Protocol-Independent Context Propagation for Sharing Microservices in Multiple Environments

Hiroya Onoe
Kyoto University
Kyoto, Japan
onoe@inet.media.kyoto-u.ac.jp

Daisuke Kotani
Kyoto University
Kyoto, Japan
kotani@media.kyoto-u.ac.jp

Yasuo Okabe
Kyoto University
Kyoto, Japan
okabe@i.kyoto-u.ac.jp

***Abstract*—In systems designed based on microservice architecture, many production-like environments should be deployed for testing, staging, debugging, and previewing. One way to reduce resource consumption while deploying many environments is to allow sharing of common microservices in multiple environments, and current mechanisms extend application layer protocols like HTTP and gRPC to propagate contexts including environment identifiers and to route requests. However, microservices also use other protocols such as MySQL, Redis, Memcached, and AMQP, and extending each protocol requires lots of effort to implement the extensions. This paper proposes PiCoP, a framework to propagate contexts and route requests independently of application layer protocols. PiCoP consists of a protocol that propagates contexts without interpreting application layer protocols by adding contexts to the front of each TCP byte stream and a proxy that uses the protocol to route requests. We design the protocol to make instrumentation into a system as easy as possible. We showed that PiCoP could reduce resource usage, that the proxy's communication delay is within a practical range, and that it makes sharing microservices in multiple environments with any application layer protocols possible.**

***Index Terms*—Microservices, Context Propagation, Test Environment, Service Mesh**

I. Introduction

Continuous integration and continuous delivery are becoming increasingly important practices for software development [1]. These practices require multiple production-like environments for testing, staging, debugging, and previewing [2], [3], and some software assists in deploying them [4]–[6]. However, access to these environments is far less frequent than the production environment, and resources consumed by these environments are wasted. In particular, in a microservice architecture [7], where a single application consists of several microservices, a large number of microservices can easily lead to a high waste of resources.

Therefore, mechanisms to share instances of services commonly used by multiple environments have been proposed in an application that adopted the microservice architecture [8]–[11]. When arranging production-like environments in the microservice architecture, only a few services are usually updated from the production environment to add new features or fix bugs. While instances of stateful or updated services need to be deployed for each environment, instances of other services can be shared in all environments. Sharing of some services can be realized by giving requests contexts that identify the environment to be accessed, propagating the contexts, and routing the requests to each environment based on the contexts.

Previous methods by Wantedly [8], Mercari [9], Lyft [10], and Ambassador Labs [11] add contexts to HTTP headers or gRPC metadata and propagate them between services. Then, they route requests based on the contexts using Envoy [12] and Istio [13] in Kubernetes [14]. In the Lyft and Ambassador Labs proposals, the contexts are propagated within services using libraries for instrumentation [15] by OpenTelemetry [16].

Microservice communication uses various application layer protocols besides HTTP and gRPC. For example, there are protocols used for communication with database servers such as MySQL [17], PostgreSQL [18], Redis [19], and Memcached [20], and protocols such as AMQP [21] and MQTT [22] used for communication with message queues. Requests using each protocol must be given contexts in some form for routing them.

The previous methods cannot be applied for other communications, such as MySQL, PostgreSQL, Redis, Memcached, AMQP, and MQTT. To extend them to these protocols, we must provide mechanisms for each protocol to propagate contexts and route requests. It takes a lot of time and effort in implementation and instrumentation.

For this problem, this paper proposes PiCoP, a framework to share microservices in multiple environments by propagating contexts and routing requests independently of application layer protocols. PiCoP consists of a protocol that propagates contexts without interpreting application layer protocols and a proxy that uses the protocol to route requests.

PiCoP protocol is based on PROXY Protocol [23] and adds a context to the front of the TCP byte stream. It does not conflict with many application layer protocols by extending the signature of PROXY Protocol version 2. It is also designed to make instrumentation into an application as easy as possible by meeting OpenTelemetry standards.

PiCoP proxy interprets the context given to a request by the protocol and routes the request to the instance of the appropriate environment. The proxy is deployed for each service. First, the proxy receives correspondence between an identifier of an environment and a destination address of an instance to which requests are to be routed in advance from an administrator. Then, the proxy receives requests on behalf of the service, interprets the context, and routes the request to the appropriate instance based on the correspondence.

We implemented a prototype and showed that it is possible to share services in a system communicating with some application layer protocols. Furthermore, we showed that PiCoP could reduce resource usage when the number of environments exceeds a specific value. The more environments there are, the more significant the reduction is. We also showed that a delay due to communication through PiCoP proxy is within a practical and realistic range compared to Istio, which is widely used in previous methods. Furthermore, we showed that PiCoP gives a context to data with any application layer protocols, making it possible to share microservices protocol-independently in multiple environments.

The main contributions of this paper are as follows:

- We proposed a framework that enables sharing microservices in multiple environments without relying on application layer protocols. We showed that the framework could reduce resource usage and that its communication delay is within a realistic range for practical use.
- We proposed a protocol that enables protocol-independent context propagation in a form that can be instrumented into the systems as easily as possible.

This paper is organized as follows. First, Section II describes related works. Next, Section III describes the design of PiCoP, and Section IV the prototype implementation. Next, Section V evaluates PiCoP, and Section VI discusses the results and protocol independence. Finally, Section VII concludes the paper.

II. Related Works

A. Sharing Microservices in Multiple Environments

Wantedly [8], Mercari [9], Lyft [10], and Ambassador Labs [11] have each proposed systems that share microservices across multiple environments. These proposals add contexts that identify environments to HTTP headers or gRPC metadata. Then, they propagate the contexts between services and route requests based on the contexts. For request routing, they use Envoy [12], which is deployed as a sidecar and acts as a proxy, and Istio [13], which builds a service mesh [24] that uses Envoy in Kubernetes [14]. The Lyft and Ambassador Labs proposals use libraries provided by OpenTelemetry [16] to instrument context propagation within services. However, these propose context propagation mechanisms for each protocol, such as HTTP and gRPC. They do not describe context propagation for other protocols, and systems that use protocols other than HTTP or gRPC for communication cannot share microservices in multiple environments.

Parker et al. [25] and Sridharan [26] discuss testing in a production environment as chaos engineering. They describe integration testing mechanisms for using services in the production environment by sending replicas of requests from the production environment to services being tested or by sending requests from services being tested to services in the production environment. In this case, they explain that giving contexts to the requests and propagating them is necessary. However, they do not discuss how to propagate contexts.

