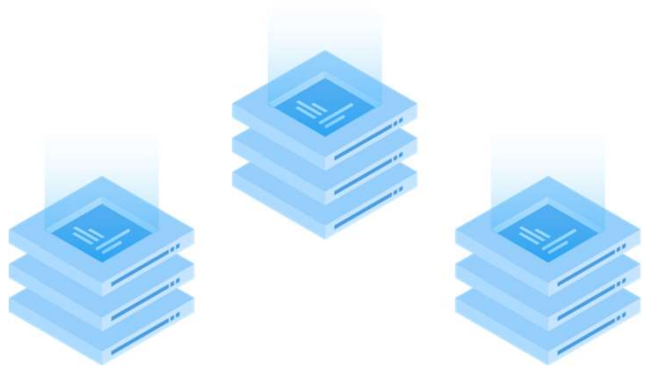
Concurrency In TiKV

Presented by Jay Lee



About Me

- TiKV Engineer at PingCAP
- Author of a few open source projects
  - raft-rs, grpc-rs, fail-rs, etc.
- Github: @busyjay

Layout

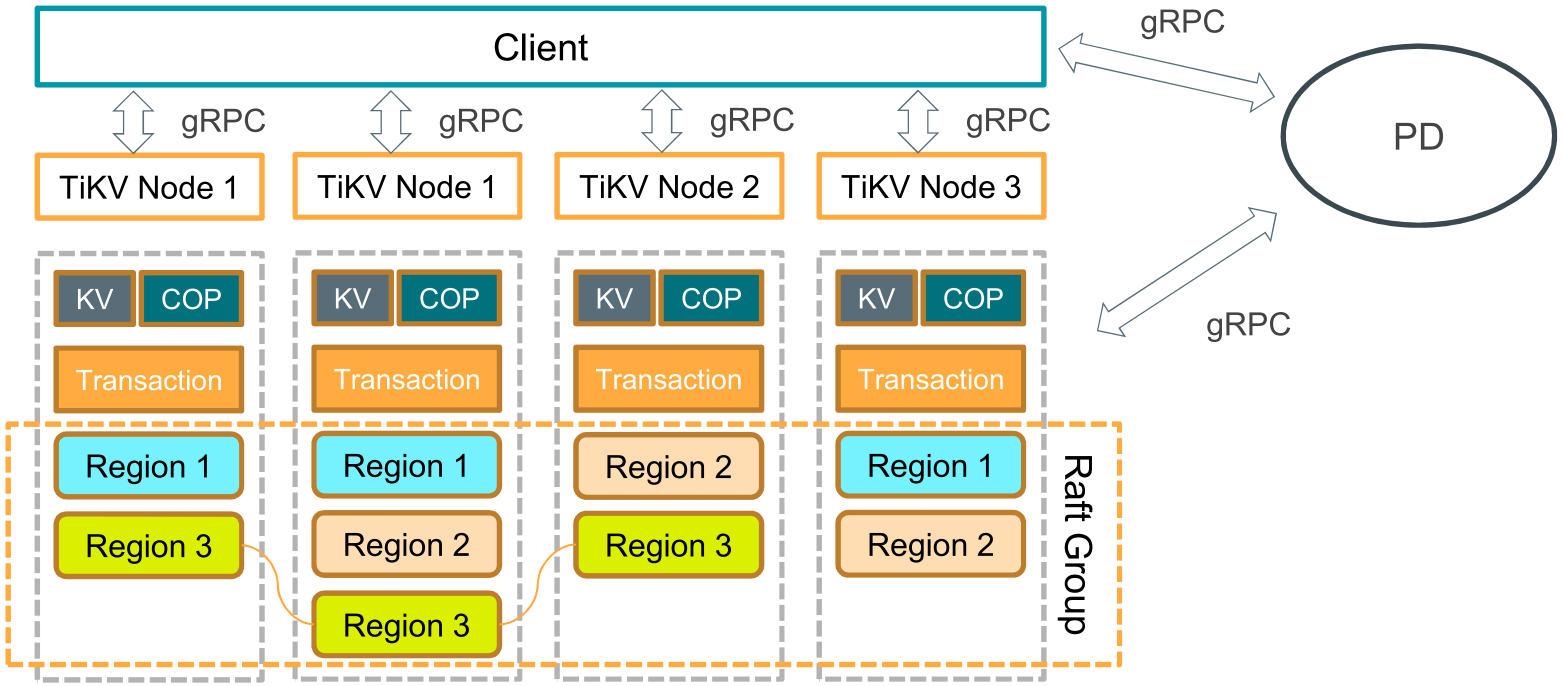
- What is TiKV?
- Classical thread model
- Batch system and Adaptive thread pool
- Metrics

