Powering Databases @ Scale with Mid-Tier Proxy

Kamlakar Singh
PayPal
May 2018

Agenda

- Why DB Proxy?
- Core Design Principles
- High Level Architecture
- Process Architecture
- DB Horizontal Scaling (Sharding)
- Monitoring/State log
- Resiliency Use Case

Some Statistics

- 100 Billion SQLs on peak day of the year
- 2 ms response time (client -> DB proxy -> DB)
- 350+ Oracle Databases, Multiple client tech stacks
- 1K NoSQL Hosts
- 43 PB Total SAN Storage
- Resilient, Available, Performant @ Scale

Why DB Proxy?

Horizontally scaled Data Access Gateway to scale, manage and protect database

DB Resource

- Efficient **Transaction aware** connection **multiplexer**
- Supports **persistent** connections from clients and to DB server

Polyglot and ORM

- Supports **C++, Java, Python**
- Works seamlessly with DAL, Hibernate, and container provided connection pool

Scalability

- **Sharding**
- Inter-cluster routing: Transparent SQL Routing to replica DB
- Intra-cluster routing: Transparent R/W split

DB Maintainability

- Transparent connection migration to support DB maintenance
- Reflects bind variables in DB session (**v$session**)

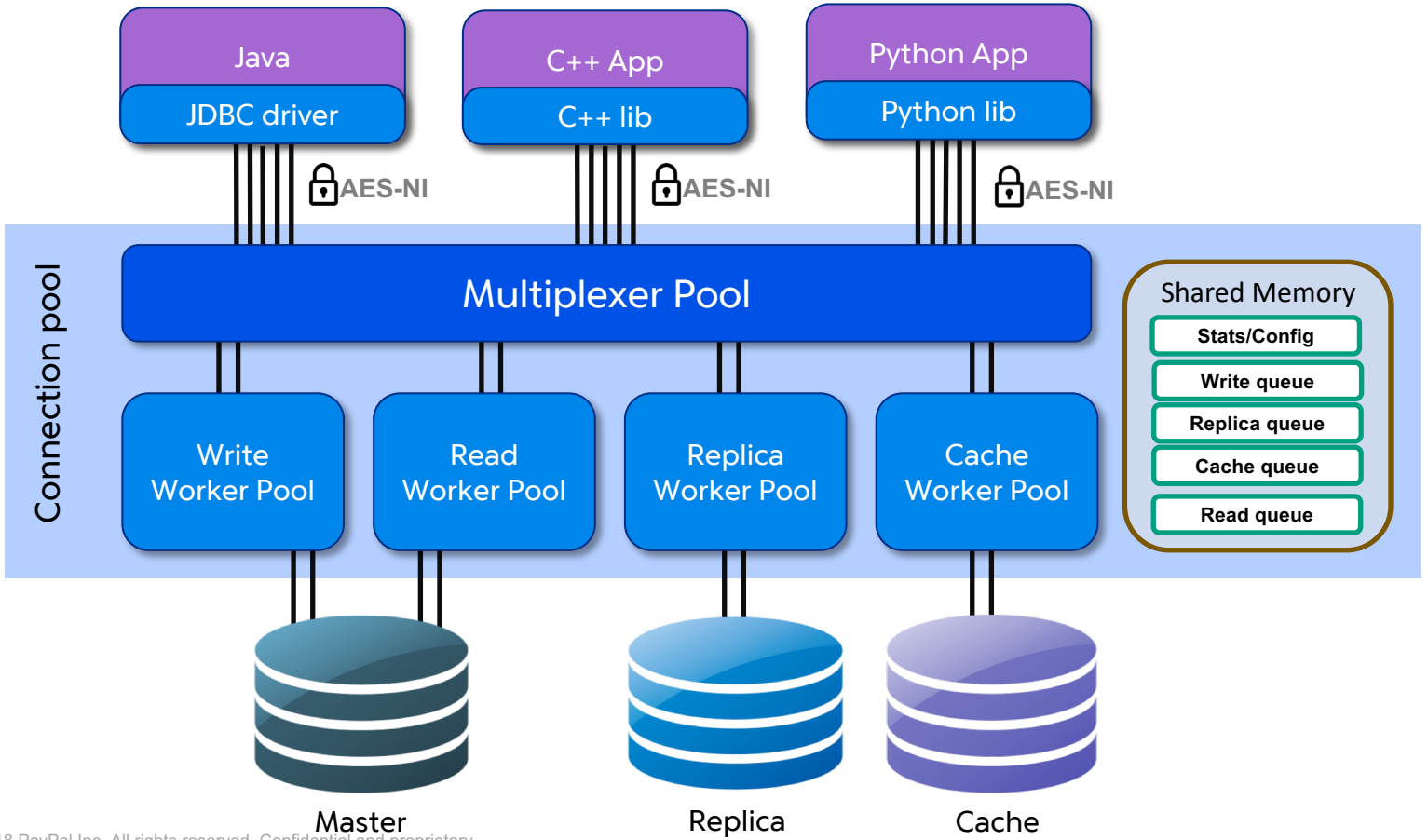
Resiliency

- Surge Protection (Queue, Bouncer and **SQL Eviction**)
- Tolerance for DB unavailability (**no markup/mark down** required)
- *Transparent SQL failover*