In parallel testing, Rahman et al. have proposed a framework for locally executing parallel tests for microservices [27]. The framework is similar in that it requires deploying multiple environments. However, they do not describe reducing total resource usage by sharing microservices across multiple environments.

B. Context Propagation

Context propagation, necessary for sharing microservices in multiple environments, is the basis for monitoring, debugging, and diagnostics of distributed systems [15]. Much research about it has been done in this area.

Dapper [28], Pinpoint [29], X-Trace [30], Pip [31], and Canopy [32] all enable tracing distributed systems by propagating contexts. Their context propagation can be applied for sharing microservices in multiple environments. However, these proposals depend on specific communication protocols and platforms.

There are some proposals for methods and frameworks to reduce the effort in instrumenting by sharing a context propagation mechanism [15], [32]–[34]. In this study, it is also significant to use these existing context propagation mechanisms as much as possible to reduce instrumentation effort. However, these require a context propagation mechanism for each protocol. OpenTelemetry [16] provides specifications and libraries to standardize context propagation mechanisms and instrumentation. Propagators API [35], a standard for context propagation, takes HTTP headers as an example of a medium for propagating contexts and requires other mediums to be compatible with HTTP headers.

There are studies by Chanda et al. [36] and PROXY protocol [23] regarding protocol-independent context propagation.

Chanda et al. pointed out the problem that context propagation depends on application layer protocols and proposed a mechanism to propagate a context across socket and pipe channels without changing applications [36]. For INTERNET sockets, the context is encapsulated in IP packets as an IP option. However, when sharing microservices in multiple environments, the context is needed per requests of application layer protocol. Therefore, sending it per IP packets is wasteful.

PROXY protocol [23] allows protocol-independently propagating source client information through a reverse proxy server. PROXY Protocol adds the information at the front of the TCP byte stream. Its version 2 signatures are designed not to conflict with many application layer protocols. PROXY protocol makes information propagation protocol-independent, and PiCoP follows suit, as described in Section III-B.

III. Design

A. Overview

This section describes PiCoP that enables sharing microservices in multiple environments with protocol-independent context propagation and request routing. PiCoP consists of a protocol that propagates contexts without interpreting application layer protocols and a proxy that routes requests to the appropriate environment based on the propagated context.

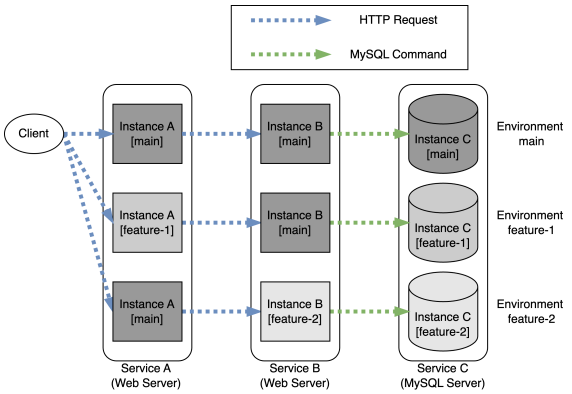

Fig. 1. The cluster deploying the application consisting of services A, B, and C

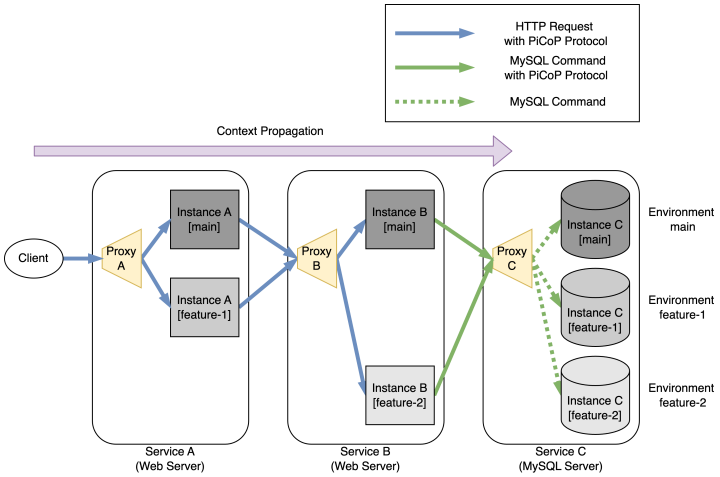

Fig. 2. The cluster installed PiCoP

PiCoP distinguishes between multiple environments using an ID that uniquely identifies the environment (environment ID). The protocol adds a context in the form of key-value pairs to a request, and the context of a request to be routed has a field with the field name "Env-Id" and the field value as the environment ID of the environment to be sent. The proxy receives the correspondence between the environment ID and a destination address from an administrator in advance. When receiving a request, the proxy recognizes the environment ID added to the request using the protocol and routes the request to the corresponding destination address.

For example, we consider a cluster deploying an application with three services: stateless services A and B and stateful service C (Fig. 1). Services A and B are web servers. Service C is a MySQL server. First, service A receives HTTP requests from clients. Next, service A sends HTTP requests to service B, service B sends MySQL commands to service C, and each processes the requests or commands. Finally, service A sends responses back to the clients.

We assume the cluster has three environments. A base environment, "main", has services A, B, and C with the same code as the production environment. In addition to "main", there are two environments: "feature-1", where only service A is modified from "main", and "feature-2", where only service B is modified.

We consider the case where PiCoP is installed in the cluster (Fig. 2). We deploy PiCoP proxy for each service. For "main", we provide instances A[main], B[main], and C[main], which is the base of services A, B, and C. For "feature-1", we deploy instance A[feature-1] with service A modified and share instance B[main] with "main". For "feature-2", we deploy instance B[feature-2] with service B modified and share instance A[main] with "main". Finally, we provide instances C[main], C[feature-1], and C[feature-2] for each environment to separate data.

First, proxy A receives a client request addressed to service A. Proxy A has route information in advance, as shown in Table. I. Proxy A forwards the request to instance A[feature-1] if the environment ID of the request is "feature-1" and otherwise to instance A[main]. Next, each instance A sends a request addressed to service B to proxy B. Proxy B forwards the request to instance B[feature-2] if the environment ID is "feature-2" and otherwise to instance B[main], based on Table. I. Each instance B sends a request addressed to service C to proxy C. Proxy C forwards the request to instance C[feature-1] if the environment ID is "feature-1" and to instance C[feature-2] if the environment ID is "feature-2" and otherwise to instance C[main], based on Table. I.

