TigerBeetle

An accounting database
High-throughput low-latency
Two-phase commit ledger transfers

Agenda

1. Background and Mission
2. Problem Statement and Design Decisions
3. TigerBeetle
   a. Safety
   b. Performance
   c. Experience
4. Mojaloop

Background and Mission

- ProtoBeetle: Output from the performance workstream
- Re-affirmed our hunch, “Build a Redis for Accounting”
  Inspired by TimescaleDB (https://www.timescale.com/)

Mission

Make it easy to build financial applications without building an account system from scratch.

Implement the latest research and technology to deliver unprecedented safety, durability and performance without adding operating cost and complexity.

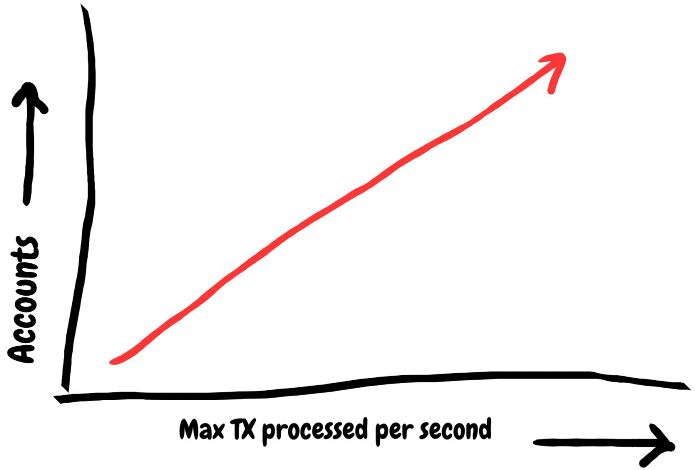
Problem 1: Strict Requirements

Real-time processing of balance updates is hard to do right:

- Updates must be done in the correct order
- Can’t be processed in parallel making horizontal scaling and sharding almost impossible
- Hard to scale a ledger without sacrificing performance and/or safety and durability

Existing architectures use **generic** databases (relational or NoSQL, on-disk or in-memory) with accounting logic enforced in the **application** code.

Problem 2: Hubs **most** impacted



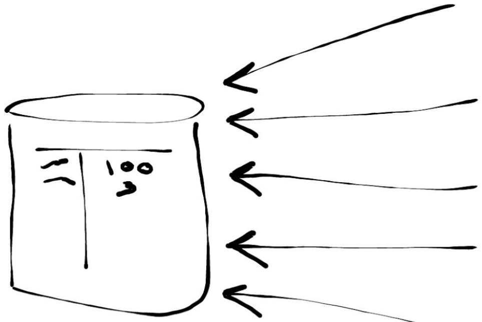
- Throughput is limited **per account** (operations on a single balance can't be done in parallel)
- Hubs have only a few peering accounts but still process high transaction volumes
- Participants split throughput among many user accounts

Problem 3: 10+ DB Queries per Transfer

Separation between **data** and **code**, **persistence** and **logic**

Developers have to re-implement accounting business logic in code on top of generic storage systems

- network delays
- multiple round trips per update
- clock skew
- hard to test
- error prone



```
prepareTransfer(payer, payee, transfer)
{
  validatePayerAccount()
  validatePayeeAccount()
  lockPayerAccount()
  checkPayerAccountLimits()
  checkPayeeAccountLimits()
  reservePayerFunds()
  storeTransferDetails()
  releasePayerAccount()
}

commitTransfer(payer, payee, fulfilment)
{
  validateFulfil()
  lockPayerAccount()
  commitPayerFunds()
  updatePayeeBalance()
  storeFulfilmentDetails()
  releasePayerAccount()
}
```

Problem 4: Hardware failures...

...assumed to be handled... somewhere?

https://www.usenix.org/conference/atc20/presentation/rebello

Can Applications Recover from `fsync` Failures?

Anthony Rebello, Yuvraj Patel, Ramnatthan Alagappan,
Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau
Computer Sciences Department, University of Wisconsin – Madison

Abstract

We analyze how file systems and modern data-intensive applications react to `fsync` failures. First, we characterize how three Linux file systems (ext4, XFS, Btrfs) behave in the presence of failures. We find commonalities across file systems (pages are always marked clean, certain block writes always lead to unavailability), as well as differences (page content and failure reporting is varied). Next, we study how five widely used applications (PostgreSQL, LMDB, LevelDB, SQLite, Redis) handle `fsync` failures. Our findings show that although applications use many failure-handling strategies, none are sufficient: `fsync` failures can cause catastrophic outcomes such as data loss and corruption. Our findings have strong implications for the design of file systems and applications that intend to provide strong durability guarantees.

