Écharde: enabling transparent and modular sharding for Cloud and Edge support services

Guillaume Rosinosky
Post-doc@Cloud Large-Scale Computing team (Pr. Etienne Rivière)
UCLouvain, Belgium

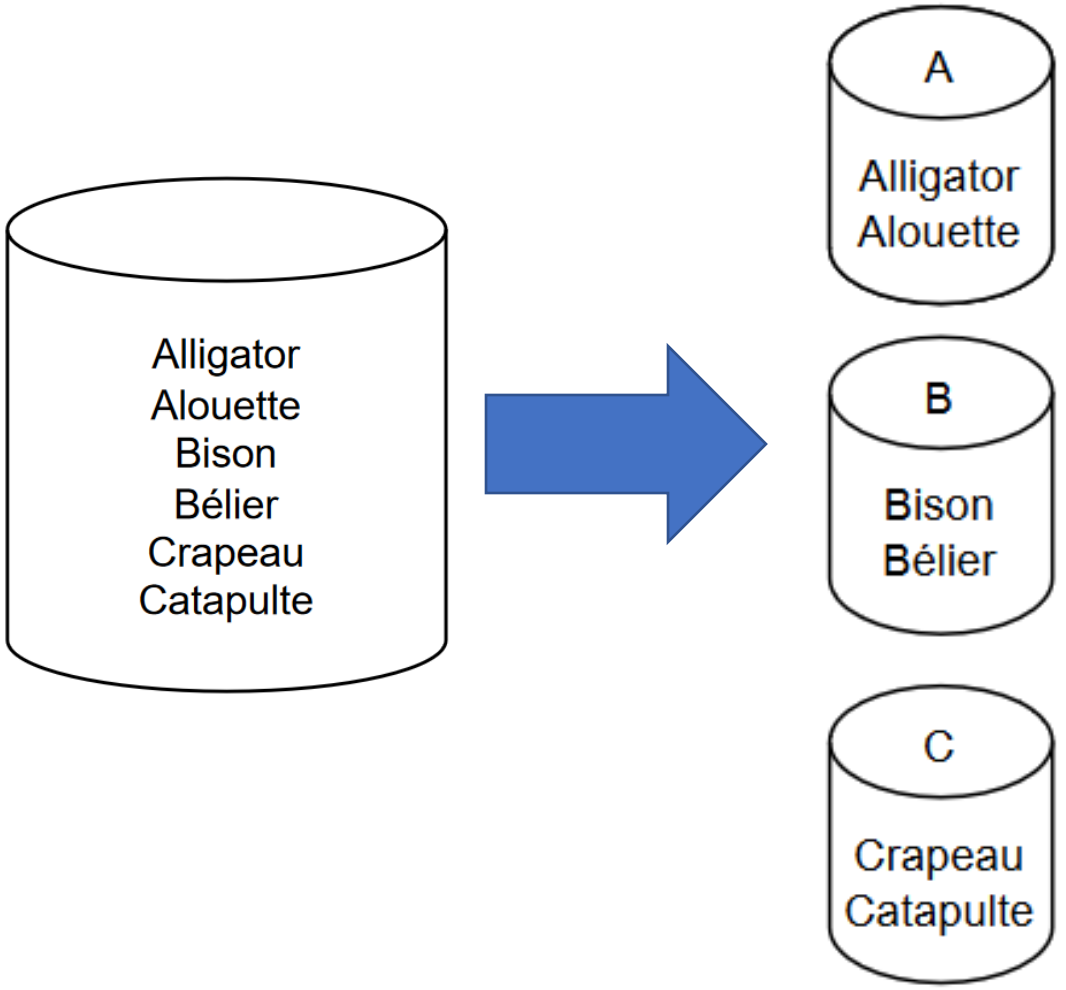
UCLouvain

Intro

- Stateless services can scale easily, but what about the services storing/transmitting state?
- Support services
  - Persistence (databases),
  - Communication (pubsub systems: Kafka, MQTT, ActiveMQ),
  - Synchronisation(Zookeeper, etcd …)
- Can become a bottleneck, especially in a distributed setting
  - Multi cloud, Edge/Fog, etc.
  - (latency, data transfer cost...)

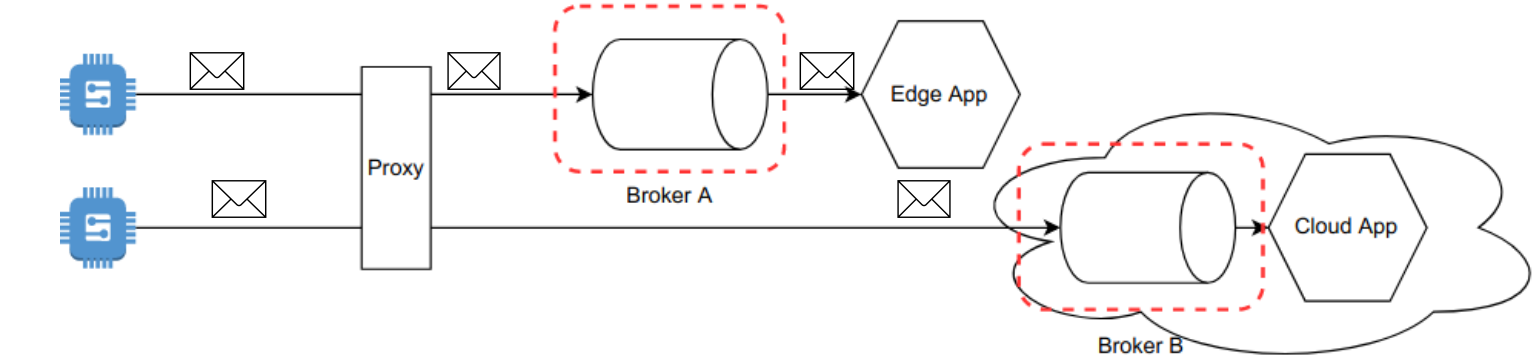
Sharding

- Split processing in multiple partitions (shards) across multiples instances
- Each shard is a single source
- Used in databases, blockchains, pubsub, etc.
- Initially scalability, but also data sovereignty, data locality, multi-tenancy
- Routing based on content of request (shard key) - content-based routing



Sharding (messaging systems)

