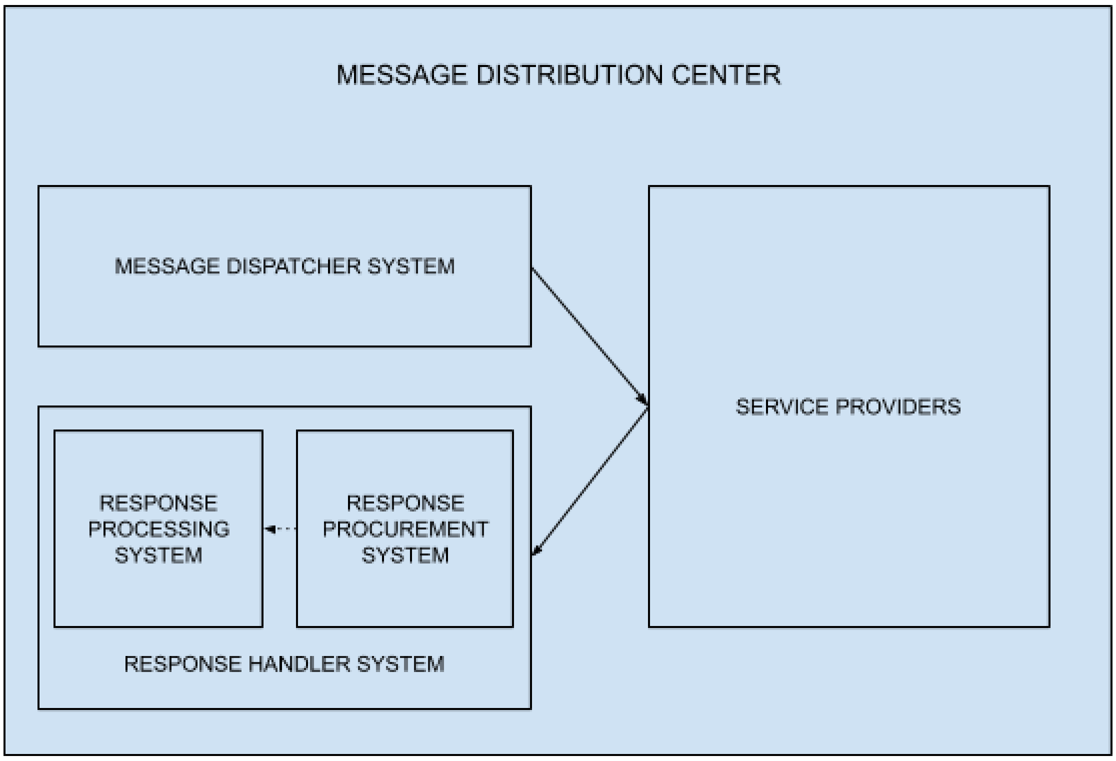
MDC

admin July 23, 2019 Leave a comment Edit

Message Distribution Center (MDC) is an architectural design that enables you to provide highly available, scalable, durable and secure real-time or near real-time communication services like SMS, email and mobile notifications. MDC is extremely modular and supports hassle-free plug and play of its underlying components.


MESSAGE DISTRIBUTION CENTER
MESSAGE DISPATCHER SYSTEM
SERVICE PROVIDERS
RESPONSE PROCESSING SYSTEM
RESPONSE PROCUREMENT SYSTEM
RESPONSE HANDLER SYSTEM

MDC operates in two stages, processing and dispatching of messages to service providers (handled by Message Dispatcher System) and processing of responses delivered by service providers (handled by Response Handler System).

MESSAGE DISPATCHER SYSTEM

1. **CLIENT**

In a burgeoning microservice ecosystem, an MDC client can be created as a plugin or package and augmented to all the microservices interested in sending messages.

Requirements :

1. Perform rudimentary validations on messages such as phone number pattern verification for sending an SMS, validate sender and receivers' email ids for sending emails and notify users with relevant warnings.
2. Append necessary details based on the type of communication.
3. Feed the processed output message to a queueing system.

What we did :

A java client is augmented as a jar to all our microservices interested in sending messages to users.