What is TiKV

What is TiKV

- Distributed transactional key-value database
- Built in Rust and powered by Raft
- originally created to complement TiDB
    - distributed HTAP database speaks MySQL protocol
- incubating project of the CNCF

TiKV Architecture Overview



PingCAP

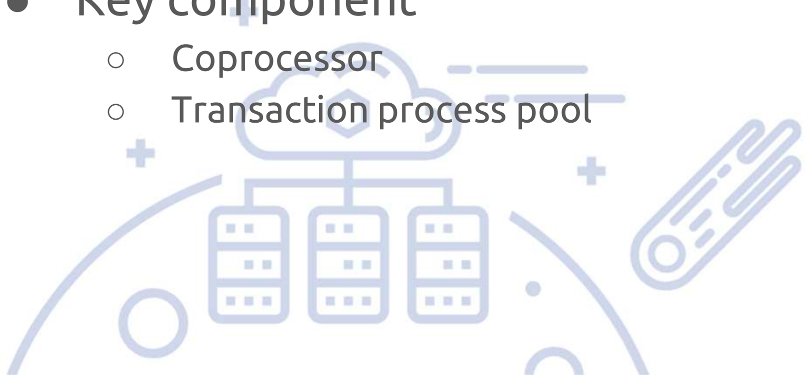
Classical Thread model

Classical thread model

- Naive queue based thread pool
- Worker thread
- Mio eventloop
- Callback/Message based asynchronization

Naive thread pool

- Mutex + Vec
  - Lack of MPMC at that time
  - Easy to implement
- Fixed number of thread count
- Basic customization support
  - Name
  - Stack size
- Key component
  - Coprocessor
  - Transaction process pool



Mio eventloop

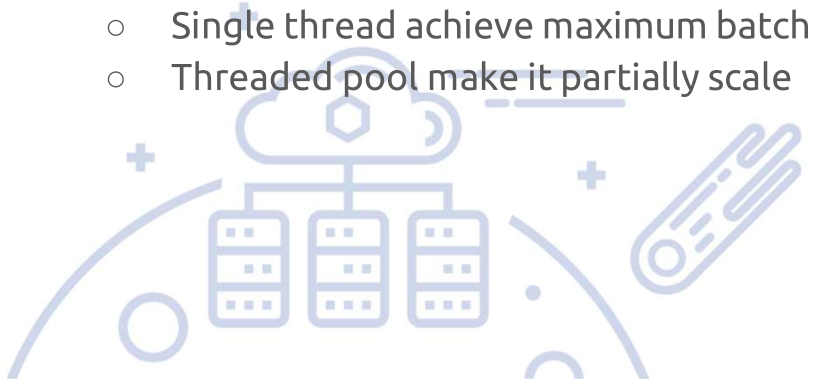
```
pub trait Handler: Sized {
    type Timeout;
    type Message: Send;
    fn ready(&mut self, event_loop: &mut EventLoop<Self>, token: Token, events: EventSet) { ... }
    fn notify(&mut self, event_loop: &mut EventLoop<Self>, msg: Self::Message) { ... }
    fn timeout(&mut self, event_loop: &mut EventLoop<Self>, timeout: Self::Timeout) { ... }
    fn interrupted(&mut self, event_loop: &mut EventLoop<Self>) { ... }
    fn tick(&mut self, event_loop: &mut EventLoop<Self>) { ... }
}
```

- Timer support
  - Time precision matters
  - Hashed wheel timer, very efficient
- Handy handler API
  - Very easy to implement pipeline + batch
- IO support
  - Supper awesome!
  - We don’t do network IO with mio though
- Channel is way faster than std channel!
- Key component
  - Raftstore

