Date: 11th April 2019

CMPT 756: Systems for Big Data (Spring 2019) – Final Project

Design Analysis of WePay's Highly Available MySQL Architecture

Anurag Bejju – abejju@sfu.ca

1. Introduction

Over the last few years, the payment industry has evolved rapidly, making it difficult for vendors and other small business owners to keep up with industries latest standards and practices. To support them, there has been a wave of new value-added entities offering large benefits to merchants by simplifying complex payment processes with just a single API integration tool. *WePay* is one such company that provides a simple payment processing integration solution that allows organizations to process payments derived from credit/debit card to automatically flow into a business' account when a sale is made. Their payment technologies have been uniquely designed to meet requirements for each type of business model by providing a great platform that can solve unnecessary 3rd party redirects and lengthy merchant onboarding. They ensure users have control over the experience by enabling payments in seconds and provisioning merchant accounts seamlessly behind the scenes.

In order to adequately facilitate these valuable features to their increasing client base, it has become crucial for them to build a reliable system architecture that can be scaled as the company grows. From its very foundation in 2008, WePay has been using MySQL as its only backend storage system which served the purpose well when data volume and traffic throughput were relatively low. But as its business was growing rapidly, they started to witness a performance degradation to the point where they could no longer run concurrent queries without a negative impact on its latency. In 2016, WePay became the top 500 fastest growing privately held startups with over *1500%* **[1]** growth from the prior year. It was paramount at this point for the big data system designers to come up with a sustainable design that can help them support their growth.

2. Background

In order to better assess their current systems architecture, it would be really helpful to understand their previous implementation. It will also be a good starting point to interpret their design requirements as well as comprehend some pain points that need to be addressed.

2.1. Components

WePay's previous failover architecture had three major components to it.

- *Master High Availability Manager*
- *HAProxy*
- *Google routes*

2.1.1. Master High Availability Manager (MHA) [2] :

In recent time, a good amount of the database system designs follow *a Primary-Secondary architecture* for MySQL database where most of the writes/changes go to a single machine called as *primary database* and every other *secondary database* replicates from it. In a scenario like this, we need solutions that can felicitate the process of choosing a new primary if an existing master fails. Since a manual process can take up to 1-2 hours, Master High Availability Manager can automate the failover process and complete it in a very short amount of time. Some of the major benefits it provides are automatic failover, short downtime, MySQL-replication consistency and requires fewer changes at the application level.

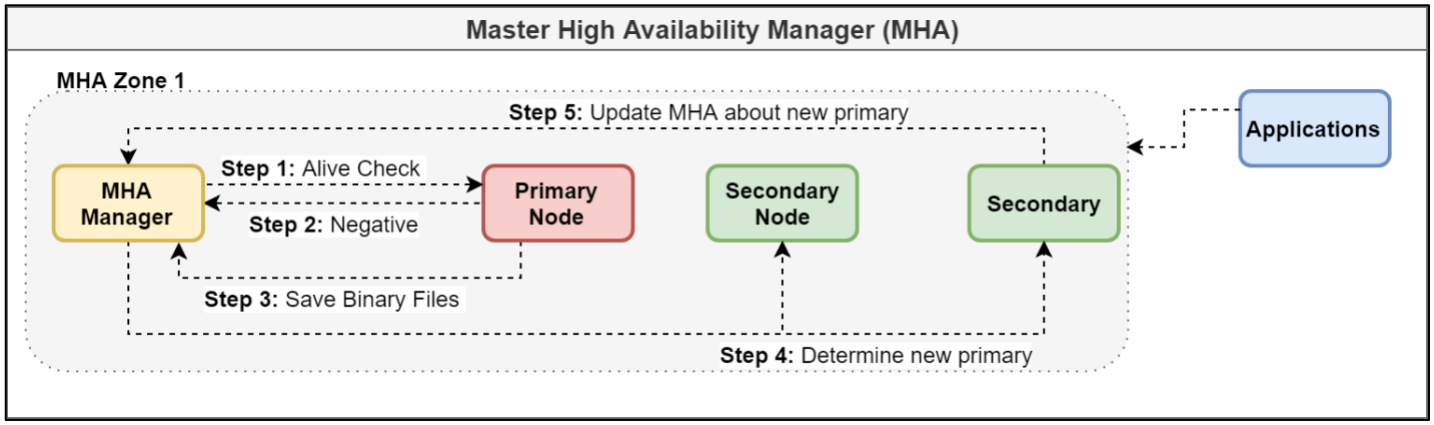

Fig 1: Master High Availability Manager Process Flow

2.1.2. High Availability Proxy [3] :

HAProxy is a popular open source software *TCP/HTTP Load Balancer* and proxying solution that can be run on Linux, Solaris, and FreeBSD. It is mostly used to improve the overall performance and reliability of a server environment by distributing the workload across multiple servers (e.g. web, application, database). It has been used in many high-profile environments, including: GitHub, Imgur, Instagram, and Twitter. Since the Standard HAProxy does not allow to change the pools at run time, WePay used a patched version that can modify them at run time. It updates the primary and secondary pools dynamically.

2.1.3. Google Routes [4] :

Google Cloud Platform (GCP) routes define the path network traffic takes from a VM instance to other destinations. These destinations can be inside a VPC network (for example, in another VM) or outside of it. Every route consists of a destination and a next hop. When a failover process is initiated, it removes one route and adds another route to the same IP address pointing to the new instance. This helps to manage failovers between HAProxies.

