# Coil’s Mojaloop Performance Work 2020

By Don Changfoot and Joran Dirk Greef

## Contents

## Test Harness

### Design - Two Sides Of The Same Coin

We designed and implemented a single-process test harness to function as both payer and payee, to initiate thousands of transfers into Mojaloop at varying levels of concurrency, receive the resulting prepare notifications and turn these around as fulfils back into Mojaloop, measuring end-to-end throughput as well as individual transfer latency (average and max), using a single non-distributed wall clock without clock skew.

### Benchmarking The Benchmark

We benchmarked our test harness on a null cipher “hello world” implementation server and were able to achieve more than 10,000 HTTP requests per second. We have confidence that our test harness will not impact throughput measurements while saturating Mojaloop.

## Baseline Throughput

### 76 TPS

With our test harness in place, we measured a baseline throughput on the order of 76 transfers per second (TPS) on our particular cluster deployment. This was using the combined handlers.

### Minimum Deployment

Our baseline number is significantly lower than the 300 TPS to 900 TPS range delivered on the ModusBox cluster. We were interested in measuring and optimizing a low-cost deployment of Mojaloop, and we intentionally set out to see what we could learn from a minimum deployment, for example running only one pod VM per handler. At the same time, and in consultation with ModusBox, we were careful to provide adequate resources for critical components such as the database, assigning each logical pod VM to its own dedicated hardware, to limit contention.

## Random Disk Seek Analysis

Our first hypothesis or hunch was that the biggest limit on throughput might be the cost of random disk seeks, and therefore our initial strategy was simply to measure the TPS difference between a cluster deployed on NVME versus a cluster deployed on spinning rust, to see if this was worth investigating further.

However, our 76 TPS baseline on NVME devices led us to conclude that the throughput issue was more serious than too many random disk seeks (though that may still be a serious issue for deployments of Mojaloop that cannot afford NVME devices). We abandoned our notion of deploying Mojaloop on HDD devices and decided to investigate the many incidental costs inherent in a distributed microservices design such as Mojaloop, by taking into account the non-zero cost of network latency, often assumed to be zero as one of the eight fallacies of distributed systems.

## Metric Analysis

### Underutilization

While establishing the baseline throughput for our minimum deployment cluster, we reviewed the existing metrics collected by Prometheus and displayed in Grafana. When optimizing a system, it is often the case that one must reduce CPU usage, or the number of network packets, or the number of disk seeks etc. **Usually, a system is made more efficient by reducing the load on physical resources so as to achieve more throughput. However, in the case of Mojaloop, what struck us was the absence of utilization of the underlying hardware.** The CPU usage across all pod VMs was low (on the order of 2%), network utilization was low (a few KB/s), and MySQL query latency was low (no slow queries). Nothing seemed to be struggling. As an anecdote, we even received a friendly dashboard warning in GCP that, according to Google, our cluster was not being utilized past a few percent and that perhaps we should downsize.

### Increase Throughput By Increasing Load

Rather than ask how we can use less CPU, the metrics led us to ask how can we use more CPU? Rather than look at optimizing CPU hotspots such as native bindings etc. we decided first to look at optimizing idle time, points in the software where the system is waiting on something asynchronous. This could be a call to *setTimeout()* or a network call, or a lack of concurrency i.e. doing things one after the other where we could be doing them concurrently to increase load. It's an interesting problem: to increase throughput by increasing load.

# Network Latency Analysis

## DNS Queries

### From DNS to DoS

Node.js has a little-known, longstanding and nasty bug where calls to Node’s *dns.lookup()* can **stall all subsequent DNS queries and all filesystem IO**. The reason is that *dns.lookup()* is implemented by calling *getaddrinfo()* by default, which is a blocking system call into the kernel and which must therefore be run in Node’s libuv thread pool in order not to block Node’s event loop. The trouble is that Node’s libuv thread pool has a default *UV_THREADPOOL_SIZE* of only 4 threads. This means that 4 slow DNS queries are enough to have a dramatic denial of service impact on the system as a whole. For a microservices architecture, this would indeed manifest itself in low throughput and low utilization.

### No DNS Queries In The Critical Path

We monkey-patched Node's *dns.lookup()* to capture the latency and endpoints of all DNS queries made by Mojaloop’s central ledger. Thankfully, this was a negative result and we found that there were no DNS queries in the critical path of a transfer from start to finish. Our search for the cause of low utilization would have to look elsewhere...

## SQL Queries

### Knex

Mojaloop uses a helper wrapper for MySQL called Knex. Knex describes itself as *a "batteries included" SQL query builder for Postgres, MSSQL, MySQL, MariaDB, SQLite3, Oracle, and Amazon Redshift designed to be flexible, portable, and fun to use*.

ORMs and query builders are known to increase the risk of object-relational impedance mismatch, for example by mapping a "find one" method to a "find all" method returning an array and then taking the first result.

### 18 Queries Per Transfer

Because of the presence of Knex, and because we wanted a better grasp on the critical business logic, instead of reading the Mojaloop source to piece the queries together, **we decided to cast a wide net** by instrumenting the Knex MySQL binding to capture all actual SQL queries sent over the network in the course of processing a single transfer:

*1 Transfer = 11 Inserts + 2 Updates + 6 Selects = 18 SQL Queries*

A single transfer results in a **network amplification factor of 18 queries**, and a **write amplification factor of 13 atomic database transactions**, all requiring fsync(), an extremely expensive system call.

### Eliminating Insert Queries

We found at least two insert queries that can be eliminated:

1. *insert into `ilpPacket` (`transferId`, `value`)* inserts a single value that can be moved to *insert into `transfer` (`amount`, `currencyId`, `expirationDate`, `ilpCondition`, `transferId`)* to combine two inserts in one.

