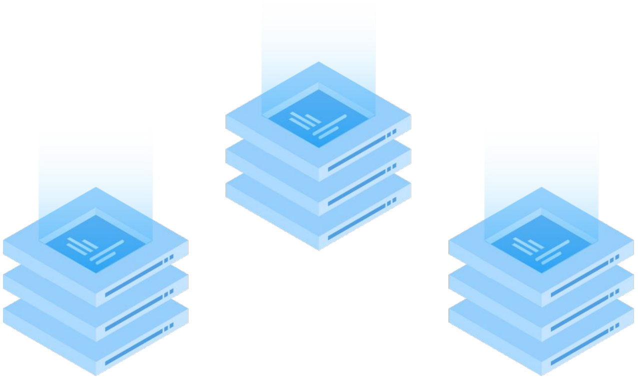
Rust network programming: Futures and gRPC

Presented by
shentaining@pingcap.com

PingCAP TiDB



About me

- 沈泰宁
- R&D Engineer @ PingCAP/TiKV
- Maintainer
    - tikv
    - grpc-rs
    - rust-prometheus
    - ...

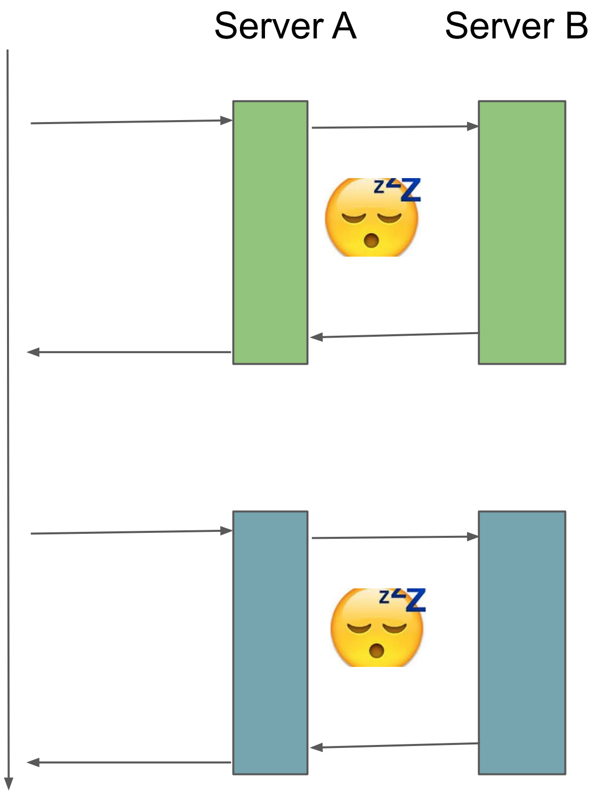
Agenda

- Async programming
- Futures
- gRPC
- Combine Futures and gRPC

PingCAP TiDB

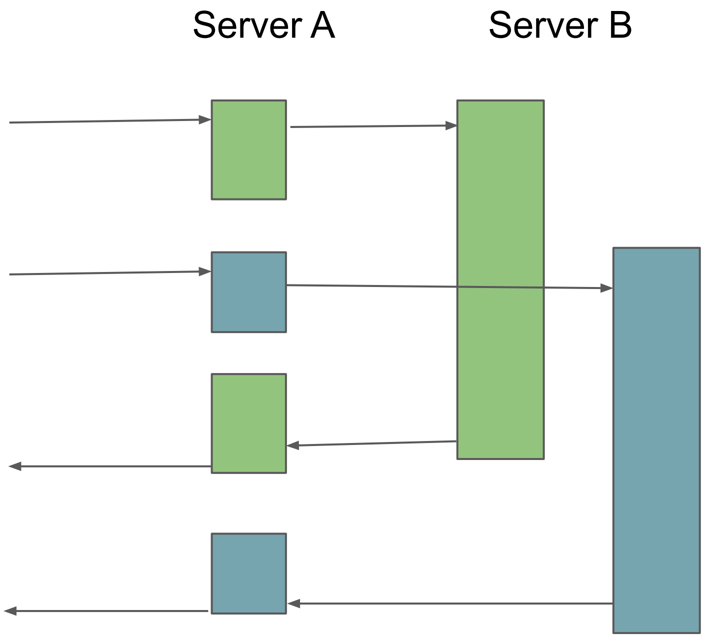
Part I - Async programming

Why Sync?