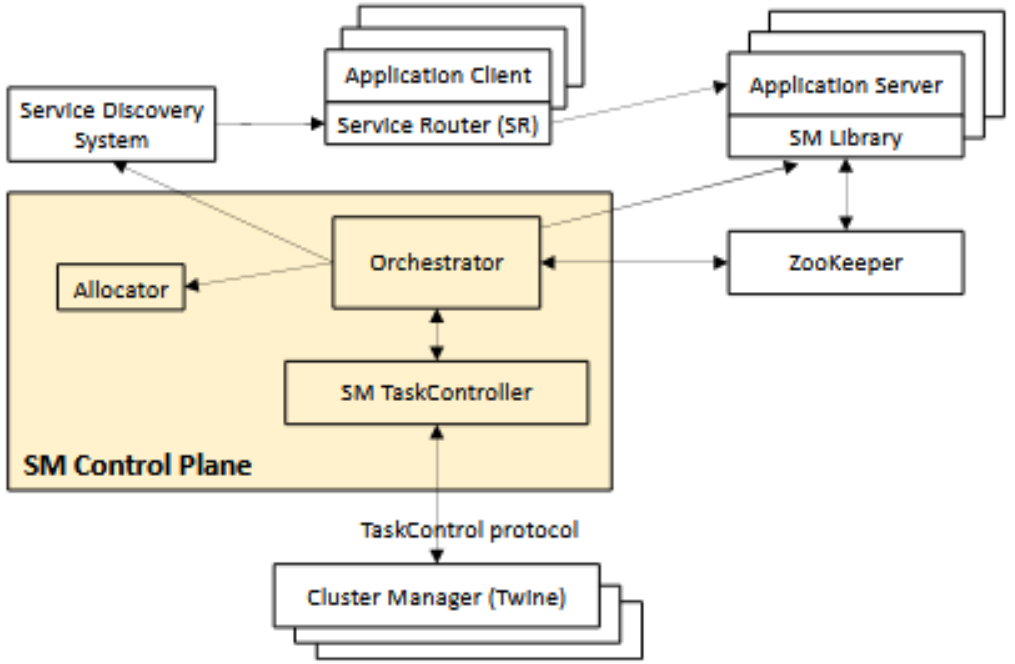
- Messaging systems
  - Mailbox used for IPC or interthread communication classified by topic
  - Most of the time, publish-subscribe pattern, usage of broker as an intermediary node
  - Examples: Kafka, Redis PubSub, MQTT, …
- Sharding could be made on topic, source of the message, payload …



Sharding as a transversal feature

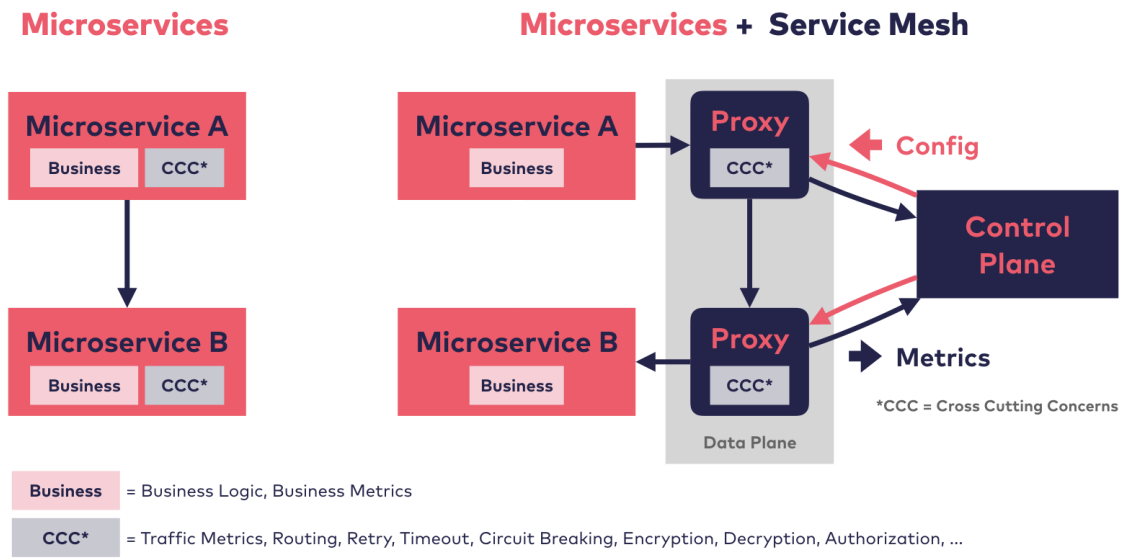
- « Manual » development can become complex to handle
- « Generic » control plane-based approaches
  - General purpose sharding frameworks:
    - FB's ShardManager / Google's Slicer
  - Solutions are *proprietary,* and need *specific libraries, and control plane*
- *Can we provide a low-code approach for support services sharding without specific client/server and control plane?*

- Example: Facebook's Shard Manager (SOSP'21)



Service meshes to the rescue

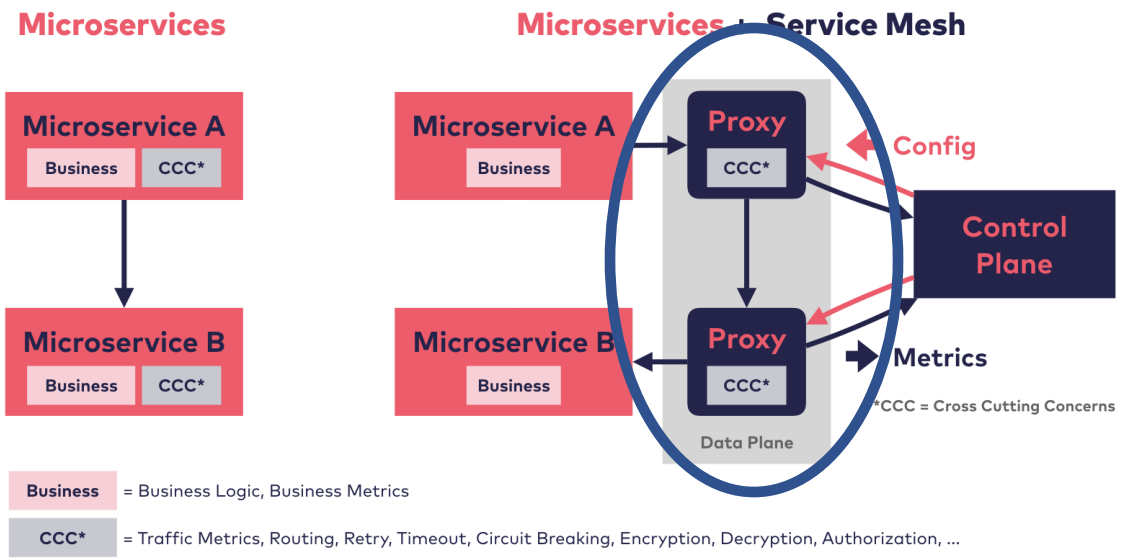
- Recent approach to manage communication between microservices without codebase modifications
- Features:
  - Mostly HTTP observability, security, encryption, routing, …
- Based on proxy intercepting communications + control plane
  - Integration with container orchestrators' control plane (Kubernetes / Docker Swarm)
  - Changes at runtime
- Increasing popularity since 2017 (Istio + Linkerd)
- **RQ1: Can we use service meshes so we can do sharding for support services?**