PingCAP

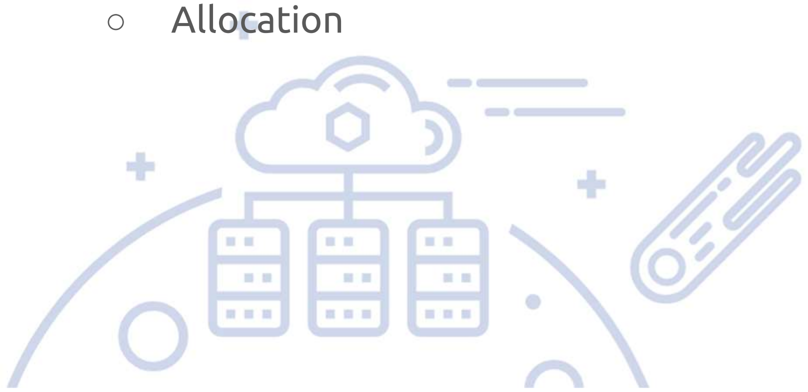
Worker thread

- A simple wrapper on std thread
- Event based handling
- Shutdown support, batch support
- Key component
    - All asynchronous routine tasks
    - Apply worker
- Powerful when combined with thread pool
    - Single thread achieve maximum batch execution
    - Threaded pool make it partially scale



Callback/Messages

- Use callback to schedule a message
- Why not just use futures?
  - No futures at that time!
- Props
  - Things are controlled more precisely
- Cons
  - Hard to read
  - Allocation

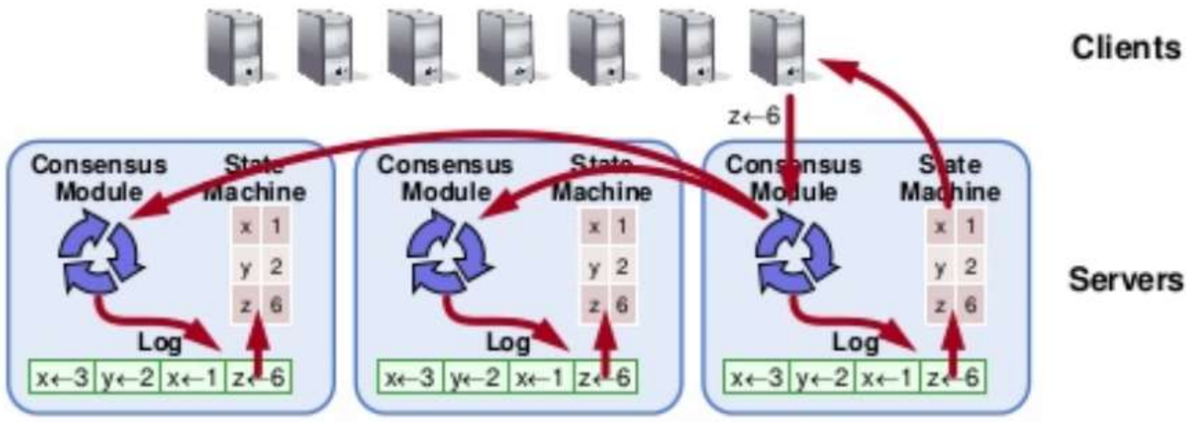


Review

- Single thread is efficient but can’t scale
- Naive thread pool is bad
    - Contention can waste CPU resources
    - Not friendly to CPU cache
- Callback/message is hard to write and read

