Database Connections Refactorings

Fixing Connection Pooling

The purpose of having read and write urls to database is to balance out connections between read and writes, so not all connections go to write.

However, the way spring + r2dbc works is that at the moment of @Transactional annotation being added on a method, it will fetch a connection from the pool, try to use same connection for multiple requests to database, and even try to merge multiple writes on the same request to database at the end of the transaction, so that it optimises as much as possible the resources.

In shipping service there is no @Transactional annotation used on a service method, which is the place where it should have given the most of its benefits. Right now service methods are not transactionals. Each request to database uses a different connection and each write is flushed immediately, robbing the system from all optimisations the framework was having built in.

The problem comes from separation of read and writes on the transaction managers. Because of that, there are multiple downsides:

1. There is no real transaction on an api operation. If one write fails, others are not rollbacked, as they are incorporated into their own transactions. Adding the proper @Transactional on the service method, will not work because of the separation of the transactionRouting class and read/writes connection pools.
2. Multiple connections are used during a single api operation. It jumps from write to read connection pool for a single service method, instead of trying to reuse same connection when possible.
3. Multiple flushes to database even when they are not necessary will happen. If one service has @Transactional annotated r2dbc and spring will try to optimise multiple writes on the same request. This is not happening now.
4. A lot of code written to maintain read operations, via dbClient and repositoryTemplate. The read and writes should all be using a single spring data repository and the footprint of custom code will dramatically go down.
5. Replicating from write to read can take in some cases up to 100ms. If it is in the same api operation a read right after the write happened, it will not be retrieved, leading to inconsistencies.

Leveraging read/writes urls does not mean absolutely each small request has to go on one url or the other. It can be accomplished by switch the connection pool based on the api method invoked.

All GET api method will go to read database.

All other api methods will go to write database.

This can be easily be accomplished with a global filter.

Eliminating the 2 transaction managers is the next step, so it will allow one service method to be annotated with @Transactional.

Eliminating read repositories will lead to removal of a big chunk of custom code and possible bugs for the future, while using only spring data repositories.

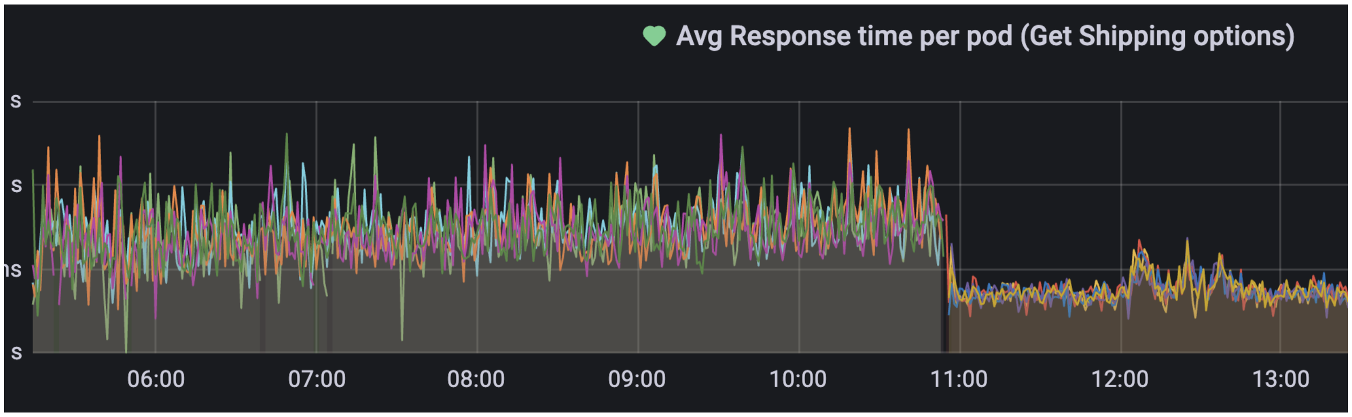
There are several planned improvements in shipping-service related to database connectivity. The most impactful is

