Sentinel : Advanced Service Discovery For Multi-Tenant Distributed SOA

Varsha Abhinandan
Gaurav
Piyush Goel
Pravanjan Choudhury
{varsha.abhinadan,gaurav,piyush.goel,pravanjan.c}@capillarytech.com, Platforms Team @ Capillary Technologies

***Abstract*—In distributed SOAs, services depend on each other and expose multiple end-points for partitioning or load-balancing. The end-points are dynamic because of failures, relocation or auto-scaling. A client needs to discover such end-points and must be proactively notified of any changes. This paper presents a framework which provides functionalities such as discovery of partitioned services, custom end-point selection, de-centralized load balancing and authentication information distribution.**

***Keywords*—Service Discovery, Name Service, Load balancing, Sharding, High availability**

I. Introduction

In distributed service oriented architectures, each service delivers a set of functionalities and depends on other remote services to perform specific operations. Each dependent service in-turn requires other network services such as databases, caches, queues etc and creates a complex service network. For e.g. in a billing or incentive management system, processing the transaction, calculating incentives and sending notifications like sms or email are handled by separate subsystems and each of these three subsystems typically maintain their own databases. A medium scale distributed set-up may contain several such subsystems and 100s of services depending on each other. Further, each service may be internally sharded, load balanced and designed for autoscaling and thus exposes multiple end-points. A client must discover an end-point/instance efficiently and must be notified of any changes to the discovered end-point information. Within a small network, a **static registry** of all the services can be maintained (configuration files or a remote cache) but once the network grows, dynamic service registration and discovery becomes much more important in order to avoid service interruption in scenarios like frequent deployment of new services, auto scaling or service relocation in case of failures. This problem in general is called as **Service Discoverability** [1] and has been addressed in several large distributed SOAs [2]–[4].

In a service network, an end-point information is described as $\langle scheme, host, port, auth \rangle$ (For e.g. jdbc:mysql://10.23.45.1:3306?user=sentinel&pass=***) which a client must obtain to make a connection and request for a functionality. In **dynamic registry**, the above information is maintained against a fully qualified **Naming** convention [5]. A common example of Naming is Domain Name System (DNS) [6], where a domain name is resolved to an IP address. In DNS, the clients have to poll for all the information and thus cannot be notified of any changes efficiently. If a failure is detected, and a de-registration command is issued to DNS, there will be at least a few seconds before this information is propagated to the clients. Further DNS doesn't provide any mechanism for managing authentication information. Services such as X.500/LDAP [7] are generally used to complete the end-point information but they cannot propagate the authentication change efficiently.

Apart from the pure service discoverability problem, choosing the right end-point amongst all the available end-points is another criteria in common distributed systems. For e.g. amongst a set of replicated databases, a batch data processing engine may want to choose the one which is in near sync with the master database. Most solutions don't provide a straight forward way of integrating such a custom selection criteria. A client may also wish to distribute its requests between all the databases evenly. Such load balancing is of two types, **centralized** and **de-centralized** and has been addressed in different ways. Round-robin DNS load balancing is an example for a de-centralized load balancing. Several extensions in DNS protocol have been used to support dynamic routing. Spotify [4] uses SRV records for extending DNS capabilities. However such solutions suffer from the propagation delay. Centralized/proxy based load balancing solutions work very well for external user facing services like web-servers. For an internal service network, a common problem with all such load balancers is that for a set of N services, N load balancers are required and these have to be discovered for the clients. The number of mediators may further go up for simple reason of fail-over. Several approaches such as Bitly's NSQ lookup [8] and Serf [9] are based on distributed messaging platforms. Serf uses a gossip based protocol [10] for membership, failure detection and custom event propagation. However both solutions are natively weak/eventually consistent [11] while ensuring high-availability.