2.2. Previous Failover Process [5] :

WePay used a MHA with a patched version of HAProxy to handle dynamic changes in the pool configuration. During a failure process, MHA kills long running updates and inform HAProxy Primary pool about the primary node going into maintenance mode. Then it initiates the role transition to a new master and again informs HAProxy about the new primary node in place. Failover of a primary HAProxy itself was taken care by using Google Routes which could be used to route client traffic to a healthy HAProxy instance.

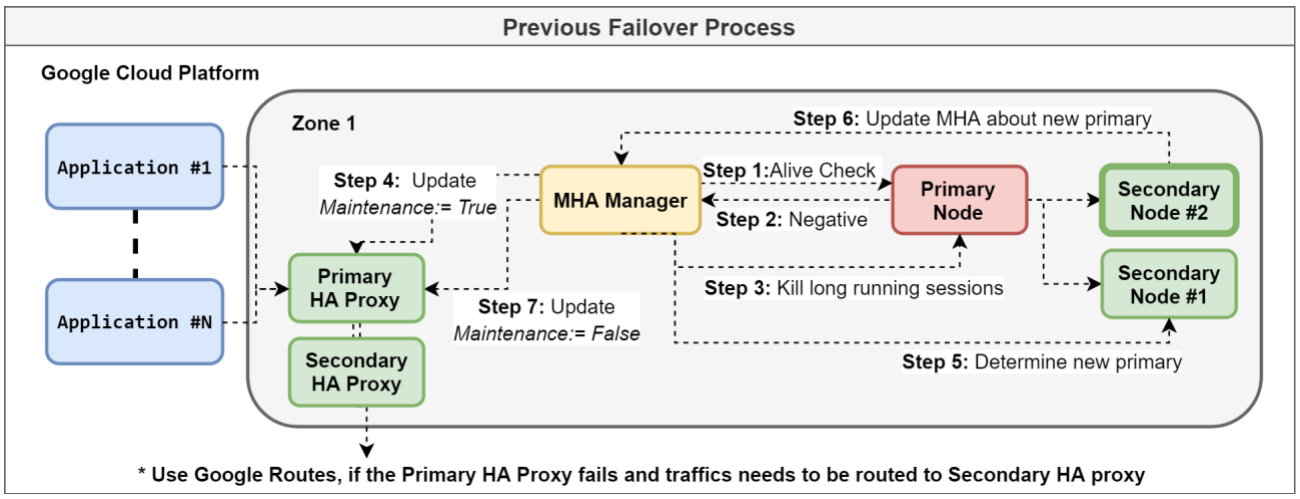

Fig 2: Previous Failover Process

2.3. Challenges:

HAProxy was a single point of failure and the challenge was compounded when Google Cloud had network issues and the routes could not be updated immediately. This resulted in a minimum downtime of around *30 minutes*. In addition to this, replication lag which was calculated by pt-heartbeat would have time skews between the MHA server and the replicas leading to miscalculations. Managing HAProxy config dynamically was also a challenge.

3. Current Architecture:

3.1. Components:

In order to better address limitations witnessed in their previous architecture and to come up with a scalable design that can help them sustain their growth, they built a new high availability solution, coupled with changes to pt-heartbeat that can handle time skews. Here are the components used to build this architecture

3.1.1. Orchestrator:

Orchestrator is an open source MySQL replication management and high availability solution. It observes MySQL replication topologies, auto-detects topology layout and changes, understands replication rules across configurations and versions, detects failure scenarios and recovers from master and intermediate master failures [6].

Impressive features of Orchestrator			
Failure detection	***Graceful failover:***	***Process Continuation*:**	***HOOKS***
If the primary node is down, Orchestrator will check with all the secondary nodes that the primary node is really down or not. It tries to seek agreement between itself and the replicas about the state of the master and decide if the replication cluster is broken de-facto.	This can be very useful when performing maintenance activities on all the nodes without breaking the replication and have a major downtime.	Once the failover has been completed, Orchestrator will facilitate the secondary nodes to replicate from the new master.	It also provides HOOKS in order to perform/run /automate any scripts during the detection of failure/pre-failover/post-failover.

WePay runs Orchestrator in a raft setup and has one server in each Google zone within a single region. It is used to conduct failure detection, perform role transitions and communicate about the primary role transition to the consul. Since the Orchestrator will be running on a raft mode, a single node is elected as the leader to effectively perform the role transition and communicate changes to other nodes in the raft. In addition, this leader should also update the consul KV store about these changes.

