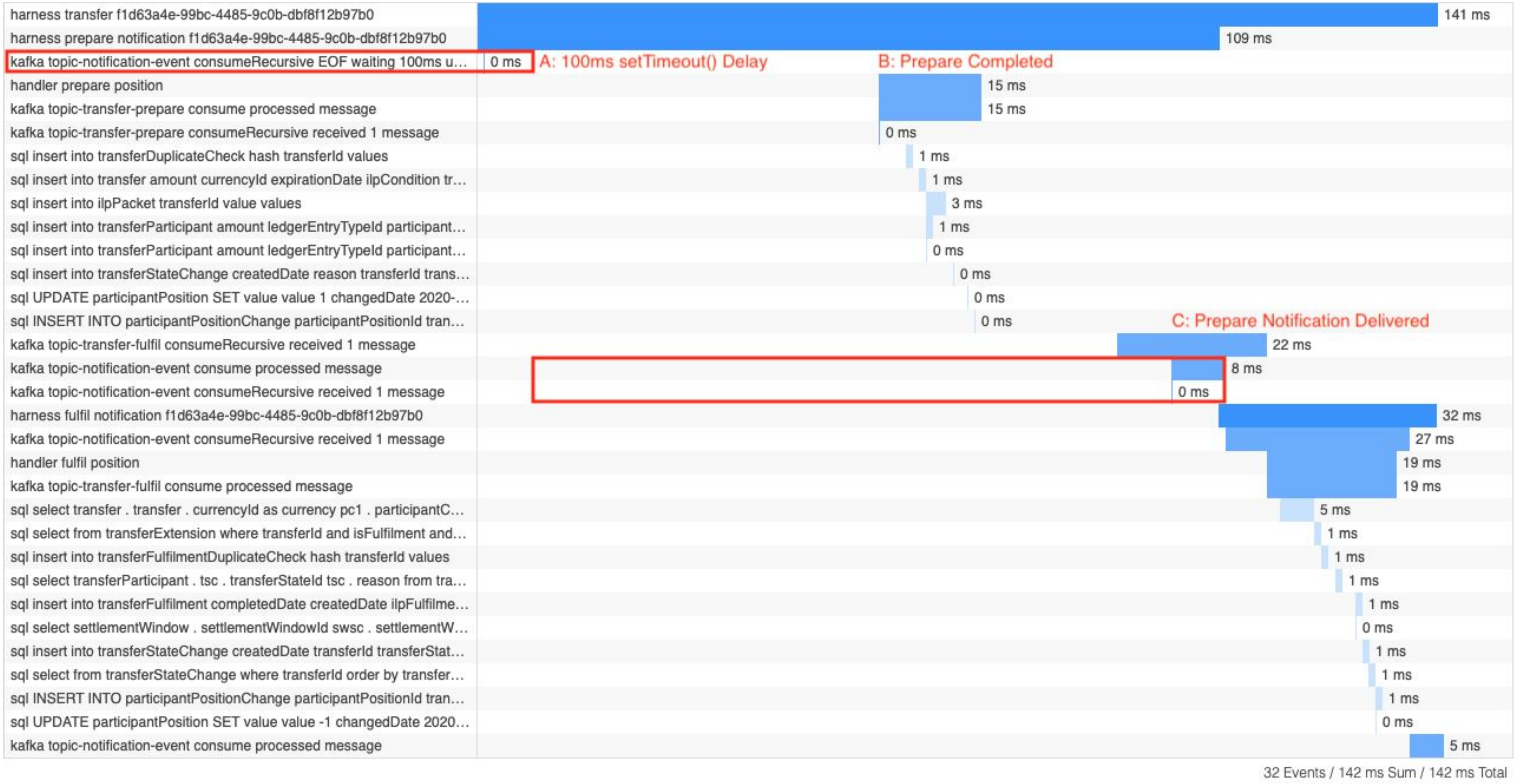
Performance

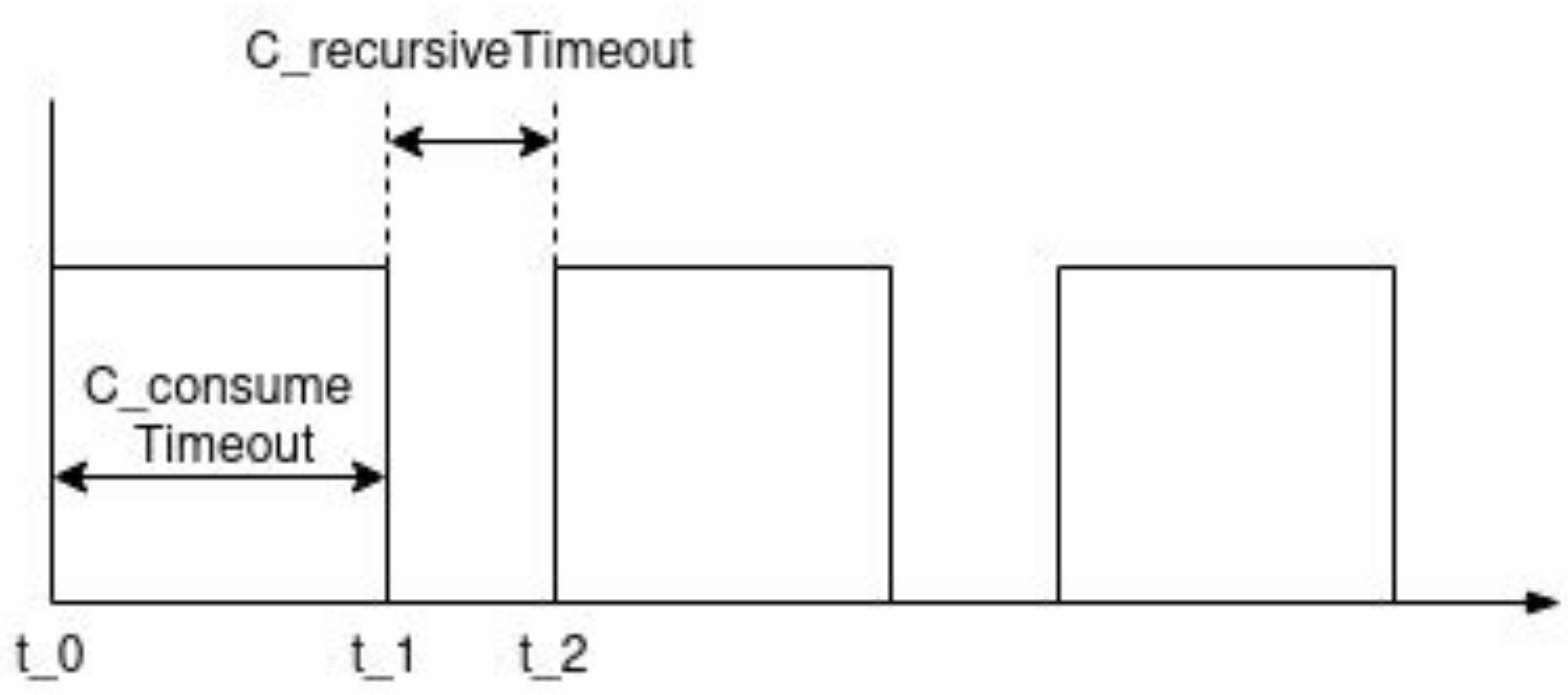
Big Gap update
Scalability POC

Big Gap Update



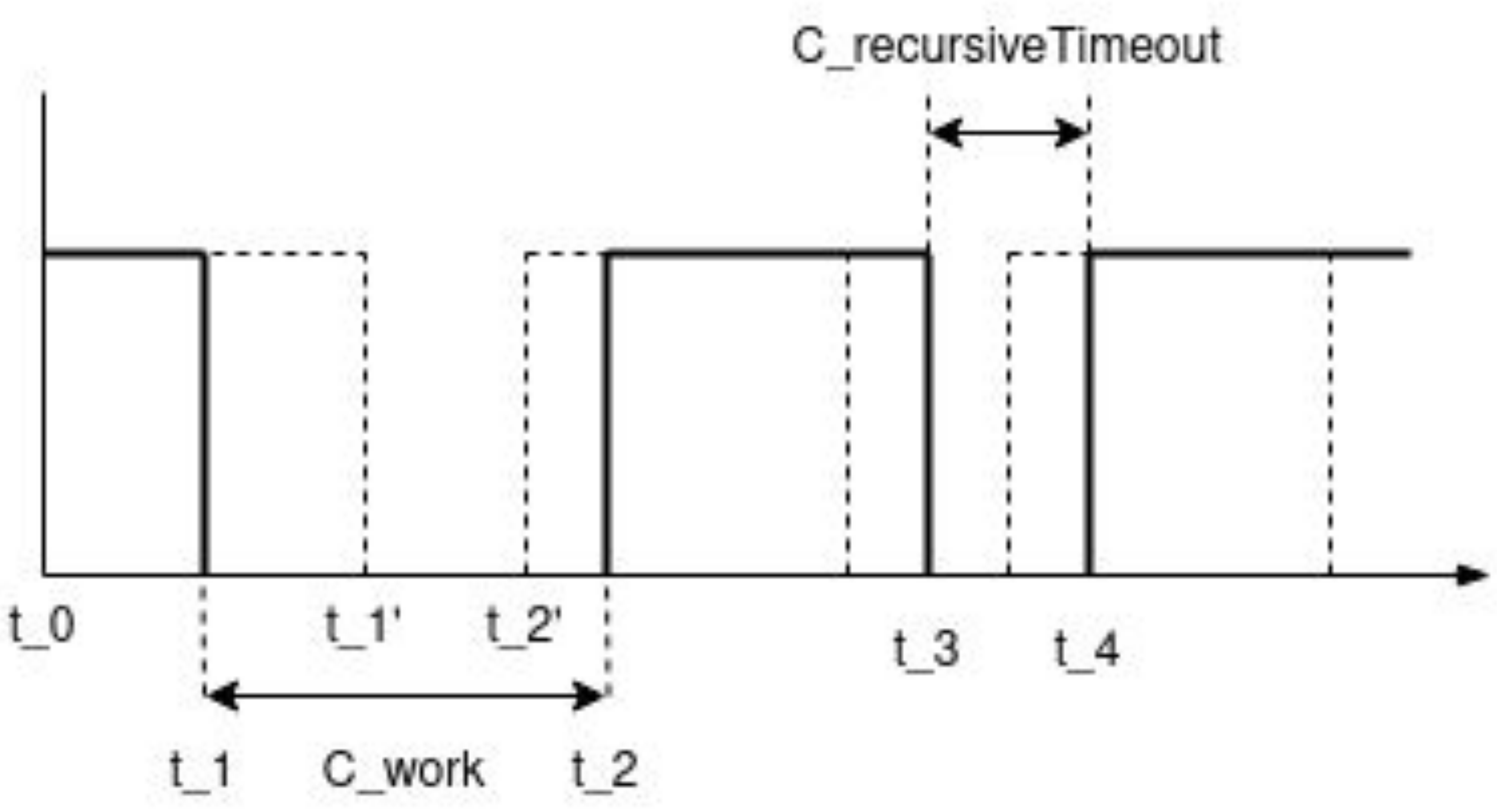
mojaloop foundation

Big Gap Update

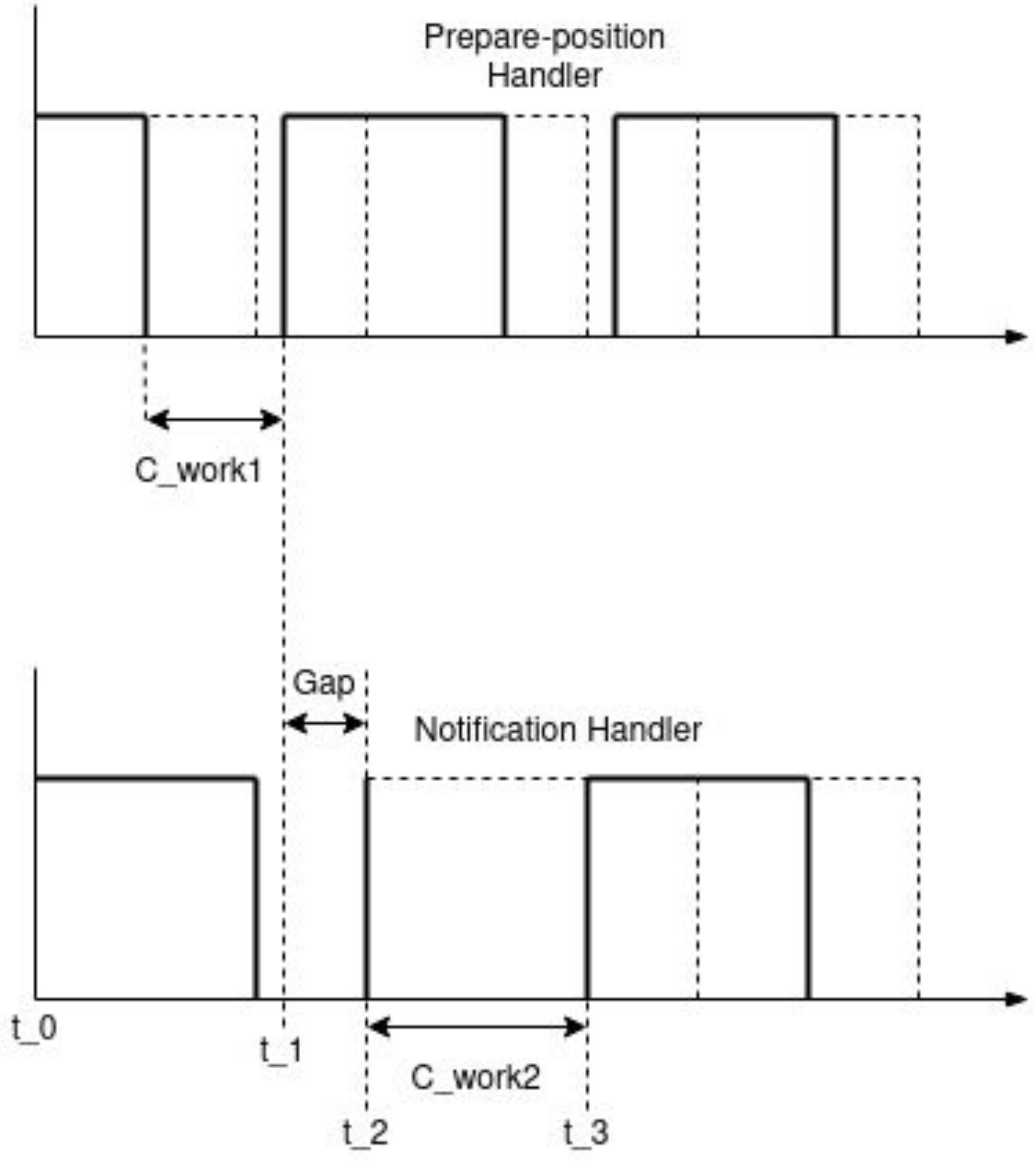


mojaloop foundation

Big Gap Update



Big Gap Update



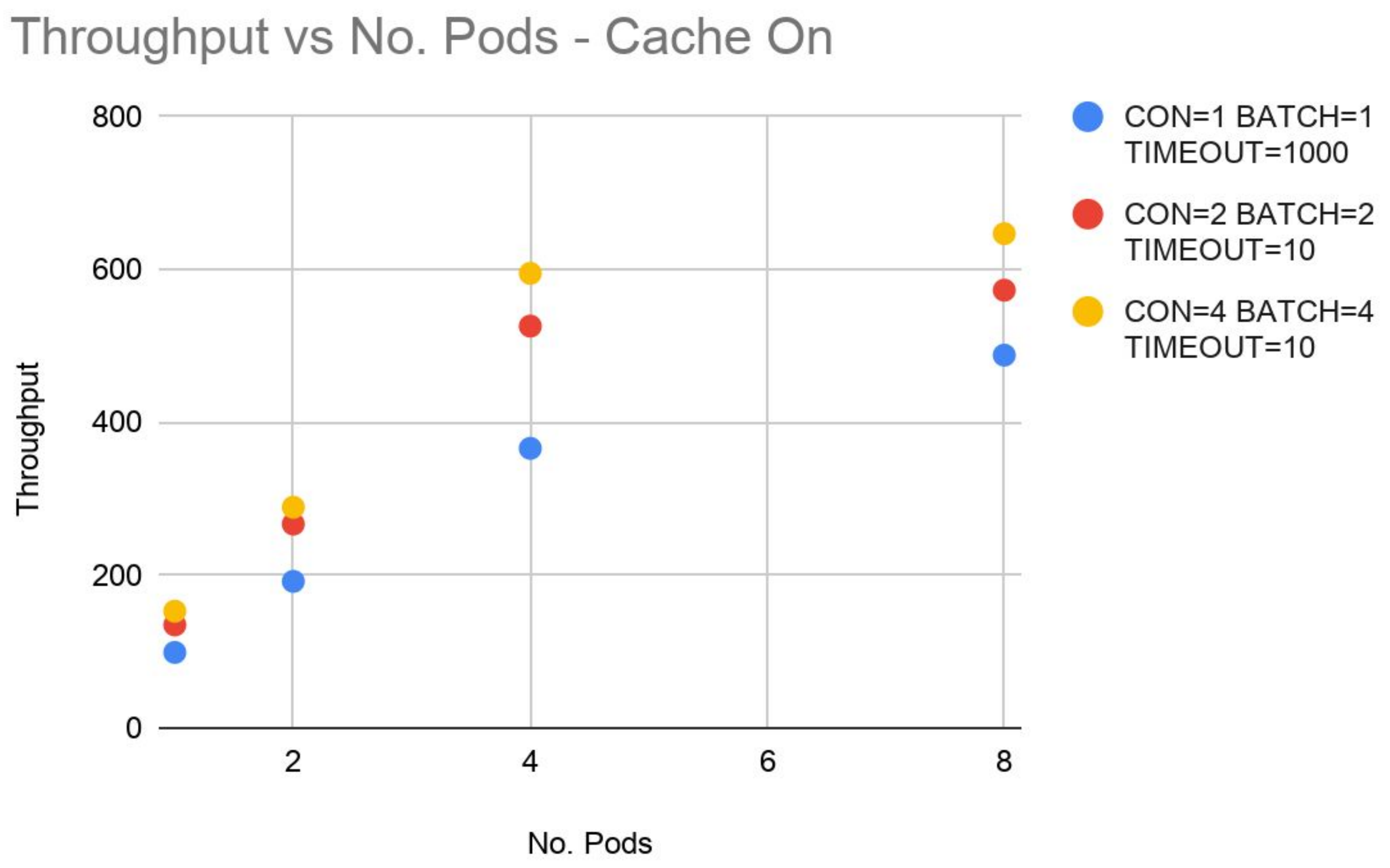
mojaloop foundation

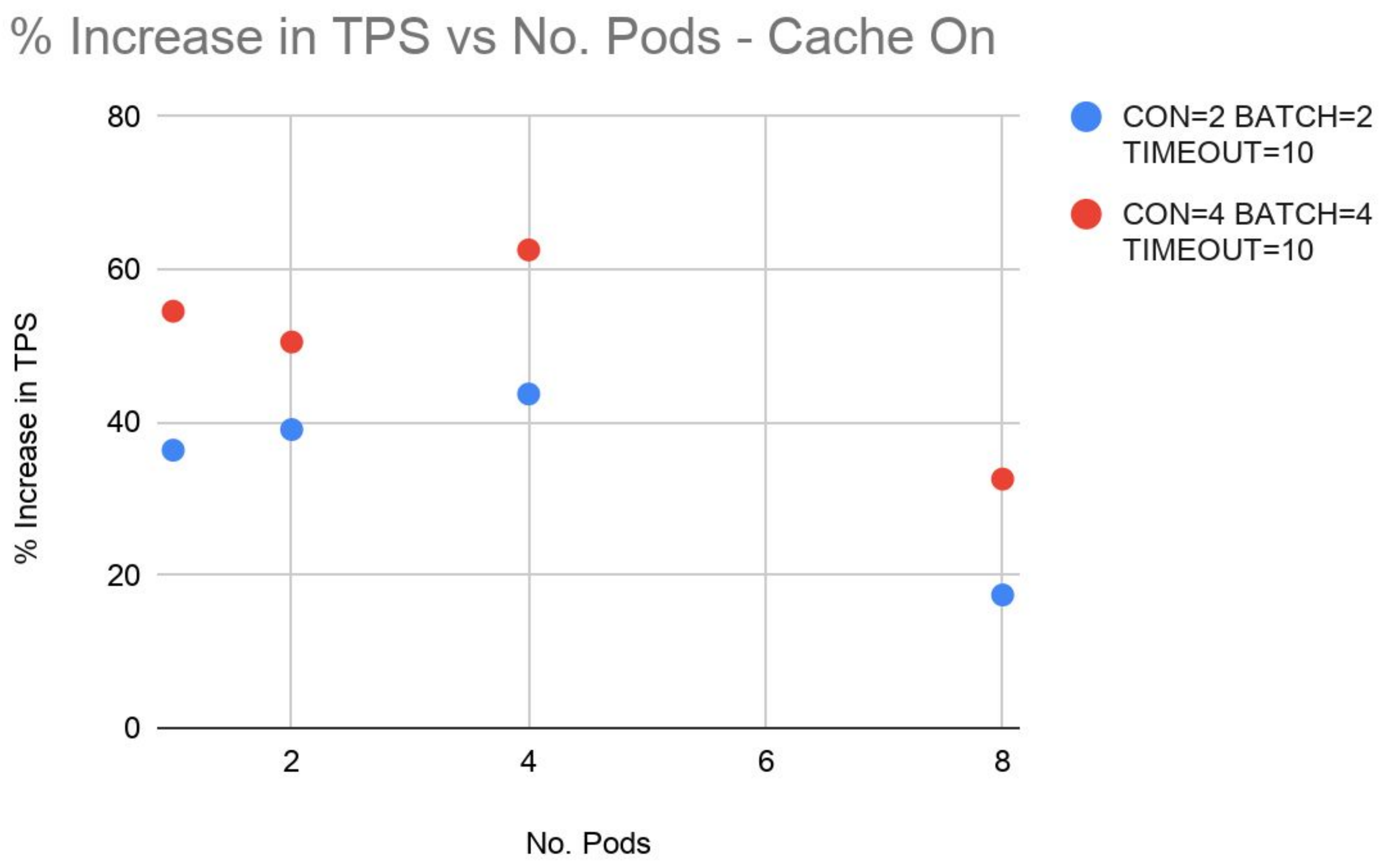
Insight: Batching

“The key to high throughput is message batching - waiting for a certain amount of messages to accumulate in the local queue before sending