With the recent advancements in distributed co-ordination, wait-free locking has been popular to create a strongly consistent dynamic registry. In wait-free value retrieval, a client can obtain a value from the registry and can set a **watch** on it for listening to any change in the data. Doozer by MIT [12], Zookeeper [13], [14] and Etcd [15] by Apache provide strongly consistent [11] store for wait-free data retrieval. When the data changes, they notify the connected clients immediately (no polling), making it ideal for infrequently-updated data sets. This allows the clients to cache heavily at their end for fast read operations. Several service discovery solutions which uses the above properties have emerged. Smartstack [2] by Airbnb uses Zookeeper datastore and provides in-app service registry/discovery and de-centralized load balancing for service networks in cloud. Eureka [3] by Netflix provides a REST based service that is primarily used in the AWS cloud for locating services for the purpose of load balancing and failover of middle-tier servers.

Sentinel framework proposed in this paper aims at providing solutions for wide variety of publishing/discovery problems commonly encountered in distributed SOAs. The framework consists of a library for service publishers, clients, shard managers, service monitors, authentication managers in distributed multi-tenant environment. Sentinel stores its data in a highly available Zookeeper data-store and extends Apache Curator [16] framework for providing the above functionalities. In particular the following problems are addressed -

a) Partitioned/Sharded Service Discovery: Partitioning/Sharding of databases/services is a common way of scaling services in a distributed system. This is particularly relevant but not limited to multi-tenant systems. A client needs to correctly discover an end-point of a service that is serving the particular chunk of data/functionality. In a SOA, each service might have different sharding requirements(range or hash partitioning) and thus the policies can't be generalized. Shard information is generally managed by the services directly or through one or multiple **Shard Managers**.

b) Custom policies for end-point selection : After a pool of service instances of a partition are selected, the client should be able to apply a selection criteria on the run-time metrics of each end-point and select a set of appropriate instances. This selection criteria may vary across clients. For e.g. many databases have read replicas for redundancy and they serve read only workloads. In such a setup, the client may want to exclude a set of instances where the replicas are lagging behind a certain threshold. These metrics (in this case a replication lag for each end-point) are typically owned by separate **Metric Collectors** or can reside in the publishing application itself.

c) Mediator-less load-balancing : In Sentinel, we aim at providing mediator-less decentralized load balancing without the need of a routing proxy in-between the clients and the service end-points. After a set of eligible service end-points are selected in the above stage, different load balancing strategies such as Round-Robin, Weighted Balancing on a metric or Sticky (for state-full applications) can be used to select a particular end-point. For e.g., to achieve load balancing among a set of postfix email gateways, a weighted balancing on the size of the pending mail-queue can be an efficient strategy.

d) Auth-change management for services: Certain services like mysql, postfix, mq etc require authentication information along with location details for the clients to connect. An **Authentication Manager** owns such information and changes the information periodically for security purposes. Such changes must be notified to all the clients so that they can reconfigure themselves.

The rest of the paper is organized as follows. We explain the notations and an overview of the Sentinel framework in Section II. Some of the implementation details of Sentinel are included in Section III. In Section IV, we discuss a few practical use-cases of Sentinel that are commonly encountered in distributed SOAs. A set of experimental results are presented in Section V. Section VI suggests future directions and concludes.

