Photo Credit annca

An Approach to Designing a Distributed, Fault-Tolerant, Horizontally Scalable Event Scheduler

Introduction

Sandeep Malik
Nov 24, 2016 · 8 min read

Designing a time based event scheduler is always an interesting problem. Doing that at scale makes it even more challenging. First of all, let's define what we mean by time based event scheduler?

A time base event scheduler is a system that can be used by a service to schedule a request that needs to be processed in *future*. The service registers an *event* in the scheduler and suspends the processing of current request.

When the stipulated time arrives, the requesting service is notified by the scheduler and former can resume processing of the suspended request. There are many scenarios in which such a system can be useful. Some of the use cases, which by no means are exhaustive, are listed below

Timing out async requests

As more and more systems move towards messaging based architectures like Kafka, JMS, etc., one of the most common scenarios is using messaging tier as asynchronous request/response bus. One system produces a "*message request*" to another system and when the response is returned by the latter, it proceeds with the rest of the processing. In some of the cases, it's important for the caller to have '**eventual response**'. It is possible that the other system has a lag, or is down, or simply misses to return back a response. In a distributed environment with a stateless micro-services paradigm, it is difficult to keep track of such *lost* requests. One solution could be to schedule a 'time-out' event, which can be triggered at stipulated SLA of the request.

When the event is triggered the current state of the request can be checked and if the response is missing then the request can be marked timed out.

System retries

Even in case of systems that interact over a synchronous channel like HTTP, it is possible that the server is down and request can not be served. A client can choose to retry, however, if the retry intervals are very short, it is unlikely to have any net positive effect. For example, if it's a system wide outage it is likely to remain so for more than a few seconds. In such cases, it might be desirable to do longer, exponential retries for which the request data may not be kept in memory.

Under these circumstances, the payload for retry can be sent to an event scheduler and can be retried later depending on the client's retry policy.

Reconciliation jobs

Sometimes in distributed systems based on batching, the data streams are split and processed by multiple clusters as micro batches. In case of failures, it may be desirable to reconcile responses so that a coherent response can be returned to client for the initial batch.

In such cases, a reconciliation event can be scheduled at the beginning of the start of the batch processing so that data can be reconciled at the end of the batch processing or the SLA of processing time.

Price related triggers

In an eCommerce world, price changes are generally made to keep the product at competent price value. Triggering such price changes could be a huge manual effort, and error-prone, and may not happen precisely at the desired time. For example, even a minute lag of price change during the peak holiday hours like Black Friday can potentially impact the annual sales.

In such cases, the price changes can be set up as 'promotions' ahead of time, which can be triggered by the event scheduler at the required time thereby saving a manual price-synchronization effort.

All of the above mentioned issues were faced by Walmart at one point or another and it became clear for us that we need to invest into a system that can keep track of *suspended requests*. Since the scale at which Walmart operates is huge, the scheduler had to be fault tolerant, and horizontally scalable.

Existing options

We started looking at various existing open source solutions like Quartz. While Quartz is a proven solution for time based scheduling with rich functionality, our problem statement was a bit different

- Many of our applications are using Cassandra as the NoSql data store. Over time we have grown considerable expertise in Cassandra and its nuances. Quartz does not seem to have a JobStore for Cassandra (or at least we couldn't find one)
- We wanted to have tunable consistency for events. For some events like system retries, etc, it was fine to write them to one node (LOCAL_ONE) but for others like promotion triggers, it was required to use EACH_QUORUM so that no event was lost.
- Scaling out the cluster was harder. We had a requirement to scale to millions of events per hour.

We also looked at Hazelcast-based JobStore implementation of Quartz but the prolonged garbage collection cycles was a risk for in memory data there by resulting in misfires.

