applications require an update message to be successfully delivered to all the interested containers. In the face of failures, i.e., when the system fails to deliver an update massage to some containers, the administrative client may require the system to report the failed container list. To meet this requirement, we simply add a low-overhead destination list-based acknowledgment mechanism: if a message is successfully delivered to all its downstream containers, an acknowledgment message with the corresponding hashing ID attached is then returned. If there is a failure, the failed container information is attached. In this way, we can easily identify the containers that fail to receive a message on the primary server side at a low cost. To reattempt delivery of a message, the primary server can calculate a new routing path only for delivering that message to the failed containers.

Thus far, we have described the message delivery designs in our system as well as the reasons for each design component. In short, on the basis of the characteristics of service and configuration management in the cloud environment, we propose a flexible *client-assisted, dynamic destination list-based and acknowledgment-based* message delivery mechanism. The design can help reduce server overhead, improve system throughput, reduce message delivery latency, improve flexibility, reduce the network cost, and improve system robustness.

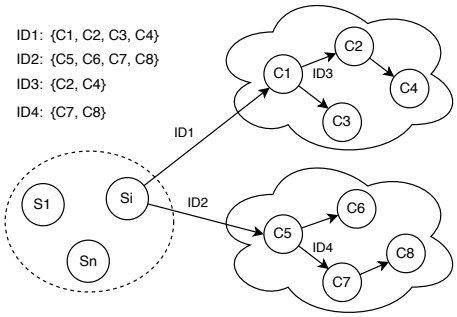

Figure 4: Message Delivery Example

Fig. 4 illustrates the message delivery process in our system. In this figure, for an event, interested containers *c1* to *c4* are located in a data center, whereas interested containers *c5* to *c8* are located in another data center. The server S_i sends the event to *c1* and *c5* only, with the destination list hashing ID, *ID1* and *ID2*, respectively, attached. Then, the containers *c1* and *c5* forward the event to the downstream containers in the same data center. In this example, *ID3* and *ID4* are used by containers *c1* and *c5* for the next level of message delivery. The whole message delivery path is a dynamically constructed spanning tree.

3.5 Incremental Update

Unlike the typical pub/sub systems in which different delivered messages are usually independent, for the scenario of service and configuration management, different messages may contain substantial amounts of duplicate content. For example, many configurations could be large, even up to hundreds of megabytes. However, the difference between different versions of a configuration is usually small. To reduce the cost, we propose adopting an incremental update mechanism, with which efficient sequential consistency can be achieved.

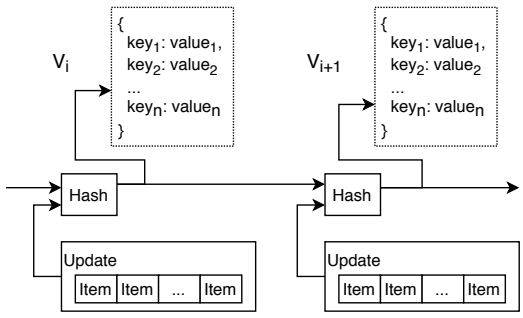

Figure 5: Incremental Update Example

As presented in Section 3.2, much content duplication often occurs between different versions of the same configuration. Rather

than delivering the whole message repeatedly, we deliver only the update operations that can be applied to the previous version. At each container, these update operations are executed on the previous version to generate the next version. To ensure that all incremental update operations are executed on the correct previous versions, we attach the corresponding previous version's hashing ID to each incremental update message. Note that neither the containers nor the servers need to maintain the update histories. Instead, they need to keep only the latest version as well as a small number of unapplied updates. This means that no extra memory cost is incurred. When the latest version of the configuration is accidentally lost by a container, the container simply requests the latest version from that configuration's corresponding server. When the arrival sequences of several updates are not in order, the containers will reorder those updates and replay them one by one to generate the correct latest configuration. This means that our system supports sequential consistency for incremental updates in a simple way. Notably, although we permit multiple administrators to update the same configuration together, the updates are be ordered at that configuration's corresponding server to eliminate potential concurrent update conflicts.

Fig. 5 briefly illustrates the incremental update strategy. In this figure, the same configuration is updated twice, generating two versions, V_i and V_{i+1}. For each version a hashing ID is generated on the basis of the previous version and the new updates. Because the sizes of updates and hashing IDs are usually much smaller than the size of a full configuration, the message delivery cost can be greatly reduced.

3.6 Adaptive Load Balancing

Through load balancing, we aim to achieve three major goals: (1) the workload is evenly distributed; (2) the system can automatically adjust to dynamic workload changes; and (3) the extra cost of load balancing should be very small. To achieve these goals, we design a simple but effective adaptive load balancing mechanism. It uses the following process:

- **Workload Score:** Each server and container periodically collects its workload, resource utilization information and QoS information. On this basis, we calculate a workload score.
- **Score Propagation:** Instead of designing specific communication messages to propagate each server or container's workload score, we utilize the existing acknowledgment message. When a server or container sends an acknowledgment message to its upstream node, it will periodically attach its current workload score to the acknowledgment message. Thus, we do not need extra messages to send the workload scores. Moreover, this guarantees that workload scores will only be transferred to the related upstream nodes.
- **Load Balancing:** Each upstream node collects its downstream nodes' workload scores. When it has a new message to deliver, the server can dynamically calculate the current message's delivery path on the basis of those nodes' latest workload scores.

