AriZona: Yet Another Coordinator

Zezhi Wang
Brown University

Anatoly Brevnov
Brown University

1 Introduction

Nowadays, most business applications are commonly run within collections of distributed and loosely coupled microservices environments. Microservices are lightweight and independent, and their stateless nature makes them flexible, scalable, and highly available.

However, it is a different story for stateful services. Consistency among services is hard to achieve, and many solutions that have attempted to achieve some aspects of this desired consistency have been proposed over the years.

One such solution is grounded in the desire to maintain consistency in storage in the presence of transactions that span multiple services. This solution is the use of *Sagas*. The saga-pattern ensures atomicity by issuing compensating requests on request failures to undo successful partial requests. Eventual consistency is guaranteed for business operations implemented in this pattern. In practice, the execution of sagas is managed by centralized coordinators that issue partial and compensating requests. The downside of this approach is that these coordinator services become single points of failure that are not scalable and are not efficient.

Some saga implementations in practice use strongly consistent distributed consensus protocols as part of their distributed storage algorithms to provide fault tolerance and consistency guarantees. But these solutions are less efficient since they are bounded by sequential execution, and providing strong consistency guarantees requires more coordination-time as the coordinator cluster scales. This causes the efficiency of such methods to degrade at scale.

Some other works like Aegean [1] propose methods of getting out of the client-server model and instead have intermediary services perform the task of coordination themselves. Such approaches help to address scalability concerns, but they still lack efficiency due to the overhead of the consensus coordination tasks among replicas of the middle services.

In this paper, we designed, implemented, and evaluated a new algorithm inspired by Apache Cassandra [11]. To provide fault tolerance, we included a highly-available cluster of saga coordinators. And to boost throughput, we allow parallel executions by assigning workloads to sub-clusters and maintain strong consistency within this sub-cluster by majority read/write quorum, similar to the log replication phase of Raft [17].

In summary, we present YAC (Yet Another Coordinator), a pure Go implementation of a coordinator service for distributed sagas that is highly available, efficient, and scalable.

2 Background

2.1 Microservices

Microservices refers to a methodology that provides a way for modern developers to build highly scalable, modular applications by decomposing the application into discrete services that perform specific business functions. These services can then be independently designed, deployed, and scaled, i.e. loosely coupled.

Contrary to monolithic systems (where components are interlinked and can only be scaled together), microservices communicate via application programming interfaces (APIs), allowing services to be written in various languages or technologies. Each service has minimal features, and there small sizes and lack of complexity make it easier for a single team to manage.

2.2 Sagas

Consider a payments application. During a transaction between two users, requests may need to be made to an Employee's service to update the employee's balance as well as to an Employer's service to update the employer's balance. If the transaction fails after updating the employee's record but not the employer's record, that would cause inconsistency (as now the employee has a diminished balance but the employer has effectively not received payment).

A solution for such use cases is the notion of *Sagas*. A saga consists of *partial requests* and corresponding *compensating requests*. Partial requests are the requests that compose our multi-service transaction. Compensating requests are requests that can roll-back changes made from partial requests. The key insight is that by sending out compensating requests on failure from any partial request, we can guarantee eventual consistency.

2.3 Consistent Hashing

Consistent Hashing [10] is a special hashing algorithm to locate servers responsible for each request. Generally, the hash function would randomly map the requests and the servers to a unit circle. Each request would then be mapped to the next server on the circle in a clockwise manner. This algorithm theoretically provides a uniformly distributed mapping of requests to servers.

2.4 Leader Election

Many distributed consensus algorithms have the leader election phase [17] [12]. It is the process of determining the server responsible for an operation. Before this process, all servers in the group would either not able to communicate with the previous leader or simply have no knowledge about the task. And after this process, every server in the group would have consensus on a unique, particular server, acting as the leader for some operations. Some leader election process [17] may go through a voting process in which the leader must gather a majority of votes from the group of servers. In this paper, we take a simpler approach based on the provided ordering from Consistent Hashing discussed in Section 2.3. The server who is closest to the hash of request when moving clockwise around the unit circle would be the leader.

2.5 Response Consensus

Consensus implies each replica within a certain group agrees on a given state.

