


# Supporting Disconnected Operation of Stateful Services Using an Envoy Enabled Dynamic Microservices Approach

Tim Farnham <sup>a</sup>

*Bristol Research and Innovation Lab., Toshiba Europe Ltd, Bristol, U.K.*

**Keywords:** Dynamic Microservices, Service Mesh, Kubernetes, Stateful Services, Envoy, Edge, Cloud, Hybrid Deployment, Continuous and Disconnected Operation.


**Abstract:** Dynamic microservice and service mesh approaches provide many benefits and flexibility for deploying services and setting policies for access control, throttling, load balancing, retry, circuit breaker or shadow mirror configurations. This paper examines extending this to support continuous operation of stateful microservices in hybrid cloud / edge deployment, without loss of data, by permitting disconnected operation and resynchronisation. These are important considerations for critical applications which must continue to operate even during prolonged cloud disconnection and node or client failure. Such service requirements are typical of retail and other scenarios in which services must run continuously, while maintaining a consistent state between cloud and edge service instances. The approach taken and evaluated in this paper exploits a lightweight Envoy proxy within Choreo connect microgateways and Consul service mesh sidecars. Envoy proxies are able to efficiently perform shadow mirroring of requests and support graceful failover, but requires additional functionality to support resynchronisation and recovery from failure that are examined in this paper.

## 1 INTRODUCTION

The use of dynamic microservice architecture and service meshes provide resilience to failures. This is by virtue of the replication of individual service instances across different resources, or availability zones, in a flexible manner, without needing the services themselves to be aware of this. However, in hybrid edge/cloud deployments this raises additional challenges for stateful services due to the need to constantly maintain synchronisation of replicas over the Internet connections. There are two main approaches to achieve this synchronisation, firstly, by replicating the file or database systems supporting the services and secondly by using service shadowing. The latter approach is preferred when there is a need to support services across hybrid environments as there is no need to rely on certain network performance or to restrict what persistent storage and database mechanisms are employed to support the various different services. This is also attractive from the point of view of combining load balancing with failover and tailoring the level of resilience when the requirements and resources change over time.

Recent analysis in (Sampaio, 2019) and (Mendonca, 2020) indicates that using properly configured retry and circuit breaker functionality for stateless microservices can greatly improve performance under transient error and overload conditions. This analysis also helps to define optimal placement and scaling for resource usage. However, little prior research has considered the optimisation of failover for stateful services under prolonged failures. Some prior investigations show that service outage can be reduced when using Kubernetes stateful set controllers with hot-standby services (Vayghan, 2019) and switching the standby to the active state when failure occurs. However, they still result in service outage and do not consider the prolonged node or network failure scenarios which result in disconnected operation and how to resynchronise the state back to the replicated services during recovery without service disruption.

In addition, investigations of optimal placement of microservices (Sampaio, 2019) show that the affinity between services is a vital factor for determining optimal placement but do not consider the failure resilience performance and disconnected

<sup>a</sup> <https://orcid.org/0000-0002-5355-3982>

operation support which require replication and resynchronisation of these service clusters. There have also been prior proposals for edge-cloud computing that does not need to compute optimal placement a-priori (Wang, 2021). This approach mirrors services at the edge and in the cloud and uses the first responses to achieve high performance, but do not consider failure and recovery.

In this paper we address the solution to providing failure resilience and disconnected operation support for stateful microservices using the commonly used Envoy proxy, within API microgateways and service meshes in a dynamic microservice architecture. In particular, the focus is on transactional client application support, which is typical in retail scenarios, such as in-store Point Of Sales (POS). In this use-case, the state information for each application session is independent of other application sessions.