—

2. QUEUING SYSTEM

Requirements :

A secure, highly performant, reliable, scalable and maintainable publish-subscribe system with supplementary functionalities like priority queuing and data retention.

Available options :

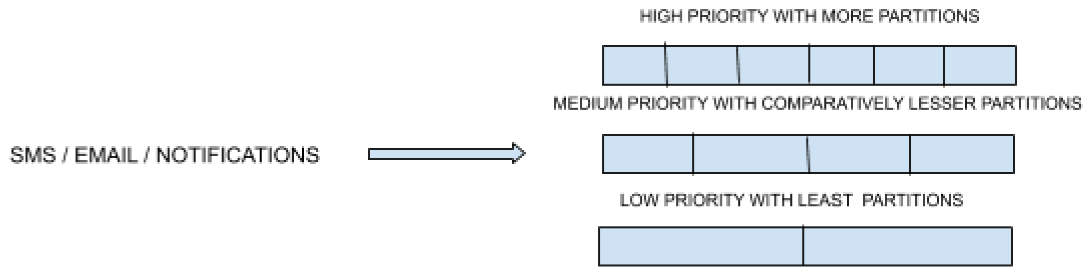
1. **Apache Kafka** – Kafka is a highly scalable stream processing platform which can read and write streams of data like a messaging system. It aids in writing scalable stream processing applications that react to events in real-time also stores streams of data safely in a distributed, replicated and fault tolerant cluster.

1. **AWS Kinesis** – Kinesis is a fully-managed cloud stream processing service administered by AWS. It can be used to build real-time and batch applications on data streams. It supports durable, indexed and encrypted storage of data.

What we did :

We chose Kafka over Kinesis as it is open source and well contributed. We configured a fault-tolerant multi-node multi-broker cluster on AWS EC2 machines with replication of data using Kafka mirroring technique.

Kafka inherently does not support priority queuing, in order to circumvent this, we created queues of different priorities by handling priority as a function of the number of partitions and consumers subscribed to a queue. So, a queue with higher priority has more partitions and a greater number of consumers subscribed than a low priority one, thus aiding faster processing of messages.


HIGH PRIORITY WITH MORE PARTITIONS
MEDIUM PRIORITY WITH COMPARATIVELY LESSER PARTITIONS
SMS / EMAIL / NOTIFICATIONS
LOW PRIORITY WITH LEAST PARTITIONS

1. **REAL-TIME PROCESSOR**

Parallel computing is a computation style of carrying out multiple operations simultaneously using one (by means of multi-threading) or many machines. Parallel computing works on the principle that large problems can often be divided into small problems and these small problems can be executed concurrently.

Following are the two forms of parallel computing :

1. **Task Parallelism** : This form of parallelism covers the execution of computer programs across multiple processors on same or multiple machines. It focuses on executing different operations in parallel to fully utilize the available computing resources in form of processors and memory. [*Running different tasks on different threads utilising full potential of available resources*]
2. **Data Parallelism** : This form of parallelism focuses on distribution of data sets across the multiple computation programs. In this form, same operations are performed on different parallel computing processors on the distributed data

subset. [*Running same task on numerous threads utilizing full potential with input data feed distributed across all the threads*]

Requirements :