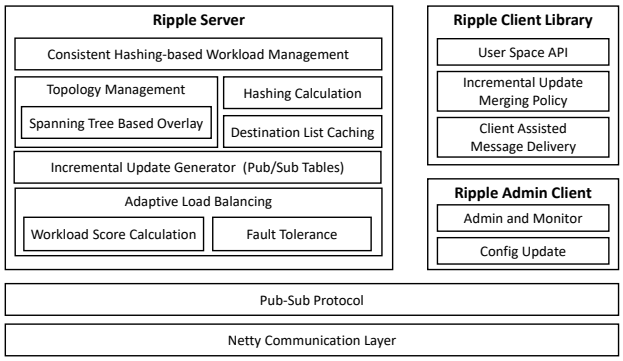

Figure 6: Major Components of Ripple

4 Implementation

We implement our proposed solution as a project called Ripple[3]. Ripple is implemented in Java with approximately 17,000 lines of code. It utilizes Netty[4], a widely used network framework in Java for reliable and high-performance communication. Ripple implements all our proposed designs presented in the above section. In this section, we describe the major components and implementation details of Ripple.

4.1 Major Components

As shown in Fig. 6, Ripple has three major components: the Ripple client library for the container, the Ripple administrative client and the Ripple server. The container client library provides several user space APIs, such as *initialize*, *subscribe* and *receive*. Developers need to modify each containerized application only slightly to invoke the user space APIs of Ripple's container client library. Additionally, the client library can process incremental updates to generate new versions of configurations. Ripple clients act as the message transfer proxies for client-assisted message delivery. The administrative client is used by administrators to update configurations for different applications and manage the Ripple server cluster. Regarding the Ripple server, its subcomponents correspond to our design of consistent hashing-based workload distribution, message delivery, incremental update and adaptive load balancing. The Ripple server, client library and administrative client share the same Netty communication layer and pub/sub protocol layer.

4.2 Workload Distribution

As presented in Section 3.3, Ripple adopts the consistent hashing-based workload distribution mechanism. The hashing algorithm that we use is *CRC32*, which is selected because hashing is a high-frequency operation and CRC32 generates faster than other hashing algorithms such as *SHA-1*. Additionaly, collisions will not lead to any side effects in this scenario. The inputs for the hashing algorithm are the servers' IP addresses and the applications' names. To improve the robustness of Ripple, each application is mapped to N servers.

[3]https://github.com/ISCAS-SSG/Ripple
[4]https://netty.io/

The first server is selected on the basis of the consistent hashing algorithm. It acts as the primary server for that application. The remaining servers are selected as the first server's next adjacent *N-1* servers in the hashing circle. These servers act as the backup servers for that application. In Ripple, each server maintains a subscription table and a publication table. The subscription table contains the subscriptions, i.e., interests of containers, and the publication table contains each application's configuration or service information. The basic communication workflow of Ripple is as follows:

(1) The administrator of an application can connect to that application's corresponding primary server to register new configurations or make updates to existing configurations. The configurations are stored in the publication table.
(2) When a container is started, it will connect to the servers corresponding to its interested applications and issue the subscriptions.
(3) When a configuration update message is delivered to a server from an administrative client, the server will match it against the subscription table to identify the interested containers. Then, the update message is delivered to the backup servers and the interested containers.
(4) Once the update message is delivered to all the backup servers and the interested containers, an *ACK* message is sent to the administrative client. In the face of delivery failures, the primary server can automatically activate the message delivery retrying mechanism. If further failures occur, the server will report to the administrative client which specific interested containers failed to receive the update.

4.3 Message Delivery

Ripple's core function is to efficiently deliver messages to many containers. As presented in Section 3.4, Ripple adopts a dynamic destination list-based and client-assisted message delivery method. Here, we describe its implementation details in terms of the following five features:

Client-assisted message delivery: When a Ripple server receives a configuration update message, it first identifies all the containers that subscribed to this configuration. Then, it splits those containers into groups on the basis of their application and data center. For each group, if the number of containers is less than a certain threshold, the Ripple server simply selects the container with the smallest workload as the message delivery proxy node, otherwise, it selects more than one container. The threshold can be set to a specific value such as 60 to ensure that the delivery path is short and that the number of messages to be transferred by a container is not large. A proxy container's received message includes its assigned target containers. The overall delivery path for a message is a spanning tree rooted at the primary server. The spanning tree usually has only 2 to 3 layers, as this is sufficient to support thousands of containers for an application.