### 1.1 Dynamic Microservices

Dynamic microservices aim to provide resilience through edge/cloud flexibility by permitting microservices to be seamlessly distributed across different environments. One example is in retail services, such as transaction services, where the storing of state information in the applications is not possible as transfer of sessions between different application clients or end devices is needed. Centrally storing the state would also prevent disconnected operation and present a performance bottleneck. Therefore, storing service state in a distributed manner, but not within the application clients, is a fundamental requirement to support tolerance to client device and application failures as well as edge disconnected operation. In such a case replicated stateful services store the state information and permit dynamic transfer of sessions to other client devices without loss of data.

In addition, tolerance to service or node and network failures necessitates replicating service instances between edge and cloud resources. In such a situation the state information needs to be kept in synchronisation as it must not be permitted to have inconsistent state information to facilitate failover and continued operation. Stateful service replication must be considered carefully as the normal load balancing and failover approaches supported in existing dynamic microservices frameworks, such as in Baboi (2019), do not address this specific problem.

#### 1.1.1 Service Mesh

Service mesh approaches provide a common means to control and route access to services in the so-called east-west interactions, that are service to service. In most service mesh solutions the proxy sidecars permit dynamic load balancing and failover support to ensure continued operation, using end-point health metrics, without considering the state data embedded within the service instances. However, when the service instances are stateful, consideration of state limits failover to a subset of shadow service instances that contain the most recent state. Also, to provide resilience against network failures it is necessary that state is distributed and replicated widely, but this then presents a challenge for keeping the state information synchronised. In the extreme case of prolonged disconnected operation, the most recent state may only be stored in one instance and then resynchronised with other instances on reconnection. Traditional load balancing and failover logic becomes less appropriate in these cases as they tend to assume a pool of load balancing and failover endpoints. For instance, within stateless service meshes it would be sufficient to balance load and retry or failover based only on the number of healthy and unhealthy endpoints and treating all end points equally, but with stateful services consideration must be given to the replication and resynchronisation of service state as the top priority.

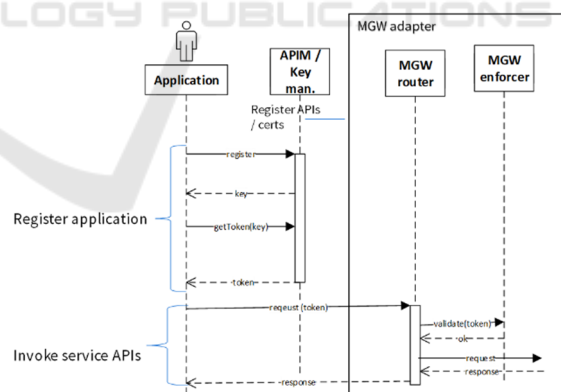


Figure 1: Microgateway operation sequence.

#### 1.1.2 API Gateway

API gateways provide the ingress point for applications to the service meshes for the so-called north-south interactions with services. They primarily act to authorise access to the service meshes but also to manage the ingress traffic through throttling based on quotas and usage policies using the policy enforcement function (see Figure 1). Hence, API

gateways are aware of the individual application sessions. It is also advantageous for reducing overhead that the same proxy implementation is used for both API gateways and service mesh sidecars. By exploiting the implicit or explicit information relating to application session embedded in the access tokens it is possible to route ingress requests based on this authorization bearer token header information. In this manner handover of user sessions from one end device to another can be performed in a transparent manner. Typically, the authorization bearer tokens are in JSON Web Token (JWT) format and can contain subject, audience, issuer, and client claim or consumer key (azp/aud) related information as well as scope and expiry information.

Microgateways are API gateways that target a distributed lightweight distributed deployment paradigm. In this way many gateway instances are used rather than centralised gateway solutions. They support cloud or edge environments and provide the API ingress points that can be integrated within Kubernetes distributed processing environments to provide ease of deployment and management. This can be either as an ingress controller or mesh proxy gateway.