Introducing BigBen

We set out to implement our own solution, which is called BigBen, inspired by the very famous London Clock Tower. BigBen offers the following capabilities

- Master slave design, so that master can decide the best strategy for distributing load across the cluster

- Extremely fault tolerant design so that if a given master goes down another can be selected almost instantaneously
- Uniform partitioning of events data in data base so that there are no hot pockets in terms of fat partitions in Cassandra
- Uniform execution of events on the cluster so that all nodes share *almost* equal load
- Maintain checkpoints so as to keep track of events that were not triggered or failed execution (mis-fires)
- Integrate with HTTP and Kafka as *default* channels for events
- Multi-tenancy. Each events belongs to one tenant, and each tenant can define its policy of how to store (consistency guarantees) and/or process the event.

Design and Architecture

BigBen is based on a 'micro-batching' architecture. The events are bucketed together in a window and these buckets are stored as partitions in data base.

When the cluster comes up, a master is selected, which starts scanning buckets at a defined granularity. The minimum granularity supported is one minute, which means the master will scan for the events every minute.

Event ingestion

Since it is unknown how many events will be scheduled every minute, partitioning the data based on bucket width (in this case one minute), may result in an uneven load on the data base. The events are therefore shard*ed* and only 1000 (configurable value) events are stored in 1 shard.

For example, say at 10:00 AM, 2030 events were received, then those events will go to three shards. Shard 1 will contain 1000 events, shard 2 will contain another 1000, and shard 3 will contain 30 events. This way the system guarantees that there are no more than 1000 events per shard.

To know which shard a given event will be assigned to, depends on the running total of events for that bucket. It is then very simple to calculate the shard index for that event using the formula below:

> *shard_index = (total_events / shard_size)*

for example, when next event is received after 2030 events, it will go to

> *2030/1000 = 2nd index (or 3rd shard)*

For fast lookup, this required maintaining a running total of event counts per bucket. BigBen uses embedded Hazelcast grid to maintain those counts in a distributed map, though other solutions like memcache, Couchbase, etc. which provide support for atomic counts, can be used as well. The map data is synced with Cassandra every few seconds.

The diagram below captures the event receive flow. The events can be received over HTTP or Kafka. Invalid events are rejected and returned immediately (shown by *red dotted line*). For others, the corresponding buckets are calculated (say the event is scheduled for 10:21:55 AM, then the bucket is 10:21:00 AM), the counts are incremented, and shard index is calculated. The event is then stored in the corresponding partition which is a combination of bucket and shard index (e.g. 10:21:00/1)