TABLE I
ROUTE INFORMATION

Proxy Name	Environment ID	Destination of Requests
Proxy A	*	Instance A[main]
	feature-1	Instance A[feature-1]
Proxy B	*	Instance B[main]
	feature-2	Instance B[feature-2]
Proxy C	*	Instance C[main]
	feature-1	Instance C[feature-1]
	feature-2	Instance C[feature-2]

In other words, if the environment ID added to a request is "feature-1", the request is processed by instances A[feature-1], B[main], and C[feature-1]. If the environment ID is "feature-2", it is processed by instances A[main], B[feature-2], and C[feature-2]. If any other one, it is processed by instance A[main], B[main], and C[main].

This way, the proxy routes requests according to the environment ID added to the requests using the protocol, allowing multiple environments to be separated while sharing some stateless services.

B. Protocol

1) Requirements and Approach: The protocol satisfies the following three requirements.

1) It can be used regardless of the type of application layer protocol.

TABLE II
THE SPECIFICATION OF PICOP PROTOCOL PER BYTES

Byte	Description
1-12	The same signature as PROXY protocol version 2
13	The upper 4 bits: the signature The lower 4 bits: the version
14,15	The byte length of the context
16-	The context in the form of key-value pairs

2) It does not conflict with any application layer protocols.

3) It is as easy as possible to instrument into systems.

Requirement 1 is necessary to propagate contexts in a protocol-independent manner. For this reason, as with PROXY protocol, data in PiCoP protocol are given at the front of the TCP byte stream.

Requirement 2 is necessary because it adds the data to the position where application layer protocol data originally exists, similar to PROXY Protocol. For this reason, the first part of the data shall be the signature of PROXY protocol version 2. Then, a version number rejected by PROXY protocol version 2 is added after the signature. PiCoP protocol is not version 3 of PROXY Protocol because PiCoP protocol is used in situations different from PROXY Protocol. PROXY Protocol is utilized for communications from a client to a server through a reverse proxy server. PiCoP protocol is mainly used for communications between services in a cluster.

Requirement 3 is necessary because the instrumentation is a barrier to installing a new context propagation mechanism. To this end, PiCoP protocol satisfies the specifications for context propagation media defined in the OpenTelemetry Propagators API [35], allowing the use of other standards and libraries for context propagation. Following the specifications also facilitates the utilization of PiCoP protocol for context propagation in various usages, such as monitoring, debugging, and diagnostics of distributed systems. We do not use Type-Length-Value (TLV vectors) of PROXY Protocol version 2, which propagates optional information, because it does not satisfy the Propagators API's specifications.

2) Specification: The service requesting a connection transmits data in PiCoP protocol. The service accepting the connection request must not insert any data until the service requesting the connection has finished sending the data.

Data in PiCoP protocol are in binary format. The specification per bytes is shown in Table. II.

The first to twelfth bytes and the upper four bits of the thirteenth byte are the signature; the first to twelfth bytes are the same as the signature of PROXY protocol version2, which is `\x0D \x0A \x0D \x0A \x00 \x0D \x0A \x51 \x55 \x49 \x54 \x0A` in the byte sequence. The upper four bits of the thirteenth byte are `\x0`. PROXY protocol version 2 will reject this part unless it is `\x2`, so it does not conflict with PiCoP protocol.

The lower four bits of the thirteenth byte are the version of PiCoP protocol, `\x1`. Other versions are rejected.

The fourteenth and fifteenth bytes are the length of the context, the number of bytes in the data after the sixteenth byte.

The sixteenth and succeeding bytes are the context. The context is a string of US-ASCII characters, which is converted to a single byte per character. The context is the format of key-value pairs that comprises US-ASCII characters that make up a valid HTTP header field. By this, the context satisfies the Carrier specification required by the TextMap Propagator as defined by OpenTelemetry Propagators API [35]. The syntax rules for the context are based on the HTTP header rules defined in RFC 9110 [37] and RFC 9112 [38]. For simplification and data length reduction, we exclude obsolete rules and regulations to improve readability, such as comments and whitespace. It is expressed as follows in ABNF [39].

```
context     = field *( CRLF field )
field       = field-name ":" field-value
field-name  = token
field-value = *( VCHAR / SP / HTAB )
token       = 1*tchar
tchar       = "!" / "#" / "$" / "%" / "&"
            / "'" / "*" / "+" / "-" / "."
            / "^" / "_" / "`" / "|" / "~"
            / DIGIT / ALPHA
```

Each field is arranged in order from the front: the field name, colon (":"), and the field value. The context is a string of multiple fields concatenated by CRLF.

The field name is a string of US-ASCII characters with symbols, digits, and alphabetic characters, excluding delimiters (`"(),/:;<=>?@[\]{}|`"). The field name is not case-sensitive. All field names must be different. The field value is a string of US-ASCII characters with printable characters, spaces, and horizontal tabs.

C. Proxy

One or more proxies are provided for each service and act as sidecars, like Envoy. When the proxy receives a request from a client on behalf of the service, it interprets the data in PiCoP protocol and gets an environment ID. It then routes the request to the appropriate instance based on the registered route information and returns the received response to the client.

The proxy has the following information.

- Service ID
- Default route
- Whether to propagate data in PiCoP protocol or not
- Route information

The service ID indicates for which service the proxy is responsible. Multiple proxies can have the same service ID because of the scale out of proxies. Proxies with the same service ID should have the same route information.

The default route is the destination address to which requests are routed when the environment ID does not match any of the registered route information. It is the address of the instance of the base environment. In the example, the address

corresponds to the destination of requests in the environment ID "*" in the Table. I.

The proxy has information about whether or not to propagate data in PiCoP protocol. When propagation at the proxy is true, the proxy adds the data in PiCoP protocol to requests from the proxy to a service. In the example, proxies A and B propagate it, and proxy C does not. This choice is made per service and does not need to be made dynamically per request. This choice is necessary because some services do not receive data in PiCoP protocol. Services in the middle of request processing in an application, such as services A and B, must propagate the environment ID to the following services. Such services send and receive requests with data in PiCoP protocol. On the other hand, a service like C, which exists at the end of request processing in an application, does not need to know the environment ID and does not need instrumentation to process PiCoP protocol. Such services will receive requests without data in PiCoP protocol.