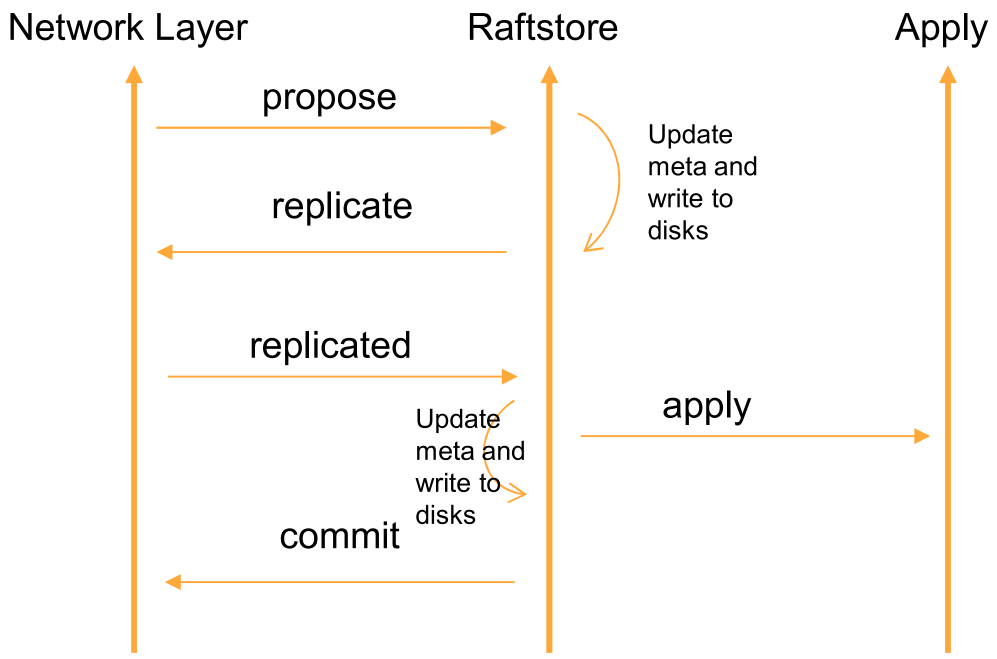
Batch System

What raftstore does



PingCAP

What raftstore does



Batch System

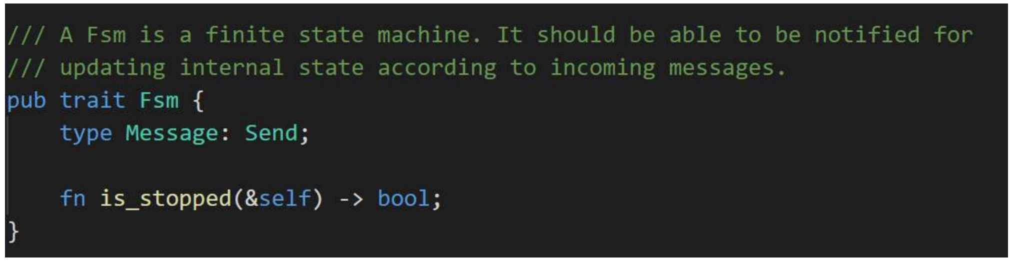
- Why single thread can also achieve good throughput
  - Pipeline + Batch
- Cons
  - Pause on a task can starve all queued tasks
  - Can’t scale on multiple cores machine
- How about just shard?
  - Hard to deal with hotspot
  - Raft groups can split and merged, hard to adjust shard

Batch System

- Every raft state machine is an actor
- They don’t share states and communicates with messages
- The problem becomes how to develop an efficient actor dispatcher

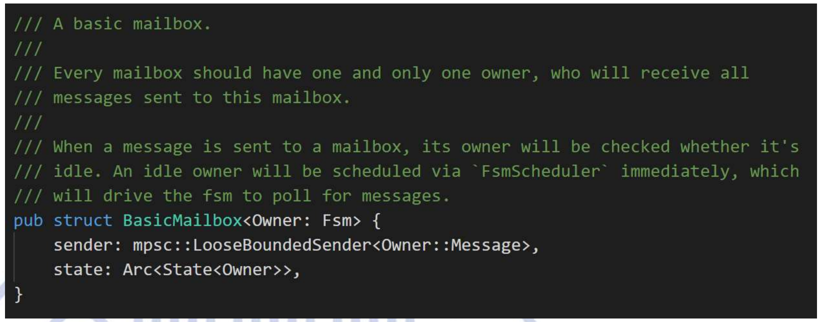
Batch system

- Actor stores in Mailbox
  - No hash query during handling
- Define actor and mailbox