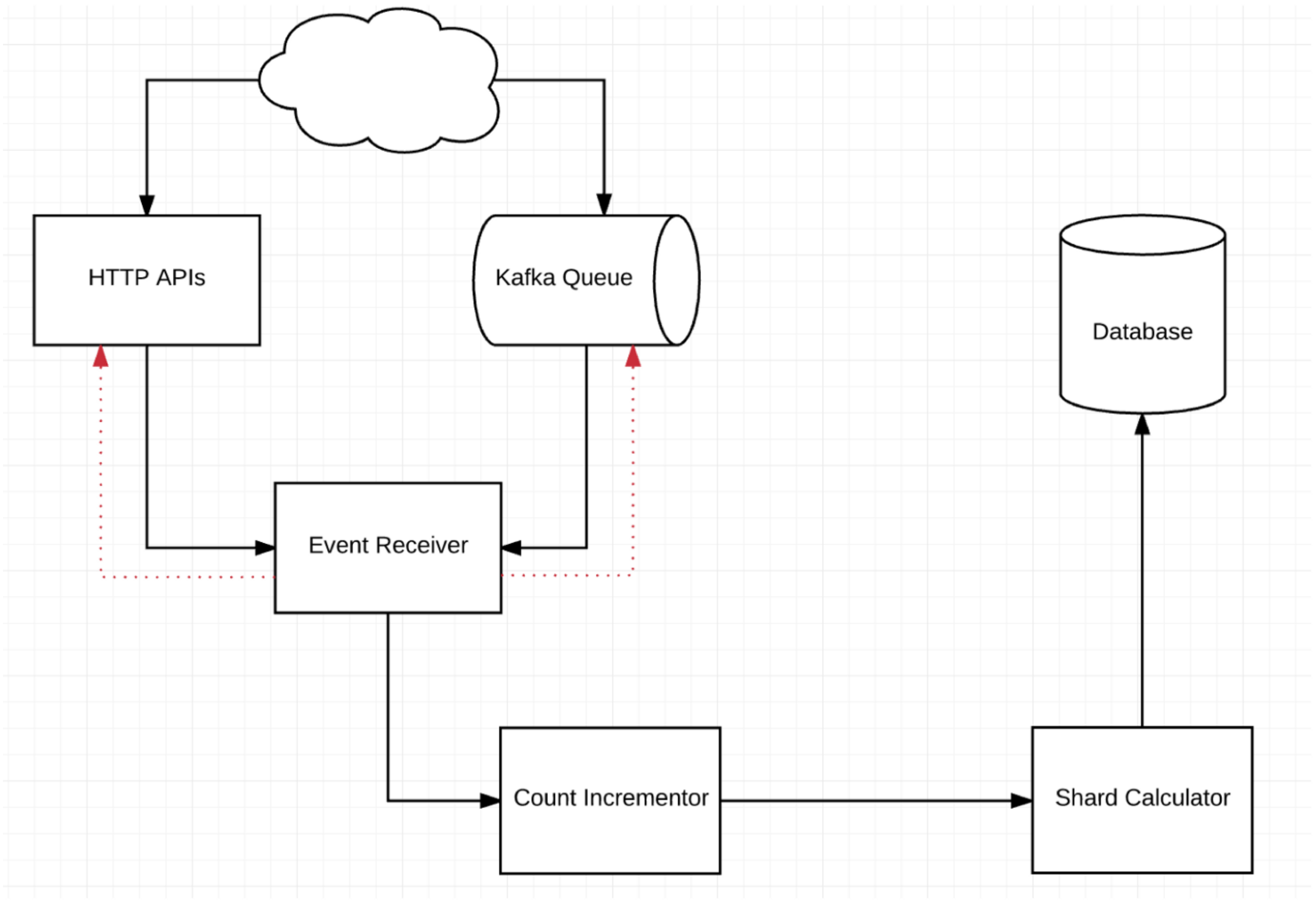

Fig 1 — Event Receive Flow

Event Processing

For the processing part, a master is elected among the cluster members. Zookeeper could be used for leader/master election, but since BigBen already uses Hazelcast, we used the distributed lock feature to implement a *Cluster Singleton*. The master then schedules the next bucket and reads the event counts. Knowing the event count and shard size, it can calculate very easily how many shards are in total. The master then creates pairs of (bucket, shard_index) and divides them **equally** among the cluster members, including itself. In case of unequal division, the master tries to take the minimum load on itself. For example, for 10:21 AM bucket, and say 6000 events, and 4 node cluster, the division would look like:

> *distribution => (bucket, array of shard indexes, node IP) => (10:21, [1,5], node1), (10:21, [2,6], node2), (10:21, [3], node3), (10:21, [4], node4)*

The master also checks if any previous buckets have failed and schedules those as well. Upon completion of the bucket, the bucket is marked processed and the status is updated in the checkpoint. The diagram below shows the processing stage (*schedule scanner is the master*)

