Architecture Discovery Report

Voxer

Oct.2021

Terry.Cho Customer Engineer

terrycho@google.com

Discrimber

This document provides architecture suggestions with google cloud products.
This document can be shared with Voxer, Google Cloud team and Sada (google cloud partner for voxer). It is customer confidential information and cannot be shared by others without approval from the voxer customer.
This architecture suggestion is provided to the customer with good intention to help the customer. The Google team doesn't have responsibility for the guidance.

Discrimber 2

Overview 5

Current Architecture 5

Use case 5

Architecture 5

Overall Message flow 5

NR (Node router) architecture 6

More detailed message flow 7

Message & header data store 8

Media data 10

Redis 10

Product roadmap & pain point 10

Product roadmap 10

Major Pain point 11

Scalability 11

Redis compaction 11

Design principal 11

Not changing basic architecture 11

Focus on scalability 11

Add suggested architecture with priority 12

Suggested Architecture 12

Centralized cluster configuration management 12

Redis 13

Dynamic client assign (Optional) 13

Redis compaction 13

Database change 14

Spanner for metadata 14

Big Table for time series data 16

Rebalancing overload 16

Type of Storage 17

Scaling 17

CBT BQ federation (BETA) 17

Image 17

Thumbnails 18

CDN Optimization 18

Archiving old image data 18

Additional Suggestion 19

Authentication with SNS 19
Personal information scanning 20
Prevent explicit contents 20
Monitoring 20
Logging architecture 21
    Log Gathering 22
    Log Exclude & export 24
        Export 24
    Export data from cloud operation suite to BigQuery 25
Distributed Transaction Tracing 26
Measuring customer feedback 26

Overview

This is an architecture discovery report based on interviews with Voxer.
Objective of the architecture discovery is to understand current architecture and recommend TO-BE architecture with google cloud products.

Current Architecture

Use case

Voxer is a high scale chat messaging mobile application , which specially has walkie talkie voice messages. Users can record the voice message and send it to destination users. It also can send other media data, such as images etc.
It has 10M+ downloads, 232 reviews with 4.0 review score

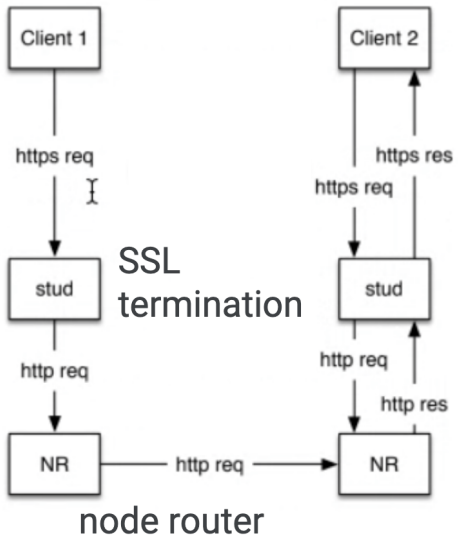
It targeted the B2C market but also supported enterprise customers(B2B) by deploying the backend system into the enterprise in an on-prem way.
They are planning a new business model with Saas style,which will provide backend with SDK. With the SDK their enterprise customer can develop their own walkie talkie mobile application clients

Architecture

To suggest recommended architecture, it needs to understand current system architecture.

Overall Message flow

The overall message flows between clients like below.



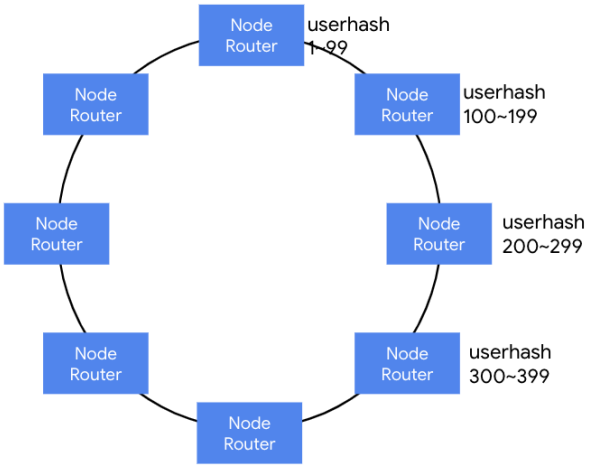
<figure. Overall message flow>

1) Client will be connected to its corresponding server, which is called as NR (Node router) thru stud. Stud is a simple SSL termination.
2) Message will be passed from the source clients to its NR (let call this as “source NR”). The message combines the message itself and the destination client.
3) The source NR can find destination NR with the destination client id. (by using consistent hashing algorithm)
4) After finding the destination NR, the message will be streamed to the destination NR, and the destination NR will stream the message to the destination client device.

NR (Node router) architecture

The node router is connected to the mobile client to send and receive messages. If mobile clients are connected to different NRs, they exchange messages through communication between NRs as described above.

To support a high scale number of users, it is using distributed architecture.