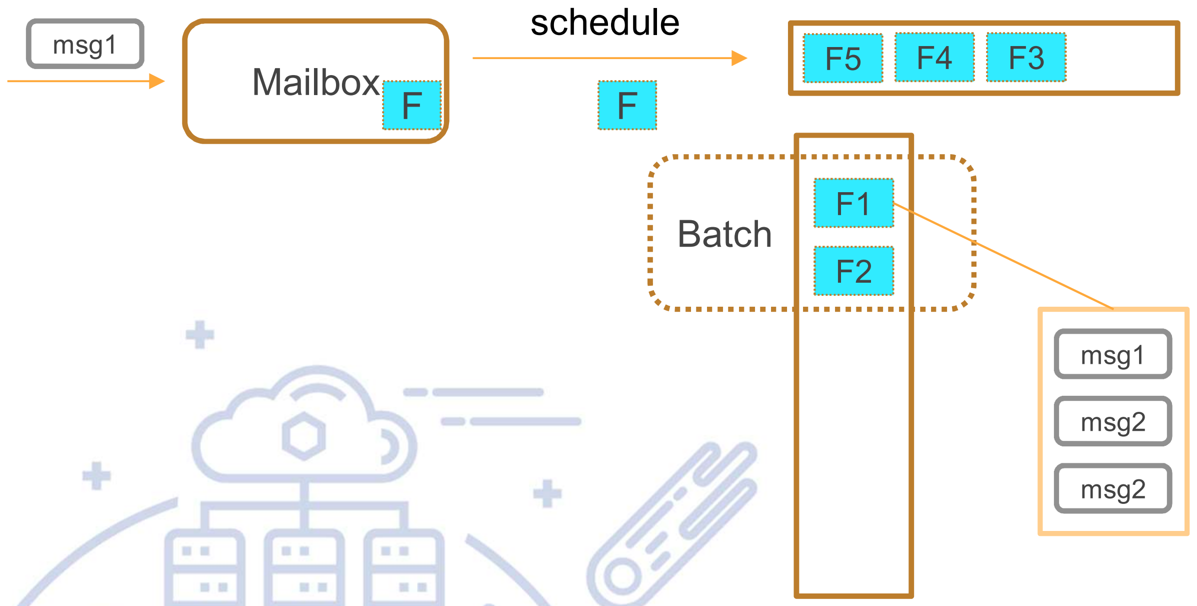
```rust
/// A Fsm is a finite state machine. It should be able to be notified for
/// updating internal state according to incoming messages.
pub trait Fsm {
    type Message: Send;

    fn is_stopped(&self) -> bool;
}
```



```rust
/// A basic mailbox.
///
/// Every mailbox should have one and only one owner, who will receive all
/// messages sent to this mailbox.
///
/// When a message is sent to a mailbox, its owner will be checked whether it's
/// idle. An idle owner will be scheduled via `FsmScheduler` immediately, which
/// will drive the fsm to poll for messages.
pub struct BasicMailbox<Owner: Fsm> {
    sender: mpsc::LooseBoundedSender<Owner::Message>,
    state: Arc<State<Owner>>,
}
```

Batch system



Batch system

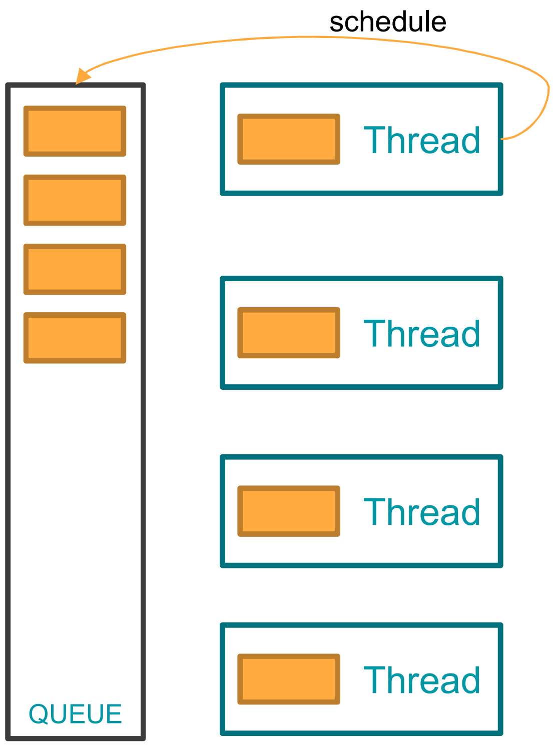
- Still achieve pipeline and batch
- Generic concepts can be shared
- Not coupling with FSM allows generic optimization
- Why not just use futures?
    - Hard to achieve batch, need to flush before next polling
- Latency reduced by 70%
- Same as old architect when only using one thread
- Key component
    - Raftstore
    - apply