2. *insert into `transferDuplicateCheck` (`hash`, `transferId`)* is a CAS-style insert-or-fail query used for duplicate checking that can similarly be eliminated by moving the hash value to the *transfer* table and by adding a bloom filter to shield the database from 99% of duplicate checks. This need not have an impact on cluster startup latency since the bloom filter would only need the last few hours of transfer hashes to initialize, assuming that the majority of duplicates have temporal proximity.

## Fsync Bottleneck Analysis

### Back-Of-The-Envelope Calculation

Since our cluster uses a single NVME device underneath MySQL, and since the average latency of fsync() on high-end NVME devices is between 0.14ms to 3.8ms, a back-of-the-envelope calculation would imply an fsync() constraint on transfer throughput (TPS) that is dependent on the model of NVME device used:

*Intel PC-3700 = 1000ms / (0.14ms fsync() * 13 writes) = 549 transfers per second*
*Intel 750 = 1000ms / (0.49ms fsync() * 13 writes) = 156 transfers per second*
*Intel PC-3100  = 1000ms / (0.79ms fsync() * 13 writes) = 97 transfers per second*
*Samsung SSD 960 PRO = 1000ms / (3.8ms fsync() * 13 writes) = 20 transfers per second*

### 156 TPS Service Ceiling Limit

**N.B. For our particular cluster, we therefore expect a service ceiling of between 156 to 549 transfers per second, necessarily imposed by Mojaloop's write amplification factor of 13 database transactions per transfer.** This fsync() bottleneck may prove to be the first or second linear constraint we bump our heads into, and without due consideration may mask any performance gains from other optimizations. In other words, this fsync() bottleneck would need to be addressed by reducing write amplification first, before the performance benefit of other optimizations can be fully assessed.

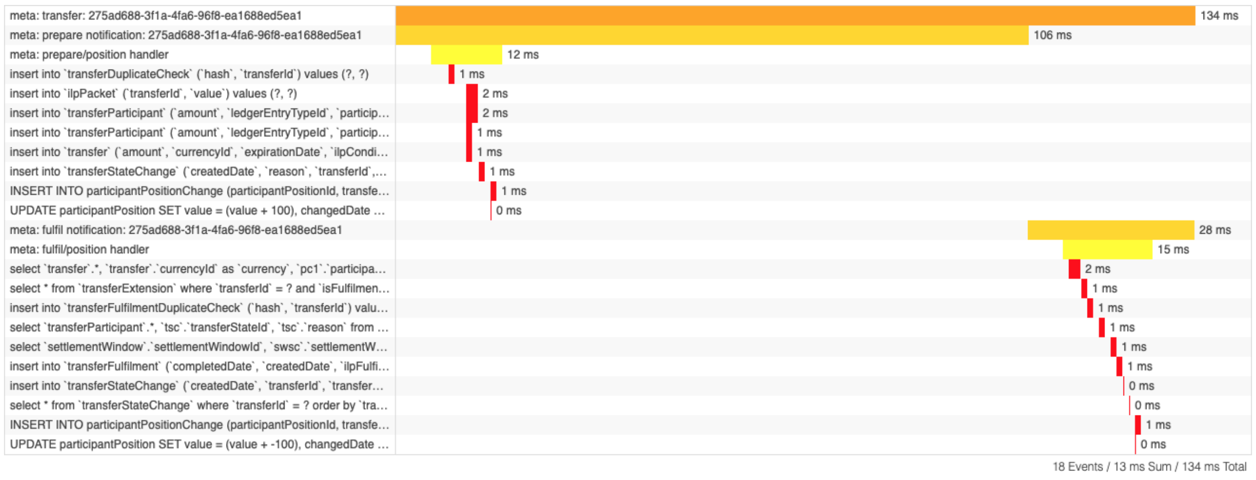
# Waterfall Analysis

## Waterfall Charts

While Mojaloop supports tracing and logging, we still needed a way to “see” the causal dependence across the SQL queries we had captured, so that we could further investigate the effect of network latency (and the distributed microservices design) on transfer throughput. Specifically, we wanted to be able to add the latencies of events not overshadowed by longer concurrent events to arrive at a true cost of network latency. A waterfall chart can make hidden problems visually obvious, and to this end we created and open-sourced *@donchangfoot/waterfall*: [https://github.com/DonChangfoot/waterfall](https://github.com/DonChangfoot/waterfall)

## Observations

To avoid misrepresenting a single transfer, we created 100 different waterfalls for 100 different transfers, all spaced one second apart to eliminate any differences due to variance. We found that all 100 waterfalls were almost exactly consistent in terms of timing so that we could have some confidence that we had a clear picture of a single transfer not distorted by variance:



```
meta: transfer: 275ad688-3f1a-4fa6-96f8-ea1688ed5ea1                              134 ms
meta: prepare notification: 275ad688-3f1a-4fa6-96f8-ea1688ed5ea1                  106 ms
meta: prepare/position handler                                                    12 ms
insert into `transferDuplicateCheck` (`hash`, `transferId`) values (?, ?)         1 ms
insert into `ilpPacket` (`transferId`, `value`) values (?, ?)                     2 ms
insert into `transferParticipant` (`amount`, `ledgerEntryTypeId`, `particip…      2 ms
insert into `transferParticipant` (`amount`, `ledgerEntryTypeId`, `particip…      1 ms
insert into `transfer` (`amount`, `currencyId`, `expirationDate`, `ilpCondi…      1 ms
insert into `transferStateChange` (`createdDate`, `reason`, `transferId`,…        1 ms
INSERT INTO participantPositionChange (participantPositionId, transfe…            1 ms
UPDATE participantPosition SET value = (value + 100), changedDate …               0 ms
meta: fulfil notification: 275ad688-3f1a-4fa6-96f8-ea1688ed5ea1                   28 ms
meta: fulfil/position handler                                                     15 ms
select `transfer`.*, `transfer`.`currencyId` as `currency`, `pc1`.`participa…     2 ms
select * from `transferExtension` where `transferId` = ? and `isFulfilmen…        1 ms
insert into `transferFulfilmentDuplicateCheck` (`hash`, `transferId`) valu…       1 ms
select `transferParticipant`.*, `tsc`.`transferStateId`, `tsc`.`reason` from …    1 ms
select `settlementWindow`.`settlementWindowId`, `swsc`.`settlementW…              1 ms
insert into `transferFulfilment` (`completedDate`, `createdDate`, `ilpFulfi…      1 ms
insert into `transferStateChange` (`createdDate`, `transferId`, `transfer…        0 ms
select * from `transferStateChange` where `transferId` = ? order by `tra…         0 ms
INSERT INTO participantPositionChange (participantPositionId, transfe…            1 ms
UPDATE participantPosition SET value = (value + -100), changedDate …              0 ms
                                                 18 Events / 13 ms Sum / 134 ms Total
```