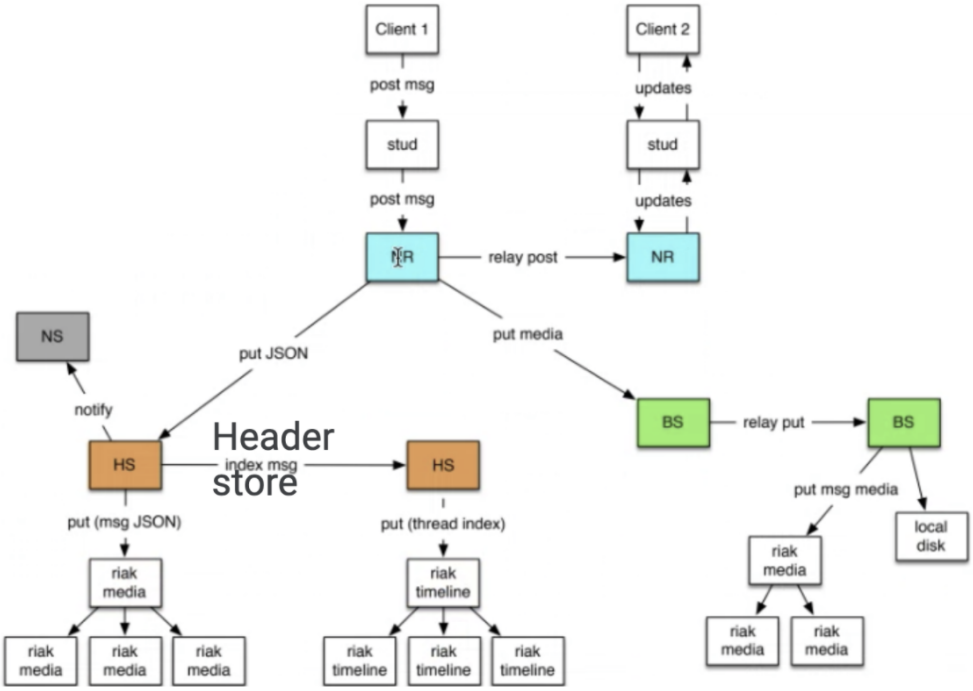


<fig. NR system topology>

- Ring topology & client to NR mapping
  It uses ring topology. The mapping between client and NR uses consistent hashing technique with “user id hash value (MD5)”. For example NR 1 connects to clients which has hash #0~99, NR 2 connects to the clients which has hash #100~199
  Initially, client is mapped 3 NR servers. In normal situation the client is communicated with the 1’st NR server. If it fails, the client will connect to 2nd NR, 3rd NR. If there is no NR,it connects to discovery server to get new NR list.
- Framework (node.js)
  The NR is implemented by using node.js
- Protocol
  Client to NR communication is using HTTP based SPDY protocol (introduced by Google but deprecated now)
- Cluster configuration management
  NR needs to know the list of NRs to send messages to the destination client thru the connected NR. The list of NR information is configured in a JSON format.
  Because of this, if it adds new NR or removes NR (or removed by incident), to update the server list, it needs to update the JSON file and restart the all NR server (in rolling update way). It brought scalability issues.

More detailed message flow

To send the message to the destination clients, it needs more component interactions to persist the data.



<fig. Detail of message flow>

NR=HS+BS

- NS (Notification Server/node.js) : push message server. It sends push messages thru android or IOS push. (example : log in other devices etc).
- HS (Header server/node.js) : It stores thread metadata, message(text), user profile etc into riak nosql database. (content type, sender, receiver, timestamp, etc)
- BS (Body Store/node.js) : It stores media & its meta data, such as image etc into local disk and riak. (blob data - binary data with some keys)

Message & header data store

After sending the message, the message, thread data is stored in the riak thru the HS server. It is called with async way from NS with node.js
The purpose of storing the data into the database is the backup purpose. For example if the user logged out and logged in again, it needs to see the previous messages. In time, it pulls the data from the storage.
It means that it has heavy writes but latency is not sensitive. Read IO is relatively lower than write but it needs faster latency.
write/read performance in the database are equally important
180 ms is the maximum latency observed, but average latency 80ms for good performance.

This is sample of the message data in riak


```
messages bucket
{
  "to":[
    "f._____.voxer.1578692462220.59234133"
  ],
  "message_id":"f._____.voxer.1578692462220.59234133:profile",
  "create_time":1578692462.327,
  "normalized_create_time":1578692462.327,
  "posted_time":1578692462.327,
  "from":"f._____.voxer.1578692462220.59234133",
  "content_type":"profile",
  "profile":{
    "first":"F",
    "last":"1578692461711Test",
    "phone":"55555555555",
    "dialing_code":"+1",
    "email":"funtest+1578692461711gcp1-stage.voxer.com0.3991@voxer.com",
    "username":"f09994973",
    "type_of_signup":"normal",
    "last_auth_source":"voxer",
    "device_info":{
      "tz":[
        "PST"
      ],
      "system_version":[
        "10"
      ],
      "client_version":[
        "3.0.1"
      ],
      "system_name":[
        "iPhone OS"
      ],
      "uuid":[
        "c2d8c3e2feef5096c3ea4d833307bcfc"
      ],
      "mac":[
        "000000001111"
      ]
    },
    "is_validated":false,
    "display_name":"F 1578692461711Test"
  },
  "user_identities":{
    "emails":[
      "funtest+1578692461711gcp1-stage.voxer.com0.3991@voxer.com"
    ],
    "phones":[
      "55555555555"
    ],
    "facebooks":[

    ],
    "usernames":[
      "f09994973"
    ]
  },
  "last_modified_time":1578692462.327,
  "account_flags":[
```

```
  ]
}
```

Media data

The media data is stored into local disk and riak (meta data) after sending into the destination NR.
The image data is stored in the original format without compression in local disk
The image is delivered to the client by using CDN. CDN is effective for reducing delivery latency especially for multiple clients. (some chat rooms have 50~100 user clients)

Redis

Redis is used to store the unread message counts. It has 8~10 instances with 16 CPU with 28GB memory per instance.In the compaction time the instance is freezing due to a lack of the memory. To solve this problem, the operation team is increasing the memory when the compaction happens once a month and reducing the memory again after the compaction is done.
Redis is running on a VM but shared with other services
Not subscribed via the GCP marketplace

Product roadmap & pain point

In order to suggest enhancements to the architecture, we need to understand the product roadmap and current pain points. These are the roadmap and the pain points that were identified during the discovery workshop.

Product roadmap

- **SDK**
  It will release an SDK based product for enterprise. Their end customer can create their own Walkie Talkie app by using this SDK. **The backend will be created separately from the current legacy backend.**
  The SDK server will be created with new cluster and improved architecture (database will be replaced from riak to other gcp native database)
- **Data gathering**
  To future plan, the team is planning to store the data for further analysis. Voice messages will be stored in text by converting it with speech-to-text APIs.

Major Pain point

Scalability

The NR list is configured by JSON files. To add the additional nodes, it should change the configuration file in each NR server and restart it with a rolling update. It makes hard to add new NR instances

Redis compaction

Redis periodically runs compaction to store the data into memory. To do that, redis forks child processes. New process forks means replicating the same process, which needs double the size of the memory. If the memory size of the machine is not enough, it will use disk space for the swapping. For this reason, it will use virtual memory in the disk and bring performance impact.

