Event Manager

Design Document

Assumptions:

As the problem statement was too abstract, I have made few assumptions before proceeding with the implementations. Below are the assumptions made.

- I went with an approach of building a web application inst

1.What is Event Manager

Event manager is an application which monitors the incoming requests from clients and persist the event data like ip-address, last request time etc. It keeps monitoring to see any client is idle without any activity for the configured amount of time and log the message for the same.

2.Requirements

Event manager should meet the following requirements:

Functional Requirements:

1. It should monitor the incoming requests and persist the event data.
2. It should identify the clients who are not doing any activity for the specified amount of time and log the client information.

Non Functional Requirements:

1. System should be highly scalable.
2. Monitoring shouldn’t impact significantly to the actual request processing.

3.Capacity Estimation

Let's assume on average we get around 100 million requests per day with 5 million daily active clients.

Storage Estimates:

Let say each event needs around 40 bytes of storage to store the event metadata(like ip-address, method, timestamp).

100M * (200) = 4Gb/day

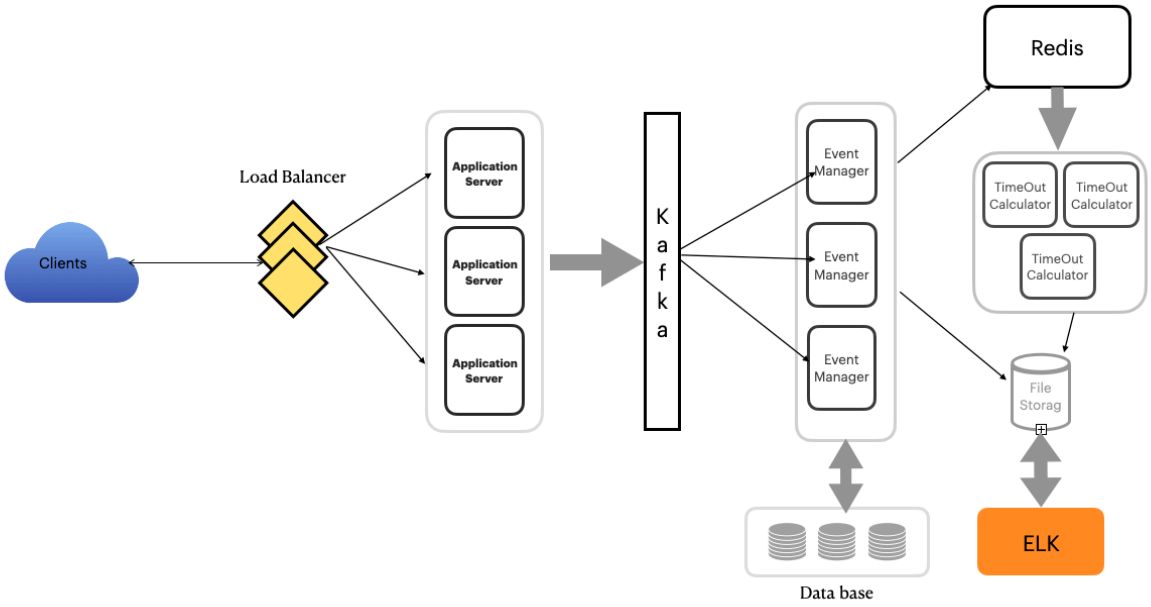
What would the storage need for 5 years.

4 * 365 * 5 = 7300Gb

4.High level design

We need a system that can handle more than 100 million requests a day. It's clear that we need a scalable system.

At a high level we need multiple application servers to serve all these requests with load balancers in front of them for traffic distribution. On the backend we use kafka messaging system to achieve message persistent and asynchronous processing. On the data base front we need efficient database that can support huge writes.



On every request, we log an event into Kafka messaging system which is consumed by Event manager and persists the metadata into database and updates the timestamp of the client in redis. A scheduled job will be running periodically to check for timeout of the clients.

5.Basic system design and algorithm

As the system is distributed in nature, requests from the same client can land in any of the system and it would be tough to identify which system hold the latest activity timestamp. We chose Redis data store as centralised data store to store the last activity timestamp for each client. Which is used to identify the clients who are inactive for the specified amount of time.

Let's take an example, whenever a request comes in, event manager performs the following steps:

1. If the client id(ip-address) is not present in the Redis Map structure, insert and set the activity timestamp to current time.
2. Otherwise find the record of the key(ip-address) and update the activity timestamp with current time.
3. We can set the TTL while inserting the records into the map. We can choose to have this value little higher than the configured idle timeout so that these records will be evicted even incase they were not removed by job.
4. We use Distributed locking mechanism to read and write into the Redis datastore to avoid race conditions.
5. We scheduled a job which periodically process the records in the Redis store to see if any client passed the idle timeout configures. If so it logs the activity into logs and removes the client from the Redis data store.
6. This run on a fixed window timeline.

6.Data Sharding

Since we have huge number of events every day and our write load is too high, we need to distribute our data onto multiple machines such that we can read write efficiently. We can shard the data based on

Geo location: we can shard the database based on the geo location

Ip-range : we can shard based on the ip-ranges.