The route information is the correspondence between the environment ID and the destination address.

There is a restriction on connection reuse for requests made through proxies, such as persistent connections in HTTP/1.1 [38]. Requests with different environment IDs must be transmitted over different connections because data in PiCoP protocol are sent per connection. Simply not reusing connections at all avoids this restriction.

IV. Implementation

We implemented a prototype based on the example in Section III (Fig. 3).

We provide a proxy controller to manage proxies and route information dynamically. The proxy controller is implemented in Go and operates as a web server that accepts HTTP requests for registering proxy information and route information from administrators. The proxy information consists of an ID that uniquely identifies the proxy in the cluster, a service ID, and a destination address of the proxy used to send route information. The route information comprises a service ID, an environment ID, and a destination address. When the proxy controller receives a request to register a new proxy, it pushes all the route information corresponding to the proxy's service ID to the proxy. When it receives a request to register route information, it pushes the route information to all proxies corresponding to the service ID of the route information. The proxy controller updates route information in each proxy periodically to reduce the load rather than in real-time. In addition, the proxy information and route information persists by storing them in the MySQL server.

The proxy is implemented in Go and routes TCP connections. It accepts connections from the specified port and routes them according to the environment ID. It also works as a web server that accepts HTTP requests for registering route information from the proxy controller. The route information received from the proxy controller is stored in memory for high-speed lookup.

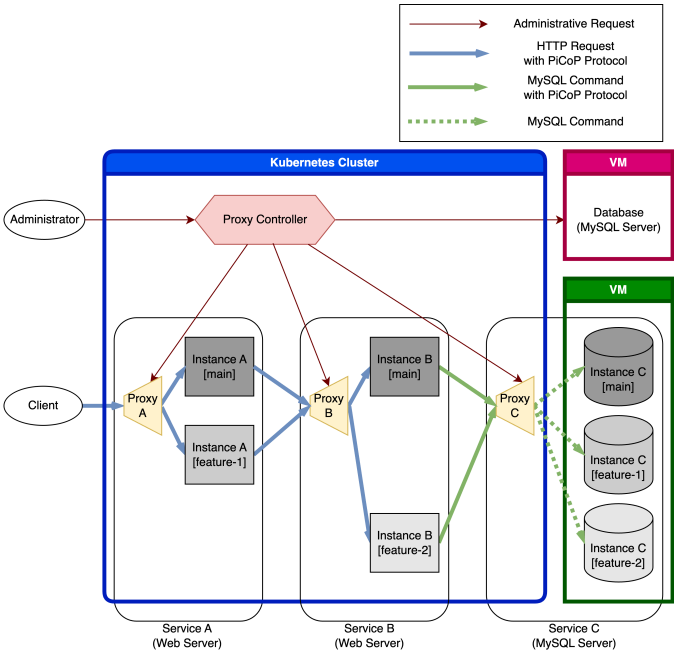

Fig. 3. Prototype implementation architecture

The instances of services A and B, web servers, are implemented in Go, using instrumentation libraries provided by OpenTelemetry to propagate the contexts of incoming requests and add them to outgoing requests.

The proxies, proxy controller, and instances of all services run as containers. All containers except service C are located on a single Kubernetes cluster. The instances of service C, MySQL servers, are located on separate virtual machines, assuming it is outside the cluster. The other virtual machine is a MySQL server for storing data managed by the proxy controller.

Through the prototype implementation, we confirmed that it is possible to share microservices in multiple environments in a protocol-independent manner. We also confirmed that the same proxy implementation could route data from two different protocols: HTTP requests and MySQL commands.

V. Evaluation

This section evaluates PiCoP's performance regarding communication delays due to the proxy and resource reduction by sharing.

A. Environment

The evaluation environment consists of four virtual machines (Table. IV) with the same specification on one physical machine (Table. III). One virtual machine sends benchmark requests. Assuming a system will run on Kubernetes, we construct a Kubernetes cluster (Table. V) on three virtual machines. One virtual machine is a control plane, and the others are worker nodes.

TABLE III
PHYSICAL MACHINE SPECIFICATION

CPU	AMD EPYC 7452 (32 cores, 64 threads)
Memory	377GiB
OS	Ubuntu 22.04.1 (kernel 5.15.0-47-generic)

TABLE IV
VIRTUAL MACHINE SPECIFICATION

The Number of vCPUs[a]	8
Memory	16GiB
OS	Ubuntu 22.04.1 (kernel 5.15.0-58-generic)

[a]One vCPU corresponds to 1 thread on a physical machine.

B. Proxy Communication Delay

We clarify the communication delay introduced by the PiCoP and existing (Istio) proxy.

1) Methods: We measure the response time by sending HTTP/1.1 requests under four conditions (Fig. 4) to a server that accepts HTTP/1.1 requests, an nginx server (version 1.23.3).

In the conditions "base" and "base+gw+istio", we send HTTP requests with "Env-Id:main" in the HTTP header. In the condition "base+picop" and "base+gw+picop", we send HTTP requests with data in PiCoP protocol whose context is "Env-Id:main".

We configure PiCoP proxy to route requests with an environment ID of "main" to the nginx server. The proxy does not propagate data in PiCoP protocol because the nginx server cannot accept the data.

The version of Istio is 1.16.1. We configure it to route requests with the value "main" in the HTTP header field "Env-Id" to the nginx server. We also configure it not to scale out its component automatically to measure in the same conditions. Other settings are defaults.

TABLE V
KUBERNETES CLUSTER SPECIFICATION

Version	1.26.0
Container Runtime	containerd (version 1.6.14)
CNI plugin	cilium (version 1.12.2)

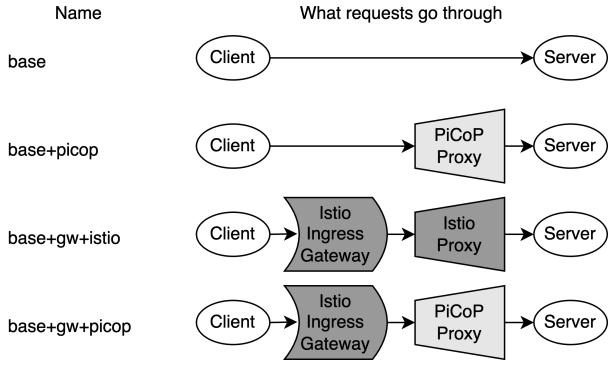

Fig. 4. Conditions for measuring proxy communication delay

The conditions for the requests are the same as Istio performance evaluation [40], with 1kB as HTTP request payload and 1000 requests per second. We send a total of 10000 requests for 10 seconds and measure their response time with various numbers of simultaneous client connections.