Destination list hashing ID: To reduce the overhead for the extra destination list, we calculate the destination list's hashing ID and simply use this ID to represent the destination list. Specifically, we combine the IP address and port of each container in the destination list to build a string. Then, we calculate the *SHA-1* value of the string as the ID of the destination list. We use SHA-1 rather than CRC32 as collisions will lead to incorrect destination list. Upon delivering a message, only the destination list ID is attached to the message, rather than the whole destination list. The mapping from destination list IDs to destination lists is cached in the containers. We use the LRU algorithm for caching. When a destination list ID is not successfully hit in the cache, related information can be requested from the current node's upstream node.

Acknowledgment and retry: For robust message delivery, we support a fault tolerance mechanism that is based on *Acknowledgment and retry*. When a node successfully delivers a message to its downstream destinations, that node sends an acknowledgment message to its upstream node. If message delivery to some destination nodes fails, the failed nodes' addresses are attached to the corresponding acknowledgment message. The Netty-based timeout mechanism is used to detect a failure. In this way, the primary server knows which specific containers failed to receive the message and initiates the retry process by using a different proxy container and routing path. When the number of retries reaches a threshold, the primary server reports the failure information in the acknowledgment message to the administrative client.

Isolation and security: Client-assisted message delivery is a special feature of Ripple. To eliminate potential negative impacts, we provide isolation and security guarantees. Ripple implements *application plus data center* level isolation. Specifically, a container A can help transfer a message to another container B only when (1) they belong to the same application, (2) they are in the same data center and (3) a Ripple server attaches B to the destination list of the message to A. For special applications that do not permit their servers, i.e., containers, to communicate with each other, Ripple also allows users to disable the client-assisted delivery mechanism.

4.4 Incremental Update

The incremental update starts from administrators' inputs. Configurations are usually written in plain text format, such as XML, JSON or key-value pairs. When an administrator submits an update message, it is transferred via various atomic update operations, including *add, delete and update*. The primary server executes these operations on its locally stored version of the configuration to generate a new version. The new version's SHA-1 hashing ID is calculated. The primary server uses the previous version's hashing ID, the update operations and the latest version's hashing ID to construct an update message. When the update message is received by a container or a backup server, it will execute the update operations on its stored version of the configuration only if the hashing IDs match. In this way, we can ensure that even when update messages arrive in the wrong order, the latest configuration can be generated correctly. Therefore, Ripple ensures sequential consistency. In case of primary server failure, the clients, i.e., containers, automatically switch to the first backup server.

4.5 Adaptive Load Balancing

As described in Section 3.6, Ripple relies on a workload score to support adaptive load balancing. The workload score is calculated on the basis of the workload, the past message delivery latency and the available resources. When the workload is high, the past message delivery latency is higher and the amount of available

resources is lower, the workload score is higher. Specifically, the formula for calculating the workload score is as follows:

$$workload\ score = \frac{N_{req} \cdot P_{avg} \cdot L_{avg}}{F_{cpu} \cdot F_{mem} \cdot F_{net}}$$

In the formula above, N_{req}, P_{avg} and L_{avg} represent the number of processed update messages, the average message payload size and the average message processing latency, respectively, and F_{cpu}, F_{mem} and F_{net} represent available CPU, memory and network resources, respectively.

5 Evaluation

Ripple aims to provide efficient, scalable and robust large-scale service and configuration management. Regarding performance, the QPS and event delivery latency are the two key evaluation metrics that are of interest to us. A good solution should be able to provide high QPS and low event delivery latencies under different environments and workloads. For comparison with Ripple, we use Nacos [17] and Padres [24] as baselines. Each baseline is a representative solution; moreover, significant changes are not needed to provide the necessary functions for service and configuration management. Nacos is a state-of-the-art solution that was specifically designed for service and configuration management. Padres is a representative pub/sub system. Although pub/sub does not have distributed storage by default, we apply various changes and extensions to Padres to support this. We use both synthesized and real-world workloads for the experiments, as we show below.

5.1 Setups of Synthesized Experiments

The performances of different algorithms can change dynamically when the environment and workload vary. For an intuitive comparison, we first use different synthesized experiment setups to run the evaluation. The related workloads are simplified from Alibaba's real-world workload.

As shown in Table 2, we consider a variety of controlled experimental conditions: number of servers, number of clients, system QPS, average payload size and incremental event update ratio. With these configurations, we can generate different combinations of experimental setups. The default configuration for our experiments is as follows: 10 servers, 500 clients, 1,000 QPS, 10 KB average payload size, and an 80% incremental update ratio.

Table 2: Experiment Setup Configurations

Server Number	1, 5, **10**, 15, 20
Client Number	50, 250, **500**, 2500, 5000
System QPS	100, 500, **1000**, 1500, 2000, 2500
Average Payload Size	1KB, 5KB, **10KB**, 50KB, 100KB
Incremental Update Ratio	0%, 10%, 20%, ..., **80%**, 90%,100%

We evaluate Ripple in a cluster consisting of numerous virtual machines as the server nodes. Each server node has 2 virtual CPUs, 4 GB of memory and 40 GB of NVMe SSD storage. We also use virtual machines as workload generators to deploy client nodes. Each client node has 1 virtual CPU, 4 GB of memory and 40 GB of NVMe SSD storage. We limit the memory usage of JVM to 2 GB.