This can be achieved via the use of quorums. Generally speaking, a coordinator has to obtain acknowledgments from read and write quorums for read and write operations, respectively. The size of read and write quorums can be denoted by r and w. Strong consistency, or safeness, is satisfied when $r+w > n$, where n denotes the number of servers within such cluster. This inequality guarantees that the read quorum and write quorum has an intersection of at least one server by the pigeonhole principle, which means at least one server would witness both operations. This common server would hence be a tie-breaker to ensure strong consistency. This consistency model would ensure that a read would return the most recent write, but it does not necessarily ensure total ordering.

3 Related work

Distributed Sagas has been proposed [14] [18] as a method to provide support for distributed transactions. Distributed Sagas are the utilization of Sagas [4] in distributed environments. It requires that requests must be able to abort, be idempotent, and have compensating requests that are commutable to the original ones. If any of the required requests is unsuccessful, all corresponding compensating requests would get executed to return to the original state. This guarantees Atomicity.

Aegean [1] is a distributed system that rethinks the replication model beyond the client-server model. It described how traditional ways of replication would be both incorrect and inefficient in presence of multiple services and nested requests. Therefore, the authors defined a new form of correctness, *indistinguishable*, which relaxes the requirement of linearizability and allows other modes of execution such as parallel execution. To guarantee this ordering, they proposed *Response Durability*, which requires that each service replica broadcasts acknowledgments to others upon receiving partial responses, and that each service replica would only consider a response being durable when it has received a quorum of acknowledgments. By doing this in the middle service instead of using coordinators, the scalability of the system is drastically improved, as coordinators would no longer be the bottleneck. The authors also mentioned the problem of duplicated requests. And to solve this problem they proposed *Server Shim*, which takes care of collecting a quorum of requests from replicated service before forwarding to backend service. This component ensures that the backend service is not processing the same request multiple times for replicated clients, without needing to store a *request cache* for each request.

Distributed storage systems are another approach to make coordinators fault-tolerant. Various strongly consistent systems like etcd [3], MongoDB [16], Apache Zookeeper [7], and Apache Cassandra [11] are popular approaches towards implementing highly available coordinators. Our system is partially inspired by Cassandra in that our protocol also utilizes Consistent Hashing and tunable quorum sizes. Cassandra uses the consistent hashing algorithm to locate the responsible node and Probabilistic Quorum System (PQS) [2] for consistency guarantees. It also ensures strong consistency when read quorum plus write quorum is larger than the node count.

4 System Architecture

The general model consists of three parts: 1) client services, 2) a cluster of coordinators, and 3) backend services. This paper focuses on the cluster of coordinators.

In our architecture, each coordinator maintains a *status* in memory, which is a mapping from a specific *request* to the sub-cluster of coordinators responsible for this request and a mapping from the *partial_requests* in this *request* to the status of those partial requests. An illustration of our system architecture is shown in Figure 1, where the yellow node represents a client service, the

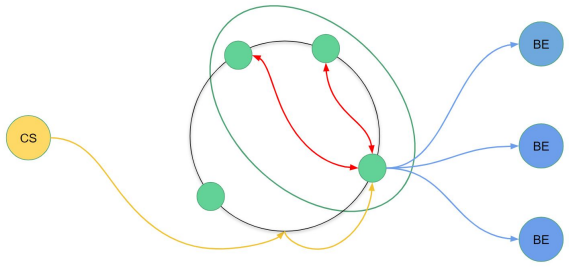

Figure 1: System Architecture

green nodes represent coordinators, and the blue nodes represent backend services.

The proposed system, different from vanilla Distributed Sagas [14] [18] and Qbox [13], should be fault-tolerant. And the systems should yield better performance than Aegean [1] or distributed storage based systems such as Zookeeper [7], due to the reduced message complexity from $O(n^2)$ to $O(k)$, where n is the total number of coordinators and k is the tunable sub-cluster size ($k \leq n$). Also, the system should have higher availability and throughput than Raft [17] based coordinators, since the sub-cluster setup would allow more requests to be processed in parallel, rather than having an entire cluster of coordinators working on the same requests.

However, the more we allow parallel execution to improve efficiency, the less ability we have to provide isolation guarantees. Of the possible countermeasures of concurrent access of resources in Saga patterns provided by Microsoft [15], our system supports:

- Semantic lock: we support usage of application-level locks, where compensating requests use semaphores to indicate progress.
- Commutative updates: we support requests that can be executed in arbitrary order without producing different results.

When using our protocol, most of the work to provide measures of isolation must be taken by the backend services (such as application-level locking), as our protocol focuses solely on the coordinator cluster and provides no isolation guarantees on its own.