Adaptive Thread Pool

Naive Thread Pool

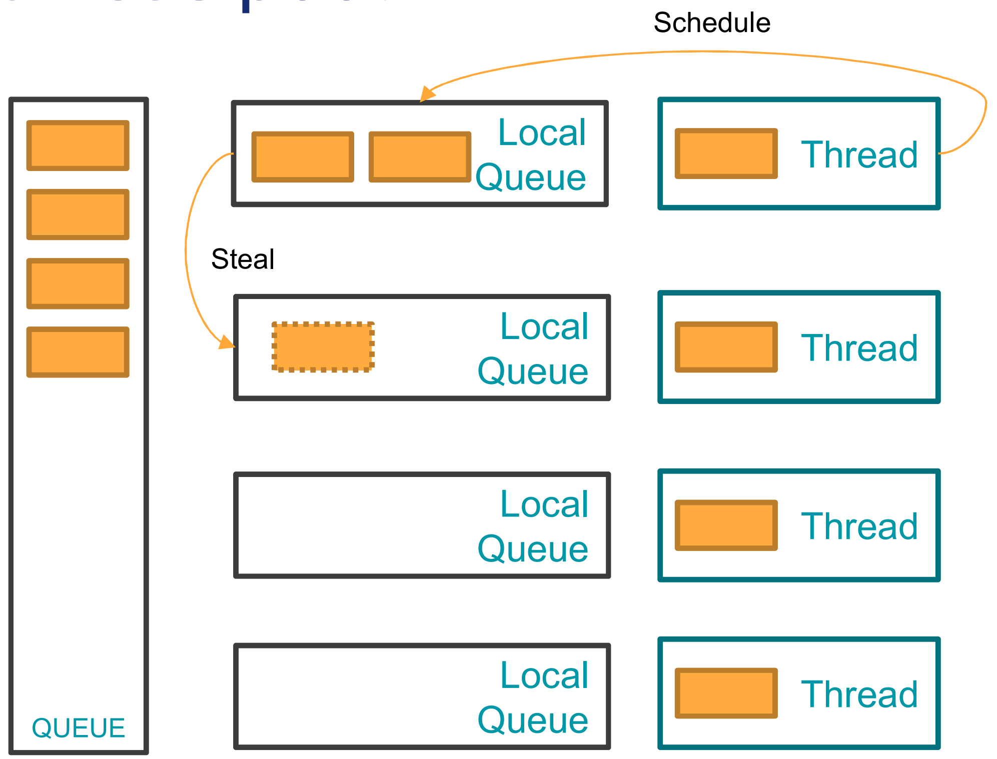
- Naive thread pool can waste CPU
- Optimized for greedy latency instead of throughput

Naive thread pool



PingCAP

Adaptive thread pool



Adaptive Thread Queue

- Active thread count can be configured
  - Balance between throughput and latency
  - Other threads are woken only when load is too high
- Avoid contention by pulling tasks to local in batch
  - FILO for local queue for better cache coherence
  - FIFO for work stealing to avoid contention
- Still experiment at https://github.com/busyjay/chocolates
- Tokio-threadpool is a good alternative
- Context is reduced by 5%, latency is reduced by 14% ~ 20%

Metrics

Metrics

- Metrics become problems with concurrency
    - It’s actively updated
    - Contention and cache coherence
- Lock free is not the silver bullet
    - Dimensions requires locks
    - It’s not efficient enough sometimes

Thread local Metrics

- Every metrics have a thread local copy
- Updates are flushed to global periodically
    - Requires support from thread and thread pool
- Work well if there are a few threads

Core local metrics

- Similar to thread local but every copy is bound to CPU core
- Works better if there are a lot of threads
  - Typically more than the number of CPU cores
- Aggregation can be done by querying a fixed size vector

Struct local metrics

- Every struct has a local copy of the metrics
- It’s extremely efficient
- Work poorly if the metrics are too large
    - Moving a struct in Rust is memory copy

Metrics as struct

- Static dimensions can be initialized as fields
- Query a dimension is done at compile time
    - No locks or hash
- https://github.com/pingcap/rust-prometheus

PingCAP KV

PingCAP

Thank You !