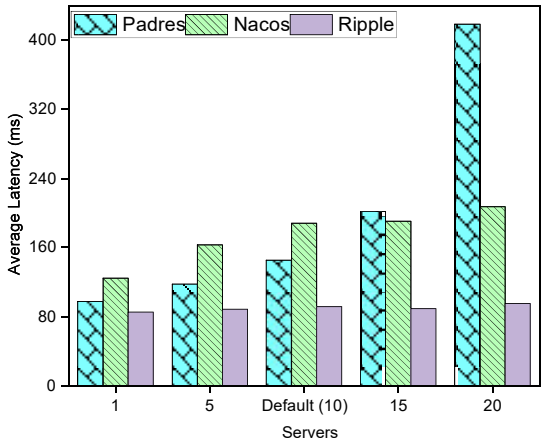

Figure 7: Changing the Number of Servers

5.2 Comparison with Related Work

In this section, we present the detailed experimental results for Ripple, Nacos, and Padres under the different controlled experimental conditions. Additionally, we report the system overheads of these algorithms under the same workload.

5.2.1 Number of Servers. As shown in Fig. 7, when the number of servers in the system increases, the latency changes are different for different systems. For Nacos and Padres, the event delivery latency increases with the number of servers. The major reason is that these algorithms either route messages to all the servers in the cluster or route messages on the basis of a reverse tree. When there are more servers, the number of nodes on the message delivery path increases. This increases the latency. However, for Ripple, the latency does not obviously change when the number of servers in the system increases. This is because Ripple utilizes a consistent hashing-based workload distribution, and a message usually needs only a single server for delivery. When there are more servers, the system can handle a higher workload without obviously affecting the event delivery latency.

5.2.2 Number of Clients. As shown in Fig. 8, when the number of clients increases, the event delivery latencies of all algorithms increases. This is because a greater number of clients means that a message needs to be delivered to more nodes. However, overall, the latency increase is more obvious for Nacos and Padres. Nacos cannot pass the test when the total number of clients increases to 2,500. In contrast, Ripple's event delivery latency does not obviously change when the number of clients is equal to or less than 500. The event delivery latency of Ripple starts to obviously increase when the number of clients increases to 2,500.

5.2.3 Overall QPS. Here, we evaluate the relationship between the system's overall QPS and the event delivery latency. As shown in Fig. 9, increasing the QPS results in higher event delivery latency for all algorithms, which is expected. However, under different QPS workloads, Ripple always outperforms Nacos and Padres.

5.2.4 Average Payload. When the average payload size increases, the system needs more network bandwidth to transfer the data.