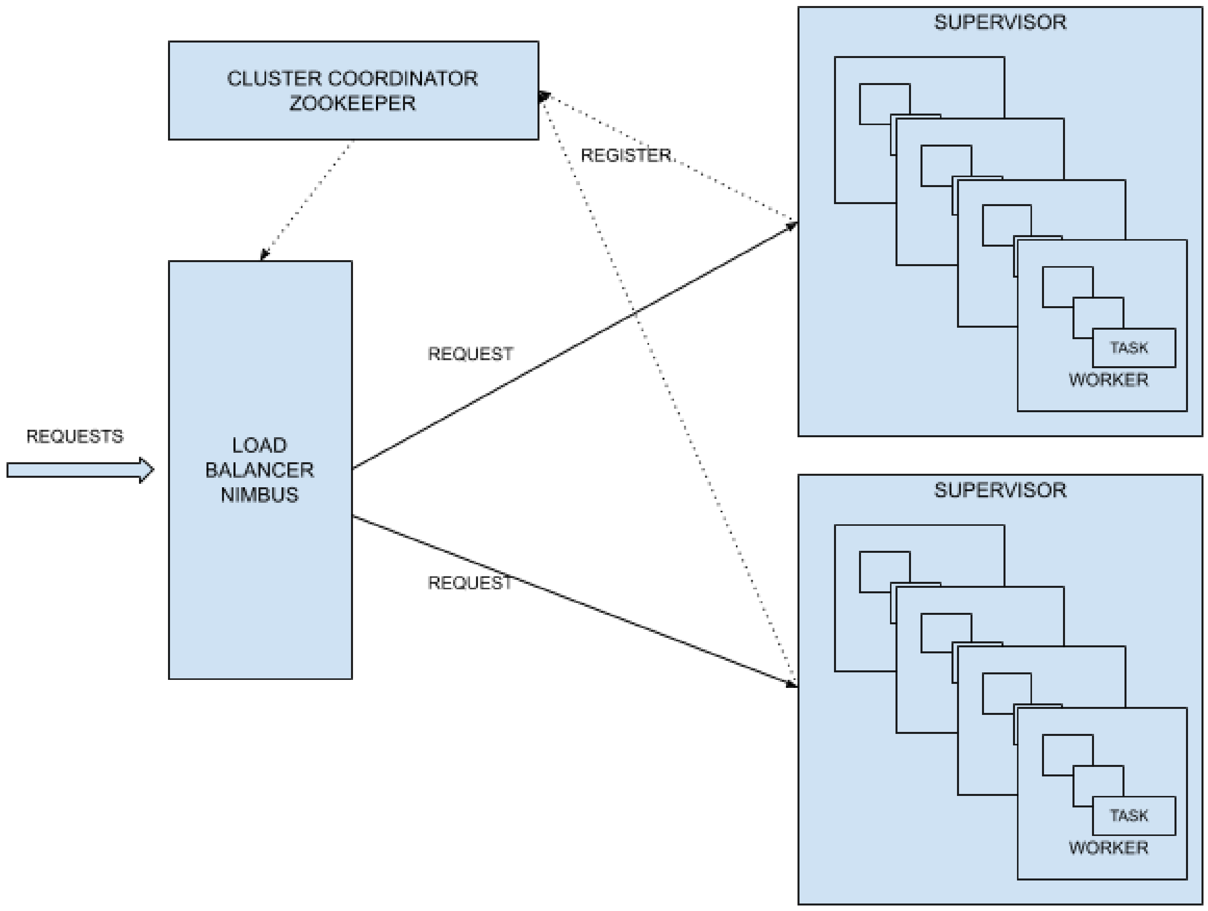
A real-time or near real-time computing system competent to handle unbounded streams of data and support smooth integration with clients written in different programming languages.

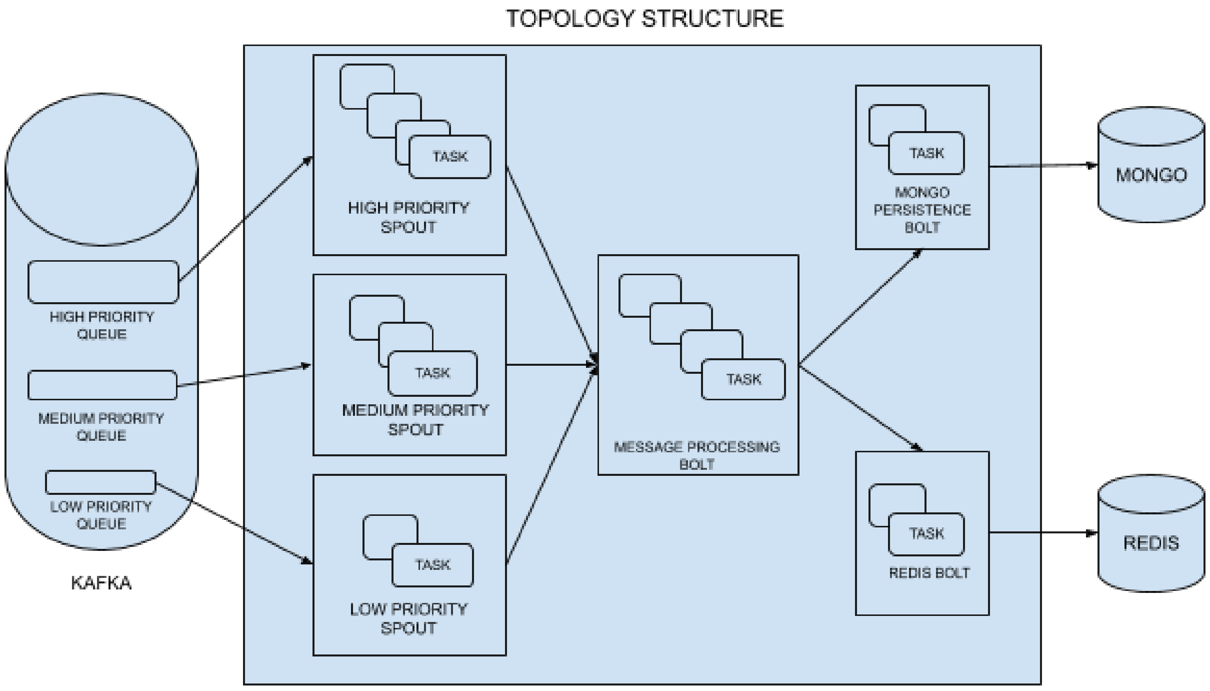
Available options :

