Futures in TiKV

Peng Qu

2019.04.19

CONTENTS

What's futures and what's TiKV

Basic Future usage

Basic Stream usage

Collect Future into Stream

Executor-Task style batch system

What’s Futures

What's Futures

- https://github.com/rust-lang-nursery/futures-rs

  Traits including Future and Stream

  mpsc/oneshot channels

  Utilizes like executor/task

- https://github.com/tokio-rs/tokio

  High level components in Future

  tokio_threadpool, which can spawn Futures

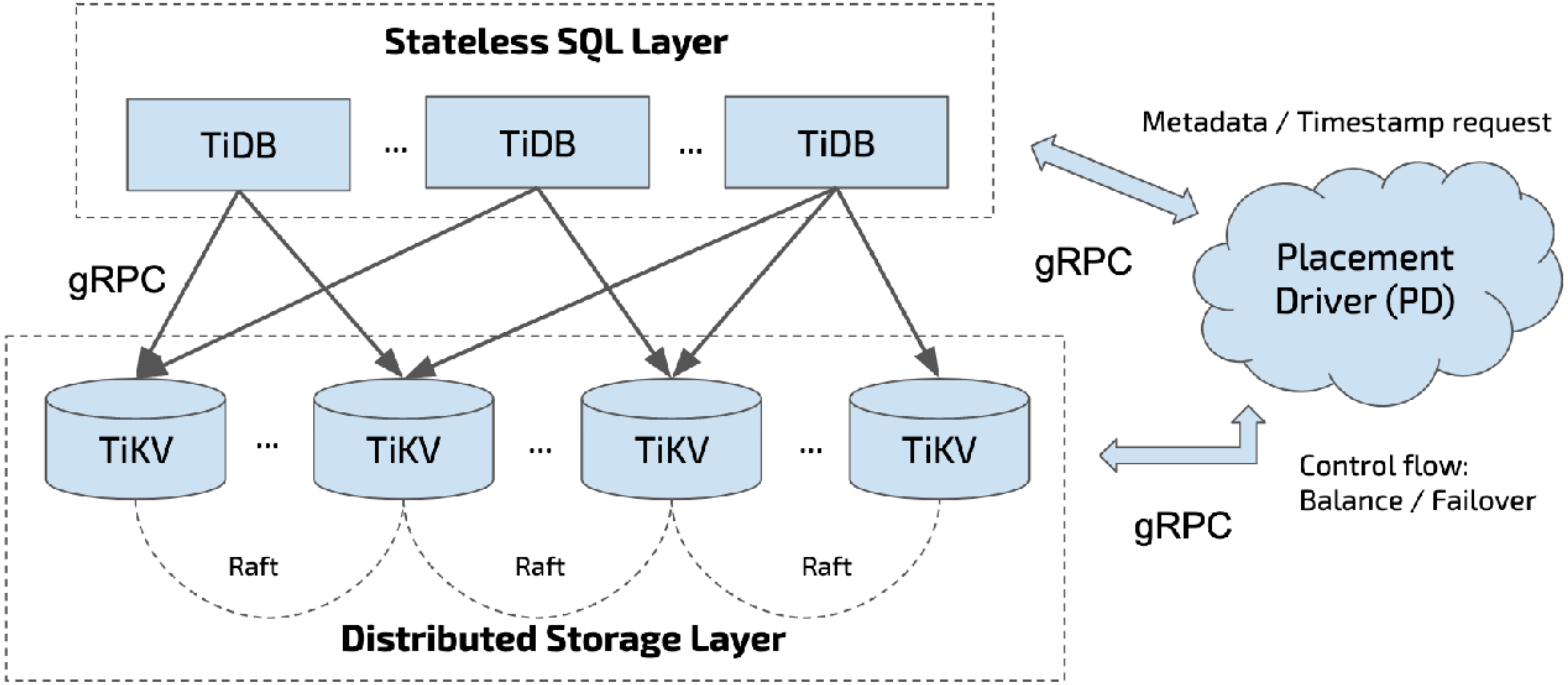
```rust
use std::sync::mpsc as std_mpsc;

use futures::sync::mpsc;
use futures::{stream, Future, Sink, Stream};
use tokio_threadpool::Builder;

fn main() {
    let input = (0..4096).collect::<Vec<u32>>();
    let pool = Builder::new().pool_size(1).build();
    let (producer, consumer) = mpsc::channel(128);
    pool.spawn(
        stream::iter_ok::<_, ()>(input)
            .forward(producer.sink_map_err(|_| println!("sink error")))
            .map(|_| println!("produce finish")),
    );
    let (tx, rx) = std_mpsc::sync_channel(128);
    pool.spawn(
        consumer.for_each(move |i| Ok(tx.send(i).unwrap()))
            .map(|_| println!("consume finish")),
    );
    assert_eq!(rx.into_iter().count(), 4096);
    println!("finished");
}
```