them off in one large message set or batch to the peer. This amortizes the messaging overhead and eliminates the adverse effect of the round trip time (rtt).”

https://github.com/edenhill/librdkafka/blob/master/INTRODUCTION.md#high-throughput

Concurrency





mojaloop foundation

Code here, data there

Prepare

insert into transferDuplicateCheck
    insert into ilpPacket
    insert into transferParticipant
    insert into transferParticipant
    insert into transfer
        insert into transferStateChange
            insert into participantPositionChange
            update participantPosition

Fulfill

insert into transferFulfilmentDuplicateCheck
    insert into transferFulfilment
        insert into transferStateChange
            insert into participantPositionChange
            update participantPosition

Insight: Batching

- 13ms for 1 transfer across 18 database queries
- What if we could do 4ms for 100 transfers all in 1 database query?

Crux

"Two-phase commit balance tracking is a simple problem, but a hard bottleneck to get around with relational databases, even in-memory databases or stored procedures"

Introducing TigerBeetle (early preview)

Purpose-built two-phase accounting database

Performance and Safety: Choose Two

TigerBeetle: More performance

- Amortize network and fsync costs by 100x with batching
- Balance tracking in the database
- Use redundancy to tolerate tail latency and mask gray failure

"The major availability breakdowns and performance anomalies we see in cloud environments tend to be caused by subtle underlying faults, i.e. gray failure rather than fail-stop failure."

"Disk and SSD RAIDs experience at least one slow drive 1.5% and 2.2% of the time."

TigerBeetle: Why more performance?