Visualizing a single transfer as a waterfall chart revealed two striking data points, namely a relationship between SQL queries and throughput, and a “Big Gap” between critical work sections.

## Relationship Between SQL Queries And Throughput

We saw that the cumulative sum of network and database latency across all SQL queries for a single transfer (the red bars) was 13ms. This latency was measured after a warmup and without any concurrent load.

By accident or by habit, we did a back-of-the-envelope calculation comparing the cumulative latency of the SQL queries for a single transfer with throughput. We did not expect to see much relationship between the cumulative latency of the SQL queries for a single transfer and throughput. However, what we found surprised us:

*1000ms / 13ms cumulative latency of the SQL queries for a single transfer = 76 TPS*

**In other words, the baseline throughput of the entire cluster under concurrent load already established as 76 TPS, was EXACTLY the same as the throughput estimate derived from only the critical SQL work performed for a single transfer.**

We had expected the cluster throughput under concurrent load to be higher for several reasons:

1. The database can handle concurrent load.
2. The network can handle concurrent load. At least half of the 13ms cumulative latency of the SQL queries for a single transfer was probably network latency (with around 6ms of database latency spent mostly on fsync i.e. 0.49ms fsync() * 13 writes = 6ms).
3. We assumed that Mojaloop would process transfers concurrently.

We revisit this last assumption in our Concurrency Analysis.

## A “Big Gap” Between Critical Work Sections

Our second takeaway from the waterfall for a single transfer was that the critical work for a transfer could be divided into two sections, the prepare handler and the fulfil handler. However, the cumulative latency for these two critical sections was only 13ms, compared to total end-to-end transfer latency of 134ms, an order of magnitude greater than the critical work.

We were fairly confident that:

1. The network latency between the Mojaloop cluster and our test harness was on the order of 1ms to 3ms at most and that our test harness was not lagging in turning prepare notifications around as fulfil requests.
2. We were not suffering from clock skew, since our external test harness clock was a single wall clock, and since our internal central ledger clock was also a single wall clock.
3. Any reduction in this “Big Gap” would improve latency but not throughput since we had already established an exact relationship between throughput and the 13ms cumulative latency of the SQL queries (or critical work) for a single transfer.

After sharing our initial result with the Mojaloop community, we decided to investigate further.

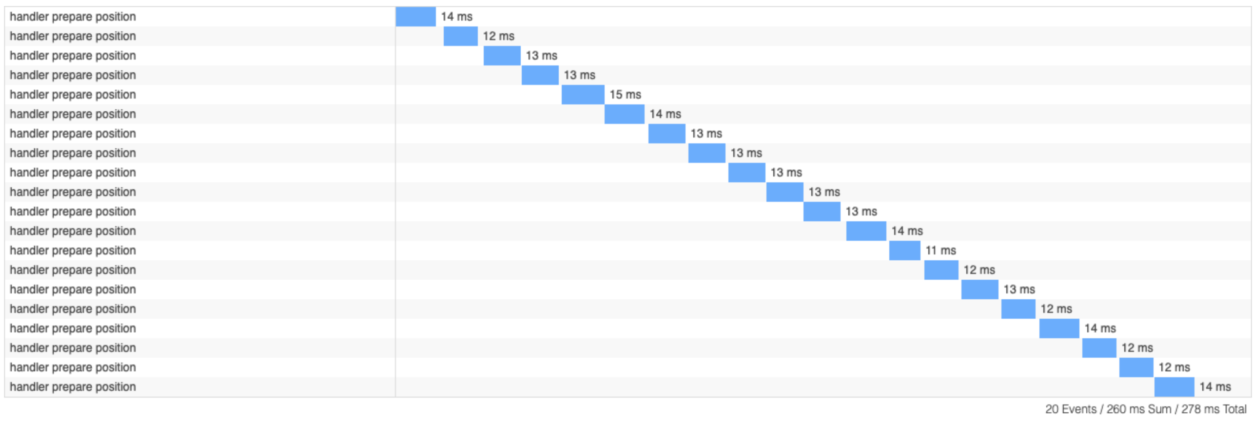
## Concurrency Analysis

### The C10K Problem

Our Waterfall Analysis revealed an exact relationship between the critical SQL work of a single transfer and throughput, which should not be the case, unless transfers are processed sequentially by handlers. However, Node.js was designed to solve the C10K problem and is able to process thousands of asynchronous events concurrently for high throughput. Failing to exploit concurrency would certainly contribute to the underutilization and low throughput that we were seeing. We decided to investigate this forced serialization hypothesis.