4.1 Coordinator Cluster

The coordinators form a ring structure. Each coordinator has a distinct hash value mapped onto a unit circle, and their location can be treated as the point on the circle. Coordinators also keep track of each other's statuses and which IP addresses they can be reached at. In our implementation, coordinators accomplish this using a Kubernetes client to periodically query the Kubernetes Service in order to maintain a **member_list** (which includes hash values, IP addresses, and statuses of peer coordinators). This is how coordinators can keep track of the addresses and vitality of peer coordinators.

4.2 Locate the Responsible Coordinator

A client's request (r) is sent from the client service and is load balanced to a random coordinator. Upon receiving the client request, the coordinator computes a hash (h) based on the client service generated ID to determine where this request falls on our circle of hashes. It then computes the next coordinator (c) that appears on the circle in clockwise order. If c happens to be itself, it would proceed to execute the saga as described in Section 4.3. Otherwise, the coordinator would respond with a redirection status and address of c, asking the client service to send r to such a coordinator.

4.3 Sub-Cluster Formation

The responsible coordinator, upon receiving such a request, would initiate a local leader commit phase by sending this request to a pre-defined number of coordinators that are clockwise next to it on the circle. Let's name this sub-cluster as $\mathbb{C}$. Following similar steps as the log replication phase of Raft [17], the leader coordinator would wait for a majority quorum of these coordinators to reply with confirmation before proceeding. The coordinators within $\mathbb{C}$ would reply acknowledgment if it has never seen this request before, or if it has not-newer state of this request. Otherwise, if some coordinator in $\mathbb{C}$ has a newer state, it would reply with rejection and piggyback its state to the proposed leader. The leader, if received a message with a newer state, would update its state locally, and repeat the steps in this section again.

4.4 Leader Re-election

At this point, the coordinators in $\mathbb{C}$ would have knowledge of this request, and the members in this sub-cluster. We consider the failure scenario that the leader coordinator (the one with a hash value right next to the request clockwise on the circle) crashes or becomes unresponsive. Since every coordinator is frequently polling the Kubernetes Service for information on its peers, the coordinators in $\mathbb{C}$ would know with a high probability the vitality of its leader coordinator. Once the leader coordinator is inactive for a certain period, the coordinator next to it on the circle would repeat the process in Section 4.3. As a result, it would become the new leader and proceed with the execution. Pseudo-code for this process is presented in Algorithm 1.

Algorithm 1 Sub-cluster Formation

```
1: n: pre-defined sub-cluster size
2: status: map from requests to status
3: procedure ELECT(request)
4:     save request locally
5:     hash ← ConsistentHashing(request.IP)
6:     ℂ ← next n node on circle
7:     initiate witnesses to empty list
8:     broadcast request to ℂ
9: procedure RECEIVING(peer_response)
10:    if peer_response is ACK then
11:        add peer_response.ID to witnesses
12:        if size of witnesses = ⌈(n+1)/2⌉ then
13:            proceed execution of request
14:    else
15:        update status to peer_response
16: procedure RECEIVING(leader_init)
17:    if request of leader_init not in status then
18:        reply ACK
19:    else
20:        reply status[request]
```

Algorithm 2 Request Processing

```
1: n: pre-defined sub-cluster size
2: status: map from requests to status
3: procedure PROCESS(request)
4:     req_m ← map(partial requests→empty lists)
5:     broadcast partial requests in request
6: procedure RECEIVING(p_req_resp)
7:     update status
8:     add self ID to req_m[p_req]
9:     broadcast p_req_resp to ℂ
10: procedure RECEIVING(peer_req_resp)
11:    add peer ID to req_m[p_req]
12:    if size of req_m[p_req] > ⌈(n+1)/2⌉ then
13:        proceed handling next partial request
14: procedure RECEIVING(leader_p_resp)
15:    update status
16:    reply ACK
```

4.5 Request Processing

After receiving sufficient confirmation from peers, the leader coordinator would now send the nested request to corresponding backend services for processing. To avoid the duplicated request problem mentioned in Aegean [1], here only the leader would send the request. There are two possible scenarios:

- If a partial request is successful, the leader would broadcast the result to coordinators in $\mathbb{C}$ and ensure a majority quorum of confirmation. And each coordinator would mark the partial request as successful. If all the partial requests are successful, the leader would now broadcast *commit* message to coordinators in $\mathbb{C}$, wait for the majority of responses.