Each proxy and nginx server runs as a single container on Kubernetes. We place these containers on one virtual machine. Their resource usage is limited to 2vCPUs and 1GiB memory, similar to the default value for Istio proxies[1].

2) Results: We show the 90th and 99th percentile of response time measured for each condition and the number of simultaneous client connections in Fig. 5 and 6.

By comparing "base" and "base+picop", we clarify the communication delay introduced by PiCoP proxy. Regardless of the number of simultaneous client connections, the response time with PiCoP proxy increased in comparison to without them. In the 90th percentile, the delay by PiCoP proxy was between 3.2 ms and 12.3 ms. In the 99th percentile, the delay by PiCoP proxy was between 6.7 ms and 13.3 ms.

By contrasting "base+gw+istio" and "base+gw+picop", we illustrate the difference in communication delay between PiCoP and Istio proxy. In the 90th percentile, the delay by PiCoP proxy was almost equal to the delay by Istio proxy. In the 99th percentile, PiCoP proxy was 0.3 ms to 2.7 ms slower than Istio proxy.

C. Resource Reduction

We clarify the amount of resource reduction by sharing microservices using PiCoP.

1) Methods: We place a load on microservices (nginx servers (version 1.23.3)) by continuously sending requests under two conditions: In the "share: yes" condition, microservices are shared by using PiCoP. In the "share: no" condition, microservices are not shared. We scale out microservices and proxies according to CPU utilization and count the total number of containers of them.

In the "share: yes" condition, we send requests to a single microservice via PiCoP proxy (Fig. 7). The proxy does not propagate data in PiCoP protocol because the nginx server cannot accept the data. On the other hand, in the "share: no" condition, we send requests evenly to microservices equal to the number of environments (Fig. 8).

Assuming deploying one environment per developer or feature development, we change the number of environments between 1 and 100.

We assume that a few developers and automated test execution programs access each environment for staging, testing, debugging, and previewing. In other words, unlike production environments, we do not assume that many and unspecified people access their environments. Therefore, we send 100 or 1000 requests per second in total for all environments from clients equal to the number of environments. In other words, in the "share: yes" condition, the proxy is connected to clients

[1]https://github.com/istio/istio/blob/1258e3fdad4421078dbd0962c3df09b8e9bc752b/manifests/charts/istio-control/istio-discovery/values.yaml#L359-L365 (Accessed: 2023-01-31)

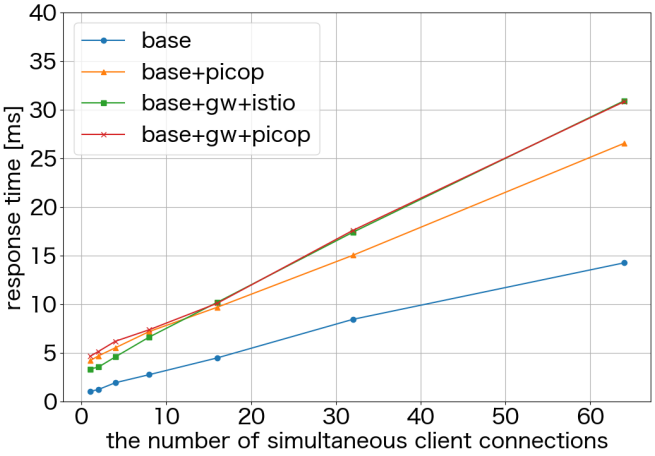

Fig. 5. The 90th percentile of response time

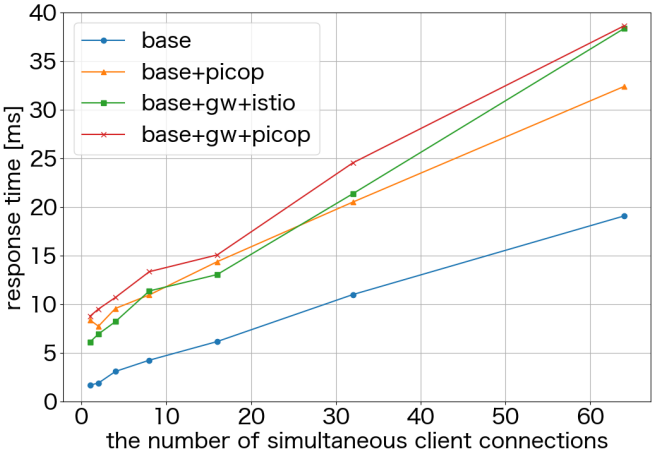

Fig. 6. The 99th percentile of response time

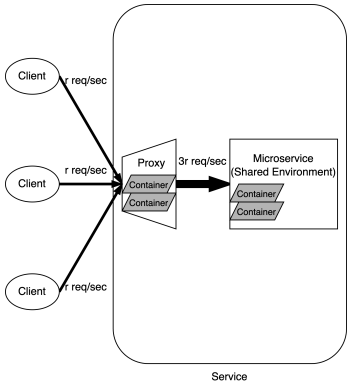

Fig. 7. The condition of sharing microservices

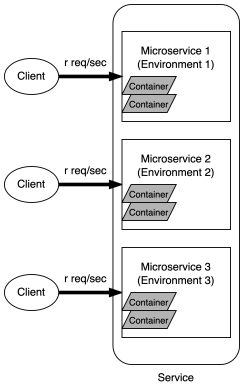

Fig. 8. The condition of not sharing microservices

equal to the number of environments, and in the "share: no" condition, each client connects to one nginx server.

In the " share: yes " condition, we send HTTP/1.1 requests with PiCoP protocol data whose context is "Env-Id:main". In the " share: no " condition, we send HTTP/1.1 requests with "Env-Id:main" in the HTTP header. In addition, as with Istio performance evaluation [40], the number of bytes in the payload of the HTTP request is set to 1kB.