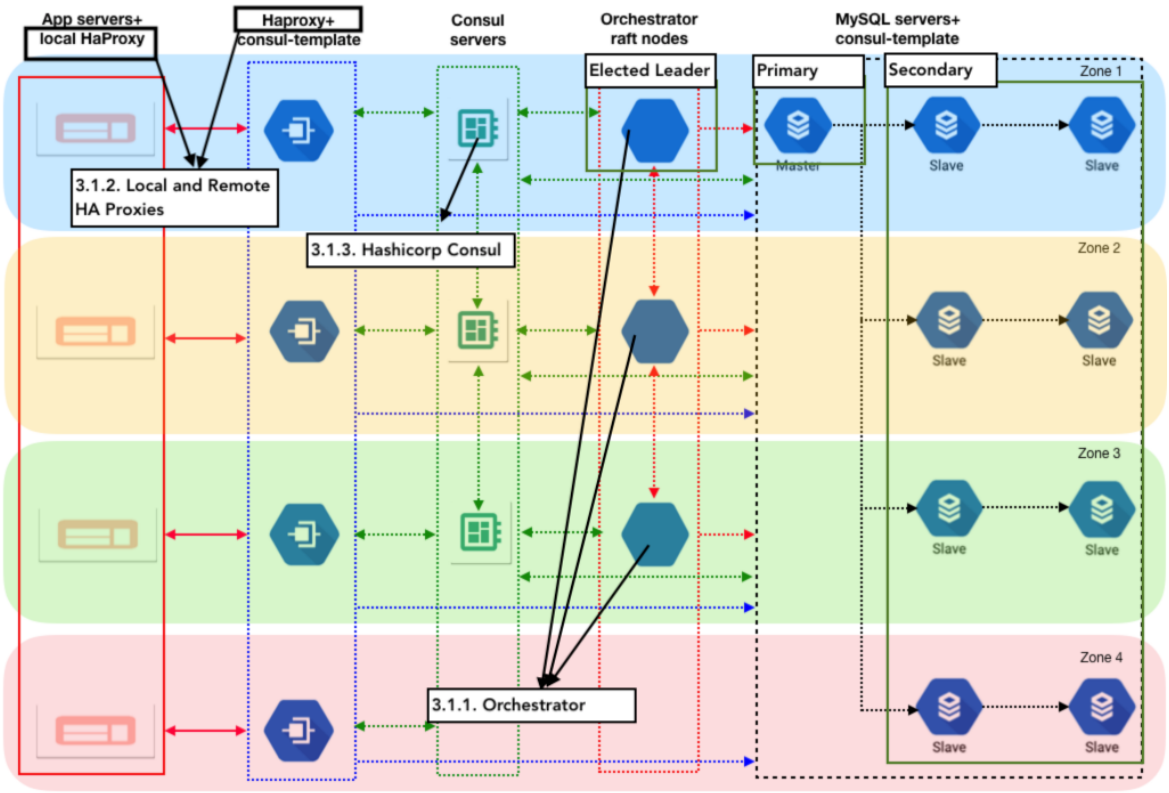

Fig 3: Current Architecture – Components and Structure

3.1.2. Local and Remote HAProxy:

As discussed in the above section, HAProxy is the world's fastest and most widely used software load balancer, that allows the proxy to know when a node is offline. In contrast with the previous architecture, HAProxy runs in two layers in the new one. A single HAProxy instance runs with each application client, either as a local process or as a sidecar in their Kubernetes deployments. These applications connect to a remote HAProxy located in a different zone. This is done to avoid a unique scenario where the MySQL master, the Orchestrator leader node and the remote HAProxy are all in the same zone. During an unforeseen scenario where a network issue arises, this zone containing clients and one of the remote HAProxy could become completely isolated.

In such a case, the primary orchestrator node becomes unreachable and a new leader is elected. When the new leader detects that the primary MySQL primary node is isolated, it updates all remote HAProxies except for the affected HAProxy about the promoted MySQL primary node's identity. Even though the clients in the zones that were not impacted start writing to the new master, the clients in impacted zones still connect to the remote HAProxy in it's same zone and write to the demoted primary node.

This is called *split-brain scenario* [7] where we have a new master in place and the affected zone has no idea about it. In order to avoid such a scenario, clients are connected to a remote HAProxy in a different zone. To felicitate this process, the first layer of local HAProxy sits on the client machines and connects to the remote (second layer) of HAProxy. The second layer of HAProxy is distributed across multiple Google zones that connect to the same set of MySQL servers. The remote HAProxy instances are aware of the current topology, and the local ones connect to the remote ones to get updated master information.

3.1.3. Hashicorp Consul for the KV store [8]

Hashicorp Consul provides a service discovery solution with DNS services. WePay utilizes it as a highly available key-value (KV) store. A Consul's KV store contains important information about identities of cluster masters and stores a set of KV entries indicating the cluster's all primary and secondary nodes fqdn, port, ipv4, ipv6 values. In the current architecture, each HAProxy node runs consul-template that listens on changes to clusters primary and secondary nodes data and dynamically produce a valid configuration file.

It also automatically reloads HAProxy whenever there is any change to it. Once this change is observed in Consul the identity of the primary node, each HAProxy reconfigures itself by setting the new master as the single entity in a cluster's backend pool, and promptly reload to reflect it. WePay deploys consuls across multiple Google zones in a single region making it highly available. They don't get affected by unexpected network issues in a single Google zone.

3.1.4. pt-heartbeat injection [9]

Instead of managing the startup/shutdown of the pt-heartbeat service on promoted/demoted primary node, they opted to have pt-heartbeat script running from all secondary nodes. This is a little different as compared to the MHA machine in the former architecture where a single server runs a pt-heartbeat script. Pt-heartbeat is measured by running it on each replica. A row is inserted into the master and then look for that row in order to capture replication lag during it. With each secondary node calculating its own lag, it eliminated the need to depend on a single server to effectively it.

This setup also ensures time skews don't result in slave lag. Since pt-heartbeat is writes to the primary node, it is vital to always know about its identity. This can be achieved by using consul-template on all MySQL servers, which listens to the same KV store for mastership information. When a primary node is changed in the KV store, consul-template also updates the pt-heartbeat config and restarts it.

4. Architecture:

Using the above-mentioned components, WePay was able to design an architecture that addressed most of the limitations present in its previous implementation. It also helped them in simplifying and decoupling failover tasks into several subtasks.