In the example, the producer and the consumer run in one thread

What’s TiKV

RUSTCON ASIA 2019

What's TiKV



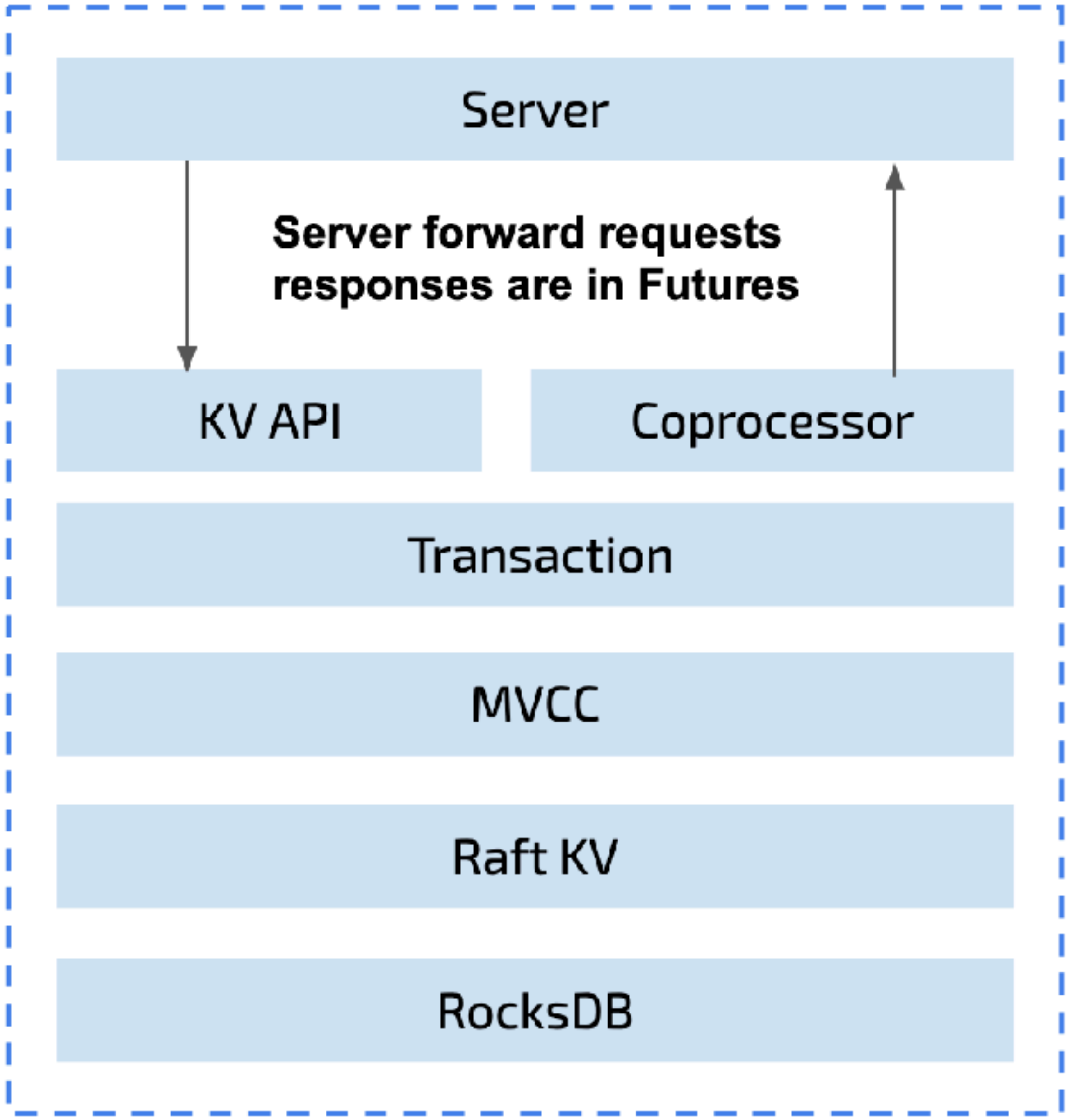
TiKV is a distributed KV storage engine. Data is replicated with Raft protocol

TiKV' internal & where are futures

Server layer retrieves requests, then forward them. Responses are in Futures, which are spawned in gRPC threads.

For some duplex streaming requests, there is also a usage in server layer that collects Futures into a Stream.