Each proxy and nginx server runs as multiple containers on Kubernetes. Each container's resource is limited to 0.1 vCPUs and 128MiB memory. For scaling out, we use Kubernetes Horizontal Pod Autoscaler. Containers are scaled out so that the CPU utilization of all containers meets the default threshold[2] of 80% or less and that the number of containers is minimized.

We measure the total number of containers in the state where the number of containers is constant and where 99% of the requests sent are returned within one second in the last 60 seconds.

[2]https://github.com/kubernetes/kubernetes/blob/c090810c4c96e0c5acc05ab2094a5d46669cae86/pkg/apis/autoscaling/annotations.go#L34 (Accessed: 2023-01-31)

2) Results: We show the results for 100 and 1000 requests per second in Fig. 9 and 10.

In the "share: yes" condition, the number of containers remained approximately constant regardless of the number of environments, ranging from 3 to 6 for 100 requests per second and from 27 to 37 for 1000 requests per second. In the "share: no" condition, the number of containers increased proportionately to the number of environments. The proxies and nginx servers were not scaled out, and the number of environments matched the number of containers, except for the case of 1000 requests per second and one environment.

Regarding 100 requests per second, resource usage in the "share: yes" condition was lower than in the "share: no" condition when the number of environments was 20 or more. In the case of 1000 requests per second, resource usage in the "share: yes" condition was lower than in the "share: no" condition when the number of environments was 40 or more.

VI. Discussion

Based on the results of Section V, we discuss PiCoP's performance in terms of proxy communication delay and resource reduction by sharing microservices. We also discuss the degree and constraints of protocol independence of PiCoP.

A. Proxy Communication Delay

Communication through PiCoP proxy is slower than without the proxy.

Compared to Istio, which is widely used in the existing method, the delay of PiCoP proxy is almost the same as Istio proxy. It should be noted that while Istio proxy has various features other than request routing, it is an extension of Envoy, a high-performance proxy implemented in C++ [12]. The delay of PiCoP proxy, which is comparable to that of Istio proxy, is within a practical and realistic range.

B. Resource Reduction

We consider the total resource usage by regarding the total number of containers.

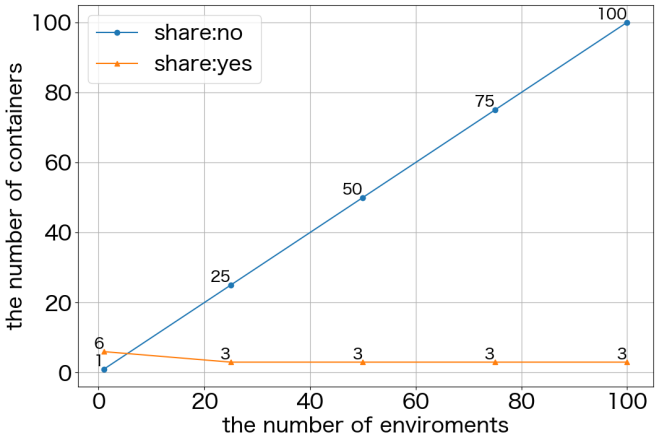

Fig. 9. Relationship between the number of environments and the number of containers at 100 requests per second

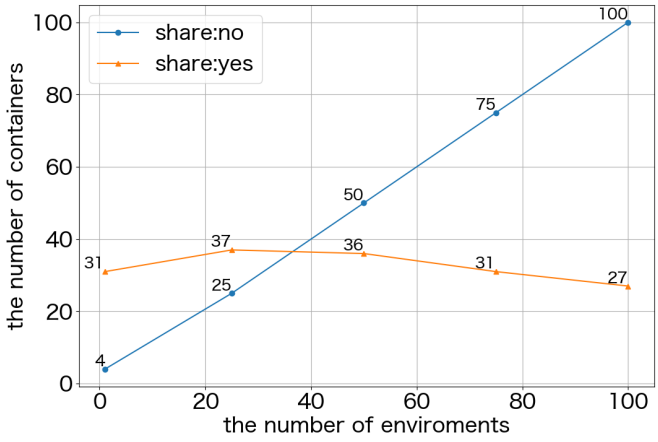

Fig. 10. Relationship between the number of environments and the number of containers at 1000 requests per second

Environments for testing, staging, debugging, and previewing have low access by only a few clients. Therefore, if microservices are not shared, resources are required in proportion to the number of environments, resulting in a large amount of waste. On the other hand, if shared, access to all environments can be put together, but proxies are required. If the number of environments is small, the increase in resources due to proxies is more significant than the reduction in resources due to sharing, so sharing cannot reduce overall resources. On the other hand, if the number of environments is large, the resource reduction due to sharing is more significant than the increase due to proxies, so sharing can reduce overall resources.

When not shared, resource consumption increases proportionately to the number of environments. On the other hand, when shared, resource usage remains constant regardless of the number of environments. Therefore, resource reduction due to sharing increases as the number of environments increases.

Also, the higher the number of requests per second, the more increased lower limit on the number of environments where sharing can reduce resources.

C. Protocol Independence

We discuss, separately for inter-service and intra-service propagation, how protocol-independent PiCoP is and its constraints.

1) Inter-Service Propagation: PiCoP is independent of application layer protocols because it does not interpret them. Therefore, it can propagate contexts between services in the same way for all protocols, including protocols that cannot have optional data (Table. VI). Furthermore, PiCoP allows common proxies to route requests for any application layer protocol.

For all protocols, there is a restriction on connection reuse, as described in Section III-C. For example, HTTP/1.1 persistent connections [38] must only be allowed for requests with the same context. Otherwise, if an application layer protocol library is implemented to pool and reuse connections, reuse must be allowed only for requests with the same context.

We compare the performance degradation due to this restriction with existing methods. First, we consider the case that does not share microservices in multiple environments and deploy them for each environment. In that case, connections are not also reused between environments like PiCoP. Therefore, the performance of PiCoP is similar to the case. Next, we consider the case of existing systems that propagate context through HTTP headers, as described in Section II-A. In that case, we can use the persistent connections of HTTP without any restrictions. Therefore, compared to this case, PiCoP has the overhead for establishing TCP connections each time.