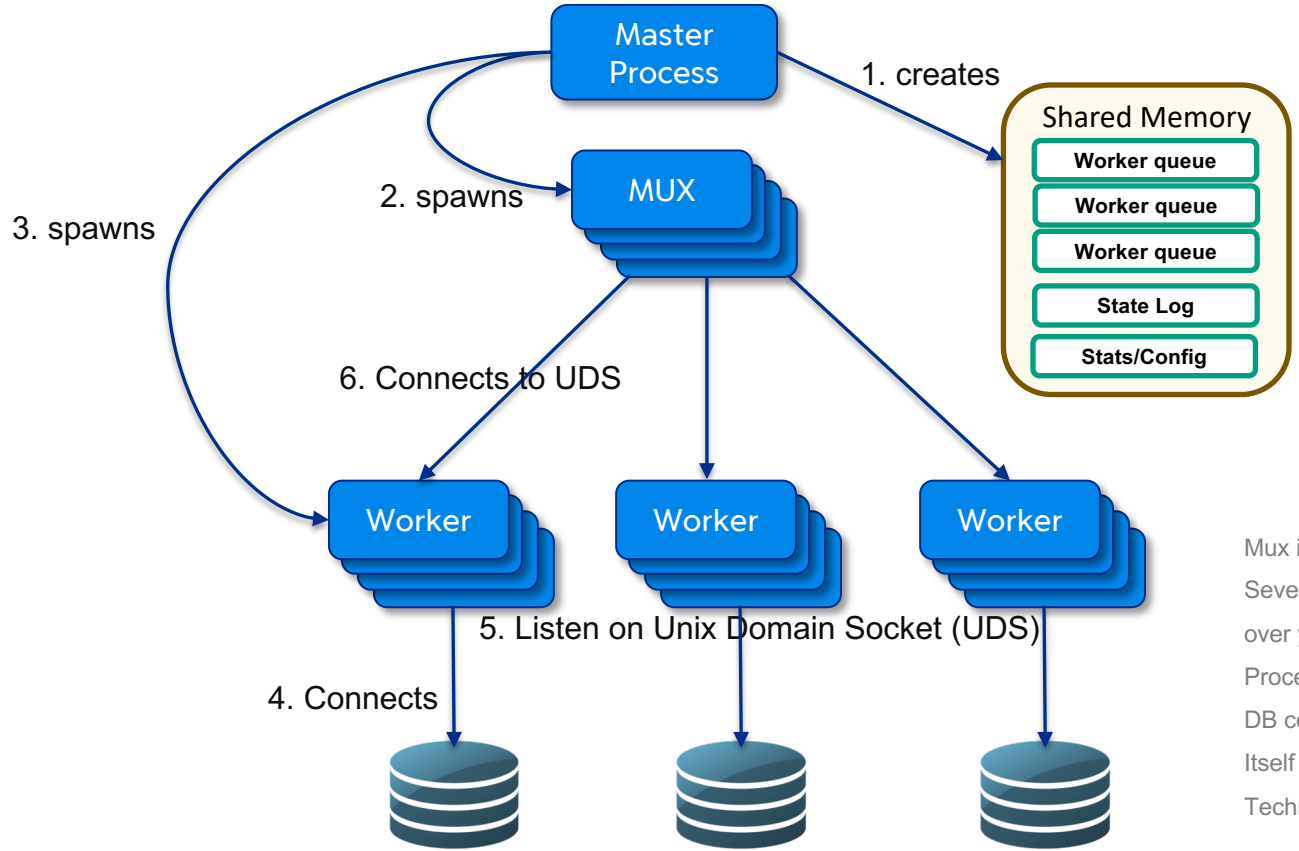
Principles

- Learn how to keep it simple
- Be in love with problem statement not with the solutions (solutions of today require continual adjustment, often times complete re-architecture)
- Thin client (micro-service friendly, features/complexity in server)
- Developers want features/capabilities not ilities (it is Platform team's job to ensure ilites)
- Backward compatibility, correctness trumps performance
- Resiliency is harder than performance
- CPU/memory neutral: every major release to undergo extensive profiling
- Creativity is needed during design, well defined policies in production
- Default like crazy, prefer convention over configuration
- Robustness/Quality