RaftKv appends users' write into Raft logs. There is a batch system inspired by Futures in it.



TiKV Layer Structure

Basic Future usage in TiKV

Basic Future usage in TiKV

```rust
impl tikvpb_grpc::Tikv for Service {
    fn kv_get(&mut self, ctx: RpcContext<'_>, req: GetRequest, sink: UnarySink<GetResponse>) {
        // Forward the request to background thread pool, and get a response future.
        let f = future_get(&self.storage, req);
        // Send the response back to the client.
        let f = f.and_then(|res| sink.success(res).map_err(Error::from));
        // Spawn the future into gRPC threads.
        ctx.spawn(f);
    }
}
```

The key is how `spawn` is implemented

Basic Future usage in TiKV

```
impl<'a> RpcContext<'a> {
    pub fn spawn<F>(&self, f: F:Future<Item = (), Error = ()> + Send + 'static>) {
        // Spawn the future so that we can get its task.
        let s = executor::spawn(Box::new(f) as BoxFuture<_, _>);
        // Create a `Notify` which will kick the bounded completion queue if
        // Notify::notify is called.
        let notify = Arc::new(SpawnNotify(self.kicker));
        // Pull the future in current thread.
        poll(&notify, false);
    }
}
```

Basic Stream usage in TiKV

Basic Stream usage in TiKV

```rust
impl tikvpb_grpc::Tikv for Service {
    fn coprocessor_stream(
        &mut self,
        ctx: RpcContext<'_>,
        req: Request,
        sink: ServerStreamingSink<Response>,
    ) {
        let stream = self
            .cop
            .parse_and_handle_stream_request(req, Some(ctx.peer()))
            .map(|resp| (resp, WriteFlags::default().buffer_hint(true)))
            .map_err(|e| GrpcError::RpcFailure(RpcStatus::unknown));
        ctx.spawn(sink.send_all(stream));
    }
}
```

A server streaming example, the key is the sink

Basic Stream usage in TiKV

```rust
impl tikvpb_grpc::Tikv for Service {
    fn raft(
        &mut self,
        ctx: RpcContext<'_>,
        stream: RequestStream<RaftMessage>,
        sink: ClientStreamingSink<Done>,
    ) {
        let ch = self.ch.clone();
        ctx.spawn(
            stream.for_each(move |msg| ch.send_raft_msg(msg).map_err(Error::from))
                .then(|res| {
                    let status = match res {
                        Err(e) => RpcStatus::new(RpcStatusCode::Unknown, Some(msg)),
                        Ok(_) => RpcStatus::new(RpcStatusCode::Unknown, None),
                    };
                    sink.fail(status)
                }),
        );
    }
}
```

A client streaming example, the key is the stream

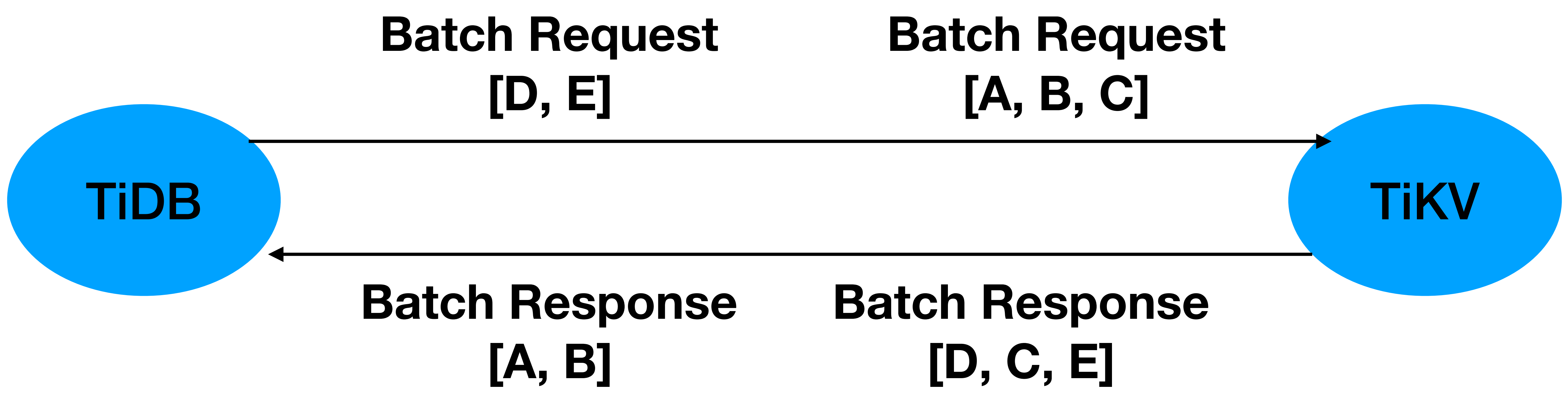
Basic Stream usage in TiKV (batch)

```
impl RaftClient {
    pub fn send(&mut self) { }
    pub fn flush(&mut self) {
        for conn in self.conns.values_mut() {
            // Get a notifier and do the noitfy explicitly.
            if let Some(notifier) = conn.stream.get_notifier() {
                if !self.grpc_thread_load.in_heavy_load() {
                    notifier.notify();
                    continue;
                }
                let wait = self.cfg.heavy_load_wait_duration.0;
                let _ = self.pool.spawn(
                    self.timer
                        .delay(Instant::now() + wait)
                        .inspect(move |_| notifier.notify()),
                );
            }
        }
    }
}
```