4.1. Normal Workflow:

WePay has both *monolith* and *micro service applications* as part of their environment, and are spread across multiple zones inside a Google region. For monolith applications HAProxy will be running locally whereas for microservices which reside in Kubernetes, they run an HAProxy sidecar with remote HAProxy configurations.

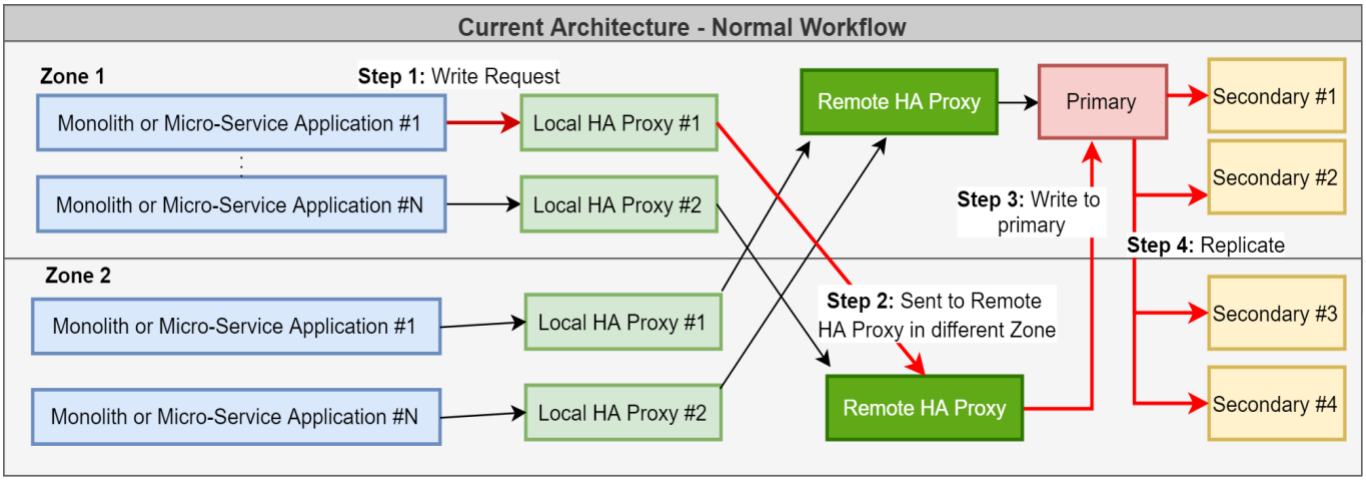

Fig 4: Current Architecture – Workflow

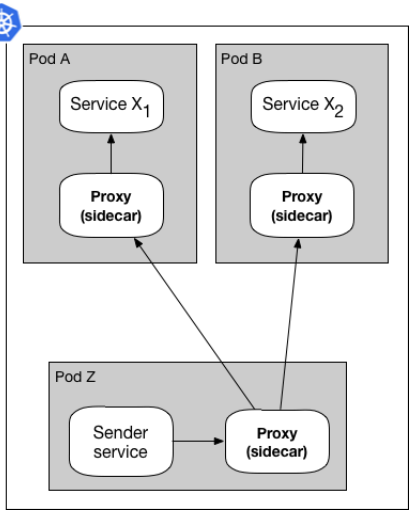

Fig 5: HA Proxy Sidecar Implementation

A *HAProxy sidecar* [10] is an instance of HAProxy that listens on localhost so that the application can talk to the remote endpoints that it needs to without needing to know exactly where those endpoints live. Routing traffic through a local proxy can beneficial in order to enforce fine-grained authorization rules.

The HAProxy handler continues to check for changes to service registrations and updates itself when needed. Due to this setup, applications will not have visibility into who the current primary is. The application connects to write and read MySQL hosts via its local HAProxy. The local HAProxy passes the request to the remote (second tier) HAProxy, which is aware of the current primary and its secondary nodes.

In the previous implementation, HAProxy was a limitation which used Google network routes in its failover process. In order to address this in the new architecture, multiple HAProxy instances were introduced to achieve high availability in this situation. Applications connectivity in the new design end locally.

4.2. Planned Failover Scenario [11] :

A planned failover is a process of smoothly switching from a primary node to its replica with minimum interruption in its operation. This can be initiated if you plan to perform maintenance, data center migration or software upgrade of the primary nodes. You can also schedule it if advance notice is issued about any network outage that might take primary nodes offline. Here is the workflow when a planned failover process is initiated by WePay's application support engineers:

Step 1: First the orchestrator primary node gets all the secondary nodes behind the elected candidate primary node.
Step 2: Then it performs a series of pre-failover checks and ensures no DDLs (i.e Data Definition Language tasks), no errant transactions are running on the newly elected primary node. Data Definition Language is a series of statements that deal with database schemas and descriptions on how the data should reside in a particular MySQL database.
Step 3: Then it also checks if any Data Manipulation Language queries are running on the demoted primary node. Data Manipulation Language deals with data manipulation and includes most common SQL statements that are used to store, modify, retrieve, delete and update data in a database.
Step 4: Then it also checks if consul-client and consul-template are running on remote HAProxy.
Step 5: After these checks are completed, the orchestrator primary node initiates the role transition process. It first updates all remaining Orchestrator nodes regarding the newly promoted primary node in place.

Step 6: Then it updates the consul KV store and moves the demoted MySQL primary node behind the promoted one.
Step 7: The remote HAProxy is notified and updated about the new primary nodes identity through consul-template.
Step 8: After this, the consul-template updates the HAProxy config file and reloads HAProxy. This now enables the clients to connect to the new master. It also updates the pt-heartbeat config file on all MySQL nodes and restarts pt-heartbeat.
Step 9: After this, Pt-heartbeat starts writing to the new primary node to measure replication lag.