Design principal

Based on the pain points, roadmap and it defines design direction like below

Not changing basic architecture

Good thing about the Voxer app is immediate response time. It means that if the user sends a message to the destination user, the message is delivered very quickly. This is by sending the message directly to the corresponding client with streaming without any intermediate message delivery like message queue. Also the client open connection with the NR server all the time and the communication can happen quickly thru the opened connection.

The immediate message delivery is the key feature why the end users like voxer app followed by google play store app review.

For this reason, it will not change the main NR + SPDY connection based message flow architecture

Focus on scalability

Followed by the interview, current main pain point is scalability issue and redis compaction issue. In the SDK serverd design, it will also change the database from the riak to gcp native database. It is a big change.
With this background, this architecture suggestion will focus on removing the current pain points as well as potential problems (in database design)

Add suggested architecture with priority

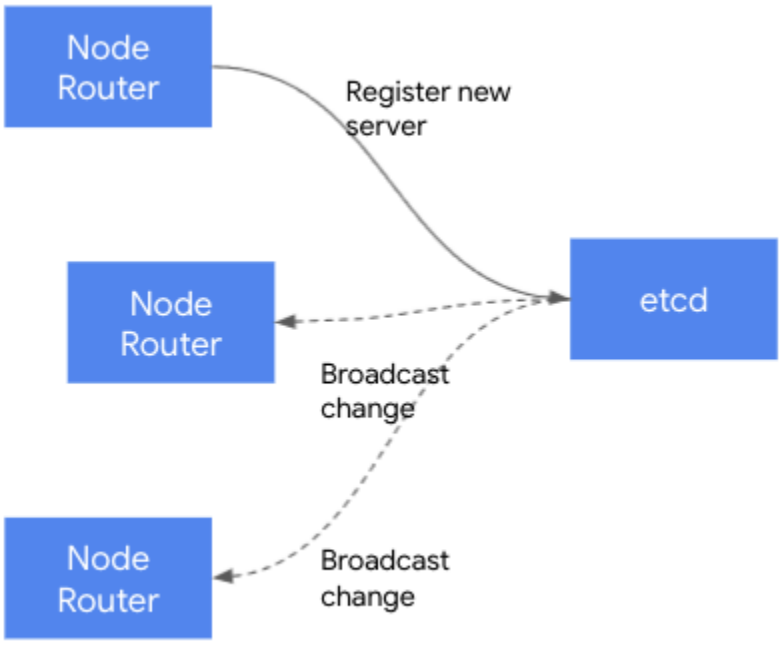
In addition to pain points, there are many areas that can be improved. But the engineering team's resources are limited, it will suggest additional improvement points based on lower effort and focusing on what needs to be changed at the beginning.

Suggested Architecture

Centralized cluster configuration management

The reason for the scalability issue is due to file based configuration of the NR server lists. To solve the problem we can store the list in centralized space and synchronize it across multiple servers. It can remove rolling update to apply the change.

As a centralized configuration server, there are multiple choices
Zookeeper is used for Hadoop to manage configuration data. Etcd is developed to cover the cons of the Zookeeper. It is used in the Kubernetes cluster for configuration management.



If a new server is started, the server can add their own profile into etcd and the etc can broadcast the changes to other servers. (You can use the watching feature in etcd to catch up the changes). Also if a server is removed purposely, the server can remove its profile from the etcd before the shutdown.

But if a server incidentally shutdowned before sending the removing signal, the etcd may have wrong information. To prevent this it needs to have a heart-beat, it needs to send the server alive check periodically and if the server doesn't respond to it, the server can be removed.

Redis

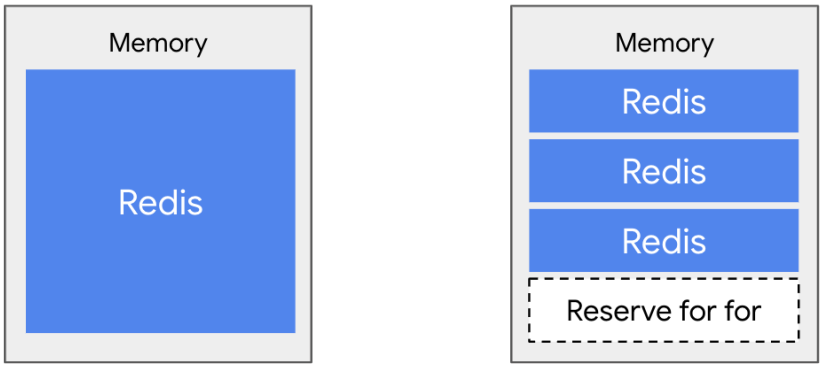
Etcd is a good solution to manage this type of configuration management but it can bring additional technical stacks and learning curves. As an alternative solution, we can use redis. The system is already using redis and it can remove duplication of similar solutions. It can store the server list in redis and the change can be notified by redis pub/sub to the servers.
For the redis data modeling and design, it is recommended to get support from redis 3'rd party saas vendor

Dynamic client assign (Optional)

If a new NR server is added, new clients need to be connected to the new server to distribute the connection among the NR servers evenly. In my understanding clients are balanced to NR servers with Hash, which may not distribute the load evenly across the NR servers.
To solve this problem,we can add a client-to-server look up server.
When the app is started, the client can connect the look up server to get its paired server. In the look up server, it will manage mapping tables between client to server and give appropriate server ip to the client. The mapping table can be managed in redis.
This is optional, because client to server connection mapping and fail-over/fail-back seems not have a problem in current architecture.

Redis compaction

One of the problems is redis compaction. Redis dumps its data into the disk with the compaction behavior and the compaction time; it needs double the size of memory to support the redis process fork. Because the memory is not enough, it uses virtual memory(disk), brings performance issue.



<fig redis memory >

One of the options to solve this problem is that instead of running one redis instance on one machine, there is a way to run several small instances on one machine and reserve memory for a fork. The forked process size of the small redis instance will require smaller memory and we can reserve the memory in the same machine.