The BatchSender is in tikv/components/tikv_util/src/mpsc/batch.rs

Collect Futures into Stream

Collect Futures into Stream



- A batch request contains many little requests
- TiKV handles every little request in a Future
- TiKV needs to collect all Futures into a Stream

Collect Futures into Stream

```
fn collect_futures_into_stream(sink: Sender<Response>, f: impl Future<Item = Response, Error = ()>) {
    // Where to spawn `f`?
    let f = f.and_then(move |resp| sink.send(resp));
    // Don't need to spawn it into any threads, just set a callback.
    poll_future_notify(f);
}

fn poll_future_notify<F: Future<Item = (), Error = ()> + Send + 'static>(f: F) {
    let spawn = Arc::new(Mutex::new(Some(executor::spawn(f))));
    let notify = BatchCommandsNotify(spawn);
    notify.notify(0);
}

impl<F: Future<Item = (), Error = ()>> Notify for BatchCommandsNotify<F> {
    fn notify(&self, id: usize) {
        let n = Arc::new(self.clone());
        let mut s = self.0.lock().unwrap();
        match s.as_mut().map(|spawn| spawn.poll_future_notify(&n, id)) {
            Some(Ok(Async::NotReady)) | None => return,
            _ => *s = None,
        }
    }
}
```

Executor-Task style batch system

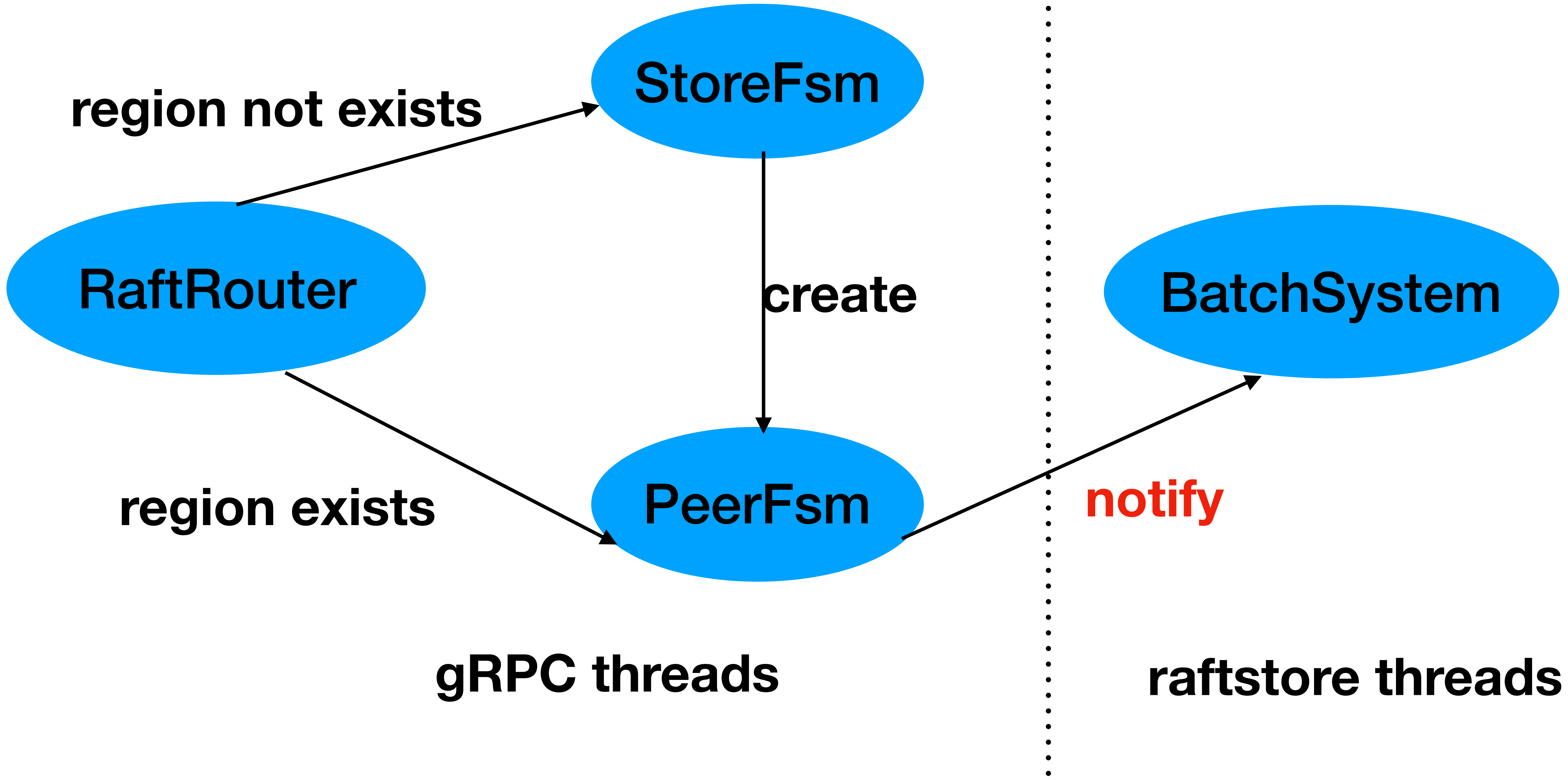
Executor-Task style batch system

- There are thousands regions per TiKV.
- Handle Raft messages could be a bottleneck.
- A mediocre implementation is easy to write, but
  - some regions could be hungry
  - hard to consolidate a write batch
  - hard to refactor old code
- So we need a better implementation…

```rust
use std::sync::mpsc as std_mpsc;

use futures::sync::mpsc;
use futures::{stream, Future, Sink, Stream};
use tokio_threadpool::Builder;

fn main() {
    // Suppose we have 4 threads to handle Raft messages.
    let executor = Builder::new().pool_size(4).build();
    // Suppose we have 10k regions on a TiKV.
    let mut regions = Vec::with_capacity(10000);
    for _ in 0..10000 {
        let (_, rx) = mpsc::channel(1024);
        regions.push(rx);
        executor.spawn(rx.for_each(|msg| handle_raft_msg(msg)));
    }
}
```

Executor-Task style batch system

- Fsm means Finite-State Machine.
- PeerFsm will notify BatchSystem if need.