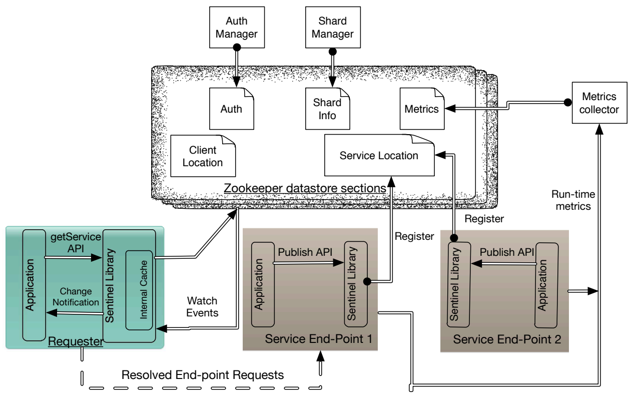

Fig. 1: Sentinel High Level Diagram

II. Sentinel Service Overview

A. Terminology and Notations

Let us consider a network of n services $\{S_1, S_2 . . S_n\}$. Each service may be partitioned/sharded with different sharding strategies called $Policy$. Let us assume the service S_i has h partitions $\{P_1, P_2 . . P_h\}$. In multi-tenant platforms, one datastore typically contains collected data of multiple tenants and a $Tenant\_ID$ serves as a shard key K. A set of such keys correspond to one partition (for e.g. $K_1 \rightarrow P_1, K_2 \rightarrow P_1, K_3 \rightarrow P_2$) which can be hosted by one or more instances of S_i. Each such instance S_i^k has a unique $Label_k$, and can be uniquely identified by I=$\langle Name_{s_i}, P_y, Label_k \rangle$.

A client C_j requests a service S_i with a Naming pattern; $Name_{s_i}$and a key K for which the data/services are required. The endpoint information of S_i^k returned to C_j can be expressed as $\langle scheme, host, port, Auth_{ij} \rangle$ which is similar to a URI notation. The $host, port$ of the URI is called **location information** and changes on service migration. Authentication information, $Auth_{ij}$ is specific to a client and a service combination. Additionally, the client specifies a $SelectionPolicy$ (Among standard or custom selection policies) to choose the final end-point. These selection policies are typically based on a set of service end-point metrics M_i^k (for each S_i^k), published by the **metric collectors**. This **dynamic registry** in Sentinel is maintained in Zookeeper znodes. Sentinel synchronizes the znode data of different Zookeeper sections in memory ($Internal\_Cache$). The synchronization delay between the Zookeeper sections and the $Internal\_Cache$ is termed as $T_{propagation}$.

B. Sentinel Architecture

The High level diagram of Sentinel is shown in Fig. 1. Sentinel uses Zookeeper to store and mediate service endpoint information. This data-store has various sections such as services, auth, metrics, shard_info and clients. These sections are used by **service publishers** to register and deregister their availability, by **authentication manager** to add or update the authentication information of the services which requires authentication, **shard manager** to publish the shard information, **metric collectors** to periodically publish the run-time metrics of different instances of the services. When a **client** uses sentinel library APIs to discover the endpoint information of a service, relevant sections related to the requested service are fetched and stored in the $Internal\_Cache$. Any subsequent changes in the sections are delivered to the client asynchronously (with wait-free value retrieval) without any need of polling. In rest of this section, we explain the data-store structure and the APIs.

C. Sentinel data-store Structure on Zookeeper

Sentinel data-store sections are organized in a hierarchical structure in znodes (zookeeper nodes) which are similar to a directory structure. The first level of znodes in this hierarchy are $/services$, $/auth$, $/shard\_info$, $/metrics$ and $/clients$, each pertaining to one of the section explained in the previous section. This hierarchy and its content is shown in Fig. 2. The nodes in Zookeeper are of two types; a) Persistent b) Ephemeral. Ephemeral znodes exists as long as the session that created the znode is active. Sentinel uses ephemeral nodes for information such as instance availability or client book-keeping. **Data watch** is set on nodes for which change in the stored data in the node has to be notified to the clients and **child watch** [17] is set on the nodes for which change in the children nodes are to be notified to the clients.