Problem 5: Corruption is an issue

redis / redis

Code | Issues 1.7k | Pull requests 663 | Actions | Projects 3 | Security | Insights

Silent data corruption in Redis #3730

Open aganesan4 opened this issue on Jan 6, 2017 · 2 comments

aganesan4 commented on Jan 6, 2017

Redis does not use checksums for its entries in its appendonly file. Without this, Redis is vulnerable to silent data corruptions resulting from underlying problems in disks and file systems atop them [1,2].

We setup a redis cluster with three nodes. In a small test case where the underlying disk/FS corrupts the key or value in the master's appendonly file, Redis can silently return corrupted user data on a read request.

Moreover, the master slave resynchronization protocol in Redis spreads the corrupted data to other intact slaves. We reproduced this scenario using our testing framework.

Is there a reason why Redis doesn't use checksums to protect the data in appendonly file from data corruptions?

[1] https://research.cs.wisc.edu/wind/Publications/zfs-corruption-fast10.pdf
[2] http://www.cs.toronto.edu/~bianca/papers/fast08.pdf

👍 2

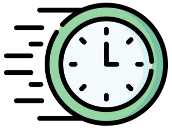
antirez added AOF operations labels on Jan 13, 2017

antirez commented on Jan 13, 2017 Contributor

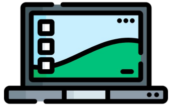
Hello. Yes... this is desirable indeed. A few points in order to articulate further discussions:

Motivation

Safety



Performance

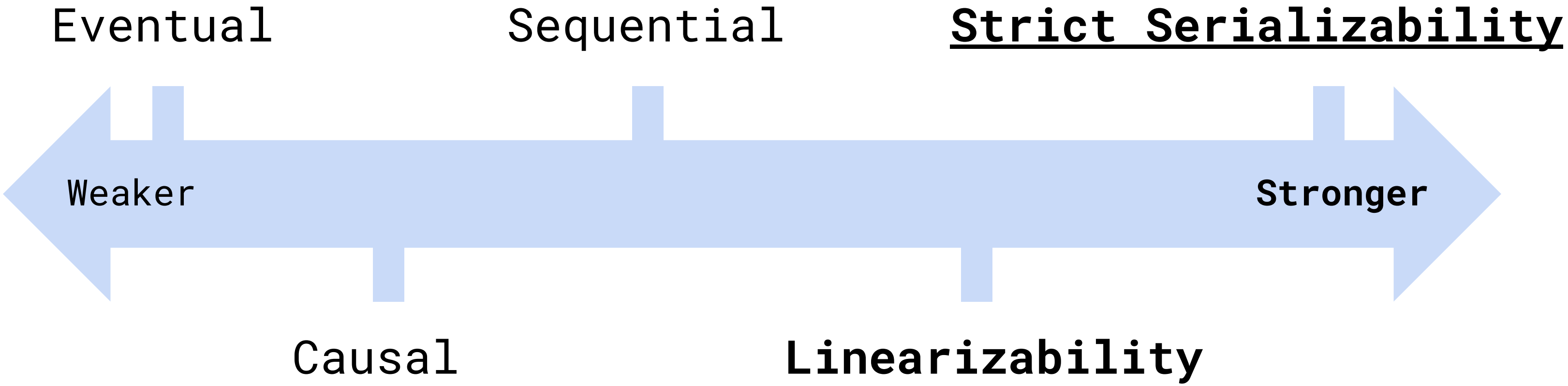


Experience

Safety

- **Replicated state machine for <u>fault-tolerance</u>**
  Classic LMAX design
- **Viewstamped replication for <u>strict serializability</u>**
  Automated leader election and reconfiguration
  (Multi-Paxos + Flexible Paxos)

Safety - Consistency Models

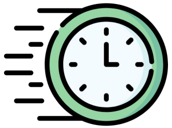


https://jepsen.io/consistency
http://www.bailis.org/blog/linearizability-versus-serializability/
https://www.cs.princeton.edu/courses/archive/fall18/cos418/docs/p8-consistency.pdf