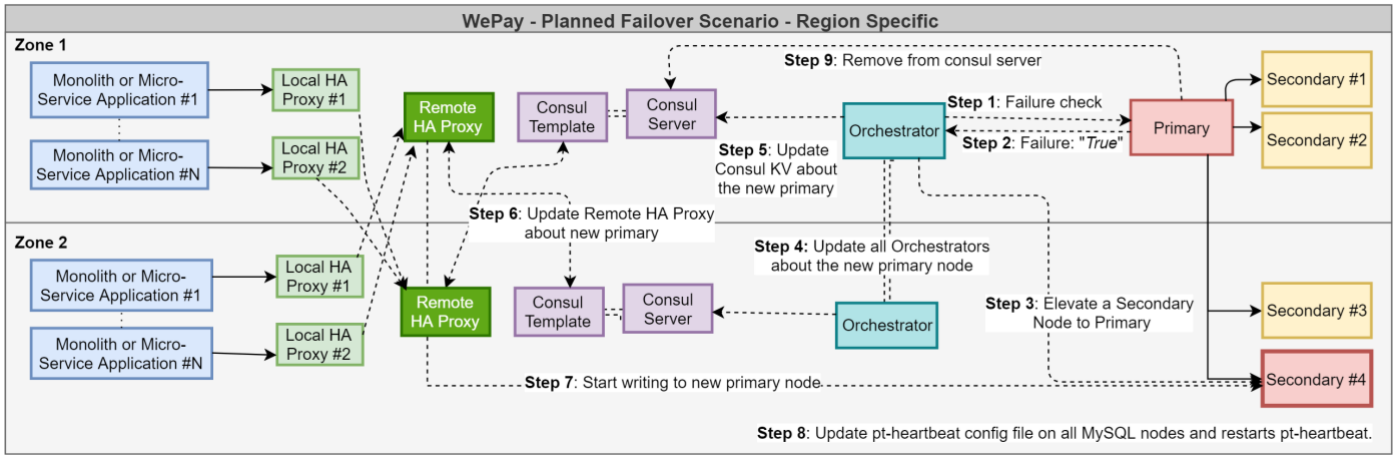

Fig 6: WePay – Planned Failover Scenario

4.3. Unplanned Failover Scenario:

This failover procedure is implemented if the primary node becomes unavailable due to an unforeseen event that might have happened. In cases like unexpected equipment failure or network issues, an unplanned failover process is executed and is slightly different from the planned failover process.

Step 1: The orchestrator first detects the failure when the primary node dies abruptly.
Step 2: This triggers the unplanned failover procedure where it has to initiate the recovery process from the primary node crash and position a new primary node in place.

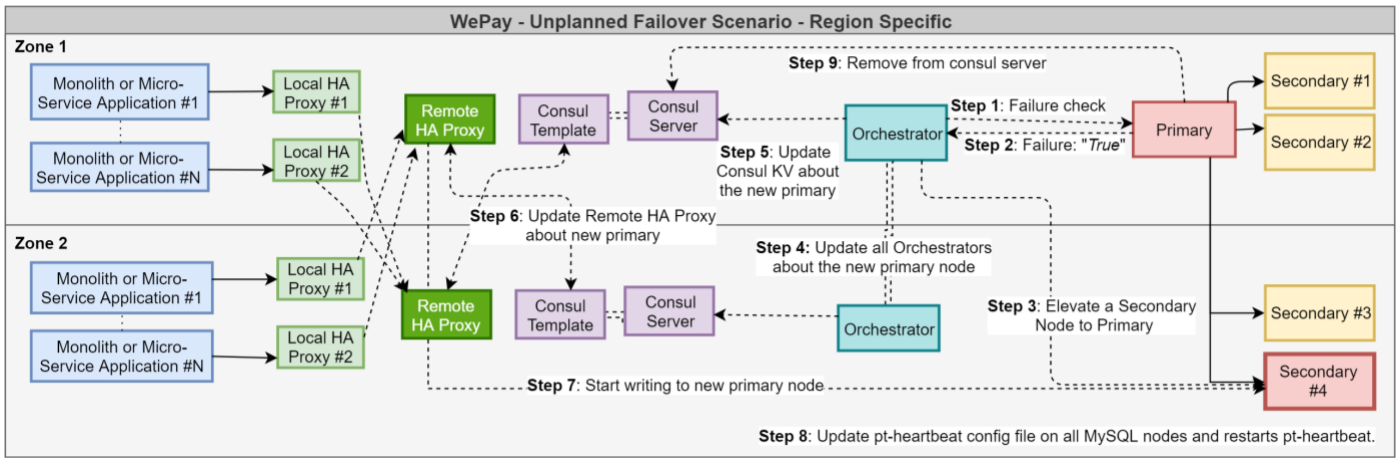

Fig 7: WePay – Unplanned Failover Scenario

Step 3: Then the newly promoted orchestrator leader updates the remaining orchestrator nodes with the new primary nodes identity as well as update the consul KV store about it.
Step 4: Through consul-template updates the HAProxy config file and reloads HAProxy, making it aware about the new primary nodes identity.
Step 5: It also updates the pt-heartbeat config file on all MySQL nodes and restarts pt-heartbeat.
Step 6: At this point, clients start to connect to the new primary node
Step 7: Finally, the failed/crashed primary node is removed from the secondary node pool in consul KV store.