1) $/services$: This path stores the location information of all service instances. This path has a persistent child znode $/services/Name_{S_i}$ for each service S_i . An ephemeral node for each instance S_i^k of S_i exists under this persistent znode which stores the location details of S_i^k. A **child watch** is set on$/services/Name_{S_i}$ to get notifications on instance registration and de-registration.
2) $/auth$: Each service which requires authentication has a persistent child znode under this path, i.e $/auth/Name_{S_i}$ for each S_i. $Auth_{ij}$ is specific to the client and a service. Hence for each client C_j of S_i, a persistent znode $/auth/Name_{S_i}/C_j$ stores $Auth_{ij}$ using asymmetric public/private keys. A **data watch** can be set on this znode to get notifications for $Auth_{ij}$ change.
3) $/shard\_info$: The path stores one persistent node of each type of sharding policy supported by Sentinel. In case of range partitioning sharding policy, each K_y corresponds a persistent child znode $/shard\_info/policy/K_y$ and stores P_y as its value. A **child watch** is set on $/shard\_info/Policy$ to get notifications for (K_n, P_y) changes.
4) $/metrics$: The run-time metrics of each S_i^k of S_i is stored as a persistent node $/metrics/Name_{S_i}/P_yLabel_k$. The data stored in this node is a simple map of the metric key and values. It should be noted that this node is created by a separate metric collector which publishes on behalf of the instance, hence persistent znodes are created instead of ephemeral znodes. The combination $Name_{S_i}, P_y, Label_k$ uniquely identifies an instance (Section. II-A).
5) $/clients$: This path stores the information required to identify service dependencies. This path has a persistent child znode $/clients/Name_{S_i}$ for each service S_i . An ephemeral node for each C_j of S_i exists under this persistent znode which stores the location details of C_j

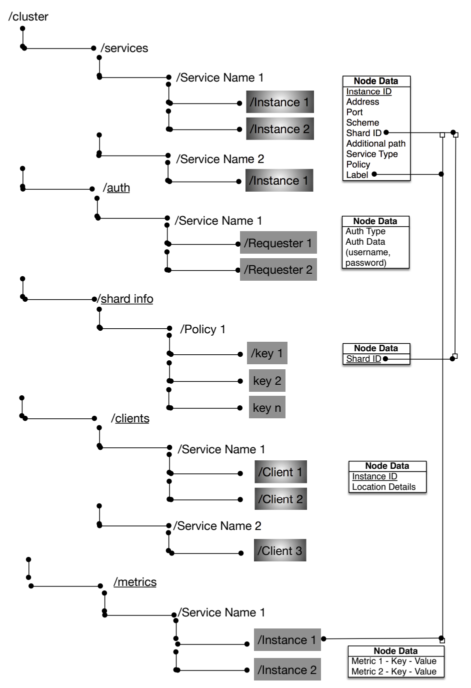

Fig. 2: Sentinel Znode section structure

D. Sentinel APIs

We now explain the Sentinel APIs and the usage in the respective modules.

1) Service Publishers: The service publishers use the below two APIs exposed by Sentinel:

1) $registerService(S_i^k)$:S_i^k publishes itself by providing its location details and the partition P_y it belongs to. Sentinel stores this information as an ephemeral znode data in the $/services/Name_{S_i}$ path.
2) $deRegisterService(S_i^k)$: S_i^k deregisters itself by using this API which removes the corresponding ephemeral node. This action results in all the dependent clients being notified about the service unavailability.

2) Authentication Manager: The $Auth$ addition and its periodic update for a service is owned by the Authentication Manager. It using the $updateAuth(Name_{S_i}C_j, Auth_{ij})$ API exposed by Sentinel to store $Auth$ information the of service S_i for the client C_j under the $/auth$ path. This action triggers a service-change event and is notified to the client C_j of S_i .

3) Client: The end-point information of the service S_i required by C_j to connect to a particular instance of S_i can be discovered by using the $getService(Name_{S_i}, K_y, SelectionPolicy)$ API which returns to the client with $\langle scheme, host, port, Auth_{ij}\rangle$ which comprises of the location details fetched from $/services$ path and the $Auth_{ij}$ fetched from$/auth$ path . The service resolution flow is explained in detail in the Section. III.

4) Shard Manager: Shard Manager allocates shards for services based on the $policy$ defined for each service. It integrates with Sentinel to push the (K_n, P_y) pair of a $Policy$ into the $/shard\_info$ path by using the API $updateShardInfo(K_y, P_y)$. This action triggers a key-shard change event and propagates the change to the listeners of this event.