1. **Apache Storm** : Apache Storm is a free and open source distributed realtime computation system which does real time processing what Hadoop did for batch processing. Storm integrates with the queueing and database technologies you already use. A Storm topology consumes streams of data and processes those streams in arbitrarily complex ways, repartitioning the streams between each stage of the computation however needed. Storm is based on task parallelism.
2. **Apache Spark** : Apache Spark is an open-source distributed general-purpose cluster-computing framework. Spark is based on data parallelism.

What we did :

We chose Apache Storm as task parallelism is the cornerstone for our application. It provides different modes of processing like exactly once, at least once and at most once whereas spark provides only exactly once processing mode. It is flexible and can be seamlessly integrated with different sources of data streams like Kafka, SQL, MongoDB etc.Low latency, a coveted feature for a real-time processor, Storm aces this criterion compared to Spark.





1. **PERSISTENCE**

Requirements :

In MDC, the message object traverses different components and each component traversed shall append respective metadata to it. Persistence of this message object during its traversal has the following advantages:

1. Fetch the communication status(like delivered, failed) for the message accurately
2. Determine the exact point of failure in a highly scaled distributed system
3. Generate downstream reports.

Since the message can correspond to one amongst different types of communication, its structure can be extremely volatile. A profoundly performant schema-less data store can be the most viable option for persistence. Also, a database which supports REST-like queries can expedite the generation of reports.

Available options:

1. **MongoDB :** MongoDB is schema-less and uses JSON-like documents for persistence. Data can be accessed using MongoDB query language.

1. **Cassandra :** Cassandra uses wide column stores which utilize rows and columns but allows the name and format of those columns to change. It uses a blend of a tabular and key-value. Unlike a typical relational database management system (RDBMS), tables can be created, altered, and dropped while the database is running and processing queries.

Column families are similar to tables in RDBMS and contain rows and columns, with each row having a unique key. Unlike a traditional RDBMS, all rows in a table are not forced to have the same columns. These columns can also be added on the fly and are accessed using the Cassandra Query Language (CQL). While CQL is similar to SQL in syntax, Cassandra is non-relational, so it has different ways of storing and retrieving data.

What we did:

We chose MongoDB as it is schema-less and supports multiple or nested indexes. We maintain different collections for different types of communication. Rest Heart is a rest service written over MongoDB. We use it for processing simple queries to which attributes for filtering and sorting can be passed as query parameters.

1. **CACHE** :

Improve performance of the existing system by avoiding conventional database markedly known for delayed seek time and retrieve data from high throughput and low latency in-memory data stores.

Requirements :

Highly available in-memory data store which features versatile data structures, geospatial, transactions, on-disk persistence, and cluster support.