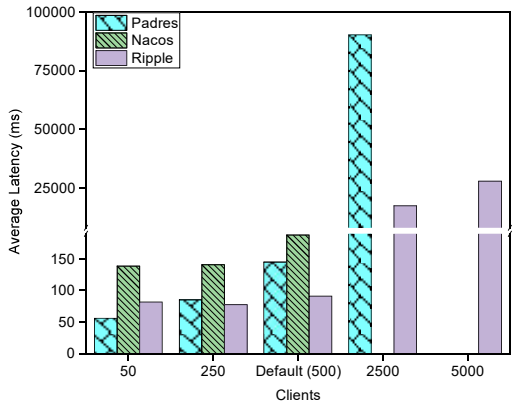

Figure 8: Changing the Number of Clients

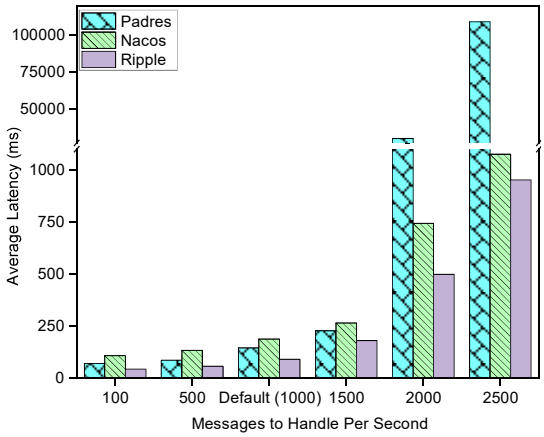

Figure 9: Changing the System's QPS

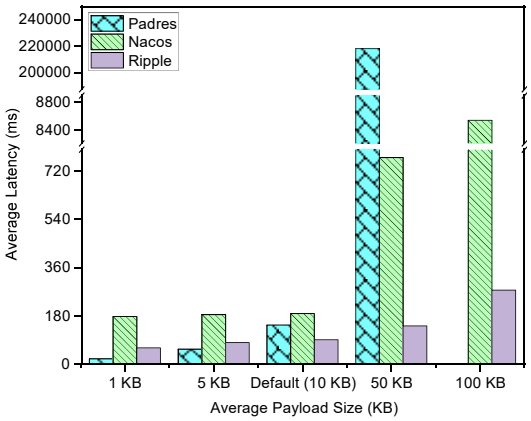

Figure 10: Changing the Average Payload Size

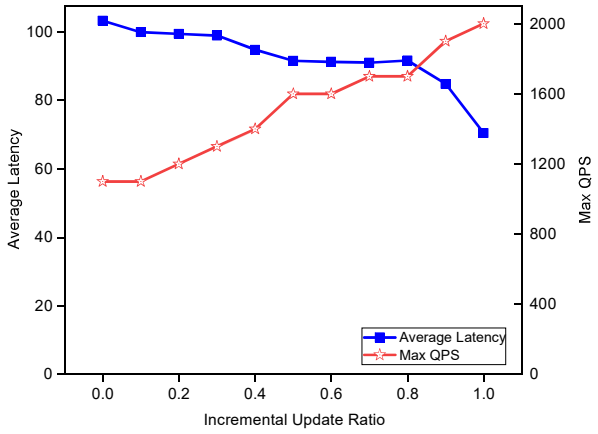

Figure 11: Changing the Incremental Update Ratio

Fig. 10 shows the algorithms' event delivery latencies as the average payload size increases. As expected, a larger payload results in a higher event delivery latency. However, Ripple has a smaller event delivery latency increase. This is because Ripple uses less network bandwidth to handle the same workload. As a result, Ripple can support a high average payload size.

5.2.5 Incremental Update Ratio. Incremental update is very useful for situations in which high ratios of duplication may occur. Configuration management is a typical workload with a high ratio of duplication. As Nacos and Padres do not have mechanisms for handling the incremental update workload, their performance will not change with the incremental update ratio. Therefore, we need to evaluate only Ripple's performance under different incremental update ratios.

As shown in Fig. 11, when the incremental update ratio is higher, the message delivery latency of Ripple is lower. For example, when the incremental update ratio is 0, the average message delivery latency is approximately 103 ms, whereas when the incremental update ratio is 100%, the average message delivery latency decreases to 70 ms. This is because Ripple needs to transfer only the updated part of a configuration rather than the whole latest configuration. Thus, the network overhead is obviously reduced.

Table 3: System Overhead Comparison

	Nacos	Padres	Ripple
CPU Usage (%)	41.42	56.37	28.57
Memory Usage (MiB)	1255.02	1607.61	894.59
Network Cost (KiB/s)	423.69	1837.21	123.82

5.3 System Overhead

Latency and QPS measure the performance of algorithms from the perspective of clients. On the server cluster side, we care about resource utilization. A better algorithm uses fewer resources to handle the same workload. Therefore, we report the CPU, memory and network usage of Ripple, Nacos and Padres under the same default workload.

As shown in Table 3, Ripple's CPU, memory and network overheads are all obviously smaller than those of Nacos and Padres. Compared with Nacos, Ripple reduces the CPU, memory and network usage by 31%, 29% and 71%, respectively. Compared with

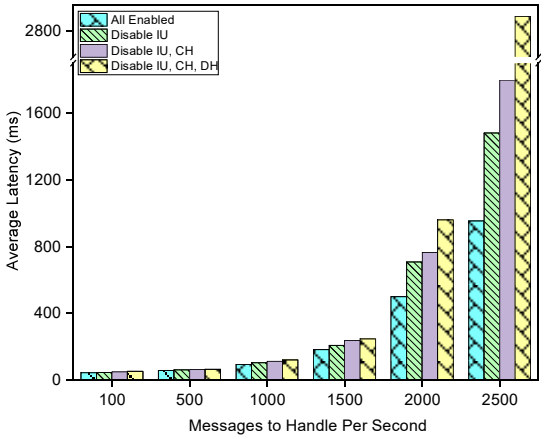

Figure 12: Self-Comparison Experiment on Ripple

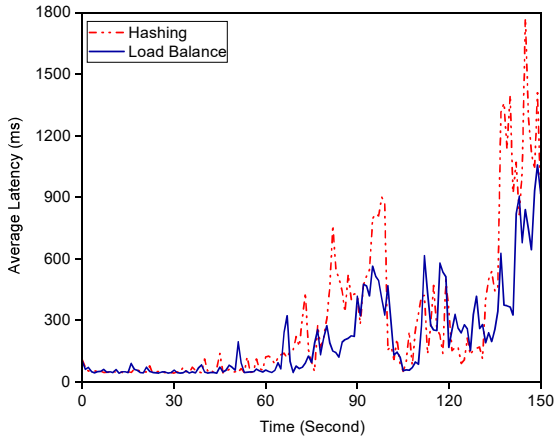

Figure 13: Adaptive Load Balancing Experiment

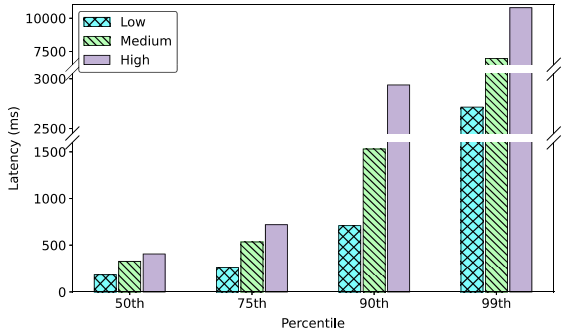

Figure 14: Real-world Experiment

Padres, Ripple reduces CPU, memory and network usage by 49%, 56% and 93%, respectively.