- In BatchSystem:
  - many background threads
  - batch strategy to avoid hungry
  - hung a region if it's temporarily unavailable



Executor-Task style batch system

```
// RaftRouter will call `MailBox::send`and `MailBox::notify` to forward
// Raft messages and notify downstream compponents to handle them.
impl BasicMailBox {
    // FsmScheduler can schedule a PeerFsm to a BatchSystem to run.
    fn notify(&self, scheudler: &FsmScheduler) {
        let prev_state = self.state.compare_and_swap(IDLE, NOTIFIED);
        if prev_state == IDLE {
            // IDLE means the PeerFsm is not running. So schedule it.
            let mut owner = unsafe { Box::from_raw(self.data) };
            owner.set_mailbox(Cow::Borrowed(self));
            scheduler.schedule(owner);
        }
    }
    // Called in BatchSystem to pu `fsm` back to the mailbox.
    fn release(&self, fsm: Box<Owner>) {
        assert!(self.data.swap(Box::into_raw(fsm)).is_null());
        let prev_state = self.state.compare_and_swap(NOTIFIED, IDLE);
        if prev == DROP {
            // It's destroyed instead of released, deconstruct it.
            drop(unsafe { Box::from_raw(self.data) });
        }
    }
}
```

Implementation of notify and release

Executor-Task style batch system

- A batch contains many regions, so no hungry.
- It's easy to consolidate a write batch from many regions.
- More easily to be integrated into old code

```
impl BatchSystemPoller {
    // The loop of BatchSystem's background threads.
    fn poll(&mut self) {
        // `Batch` is like a Vector of Owner.
        let mut batch = Batch::with_capacity(self.max_batch_size);
        self.fetch_batch(&mut batch);
        while !batch.is_empty() { // It's empty means the region is destroyed.
            let mut raft_readies = Vec::new();
            for peer in batch.iter() {
                raft_readies.push(self.handle_raft_messages(&peer));
                if peer.temporarily_unavailable() {
                    batch.release(peer);
                }
            }
            self.handle_raft_readies(raft_readies);
            self.fetch_batch(&mut batch);
        }
    }
}
```

Implementation of BatchSystem's loop

THANKS