- If a partial request fails, the leader would broadcast abort messages to coordinators in $\mathbb{C}$, and ensure the majority of coordinators have the state being *Aborted*. Then, the leader would send out all compensating requests to backend services. Upon receiving a successful response from compensating requests, the leader would follow similar steps that to broadcast the response and ensure the majority of confirmation.

In both scenarios, the leader would then proceed to send the corresponding response to the client. After successfully sending the response, the leader would then broadcast a *completion* message to other coordinators and wait for the majority of confirmations. Pseudo-code for these steps is presented in Algorithm 2.

4.6 Request Continuation

In this part, we consider the failure scenario of the leader failing during request processing (Section 4.5). Since for each partial response the leader makes sure the majority of the coordinators in $\mathbb{C}$ witness it, we are safe to infer that most coordinators would have consensus on the state of responses before the current partial request. When the leader crashes, $\mathbb{C}$ would follow Section 4.4 to elect a new leader. This leader would reflect the newest state of the correct coordinators. It would, in order to abort such saga request, re-send all the compensating requests that need to be sent based on the current state. It is possible that the backend service or the client may receive a few duplicated messages. But, duplicated requests are preferable to missing requests (given that our protocol requires idempotent partial/compensating requests). We are not using the *Shim Layer* as Aegean [1] does, because, normally, only the sub-cluster leader is sending out the request, and *Shim Layers* with quorum size one would only lead to inefficiency and more points of failure.

4.7 Request Tiers

To further support more execution modes, we introduce the idea of *tiers*. The motivation behind tiers is that we want to maximize the ability to execute partial requests in parallel. However, we realized that certain use cases could exist where users may want to execute certain partial requests *before* other partial requests in a multi-step transaction. This led us to adopt a blended sequential/-

```
{"tiers":{
  "0":{
    "req1":{
      "partial_req":{
        "method":"POST",
        "url":"http://1.1.1.1:80",
        "body":""
      },
      "comp_req":{
        "method":"DELETE",
        "url":"http://1.1.1.1:80",
        "body":""
      }
    }
  }
}}
```

Figure 2: Request Body

parallel approach. Clients group their requests into tiers. Batches of requests are executed in parallel at each tier in an ascending manner. This means that every partial request at tier n+1 is executed *after* any and every partial request in tier n. Additionally, if any partial request at tier n fails, coordinators need only roll back all the partial requests from tiers n and below. An example request body is shown in Figure. 2

5 Prototype

In this section, we introduce our prototype of Yet Another Coordinator and a baseline coordinator using ZooKeeper [7] with implementation details.

5.1 Prototype Implementation

Our prototype is built in Golang. The messages between coordinators as well as between services are handled by *Gin* [5], a high-performance HTTP web-framework written in Go. The consistent hash algorithm is achieved via using *hashring* [6], a consitent hashing implementation in Go. Additionally, we used *xid* [20] to randomly generate uuid's for request IDs in dummy services. The code for our coordinator is publicly available at this repository.

5.2 Baseline Implementation

Since a single coordinator is not fault-tolerant, many usages of sagas pattern in the industry involve replicated coordinators communicating with some algorithms to ensure strong consistency. Some example algorithms would be Raft [17], Paxos [12], and Zab [9]. Due to our limited timeframe, we were only able to build our baseline coordinator using Zookeeper [7] as a data storage mechanism. Zookeeper is a service for coordinating applications utilizing the Zab algorithm. We selected Zookeeper due to its universal usage in numerous projects and its guarantee of sequential consistency. The code we used to build this baseline coordinator implementation is available at this repository.

6 Evaluation

Similar to Qbox [13], we ensured that our prototype integrated with a modified version of the Istio *Bookinfo* Application [8]. Additionally, we evaluated the performance of our protocol at scale by constructing dummy services with more complicated Sagas that we can use to test our coordinator cluster on.

6.1 Integrating with Bookinfo

The first step towards effective testing is demonstration of basic functionality and correctness. We followed the path of Qbox to modify the Istio bookinfo application, which is a book catalog application containing four microservices: *productpage*, *reviews*, *ratings*, and *details*. Since it does not have native distributed transactions, we modified *productpage*, *ratings*, and *details* to expose an endpoint in *productpage* triggering sagas across *ratings* and *details* in order to add books. We also added endpoints in *details* and *ratings* to handle compensating requests (in the same manner as Qbox). These endpoints are idempotent given an existing book. We further modified the *productpage* service to adapt the request format to match the format supported by our coordinator implementation (as described in Section 4.7).