### Testing The Forced Serialization Hypothesis (Concurrency=1)

We reran our load test to generate 20 concurrent transfers and plotted the resulting prepare/position handler start and end times as a waterfall. This revealed that each Mojaloop handler was indeed forcing transfers to be processed one at a time within the handler, by requesting only one message from Kafka, then processing this message, before going back to Kafka for another message, thereby wasting the Node.js process and associated libuv threadpool, since most of the critical work in processing a message represents asynchronous network requests to the MySQL database:



## Fixing Concurrency

### Processing Multiple Transfers Concurrently Within A Handler

We found that *central-services-stream/src/kafka/consumer.js* already made provision for processing messages using flow control with different levels of concurrency (with the default being no concurrency). We therefore needed only to find a way to consume multiple messages at a time from Kafka.

### Consuming Multiple Messages At A Time From Kafka

The librdkafka library makes a critical throughput tuning recommendation:

*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).*

While Mojaloop exposed a *batchSize* config option per handler, to configure the number of messages (not bytes) returned per poll to Kafka, we found two bugs in *central-services-stream/src/kafka/consumer.js* when increasing *batchSize*.

#### Unbounded Poller Instantiations

Although each message received from Kafka is passed to a synchronous queue called _*syncQueue*, an instance of the async library's queue object, every single processed message results in a call to _*consumeRecursive()*, which in turn launches a never-ending poll against Kafka that will make a network request to Kafka every 1100ms (*consumeTimeout=1000ms* + *recursiveTimeout=100ms*).

When *batchSize* is 1, this is not a problem, but when *batchSize* is greater than 1, an additional poll against Kafka is launched **for every transfer** flowing through Mojaloop. After several thousand transfers, Kafka would be brought to its knees by the sheer number of pollers.

**We submitted a fix to add a guard within _*consumeRecursive()* to launch a poll against Kafka only if no other polls exist, no matter how many times _*consumeRecursive()* is called or from where.**

## Four Second Latency Spike

We found another interesting case where sending 20 concurrent transfers into Mojaloop with a *batchSize* of 8 would result in a four second latency spike for the last 4 transfers:

```
6adb4ae6-ca05-45e7-8f9f-9a8e1fe67ed0   300ms
7d9bcddc-e52b-4fae-8261-44ce4e22adb3   421ms
fc059148-6408-4c02-9e98-095bf6adb961   433ms
0574caf4-b5c3-4217-8243-9f160cf576aa   438ms
79d6f28c-282d-4cb9-bc2f-2c314db52b2f   441ms
7bd9f47f-16bb-4790-9c27-2a46882086e7   445ms
c6b35825-074c-4aee-baeb-c8f43f1da9c6   448ms
775d4dea-93f9-401e-882c-9234f4d79f2b   449ms
92d1d88a-1452-47f1-aadc-1ad35de3cbc7   583ms
b32b7aec-724c-4047-9cdd-f0389ef7953f   589ms
340422c2-957b-4b6f-9ecd-75d20ed5d227   592ms
433e6e92-9736-4d9a-af64-13a9d4e06817   593ms
be16e1d4-fd27-4fe2-b872-f2e9e7c2b4db   599ms
a6ee1d6e-71d2-4f1e-8886-50ee2887f42b   600ms
79f54796-150b-42f2-8037-2fd34f0b5b5a   607ms
444c8ef8-03af-4c03-bb95-28a0187370cf   608ms
0ca93c5e-ceba-460e-9447-59c2816b720b  4254ms
22410a91-4d13-45ca-97a4-610262624bf4  4257ms
24a8fdf2-bc80-400c-aa02-c3f2c7c80992  4262ms
4d7a826e-a097-4ba0-8f40-47da0fdfa6fe  4266ms
```

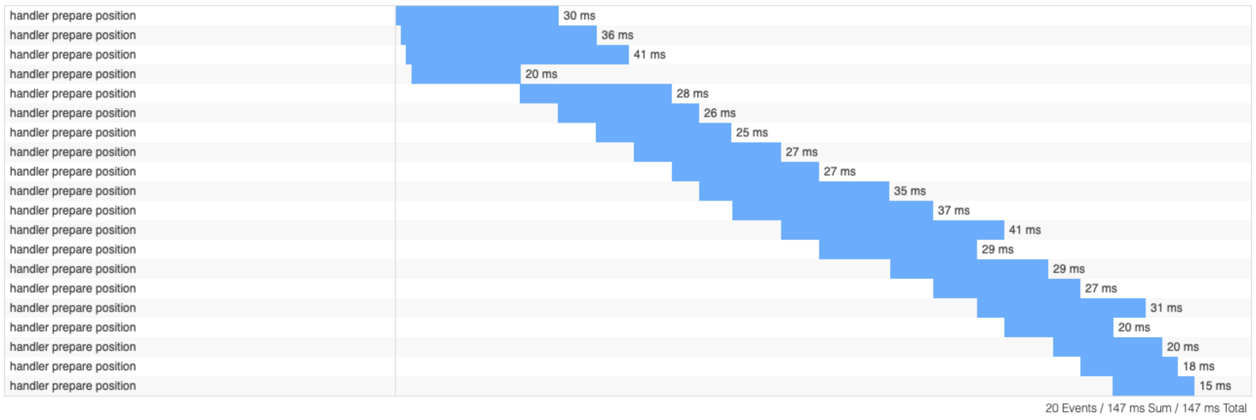
**In fact, we were able to generalize and reproduce this four second latency spike for any test where the number of concurrent transfers into Mojaloop was not a multiple of the *batchSize* of the number of messages consumed at a time from Kafka.**

The reason for this surprising behavior was a little-used config variable called *consumeTimout*, also used by *central-services-stream/src/kafka/consumer.js*, which was by default set to 1000ms i.e. one second. This *consumeTimeout* is passed to Kafka and instructs Kafka to wait at most one second if it has less than *batchSize* messages available to consume. In other words, if the consumer asks for four messages, and Kafka only has three available, then Kafka will wait a full second before returning whatever messages it has. We found this one second delay was encountered by both the prepare and fulfil handlers as well as both notification handlers, resulting in a four second latency spike across a transfer.

