CQS Architecture

High Level Architecture For CQS

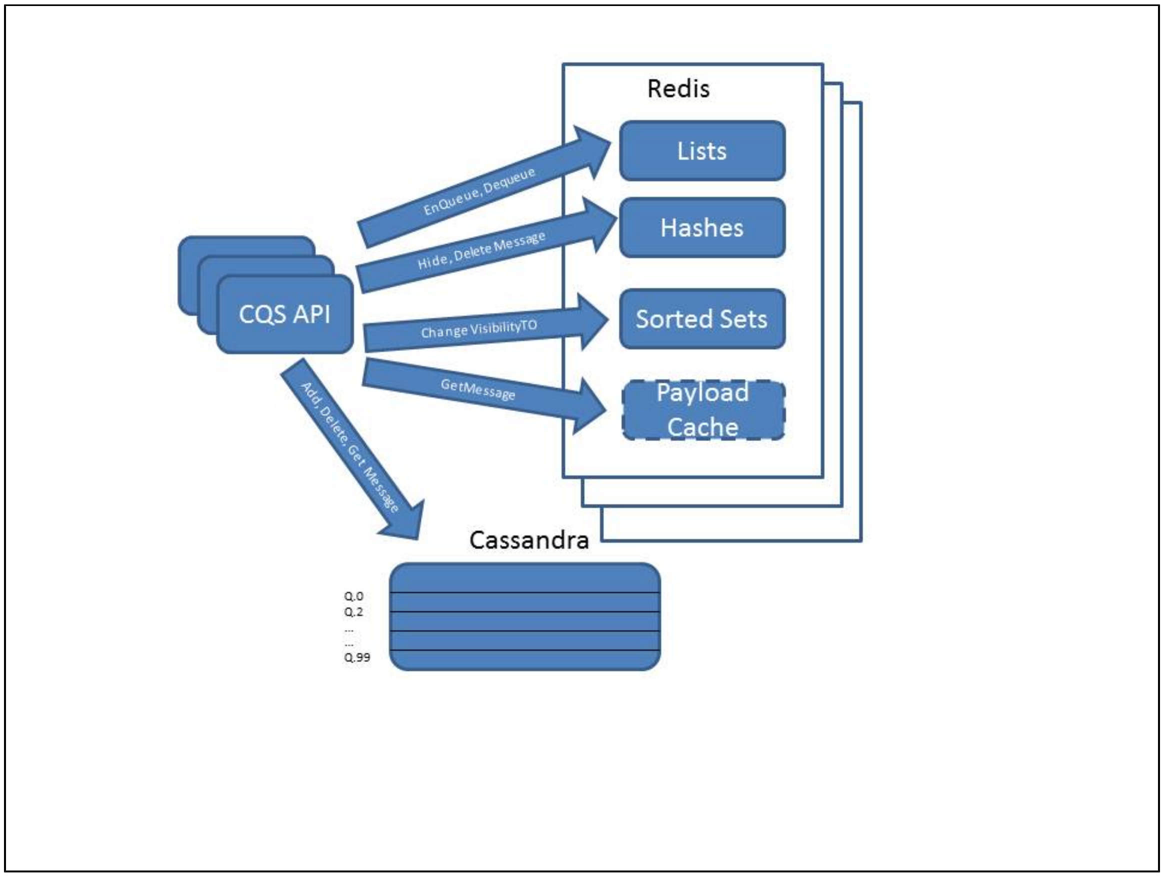
Overview

This document describes at a high level, the architecture for CQS. For details about CQS functionality and API, please refer to the Amazon SQS specification.

Design Challenges

We narrowed down our design challenges to the following:

- Avoid very wide rows in Cassandra. We noticed after a lot of testing that Cassandra has issues with rows having more than 500k columns while we were reading/deleting columns
- Mitigate the duplicate messages that can be returned on multiple concurrent receiveMessage calls. While we don't do our best effort, we do a good effort in reducing duplicates
- Minimize churn in Cassandra for performance reasons. The functionality for hiding, changing-visibility messages is best done outside of Cassandra
- Having a central location for queue is better from non empty response point of view
- Need very low latency for send/Receive/Delete Message calls (~10ms)
- Minimize duplicates
- FIFO-ness isn't important



Description

The architecture is described in three components. The API component, The Redis Component and the Cassandra component

API Component

This component implements the Amazon SQS specification and contains all the logic for using the Redis and Cassandra Components. It also provides the Admin interface