Safety

- **Replicated state machine for fault-tolerance**
  Classic LMAX design
- **Viewstamped replication for strict serializability**
  Automated leader election and reconfiguration
  (Multi-Paxos + Flexible Paxos)
- **Hash-chained journal entries**
  Detect and repair disk corruption
- **Leader-based timestamping**
  Handle clock skew between replicas

Performance



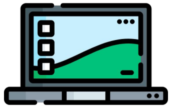
- **Built for purpose DB**
  Simple data structures (account, transfer)
  “Accounting” business logic built-in
  Tightly scoped domain
- **Batching (“everything is a batch”)**
  Amortize network/storage costs
- **Optimized I/O**
  Zero-copy from TCP to disk, state and back (Direct I/O)
  Zero-syscall networking and storage I/O (io_uring)
  Zero-deserialization (fixed-size data structures)
  Minimal memory cache misses (cache line alignment)

Benchmark

```
$ zig run src/benchmark.zig -O ReleaseSafe
connecting to 127.0.0.1:3001...
connected to tigerbeetle
creating accounts...
100000 transfers...
200000 transfers...
300000 transfers...
400000 transfers...
500000 transfers...
600000 transfers...
700000 transfers...
800000 transfers...
900000 transfers...
1000000 transfers...
=============================
527704 transfers per second

p100 create_transfers max latency per batch of 10,000 = 23ms
p100 commit_transfers max latency per batch of 10,000 = 13ms
```

Experience



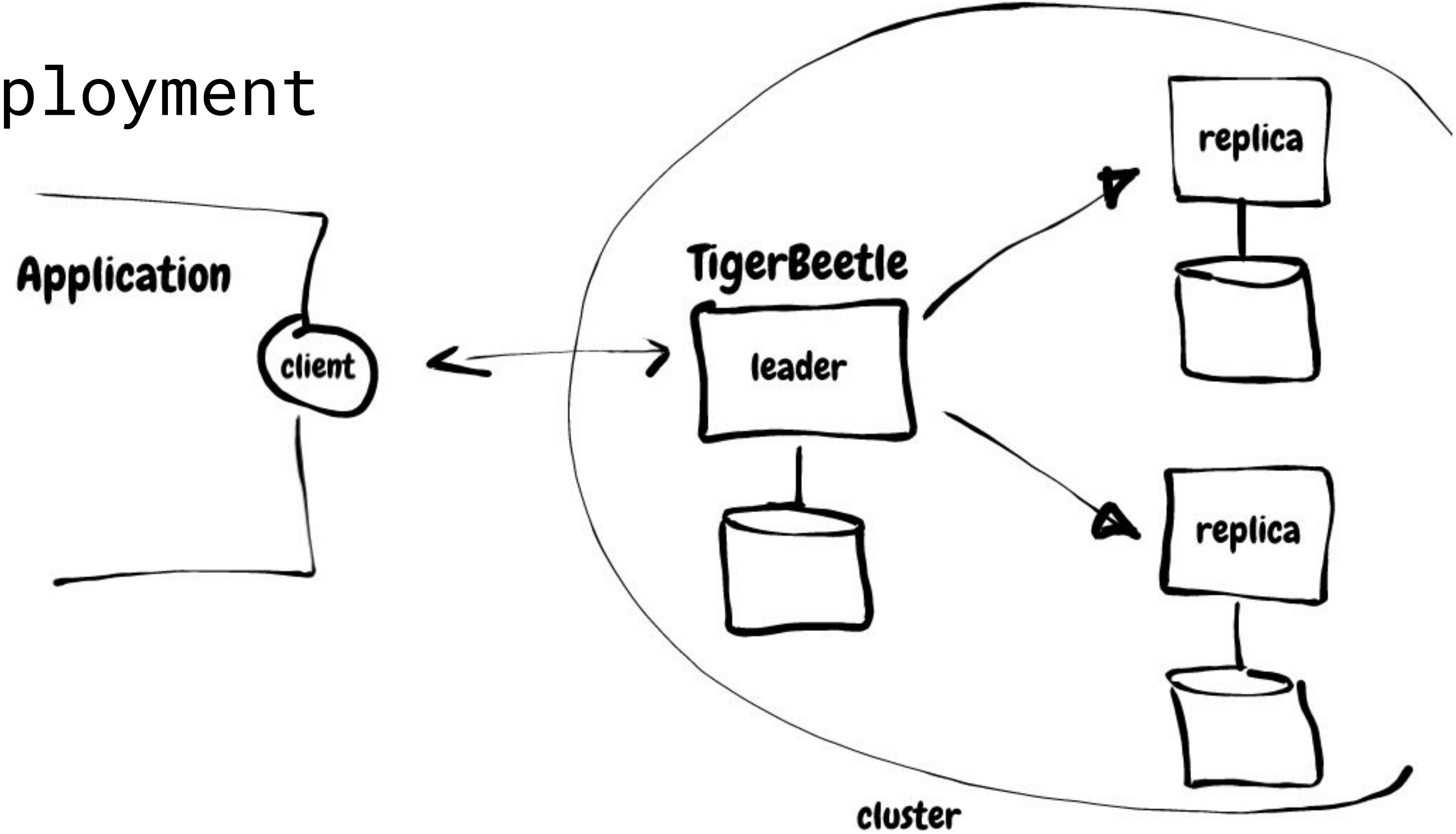
- **Single binary**
  Download and run (or use simple toolchain to compile)
- **Simple config**
  Pre-compiled profiles (development, production)
  Easy to add new profiles to binary (single DSL file)
- **Smart client**
  Batching done by client
  Client works seamlessly with automated leader changes
- **Rich domain specific API**
  Clean separation between persistent or stateless services

