Qbox: A Saga Execution Coordinator For The Service Mesh

Akshat Mahajan
Brown University

Changhao Wu
Brown University

Abstract

We present Qbox, a network proxy that doubles as a coordinator for distributed sagas. Qbox significantly reduces operator overhead by eliminating the need for centralized publish/subscribe systems in distributed saga implementations. Qbox natively supports nested sagas, enabling federated topologies for large-scale microservice architectures, as well dynamic-valued saga messages, enabling drop-in support for RESTful architectures. No client library is needed to invoke a saga. We outline Qbox's unique design and architecture, and present our comparative findings with RabbitMQ for a variety of service topologies.

1 Introduction

Modern software architectures are typically composed of *microservices*: lightweight, independent services that are dedicated to ownership of a single business task. A key challenge that migrations to microservice-oriented architectures encounter is the increased complexity of implementing business operations over these services. For example, the operation of processing an order must not only monitor a payments service and an invoice service for completion, but also purge all references to the order should one of these services succeed but the other fail.

Sagas are a popular technique to implement such all-or-nothing business logic in microservice contexts. The saga protocol uses *transactions* (idempotent requests to perform an individual business task) fanned out asynchronously to child microservices — if one such transaction fails, then *compensating transactions* (idempotent requests to undo the effect of the transaction) are issued to all other tasks [5]. Sagas guarantee eventual consistency of all business operations implemented this way. Sagas in practice employ an *event bus* that fans out transactions and compensating transactions to listening services, and a coordinator that tabulates transaction responses and issues compensating transactions.

Saga implementation in practice suffer from a number of drawbacks. Event buses are typically implemented by reliable distributed message brokers or publish-subscribe servers, and thus high operational overhead. Centralized event buses form a tier-zero service that must be always available. Special client libraries have to be introduced to interface with message brokers. Configuration for the broker may be managed separately, typically at a higher level than what individual service owners manage. Coordinators must either be implented inside existing services or exist as separate services, becoming maintenance burdens. Since event buses only deliver messages but do not translate them to service calls, it falls on the service owners to implement mapping messages to the actions their applications take. The planning needed to address all of these issues hinders widespread saga adoption.

In this paper, we explore doing away with centralized event buses entirely, instead implementing saga coordinators as network proxies. This approach has many benefits. Significantly, operational overhead is significantly reduced: proxies can be cheaply federated, replicated and distributed in a variety of topologies, allowing horizontal scalability. No client-side libraries or event listeners are needed for triggering a transaction — it is sufficient for applications to expose a REST interface that any calling application may invoke. Finally, we remove the headaches of managing concurrency control by absorbing those concerns into our proxy — once a saga is triggered, our proxy coordinator ensures that a distributed saga either aborts or commits successfully.

We present Qbox, a pure Python implementation of a proxy coordinator. We discuss its behaviour and design on a sample collection of microservices, and present performance comparisons with a traditional event bus implemented with RabbitMQ on cloud orchestration platforms.

2 Related Works

There are a number of libraries that implement the saga pattern on behalf of microservices [9] [10]. Axon [1] implements a dedicated coordinator service called the Axon server, exposes Java annotations that register function calls with the Axon server as endpoints for transactions, and models appli-

cation data as a log — the Axon server manages log rollback on failure of a transaction. Eventuate [2] offers function decorators that register application code as sagas with a message broker, typically one of Apache Kafka, RabbitMQ, Redis or ApacheMQ. Red Hat's Narayana LRA [6] allows applications to join a membership cluster on-demand, and utilizes a Narayana coordinator service to route messages sent to this service to the correct microservice.

Work on distributed transactions has gone towards overcoming the need for specialized client libraries for polyglot services. Pardon and Pautasso [7] recommend augmenting centralized coordinators with a REST interface that allows templated compensating transactions that can be triggered for a specific dynamic resource using a generated URL.

Another research direction for sagas has been to distribute coordinators. SagaMAS [4] embeds a service called an *agent* with eacmicroservice. Each agent maintains heartbeats with neighbouring agents. A single agent propagates transactions to the corresponding agents asynchronously.