## 2 SERVICE REPLICATION AND RECOVERY

### 2.1 Shadow Mirror

Service shadow mirroring can be performed within microgateway or service mesh proxies to duplicate all incoming requests towards a shadow endpoint or cluster (see Figure 3). This capability has historically been used for background testing prior to canary rollout of new versions of services rather than to replicate service state across different instances. However, for data manipulation using REST write operations, such as PUT, POST and DELETE then the shadowing of requests can replicate the service state. Envoy proxies can permit very efficient shadow mirroring, based on a fire-and-forget approach rather than handling responses. An alternative approach for shadow mirroring is duplicate two or more identical request paths towards different service endpoints and discard the responses from all the endpoints that are slower than the first to respond. This is similar to the proposed approach in (Wang, 2021), which claims to provide performance benefits if the processing can also be aborted on the slower endpoints. However, in our case the processing must continue for reliably

replicating the session state. Therefore, our proposed approach is to set the primary path to the most likely path to fail in order that the secondary path has the state already available prior to the failover.

In comparison, data replication techniques that operate on the database or file system level can be used. However, when operating over the cloud/edge boundary the likelihood of disconnection or performance bottleneck is high. Therefore, the local edge environment is often treated as a separate cluster to the cloud environment. The disconnected operation creates a challenge when the number of replicas is low, which is our scenario of interest. In this case the use of two out of three or four out of six quorum techniques are not possible, which are typical in high performance state replication solutions.

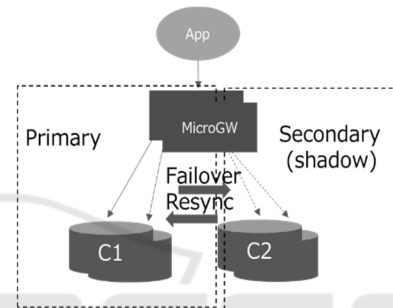


Figure 2: Endpoint clusters for primary and secondary.

#### 2.1.1 Resynchronisation

Resynchronisation occurs to recover state following failure of the network, nodes, applications or services. In the approach in which requests are cached and replayed the process is an identical repeat of the initial application write (i.e. REST based POST, PUT, DELETE) requests. Hence, the main criteria are that the order of requests are maintained and no duplicate requests are made towards the same service instance or cluster and no requests are lost. If these conditions are fulfilled the state will be recovered. An alternative approach to achieve the same goal is to use a replicated database middleware such as Middle-R and C-JDBC or file system solution such as AWS DataSync. However, in such approaches the support for disconnected operation across the edge/cloud boundary relies on capabilities within the specific database technology or file system Persistent Volume (PV) storage classes such as NFS, Lustre, AWS EFS or S3. At present there are at least 124 Container Storage Interface (CSI) drivers supported by Kubernetes, each having different capabilities and implementations. This makes file system specific approaches complex to support. Also, cross-

edge/cloud approaches such as AWS DataSync only support a few CSI with certain limitations, e.g. within snowball edge devices there is no support for symbolic links and files must not be modified during the synchronisation process. In addition, the transfers must be periodically scheduled and so can remain out of sync for long periods. This limits the performance and prevents continuous operation of the services as updates would be lost during local node failure and services would need to be suspended during backup and resynchronisation.

## 2.2 Envoy Proxy

The Envoy proxy is a popular and efficient lightweight implementation written in C++ and configured via yaml files or a gRPC control plane. It provides flexible L4 and L7 filtering capabilities that permit customised handling of traffic. The main advantages of the Envoy approach are that it is self-contained and high performance with a small memory footprint, it has first class support for HTTP/2 and gRPC for both incoming and outgoing connections. It is a transparent HTTP/1.1 to HTTP/2 proxy and supports advanced load balancing features including automatic retries, circuit breaking, global rate limiting, request shadowing, zone local load balancing.