The API

- 4 commands supported:
  - Create account
  - Create transfer
  - Commit transfer
  - Lookup account
- Commands submitted in batches and ACK’ed as a batch
- Only failures in ACK response

Create 1,000 accounts… (all ok)
Create 10,000 transfers… (transfer 7 failed, the rest ok)
Commit 10,000 transfers… (commit 7 failed, the rest ok)
Lookup 1,000 accounts… (by id, returns 1,000 accounts)

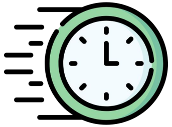
Deployment


Application
client
TigerBeetle
leader
replica
replica
cluster

Tooling

- **Node.js client (and clients for other languages)**
  Open-source community
- **Integration with the storage hierarchy**
  Drain warm data out to SQL, and then out to cold storage
- **Disaster recovery**
  Backup and restore from snapshots
- **Deployment**
  Docker and Kubernetes
- **Monitoring**
  Prometheus

Progress



- **200,000 transfers per second (July 2020)**
  “ProtoBeetle” prototype
- **500,000 transfers per second (October 2020)**
  “AlphaBeetle” full single node safety and performance
- **Leader election and replication (March 2021)**
  “BetaBeetle” fault-tolerant cluster
- **Production (January 2022)**
  “TigerBeetle”