Available options :

1. **Redis :** Redis is an open source, in-memory data structure store, used as a database, cache and message broker. It supports data structures such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps, hyperloglogs, geospatial indexes with radius queries and streams. Redis has built-in replication, Lua scripting, LRU eviction, transactions and different levels of on-disk persistence, and provides high availability via Redis Sentinel and automatic partitioning with Redis Cluster.
2. **Hazelcast :** Hazelcast is an in-memory computing platform that manages data using in-memory storage and supports distributed processing for breakthrough application speed and scale.
3. **AWS Elasticache :** AWS elasticache offers fully managed both open source in memory data sources redis and memcached.

What we did :

We chose Redis, an in memory data store for caching. We also use it to store expiry times pertaining to a retryable message i,e, when a message dispatched fails to deliver to an end user, we mark that message as retryable and store it with an expiry attached in the cache.

SERVICE PROVIDERS

EMAIL SERVICE PROVIDER

Requirements :

Regular communication with customers through emails for sending updates, messages, receipts and advertising content is ubiquitous. Managing custom SMTP servers is fraught with disadvantages. Instead, reliable, cost-effective service catering to the above requirements is preferable.

Available options :

Following are few reliable email service providers like AWS SES (Simple Email Service), Sendgrid, Mailjet etc.

What we did :

We chose AWS SES as our email service provider as it reliable and cost-effective service backed by Amazon. It provides numerous metrics to track delivery status, unread status etc.

Generates accurate alerts following an increase in bounce/ complaint rates.

SMS SERVICE PROVIDER

Requirements :

An ideal SMS service provider should be competent to fulfil the following requirements :

1. Provide an ability to choose different modes of service(credit-based or SMS based) across different countries.
2. Provide an ability to configure multiple sender IDs (metadata of sender i,e, the name of the sender and corresponding details).

3. Provide support for all types of interfaces and protocols like HTTP, HTTPS, SMTP, FTP, SMPP etc.
4. Provide a reliable mechanism for clients to fetch responses for dispatched messages.
5. Provide excellent documentation and support.
6. Service provider equipped with an excellent network bandwidth aiding low latency, reliable message delivery.

Available Options :

Following are few illustrious gateway providers :

1. Kaleyra (Solutionsinfini)
2. SMSCountry
3. Unicel
4. eminent systems etc.

What we did :

We chose Kaleyra as our primary service provider as they provide accurate reports, push response mechanism and ensure high success rate at low cost. On the contrary, we also made sure that our components are flexible to support the integration of any service provider conforming to the requirements.

NOTIFICATION SERVICE PROVIDER

Requirements :

As smartphone has become an integral part of our lives, push notifications ease businesses to provide information to all their users and keep them engaged with the product.

Available options :

Cost of service and integration time are two factors which play a crucial role in choosing the service provider. Following are few reliable, cross-platform service providers AWS SNS, FCM (Firebase Cloud Messaging), Pushwoosh, Urban airship, Pushengage etc.