~~AFN-4350~~ - Remove / Refactor read only/ repository template code. CLOSED

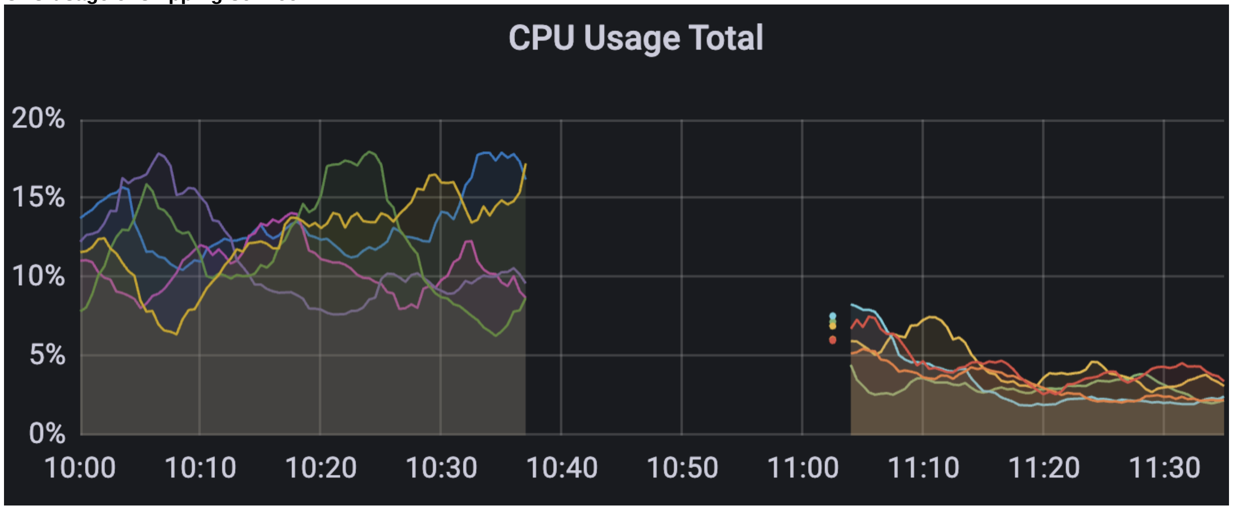
After the release of this refactoring following metrics were observed:

At shipping-service level:

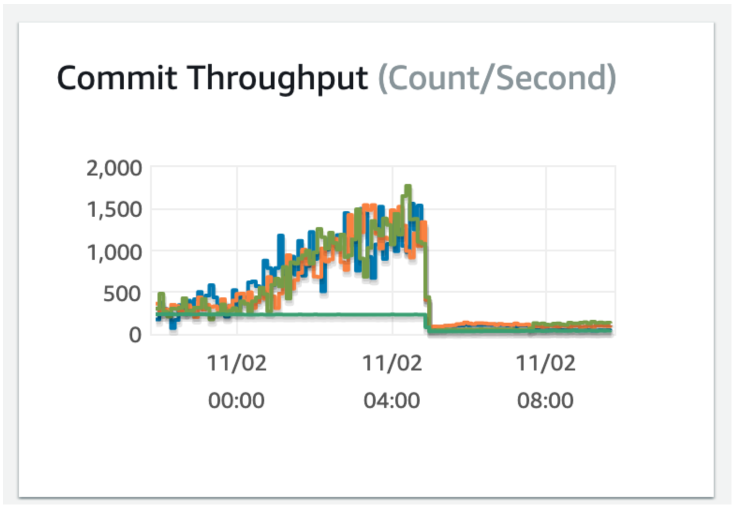
Response times for most used api:

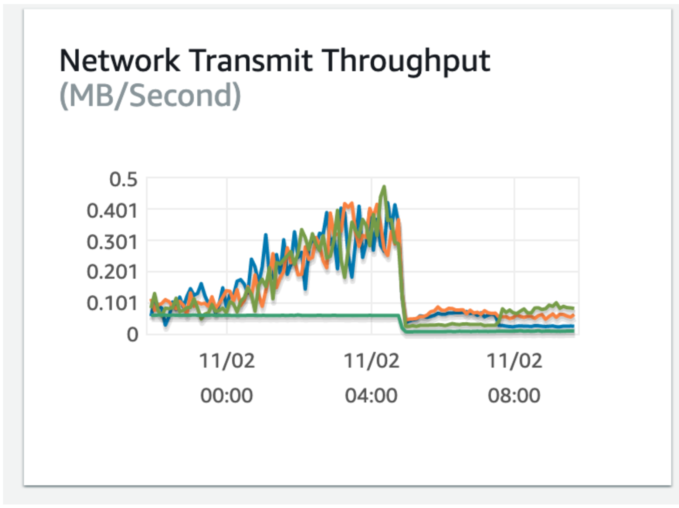


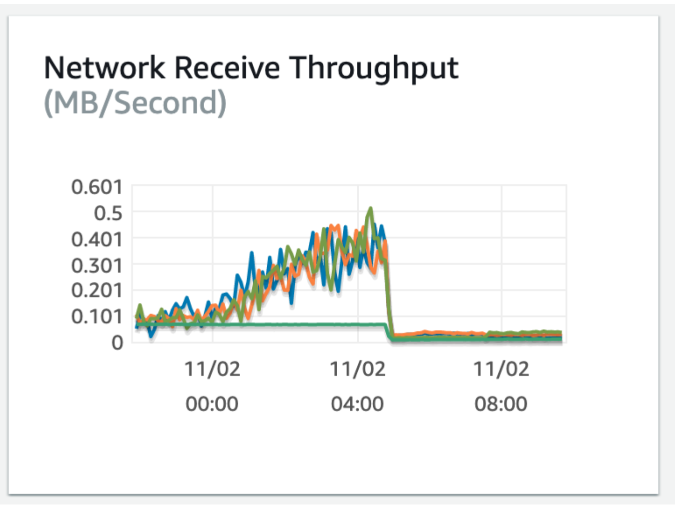
CPU usage of shipping-service:

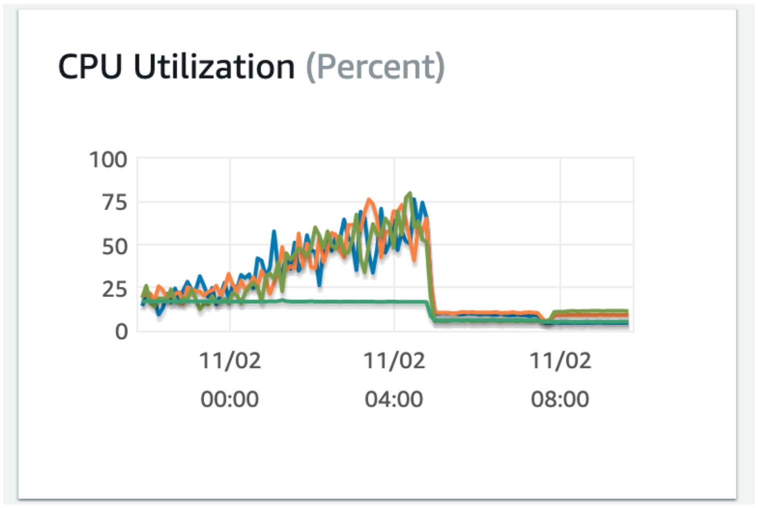


Database load of production:









Refactoring queries optimise requests to Database

~~AFN-4360~~ - Reduce queries to database by order them in a differently CLOSED

The services at the moment are implemented with absolute disregard of the footprint on performance all the queries have.

With a little re-organisation of the queries, a lot of them can be eliminated.

Example: delete or update by multiple parameters.

It makes no sense to first select by one parameter, then 2, then more just to throw the same 404 exception to the client. Having queries on database to simply produce logs has a big impact on performance for almost 0 gains.

Proper way is to delete/update by all those parameters and check the return parameter to see if the entity was indeed in the database or not. This eliminates multiple read queries happening before the write.

Second optimisation here is to start using transactions on the service methods to group multiple updates/inserts into a single request.

AWS RDS Proxy

Another good practice and optimisation is to use AWS RDS Proxy.

https://aws.amazon.com/rds/proxy/

This looks perfect to manage connections and minimise load on the DB level.

After initial successful testing, after some time it was noticed a restart of the pods in production and staging every 24 hours. This was due to proxy cutting connections down after 24 hours. Investigations still ongoing how to properly configure driver connections to match this expectation.