- **Tooling (ongoing)**
- **Independent network and storage fault model audits**

Demo

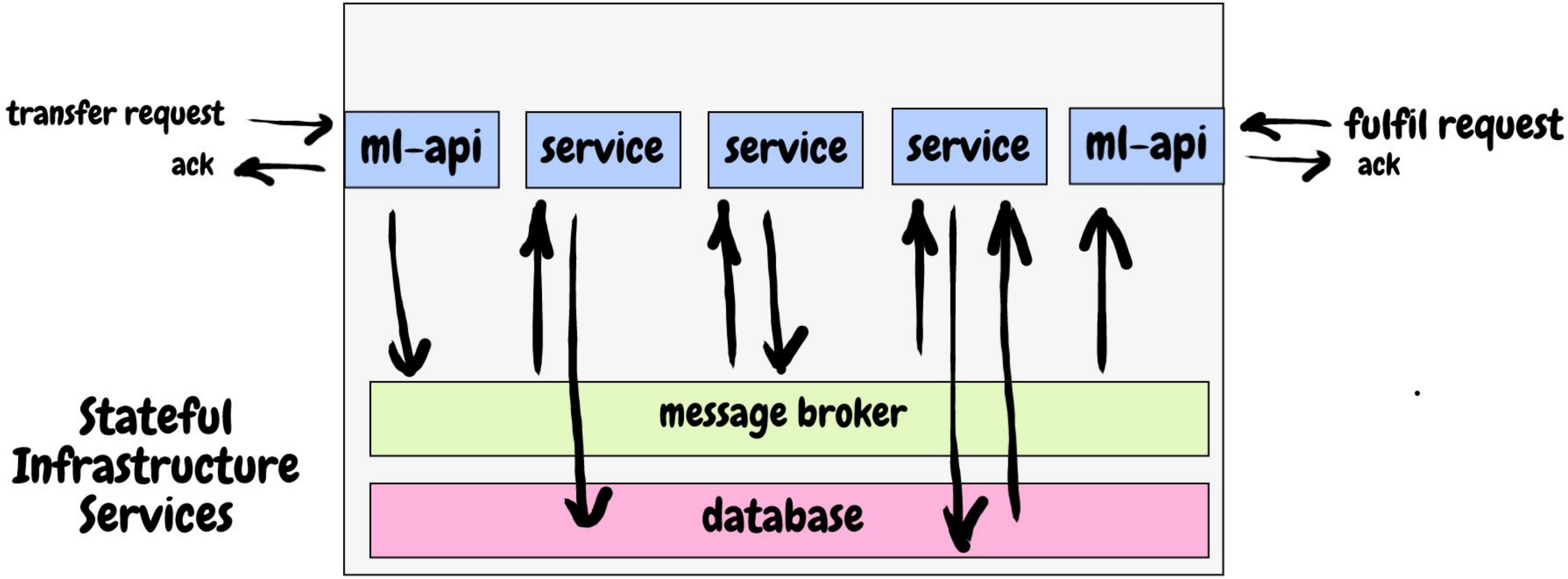
Mojaloop

- **Focus on “horizontal” scalability in current architecture**
  Stateless application services
  Streams and databases as infrastructure

- **Additional network and disk IO**
  Every call to the persistence layer adds up

- **Complex system of interdependent services**
  Difficult to manage and understand
  Expensive to run (inefficiency = cost)