Best recommendation is using managed services like redis labs and get guidance from the vendor. You can use the redis labs in google marketplace with integrated billing & support with google cloud

Database change

Database is used to store the metadata and querying chatting (conversation data). To support scale, it is recommended to use nosql but the nosql has different characteristics.
For the database design, it needs to separate the type of the data

- Meta data : user profile etc, which queries simply with K/V
- Time Series data : chatting message (conversation) , it needs scanning with time series. For example, if I want to get a message from yesterday to today. It should scan the data from yesterday. In general nosql stores the data in a distributed way and the structure is not good to scan the data. To support the scanning it needs to support index or other technology. Some types of NoSQL such as HBase, Cassandra support sorted keys and it will store the values in the same place with sorted order. It can help to scan data based on the index.

Spanner for metadata

Spanner is newSQL which is based on NoSQL with SQL. It supports consistency across multiple instances.
Spanner is widely used for metadata stores in google. It is good for parent-child data structure.
For example if we have singer - album data and singer has multiple albums like below

PK

Singers

SingerId	FirstName	LastName	SingerInfo
1	"Marc"	"Richards"	<Bytes>
2	"Catalina"	"Smith"	<Bytes>
3	"Alice"	"Trentor"	<Bytes>
4	"Lea"	"Martin"	<Bytes>
5	"David"	"Lomond"	<Bytes>

```
CREATE TABLE Singers (
  SingerId   INT64 NOT NULL,
  FirstName  STRING(1024),
  LastName   STRING(1024),
  SingerInfo BYTES(MAX),
) PRIMARY KEY (SingerId);
```

PK

Albums

SingerId	AlbumId	AlbumTitle
1	1	"Total Junk"
1	2	"Go, Go, Go"
2	1	"Green"
2	2	"Forever Hold Your Peace"
2	3	"Terrified"

```
CREATE TABLE Albums (
  SingerId     INT64 NOT NULL,
  AlbumId      INT64 NOT NULL,
  AlbumTitle   STRING(MAX),
) PRIMARY KEY (SingerId, AlbumId);
INTERLEAVE IN PARENT Singers ON DELETE
CASCADE;
```

For the singers and albums table we can define interleaving relationships.
The interleaved table will be stored physically like below. So getting singer and album in a query can be very fast.

Singers(1)	"Marc"	"Richards"	<Bytes>	
Albums(1, 1)				"Total Junk"
Albums(1, 2)				"Go, Go, Go"
Singers(2)	"Catalina"	"Smith"	<Bytes>	
Albums(2, 1)				"Green"
Albums(2, 2)				"Forever Hold Your Peace"
Albums(2, 3)				"Terrified"

Please be aware that spanner provides high availability, it needs minimum 3 instances and can be relatively expensive. As an alternative, Firestore can be a good choice because it can support JSON data format.

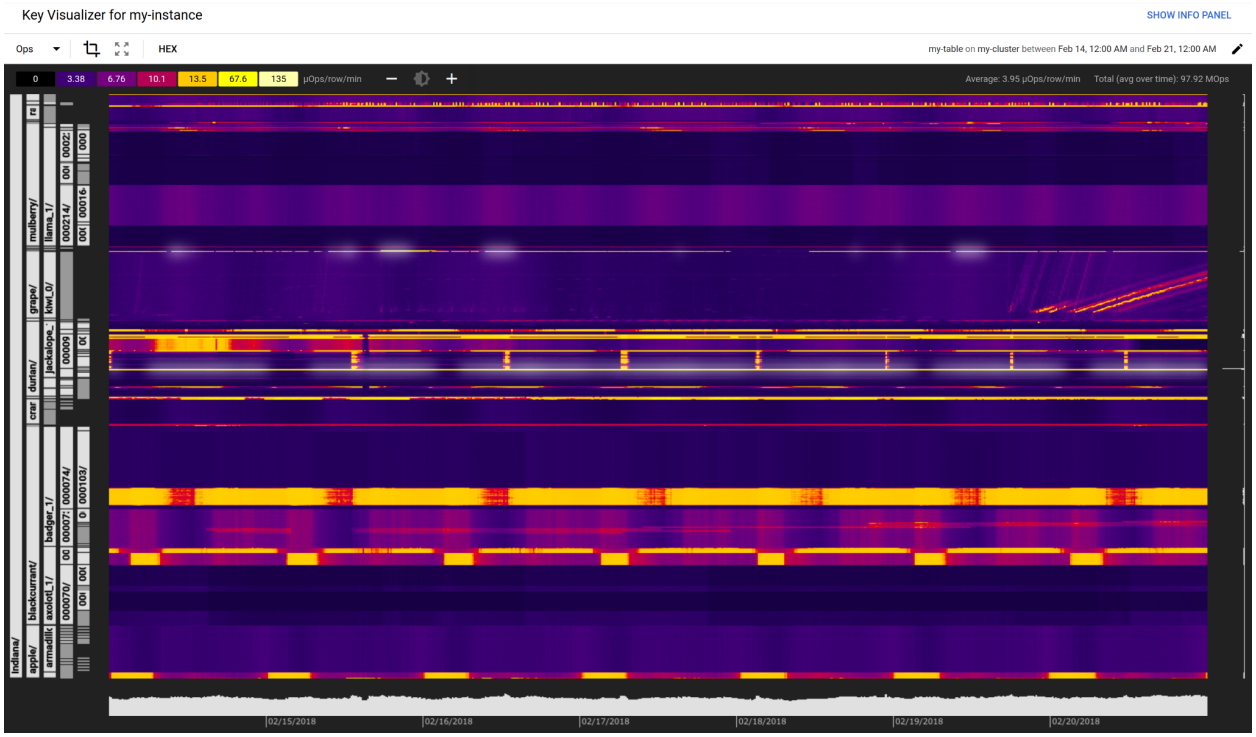
Big Table for time series data

Cloud Bigtable(aka CBT) is the origin of Apache HBase NoSQL. It is key value store and the data is sorted by primary key. So in the chatting message scanning scenario, CBT can be right choice.
When looking for a chat message, it has to go to the designated time in the chat room and scan continuously, so you need to design the key accordingly.
So the key can be like this

- {room-id-hash}-{timestamp}

Since data is distributed across multiple shards according to the key value, the key must be evenly distributed. It is recommended to design as much as possible.