https://servicemesh.es/
CC InnoQ

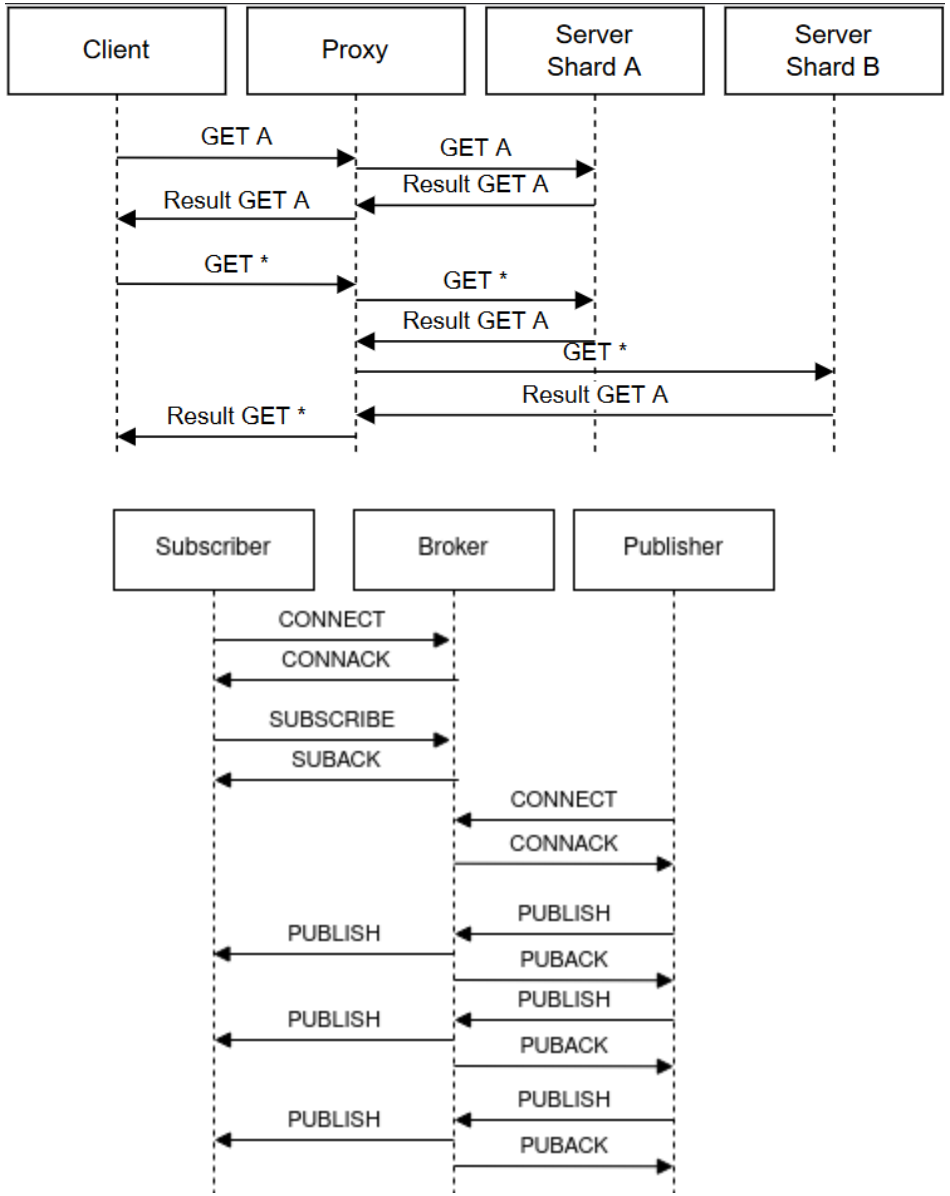
Service meshes capabilities and limitations

- Proxy (reverse proxies)
  - Routing capabilities: HTTP, TCP, UDP
  - Content-based routing
    - All: HTTP
    - Some rare TCP plugins but not for all, and dependant on proxy
  - Envoy, Mosn, Linkerd-proxy, Nginx,…
- **RQ2: can we enhance proxies with a modular & low code approach so they are able to enact CBR for support services?**



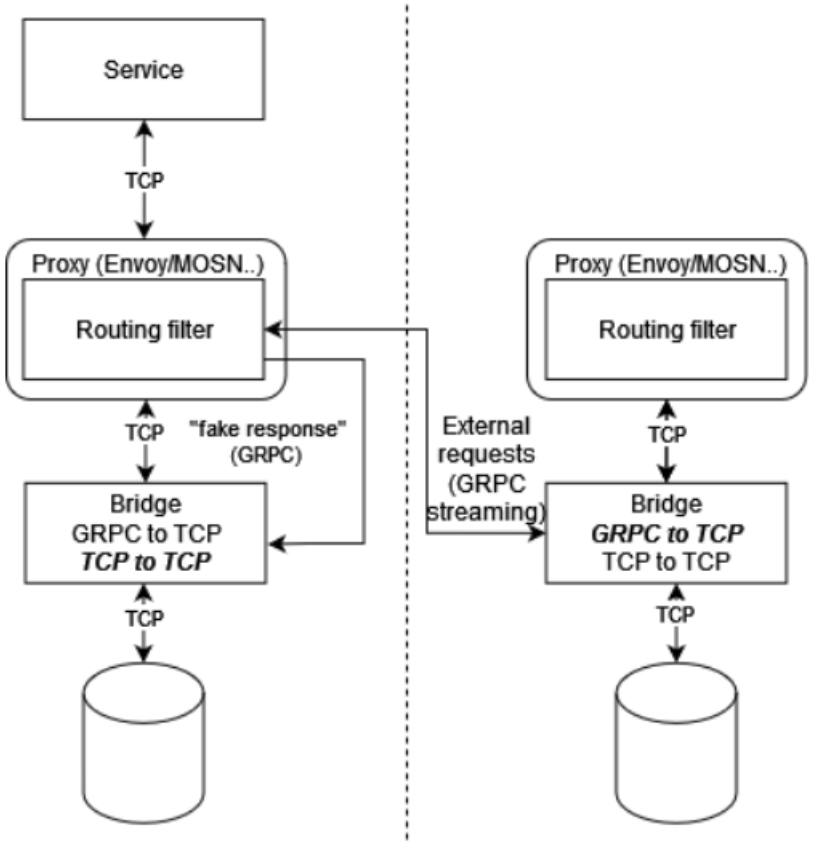
Our proposition

- Modular plugin able to route support service requests
  - But each protocol is different!
  - … we don't need to recode them completely.
    - Request/response, pub/sub patterns common between low-level protocols
    - Usage of serialization/deserialization when needed (widely available serde libraries)
    - Split / merge requests
    - Some queries do not have the sharding key ? Broadcast (slow) or replay.