5.4 Self-Comparison

As presented in Section 3, Ripple includes four major design components: consistent hashing-based workload distribution, dynamic destination list-based message delivery, incremental update and adaptive load balancing. Among them, the first three design components aim to improve performance. The experiments presented in the above section show the effectiveness of utilizing these three design components together. However, they do not show the effects of each design. Therefore, here, we conduct a group of self-comparison experiments.

As shown in Fig. 12, given the same QPS, Ripple has the lowest event delivery latency when all three design components are enabled. The differences are more obvious when the QPS is higher. For example, when the QPS is 100, the latency increases by approximately 5%, 11% and 20% when the consistent hashing-based workload distribution, dynamic destination list-based message delivery and incremental update are disabled, respectively. In contrast, when the QPS is 2,500, the latency increases by 55%, 88%, and 200%, respectively.

5.5 Adaptive Load Balancing

Adaptive load balancing for Ripple is designed to support an unbalanced and dynamically changing workload, which is very common in real-world applications. To measure the effects of Ripple's adaptive load balancing design, in this group of experiments, we change the workload distribution dynamically while monitoring the system's performance metrics.

Fig. 13 compares the cases in which Ripple's load balancing feature is enabled and disabled. In this experiment, we control the system QPS as follows: from 0 s to 30 s, the QPS is set to 500; from 31 s to 60 s, the QPS is changed to 1,000; from 61 s to 90 s, the QPS is changed to 2,000; from 91 s to 120 s, the QPS is changed back to 500; and from 121 s to 150 s, the QPS is changed to 1,500. Given these time ranges, when adaptive load balancing is enabled, Ripple has better performance: on average, adaptive load balancing helps reduce the event delivery latency from 287.73 ms to 208.21 ms.

5.6 Real-World Workload

As there are no public benchmark workloads for large-scale service and configuration management, we mimic Alibaba's real-world workload during a specific Double 11 day in previous years. For this workload, the number of configurations is 3 M, the number of messages to be delivered is 1,801 M, the number of Nacos servers is 363, the average QPS is 139 K and the peak QPS is 381 K. Due to cost limitations, we cannot setup a test cluster of this scale. Given this scale of the workload, when we use our affordable tens of servers to run the evaluations, Nacos and Padres become overloaded; thus, here, we only report the performance metrics of Ripple.

In this experiment, we set up a cluster containing 60 servers. Each server is configured to utilize 8 CPUs, 32 GB of memory and 128 GB of NMVe SSD storage. We record the average message delivery latency (50%, 70%, 90% and 99% percentiles) when we set the QPS to 26 K, 83K and 139 K. As shown in Fig. 14, the system is not overloaded even when we increase the QPS to 139 K. The average message delivery latency is less than 300 ms. This group of experiments proves that Ripple is efficient at handling large-scale real-world workloads with a limited number of machines.

6 Conclusion

We propose an efficient and scalable solution for large-scale service and configuration management in the cloud, called Ripple. It utilizes the pub/sub paradigm and related techniques. However, unlike existing pub/sub systems, we propose several design features and optimizations on the basis of the characteristics of our application and the cloud environment, including consistent hashing-based workload distribution, dynamic destination list-based and client-assisted message delivery, incremental update, and adaptive load balancing. Ripple integrates several existing practical techniques with our newly proposed novel mechanisms to provide an efficient solution. The experiments show that our solution significantly improves the system throughput, reduces the message delivery latency, saves hardware resources, and achieves good scalability, efficiency and robustness. In addition, it clearly outperforms existing solutions.

Acknowledgments

This work is supported by the Alibaba Group through the Alibaba Innovative Research (AIR) Program (No. 1223), the Youth Innovation Promotion Association, Chinese Academy of Sciences (Grant No. 2023118), Major Project of ISCAS (ISCAS-ZD-202302), and the Guangdong Power Grid Limited Liability Company under Project 037800KC23090006. We would like to thank Yanlin Li, a staff engineer at the Alibaba Group and a member of the Nacos Project Management Committee, for his valuable discussions and help. We also would like to thank our shepherd David Eyers for his guidance and detailed feedback on our manuscript.