Note: In case the primary node is available, it is always preferred to perform a planned failover process.

5. Design Objectives and Tradeoffs:

In a book called *Beginning database design solutions*, [12] Rod Stephens talks about the very first step in your quest for successful database design is to *understand database goals*. He encourages data engineers to have a fair amount of understanding about the kind of database they want to build as well as have an idea about the use case and limitations they might face. In fact he states that "*working with a powerful database tool without goals is like flying a plane through clouds without a compass: you have the tools you need but no sense of direction*".

From the blog post, it was very clear about the design objectives that were crucial to develop the database architecture at WePay. So here are some of the design objectives that were addressed in its new architecture:

5.1. Resilience to failure and recovery:

One of the course recommended book *Chaos Engineering* by *Ali Basiri, Nora Jones, Aaron Blohowiak, Lorin Hochstein, Casey Rosentha* [13] was a great source to really understand the importance of *resilience* in modern database designs. It talks about the benefits Chaos Engineering brings in order to learn about weaknesses in a system design that could potentially lead to outages which might cause great discomfort to customers. It provides companies to proactively fix these limitations, and go beyond the reactive processes that currently dominate most incident response models.

As we have seen in the previous sections, WePay has properly assessed their previous implementation and came up with an improved design architecture that was much more *resilient* with its failover mechanism. It was very evident in the blogpost and in my conversation with *Akshath Patkar*, the author of *Highly Available MySQL Clusters at WePay* article that failure management ranked very high in terms of their design goals. The workflow designed for planned and unplanned failover scenarios, ensure a reliable primary node role transition takes place.

5.2. Highly Availability:

WePay provides merchants like meetup, gofundme, freshbooks a complete payment solution with its API-based integrated payments system. It is vital for the operations of these companies to have a highly available and resilient payment service that improves the overall user experience. Any prolonged unexpected downtime could greatly affect how usable and enjoyable the user experience is with a particular product or application. Therefore, WePay ensures Highly Availability of MySQL server with its current architecture across a single region. It can withstand complete single zone outage and can manage complete network isolation of a single zone. They have also reduced the total outage time from *30 minutes* to *40-60 seconds* (in the worst case scenario).

5.3. Read-Your-Write consistency:

Consistency wasn't particularly discussed in this blogpost but was talked in an info session conducted by Joy Gao, a senior software engineer at WePay **[14]** . In this video, she sheds light on how they achieve *Read-Your-Write consistency* with just using the inbuilt provisions provided by a MySQL database. Read-Your-Write consistency is the idea that when some data is being updated, the user is expecting to read what was just written. It is really important to ensure stale data is not being read. In a use case like account balance, there needs to be some sort of guarantee that users do not withdraw money to go into negative balance. This is achieved by using the write-ahead log feature offered by a MySQL Database system. Just like its name, it first updates data into a write-ahead log and then update the data into the storage file

It helps in crash recovery where database upon restart can look at the commit log, replay the change and restore the corrupted data. It also improves write performance and provide streaming replication. This particularly helps replicas to just look at the commit log, apply the change, and then update these replicas asynchronously. So WePay specifically uses change data capture with the write-ahead log, to get the best of all the worlds. They need not worry about implementing distributed transaction and still get all of the transactional guarantees. Also since they asynchronously tail a MySQL Binlog, they need not worry about impacting the performance when the data is written into the database asynchronously **[15]**.

5.4. Scalability:

As WePay powers payment experiences across multiple applications and services, it has become a nucleus of a complex ecosystem for companies that require payment processing capabilities for their products or applications. Since its inception, they have used MySQL as there only backend storage which served its purpose well when data volume and traffic throughput were relatively low. But in 2016, their business was growing at a very rapid pace, resulting in serious performance degradation which hugely impacted their ability to perform the most critical operation - *fraud detection*.

This propelled them to create a new stream analytics [16] pipeline for fraud detection that would give them answers to queries in near-real time without affecting their main transactional business system. To achieve this, they have built and deployed a pipeline to production using Apache Kafka and Google Cloud Platform (GCP) services like Google Cloud Dataflow and Cloud Bigtable that provided them with a horizontally scalable storage system with super-low read latency and high write throughput. Some of the components they have used are:

Streaming event source (Apache Kafka): Since their data sources need to serve both streaming and batch pipelines, WePay adapted this to meet their high scalability and low latency needs. It helped them instrument various checkpoints inside their application code to emit Apache Avro messages as Kafka events in real time.

Aggregation engine (Cloud Dataflow): They needed a streaming/batch aggregation engine that had the ability to process a large number of events based on custom event time, unify streaming/batch processing and elastic resource allocation to handle bursty event patterns. They noted that Cloud Dataflow would fulfill all these requirements.