- Reduce deployment costs
- Increase availability
- **Increase community engagement!**

TigerBeetle: More safety

- Not just strict consistency and crash safety
- Detect disk corruption (3.45% per 32 months, PER DISK)
- Synchronous quorum replication to at least 4 of 6 nodes
- Prevent split-brain when failing over from primary to secondary
- Database migrations must not impact uptime

TigerBeetle: Developer friendly

All you have to do is:

1. send in a batch of prepares to TigerBeetle (in a single network hop)
2. send in a batch of fulfills to TigerBeetle (in a single network hop)

Show us the numbers

MySQL Mojaloop Cluster (our baseline)		76 FTPS
TigerBeetle (early preview)		**200 000 FTPS**
ml-api-adapter + TigerBeetle		880 FTPS
fast-ml-api-adapter + TigerBeetle		1 757 FTPS

Show us the architecture

LMAX + Quorum Replication = Performance + Safety

We are actively developing TigerBeetle.

Visit TigerBeetle on Discourse to receive updates:
https://community.mojaloop.io/t/performance-engineering/53

mojaloop foundation

Scalability PoC update

Motivation

A system where the **deployment unit is as cheap as possible**

and the whole solution **scales horizontally**

with **minimal cost** increments

Objectives

Design and build the central-ledger using patterns such as Event-Driven,

Event-Sourcing and CQRS to deliver better scalability with less cost

Have a plan to replicate the learnings to the rest of the platform

Smart Objectives

1. Min Deploy: 1,000 FTPS, 99% < 1 sec, 1 hour durability, on minimum hardware footprint (required minimum HW footprint to be defined by the PoC)

2. Sub-linear cost to scale for each one unit increment of capacity. The hardware cost of one increment is significantly less than the minimum deployment footprint. The architecture isolates the high-load components to scale separately from the low load ones.

3. Demonstrated Scaled Perf: nearly 5,000 FTPS, 99% < 1 sec, 1 hour durability for 5 scale units, plus min deploy unit. Proving point (b)

4. Understand how much the PoC design scales, max scaling before 99% < 1 sec fails or error rate increases (optional)

Milestones

1st - Prove scalability according to guiding principles

2nd - Introduce Event-Sourcing and CQRS

What we did

Split the transfers and participants and designed the transaction

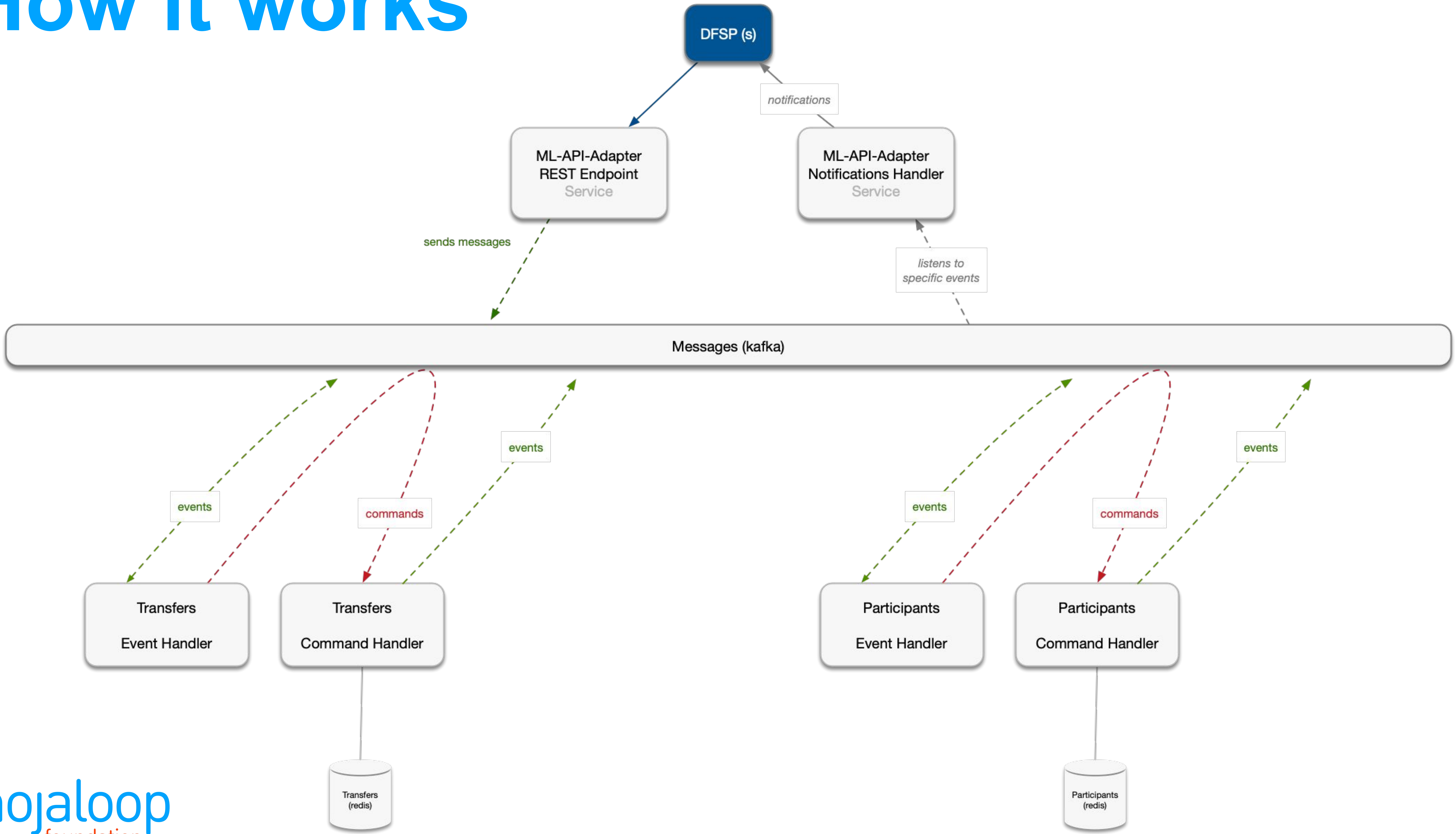
Built a distributed event-based transaction system based on the single