**We submitted a fix to reduce the default consumeTimeout latency from 1000ms to 50ms and we propose that Mojaloop implement config validation in future to warn operators about misconfigured config options.**

## Testing Concurrency Throughput Improvements (Concurrency=4)

After updating Mojaloop to process messages within handlers concurrently and then setting the concurrency level per handler to 4, we reran our load test to generate 20 concurrent transfers and plotted the resulting prepare/position handler start and end times as a waterfall:



The result confirmed that Mojaloop was now able to process messages concurrently, with messages within a handler now being processed 4 at a time. As soon as the first message finished processing (after 20ms), the next message was immediately processed, and so on.

## 50%-100% Throughput Improvement

The cumulative latency for the concurrent waterfall is 147ms in total, **almost half the cumulative latency for the baseline waterfall** of 278ms, representing a throughput of 136 TPS (1000ms / (147ms / 20 transfers)) compared to the baseline throughput of 71 TPS, and **resulting in a throughput improvement of 91%** ((136 - 71) / 71 * 100) for this test run.

## Kafka Network Latency Amortized (Bonus)

In addition to the above result, whereas the network latency to Kafka in between handlers was clearly contributing to the total latency of the baseline waterfall (278ms - 260ms = 18ms), **the network latency to Kafka is now eliminated**, since we were also able to optimize Mojaloop to fetch messages from Kafka in the background while still processing messages in the foreground, amortizing the network latency to Kafka. However, we recommend that this bonus optimization be disabled to ensure Kafka auto-commit safety, since Kafka may auto-commit the consumed log offset for messages still being processed as soon as the next batch of messages is requested.

## Finding The Concurrency Sweetspot

With our minimum deployment, we reran our load test to generate 1000 transfers, sending 100 transfers into Mojaloop concurrently, in order to find the concurrency/throughput sweetspot:

| Concurrency | Test Run | Batch size | Throughput | Latency Avg (ms) | Latency Max (ms) |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 69 | 1381 | 1562 |
| | 2 | 1 | 71 | 1341 | 1629 |
| | 3 | 1 | 73 | 1310 | 1525 |
| | | | | | |
| 4 | 1 | 4 | 133 | 718 | 985 |
| | 2 | 4 | 130 | 728 | 1041 |
| | 3 | 4 | 134 | 715 | 1041 |
| | | | | | |
| 8 | 1 | 8 | 139 | 685 | 983 |
| | 2 | 8 | 143 | 651 | 903 |
| | 3 | 8 | 146 | 654 | 881 |
| | | | | | |
| 16 | 1 | 16 | 131 | 738 | 1199 |
| | 2 | 16 | 112 | 848 | 1766 |
| | 3 | 16 | 115 | 840 | 1669 |
| | | | | | |
| 16 | 1 | 10 | 133 | 724 | 1128 |
| | 2 | 10 | 128 | 756 | 1473 |
| | 3 | 10 | 138 | 696 | 967 |

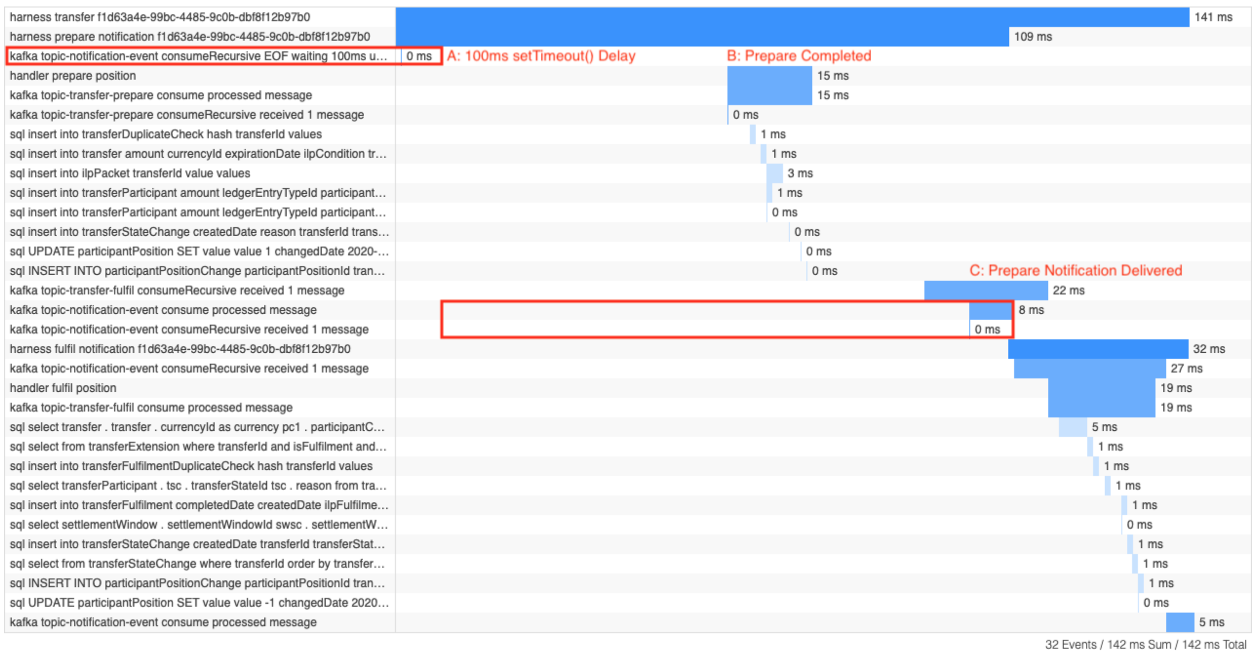
## Big Gap Analysis

### Summary

The term “Big Gap” refers to the large period of time, on the waterfall chart for a single transfer, between the prepare and fulfill handlers processing a message. We do not think this would be a problem in a production environment as increased queue depth and multiplicity of handlers minimizes the effect. However, operators need to be aware of this and we recommend that sanity checks be introduced in the code for queue consumption settings.