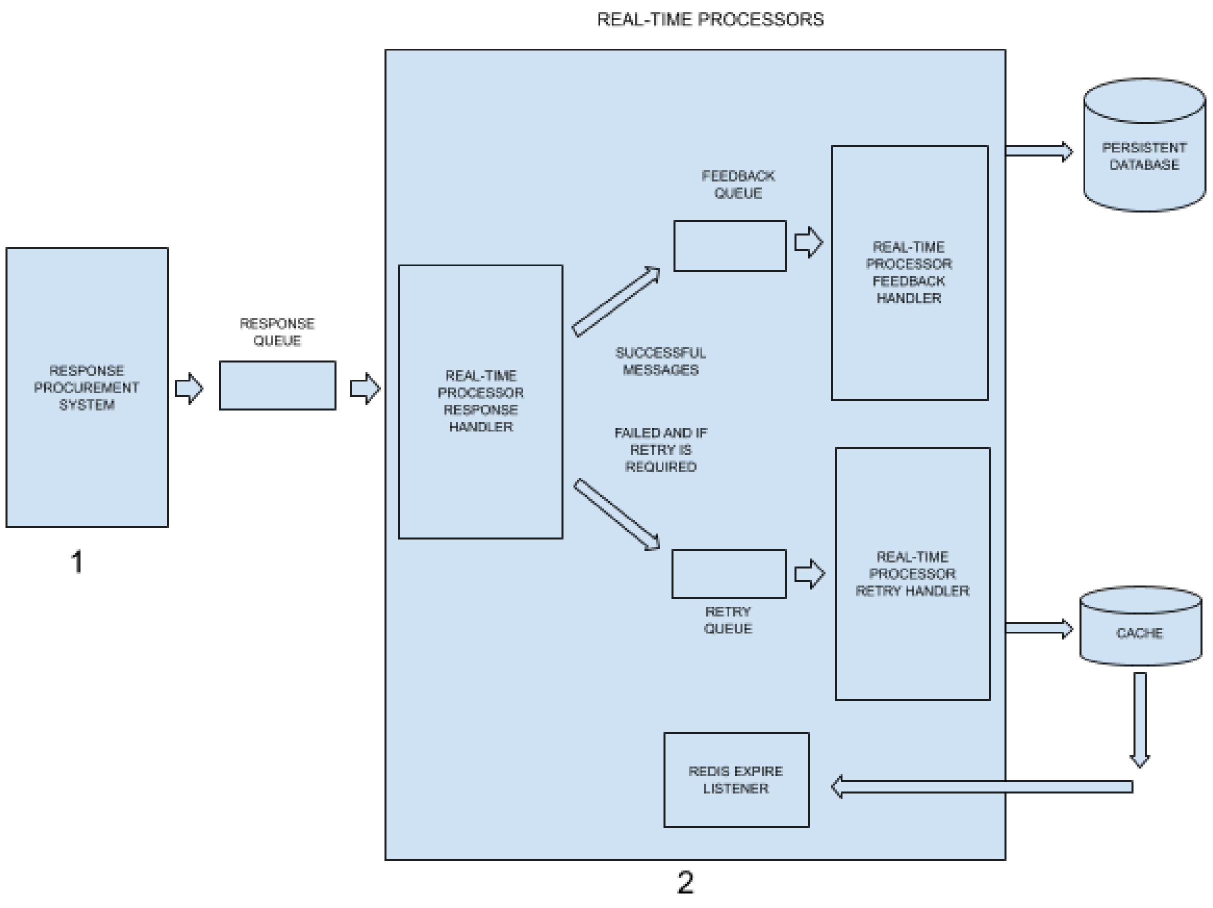
What we did :

We chose AWS SNS as service provider as we are already using so many AWS services and cost of service is very less. Other main reason is that it provides easy way of fetching delivery status

---

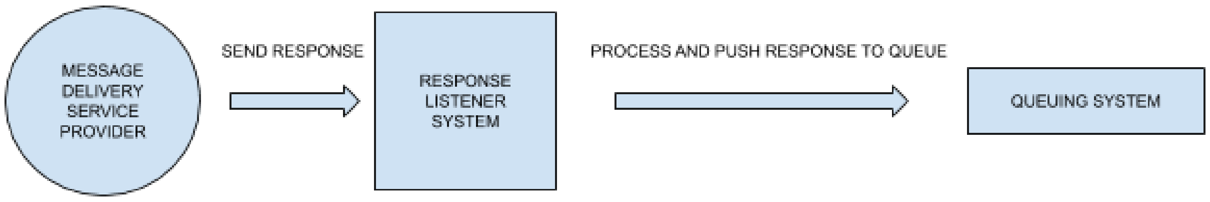
RESPONSE HANDLING SYSTEM

Response Handling System manages responses procured from service providers. It operates in two stages, procurement of responses from service providers (handled by Response Procurement System) and processing of obtained responses (handled by Response Processing System)


REAL-TIME PROCESSORS
RESPONSE PROCUREMENT SYSTEM
1
RESPONSE QUEUE
REAL-TIME PROCESSOR RESPONSE HANDLER
FEEDBACK QUEUE
SUCCESSFUL MESSAGES
FAILED AND IF RETRY IS REQUIRED
RETRY QUEUE
REAL-TIME PROCESSOR FEEDBACK HANDLER
REAL-TIME PROCESSOR RETRY HANDLER
PERSISTENT DATABASE
CACHE
REDIS EXPIRE LISTENER
2