responsibility principle - split transfers and participants

Event-sourcing and CQRS - in progress

mojaloop foundation

How it works



mojaloop foundation

How it works

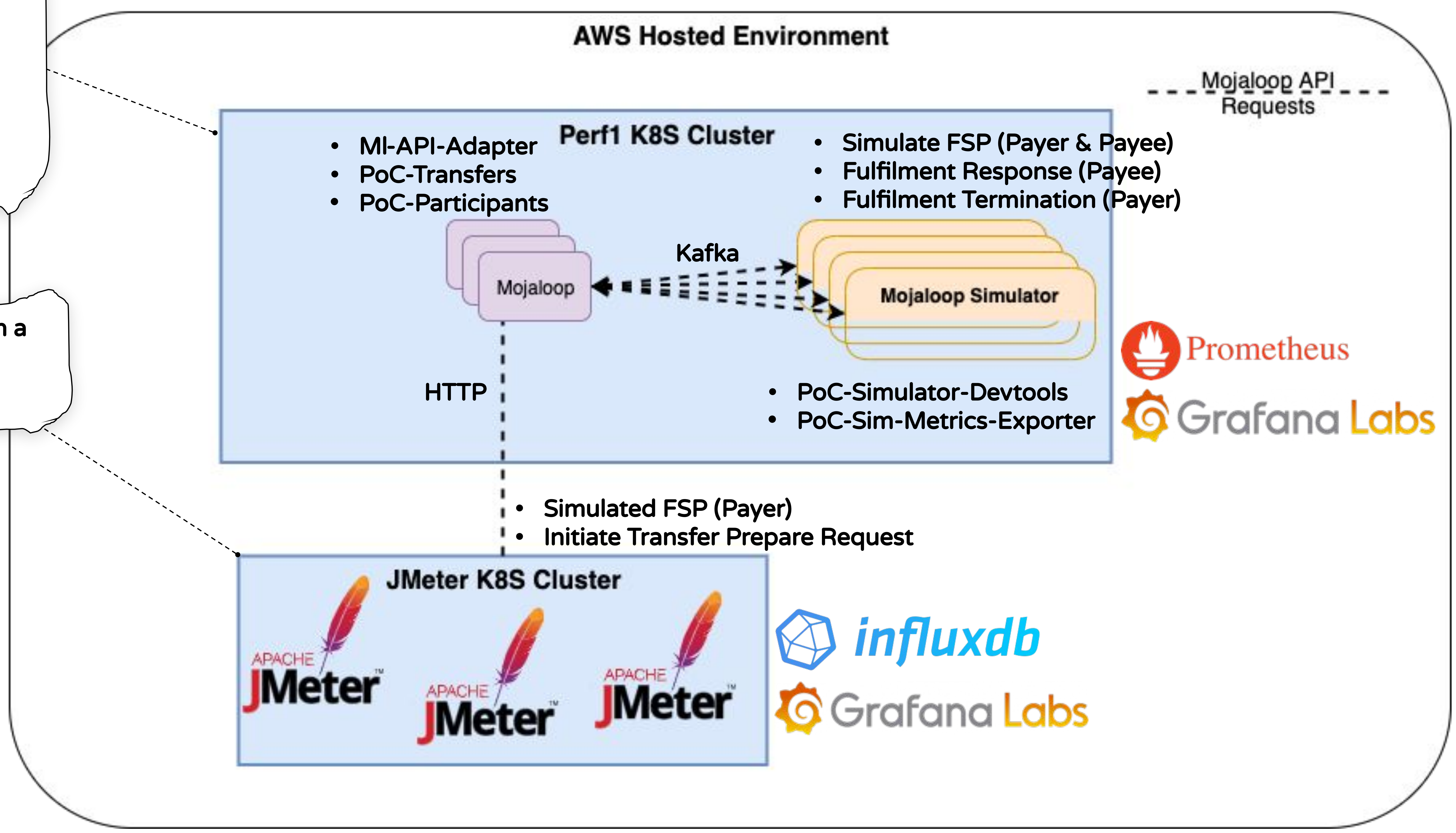
mojaloop foundation

Performance Environment Overview

Perf1 Environment

Scalable Kubernetes Cluster
1. Partitioned nodes for Redis, Kafka, Mojaloop, etc
2. NVME Storage for high disk IOP (Redis)
3. HA/Replication not active

Scalable JMeter cluster with a Master node and slaves



Environment
- Kubernetes Version: v1.11.6
- Rancher: v2.1.6
- Docker Version: 17.03.2-ce
- Kernel Version: 4.4.0-1052-aws
- Operating System: Ubuntu 16.04.4 LTS

Performance Environment Runtime Setup

Components	HW Types
ML API Adapter: API Service	c5.2xlarge
Simulators	
Transfer Handlers	c5.xlarge
Participant Handlers	
Metric Exporter	
Redis	i3.xlarge
Kafka	r5.xlarge
Zookeeper	

Exported Metrics
(operations per second + processing time):
moja_tx_transfer: End-to-end (Prepare + Fulfil)
moja_tx_transfer_prepare: Prepare only
moja_tx_transfer_fulfil: Fulfil only
moja_transfer_handler_evt: Transfer Evt Handler
moja_transfer_handler_cmd: Transfer Cmd Handler
moja_participant_handler_evt: Participant Evt Handler
moja_participant_handler_cmd: Participant Cmd Handler
moja_simulator_handler_evt: Simulator Evt Handler

Performance Run 1: 8 FSPs

% > 1 Sec: 0.03%

FSPs: 8

PoC Containers	Count
poc-transfers	24
poc-simulator-dev-tool	24
poc-participants	8
poc-ml-api-adapter-service	8
poc-simulator-metrics-exporter	6
poc-kafka-exporter	1
Total	**71**

Components	HW Nodes
ML API Adapter: API Service	1
Transfer Handlers	6
Participant Handlers	
Simulators	1
Metric Exporter	1
Redis	1
Kafka (5)	5
Zookeeper (3)	
Total	**14**

#	Indicators	Target	Actuals
T1	Financial Transactions Per Second (fps)	>= 200 fps	**1002 fps**
T2	Time for performance run	>= 1 Hour sustained run	**1 Hour**
T3	% of transactions that took longer than a second	< 1%	**0.03%**

moja_tx_transfer:

Financial Transfers / Sec: 1002

Avg Response Time: 65ms

moja_tx_transfer_prepare:

Financial Prepares / sec: 1002

Avg Response Time: 34ms

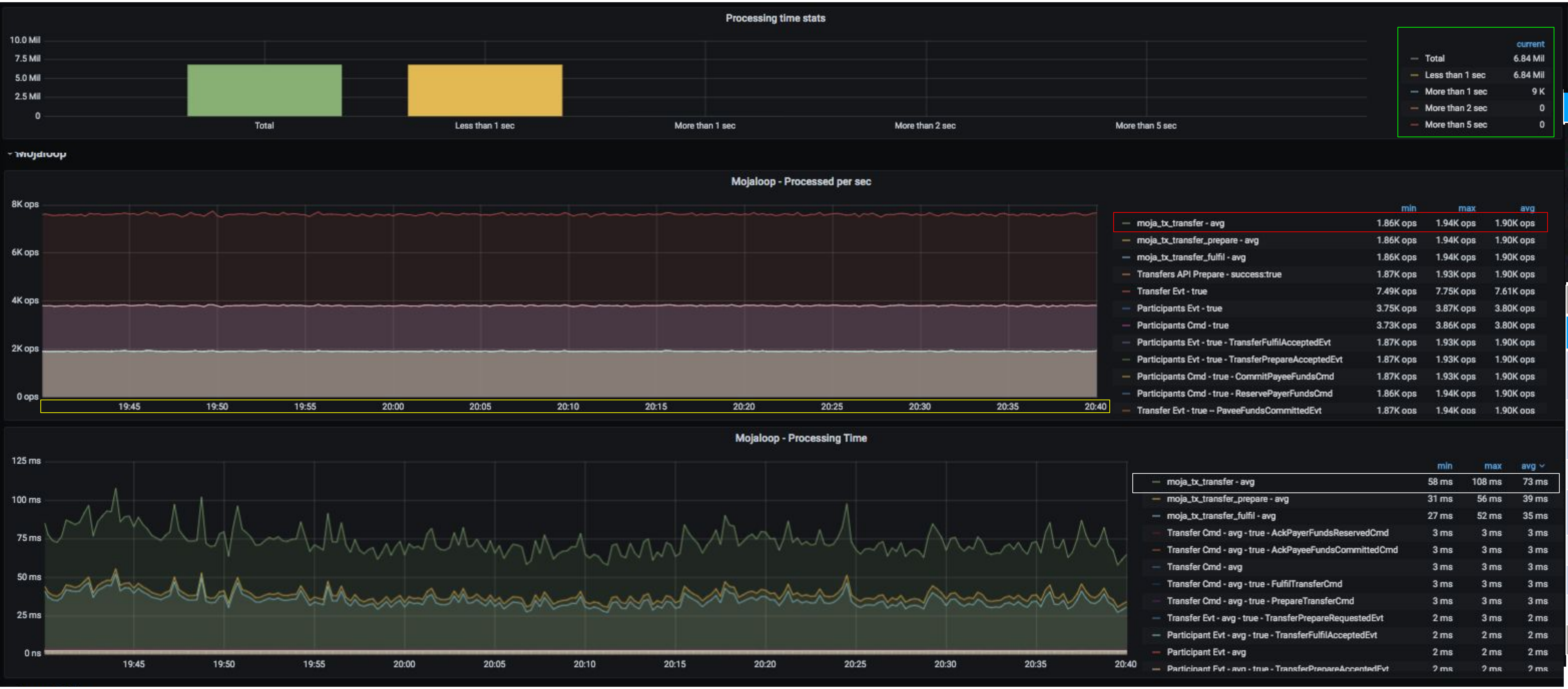
moja_tx_transfer_fulfil:

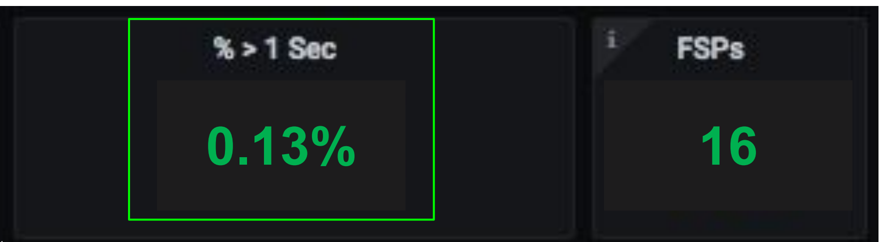
Financial Fulfils / sec: 1002

Avg Response Time: 31ms

mojaloop foundation

Performance Run 2: 16 FSPs





PoC Containers	Count
poc-transfers	48
poc-simulator-dev-tool	48
poc-ml-api-adapter-service	24
poc-participants	16
poc-simulator-metrics-exporter	12
poc-kafka-exporter	1
Total	**150**

Components	HW Nodes
ML API Adapter: API Service	2
Transfer Handlers	12
Participant Handlers	
Simulators	2
Metric Exporter	2
Redis	3
Kafka (9)	9
Zookeeper (3)	
Total	**28**

#	Indicators	Target	Actuals
T1	Financial Transactions Per Second (fps)	>= 200 fps	**1900 fps**
T2	Time for performance run	>= 1 Hour sustained run	**1 Hour**
T3	% of transactions that took longer than a second	< 1%	**0.13%**

moja_tx_transfer: **moja_tx_transfer_prepare**: **moja_tx_transfer_fulfil**:

mojaloop foundation

What we achieved against objectives

1. Min Deploy: 1,000 FTPS, 99% < 1 sec, 1 hour durability, on minimum hardware footprint (required minimum HW footprint to be defined by the PoC) - **Achieved**

2. Sub-linear cost to scale for each one unit increment of capacity. The hardware cost of one increment is significantly less than the minimum deployment footprint. The architecture isolates the high-load components to scale separately from the low load ones. - **Achieved**

3. Demonstrated Scaled Perf: nearly 5,000 FTPS, 99% < 1 sec, 1 hour durability for 5 scale units, plus min deploy unit. Proving point (b) - **We are on our way to get to the 5000k**

4. Understand how much the PoC design scales, max scaling before 99% < 1 sec fails or error rate increases (optional) - **This is a moving target, we keep getting more**

Scalability

We are close to linear

Theoretical limit is the participants -> we can do ~240 *ops / sec / participant -> 120 **FTP / sec / participant

- 4x DFSPS -> ~480 FTP / sec
- 8x DFSPS -> ~960 FTP / sec
- 16x DFSPS -> ~1920 FTP / sec
- 32x DFSPS -> ~3840 FTP / sec
- 42x DFSPS -> ~5040 FTP / sec

To achieve even more performance per participant:

1. Event-sourcing (Redis dependency will be drastically reduced)
2. Virtual positions (sharding the position through different participant handlers)
3. Batching by FSP (workloads are already in an appropriate FSP bucket - i.e. partitions)
4. Provision hardware with faster CPUs (scaling up to achieve a higher throughput)

Notes:
* Based on c5.xlarge
** Financial Transactions per Second

Performance

We got better performance, i.e., lower latency

65 ms - 8x DFSPs - 1000 FTPS

73 ms - 16x DFSPs - 1900 FTPS

?? ms - 32x DFSPs - ??00 FTPS

Results from 1 hr tests where requests longer than 1 sec were fewer than 1%

Hardware and cost perspective

8x FSPs - 1000 FTPS

14x machines

2,655 USD / monthly

base cost

16x FSPs - 1900 FTPS

28x machines

5,087 USD / monthly

1.9x base cost

Costs were calculated using on-demand - substantial savings can be gained by using a mix of reserved instances for the base capacity and on-demand/spot for elastic scaling

mojaloop foundation

8x FSPs - 1000 FTPS

Central Ledger

- 8x Participants + 24x Transfers - 6x c5.xlarge
- 8x ML-API-Adapter - 1x c5.2xlarge

Infrastructure

- 5 way Kafka cluster - 5x r5.xlarge
- Single Redis - 1x i3.xlarge

Others

- 24x Simulators (event based) - 1x c5.2xlarge

6	c5.xlarge
5	r5.xlarge
1	i3.xlarge
2	c5.2xlarge
14	**Total**

2,655 USD / monthly
(base cost)

On demand Ireland - https://calculator.aws/#/estimate?id=accce4e278dff81fd059271613f9cc219ac9bd8e

mojaloop foundation

16x FSPs - 1900 FTPS

Central Ledger

- 16x Participants + 48x Transfers - 12x c5.xlarge
- 24x ML-API-Adapter - 2x c5.2xlarge

Infrastructure

- 9 Way Kafka cluster - 9x r5.xlarge
- 3 Separate Single Redis - 3x i3.xlarge

Others

- 24x Simulators (event based) - 2x c5.2xlarge

12	c5.xlarge
9	r5.xlarge
3	i3.xlarge
4	c5.2xlarge
28	**Total**

5,087 USD / monthly
(1.9x base cost)

On demand Ireland - https://calculator.aws/#/estimate?id=719ac15bc14e9eaeaf0f59a3655e8d727ebcbeda

mojaloop foundation

Learnings

1. Architecture provides clear boundaries - each service does one thing (Single Responsibility Principle)
2. Typescript helps with code readability & maintainability
3. Lots of experience gained on the various node.js kafka clients (rdkafka, node-kafka and kafkajs) - confirmed rdkafka is the best option
4. Tried gzip compression, looks like it is working - more tests are required, might not need it everywhere
5. Monorepo helped so far with workflow, but leads to more complex pipelines - need to find the right balance in the future
6. Domain-driven Design (DDD) tactical patterns and code structure helps with maintainability
7. Manual partitioning for the participants ensures optimal distribution of load

What we can do better

1. Separate participants even further with virtual positions
2. Separate participants metadata from position
3. Optimize hardware usage
4. Optimize duplicate stores to use less storage

Conclusion

1. Better horizontal scaling by segregating the different functions across different services and datastores
2. Separate transfers and participants lead to easier partition of datastores
3. Better maintainability / extensibility
4. This design shifts the burden from a shared DB to kafka, which is horizontally scalable by design
5. Reduced latency and faster response times
6. Retrofitting is simple and a migration/bootstrapping of data is feasible

Next steps - option 1

Incorporate learnings to current central-ledger

Pros:
- We can cherry pick the minimum change
- Build and deploy pipelines setup already

Cons:
- Unlikely to deliver the full benefit
- May end-up being a mix of standard Javascript & Typescript
- Misses the opportunity to simplify the code
- To incorporate a significant part of learnings it might take longer than a rewrite (the other options)
- Depending on which design principles to apply, a large refactor of code might be required (risky)

mojaloop foundation

Next steps - option 2

Make the PoC production ready and replace the current one

Simple event-driven + CQRS

Pros:
- Single Responsibility Principle - separation between transfers state and participants position updates delivers increased the required scalability
- CQRS read / write separation means that we can have many views as we want without performance hit
- Typescript provides strong types and improved code quality
- DDD code structure provides abstractions and mechanisms to reduce coupling with infrastructure and evolve the product and without rewrites
- Granular events enable future async integration of settlements, auditing, fraud detection mechanisms or other real-time/manual decisions

Cons:
- Distributed transaction architecture is more complex than a DB centric transaction
- Requires better monitoring tools to fully understand how the system is behaving
- Still updates a local store and the queue, not a single atomic write that would provides additional scalability and performance (option 3 solves this)

mojaloop foundation

Next steps - option 3

Make the PoC production ready and replace the current one

Event-Sourcing + CQRS

Pros:
- Same as option 2
- Single atomic write provides more scalability and performance
- Effortless passive in-sync geo replication - mirror maker replicates only the kafka topics that contain state/snapshot messages, and the readside CQRS handlers update the read side from those events

Cons:
- Distributed transaction architecture is more complex than a DB centric transaction
- Requires better monitoring tools to fully understand how the system is behaving
- Additional complexity than option 2, due to loading from events + snapshots and keeping offsets

Next steps - Recommendation

Go for option 3 during PI 11

Event-Sourcing + CQRS

We need to finish testing the event-sourcing code,
if we don’t get (or care about) the additional benefits,
we can fallback to option 2

Next steps - Plan

Properly test CQRS and ES and execute load tests

Write a CQRS handler to keep necessary DB Data up to date

Test scaling to 32 or more FSPs (or up to 5000 FTPS)

Make it production ready

Create the bootstrapping/migration tool

Next steps - What it takes

With what we have built already,

we think that we can make all this work during PI 11

If this goes forward we need to work on a detailed plan with resources

Team work

Amar Ramachandran (Modusbox)

Donovan Changfoot (Coil)

Miguel de Barros (Modusbox)

Pedro Barreto (Crosslake)

Roman Pietrzak (Modusbox)

Sam Kummary (Modusbox)

Q&A

Continue the discussion at

https://community.mojaloop.io/t/performance-engineering/53

Code at
https://github.com/mojaloop/poc-architecture