### Waterfall

We instrumented the *central-services-database* and *central-services-stream* libraries to capture how the time for a transfer is spent:



As can be seen, the total time for the transfer is 142ms of which at least 100ms is due to a *setTimeout()* delay.

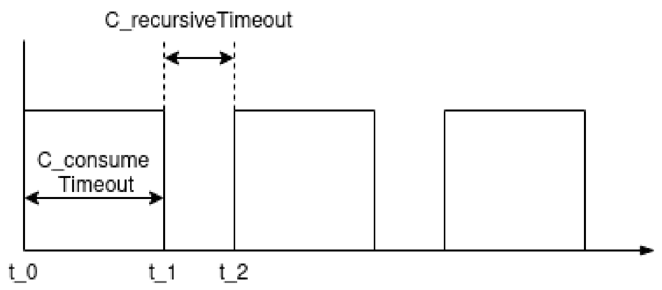
## Mojaloop Kafka Consumer

Mojaloop supports different ways to consume from a Kafka topic. This focuses on just one method as this was the default at the time of testing *viz.* Recursive consuming with synchronous message processing. A consumer subscribes to a Kafka topic at which time a recursive call to consume is started. When consuming, a batch (C_batchSize) of messages is requested. if there aren't enough messages to satisfy C_batchSize, Kafka will wait for up to C_consumeTimeout before returning what messages it has. Once the consumer has the messages it then adds them to a queue to be processed at the configured concurrency.

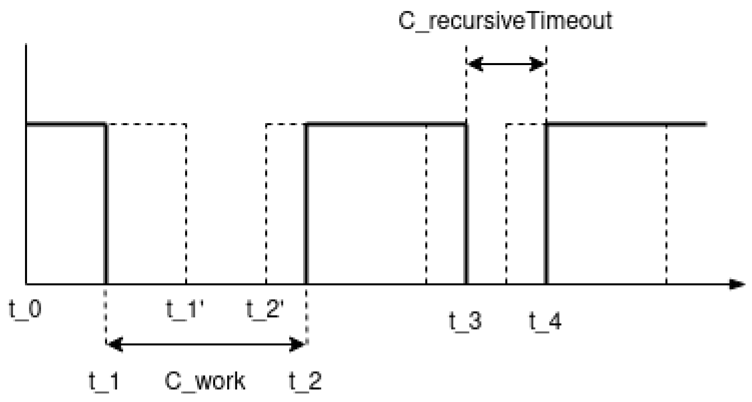
| **Constant** | **Default Value** | **Description** |
|---|---|---|
| C_recursiveTimeout | 100ms | The setTimeout() delay before consuming from the topic again if there are no messages or there has been an error. |
| C_work | Varies | The time taken for a handler to process a message. |
| C_consumeTimeout | 1000 ms | The time that Kafka will wait up to for C_batchSize to be met before returning the messages. |
| C_batchSize | 1 | The number of messages consumed from a topic at a time |
| C_concurrency | 1 | The number of messages that are processed concurrently. |

## Illustration

To illustrate this, consider the case where there are no messages in the Kafka topic. A timeline of what is happening would look like:



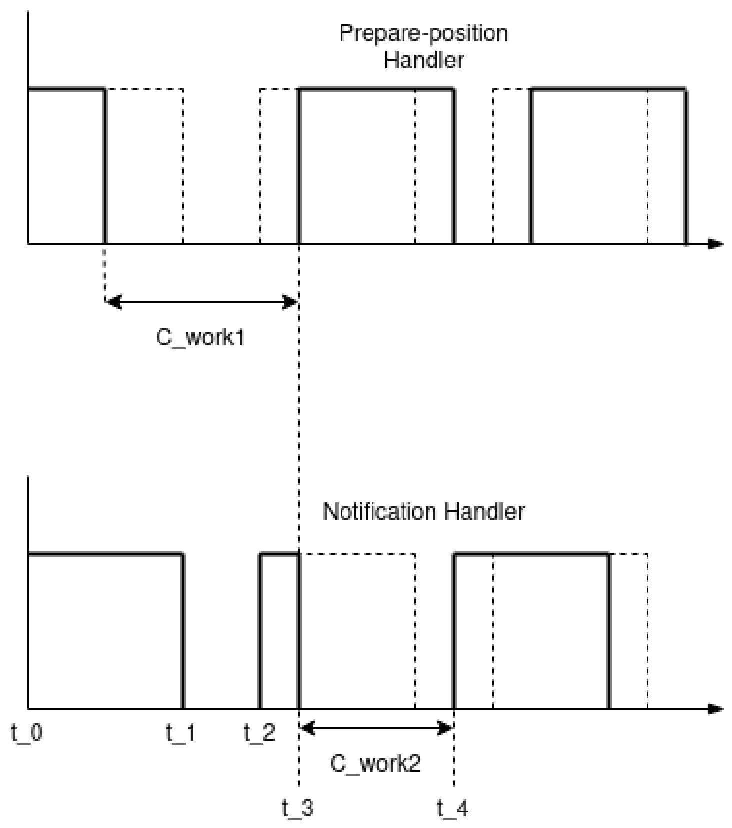
In the figure above, time elapses on the horizontal axis and the vertical axis represents when the Kafka topic is being connected to by the consumer. At t_0, the consumer connects to the Kafka topic. Since there are no messages in the topic, Kafka will wait for C_consumeTimeout before closing the connection at t_1. The consumer will wait for C_recursiveTimeout before connecting to the topic at t_2. You can see that the effect is to create windows where the Kafka topic is checked for messages. Now consider the case where a single message is placed into the Kafka topic:



The dotted lines illustrate the first case where there are no messages. The bolder lines illustrate how this changes when a message is placed onto the topic and then consumed for processing. Noting that C_batchSize = 1, Kafka will return the message it has at t_1. The consumer will then process this message for C_work before connecting to the topic at t_2. As can be seen, straight after the handler finished its processing it consumes from the Kafka topic again. This results in the windows where the topic is checked for messages to be shifted to the right of the original windows.