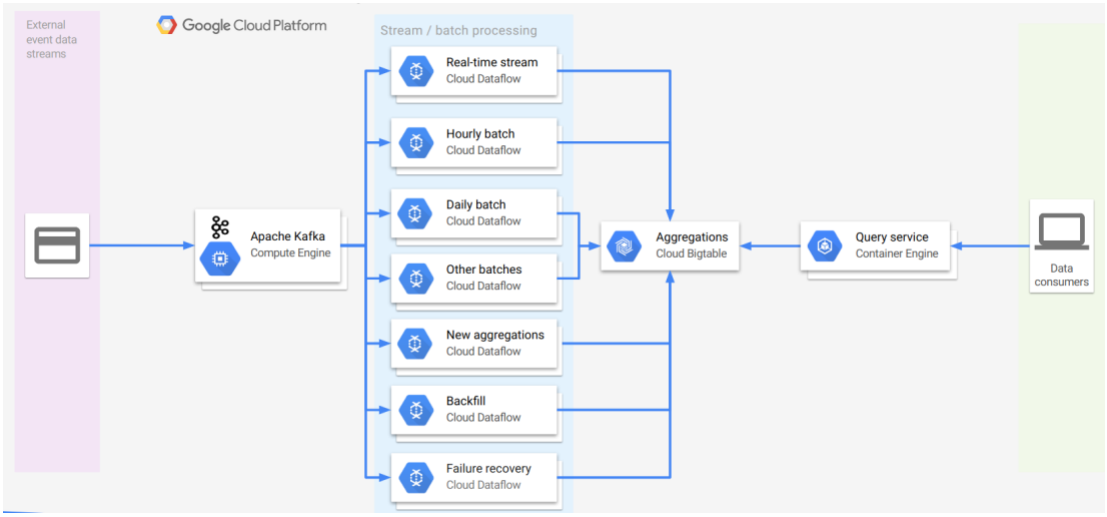

Fig 8: WePay – Fraud Detection stream analytics pipeline

They also wanted to scale their product deployment pipeline by evolving their core infrastructure and broaden their feature set which would eventually deliver faster transactions for their payment processors. Initially, WePay had a monolithic code base that comprised of all their core services. Since they ship all their changes once per week, they used to rely on logs to check for errors and bugs in its code.

This was a problem as a huge error or bug in any part of the app might require a complete rollback which would drain the customers and the company alike. This was definitely not a scalable solution, which prompted them to move to a micro services architecture that provided them with a streamlined deployment pipeline. But migrating away from the monolith while maintaining, or exceeding, previous service levels wasn't an easy task to achieve.

These services could be in completely different languages and might take completely different approaches but must ensure customers should never see any service interruption during their transition. In order to facilitate these migrations, they decided to take a controlled rollout approach by using Split [18] as their platform for feature flags. Split is an intelligent software delivery platform that WePay uses to ship new, discrete features at a rapid pace by testing them internally in a safe and secure environment and then with select customer groups, before eventually rolling out these new features to all of their customers. This helped them scale much faster and provide valuable services to a wider range of customers efficiently.

5.5. Tradeoffs:

This section was particularly difficult to assess as there design goals were very well defined and specific in nature. Upon having a conversation with *Akshath Patkar*, the author of *Highly Available MySQL Clusters* blogpost, he said that tradeoffs are inevitable in every system design but they become insignificant if they are not a deemed as a priority by system designers. On further enquiring, he did speak about the complexity the new architecture brings as compared to its old one.

Particularly, in order to solve HAproxy becoming a single point of failure, they now have three new load balancers (local HA Proxy, Remote HA Proxy and Consul Template) in between client and MySQL servers. Any one of the three are not in sync, the automatic failover plan would halt and manual intervention is required. This is a tradeoff they had to make in order to achieve all their system goals. They have fairly reduced the risk of this scenario by implementing pre failover hooks where if a consul-client or consul-template is down, HA Proxies we will not initiate an automatic failover procedure.

6. Future Work

Even though their current architecture does address most of their design goals, they still have listed few additions/ modifications they would like to implement. Some of those future objectives are:

Update configuration of local HAProxy dynamically: As we have seen the current architecture section, the remote HA Proxies configuration can be changed dynamically using consul-template. But they are now working on changing the configuration for a local HAProxy present in both monolith and microservice applications dynamically.

Change their proxy layer to proxysql: ProxySQL [19] is a high performance, high availability, protocol-aware proxy for MySQL and forks (like Percona Server and MariaDB). In addition, it comes with a GPL license. A proxy layer between applications and databases would typically consist of multiple proxy nodes for high availability. This is no different for ProxySQL. ProxySQL, just like other modern proxies, can be used to build complex logic for routing queries.

You can add query rules to send queries to particular hosts, you can cache queries, you can add and remove backend servers, or manage users that are allowed to connect to the ProxySQL and MySQL. However, numerous ProxySQL nodes in the proxy layer introduce another problem - synchronization across distributed instances. Any rules or other logic need to be synchronized across instances, to ensure they behave in the same way.

MySQL server creation using Google snapshots of data disks: Snapshot feature [20] would enable them to schedule regular, full-system snapshot backups, form their zonal persistent disks or regional persistent disks. As compared to normal backups, snapshots will drastically reduce the recovery time for large data sets (files and DB of more than 10GB). It would also help if the server configuration becomes corrupted or if accidentally there is a loss of important data, enabling WePay to initiate MySQL server creation using Google snapshots of their data disks.

Scale to multiple regions: As we have observed in the current architecture description, the failure mechanism is restricted to just one region. It would be great if they scaled up this architecture to multiple regions.

7. Conclusions:

WePay has been a wrecking force in the payment processing service industry. From becoming the top 500 fastest growing privately held startups with over 1500% [21] growth in 2016 to its founders, Rich Aberman and Bill Clerico being identified as the Best Young Tech Entrepreneurs by Bloomberg Business, WePay's success story definitely intrigued me to find out more about their current architecture as well as get a holistic understanding of their system design that is able to sustain such massive growth. Upon reading blogs, watching videos and speaking with the data engineers at WePay, I was not only able to assess their impressive architecture but also understand core design challenges that needed to be effectively addressed while improving a structure of a system.