We then deployed this application and a coordinator cluster on GKE (Google Kubernetes Engine). The *productpage* sends saga requests to the coordinator service and proceeds with our protocol as in section 4.5. The modified Bookinfo is available in this branch on our Github repository.

6.2 Correctness Testing

After deploying the Bookinfo application on GKE, we deployed a cluster of 5 coordinators, with a sub-cluster size of 3, on the same GKE cluster. We then manually triggered *productpage* to repeatedly sending saga requests to coordinator service and check for successful responses. Next, we manually failed the *ratings* service, then the *details* service, and then both of them together, while verifying compensating requests were correctly sent. To test the fault-tolerance claim, we ran-

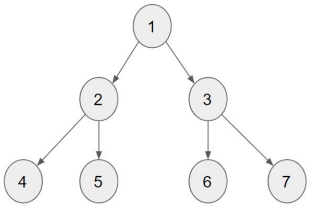

Figure 3: DAG

```
{"svc":{
    "dummy1": "dummy2,dummy3",
    "dummy2": "dummy4,dummy5",
    "dummy3": "dummy6,dummy7",
    "dummy4": "",
    "dummy5": "",
    "dummy6": "",
    "dummy7": "",
}}
```

Figure 4: JSON

domly failed the coordinator pods one at a time during the process previously described and verified all compensation requests are correctly sent and received.

6.3 Creating Dummy Services

In order to test the performance of our coordinator cluster at scale and its ability to support more intricate saga patterns, we realized that purely relying on Bookinfo was an insufficient option (as this application only has four services). To the best of our knowledge, there are no large open-source microservice applications relying on saga execution patterns and nested sagas patterns. Therefore we decided to build dummy services that could support arbitrary saga architectures in-house.

Users of our testing pipeline would need to provide a saga, represented as a Directed Acyclic Graph (DAG). We use this graph as a blueprint from which we can build arbitrary service architectures. Each of these dummy services share the same docker image and follow the same execution logic. Users encode this DAG in a JSON file. This file must include descriptions of all services and their child services separated by commas. An example of a DAG and its corresponding JSON representation is provided in Figures 3 and 4, respectively.

We also provide a Python script to consume this JSON file and generate YAML configuration files for the dummy services and deployments as well as start-up and tear-down scripts for these services and deployments. We can use this start-up script to deploy these services. The complete code for this dummy-service setup is available in this repository.

topo	# svc	mean	50%	95%	max
1-3	4	93.58	78.73	204.18	206.69
1-14	15	124.88	120.65	150.57	236.91
1-2-2	6	109.62	95.89	199.28	205.61
1-3-3	13	109.68	104.14	139.41	225.82
Big1	20	195.41	202.35	210.96	267.83
Big2	36	204.75	203.17	245.57	283.45

Table 1: The latency evaluation of Baseline (Zookeeper)

The dummy service receives HTTP requests from upstream services via two endpoints:

- `POST /<id>`: If the dummy service is a sink-node in the DAG, this endpoint would simply store the value of id locally. Otherwise, it would create a nested request to the same endpoint of all its child services.
- `DELETE /<id>`: Similar to the previous endpoint, if it is a sink-node, it would delete the value of id locally and respond *OK* to the upstream service. It would respond *BadRequest* if such id does not exist locally. If it is not a sink-node, it would send nested requests to the same endpoints of all its child services.

For the example shown above, we illustrate the workflow as follows.

1. `dummy1` will start the execution and send nested requests to coordinator service
2. The coordinators would execute the partial requests, i.e. sending them to `dummy2` and `dummy3`
3. `dummy2` and `dummy3` would both issue saga requests to coordinator service
4. The coordinators would send partial requests to `dummy4`, `dummy5`, `dummy6` and `dummy7`
5. `dummy4`, `dummy5`, `dummy6` and `dummy7` are all leaf nodes, and thus would execute locally and respond back.

By using these dummy services, we can create dummy applications with an arbitrary number of services and an arbitrary execution graph.