Our work combines these latter two research efforts by offering RESTful interface abstraction and an embedded service with each microservice.

3 Design

Qbox is designed as a network proxy that attaches itself to the network namespace of a single pod on a Kubernetes cluster.

Qbox acts in one of two modes for any given request it receives. The first is *proxy* mode, where it transparently forwards all outbound messages unchanged to its final destination and all inbound messages to the application. The second is *coordinator* mode, where inbound or outbound requests matching a certain value trigger the transmission of a series of predefined transaction requests to all other services. Qbox allows the contents of these requests to be dynamic by introducing the concept of *message interpolation* — transactions may have template values that are filled in from values contained in the inbound parent request. Message interpolation supports calling dynamic entities in REST architectures and allows the calling application to inspect what went wrong in the event of a failed saga.

If any transaction request fails, Qbox issues compensating requests for all services that have processed transaction requests already. These compensating requests are predefined and also support message interpolation. Message interpolation here may also include value injection from responses to any parent transaction request, allowing rich composition of compensating requests.

3.1 Execution Handling

Qbox currently features *sequential execution* of transaction messages — transactions are executed in a user-defined order (see section 4), and the service waits until each transaction is completed before issuing the next transaction. In our implementation of Qbox, we use the native multithreaded HTTP server embedded in Python 3.8. Each new client request is handled in a separate thread, allowing Qbox to handle multiple application triggers in parallel even if, within a thread, transactions are sequentially issued.

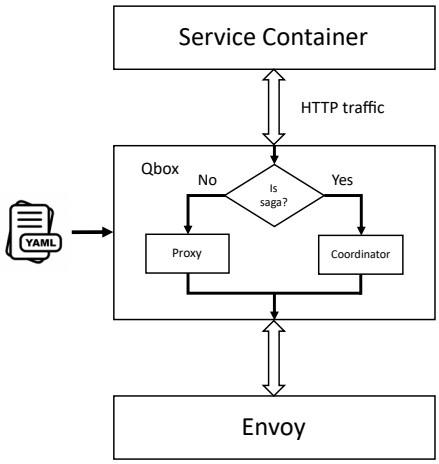

Figure 1: Overview of Qbox's architecture

3.2 Nested Transactions

Microservice topologies today feature a large number of intermediate services that applications may have to call. A nested transaction arises when service must itself issue a transaction in order to respond to a pending one. In traditional saga implementations, this happens asynchronously, with the intermediate service placing a message on the appropriate channel for a dependent service.

In our more synchronous model, we do not have the luxury of message queues. Instead, to achieve the same effect, a single Qbox may, as part of a transaction, directly trigger a saga on another Qbox. This accomplishes the same result at the expense of blocking overhead as the transaction must now wait for these synchronous tasks to complete.

4 Configuration

All saga triggers, transactions, expected responses, issued compensating requests, and final responses to the calling application are specified by the user in the Qbox configuration file. This Qbox configuration file is a YAML file and follows a simple hierarchy. For proof-of-concept purposes, this configuration file is stored as a Kubernetes ConfigMap and injected manually into the sidecar's filesystem.

Here is an example of a configuration YAML with message interpolation involved:

```
{
```

```
    "host": "me.svc",
    "matchRequest": {
        "method": "GET",
        "url": "http://localhost:20000",
        "headers": {"Start-Faking": "True"},
    },
    "onMatchedRequest": [{
        "method": "GET",
        "url": "http://.../${parent.headers.EX}",
        "headers": {
            "MY_HEADER": "${parent.headers.EX}",
        },
        "isSuccessIfReceives": [{
            "status-code": 200, "body": "success"
        }],
        "onFailure": [{
            "method": "DELETE",
            "url": "http://.../${root.headers.EX}",
            "headers": {
                "SHOULD_NOT_EXIST":
                "${parent.headers.FOO:bar}",
            },
            "timeout": 3,
            "maxRetriesOnTimeout": 1,
            "isSuccessIfReceives":
            [{"status-code": 200}],
        }],
        "timeout": 30,
        "maxRetriesOnTimeout": 3,
    }],
    "onAllSucceeded": {"status-code": 200},
    "onAnyFailed": {"status-code": 200},
}
```

An interpolation is an expression of the form `${EXPR[:DEFAULT]}` inserted into another string. The expression will then be evaluated by the coordinator right before sending. As is immediately visible, they may be used anywhere in a message except as header keys. If any value referenced by `EXPR` does not exist, then the value specified by `DEFAULT` is inserted instead.