Redis Component

We hash the Queue-name to get the redis server where the queue cache data will live. We use the Redis List datastructure to represent a queue and we use a Redis Hashtable data-structure to capture hidden messages. This data-structure choice makes SendMessage(), RecieveMessage() and DeleteMessage() constant time operations and does some effort in maintaining order of the messages. The process re-visible messages() requires a worker thread in the API to enumerate through all the hashtable keys and check if a message's hidden timestamp is in the past. These messages constitute the Revisible set. We delete the Revisible set from the hidden-message hashtable and enqueue them back into the queue. The complexity of processing re-visible messages is O(h) where h is the number of the hidden messages.

When an API server gets a Recieve/Send/DeleteMessge(Q) for a given queue, the API checks for the key Q.STATE in the redis-serer owning the queue. If the key doesn't exist, the API sets the Q.STATE to Initializing (atomically) and populates the cache by reading Cassandra. While this is happening, if other requests for the queue are encountered, they will all see the Initializing state and go directly to Cassandra to serve the messages. We call this mode Return-From-Cassandra. While in this state, we provide no message hiding ability so users could get duplicates.

Note: We operate Redis in a non-persistent mode and we depend on it being empty when it is re(started).

Cassandra Component

We shard a queue into 100 rows each named as <Q>.<num>. To write a message for a queue, we randomly pick a row between 0-99 and write messages to it. This helps keep the rows small and helps scaling the queue out to multiple Cassandra nodes. The only operations we support in the Cassandra portion is the add/remove message(). The Cassandra schema has row-id be the <Q>.<num> value. The column-name is a composite key with two longs, the first being a combination of time-in-millis and an in-memory atomic increment and the second being a API-node specific number. The message-id is composed of the row-id & column-name for a message.

Availability

When a redis-server fails, the queues it hosted are unavailable from the cache and the API's would resort to the Return-From-Cassandra state. Again, duplicates have a high chance in this state.

When a data-center fails, the traffic for a Queue hosted by dc1 now goes to dc2 and the API servers would treat it as if getting hit for the first time for that Queue (Initializing step). We would use the EXPIRE capabily on the entire Q and update the expiration time on every operation on a queue. This helps us auto-expire the entire queue if the failed data-center comes back online and all traffic for Q goes back to dc1 hence auto-expiring unused Q in dc2.

Monitoring

We provide JMX monitoring for many aspects of CQS including:

- Cache Hit ratio

- Number of messages in specific queues
- Oldest message in a queue
- Number of hidden messages per queue

Linear Scalability

The API Servers are completely stateless and more can be added as needed ensuring scalability of API layer. Redis servers also dont communicate with each other and more can be added as needed. However, the redis server to pick for a queue is done at the API layer by a client library. There is no consistent hashing of redis-servers so adding new redis servers would require re-hashing queues. The Cassandra layer was chosen for its linear scalability and sharding the queue across 100 rows ensures utilization of multiple nodes per queue.

Performance

- Downingtown test environment with 2 API servers behind a load balancer and an 8 node Cassandra ring (all VMs)
- 10 queues with 2 senders and 2 receivers per queue
- Test scenario: Short burst of sending messages followed by a longer period of reading messages from the queues; simulate message processing time of 45 ms for each received message
- Duplicates: 1 / 1 mio
- Loss: 1 / 2 mio

Send Goal (msg/queue/sec)]	Msg Sent / Sec	Empty Pct	Total Ops / Node / Sec	Avg (ms)	Median (ms)	P95 (ms)	P90 (ms)	Max (ms)
10	97	95	1156	3	2	8	4	5252
50	420	77	1324	10	2	22	14	5379
75	528	65	1299	18	3	35	19	5406
100	559	55	1170	26	4	50	25	5163
250	706	29	1195	38	12	160	67	5060
500	775	16	1233	40	13	150	71	5020
1000	718	11	1127	47	14	150	81	5458

Admininstration

For administrative purposes we offer a admin endpoint for all CQS endpoints: http://<endpoint>/ADMIN This interface lets administrators create, delete, list and browse queues. Furthermore administrators can SendMessage() on a queue.

Limitations

Our design hashes a queue to a single Redis server which may be a single-point of failure (for hiding messages) and prove as a scalability bottleneck for a single-queue. In our testing, though a single Redis server is capable of around 40k operations/second.