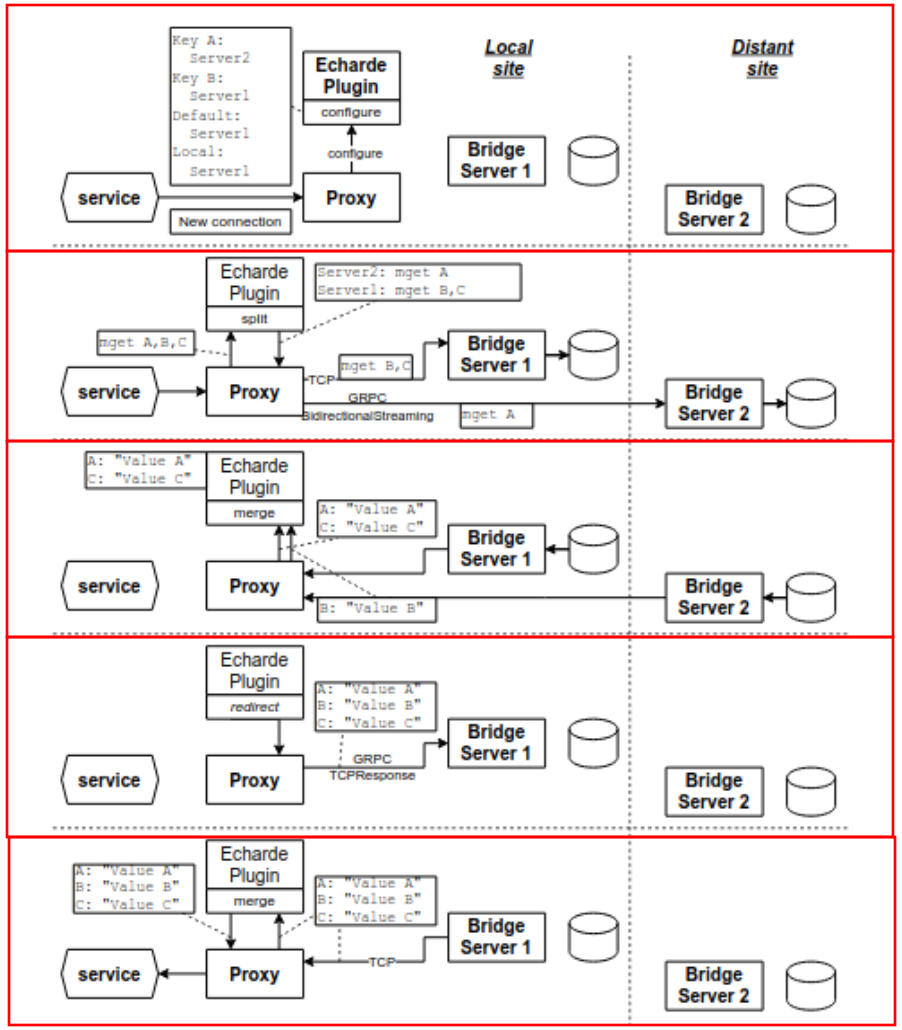


Echarde

- Modifying the proxy codebase is not very modular...
- Usage of WebAssembly *Proxy-Wasm*:
  - Available on Envoy, Istio, MOSN, OpenResty (Nginx)
  - Can be modified / configured at runtime
  - Features :
    - Read/replace client/server sent TCP packets, HTTP headers/payload
    - Asynchronous HTTP/GRPC client
  - Missing features:
    - add way to initiate new client->server and server->client TCP packets
    - HTTP/GRPC server



Example of Redis MGET orchestration



Écharde: enabling transparent and modular sharding for Cloud and Edge support services

RQ2: integration in service meshes

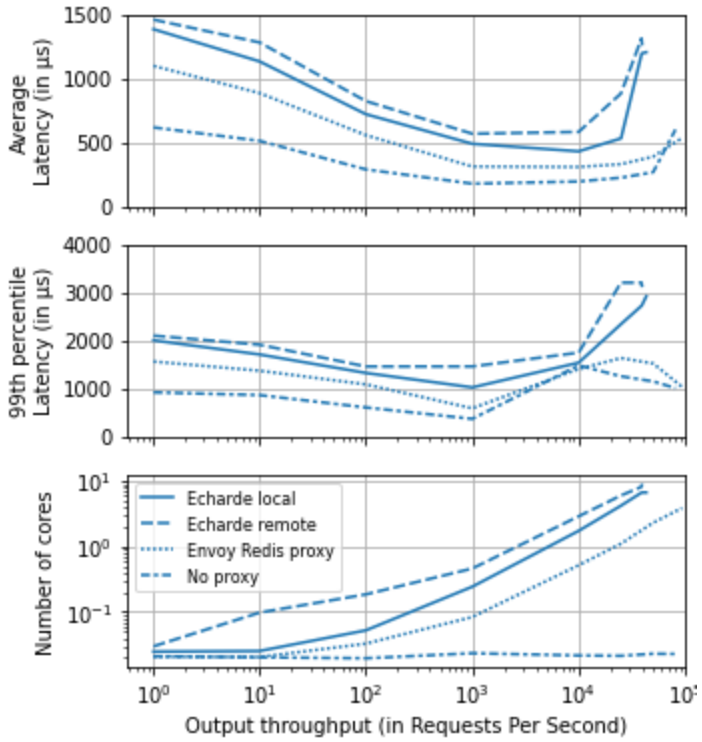
- Our approach can be naturally integrated with service meshes
- Example configuration service with Istio:
- TCP/GRPC bridge: sidecar of the support service

```
apiVersion: networking.istio.io/v1alpha3
kind: EnvoyFilter
metadata:
  name: echarde-outbound-tunnel
  namespace: istio-system
spec:
  workloadSelector:
    labels:
      tier: redis-routing
  configPatches:
  - applyTo: FILTER_CHAIN
    match:
      context: SIDECAR_OUTBOUND
      listener:
        portNumber: 6379
    patch:
      operation: INSERT_FIRST
      value:
        filter_chain_match:
          destination_port: 6379
        filters:
        - name: envoy.filters.network.wasm
          typed_config:
            "@type": type.googleapis.com/envoy.extensions.filters.network.wasm.v3.Wasm
            config:
              name: "shard_router"
              root_id: "shard_router"
              configuration:
                "@type": type.googleapis.com/google.protobuf.StringValue
                value: |
                  workload_type: redis
                  type_proxy: downstream
                  default_cluster: redis1-master
                  local_cluster: redis1-master
                  routing_table:
                    user1:
                      type: key
                      cluster: redis2-master
                    alert:
                      type: value
                      cluster: redis2-master
```