The full set of valid `EXPR` values and when they are guaranteed to be expected is large. An `EXPR` expression parses as `context(.response?).attribute(.key?)`, where the `?` indicates an optional segment. The `attribute` can be set to values `headers` or `body`, to indicate you want the value of that corresponding populated from a specific message — the optional `key` segment inserts the value of a specific header. The value of `context` can be set to `parent` (the message that triggered the request in which this expression is being evaluated) or `root` (the message that initially triggered the saga trigger). The optional `.response` segment sets it to the received *response* for `context`. The `attribute` can be set to `headers` or `body`, to indicate you want the value of that attribute populated from this message context — the optional `key` segment inserts the value of a specific header.

In this configuration, saga triggers are specified by the details in `matchRequest`. A GET request that is outbound for URL `localhost` on port 20000 and has at least one header with key-value `"Start-Faking": "True"` will trigger transactions specified in `onMatchedRequest`. Each transaction defines an HTTP method type, a URL that should be DNS-resolvable, specifies a successful response in `isSuccessIfReceives`, and specifies a compensating request in `onFailure`.

In the example above, we only have one transaction that is issued, namely a GET request with empty body, a URL that is set to the header key `EX` from the message seen in `matchRequest`, and a header set to the same value to `foo.svc`. It accepts any response with HTTP status code 200, and waits for 30 seconds for this to arrive before closing the connection. Since no retry limit is specified (uses a key called `maxRetries`), if those 30 seconds elapse, Qbox will immediately issue the compensating request specified in `onFailure`, which is just an empty DELETE request to a URL that is set to the header key `EX` from the message seen in `matchRequest`, and a header set to the same value.

Independent of whether the specified `isSuccessIfReceives` HTTP status code of 200 is received by this compensating request, Qbox will respond to the application that originally triggered it with an empty request that only contains the HTTP status code of 200 and close the connection to the application — if the compensating request never received the expected confirmation, Qbox will ideally attach details of the failed transaction in the body. Of course, if the original transaction had succeeded, Qbox would just return the response specified in `onAllSucceeded` instead.

We don't allow embedding response / request values from compensating requests as interpolation placeholders on the basis that there is no reason to use them — an application only cares if the initial transactions succeeded or failed, or whether there are dangling resources, and has no use for values received from or made by compensating requests.

5 Prototype

In this section, we first introduce our prototype of the Qbox with its implementation details. We show how Qbox works with an off-the-shelf application using service mesh with minor modifications. In addition, since we want to test the performance of the Qbox and the benefit of the distributed saga pattern, we build a testbed that supports arbitrary number of services that execute complicated saga DAG using Qbox.

5.1 Integrating Qbox into Bookinfo

For our test work, it was important to first demonstrate full functionality with a sample microservice. We eventually settled on the Istio *bookinfo* application, a suite of four microservices that together deliver a consistent user experience. Our choice to use this particular collection was driven by a few constraints, namely that we only intended to support HTTP protocol and that these microservices would have to be easy to modify without much familiarity. Other, more comprehensive microservice collections use more specialized protocols on top of HTTP and are considerably more complex to adapt to our use case.

To accomplish this and demonstrate viability, we built with Terraform a GKE (Google Kubernetes Engine) cluster that would first install the sample Bookinfo application and enable an Istio sidecar proxy on all containers. We then manually installed Qbox as another sidecar container for all of them.