PiCoP depends on transport layer protocols, and PiCoP proxy must be implemented for each transport layer protocol. PiCoP protocol and proxy assume to be used with TCP. For TCP, by adding data in PiCoP protocol to the front of the byte stream, PiCoP works as shown in the prototype implementation in Section IV. For UDP, context propagation is also possible by adding data in PiCoP protocol to the front of the datagram. In unidirectional communication, a proxy must be deployed for the service receiving requests, and in bidirectional communication, a proxy must be deployed for both services communicating with each other. For protocols that provide bidirectional streams over UDP, such as QUIC [45], we also can add data in PiCoP protocol at the front of the stream. This way, total data can be reduced compared to sending it per UDP datagram.

2) Intra-Service Propagation: In services in the middle of request processing in an application, such as a web server, it is necessary to extract contexts from incoming requests, propagate the contexts within the services, and add the contexts to outgoing requests. As with the existing methods, we must instrument the process to extract, propagate, and add contexts into each service. We can instrument context propagation within services using libraries maintained by OpenTelemetry. On the other hand, for instrumenting the process to extract and add contexts, we need to implement libraries for PiCoP protocol per each language and application layer protocol, as

TABLE VI
WHICH APPLICATION PROTOCOLS CAN HAVE OPTIONAL DATA

Protocols that *cannot* have optional data	**Protocols that *can* have optional data**
MySQL, PostgreSQL, Memcached, Redis, MongoDB Wire Protocol [41], MQTT, Kafka Wire Protocol [43], TDS (Microsoft SQL Server) [44]	HTTP, gRPC, AMQP, Cassandra Native Protocol [42]

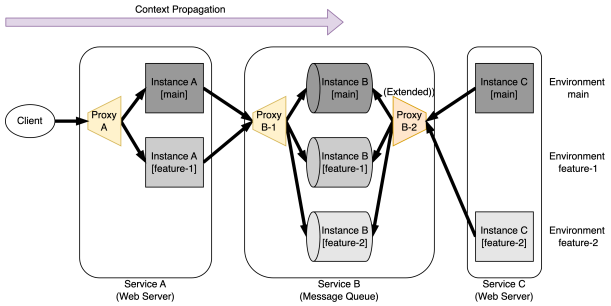

Fig. 11. Cluster for an application that processes asynchronously using a message queue

with OpenTelemetry.

In services at the end of request processing in an application, such as a database, it is not necessary to propagate contexts within services.

In asynchronous processing, there can be situations where a service sending requests has no context. Therefore, it is necessary to extend PiCoP proxy and isolate environments according to the characteristics of the processing. For example, we consider separating environments in a cluster shown in Fig. 11. Service A and C are web servers, and service B is a message queue. Service A receives requests from a client and publishes messages to service B. Service C pulls the messages from service B. In this case, contexts are propagated from a client to service A, but not to service C. Therefore, requests from service C to service B have no context. For this reason, proxy B-2 for service B, which accepts requests from service C, must route a request according to from which instance C the proxy received the request. If instance C[feature-2] sends a request to service B, proxy B-2 sends the request to instance B[feature-2]. On the other hand, if instance C[main] sends a request to service B, proxy B-2 needs to distribute the request equally to instance B[main] and instance B[feature-1].

VII. CONCLUSION

In this paper, we proposed PiCoP, a framework to achieve shared microservices in multiple environments by propagating contexts and routing requests independently of application layer protocols. PiCoP consists of a protocol that propagates contexts without interpreting application layer protocols and a proxy that uses the protocol to route requests. The protocol adds context to the front of the TCP byte stream. It can be used without conflict with many application layer protocols. It is also designed to be as easy to instrument into a system and can be applied to context propagation for other purposes. The proxy interprets a context given to a request by the protocol and routes the requests to the instance of the appropriate environment based on the route information.

We have implemented a prototype of PiCoP and confirmed that sharing microservices in multiple environments in a system communicating with some application layer protocols is possible. We showed that sharing microservices with PiCoP can reduce resource consumption and that the reduction increases with the number of environments. We also showed by performance evaluation that the prototype proxy could be used within a practically realistic delay compared to Istio. Furthermore, we showed that PiCoP enables context propagation between services in any application layer protocol, including protocols that cannot have optional data. PiCoP allows common proxies to route requests for any application layer protocol. On the other hand, we showed that implementation for context propagation within services is still required per language and application layer protocol. We also showed restrictions on connection reuse and asynchronous processing.

Resolving these restrictions is a future challenge. It is also future work to evaluate the applicability of PiCoP protocol to context propagation for other purposes in practice.

The implementation of PiCoP is available at https://github.com/picop-rd.

ACKNOWLEDGMENT

This work was supported in part by JSPS KAKENHI 21K17732.

REFERENCES

[1] M. Shahin, M. Ali Babar, and L. Zhu, "Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices," *IEEE Access*, vol. 5, pp. 3909–3943, 2017.
[2] M. Fowler and M. Foemmel, "Continuous integration," 2006.
[3] J. Humble and D. Farley, *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.
[4] Jenkins X, "Previews." [Online]. Available: https://jenkins-x.io/v3/develop/environments/preview/ (Accessed: 2023-01-31).
[5] Vercel, "Preview deloyments." [Online]. Available: https://vercel.com/docs/concepts/deployments/preview-deployments (Accessed: 2023-01-31).
[6] Telepresence, "Telepresence." [Online]. Available: https://www.telepresence.io/ (Accessed: 2023-01-31).
[7] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: yesterday, today, and tomorrow," *Present and ulterior software engineering*, pp. 195–216, 2017.
[8] Wantedly, "kubefork-controller." [Online]. Available: https://github.com/wantedly/kubefork-controller/ (Accessed: 2023-01-31).
[9] S. Rajesh, "Dynamic service routing using istio." [Online]. Available: https://engineering.mercari.com/en/blog/entry/20220218-dynamic-service-routing-using-istio/ (Accessed: 2023-01-31).

[10] M. Grossman, "Scaling productivity on microservices at lyft (part 3): Extending our envoy mesh with staging overrides." [Online]. Available: https://eng.lyft.com/scaling-productivity-on-microservices-at-lyft-part-3-extending-our-envoy-mesh-with-staging-fdaafafca82f (Accessed: 2023-01-31).