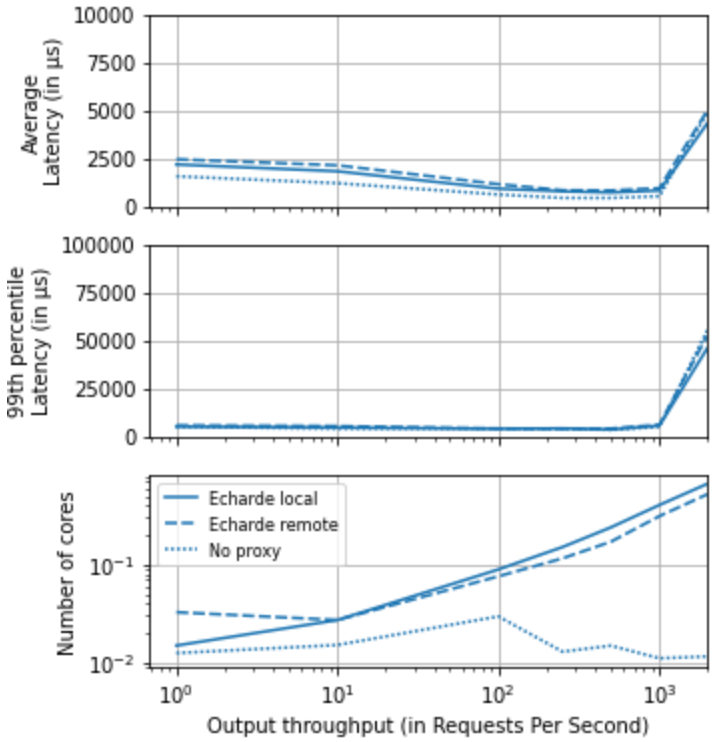
Microbenchmarks: WIP

- Targets: Redis, InfluxDB, MQTT, Kafka
- Microbenchmarks
  - **Raw overhead**
  - **Breakdown**
  - Distributed (*WIP*)
- Docker Swarm
  - 3 nodes: injector, proxy, database/broker+bridge
  - YCSB (Redis), YCSB-TS (InfluxDB), Berserker (MQTT, Kafka)

Overhead (payload: 1kb)

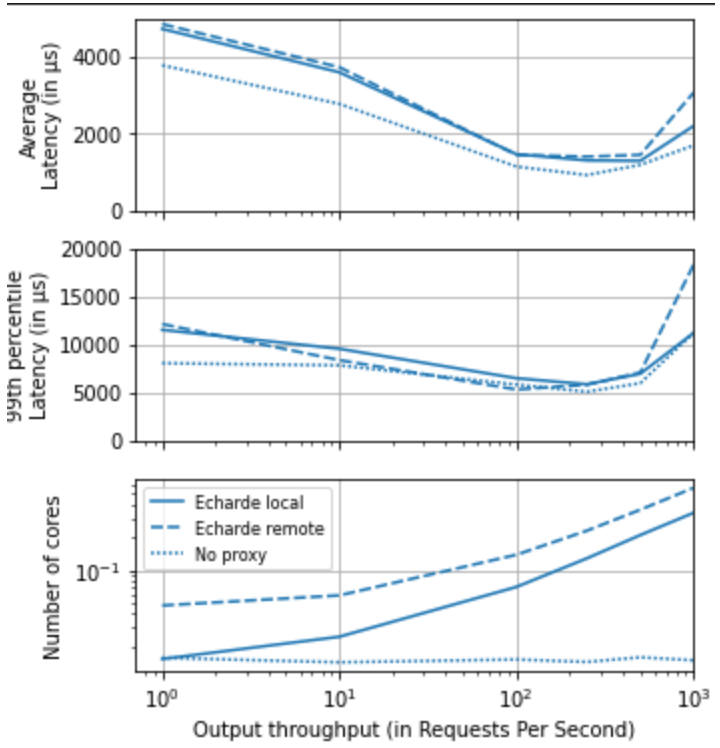


Redis

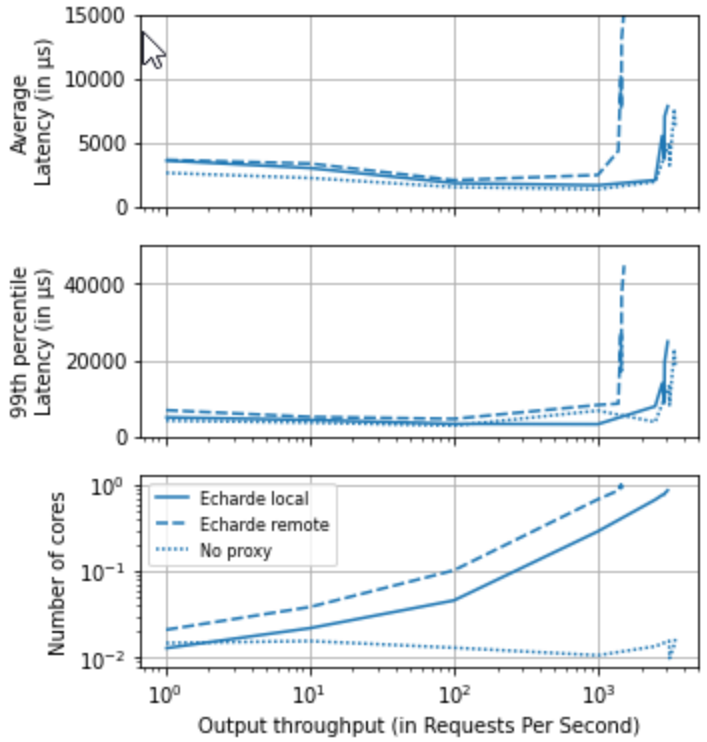


MQTT

Overhead (payload: 1kb)