References

[1] Micah Adler, Zihui Ge, James F Kurose, Don Towsley, and Steve Zabele. 2001. Channelization problem in large scale data dissemination. In *Proceedings Ninth International Conference on Network Protocols. ICNP 2001.* IEEE, Riverside, CA, USA, 100–109.

[2] Atul Adya, Gregory Cooper, Daniel Myers, and Michael Piatek. 2011. Thialfi: a client notification service for internet-scale applications. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 129––142. https://doi.org/10.1145/2043556.2043570

[3] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R.E. Strom, and D.C. Sturman. 1999. An Efficient Multicast Protocol for Content-Based Publish-Subscribe Systems. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems (ICDCS '99)*. IEEE Computer Society, USA, 262.

[4] David Bernstein. 2014. Containers and Cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing* 1, 3 (2014), 81–84.

[5] Luis Felipe Cabrera, Michael B. Jones, and Marvin Theimer. 2001. Herald: Achieving a Global Event Notification Service. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems (HOTOS '01)*. IEEE Computer Society, USA, 87–92.

[6] Fengyun Cao and Jaswinder Pal Singh. 2005. MEDYM: match-early with dynamic multicast for content-based publish-subscribe networks. In *Proceedings of the ACM/IFIP/USENIX 2005 International Conference on Middleware* (Grenoble, France) *(Middleware '05)*. Springer-Verlag, Berlin, Heidelberg, 292–313.

[7] Antonio Carzaniga, Cyrus Hall, Giovanni Toffetti Carughi, and Alexander L Wolf. 2009. *Practical high-throughput content-based routing using unicast state and probabilistic encodings.* Technical Report. Università della Svizzera italiana.

[8] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. 2001. Design and evaluation of a wide-area event notification service. *ACM Trans. Comput. Syst.* 19, 3 (Aug. 2001), 332–383. https://doi.org/10.1145/380749.380767

[9] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. 2003. Splitstream: High-bandwidth multicast in cooperative environments. *ACM SIGOPS operating systems review* 37, 5 (2003), 298–313.

[10] Miguel Castro, Peter Druschel, A-M Kermarrec, and Antony IT Rowstron. 2002. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in communications* 20, 8 (2002), 1489–1499.

[11] Paolo Costa, Matteo Migliavacca, Gian Pietro Picco, and Gianpaolo Cugola. 2003. Introducing reliability in content-based publish-subscribe through epidemic algorithms. In *Proceedings of the 2nd International Workshop on Distributed Event-Based Systems* (San Diego, California) *(DEBS '03)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/966618.966629

[12] Paolo Costa and Gian Pietro Picco. 2005. Semi-probabilistic content-based publish-subscribe. In *25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*. IEEE, Columbus, OH, USA, 575–585.

[13] Gianpaolo Cugola, Elisabetta Di Nitto, and Alfonso Fuggetta. 2001. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE transactions on Software Engineering* 27, 9 (2001), 827–850.

[14] 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. *ACM SIGOPS operating systems review* 41, 6 (2007), 205–220.

[15] Stephen E. Deering and David R. Cheriton. 1990. Multicast routing in datagram internetworks and extended LANs. *ACM Trans. Comput. Syst.* 8, 2 (May 1990), 85–110. https://doi.org/10.1145/78952.78953

[16] Cloud Native Computing Foundation. 2016. *Etcd: a distributed, reliable key-value store for the most critical data of a distributed system.* Cloud Native Computing Foundation. https://etcd.io/

[17] Alibaba Group. 2019. *Nacos: An easy-to-use dynamic service discovery, configuration and service management platform for building cloud native applications.* Alibaba Group. https://nacos.io/en-us/

[18] Long Guo, Dongxiang Zhang, Guoliang Li, Kian-Lee Tan, and Zhifeng Bao. 2015. Location-Aware Pub/Sub System: When Continuous Moving Queries Meet Dynamic Event Streams. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 843–857. https://doi.org/10.1145/2723372.2746481

[19] Abhishek Gupta, Ozgur D. Sahin, Divyakant Agrawal, and Amr El Abbadi. 2004. Meghdoot: content-based publish/subscribe over P2P networks. In *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware* (Toronto, Canada) *(Middleware '04)*. Springer-Verlag, Berlin, Heidelberg, 254–273.

[20] HashiCorp. 2019. *Consul: a tool for service discovery and configuration.* HashiCorp. https://www.consul.io/

[21] HedWig. 2014. *TIBCO rendezvous concepts.* TIBCO Software, Inc. https://docs.tibco.com/pub/rendezvous/8.3.1_january_2011/pdf/tib_rv_concepts.pdf

[22] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference* (Boston, MA) *(USENIXATC'10)*. USENIX Association, USA, 11.

[23] Mihalea Ion, Giovanni Russello, and Bruno Crispo. 2010. An implementation of event and filter confidentiality in pub/sub systems and its application to e-health. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (Chicago, Illinois, USA) *(CCS '10)*. Association for Computing Machinery, New York, NY, USA, 696–698. https://doi.org/10.1145/1866307.1866401

[24] Hans-Arno Jacobsen, Alex Cheung, Guoli Li, Balasubramaneyam Maniymaran, Vinod Muthusamy, and Reza Sherafat Kazemzadeh. 2010. The PADRES publish/subscribe system. In *Principles and Applications of Distributed Event-Based Systems.* IGI Global, Hershey, Pennsylvania, USA, 164–205.