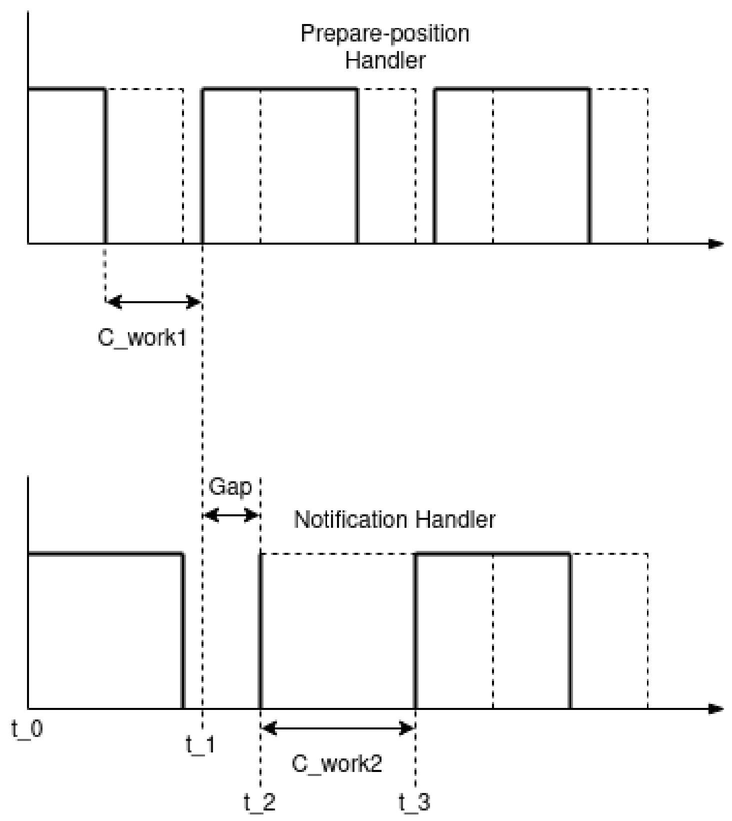
## Latency Gaps

There is a potential for latency to be introduced when handlers are connected in series. In the case of Mojaloop, this means the Prepare-position handler produces to the same topic that the Notification handler consumes from. This is illustrated in the figure below:



Here we have assumed that the handlers are using the same C_consumeTimeout, C_recursiveTimeout and are started at the same time. The top part of the above figure is the Prepare-position handler receiving a message in its window. It starts to process the message. In the meantime, the Notification handler connects to the notification topic at t_2. At t_3 the Prepare-position has finished processing the message and produced it to the notification topic. Since this has fallen inside of the window where the Notification handler is connected to Kafka, Kafka will return the message to the notification handler. In this case, there is no latency.

Let us consider the scenario illustrated in the figure below.



This is similar to the previous scenario except the Prepare-position handler produces to the notification topic at t_1. This is in the C_recursiveTimeout of the Notification handler. This introduces a Gap = t_2 - t_1 as the Notification handler will only connect to the topic at t_2.

## Analogy

Consider a train arriving at a station to collect passengers. Messages are passengers arriving according to some distribution. The constraint is that the train will leave when it is full or when the departure time has been reached; whichever comes first. C_work is the time it takes for the train to go to its destination and return. Here we can see the throughput of this system is dependent on how many people the train can carry at a time. The wait-time or latency for passengers is dependent on the time they arrive and the arrival windows of the trains.

## Latency Bound

Based on the above observations, we can give an upper bound for the latency. Consider the below variables:

| Variable | Description |
|---|---|
| x_1 | Start offset for handler 1 |
| x_2 | Start offset for handler 2 |
| C_stagger | x_1 - x_2 |
| x | Offset for when message is put onto handler 1 topic |
| C_consumeTimeout1 | Consume timeout for handler 1 |
| C_consumeTimeout2 | Consume timeout for handler 2 |
| C_work1 | Time for handler 1 to process message |
| C_work2 | Time for handler 2 to process message |
| C_recursiveTimeout1 | Recursive wait time for handler 1 |
| C_recursiveTimeout2 | Recursive wait time for handler 2 |

Then

$$\text{Gap} = C\_consumeTimeout2 + C\_recursiveTimeout2 + x\_1 - (x\_2 + x + C\_work1)$$

Let C_stagger = x_1 - x_2

$$\text{Gap} = C\_consumeTimeout2 + C\_recursiveTimeout2 - (x + C\_work1) - C\_stagger$$

We assume that (-C_recursiveTimeout2) < C_stagger < C_recursiveTimeout2 then

$$\text{Gap} < C\_consumeTimeout2 + 2 * C\_recursiveTimeout2$$

## Discussion

From the above we see that the latency is dependent on the constants used for the handler consuming from a topic. For a Mojaloop prepare leg, the Prepare-position handler consumes from the Prepare topic and the Notification handler consumes from the Notification topic.  Since both handlers are configured to use the same C_recursiveTimout and C_consumeTimeout:

$$\text{Prepare Gap} < 2 * (C\_consumeTimeout + 2 * C\_recursiveTimeout)$$

Similarly for the Transfer leg, the Fulfill-position handler consumes from the Fulfill topic and the Notification handler consumes from the notification topic.