Bookinfo is nominally a book catalog, and consists of the *productpage*, *reviews*, *ratings* and *details* services. It does not natively need nor require a distributed transaction. We modified *productpage*, *ratings* and *details* to expose an endpoint whereby clients may request *productpage* to add a book to the catalog. This in turn triggers a saga across *ratings* and *details* where these respective details are added using dedicated endpoints on those services. Endpoints for handling compensating transactions were added to *ratings* and *details*. All of these endpoints were designed for idempotency given an existing book. The modifications needed to do this were minimal, with about 100 lines of code total over three microservices.

After these changes, we added a configuration virtually identical to that specified in Section 4. We checked for successful responses if all were up, and injected failure by deleting one of the child microservices to verify compensating transactions were processed. The complete code with Bookinfo is available in this commit on our GitHub repository.

6 Evaluation

6.1 Testing Qbox At Scale

To demonstrate the performance of Qbox handling complicated saga request, *e.g.*, nested sagas, it is undesirable to only use Bookinfo, since this application only has 4 services which is not enough for a load test. On the other hand, there is no open-source applications using microservice pattern that largely use saga execution and has a complex saga execution graph. Therefore, we have built our own testbed for the load test. This testbed includes a dummy service, which receives sagas from upstream and creates arbitrary numbers of sagas to downstream services, and an automatic testbed generator, which initiates multiple copies of the dummy service with different service name, creates different saga execution graphs based on the parsed graph definition provided by users, and generates configuration files for every Qbox.

```
{
    "root": "service1",
    "nodes": ["service1","service2","service3",
        "service4","service5"],
    "edges": {
        "service1": ["service2", "service3"],
        "service2": ["service4"],
        "service3": ["service5"]
    }
}
```

Figure 2: An example configuration for automatic testbed generator

For the dummy service, it receives four URLs from the upstream service, "`/add/id`", "`/delete/id`", "`/saga_add/id`" and "`/saga_delete/id`". For "`/add/id`" and "`/delete/id`", the dummy service will add or delete an entry using the id inside the URL as a transaction. After the transaction is finished, the dummy service will return a 200 message back to the upstream service. For "`/saga_add/id`" and "`/saga_delete/id`", beside executing a transaction, the dummy service also send a new saga request using the same id provided in the URL to the Qbox sidecar, which will initiate a sequence of requests to the downstream services. The dummy service will wait until the Qbox return a message indicating the saga execution is succeeded or not; then the dummy service will return a message to its upstream service accordingly.

For the automatic testbed generator. it at first has to take in a user defined saga execution graph. Below is an example of how user can define a saga execution graph. Every service will share the same copy of the dummy service only with a different name. In this specific case, `service1` will start the execution of the whole saga. First, `service1` sends the saga request to the local Qbox. Then based on the configuration file which is generated by the testbed generator, Qbox sends out request to `service2` and `service3`. Since `service2` and `service3` are intermediate nodes, which means they initiate new sagas to downstream services, will receive URLs like "`/saga_add/id`". `service2` and `service3` then send their saga requests to Qbox; and Qbox will send "`/add/id`" to `service4` and `service5`, since there is no more saga execution after `service4` and `service5`.

By using a such dummy service and automatic testbed generator, we can create our own application consisting arbitrary number of services and saga execution graph by only writing a JSON file to define the structure of the execution graph. In this way, we can load-test our prototype of Qbox at any scale.

6.2 The baseline cluster

In addition, instead of testing our Qbox design alone, we want to compare the performance of Qbox (which executes saga in a distributed fashion) with a centralized version of saga coordinator. However, to the best of our knowledge, there is no open-source implementation that is able to support saga execution in a centralized way. We therefore built our own naive centralized saga coordinator. For our implementation, we used RabbitMQ [8] as the message broker.

Each microservice has its own dedicated message queue. The microservice that initiates the sage execution writes to these message queues, and includes in the message its own message queue for transaction responses. Once the microservice decides to initiate a saga execution, it will push message into all message queues, where downstream services listen to Sagas, together. This means that unlike in Qbox, the centralized saga coordinator is executed in an asynchronous way. Our implementation of centralized version is also scalable, which means that it can also take in a service topology and generate Kubernetes YAML files for those microservices, *e.g.*, the code shown in figure 2. The automatic testbed generator will generate Kubernetes manifests for centralized version. In the manifests, we use environment variables to indicate how many channels should be created and to assign roles of producers and comsumers for every service according the configuration file.