The Envoy proxy is used in many service mesh solutions, such as istio, Kong mesh, Consul and also within API microgateways such as Choreo connect, Apigee edge, Kusk gateway, Ambassador edge stack or Kong within the recent Envoy Gateway initiative, which is aiming to standardise an API for the configuration of Envoy based microgateways. Therefore, it is a logical choice for supporting stateful microservice resilience. However, it does not currently support the necessary caching and resynchronisation functions to permit resilience to prolonged failure. This is an intentional omission as it is a very high performance and lightweight proxy solution which would be sacrificed if it were to support these features. Hence, our proposed solution approach is to combine Envoy with an external caching capability for failover endpoints in order to support resynchronisation without compromising performance.

### 2.2.1 Failover and Load Balancing

The Envoy proxy supports graceful failover based on the proportion of healthy endpoints within a cluster using an overprovisioning factor. Endpoint health is monitored using active probing in a fully

decentralised manner. This provides a means to balance load across healthy endpoints and also support failover at the same time. Such probabilistic failover can be compatible with stateful services if the endpoint clusters do not have common cross-session state information and sticky session load balancing policies used, such as ring hash and maglev (Eisenbud, 2016). Alternatively, a cluster-wide shared PV can be used in which all services in a cluster share the same or a replicated persistent storage resource, such as a distributed file system. Then, a round robin, least request or random load balancing about those instances can be performed. State replication between the edge and cloud is still required for resilience to network failures and so request shadowing between edge and cloud is a simpler solution compared with implementing a shared PV across the edge-cloud boundary that can support disconnected operation. Endpoints should also be replicated in different clusters and availability zones to avoid site or network faults causing a single point of failure.

## 2.3 External Caching

The main aim of the caching proxy functionality is to provide a means to store service requests in order to perform resynchronisation of service state when endpoints recover or become available again. Therefore, it is only required in the failover chain (cluster) endpoints (see Figure 4). The solution used for supporting caching is with a local postgres database. As postgres can support an efficient, high performance and resilient solution it is an ideal candidate even for relatively lightweight edge nodes. At present only the POST, DELETE and PUT REST based API requests are cached as these correspond to the stateful write operations. However, to support other APIs such as websocket or streaming APIs it would be necessary to develop corresponding and more suitable caching approaches. The advantage of focussing on REST based APIs is that the caching solution is very simple. The postgres table used to store the requests consists of two elements, namely the HTTP request header and the payload request body. Hence, when replaying the requests the table is retrieved in sequence and converted back into HTTP requests.

## 2.4 Duplicate Removal

An important issue that arises from the proposed approach (as shown in Figure 4) results from the duplication of requests. As the shadow and the failover endpoints are towards the same service

instances, during failover the same requests are sent to an endpoint via the two different routes. This would be unacceptable in situations in which the service is not able to detect duplication, so to avoid this a duplicate removal function is inserted. Ideally this would be embedded within the Envoy router itself and a solution based on using the aggregate cluster feature of Envoy was explored, but currently this feature is not security hardened and requires trusted upstream and downstream endpoints. Therefore, to avoid increasing the Envoy complexity and need to store state within the proxy an external function was created. The way in which the duplicate requests are removed is by exploiting additional header information inserted by the Envoy router within the Choreo connect MGW. The header parameters used for detecting duplicate requests are:

- x-wso2-cluster-header - primary cluster
- x-envoy-internal - true if in shadow chain
- x-trace-key - unique id for request

The way in which the data is distributed is by using an additional shared control plane which shares this information between the caching proxy instances and duplicate removal functions (see Figure 3). A circular buffer containing these header parameters is maintained in each duplicate removal function which is checked for each incoming request to determine whether to route it. The control plane used for sharing the request header information is based on the Object Management Group (OMG) real-time publish subscribe Data Distribution Service (DDS) protocol. This is a fully decentralised UDP based protocol that permits efficient distribution across the various duplicate removal and cache instances within the clusters. The OpenSplice community edition implementation of DDS was selected for this purpose due to the lightweight and high-performance implementation. The performance of different approaches for supporting DDS are compared by (Kang, 2021) and show that for multi-subscriber use cases, DDS security produces better throughput performance than security provided by virtual networks. The reason is that DDS security only ciphers the data once when transmitting samples to multiple recipients. Hence, DDS multicast is a good choice for the shared control plane. We selected to use the WeaveNet Container Network Interface (CNI) plugin approach to support the DDS reliable multicast on the edge cluster. This is because it uses an encapsulated multicast to unicast approach which makes transmission over wireless or heterogeneous edge networks such as WiFi/Ethernet more suitable and efficient with reliability QoS class.