High Level Architecture

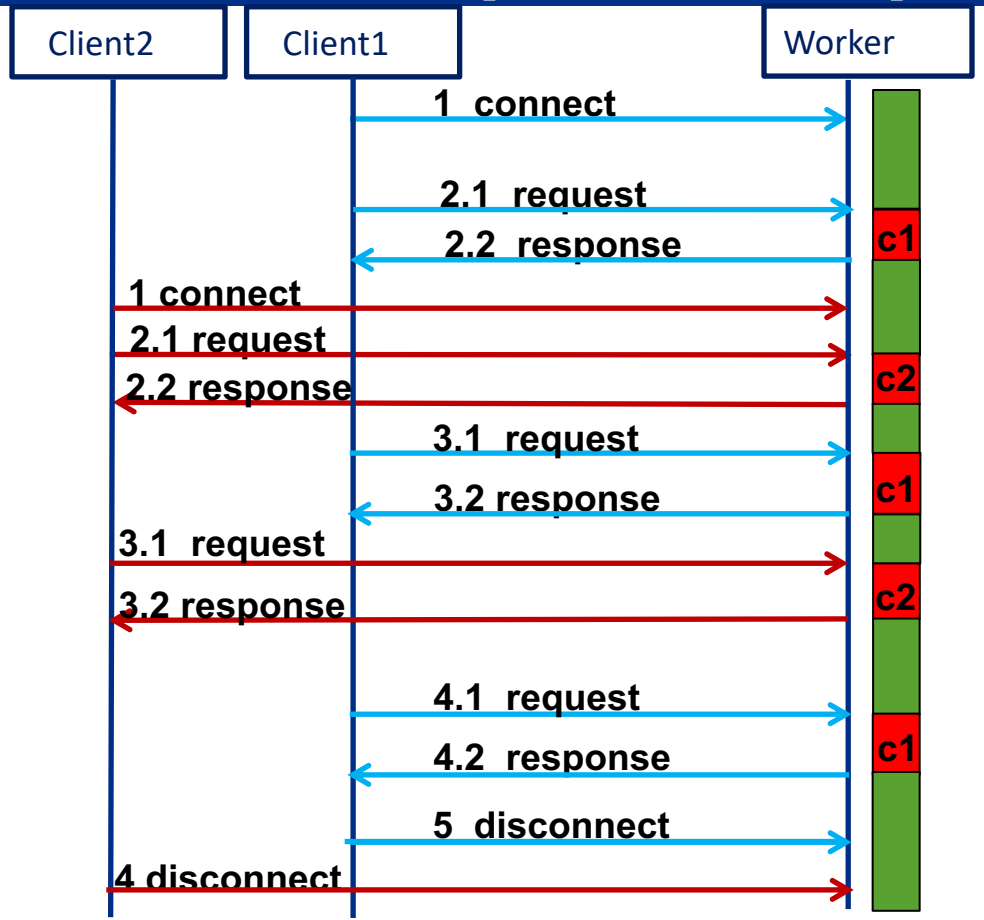


Process Architecture

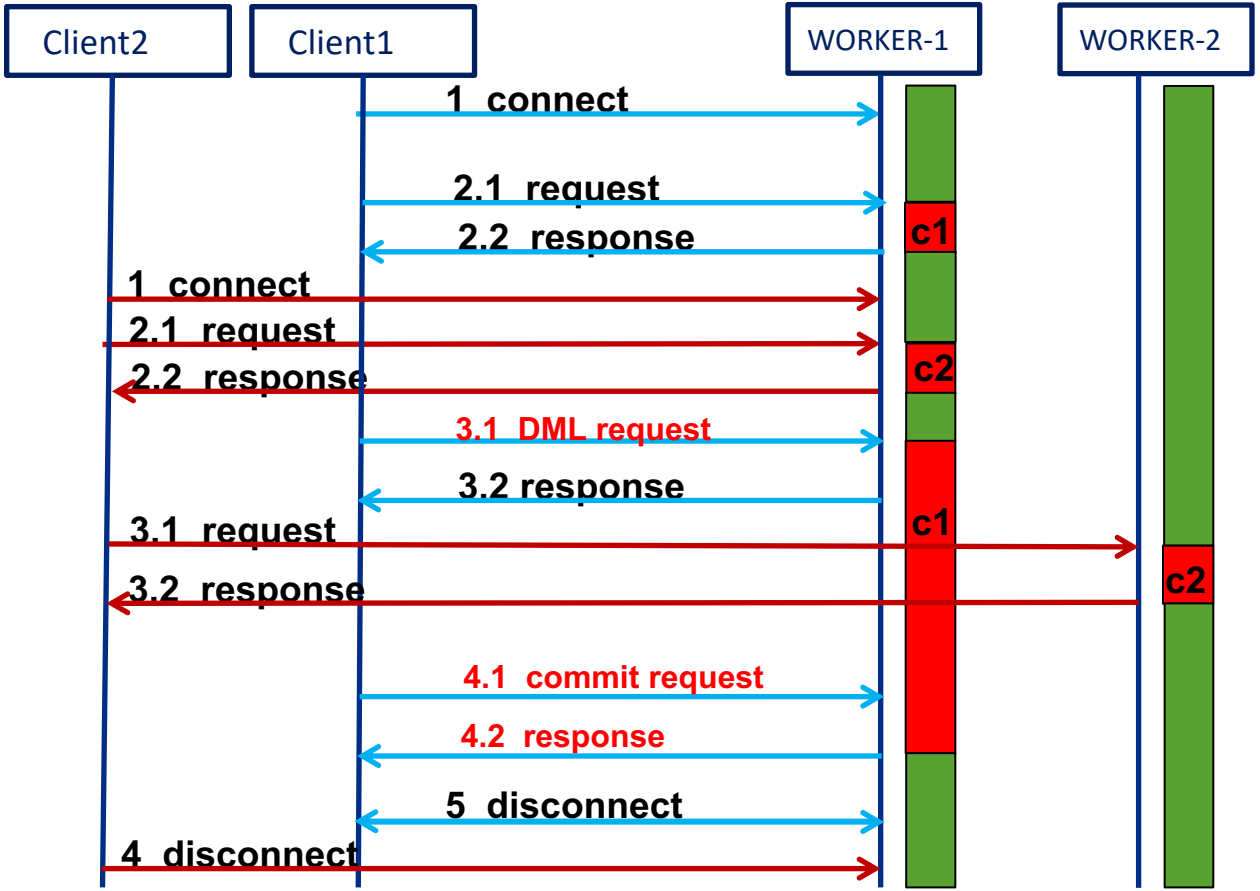


Mux is highly concurrent code with Several capabilities added year over year. Worker is simple Procedural code dealing with single DB connection. This architecture lends Itself easily to support other DB Technologies such as MySQL, PostgreSQL

How Multiplexer Helps?