In this way, we can easily set up a Qbox cluster and a baseline cluster in one click, and conveniently test among different saga execution scenarios.

6.3 Load testing

As we have integrated Qbox into Bookinfo, the correctness of Qbox's implementation has been tested. In this section, we focus on the saga execution performance running on Qbox comparing to on the baseline implementation. Namely, under different service topologies, we want to evaluate the latency of requests to a service that initiates saga executions.

For a fair comparison, we deploy the Qbox and the baseline implementation on GKE with same amount of resources. Each compute node has 6 vCPUs and 22.50GB memory. The load testing tool we used is Vegeta [11], which can send HTTP requests to services in a given sending rate and time and generate a report showing the latency distribution of all requests sent.

We deployed 5 different topologies on the GKE. In table 1 and 2, we include the number of services and number of nested sagas to characterize different topology. (One nested saga means one saga will trigger the execution of another saga.) For every topology, Qbox and baseline implementation will be tested separately. Since Qbox is implemented in Python, which is not a production-grade implementation, we limit the request rate to 10 requests per second. Table 1 and 2 are the results of our evaluation.

Table 1: The latency evlaution of Qbox implementation

topo	1to4	1to2to2	1to9	1to3to6	1to5to5
# service	5	5	10	10	11
# nested saga	0	2	0	3	5
min	80.5ms	60.7ms	136.2ms	216.7ms	308.6ms
mean	113.6ms	84.0ms	489.8ms	892.1ms	1.116s
50%	101.6ms	78.9ms	498.0ms	895.0ms	1.157s
90%	131.9ms	102.5ms	746.5ms	1.39s	1.585s
95%	201.3ms	106.8ms	774.7ms	1.409s	1.674s
99%	394.6ms	252.3ms	884.8ms	1.539s	1.899s
max	400.3ms	302.5ms	884.9ms	1.557s	1.99s

Table 2: The latency evlaution of centralized implementation

topo	1to4	1to2to2	1to9	1to3to6	1to5to5
# service	5	5	10	10	11
# nested saga	0	2	0	3	5
min	130.5ms	136.0ms	97.44ms	136.3ms	139.7ms
mean	143.0ms	152.0ms	115.6ms	164.7ms	168.1ms
50%	139.6ms	143.0ms	108.6ms	155.1ms	159.5ms
90%	147.2ms	178.8ms	128.3ms	201.1ms	183.3ms
95%	150.1ms	203.2ms	180.1ms	208.1ms	243.2ms
99%	257.9ms	302.1ms	230.7ms	303.9ms	312.9ms
max	289.4ms	302.2ms	270.0ms	306.3ms	321.6ms

There are three observations we want to discuss and it becomes the guidance for our future works to improve the Qbox's performance and usabiliy.

First, when the number of services is small, *i.e.*, topo 1to4 and 1to2to2, the performance of Qbox implementation is better than the centralized version. We attribute this to the Qbox's simplicity of processing logic. For the centralized version, RabbitMQ will take large amount of time maintaining concurrent message queue, which involves in complex concurrency problems and business logic. This echoes the one of the purpose implementing distributed saga coordinator — distributed saga execution is able to reduce the overhead handling complicated processing logic.

Second, when the number of services grows up, the Qbox implementation's processing latency grows substantially, *i.e.*, topo 1to4 versus topo 1to9. To found the root cause, We timestamped the request traffic and crosschecked the log in different services. The conclusion is that since right now we are using sequential execution of transaction, which means Qbox will send consecutive requests to one service and then once receive the response from the first service, it hammers the second service. Therefore, the sequential execution strategy leads to the unsaturated load of downstream services. We also used htop to monitor the workload during load testing and conformed that load of mock services is lower than the centralized version, which uses asynchronous execution stratey.