5) Metric Collectors: Metric Collectors send run-time metrics of each S_i^k and publish them to Sentinel at regular intervals using the API $updateMetrics(S_i^k, MAP\langle Metric, Value\rangle)$.

III. Sentinel Implementations

A. Service resolution

The steps involved in Sentinel to discover the endpoint information of a service is presented as a sequence diagram in Fig. 3. This section explains the procedure in the following three steps -

1) Step 1: In the first step, a client C_j requests the Sentinel library to provide an instance of service S_i by calling the $getService(Name_{s_i}, K_j, selection\_policy)$ method. If this API is called for the first time, Sentinel library first sets a watch on the $/services/Name_{s_i}$ znode and fetches all the instances of S_i. Then it fetches the $Auth_{ij}$ from the $Auth$ section of the Zookeeper. Depending on the sharding policy that the service supports ($policy$), it fetches the (K, P) pairs from each of the znodes in the $/shard\_info/policy/$. The combined information is stored in the $Internal\_Cache$ which is used for all subsequent calls.

2) Step 2: In this shard filtering step, the partition (P_y) for the key (K_j) is identified by consulting the (K, P) combinations. All the instances that belong to the partition P_y are selected for the next step.

3) Step 3: For each instance (S_i^k) in the selected list, the list of metrics M_i^k is fetched and updated in $Internal\_Cache$. These metrics are sent to the $Selection\_Policy$ class for final selection of the instance. The location and $Auth$ details of the selected instance are sent back to the client as the response of the API.

It can be noted that in any of the Zookeeper data fetches, a watch is set on the corresponding node. In case of any changes to the data, Sentinel gets a watch event to update the corresponding section of $Internal\_Cache$, asynchronously. This update has a propagation delay of $T_{propagation}$ which though small should be handled in specific cases. One of such cases is when a new K_n is added to an existing partition P_y and a new key pair (K_n, P_y) is created by the shard manager. Additional Zookeeper calls are required (Step 10 of Fig. 3) to fetch the (K_n, P_y) and explained as a procedure in the next subsection.

B. Propagation Delay and Measures for specific cases

In Sentinel framework, the look-up of services inside $Internal\_Cache$ is crucial for fast read operations. This cache update happens by using Zookeeper watch events which is efficient but may suffer from a small amount of $T_{propagation}$ delay. We now discuss the cases where the lag can be tolerated or needs to be handled to marginalize service disruption.

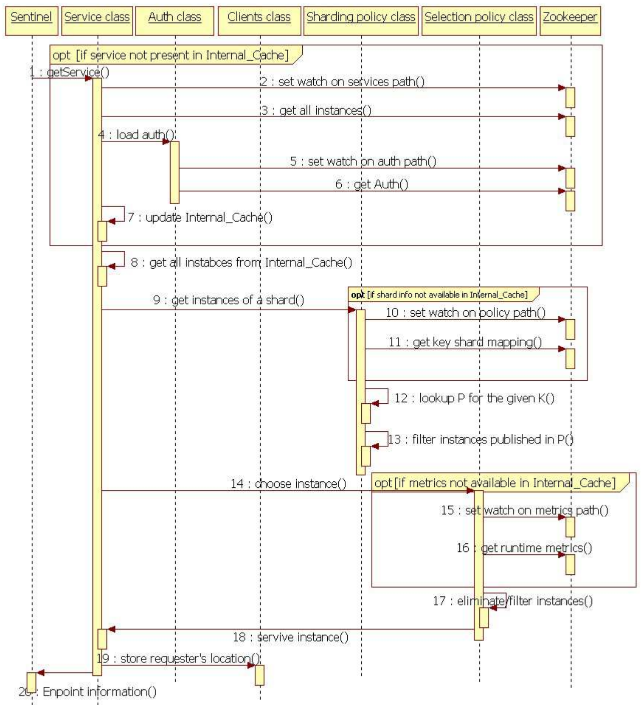
Sentinel
Service class
Auth class
Clients class
Sharding policy class
Selection policy class
Zookeeper
opt [if service not present in Internal_Cache]
1 : getService()
2 : set watch on services path()
3 : get all instances()
4 : load auth()
5 : set watch on auth path()
6 : get Auth()
7 : update Internal_Cache()
8 : get all instabces from Internal_Cache()
9 : get instances of a shard()
opt [if shard info not available in Internal_Cache]
10 : set watch on policy path()
11 : get key shard mapping()
12 : lookup P for the given K()
13 : filter instances published in P()
14 : choose instance()
opt [if metrics not available in Internal_Cache]
15 : set watch on metrics path()
16 : get runtime metrics()
17 : eliminate/filter instances()
18 : servive instance()
19 : store requester's location()
20 : Enpoint information()