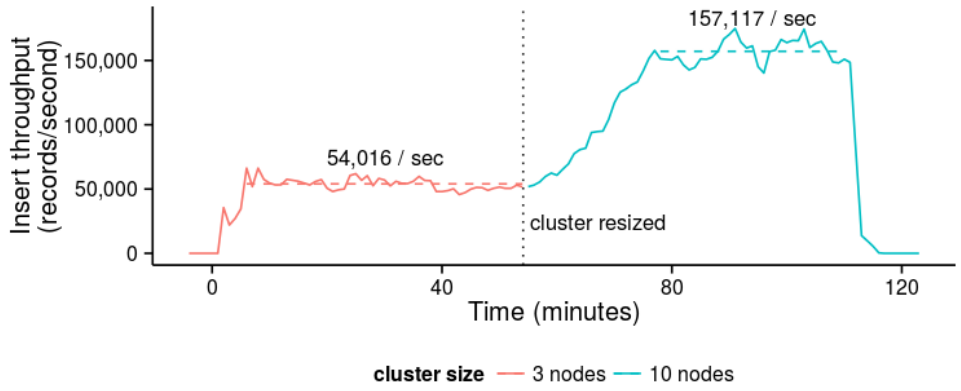
CBT provides a key visualizer and you can find the hotkey and key distribution easily.



Rebalancing overload

Traditional problem of NoSQL is rebalancing. When it adds a new instance or removes an instance, it needs to redistribute the data to the new instance (or from the removed instance). It brings performance issues. But in CBT that is managed service and it separates the computing node and data storage node. With this architecture, if it adds additional nodes, it

simply adds computing nodes only and re-pointing the storage. It removes rebalancing overload.



<fig. Node scaling rebalancing overload test>

Type of Storage

The CBT has two different types of storage (type), which are SSD and HDD. SSD is faster and more expensive. HDD is relatively slower but cheaper. In SSD, single row retrieval takes one-digit ms (5~8 ms). After creating the cluster, it cannot change the storage type

Scaling

CBT doesn't support auto scaling but it supports manual scaling. The scaling can be automated with script. (Scaling guidance). This is a very good feature, you can schedule the scaling based on usage (scale up day time and scaled down night time), it can save your cost.

CBT BQ federation (BETA)

The chatting data can be used for analytics later. To do that the data in the CBT needs to be exported to data warehouse or archiving storage.
CBT supports export and federation feature. BigQuery can query the data in the CBT and store it in the BigQuery(guidance). It doesn't need any additional implementation.
Also, the CBT data can be exported and stored in the GCS (with avro format)
The exported data can be used for further analytics. If you don't query for the message, it is better to store the data in GCS (BigQuery storage cost is higher than GCS)

Image

The Voxer app sends images and animated GIFs. It uses network egress and CDN. It costs a lot. To reduce the cost and improve the performance, there are a number of suggestions:

WebP

Animated GIF format is very heavy. WebP can support the animation and also reduce the size of the file. As well as animated GIF, the webP can reduce the image size itself by about 25~30% compared to JPEG.

Thumbnails

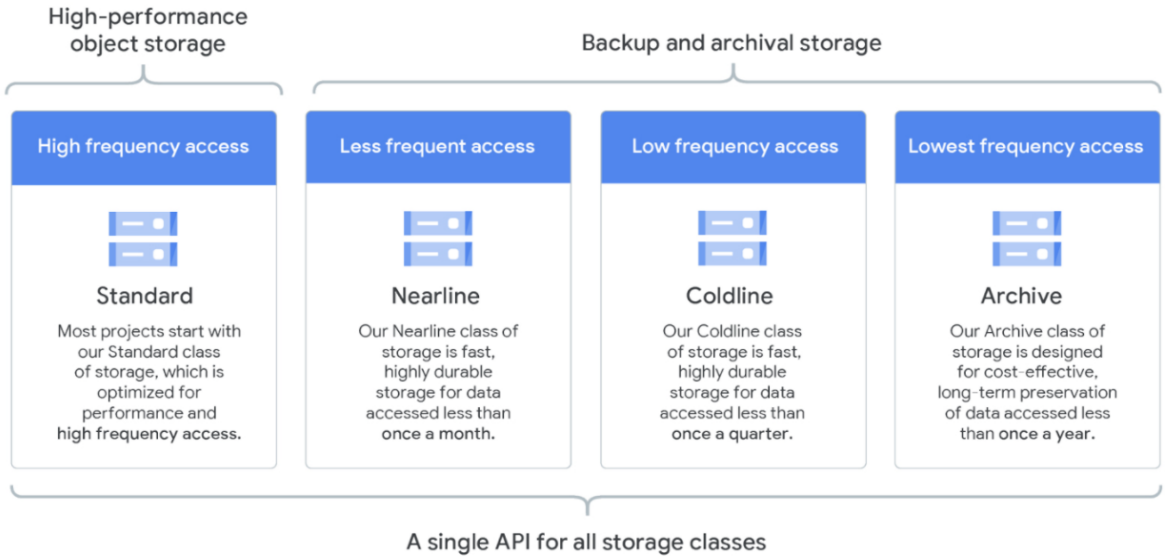
Current architecture sends the full image. Rather than sending the full image, it will be better to send compressed small thumbnails first then, when the user clicks the image we can send the full image. It can reduce the image transfer cost as well.

CDN Optimization

Sending the image is using CDN. CDN is good to reduce latency , especially to deliver the contents to multiple users. But the Voxer application is chatting-based and the number of users in each chatting room may not be huge, which means CDN may not be required.
Followed by the interview, there is a case that has many users (50+) in a single room. Hence CDN needs to be selectively applied depending on the number of users in the room.
If the room has more than 50+ users, the client will go to CDN to get the images. If not, it goes to the origin directly.
Google Cloud Storage is recommended as an image origin.