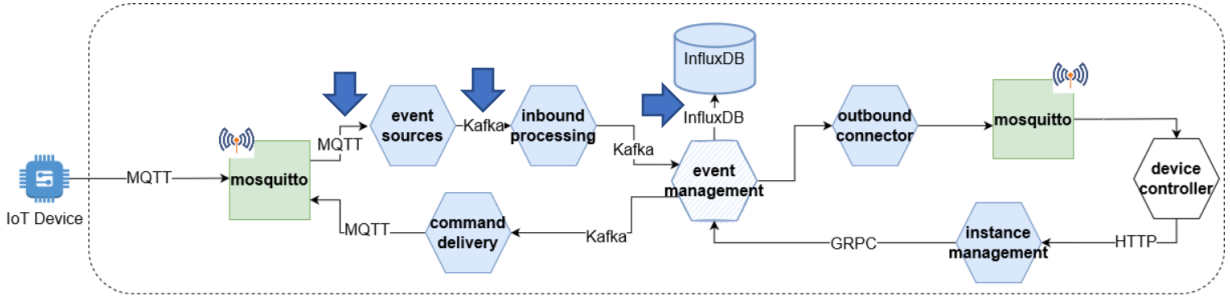
Kafka

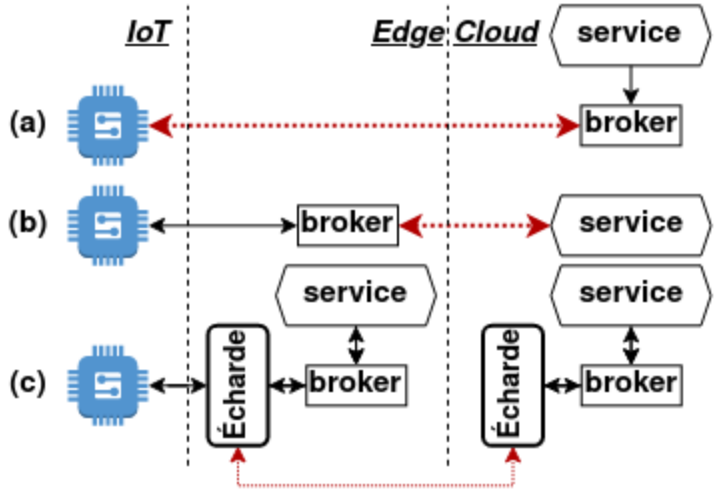


InfluxDB

Macrobenchmark: WIP

- Macrobenchmark (*WIP*)
  - Distributed Cloud/Edge IoT scenario
    - Goal:
      - minimize response time for alerts
      - minimize data transfer to Cloud
  - Platform: Sitewhere µservice application using MQTT, Kafka, InfluxDB
- Infrastructure:
  - Kubernetes on Azure DC
    - "Cloud": Ireland DC
    - "Edge": Paris, Zurich, Frankfurt, low-end machines
  - Several hundreds of IoT devices (IoT-lab) (Grenoble, Saclay, Strasbourg)





Conclusion

- Modular sharding approach for proxies and service meshes
  - Implemented:
    - PubSub: MQTT, Kafka
    - Database: Redis, InfluxDB
- 300 LoC (Redis), 140 LoC (MQTT), 170 (InfluxDB), 320 (Kafka)
- Overhead (no routing): max 1ms, ~ 0.8 CPU per 1K RPS
- Experiments in progress

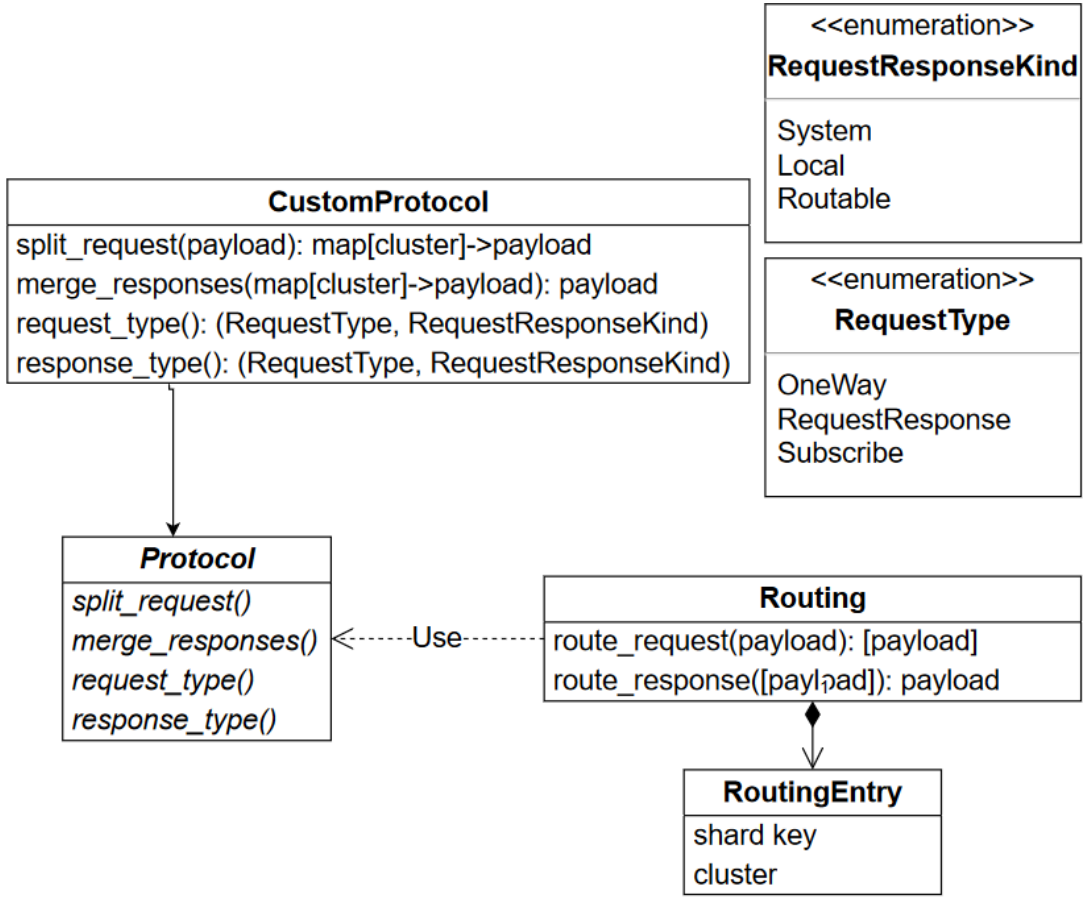
- Future work:
  - Usage of method for more features: caching, HA, consistency, security
  - Live migrations
  - Network offloading (smart NIC)
  - …
  - More generally, use mesh for transversal computing

