ECE 590:
Scalability

Faris Sbahi, Andrew Bihl, Xinghao

April 6, 2018

1 Code Analysis

We designed our Exchange Engine using Golang. One of the primary motivators for doing so is Go's built-in concurrency system, centered around the use of goroutines. Goroutines are similar to threads, however instead of there existing a one-to-one mapping of threads to goroutines, each goroutine is mapped to a group of threads. Hence, once the swap context and thread setup overhead becomes too high, Go will start mapping parallelized executions to the same threads.

Hence, each time our server accepts a request, the request is handled as part of a separate goroutine. This is our first level of parallelization.

Furthermore, we parse the received XML as part of a stream which allows us to start performing a job in parallel (e.g. buy order) as we continue reading. This seems especially useful for requests that consist of an extremely large amount of data.

When dealing with "transaction" and "create" requests, idempotency and atomicity become increasingly important because there are several places that a given request can fail. Hence, it was clear that we needed to execute certain operations atomically and rollback if a single command in the operation failed. For example,

If a goroutine holds a RWMutex for reading and another goroutine might call Lock, no goroutine should expect to be able to acquire a read lock until the initial read lock is released.

2 Experiments

2.1 Scalability

As mentioned in our code analysis, it was left to measure whether Reader-Writer Locks substantially improved our performance. One simple test was to send 1000 requests in parallel that are reader-intensive

```
time seq 1000 | parallel -n0 "cat transaction/query/1.txt | nc localhost 12345"
```

Average over 10 experiments, we found that the RWMutex method took 15.191s, substantially better than 19.463s found using regular mutexes.

WE NEED A LOT OF CHARTS HERE

I INCLUDED ONE OF HILTON'S CHARTS BELOW

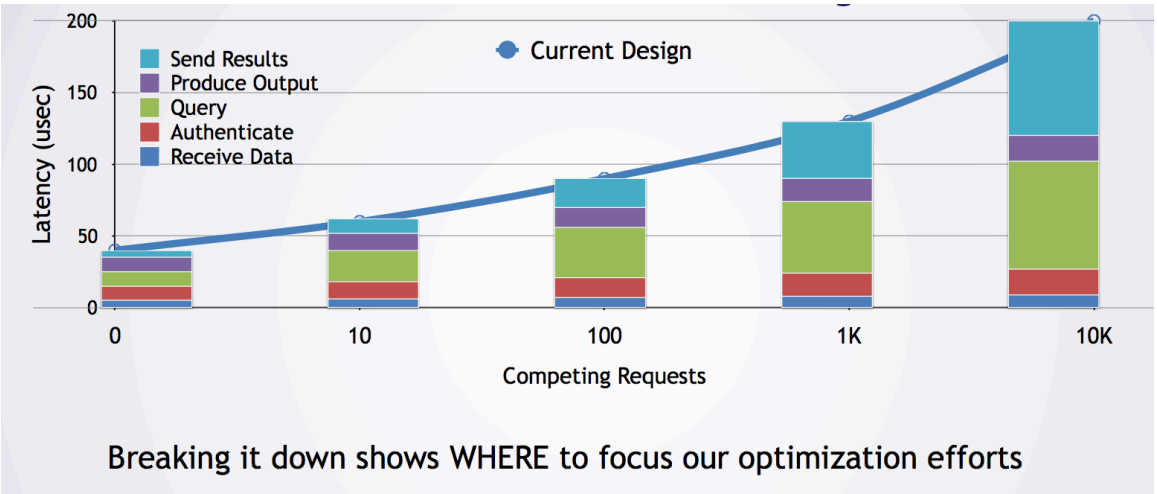

Figure 1: Scalability Breakdown

EXPLAIN HOW WE GOT THIS DATA EXPERIMENTALLY

2.2 Correctness

We developed specialized test cases that are executed as part of our `tests.sh` script. These ensure that we have correctness for the standard and edge cases that we came up with. Furthermore, we dump the database as part of our separate debug log each time we handle a request. Hence, we are able to see whether the correct values exist in the database at each step.

Furthermore, we tested for datastore failures my taking down the Redis and Postgres containers and random points of our testing and ensured that the same results were achieved.

3 Discussion

Evidently, we found that using Redis as a cache seriously improved our ability to scale.

Furthermore, we found that INSERT is our greatest bottleneck. One way we could mitigate this, if we were to continue working on this project is to ....