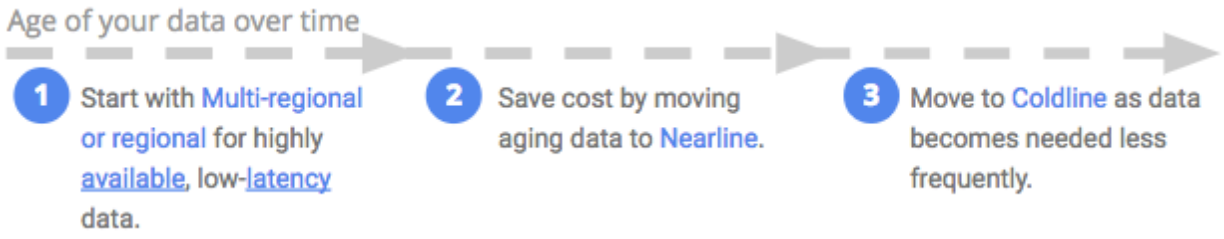
Archiving old image data

The images are accessed frequently after uploading but the old image will not be accessed frequently. GCS has different type of storage with different prices but same performance.
Google cloud provides multiple classes of storage with different prices.
It has standard, nearline, coldline and archiving storages.
All the storage tier has the same API interface and same performance. Only difference is cost.
Higher storage (standard) has more expensive storage cost and lower access cost. Lower storage (archive) has lower storage cost and higher access cost.



<fig. Google cloud storage tier>

Also the storage class can be automatically changed by using storage lifecycle management feature in the Google Cloud storage



<fig. Concept of storage life cycle management>

For example, old files can be moved to cheaper storage class based on the rule and save the cost.

Also depending on the data residency, you can select the location type. Multiple storage will replicate the files to multiple regions in the data center to provide lower latency. Region storage will store the data in a single region with lower price. That is one another way to save storage cost

Location Type	Availability SLA[1]	Typical monthly availability
multi-region	99.95%	>99.99%
dual-region	99.95%	>99.99%
region	99.9%	99.99%

<fig. SLA by location type>

Additional Suggestion

Authentication with SNS

Voxer app is using email based authentication but sns id (facebook,twitter,google etc) is more convenient. It can prevent users from leaving in the sign up phase.
But OAuth,SAML authentication implementation takes a long time and is hard to implement.
In Firebase, it has firebase authentication for mobile app and it is pre-built in and very easy to apply. By supporting more SNS id in authentication, it can help more users to join the service easily.

Personal information scanning

Storing personal information is important. If the storage and system is designed to store the personal information, it is not a problem. But sometimes unexpected personal information can be stored in the system which is not designed to store the personal information.
Particularly the chatting message can have a lot of personal information and it can be against compliance. To prevent this problem, google cloud provides machine learning based DLP api which can detect personal informations like email, phone #, ssn etc for multiple nations.
Not just for the text message, it can detect personal information in the images.
The DLP api can be applied in two different ways

- Scan stored data
  It can scan stored data in GCS or BigQuery to d
- Scan streaming data
  It can inspect text string or image by using API

Prevent explicit contents

One of the problems in chatting & SNS apps is explicit contents, such as adult, spoof, medical, violence and racy. These types of contents cause hatred.

Google Cloud Vision API can detect the explicit image automatically and you can remove the image (or blur) to prevent the sharing the explicit contents

Monitoring

Knowing customer behavior is important. Much of user event information can be gathered thru firebase and infrastructure metrics can be monitored by Cloud Operation Suite - Monitoring module.
But it is hard to get application specific metrics like

- Concurrent users per NRU
- Average response time for the API etc

It can be gathered at the application level. However all monitoring metrics should be centralized to a single place and provide a unified view. Google Cloud Operation Suite supports logging and monitoring. In monitoring, it supports custom metrics, which can define new metrics from the application. The metrics can be displayed in the dashboard with other infrastructure metrics.
To capture the application centric metrics, it can send the metric value by using API.

Other option is that we can extract the metrics from the log
For example if you have latency value in log string, you can extract the value and use it as a monitoring metrics. The metrics are handled as common metrics in Cloud Operation monitoring systems. And can be shown together with other metrics in dashboard
(Note : this is a recommended way because it doesn't need to add monitoring code in the application. It simply needs to add required fields in log string)

Another alternative solution is using prometheus, which is an open source monitoring repository.(it is used in Kubernetes as a monitoring metrics repository)
It is widely used but it has cons

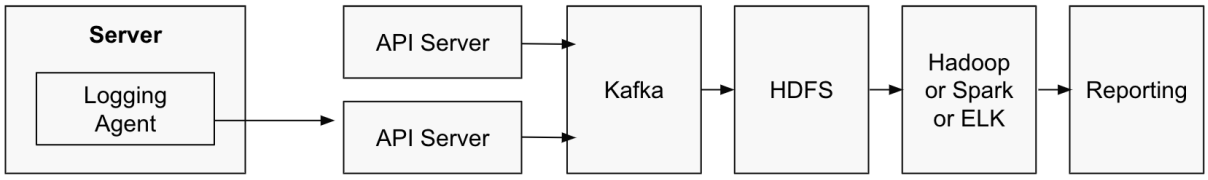
- It cannot support scalability (single instance only)
- It cannot support HA

Those problems can be solved with other open sources like thanos but it is still hard to manage.
To solve this problem, google cloud provides managed prometheus service which removes the cons in the open source. Please be aware that it is the preview phase.

Logging architecture

Logging takes lots of resources to build and maintain. In addition, the modern logging system is used as a pipeline for **data collection** as well as logging for troubleshooting.
Cloud operation suite logging is very good to fit this requirement.

Data from the server side can be collected through a logging pipeline. In traditional log gathering and analytics architecture. It uses open source based methods like below:

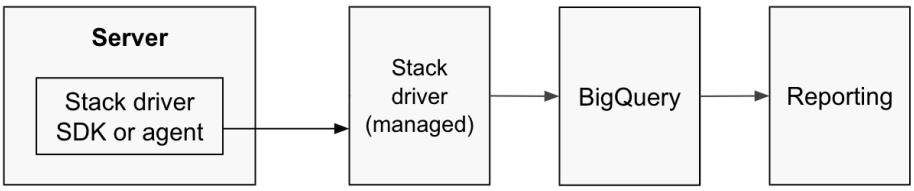


<fig. Traditional log gathering and analytics architecture >

- Log ingestion was made by user application
- Collected log is transferred to API server
- It will be sent to Kafka for queuing.
- The log will be stored in file systems like HDFS or AWS S3 etc.
- Log analytics is done by using Hadoop and Spark.