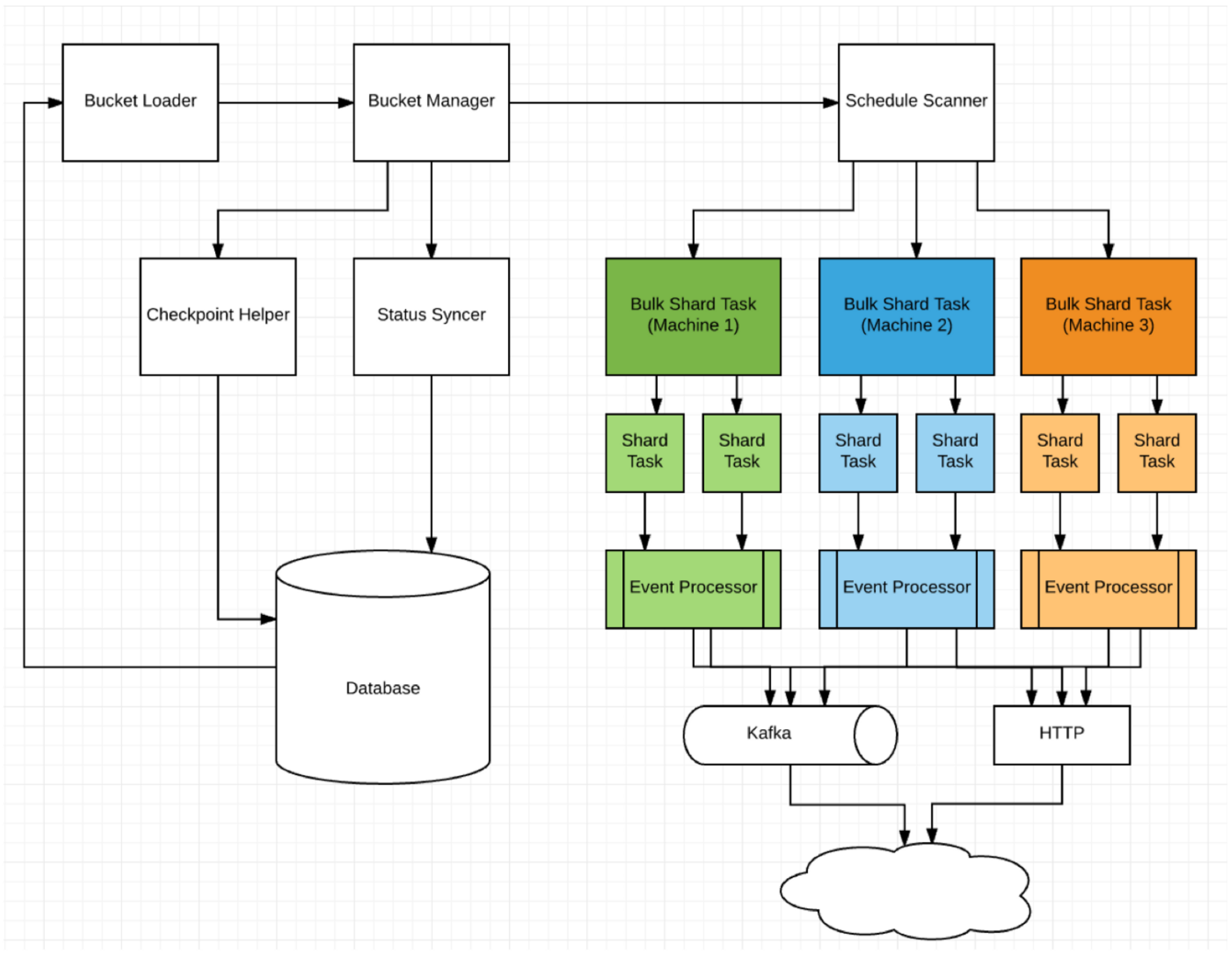

Fig 2 — Event Processing Flow

The various components in the processing workflow are:

- Bucket Manager: maintains the snapshots of loaded bucket in memory
- Bucket Loader: loads previous buckets in memory in a rate-limited/throttled way
- Checkpoint helper: syncs the checkpoint every few seconds
- Status syncer: syncs the status of bucket, once done

Fault Tolerance Design

As any other distributed system, there can be multiple failure points. Some of them are:

- Master becomes unavailable
- Events processing fails
- Data base is down

BigBen uses different techniques to handle failures. For the fault tolerance of master, we use a distributed lock in Hazelcast. Note that this lock is *never released* in the life time of the node. Only if the node goes down, the other member will try to acquire the lock. This provides for a very robust fault tolerance and we validated this during our Chaos Monkey testing. Below diagram shows the master fail over scenario

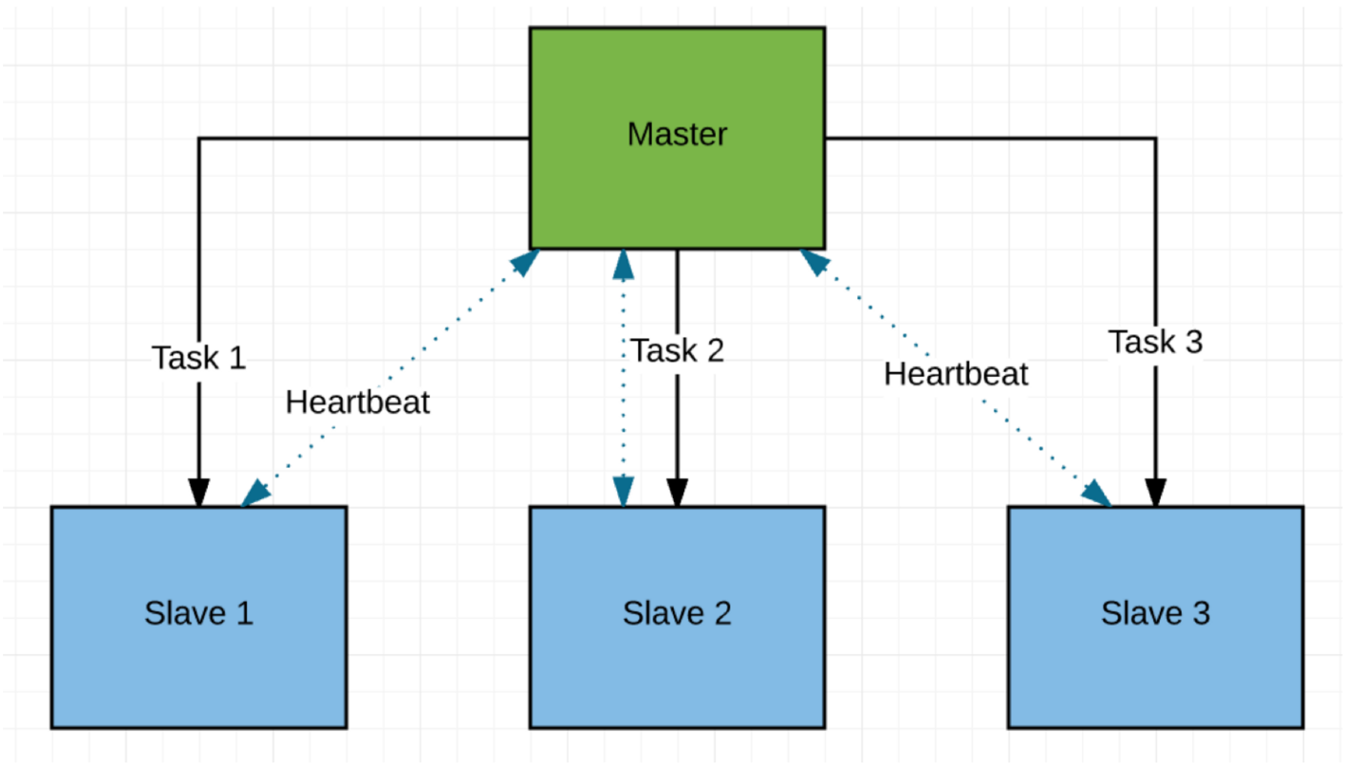

Fig 3 — Master (Cluster Singleton) distributing tasks among slaves

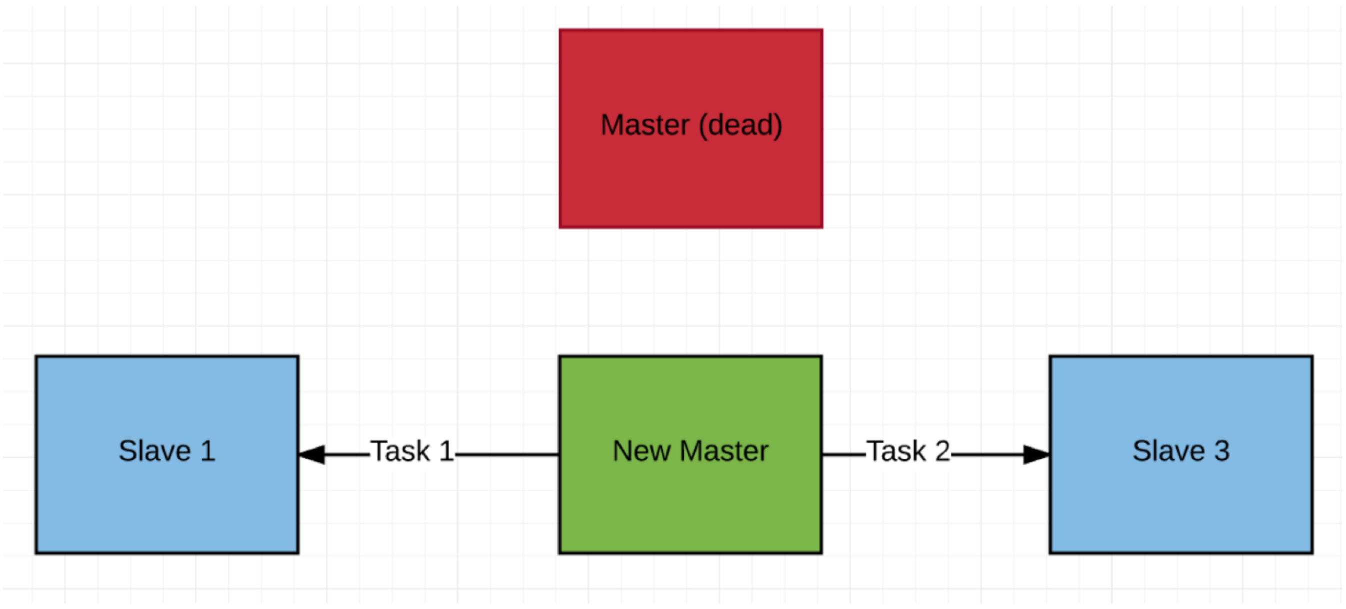

Fig 4 — Master down, new slave promoted to master, and starts distributing tasks

In cases where event processing is failed or data base is down, the checkpoint is marked with those failed buckets. On every new bucket scan, the master scans the checkpoint as well to see if any previous buckets are in failed state. If found, then those buckets are also scheduled.

BigBen uses a configurable '*look back*' parameter regarding how far in the past it has to go to keep trying for failed buckets/events. By default, this value is 1 day.

Performance benchmarking

We bench marked the cluster's performance with the following parameters:

- 4 machines, Cent OS 6+, 8 Cores, 16 GB RAM, min Heap 2 GB, max Heap 8 GB
- Event payload size: 500 bytes
- Cassandra 6+6 bare metal cluster
- Write Consistency: LOCAL QUORUM
- Read Consistency: LOCAL QUORUM
- Ingestion Rate: 1 Million events ingested in 52 seconds. => **70 Million Events per Hour**

- Processing Rate: 1 Million Events in 60 seconds => **60 Million Events per Hour**

Next Steps

I hope you found this post informative. ~~Keep an eye out, as we will be open sourcing BigBen very soon~~. We have open sourced BigBen. Let us know if any questions. Thanks for reading!

Big Data Programming Cassandra Distributed Systems NoSQL