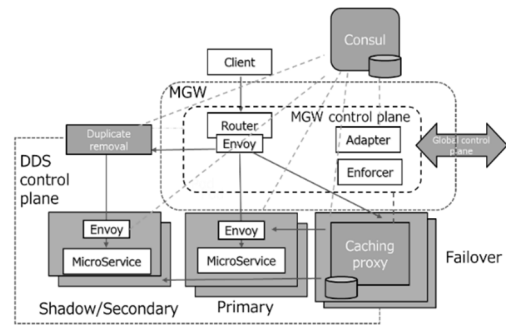


Figure 3: Control and data planes with MGW and Consul.

### 3 IMPLEMENTATION AND EVALUATION

In order to evaluate the proposed approach to stateful service resilience a test environment was setup within AWS VPCs and edge environments. The setup consists of three Kubernetes clusters with a K3S cluster within the edge and two EKS clusters for the cloud data centre services spread across 3 availability zones. A federated consul service mesh is configured across the Kubernetes data centre clusters, representing the edge and cloud, using the proxy default local mode. In this manner the interaction between the edge and the cloud is directed via mesh gateways. The Envoy based Choreo connect Microgateway is used with load balancer as the ingress for application requests.

The Choreo connect Envoy microgateway router deployments provide the ingress point for client application requests. WSO2 API managers (v4.1.0) are used to configure the microgateway routers via adapters that are located in the edge cluster along with policy enforcers. A global AMQP based control plane between adapters and enforcers with the API manager is used for this together with a local gRPC based control plane between the adapter and enforcers and the routers. In this manner the overheads are minimised, while at the same time maintaining high tolerance to individual node failures in order to support continuous operation. Enforcers can cache subscription and access token related information locally to support fault tolerance and improve performance. It is also necessary to have at least 2 adapter and enforcer instances to provide this tolerance to individual node failures.

#### 3.1 MGW Configuration

The Envoy router is configured via the local Choreo connect MGW xDS / gRPC control plane from the

adapter. The adapter retrieves the API definitions from the API manager and consul server using the corresponding REST APIs and creates the configurations for the routers. Each router is configured identically for each instance. For each incoming request the header authorisation bearer token is verified using the Choreo connect enforcer over the gRPC control plane. The enforcer retrieves the subscription information relating to the token claim from the API manager. The subscription information contains the accessible APIs as well as corresponding quota and scope related information. This permits the enforcer to decide whether to permit routing of the request. The API usage data is recorded by the enforcer and is updated globally with all enforcer instances via the API manager and to the optional Choreo analytics service.

In addition, the Choreo connect adapter has been updated to include the shadow mirror policy configurations. This permits the transparent shadowing of requests towards the shadow endpoints, which is via the duplicate removal function instances. The adapters also configure the graceful failover cluster policies based on the default overprovisioning factor of 1.4. The primary endpoints being the priority level 0 and the secondary being priority level 1. The HTTP based health checking option is configured for each cluster. In this manner the failover from primary to secondary starts when there is less than 72% of endpoints in the P0 cluster in a healthy state.

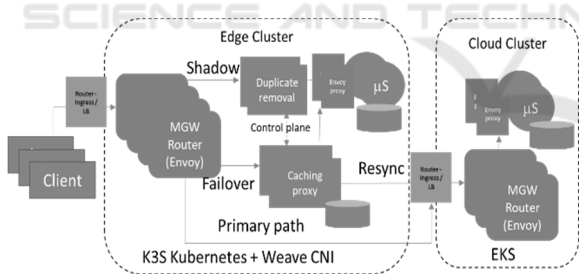


Figure 4: Configuration of edge / cloud cluster routing.