As an alternative solution, ELK(Elasticsearch, Logstash and Kibana) stack is also widely used. But the problem of this architecture is that it needs lots of implementation & operation overload and needs a high learning curve to handle each product. And a data analytics job itself takes a long time to run a spark/hadoop based job.

Modern big data analytics architecture is relying on managed services like Google bigquery. It will remove operation complexity and the learning curve. It helps engineers to focus on data analytics itself.



<fig. cloud operation suite based log gathering and analytics architecture>

Log Gathering

cloud operation suite logging is a managed logging solution. It provides two options for collecting logs.

Option 1. SDK
It provides SDK which is integrated with well-known logging frameworks, such as slf4j in java

Here is a Java example. You don't need to change code, you can simply add cloud operation suite logging library into maven pom.xml and use standard slf4j api like below

Maven pom.xml

```
<dependency>
<groupId>com.google.cloud</groupId>
<artifactId>google-cloud-logging-logback</artifactId>
<version>0.30.0-alpha</version>
</dependency>
```

Java example

```
package com.google.example.stackdriver;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;


public class App {
 private static final Logger logger = LoggerFactory.getLogger(App.class);

 public static void main(String[] args) {
   logger.info("My Hello Log4j");
 }
}
```

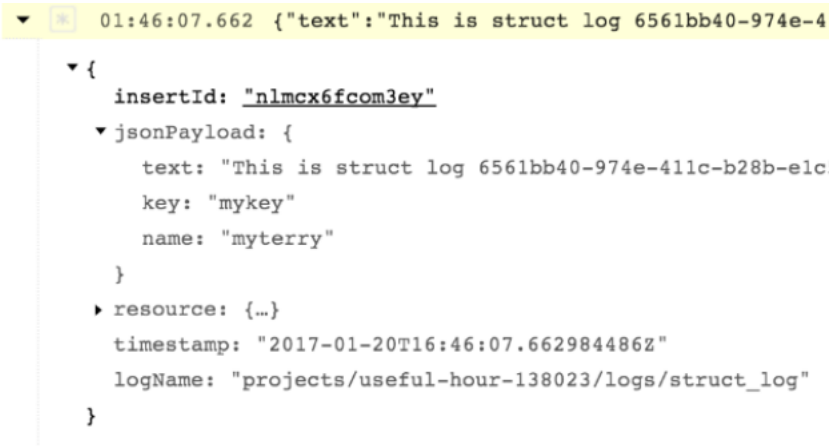
Here is a guide of cloud operation suite logging SDK
https://cloud.google.com/logging/docs/reference/libraries#client-libraries-install-java

Structured log format

If it stores a log with a single string, it needs an additional parsing phase after it has been stored. Cloud operation suite logging can support structured log format with json.
Here is node.js example for structured log

```
@app.route('/slog')
def struct_log():
  struct  = "This is struct log "+ str(uuid.uuid4())
  slogger.log_struct({
          'name':'myterry',
          'text':struct,
          'key' : 'mykey'})
  return struct
```

The structured log payload will be stored under json Payload element in a log like below



When it is exported into BigQuery (big data analytics database), each of the elements in the json Payload will be mapped into separate columns.

ı.first	operation.last	jsonPayload.name	jsonPayload.key	jsonPayload.text	trace
		myterry	mykey	This is struct log 69c515d7-a319-4448-baf5-fe627d9e876f	null
		myterry	mykey	This is struct log 2016b335-bcd7-4d58-8822-f2656548c8b1	null
		myterry	mykey	This is struct log 9030ae97-b238-4699-8cdc-1d7d544b1df6	null
		myterry	mykey	This is struct log 8de022c8-0498-4112-991d-158e8a9653f7	null
		myterry	mykey	This is struct log 819be625-6a73-40ac-b977-a8e24aabc601	null
		myterry	mykey	This is struct log 39b9fc95-5cc4-4b03-a3db-068938a01ee3	null

<fig. Exported logs in BigQuery>

Option 2. Log agent

If it is hard to change the application code, there is an alternative way to use log agent. https://cloud.google.com/logging/docs/agent/ log agent is a fixed version of log collecting open source (fluentd). It will consume log files and send them into the cloud operation suite logging server.

Benefit of the agent way is that it doesn't need to change application code and can be applied regardless of application language.

After a log is stored in cloud operation suite, the user can query the log with a search filter. cloud operation suite retains the log in 30 days. After 30 days, it will be removed. cloud operation suite log will stay 60 hours in high speed storage. After 60 hours, it will be moved to lower speed storage. It can provide lower search performance compared to high speed

storage. If it is still required to search, it is recommended to export the log into BigQuery and search the log in a big query.