Fig. 3: getService API flow

1) New Key addition to an existing partition - If a new (K_n, P_y) pair is added by the shard manager and a service request for K_n is originated before the $Internal\_Cache$ is updated, Sentinel library fetches this mapping from Zookeeper (Procedure 1). This makes the new key addition to be consistent across shard managers and the clients at the expense of an additional Zookeeper network call. However, it doesn't update the $Internal\_Cache$ with the value received and waits for the watch event to update the $Internal\_Cache$. The reason for doing this is to ensure the writes to the $Internal\_Cache$ happen in the same order as it happened on the Zookeeper. Update of a key with a new value (usually in the case of shard balancing) or removal of a key (deletion of a key from a partition) typically happens as a supervised offline process and hence the $T_{propagation}$ is not relevant.
2) Service End-point Migration - In the case where an end-point is migrated from one location to another, a new instance of the service is first published on the new location. Then, the old service end-point deregisters itself from Sentinel and waits for the $T_{propagation}$ time before closing itself for the clients.
3) Service End-point Unavailability - If the service end-point becomes unavailable in an unsupervised situation, the ephemeral node of the instance times out in the Zookeeper. The Sentinel library in each connected client get notified of the unavailability and switches to another end-point. During this propagation delay, the connected clients continues to request the unavailable end-point. This case generally due to client timeouts

or sudden loss of service is expected to be handled by the clients itself.

Procedure 1 Procedure for new Key addition to an existing Shard

```
Let $K_n$ be the shard resolution key by the client
if $Internal\_Cache$ contains the mapping of $K_n$ then
    Resolve $P_y$ from $Internal\_Cache$ for $K_n$
else
    Fetch $P_y$ value from the Zookeeper node $/shard\_info/policy/K_n$
end if
{After $T_{propagation}$, the $Internal\_Cache$ contains $P_y$}
```

IV. Use-cases and Applications

A. Best Database Replica Selection

Let us take a use-case of best read replica database selection problem for a batch data processing engine for read-only workloads. Each read replica S_i has a time lag (L_i) with its master (write) data set. The batch processing engine has a cut-off (L_c) on the lag and must use the replica with minimum lag. The criteria to select the correct service S_m is thus, select $L_m = Min(L_i)\ \forall L_i < L_c$. In Sentinel, this is achieved by using a custom selection policy that performs the above operation. If all replicas have lag more than L_c, the Sentinel library doesn't select any instance to the batch-data-processing engine. Further, it is possible the selected replica at some future time starts to lag more than the other replicas due to the workload of batch-data-processing engine. Sentinel library evaluates the selection criteria with each request of the batch-processing engine and sends back a better replica end-point if available at a later time.

B. Load balancing and Auto-scaling of Postfix

Next, we take an example of multiple clients relaying the emails through multiple postfix servers. The setup is similar to the previous example except that the metrics is based on the size of the pending queue (mailq) at each point and a weighted load balancing is performed for the correct postfix end-point selection. Hence, the postfix server that has large pending queue receives the least request. If the postfix server doesn't recover, the Sentinel library inside the auto-scaling modules triggers a scaling action that launches a new machine and in the start-up the machine registers as an another end-point of the same service. The unavailability of the degraded postfix server and the availability of new end-point is propagated to the Sentinel library running as part of the clients and each client starts to relay their mails to the available set of servers.