Transaction Handling

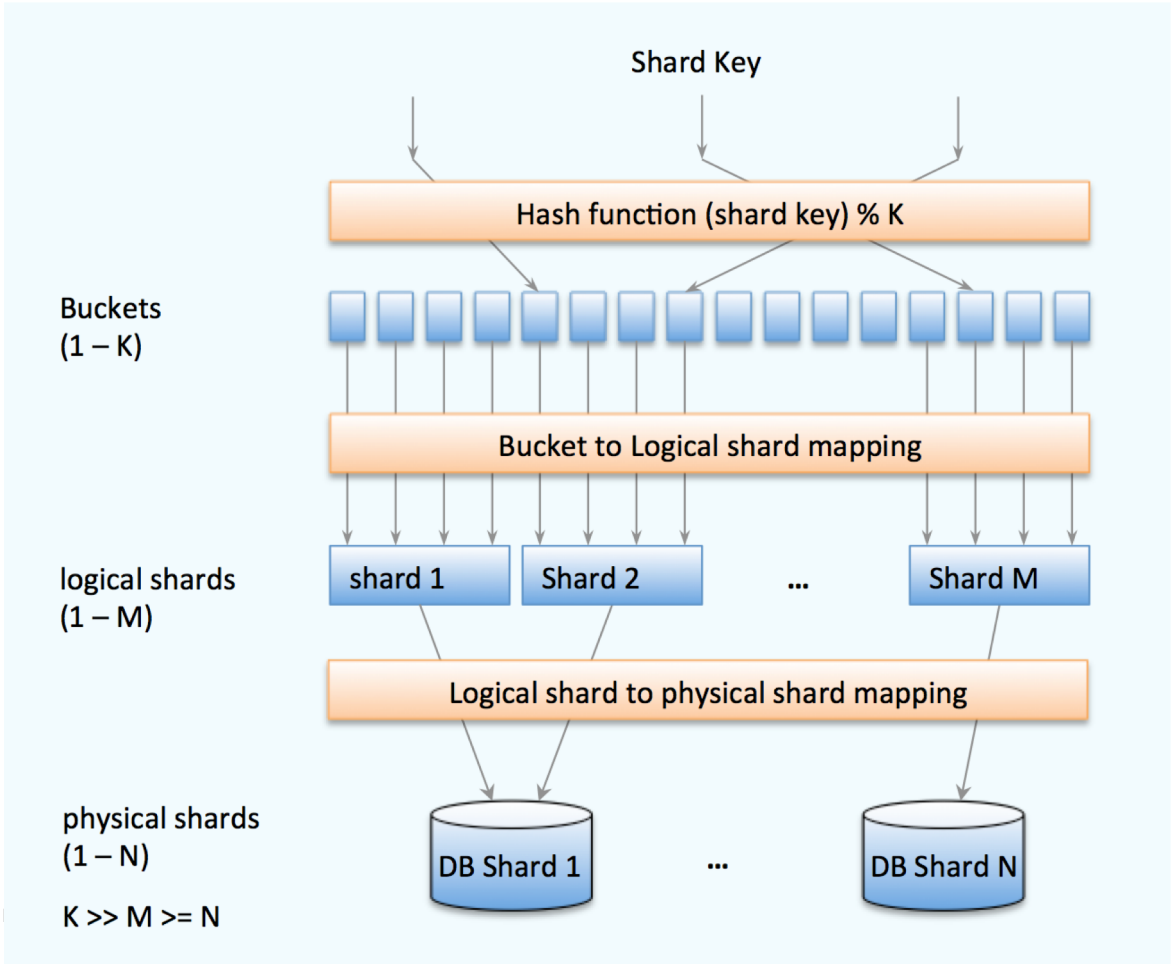


DML/Transaction Handling

DB horizontal Scaling aka Sharding

- “Entity” abstraction with well defined distribution key
- “Entity” can map to multiple tables
- Atomic transaction is restricted to single ”Entity” i.e. it can not span multiple entities (2PC is hard to scale)
- Cross ”entity” transactions managed as “business logic” or “workflows”
- Separation of “Scale-agnostic upper layer” (maps to app) and “Scale-aware lower layer” (maps to connection pool)
- “Scale-agnostic upper layer” (app) can only assume that an entity instance lives on a single machine and nothing more
- “Scale-aware lower layer” i.e. connection pool is aware of the actual placement of the entities and reports policy violations if any, to aid proactive discovery of bugs in app code
- Number of shards can be changed at connection pool layer requiring no change to the app

DB horizontal Scaling



State log

- Monitoring vs debugging
- State log is central to connection pool health monitoring


```
10/12/2017 01:19:45: vpool  init acpt  wait busy schd fnsh quce asgn idle  bklg strd
10/12/2017 01:19:45: cp.w    0   26    4    0    0    0    0    4   119   0    0
10/12/2017 01:19:45: cp.r    0   44    0    1    0    0    0    1   235   0    0
10/12/2017 01:19:46: cp.w    0   28    1    1    0    0    0    2   125   0    0
10/12/2017 01:19:46: cp.r    0   44    0    1    0    0    0    1   231   0    0
10/12/2017 01:19:47: cp.w    0   29    1    0    0    0    0    1   122   0    0
10/12/2017 01:19:47: cp.r    0   42    1    2    0    0    0    3   239   0    0
10/12/2017 01:19:48: cp.w    0   26    3    0    0    0    0    3   123   0    1
...
...
```

Resiliency Features

- Bouncer
- Surge Queue
- SQL Eviction
- Adaptive Surge Queue (CoDel Based)

Bouncer

- Configured by default
- Number of MUX processes determined by the system (No config)
- One of the MUX processes acts as Bouncer
  - Watches resource consumption by monitoring shared memory
  - When resources are all occupied, bounces new connection requests
  - Stops bouncing when system recovers

Surge Queue

- Configured by default
- MUX process maintains surge queue
    - Surge queue can grow to typically 30% of configured resources
    - Automatically backs off from accepting new connections
    - Small time out for items placed in surge queue
    - Large surge queue is counter-productive