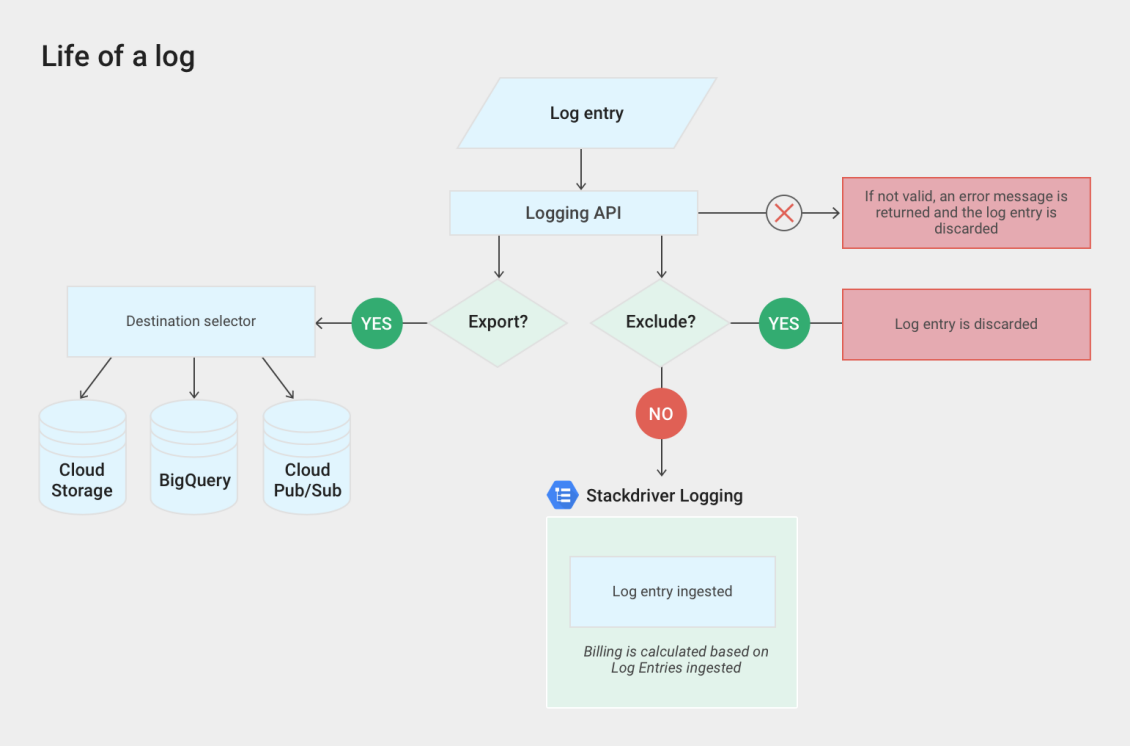
For your information, when it designs the log, it needs to consider quota and limitation of log size etc. For more details, please refer to this document
https://cloud.google.com/logging/quotas

Log Exclude & export

Exclude
If it stores all of the logs into cloud operation suite, it can cause a cost issue. cloud operation suite logging charges are based on stored log size. Here is the pricing details for cloud operation suite logging https://cloud.google.com/cloud operation suite/pricing
To save the cost of cloud operation suite, it can store the logs selectively by using the log exclusion feature (https://cloud.google.com/logging/docs/exclusions)

Export

Exclusion is a good way to save cost but it can lose the original log. Cloud operation suite provides an export feature and it can export original log before exclude.



<fig. Lifecycle of log in Stackdriver >

Log data can be exported to multiple destinations.

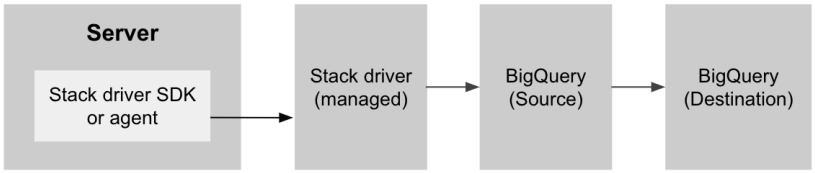
- BigQuery : Data storage and analytics system based on SQL

- GCS : Object storage to store log files. (cf. AWS S3)
- Pub/Sub : Managed Queue service to stream data to multiple subscribers. (cf. Kafka)

Best recommendation for an export destination is BigQuery. BigQuery storage cost is cheap compared to S3 and it can also directly run analytics queries in BigQuery. BigQuery is good for data lake and batch style data analytics.
If it needs real-time data analytics like spark stream or simple ETL, Pub/Sub is a good option for this scenario

Export data from cloud operation suite to BigQuery

The easiest way is using the BigQuery destination table feature.
https://cloud.google.com/bigquery/docs/writing-results#saving_query_results_to_a_table
After the original logs are stored in BigQuery, it can simply run data transformation with SQL query and then store the result into the destination table.



<fig. Stackdriver to BigQuery integration architecture with ELT(Extract-Loading-Transform) >

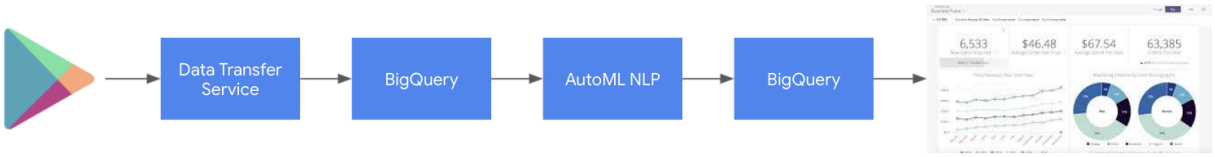
Distributed Transaction Tracing

Zipkin node js https://github.com/openzipkin/zipkin-js

Measuring customer feedback

In mobile apps, it is important to anticipate customer needs. Google play store is a good channel to listen to feedback from the customer, which has app rating and feedback (with text).
By analyzing the text comments, it can understand what the customer wants.
But gathering comments and analyzing the natural language is not easy. Fortunately google cloud provides Data Transfer Service to gather the data and Natural Language Machine Learning API to analyze the text.

Here is the architecture for this specific use case:



- Ingest google play store by using Data Transfer Service
- The transferred data will be stored in BigQuery
- AutoML NLP will consume the data from the BigQuery and run the prediction
  One of the recommended scenarios is classifying. The feature can classify the text messages into a number of classes.
  To do that, the AutoML NLP model needs to be trained with your data. To train the model, you need to label (categorize) the message by yourself manually. If you don't have a resource to do it, you can use outsourcing with google cloud AI Platform Data Labeling services. If you provide the guidance about the labeling, google cloud labeling team will categorize your data and label it.

  If you don't have time to train the AutoML NLP model, there is an alternative way. Google Cloud NLP API is a pre-trained API. The API has sentimental analysis and Entity analysis. Sentimental analysis can be used to understand the satisfaction of the feedback. But the satisfaction can be measured by review score, hence it is not recommended. Entity analysis can extract nouns, verbs, adjectives etc from the text. Noun can represent the issue (crash) and adjectives can represent satisfaction (ex. Stable, fast,slow,good,bad etc). After extracting the noun and adj, you can count it with time windows (1 hour etc). By doing this you can understand the trend of the major issue.
- After analyzing the data, the result can be stored in to BigQuery again
- The analyzed result can be visualized with a dashboard, which can be built with a data studio or looker. The data studio is free BI tool with minimum feature, looker is commercial (managed service) with advanced features

Similar pipeline were implemented in other customers, full implementation time was 6 hours with one engineer and the learning curve is very fast (who started using google cloud 3 months ago)