Horizontal Scaling + Microservices



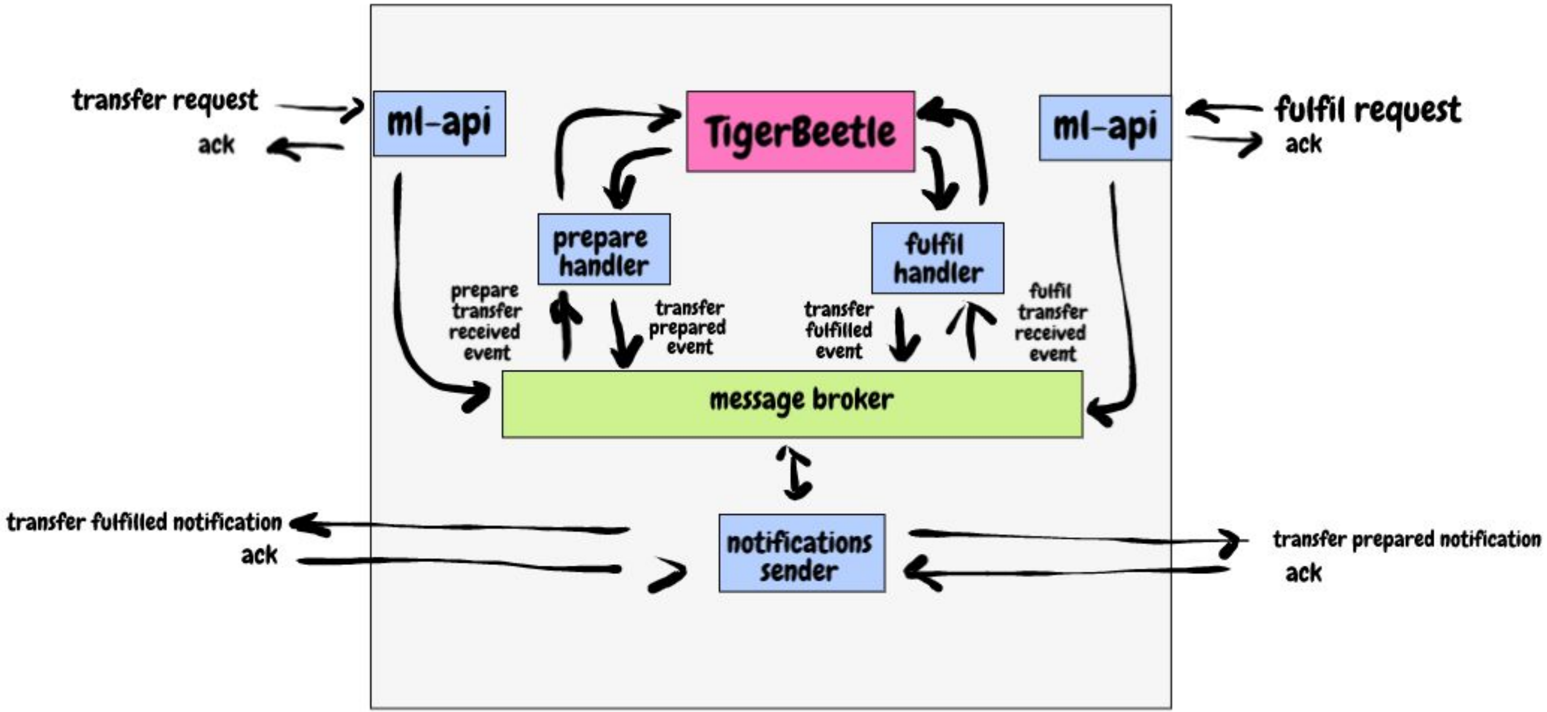
Mojaloop Integration Proposal

- **TigerBeetle is the “source of truth”**
  “Accounting” as infrastructure
  I.e. Durable transfer and account data store with accounting logic and strict serializability

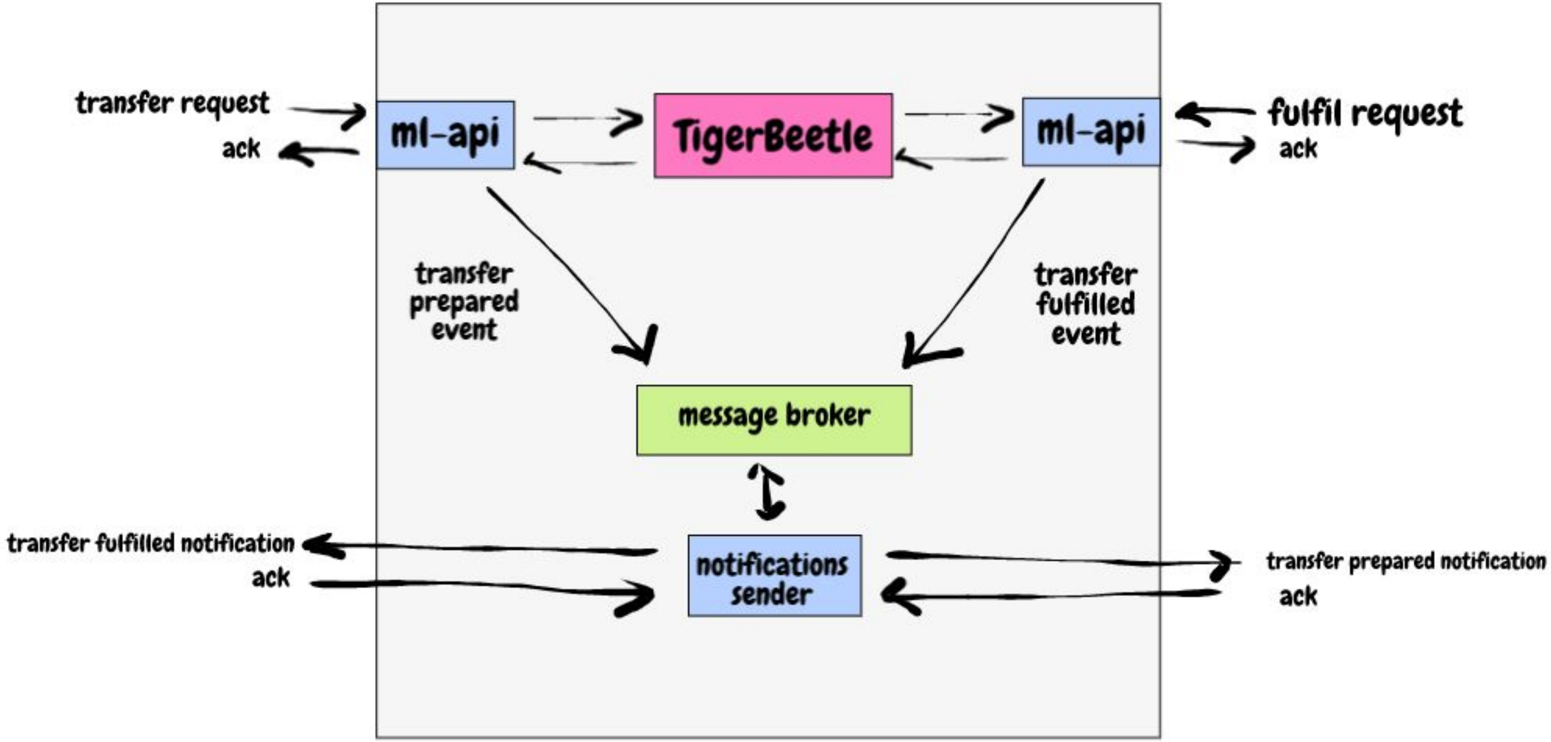
- **NodeJS services (fully stateless)**
  Translate Mojaloop API requests into TigerBeetle commands