[11] D. Bryant, "Testing microservices: You 're thinking about (environment) isolation all wrong." [Online]. Available: https://blog.getambassador.io/testing-microservices-youre-thinking-about-environment-isolation-all-wrong-84f22034a6ef (Accessed: 2023-01-31).

[12] Envoy, "Envoy." [Online]. Available: https://www.envoyproxy.io/ (Accessed: 2023-01-31).

[13] Istio, "Istio." [Online]. Available: https://istio.io/ (Accessed: 2023-01-31).

[14] Kubernetes, "Kubernetes." [Online]. Available: https://kubernetes.io/ (Accessed: 2023-01-31).

[15] J. Mace and R. Fonseca, "Universal context propagation for distributed system instrumentation," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3190508.3190526

[16] OpenTelemetry, "Opentelemetry." [Online]. Available: https://opentelemetry.io/ (Accessed: 2023-01-31).

[17] MySQL, "Mysql." [Online]. Available: https://mysql.com/ (Accessed: 2023-01-31).

[18] PostgreSQL, "Postgresql." [Online]. Available: https://postgresql.org/ (Accessed: 2023-01-31).

[19] Redis, "Redis." [Online]. Available: https://redis.io/ (Accessed: 2023-01-31).

[20] Memcached, "Memcached." [Online]. Available: https://memcached.org/ (Accessed: 2023-01-31).

[21] AMQP, "Amqp." [Online]. Available: https://amqp.org/ (Accessed: 2023-01-31).

[22] MQTT, "Mqtt." [Online]. Available: https://mqtt.org/ (Accessed: 2023-01-31).

[23] W. Tarreau, "The proxy protocol versions 1 & 2," 2020. [Online]. Available: https://www.haproxy.org/download/1.8/doc/proxy-protocol.txt (Accessed: 2023-01-31).

[24] W. Li, Y. Lemieux, J. Gao, Z. Zhao, and Y. Han, "Service mesh: Challenges, state of the art, and future research opportunities," in *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, 2019, pp. 122–1225.

[25] A. Parker, D. Spoonhower, J. Mace, B. Sigelman, and R. Isaacs, *Distributed tracing in practice: Instrumenting, analyzing, and debugging microservices*. O'Reilly Media, 2020.

[26] C. Sridharan, "Testing in production, the safe way." [Online]. Available: https://copyconstruct.medium.com/testing-in-production-the-safe-way-18ca102d0ef1 (Accessed: 2023-01-31).

[27] M. Rahman, Z. Chen, and J. Gao, "A service framework for parallel test execution on a developer's local development workstation," in *2015 IEEE Symposium on Service-Oriented System Engineering*, 2015, pp. 153–160.

[28] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," Google, Inc., Tech. Rep., 2010. [Online]. Available: https://research.google.com/archive/papers/dapper-2010-1.pdf

[29] M. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: problem determination in large, dynamic internet services," in *Proceedings International Conference on Dependable Systems and Networks*, 2002, pp. 595–604.

[30] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, "X-trace: A pervasive network tracing framework," in *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, ser. NSDI'07. USA: USENIX Association, 2007, p. 20.

[31] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat, "Pip: Detecting the unexpected in distributed systems," in *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3*, ser. NSDI'06. USA: USENIX Association, 2006, p. 9.

[32] J. Kaldor, J. Mace, M. Bejda, E. Gao, W. Kuropatwa, J. O'Neill, K. W. Ong, B. Schaller, P. Shan, B. Viscomi, V. Venkataraman, K. Veeraraghavan, and Y. J. Song, "Canopy: An end-to-end performance tracing and analysis system," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 34–50. [Online]. Available: https://doi.org/10.1145/3132747.3132749

[33] J. Mace, R. Roelke, and R. Fonseca, "Pivot tracing: Dynamic causal monitoring for distributed systems," in *Proceedings of the 25th Symposium on Operating Systems Principles*, ser. SOSP '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 378–393. [Online]. Available: https://doi.org/10.1145/2815400.2815415

[34] E. Ates, L. Sturmann, M. Toslali, O. Krieger, R. Megginson, A. K. Coskun, and R. R. Sambasivan, "An automated, cross-layer instrumentation framework for diagnosing performance problems in distributed applications," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 165–170. [Online]. Available: https://doi.org/10.1145/3357223.3362704

[35] OpenTelemetry, "Propagators api." [Online]. Available: https://opentelemetry.io/docs/reference/specification/context/api-propagators/ (Accessed: 2023-01-31).

[36] A. Chanda, K. Elmeleegy, A. L. Cox, and W. Zwaenepoel, "Causeway: Operating system support for controlling and analyzing the execution of distributed programs," in *Proceedings of the 10th Conference on Hot Topics in Operating Systems - Volume 10*, ser. HOTOS'05. USA: USENIX Association, 2005, p. 18.

[37] R. T. Fielding, M. Nottingham, and J. Reschke, "HTTP Semantics," RFC 9110, Jun. 2022. [Online]. Available: https://www.rfc-editor.org/info/rfc9110

[38] ——, "HTTP/1.1," RFC 9112, Jun. 2022. [Online]. Available: https://www.rfc-editor.org/info/rfc9112

[39] D. Crocker and P. Overell, "Augmented BNF for Syntax Specifications: ABNF," RFC 5234, Jan. 2008. [Online]. Available: https://www.rfc-editor.org/info/rfc5234

[40] Istio, "Performance and scalability." [Online]. Available: https://istio.io/latest/docs/ops/deployment/performance-and-scalability/ (Accessed: 2023-01-31).

[41] MongoDB, "Mongodb wire protocol." [Online]. Available: https://www.mongodb.com/docs/manual/reference/mongodb-wire-protocol (Accessed: 2023-04-18).

[42] A. Cassandra, "Native protocols." [Online]. Available: https://cassandra.apache.org/_/native_protocol.html (Accessed: 2023-04-18).

[43] A. Kafka, "Kafka protocol guide." [Online]. Available: https://kafka.apache.org/protocol.html (Accessed: 2023-04-18).

[44] Microsoft, "[ms-tds]: Tabular data stream protocol." [Online]. Available: https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-tds/b46a581a-39de-4745-b076-ec4dbb7d13ec (Accessed: 2023-04-18).

[45] J. Iyengar and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport," RFC 9000, May 2021. [Online]. Available: https://www.rfc-editor.org/info/rfc9000