6.4 Load Testing

In order to have a fair comparison, we deploy five of our coordinators (with sub-cluster size three) and five baseline coordinators (interacting with a five-node Zookeeper cluster) on GKE. The load testing tool we used is Vegeta

topo	# svc	mean	50%	95%	max
1-3	4	65.37	61.85	84.75	159.19
1-14	15	82.37	71.56	154.97	205.61
1-2-2	6	108.61	88.70	205.23	207.70
1-3-3	13	83.76	75.98	139.69	183.12
Big1	20	131.62	120.90	191.21	206.08
Big2	36	146.57	131.47	203.97	424.83

Table 2: The latency evaluation of YAC

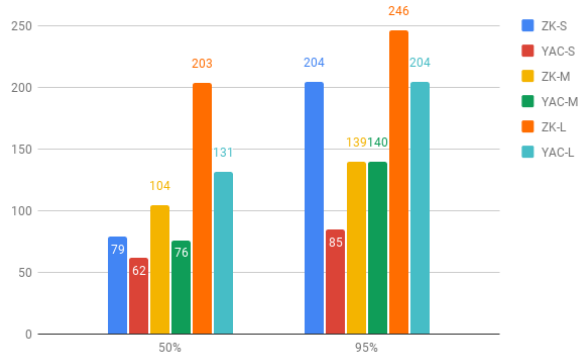

Figure 5: Latency comparison

[19], a tool that can send HTTP requests to services with a user-defined rate and time, to generate a report consisting of the latency distributions of all requests.

We deployed 6 different service topologies on GKE. In Tables 1, 2 and Figure 5, we report the results we have in milliseconds. For all of our topologies, we include the total number of services in their respective execution graphs. Additionally, most of our topologies are named in the form "x-y-z" where x is the number of services at layer 1, y is the number of services called by each layer-1 service, and z is the number of services called by each layer-2 service. There are two topologies that do not fit this naming convention, however: "Big1" and "Big2". These topologies have depths of 7, are composed of more services than the other topologies we tested on, and have less symmetrical execution graphs than the other topologies listed. The JSON representations of these topologies can be found here. Since our implementation uses HTTP to communicate between coordinators, it is far from production level. Hence we tested with the relatively low request rate of 5 requests per second over a 10-second period, and the results are shown in Tables 1 and 2.

For almost all the data we gathered, our prototype is out-performing that of the baseline. As shown in Figure 5, with selected results from small (*S*), medium (*M*), and large (*L*) topologies, when the topology is small and the relationship between services is simple, the performance of our prototype and that of the baseline is close. This may result from the small number of responses that need to be replicated and the quick speed at which dummy services respond. When the number of services gets larger, the latency benefits of our protocol begin to show.

This is partly because our algorithm requires fewer acknowledgments from peers, and we are allowing parallel execution of saga requests. However, this is not entirely due to our algorithm. Our prototype does not write to persistent storage, while the baseline does. This is something we want to change, and we have included it as part of the future directions discussed in Section 7. At the same time, the communication protocol used by our prototype is by no means an efficient one. For each partial response replication, we are initializing HTTP protocol to peers, which involves steps like more TCP handshakes and thus longer latency in the network level, while Zookeeper's network is highly optimized and does not have such an overhead.

The evaluation provides us with guidance on our next steps for this project in terms of both architecture design and implementation choices. In Section 7, we discuss in detail some plausible future directions.

7 Future Work and Discussion

Our system currently works fine as a proof-of-concept, but it has significant room for improvement. In future work, our model can be modified to support more means of isolation and can be simplified and made more performant by switching network protocols. Ease of integration is also something we need to improve in the future.

7.1 Extensive Evaluation

Due to the time limit, we are only able to test with one configuration, i.e. 5 replicas and sub-cluster size 3, and our prototype does not yet have disk storage. One of the most important tasks in the future would be to add persistent storage to our prototype and to evaluate performance with more diverse configurations. For example, we are planning to test how our prototype would perform with the sub-cluster size being the same as Zookeeper cluster size. In this case, we would need the same amount of acknowledgments from peer replicas to proceed. Some other configurations are also interesting to evaluate, and we leave them as future works.

7.2 Isolation

As mentioned in Section 4, we currently only support Isolation via Semantic locks and Commutative updates. An option can be provided for users that need other means of achieving isolation. For example, we may support *version file*, which records operations in the order they arrive and execute them in such an order. One way

to achieve this would be to include distributed locks on the different resources the coordinators have access to.