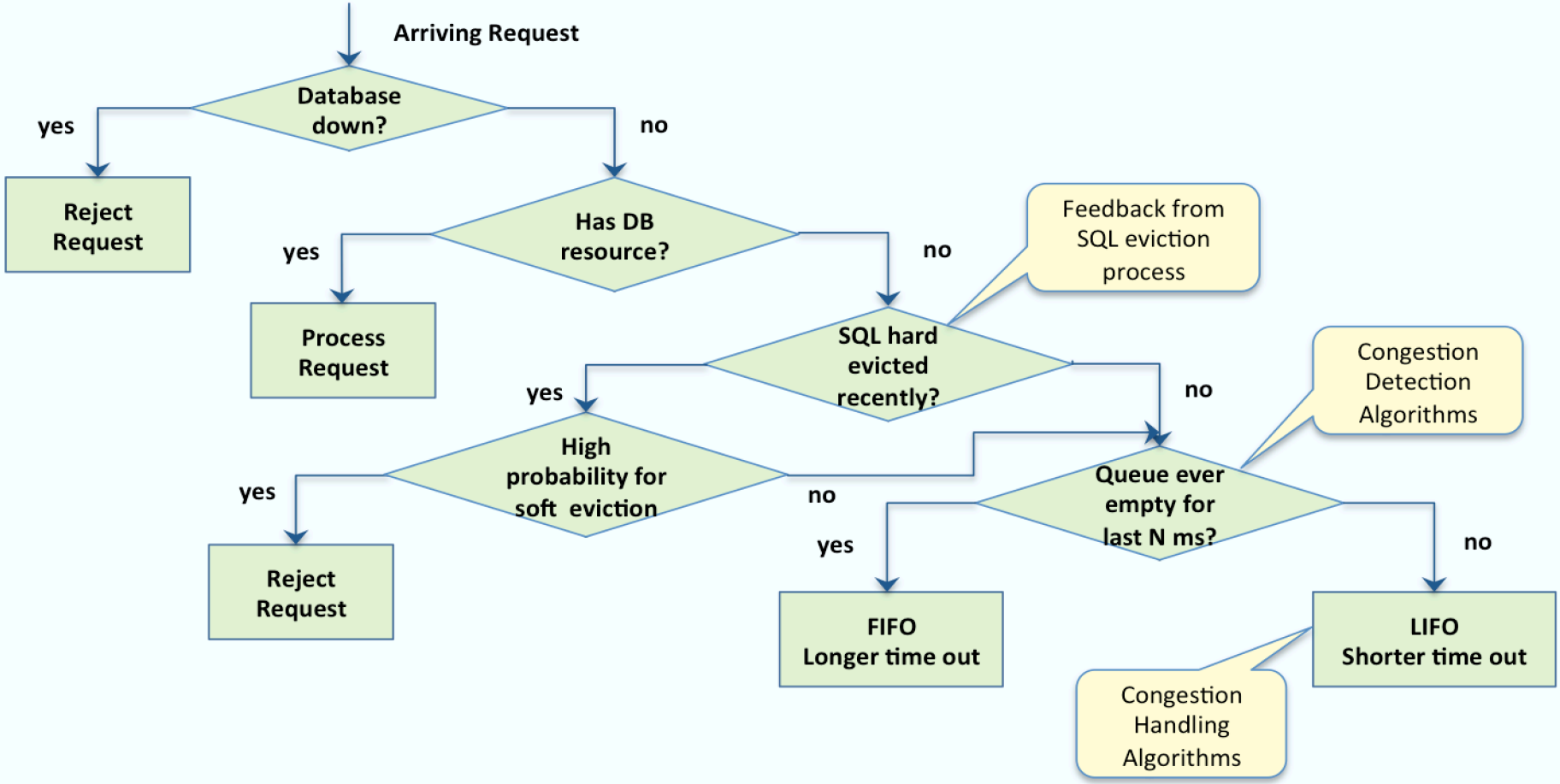
SQL Eviction

- Augment Shared memory to keep track of SQL execution times
- Evict longest running SQLs
- SQL Eviction =>
    - Evict SQL taking longer than x seconds
    - Do not nuke connection (do not cause another issue)
    - Post eviction recover Session
    - Throttle the rate of Eviction

SQL Eviction (CoDel Based)

- Minimize entropy during SQL eviction (anti-entropy)
  - **Hard-eviction** on Oracle (and “spread the bad news”)
  - **Soft-eviction** in Surge queue (based on the bad news)
  - Can’t soft-evict 100%
- Make SQL eviction **parameter-less**
  - Controlled Delay (**CoDel**) based scheme
  - Distinguish between good and bad queue
  - No need to configure tune thresholds on per service basis
- Fresh work before stale
  - Change request handling policy from FIFO to LIFO)

Adaptive Surge Queue Management



One Last Comment…

- We are planning to Open Source
  - JDBC driver
  - DB Proxy Server (Mux and Worker)
  - Support MySQL

Thank You