C. Authentication Rotation

Authentication rotation for all the services is a common task for authentication managers. With Sentinel, we propagate the $Auth$ changed information to the clients, however, this relay of information though efficient, may incur disruption in the service for few seconds. Thus the authentication manager never modifies the existing $Auth$ (For e.g. changing password of a user name) and always creates a new $Auth$ for the client and uses Sentinel to propagate to the client. Procedure 2 explains the steps of this process.

Procedure 2 Authentication Rotation Procedure

```
Let $Auth^1_{ij}$ be the existing Auth between $S_i$ and $C_j$
Create a new $Auth^2_{ij}$ in the service $S_i$ in addition to $Auth^1_{ij}$ for $C_j$
Call $updateAuth(S_i, Auth^2_{ij})$
Wait for estimated $T_{propagation}$ {Between this time $C_j$ may connect with $Auth^1_{ij}$ or $Auth^2_{ij}$}
Authentication propagation is complete and $C_j$ re-initiates connection to $S_i$ with $Auth^2_{ij}$ {For e.g. re-initiates database pool }
Remove $Auth^1_{ij}$ from $S_i$
```

TABLE I: Percentile Distribution of Sentinel APIs

Sentinel API	Percentile Distribution in (ms)				
	80	85	90	95	99
registerService	205.2	217	300.9	498	599
getService	0.015	0.016	0.017	0.017	0.968
deregisterService	75	83	89.1	100	125
updateAuth	40.2	45.9	54.6	73.3	105.16
updateShardInfo	30	34	49	67	116.02

D. Service Dependency Analysis

Another common problem in a large SOA is to find the dependencies between the service mainly for two reasons 1) Understanding the dependent services that are affected when a certain service becomes unavailable and 2) Detecting the causes of a service failure by looking at the services it depends on. In Sentinel, we can build the network dependency tree dynamically by getting the required information in the $/services$ and $/clients$ paths.

V. Experiments and Results

We now present experimental results that were conducted on a setup of 200+ network resources (service end-points), 35+ clients and a 5 node Zookeeper ensemble. We collected various metrics of all the clients for a period of several days. Sentinel framework has been designed to efficiently propagate infrequent service change/write to the client while maintaining low latency for service resolution. In the following subsections, we evaluate the proposed scheme on the collected metrics.

A. API Response time

The percentile distribution of the API response time is shown in Table I. The getService API is called over 100 million times a day from various clients and it can be seen that the 95^{th} percentile is below $20\mu s$. This low latency is achieved due to the use of the $Internal\_Cache$. The latency jump in the 99^{th} percentile is due to discovery request for new services (4 Zookeeper reads/per call) or unavailability of the requested Key in the $Internal\_Cache$ (One additional Zookeeper read/call) (Procedure 1). It can be observed that getService API has significantly low latency as compared to the other APIs that involve write operations to the Zookeeper indicating a change in any of the service information. Next, we study the number of such writes in a production environment and the amount of time it takes to propagate the change to the clients.

B. Change propagation

In this experiment, we obtain an experimental upper bound on $T_{propagation}$. Fig. 4 shows the 95^{th} and 99^{th} percentiles of