7.3 Network Protocol

Our prototype only supports HTTP requests between coordinators and between services. We believe this to have been a poor design choice that we made at the planning phase. The main issue is the request-response format of HTTP requests. Since we chose HTTP, our protocol has to send redirect responses to client services instead of simply passing them off to other coordinators. Moreover, having HTTP communication between nodes implies having short-living sockets, and thus the coordinators are spending much more time on the network layer than required. When experimenting with our evaluation, we noticed that if we significantly increased the number of requests per second we sent to our coordinators, we started to run into socket issues with our coordinators due to the heavy reliance on HTTP and the increased overload that accompanies it. Switching to protocols such as TCP, for instance, would enable our coordinator to maintain long-term connections to boost performance and avoid some of the issues described in this section.

7.4 Simplifying Integration

For this proof-of-concept, we require the client services to send exactly what partial requests and compensation requests would be. This design may introduce unnecessarily large message sizes. In production environments, we may allow client services to register some frequently-used requests and only require some unique identifier from the client service to actually perform these operations.

8 Conclusion

We have presented the design, prototype implementation, and evaluation of our protocol YAC. According to our preliminary evaluation, our proposed architecture is both highly available and highly performant when compared to a baseline implementation backed by Zookeeper. However, despite being highly available and highly performant, our protocol has limited built-in mechanisms for guarenteeing isolation. In fact, the isolation mechanisms it supports (such as application-level locking mechanisms) must be implemented by backend services (outside of the scope of our protocol). Therefore, our protocol presents a tradeoff between isolation and performance. In summary, YAC is a coordinator service that uses consistent hashing and quorums to provide a scalable coordinator for microservices invoking the distributed saga pattern.

References

[1] AKSOY, R. C., AND KAPRITSOS, M. Aegean: Replication beyond the client-server model. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2019), SOSP '19, Association for Computing Machinery, p. 385–398.

[2] BAILIS, P., VENKATARAMAN, S., FRANKLIN, M. J., HELLERSTEIN, J. M., AND STOICA, I. Probabilistically bounded staleness for practical partial quorums. *Proc. VLDB Endow. 5*, 8 (Apr. 2012), 776–787.

[3] ETCD. etcd. `https://github.com/etcd-io/etcd`, Github, Dec 2020.

[4] GARCIA-MOLINA, H., AND SALEM, K. Sagas. *SIGMOD Rec. 16*, 3 (Dec. 1987), 249–259.

[5] GIN. Gin web framework. `https://github.com/gin-gonic/gin`, Github, Jun 2014.

[6] HASHRING. hashring. `https://github.com/serialx/hashring`, Github, Jul 2020.

[7] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference* (USA, 2010), USENIXATC'10, USENIX Association, p. 11.

[8] ISTIO. Bookinfo application. `https://istio.io/latest/docs/examples/bookinfo`, Aug 2020.

[9] JUNQUEIRA, F. P., REED, B. C., AND SERAFINI, M. Zab: High-performance broadcast for primary-backup systems. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN)* (2011), pp. 245–256.

[10] KARGER, D., LEHMAN, E., LEIGHTON, T., PANIGRAHY, R., LEVINE, M., AND LEWIN, D. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 1997), STOC '97, Association for Computing Machinery, p. 654–663.

[11] LAKSHMAN, A., AND MALIK, P. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev. 44*, 2 (Apr. 2010), 35–40.

[12] LAMPORT, L. The part-time parliament. *ACM Trans. Comput. Syst. 16*, 2 (May 1998), 133–169.

[13] MAHAJAN, A., AND CHANGHAO, W. Qbox: A saga execution coordinator for the service mesh, 2020.

[14] MCCAFFREY, C., KINGSBURY, K., AND NARULA, N. aphyr/dist-sagas, May 2015.

[15] MICROSOFT. Saga distributed transactions - azure design patterns, Jul 2020.

[16] MONGODB. Mongodb. `https://github.com/mongodb/mongo`, Github, Dec 2020.

[17] ONGARO, D., AND OUSTERHOUT, J. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (USA, 2014), USENIX ATC'14, USENIX Association, p. 305–320.

[18] RIADY, Y. Distributed sagas for microservices. *Yos Riady · Software Craftsman* (Oct 2017).

[19] VEGETA. Vegeta. `https://github.com/tsenart/vegeta`, Github, Oct 2020.

[20] XID. Globally unique id generator. `https://github.com/rs/xid`, Github, Oct 2020.