### 3.2 Caching Proxy and Duplicate Removal

The caching proxy used for the failover path is an asynchronous streaming reverse proxy implementation based on the Apache non-blocking NIO libraries. This is a high-performance solution that can scale to support massive parallel session handling. It also permits access to the request header information necessary for passing to the duplicate removal function and to the complete request body for request caching. It is integrated with a postgres

database in the same pod in order to provide the local cache storage which uses a local PV for resilience to service or node failures and restarts. The cache solution also performs the resynchronisation to the remote endpoint when they return to the healthy state.

The same approach is used for the duplicate removal function implementation, but without the caching of requests. As the shadow mirroring performed by the Envoy proxy is fire-and-forget, upstream issues in the shadow chain are only detected by health check probes.

## 3.4 Performance Evaluation

### 3.4.1 Microgateway

The enforcer within the microgateway is the most computationally intensive element of the proposed approach. It provides the entry point for authorising application requests and to secure the interactions within the service mesh. The shadow mirroring itself is a low overhead approach with negligible additional latency to the primary path. The performance of the Choreo connect microgateway is summarised in Figure and indicates that around 500 requests per second can be handled per vCPU instance. In this configuration the API invocations utilise two levels of load balancing. The first is the K3S or EKS service load balancer followed by the Envoy router load balancer, which also performs authorisation bearer token validation, but no rate limits are placed on individual applications or APIs in the test configuration. There is a linear scaling in resource utilisation until the xDS control plane limits performance. This is shown in Figure and indicates that around 12 router instances can handle almost 4000 requests/s from 3200 parallel client sessions.

These results are also consistent with observation by (Johansson, 2022), which indicates that Envoy performs well with up to ~300 parallel sessions per instance. Beyond this level other load balancing

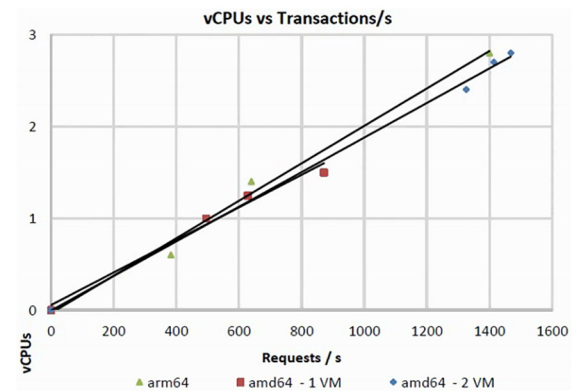


Figure 5: CPU utilisation of Choreo connect microgateway.

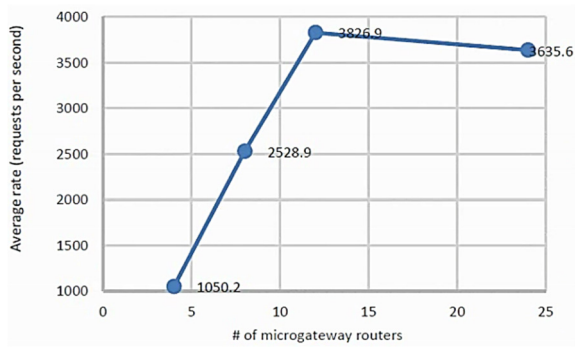


Figure 6: Microgateway throughput with round robin LB.

solution approaches, such as NGINX and HAProxy can perform better. Therefore, the two-level load balancing is preferable.

### 3.4.2 Service Mirroring

In terms of end-to-end performance of the mirrored services it is possible to see from Figure that there is a linear relationship between number of nodes in each cluster and the requests per second that can be handled. When mirroring is performed in a separate cluster the performance is reduced due to the duplicate removal and processing of shadow path requests in the mirror chains. This results in a 17% reduction for request throughput per node compared to the no mirror case.