1. **RESPONSE PROCUREMENT SYSTEM :**

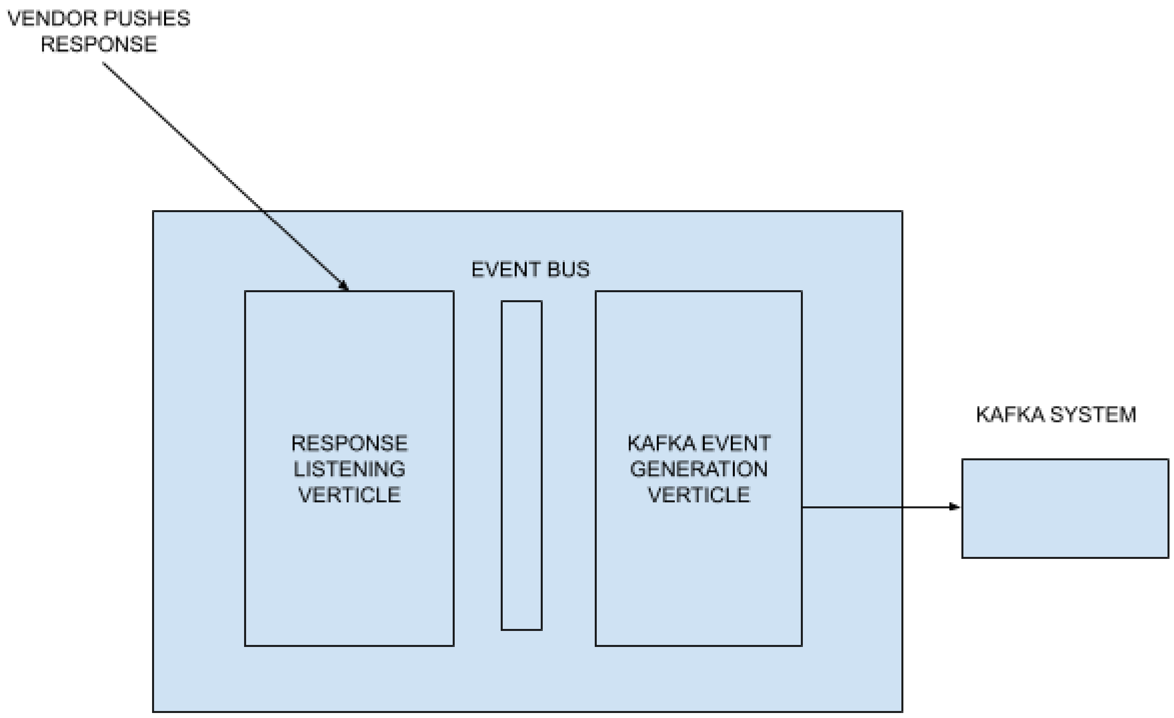
Response procurement system fetches responses from configured service providers. It consists of two major components, Response listener system which listens for responses delivered by service providers and the queuing system.


MESSAGE DELIVERY SERVICE PROVIDER
SEND RESPONSE
RESPONSE LISTENER SYSTEM
PROCESS AND PUSH RESPONSE TO QUEUE
QUEUING SYSTEM

SMS RESPONSE PROCUREMENT

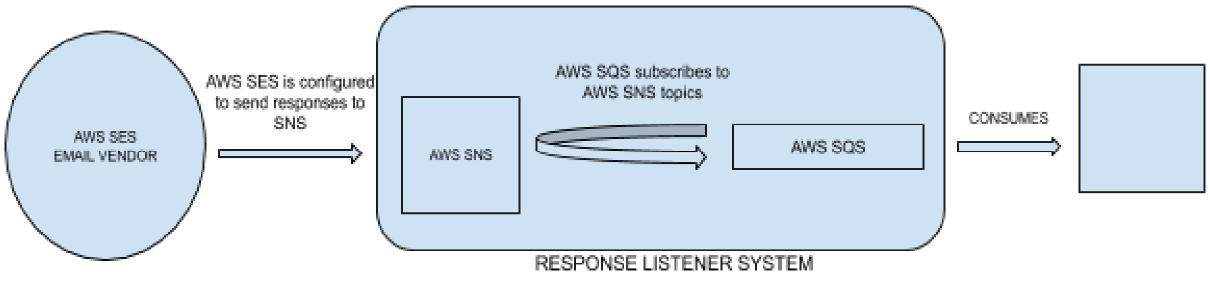
Following are mechanisms provided by SMS vendors to fetch responses for dispatched messages :

1. **Push Mechanism** : In this mechanism, the service provider typically expects the provision of a callback URL to which it delivers the generated response for the message dispatched.
2. **Pull mechanism** : In this mechanism, the service provider expects to receive an individual or a bundled request based on response ids generated for the dispatched messages. Different filters should be supported for fetching various reports.