SotA

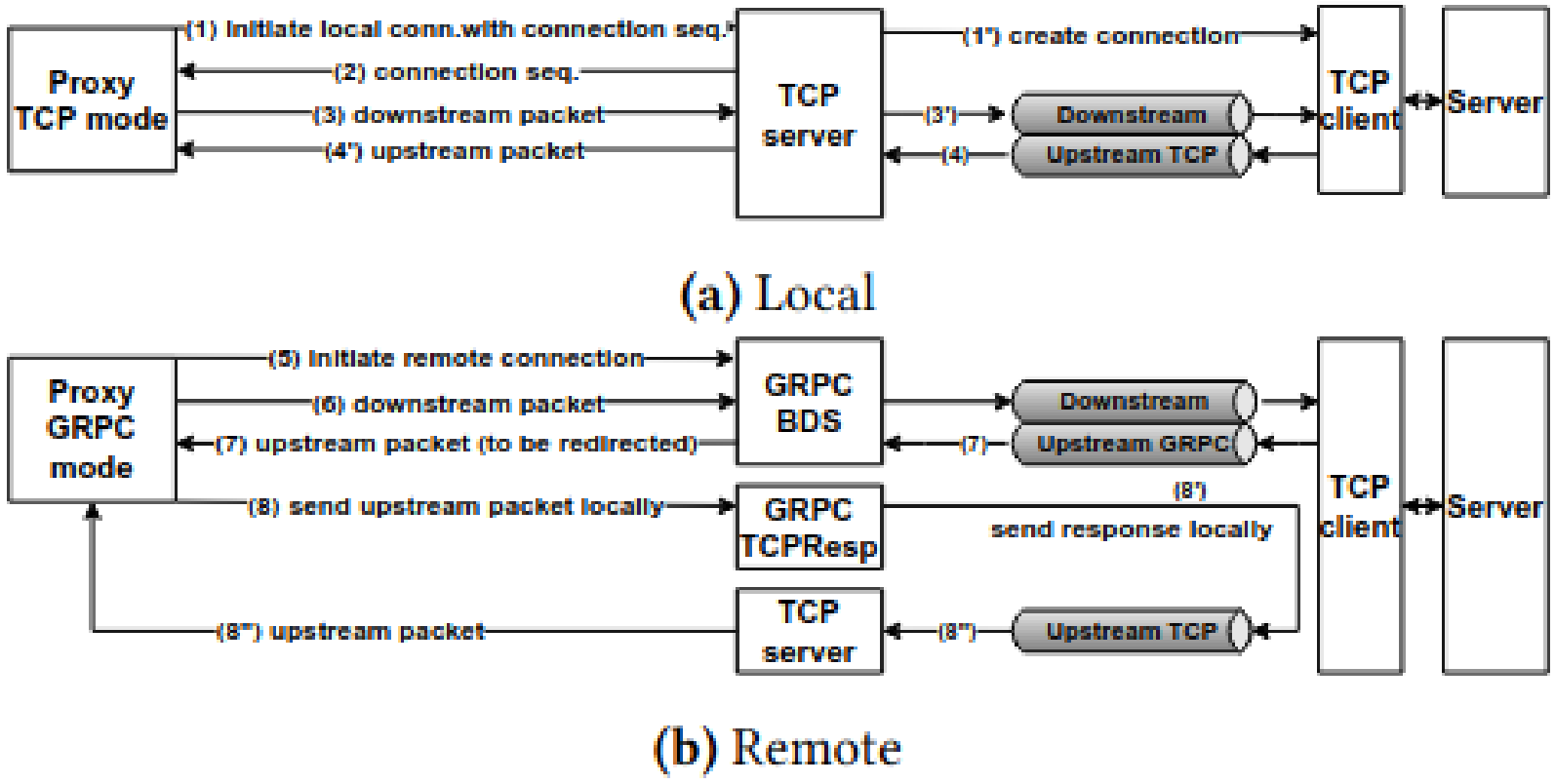
System	Control plane	Service mesh integration	Support	Extensibility	Container orchestrator integration	Sharding location	Low-code approach	Plugability
Slicer	✓	✗	DB	✓	✗	C&S	✗	✗
Shard Manager	✓	✗	DB	✓	✗	C&S	✗	✗
ShardingSphere	✓	✗	DB	✓	✗	C or P	✗	✗
Gizzard	✗	✗	DB	✓	✗	P	✗	✗
Bumblebee	✓	✓	HTTP	✗	✓	P	✓	✓
Cheops	✓	✓	HTTP	✗	✓	P	✓	✗
Aeraki Mesh	✓	✓	DB, Msg	✓	✓	P	✗	✗
Écharde	✓	✓	**Any**	✓	✓	**P**	✓	✓

Generic RR/PubSub library

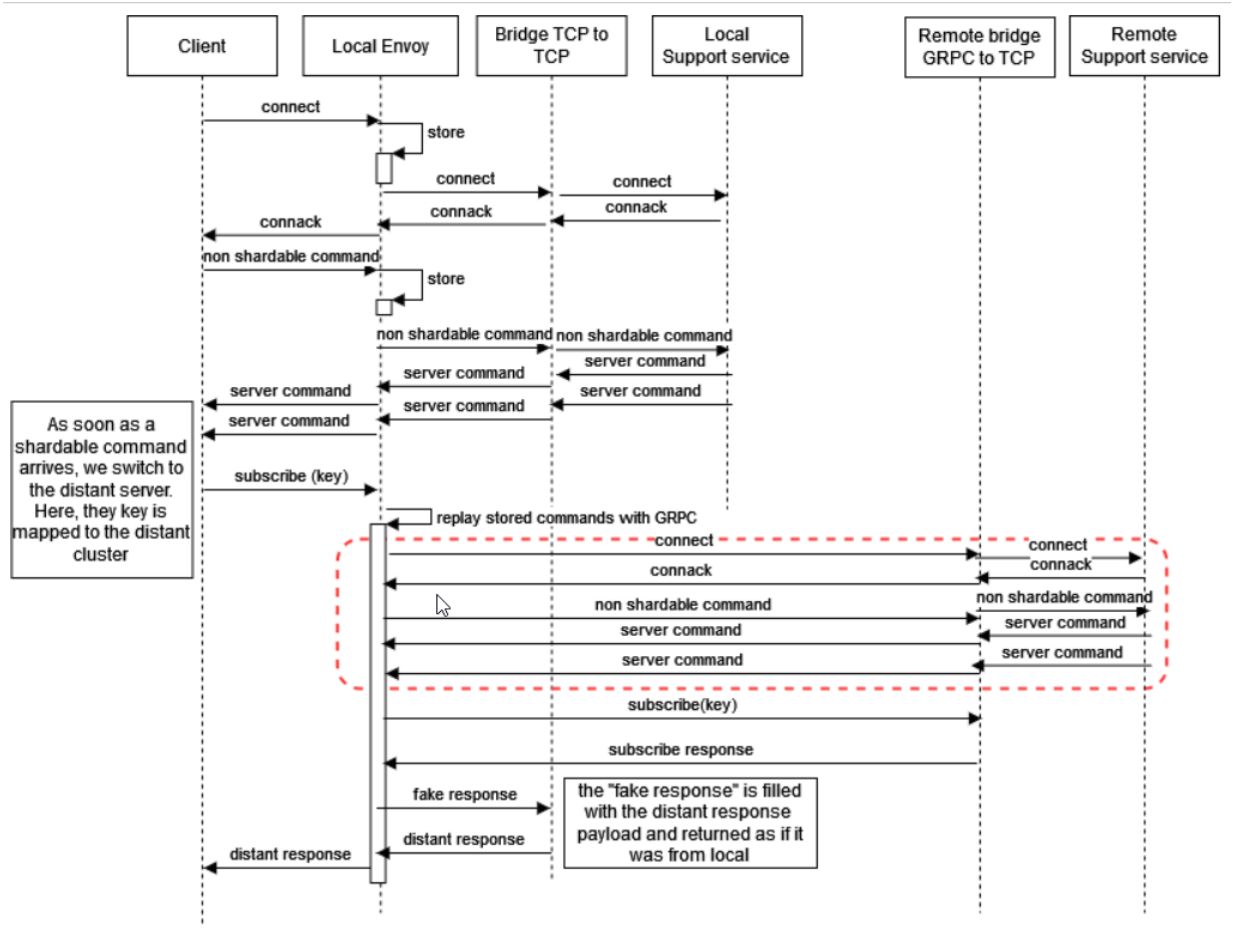
- Routing component
  - Able to orchestrate known protocols
    - Database (Request/response)
    - Messaging (Pub/sub)
  - Client->Server: takes a TCP packet, returns a map cluster->packet
  - Server->Client: takes one (or many packets), return a packet
- Protocol component
  - Usage of low-level libraries to serialize/deserialize TCP payloads (available in most languages)
    - Identify the type of request/response
    - Use appropriate shard key
  - Nagle/TCP packet split managed