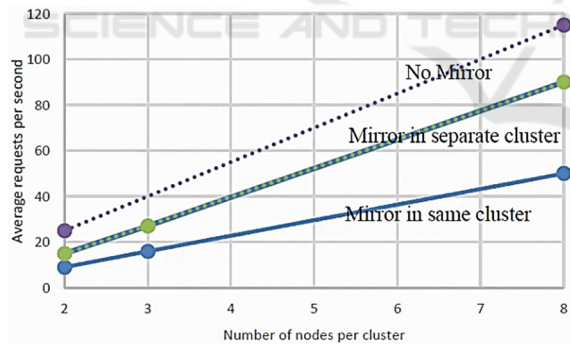


Figure 7: Mirror performance for large REST POST requests (7kByte request + 35kByte response).

However, if mirroring occurs within the same cluster, then the performance reduction is around 55% due to doubling of the number of service requests handled per node in the same cluster as well as duplicate removal checking.

### 3.4.3 Service State Recovery

The recovery resynchronisation processes using file system or database replication ensure consistency

with ACID transaction handling mechanisms. This involves complex transaction commit and consensus procedures such as the classical quorum replication approaches, such as the minimal two out of three selection. This can allow for disconnected operation but impacts on performance of all updates, when using synchronous replication, and in the case of lightweight edge deployments can be too complex and resource intensive. It is therefore common that only the master replica instance performs write operations and the replicas receive the updates from the elected master. When failure of the master or network partitioning occurs it is important that a new master is elected but can lead to an outage period. Also, during recovery all replicas synchronise with the new master, but this may not be possible if network partitions each contain a master replica node. For instance, during disconnected operation when both the edge and cloud environments contain a master replica node. In this situation there is a potential for conflicting state and to prevent this it is necessary to ensure that this situation cannot happen.

Two issues can arise from this, the first is that the requests may have succeeded after the failover to the secondary. Secondly, the replay of the requests occurs at a later point in time and so if the services have a time dependent processing behaviour the result applied in the database may be different. These issues can be avoided if the request duplicate detection mechanisms are implemented in both the shadow and failover paths, within the caching proxies, to remove duplicate requests. Secondly, primary path responses codes can be cached, near to the services, in addition to requests and cross-checked for request duplication and the service times dynamically adjusted to emulate the time of the original requests during replay.

The recovery performance is an important consideration for the viability of the approach. To evaluate this both network disconnection and node rebooting tests were used. Replication middleware methods for databases, such as Middle-R and C-JDBC are most similar to the proposed approach, but suffer from a prolonged increase in response times when failure occurs even under modest (i.e. < 50%) loads. In contrast request failover is at the API microgateway level only relies on a request or health check timeout to the service endpoint. This leads to a faster failover and recovery process. For instance, Middle-R requires 60 seconds and C-JDBC around 180 seconds (Dhamane, 2014). In contrast the API microgateway failover timeouts can be typically configured to respond in <2 seconds and immediately have the same prior performance. A 20 millisecond

delay is inserted into the shadow chain to permit duplicate request removal.

The resynchronisation recovery process has been evaluated. In the case of link disconnect the time of failure is 10 seconds and the recovery occurs at 95ms per request with overall throughput of 13.1 requests per second over a one minute test duration. The maximum request latency is 1.7s in this case. For the node reboot failure case the recovery time is marginally faster than the network disconnection, but fewer requests need to be buffered and the corresponding request replay time is longer at 235ms. This is due to the node taking some time to return to normal operation state. The maximum latency is also 1.7s in this case.