As we have learned in this paper, their previous architecture did have some design limitations that needed to be improved to better serve their growing customer base. Some of those limitations were 30 minutes minimum downtime when there is an unexpected master crash to HAproxy becoming a single point of failure. There were also other external issues that aroused due to Google network outage that needed to be considered while building the new architecture. Once they were able to identify their system objectives, they went ahead and rolled out a new robust architecture, that successfully completed planned and unplanned failovers.

Their current architecture was more reliable when a role transition has to be initiated in a planned and unplanned failover scenario. There were also able to make their MySQL server highly available across a single region which made them capable enough to run all their systems even when 2 availability zones are out. In addition to that, it can withstand complete single zone outage and can manage complete network isolation of a single zone.

The new architecture also removed all single point of failures (like HAProxy) and provided a capability to withstand a master crash. Lastly, they were able to reduce the total outage time 40-60 seconds (in the worst case scenario) from 30 minutes before.

Even though the current system design does resolve a lot of limitations present in its previous architecture, there are few things that they still intend to add/improve as part of their future work. They would like to add a process change that would update local HAProxy dynamically as well as change their proxy layer to proxysql. They would also like to enable the MySQL server creation using Google snapshots of data disks and scale up their failure resilience architecture to multiple regions. In this paper, we have also discussed few tradeoffs like multiple load balancers in between client and MySQL servers which should always be in perfect sync about primary and secondary node information in order to initiate an automatic failover process.

Overall, WePay's architecture was a great source to learn about the nuances of system design as well as understand the importance of design objectives and assessing tradeoff's while building a structure. Even though they were multiple aspects that really impressed me, the automated failover procedure was particularly interesting as it could be initiated even if 2 out of 3 zones become unavailable. Looking at their current system design as well as their future stated work, it would be very interesting to see how they advance from here.

8. Acknowledgment

I would like to express my deepest appreciation to Akshath Patkar - Site Reliability Engineer at WePay as well as the author of Highly Available MySQL Clusters blogpost for taking the time to provide great insights and extending help with my analysis. I would also like to thank Dr. Ted Kirkpatrick - Associate Professor of Computing Science – SFU for his constant support and advice during the course of this project.

9. References

[1] https://blog.wepay.com/2016/02/03/2015-a-year-of-growth-and-transition-for-wepay-2/
[2] http://yoshinorimatsunobu.blogspot.com/2011/07/announcing-mysql-mha-mysql-master-high.html
[3] https://www.digitalocean.com/community/tutorials/an-introduction-to-haproxy-and-load-balancing-concepts
[4] https://cloud.google.com/vpc/docs/routes
[5] https://wecode.wepay.com/posts/highly-available-mysql-clusters-at-wepay
[6] https://github.com/github/orchestrator
[7] https://severalnines.com/blog/how-recover-galera-cluster-or-mysql-replication-split-brain-syndrome
[8] https://github.com/hashicorp/consul
[9] https://rimzy.net/category/pt-heartbeat/
[10] https://wecode.wepay.com/posts/using-l5d-as-a-service-mesh-proxy-at-wepay
[11] https://helpcenter.veeam.com/docs/backup/hyperv/planned_failover.html?ver=95u4

[12] Stephens, R. (2009). Beginning database design solutions Rod Stephens. (Wrox programmer to programmer). Indianapolis, IN: Wiley Pub.
[13] Chaos Engineering - Ali Basiri, Nora Jones, Aaron Blohowiak, Lorin Hochstein, Casey Rosenthal - August 2017
[14] The Whys and Hows of Database Streaming - Joy Gao, a senior software engineer at WePay - https://www.youtube.com/watch?v=0K0fYHsFBZg -
[15] https://www.infoq.com/presentations/wepay-database-streaming
[16] https://cloud.google.com/blog/products/gcp/how-wepay-uses-stream-analytics-for-real-time-fraud-detection-using-gcp-and-apache-kafka
[17] https://www.globenewswire.com/news-release/2017/07/25/1057724/0/en/WePay-Chooses-Split-to-Significantly-Enhance-Customer-Experience-with-Improved-Software-Delivery.html
[18] https://wecode.wepay.com/posts/splitting-traffic-with-SplitIO
[19] https://severalnines.com/resources/tutorials/proxysql-tutorial-mysql-mariadb
[20] https://cloud.google.com/compute/docs/disks/create-snapshots
[21] https://en.wikipedia.org/wiki/WePay

10. Image References

[Fig 1] https://www.slideshare.net/ohnew/mhamaster-high-availability-manager-and-tools-for-mysql *
[Fig 2] https://cloud.google.com/solutions/best-practices-floating-ip-addresses *
[Fig 3] https://wecode.wepay.com/posts/highly-available-mysql-clusters-at-wepay
[Fig 4] https://wecode.wepay.com/posts/highly-available-mysql-clusters-at-wepay *
[Fig 5] https://wecode.wepay.com/posts/using-l5d-as-a-service-mesh-proxy-at-wepay
[Fig 6] https://wecode.wepay.com/posts/highly-available-mysql-clusters-at-wepay *
[Fig 7] https://wecode.wepay.com/posts/highly-available-mysql-clusters-at-wepay *
[Fig 8] https://cloud.google.com/blog/products/gcp/how-wepay-uses-stream-analytics-for-real-time-fraud-detection-using-gcp-and-apache-kafka

*These images have just been used as a reference point to draw the process flow diagrams. The actual images used in this project are designed by me using draw.io application.