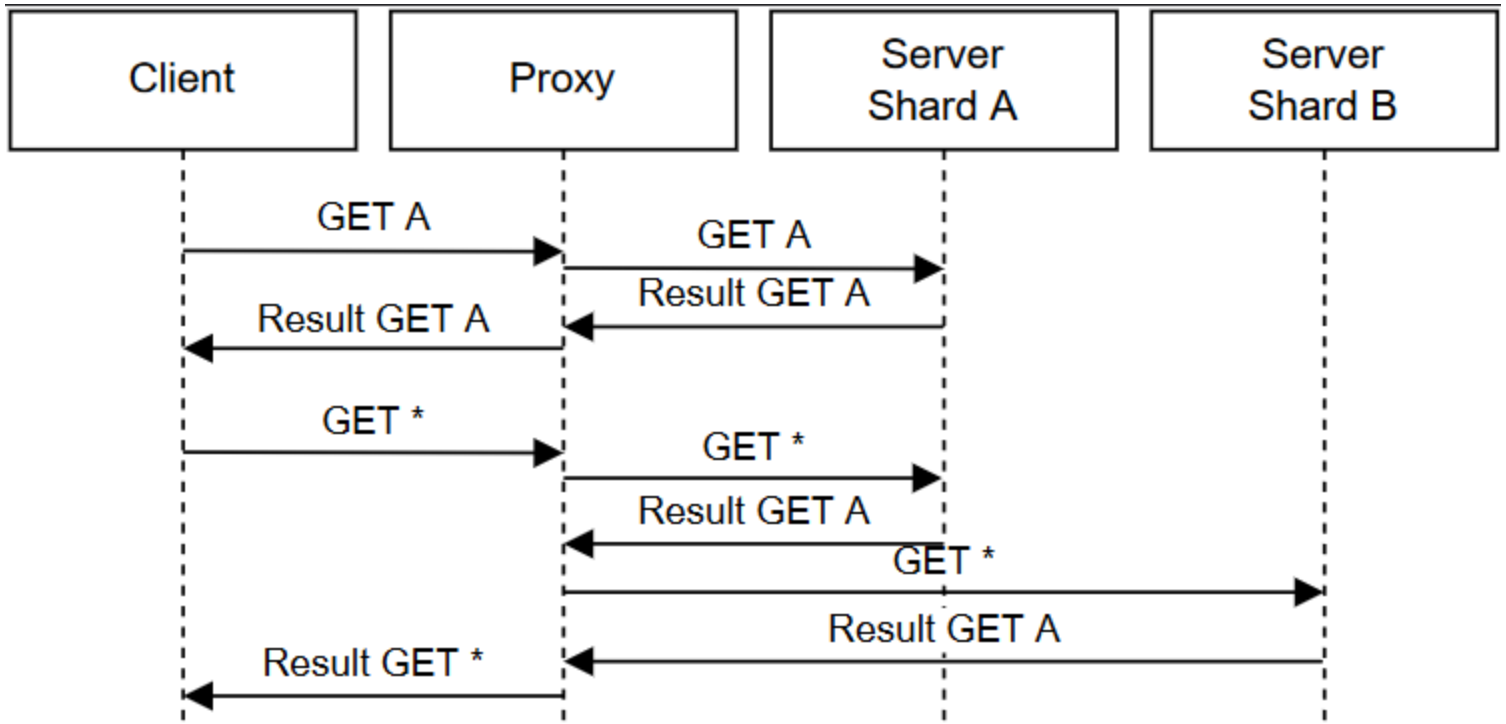
TCP/GRPC bridge



Non shardable commands

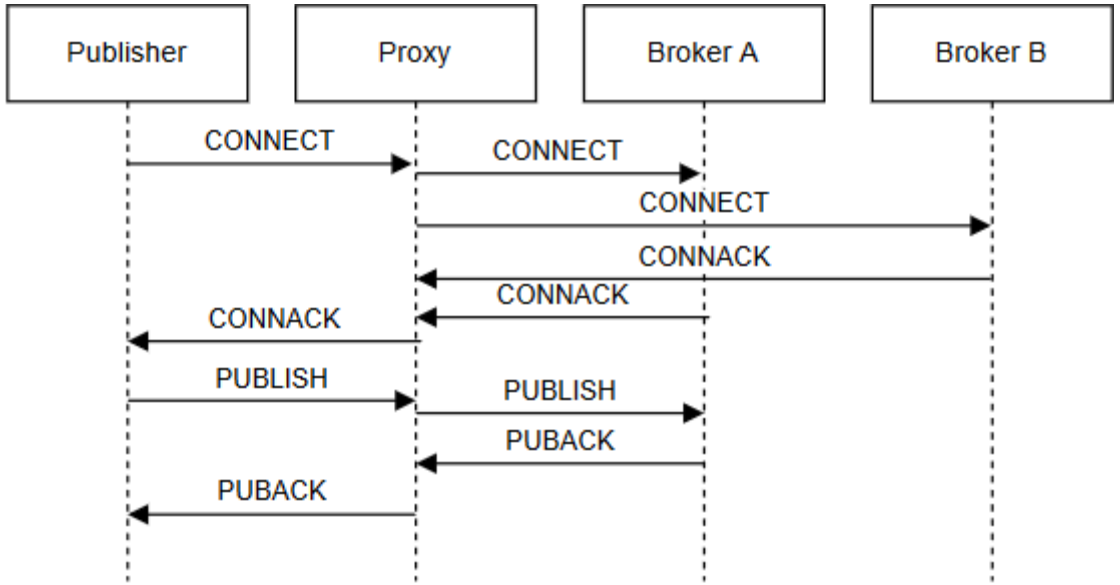


RQ2: Harness database protocol: Redis (RR)



RQ2: Harness messaging protocol: MQTT

- Publish (1 Request / 1 Response)



- Subscribe (1 Request / n Responses)