## 4 CONCLUSIONS

In hybrid edge/cloud microservice deployment it is necessary to consider the resilience of stateful services carefully in order to fully support continuous operation without performance degradation. The proposed approach, using Envoy based proxies, is a promising method of permitting replication and recovery of service state, by using shadowing, without requiring performance limiting distributed file system storage or database replication solutions across the edge cloud boundary. The approach proposed and evaluated within this paper has many advantages as it does not require services or underlying storage solutions to be modified and does not impact significantly on throughput and latency performance. For instance, 17% reduction in throughput was observed for large update requests and negligible additional latency is introduced in the primary request paths. However, it is evident that some limitations arise and have been mostly overcome or avoided. In particular, the independence between different client application sessions is assumed. In which case, the use of sticky session load balancing such as Maglev or hash tables can avoid the need for a cluster-wide shared PV storage which can be difficult or undesirable to support. Also, avoiding the use of cross-session and service state that would create conflicts between parallel sessions leading to complex state synchronisation. In addition, it is necessary that updates do not have time dependent logic (or the original time of request need to be emulated in the service). This approach has been evaluated in the context of retail services which support transaction handling, inventory management and promotion services. These require consistent and replicated service state data, across the edge/cloud,

without loss to permit continuous operation, with low overhead, even under node failure and cloud network disconnect events.

Further work is needed to evaluate the improvements that are possible by integrating the duplicate removal and caching functionality into the Envoy proxies, rather than keeping them separate.

## REFERENCES

- Sampaio, A., Rubin, J., Beschastnikh, I. and Rosa, N. (2019) Improving microservice-based applications with runtime placement adaptation. *J Internet Serv Appl* 10, 4 (2019). <https://doi.org/10.1186/s13174-019-0104-0>
- Vayghan, A. L., Saied, M., Toeroe, M. and Khendek, F., (2019) Microservice Based Architecture: Towards High-Availability for Stateful Applications with Kubernetes, *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS), 2019*, pp. 176-185, doi: 10.1109/QRS.2019.00034
- Mendonca, N., Aderaldo, C., Camara, J. and Garlan, D. (2020) Model-Based Analysis of Microservice Resiliency Patterns, *In 2020 IEEE International Conference on Software Architecture (ICSA), 2020*, pp. 114-124, doi: 10.1109/ICSA47634.2020.00019.
- Wang, Y., Xia, Y., Zhang, Y., Melissourgou, D., Odegbile, J., Chen, S. (2021) A Full Mirror Computation Model for Edge-Cloud Computing, *In IC3 '21: 2021 Thirteenth International Conference on Contemporary Computing (IC3-2021) August 2021 Pages 132–139* <https://doi.org/10.1145/3474124.3474142>
- Baboi, M., Iftene, A., Gifu, D. (2019). Dynamic Microservices to Create Scalable and Fault Tolerance Architecture. *In Procedia Computer Science, Volume 159, 2019, Pages 1035-1044, ISSN 1877-0509*
- Dhamane, R., Patino, M., Valerio, M., Peris, R. (2014) Performance Evaluation of Database Replication Systems. *In Proceedings of the 18th International Database Engineering & Applications Symposium IDEAS, July 2014 Pages 288–293* <https://doi.org/10.1145/2628194.2628214>
- Kang, Z., An, K., Gokhale, A., Pazandak, P. (2021). A Comprehensive Performance Evaluation of Different Kubernetes CNI Plugins for Edge-based and Containerized Publish/Subscribe Applications. *Conference: 9th IEEE International Conference on Cloud Engineering 10.1109/IC2E52221.2021.00017.*
- Johansson, A. (2022). HTTP Load Balancing Performance Evaluation of HAProxy, NGINX, Traefik and Envoy with the Round-Robin Algorithm (Dissertation). Retrieved from: <http://urn.kb.se/resolve?urn=urn:nbn:se:his:diva-21475>
- Eisenbud, D. E., Yi, C., Contavalli C., Smith, C., Kononov, R., Mann-Hielscher, E., Cilingeroglu, A., Cheyney, B., Shang, W., Hosein, J. D., (2016) Maglev: A Fast and Reliable Software Network Load Balancer, *In 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16), USENIX Association, Santa Clara, CA (2016), pp. 523-535.*