$$\text{Fulfill Gap} < 2 * (C\_consumeTimeout + 2 * C\_recursiveTimeout)$$

For a transfer this gives

$$\text{Total Gap} < 4 * (C\_consumeTimeout + 2 * C\_recursiveTimeout)$$

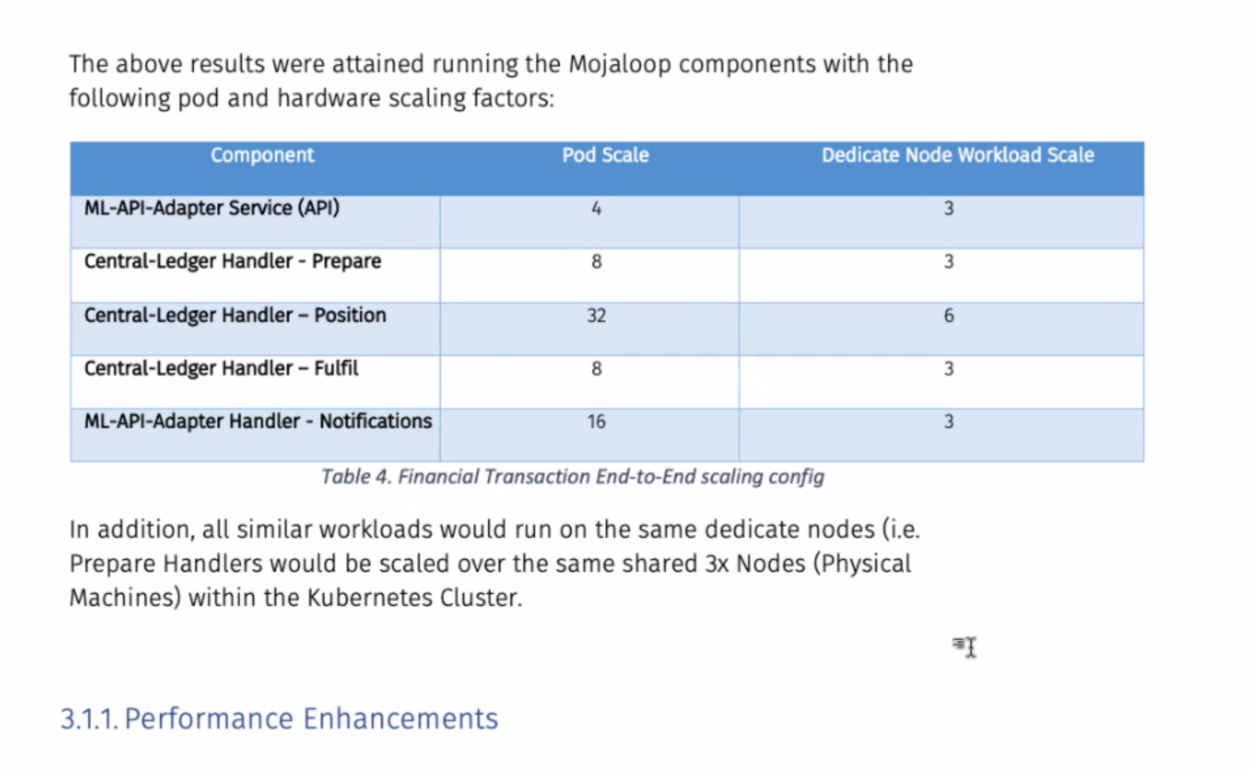
## Conclusion

Throughput and latency may or may not relate to each other, and that if they do, then this relationship may be inverse or even direct, according to the nature of the system being analyzed. In the case of Mojaloop, the “Big Gap” problem is a latency cost incurred by the fact that messages are pulled from a topic. This cost can be reduced by decreasing timeouts and better make use of that time by increasing concurrency but it is unlikely to be eradicated due to the fundamental constraints of the system. Finally, this is a problem only when there are intermittent arrivals in payments, leading to the Kafka queues not always having messages to consume. We do not think this would be a problem in a production environment as increased queue depth and multiplicity of handlers minimizes the effect. However, operators need to be aware of this and we recommend that sanity checks be introduced in the code for queue consumption settings.

## Cluster Specifications

### ModusBox Reference Cluster - AWS

1 instance = i3Xl - AWS storage optimised. 4 vCPUs with 30.5 GiB memory
Kafka + zookeeper 3 instances
1 instance for mysql



The above results were attained running the Mojaloop components with the following pod and hardware scaling factors:

| Component | Pod Scale | Dedicate Node Workload Scale |
|---|---|---|
| ML-API-Adapter Service (API) | 4 | 3 |
| Central-Ledger Handler - Prepare | 8 | 3 |
| Central-Ledger Handler – Position | 32 | 6 |
| Central-Ledger Handler – Fulfil | 8 | 3 |
| ML-API-Adapter Handler - Notifications | 16 | 3 |

*Table 4. Financial Transaction End-to-End scaling config*

In addition, all similar workloads would run on the same dedicate nodes (i.e. Prepare Handlers would be scaled over the same shared 3x Nodes (Physical Machines) within the Kubernetes Cluster.

3.1.1. Performance Enhancements

### Coil Cluster - GCP

Instance type: n1-standard-4 4vCPUs 15GiB RAM
[https://cloud.google.com/products/calculator/#id=5ccb8977-1eb6-45ed-ac41-6415dd708632](https://cloud.google.com/products/calculator/#id=5ccb8977-1eb6-45ed-ac41-6415dd708632)

| Component | Pod Scale | Dedicated Nodes | label |
|---|---|---|---|
| Kafka + ZooKeeper | | 3 | broker |
| MySQL | 1 | 1 | data |
| ML-API-Adapter | 1 | 1 | ml_api |
| Prepare handler | 1 | 1 | ml_cl_prepare |

| | | | |
|---|---|---|---|
| Position handler | 1 | 1 | ml_cl_position |
| Fulfill handler | 1 | 1 | ml_cl_fulfil |
| Notification handler | 1 | 1 | ml_notify |
| Load generator | 1 | 1 | load |
| monitoring | | 1 | monitor |
| SSD | | 32 GB | |