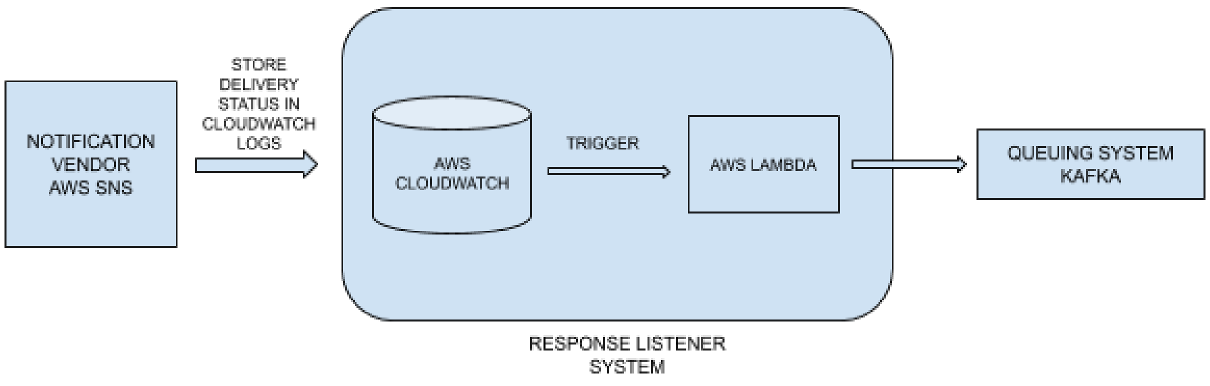
We have adopted push mechanism and used Vert.x as our HTTP server. Vert.x is an integral part of the response listener system which listens for responses delivered by service providers, it supports both synchronous(blocking) and event-driven(non-blocking) mechanisms. We chose the non-blocking mechanism so that the vendor doesn't face a delay in receiving HTTP response status. Responses thus delivered are saved in an event bus on which a verticle is written to push them into the queuing system.

EMAIL RESPONSE PROCUREMENT


AWS SES EMAIL VENDOR
AWS SES is configured to send responses to SNS
AWS SQS subscribes to AWS SNS topics
AWS SNS
AWS SQS
CONSUMES
RESPONSE LISTENER SYSTEM

We chose AWS SES as our email service provider and configured it to push all types of responses like delivery, bounce, complaint, open, click etc to respective AWS SNS topics. We chose highly reliable AWS SQS as the queuing system and configured SQS queues as subscribers to SNS topics. Self-managed storm cluster subscribes to these queues and process responses in real-time.

NOTIFICATION RESPONSE PROCUREMENT



Created our response listener system using AWS CloudWatch and AWS Lambda functions and reused our self-maintained Kafka cluster as the queuing system.

As part of the response listener system, AWS SNS provides a facility to send delivery status to CloudWatch logs. So, in order to process and push these generated responses to the Kafka cluster, we configured a cloud watch trigger which shall be fired upon an insertion event and executes a lambda function responsible for pushing these events.

2. REAL-TIME PROCESSORS

RESPONSE HANDLER

Response handler is a storm processor, which segregates message responses into two types (success and failure) based on delivery status of the message. It fetches the message from the cache rather than DB.

FEEDBACK HANDLER

Feedback handler sends feedback to clients and persists the final status.

RETRY HANDLER

Retry handler assigns an expiry time to the message based on the status of response received from the service provider.

REDIS EXPIRE LISTENER

Redis expire listener listens for all expired events and pushes them into the queuing system iff retryable else updates the final status in MongoDB.

There are other CRON jobs that run everyday to make sure all responses are updated by using **PULL MECHANISM** for fetching responses from service providers.

Leave a comment

Comment

Post Comment