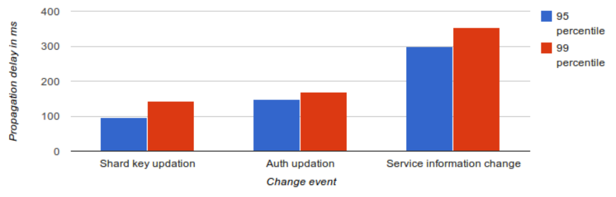

Fig. 4: Change propagation delay

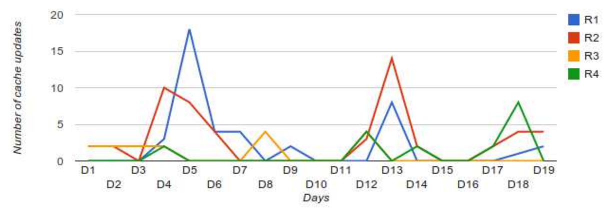

Fig. 5: Cache change count distribution

$T_{propagation}$ for three such types of change propagation. A service change typically requires fetching additional sections from Zookeeper and hence has a comparatively higher propagation delay. We found the 99^{th} percentile of all such delay to be less than $500ms$. It has to be noted that in case of a process unavailability (typically a crash in an unsupervised scenario), the ephemeral node in the Zookeeper timeouts in 30 secs (configurable), which might cause additional delay in the end-to-end propagation of change.

In Fig. 5, we plot the number of times a change that has occurred in the network that needs to be propagated to the clients. The numbers in this result depend on the nature of the application running on the network and will differ from one set-up to the other. In the set-up of this experiment, over a period of 19 days, it is observed that the number of such changes leading to cache refreshes are less than 20 for any day. Additionally, metric collectors send metrics periodically that leads to updates in the clients' caches, however, the $T_{propagation}$ is usually much less compared to the metric collection frequency.

C. Summary of Results

The results indicate that end-point information may change multiple times in a large SOA and with Sentinel library they can be propagated to the clients in less than $500ms$. The requests for end-point resolution is usually very high $(100M+)$ and with Sentinel library, resolves within $20\mu s$.

VI. Conclusion & Future Work

This paper presented Sentinel framework that supports advanced service discovery patterns like partitioned/sharded service discovery, custom selection, load balancing and authentication distribution for multi-tenant distributed SOAs. A few critical procedures, use-cases and experimental results on real-world production set-up were discussed in this paper. There are ongoing enhancements to the Sentinel library to support various other selection policies and auto-scaling requirements.

Acknowledgments

We would like to thank the various technology groups inside the company for adopting Sentinel and establishing production readiness. The open source community, mailing groups and numerous content around the Internet have helped in guiding the design of Sentinel.

References

[1] Wikipedia, "Service discovery," http://en.wikipedia.org/wiki/Service_discovery.

[2] M. R. Igor Serebryany, "Smartstack: Service discovery in the cloud," http://nerds.airbnb.com/smartstack-service-discovery-cloud/.

[3] Netflix, "Aws service registry for resilient mid-tier load balancing and failover," https://github.com/Netflix/eureka.

[4] S. Labs, "In praise of boring technology," http://labs.spotify.com/tag/dns/.

[5] W. Nutt, "Distributed systems - name services."

[6] P. Mockapetris and K. J. Dunlap, *Development of the domain name system*. ACM, 1988, vol. 18, no. 4.

[7] T. A. Howes, M. C. Smith, and G. S. Good, *Understanding and deploying LDAP directory services*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[8] Bitly, "Nsq - a realtime distributed messaging platform," https://github.com/bitly/nsq.

[9] Serf, "Serf - service orchestration and management tool," ihttp://www.serfdom.io/.

[10] A. J. Ganesh, A.-M. Kermarrec, and L. Massoulié, "Peer-to-peer membership management for gossip-based protocols," *Computers, IEEE Transactions on*, vol. 52, no. 2, pp. 139–149, 2003.

[11] W. Vogels, "All things distributed," http://www.allthingsdistributed.com/2008/12/eventually_consistent.html.

[12] MIT, "Doozer - a consistent distributed data store." https://github.com/ha/doozerd/.

[13] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems." in *USENIX Annual Technical Conference*, vol. 8, 2010, p. 9.

[14] F. Junqueira and B. Reed, *ZooKeeper: Distributed Process Coordination*. " O'Reilly Media, Inc.", 2013.

[15] A. Etcd, "Etcd - a highly-available key value store for shared configuration and service discovery," https://github.com/coreos/etcd.

[16] Apache, "Apache curator - a zookeeper keeper." http://curator.apache.org/.

[17] Z. documents, "Zookeeper watches," http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html#ch_zkWatches.