[25] Shuping Ji, Chunyang Ye, Jun Wei, and Hans-Arno Jacobsen. 2015. MERC: Match at Edge and Route intra–Cluster for Content-based Publish/Subscribe Systems. In *Proceedings of the 16th Annual Middleware Conference* (Vancouver, BC, Canada) *(Middleware '15)*. Association for Computing Machinery, New York, NY, USA, 13–24. https://doi.org/10.1145/2814576.2814801

[26] Jay Kreps, Neha Narkhede, Jun Rao, et al. 2011. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, Vol. 11. Association for Computing Machinery, Athens, Greece, 1–7.

[27] Guoli Li, Shuang Hou, and Hans-Arno Jacobsen. 2005. A unified approach to routing, covering and merging in publish/subscribe systems based on modified binary decision diagrams. In *25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*. IEEE, Columbus, OH, USA, 447–457.

[28] Peter M. Mell and Timothy Grance. 2011. *SP 800-145. The NIST Definition of Cloud Computing.* Technical Report. National Institute of Standards & Technology, Gaithersburg, MD, USA.

[29] Dmitry Namiot and Manfred Sneps-Sneppe. 2014. On micro-services architecture. *International Journal of Open Information Technologies* 2, 9 (2014), 24–27.

[30] Netflix. 2015. *Eureka.* Netflix. https://github.com/xmartlabs/Eureka

[31] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Philadelphia, PA, USA, 305–319.

[32] Lukasz Opyrchal, Mark Astley, Joshua Auerbach, Guruduth Banavar, Robert Strom, and Daniel Sturman. 2000. Exploiting IP multicast in content-based publish-subscribe systems. In *IFIP/ACM International Conference on Distributed Systems Platforms* (New York, New York, USA) *(Middleware '00)*. Springer-Verlag, Berlin, Heidelberg, 185–207.

[33] Peter R. Pietzuch and Jean Bacon. 2002. Hermes: A Distributed Event-Based Middleware Architecture. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCSW '02)*. IEEE Computer Society, USA, 611–618.

[34] RabbitMQ. 2014. RabbitMQ. http://www.rabbitmq.com/

[35] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. 2001. A scalable content-addressable network. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (San Diego, California, USA) *(SIGCOMM '01)*. Association for Computing Machinery, New York, NY, USA, 161–172. https://doi.org/10.1145/383059.383072

[36] Anton Riabov, Zhen Liu, Joel L Wolf, Philip S Yu, and Li Zhang. 2002. Clustering algorithms for content-based publication-subscription systems. In *Proceedings 22nd International Conference on Distributed Computing Systems*. IEEE, Vienna, Austria, 133–142.

[37] Antony I. T. Rowstron and Peter Druschel. 2001. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg (Middleware '01)*. Springer-Verlag, Berlin, Heidelberg, 329–350.

[38] Amazon Web Services. 2014. *Amazon Simple Queue Service*. Amazon Web Services, Inc. http://aws.amazon.com/sqs/

[39] Yogeshwer Sharma, Philippe Ajoux, Petchean Ang, David Callies, Abhishek Choudhary, Laurent Demailly, Thomas Fersch, Liat Atsmon Guz, Andrzej Kotulski, Sachin Kulkarni, Sanjeev Kumar, Harry Li, Jun Li, Evgeniy Makeev, Kowshik Prakasam, Robbert Van Renesse, Sabyasachi Roy, Pratyush Seth, Yee Jiun Song, Kaushik Veeraraghavan, Benjamin Wester, and Peter Xie. 2015. Wormhole: reliable pub-sub to support geo-replicated internet services. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (Oakland, CA) *(NSDI'15)*. USENIX Association, USA, 351—-366.

[40] Zoe Sofia. 2000. Container technologies. *Hypatia* 15, 2 (2000), 181–201.

[41] Ion Stoica, Robert Morris, David Liben-Nowell, David R Karger, M Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. 2003. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on networking* 11, 1 (2003), 17–32.

[42] Yoav Tock, Nir Naaman, Avi Harpaz, and Gidon Gershinsky. 2005. Hierarchical Clustering of Message Flows in a Multicast Data Dissemination System. In *Proceedings ofthe 17th IASTED International Conference Parallel and Distributed Computing and Systems*, Vol. 5. ACTA Press, Phoenix, AZ, USA, 1–7.

[43] Toby Velte, Anthony Velte, and Robert Elsenpeter. 2009. *Cloud Computing, A Practical Approach* (1 ed.). McGraw-Hill, Inc., USA.

[44] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. 2001. *Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and.* Technical Report. Computer Science Division, University of California Berkeley, USA.