Third, processing latency increases as the number of nested saga grows, *i.e.*, topo 1to2to2 versus 1to5to5. We think the reason falls into two parts. First, since we limit the total resource quota, more nested saga leads to more Qbox containers,

which prorate the computing resource and, in turn, drag down the performance of whole cluster. Second, since the services need to use HTTP protocol to talk to Qbox, nested saga means deeper execution path, more HTTP connection establishment and more TCP handshakes. This leads to longer latency in the networking level. In contrast, the centralized implementation saves such overhead by using long term connection to listen to RabbitMQ's server and shorter execution paths.

The evaluation provides us a guidance and direction to improve Qbox implementation's performance. In the next section 7, we discuss in detail about how to improve the Qbox's performance and make it production-grade implementation.

7 Future Work

Qbox currently works well as a robust proof-of-concept. In future work, we would like to improve performance and simplify ease of integration.

7.1 Performance Enhancements

We anticipate that the single biggest performance boost will come from rewriting the Qbox engine as a custom Envoy filter that builds on the HTTP Envoy filter. The Envoy project demonstrates extremely low overhead of 2.8 ms in the 90th percentile when tested at scale [3], and offers out-of-the-box support for a variety of protocols besides HTTP — both aspects that Qbox could take advantage of by adhering to that framework.

Further, Qbox's use of sequential execution incurs idle time — each request must wait for responses to prior requests before it can be dispatched. The use of *pipelined executions* where instead a request is dispatched as soon as its prior requests are dispatched will reduce the total time for large transaction request queues.

7.2 Simplifying Integration

For proof-of-concept purposes, we directly stored Qbox configuration as a Kubernetes ConfigMap object. For larger-scale deployments, we anticipate it would be much easier to submit configuration to a Kubernetes controller, which then dispatches or reloads proxy configuration dynamically. One way of doing this would be to extend the previously proposed rewrite of Qbox as an Envoy filter to the Istio project, which dedicates a control plane over Envoy-based proxies. Developing a custom Istio controller and using a Kubernetes custom resource definition that distributes configuration would be a promising direction to simplify deployments.

Qbox currently does not support any other protocol besides HTTP. We envision building a plugin system that allows traffic to be parsed by *plugins* — code that translate traffic content into a plaintext format that Qbox can match against. These plugins could convert serialized or encrypted content such as gRPC / HTTPS into HTTP or an intermediate representation form that the configuration could matc(h against. One challenge is that streaming data transfer mechanisms such as chunked transfer encoding in HTTP 2) result in messages that can't be parsed by Qbox without waiting for the whole message. More research must be done to identify how to optimize Qbox for streamed content.

References

[1] Axon. Axon: Framework and server for event-driven microservices. https://axoniq.io/, 2020.

[2] Eventuate. Solving distributed data management problems in a microservice architecture. https://eventuate.io/, 2020.

[3] Istio. Istio 1.5.2 performance and scalability. https://istio.io/docs/ops/deployment/performance-and-scalability/, 2020.

[4] Xavier Limón, Alejandro Guerra-Hernández, Angel J Sánchez-García, and Juan Carlos Peréz Arriaga. Sagamas: a software framework for distributed transactions in the microservice architecture. In 2018 6th International Conference in Software Engineering Research and Innovation (CONISOFT), pages 50–58. IEEE, 2018.

[5] Caitie MaCaffrey. Applying the saga pattern. https://www.youtube.com/watch?v=xDuwrtwYHu8, 2015.

[6] Narayana. Narayana. https://narayana.io/, 2020.

[7] Guy Pardon and Cesare Pautasso. Atomic distributed transactions: a restful design. In Proceedings of the 23rd International Conference on World Wide Web, pages 943–948, 2014.

[8] RabbitMQ. Rabbitmq. https://www.rabbitmq.com, 2020.

[9] Bc Martin Štefanko. Use of transactions within a reactive microservices environment.

[10] Martin Stefanko, Ondrej Chaloupka, and Bruno Rossi. The saga pattern in a reactive microservices environment. In ICSOFT, 2019.

[11] Vegeta. Vegeta github repository. https://github.com/tsenart/vegeta, 2020.