- **A durable message broker (Kafka) COULD be used:**
  To separate the API from TigerBeetle
  Allow business processes to tap into the transfer stream

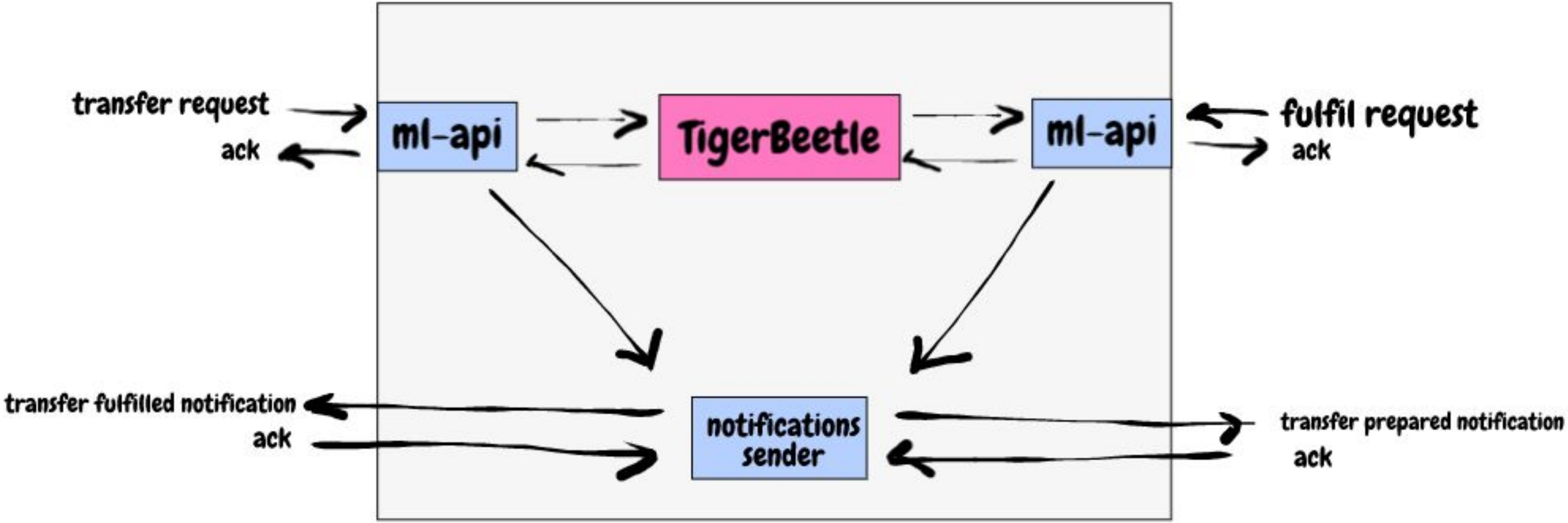
Mojaloop: Option 1 - As is today



Mojaloop: Option 2 - Simpler



Mojaloop: Option 3 - Simplest



Mojaloop: Next Steps

- Performance benchmark (like for like)
- Detailed architecture design
- Validate support for settlement use cases
- Materialized views into MySQL for warm/cold store queries