- Pros
  - Simple
  - High effective for low load service
  - Use multi threads for concurrency
- Cons
  - Block
  - Thread is heavy and wastes resources.
  - Switching thread is inefficient

Why Async?



- Pros.
  - No blocking
  - High performance
- Cons
  - Logic is split
  - Complex

Callback Hell?

```
let r = Arc::new(AtomicU8::new());
do_async_a(|a| {
   do_async_b(|b| {
       do_finish(|res| {
           r.set(res);
       })
   })
})
```

Coroutine Make It Easy

- C++: boost coroutine, libco, etc...
- Python: yield, greenlet, etc…
- Golang: Goroutine

For Golang:

```
go func(ch: channel) {
    // do something
    ch <- res
}(ch)
r := <- ch
```

Part II - Future

Zero-cost asynchronous programming in Rust

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

The Ergonomic Way

```
let future_count = do_async_a()
    .then(|res| do_async_b(res))
    .then(|res| do_finish(res));
let r = future_count.wait().unwrap();
```

```
let r = Arc::new(AtomicU8::new());
do_async_a(|a| {
   do_async_b(|b| {
       do_finish(|res| {
           r.set(res);
       })
   })
})
```

Under the Hood

```
pub trait Future {
    type Item;
    type Error;
    // Query this future to see if its value has become available.
    fn poll(&mut self) -> Result<Async<Self::Item>, Self::Error>;
    // Block the current thread until this future is resolved.
    fn wait() -> result::Result<Self::Item, Self::Error> {
        loop { self.poll(); return if it ready or error }
    }
}
pub enum Async<T> {
   Ready(T),
   NotReady,
}
```

PingCAP TiDB

Examples

```
let f = ok::<u32, u32>(1);
assert_eq!(f.wait().unwrap(), 1);

let mut f = empty::<u32, u32>();
assert_eq!(f.poll(), Ok(Async::NotReady));
```

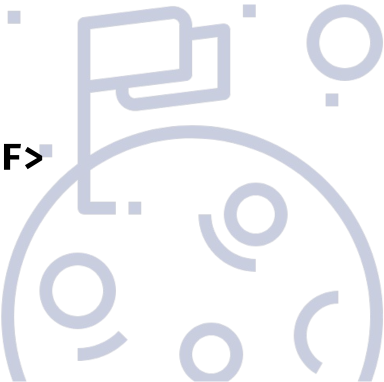
https://play.rust-lang.org/?edition=2018&gist=bf84b6eedd25603686a0714b063b9024

Combinator

```
let future = do_async_a()
    .then(|res| do_async_b(res))
    .then(|res| do_finish(res));
```

```
trait Future {
    fn then<F, B>(self, f: F) -> Then<Self, B, F>
    where
        F: FnOnce(Result<Self::Item, Self::Error>) -> B,
        B: IntoFuture { … }

    fn map<F, U>(self, f: F) -> Map<Self, F>
    where
        F: FnOnce(Self::Item) -> U { … }
    ...
}
```



Synchronization

- oneshot
  - Single producer/Single consumer
- mpsc
  - Multi producers/Single consumer

```rust
let (tx, rx) = oneshot::channel::<u32>();
thread::spawn(move || {
    thread::sleep_ms(3000);
    tx.send(1).unwrap();
});
assert_eq!(rx.map(|x| x + 1).wait().unwrap(), 2);
```

Stream

```
pub trait Stream {
    type Item;
    type Error;
    // Attempt to pull out the next value of this stream.
    // Ready(Some) means next value is on the stream
    // Ready(None) means the stream is finished
    fn poll(&mut self) -> Result<Async<Option<Self::Item>, Self::Error>>;
}
```

Sink

```
pub trait Sink {
    type SinkItem;
    type SinkError;
    fn start_send(self, item: Self::SinkItem)
       -> StartSend<Self::SinkItem, Self::SinkError>;
    fn poll_complete(&mut self) -> Result<Async<()>, Self::SinkError>;
    fn close(&mut self) -> Result<Async<()>, Self::SinkError>;
}
```

Task

- If the future is not ready?

  ```
  let handle = task::current();
  ```

- If the event of interest occurs?

  ```
  handle.notify();
  ```

- What to do after notify?

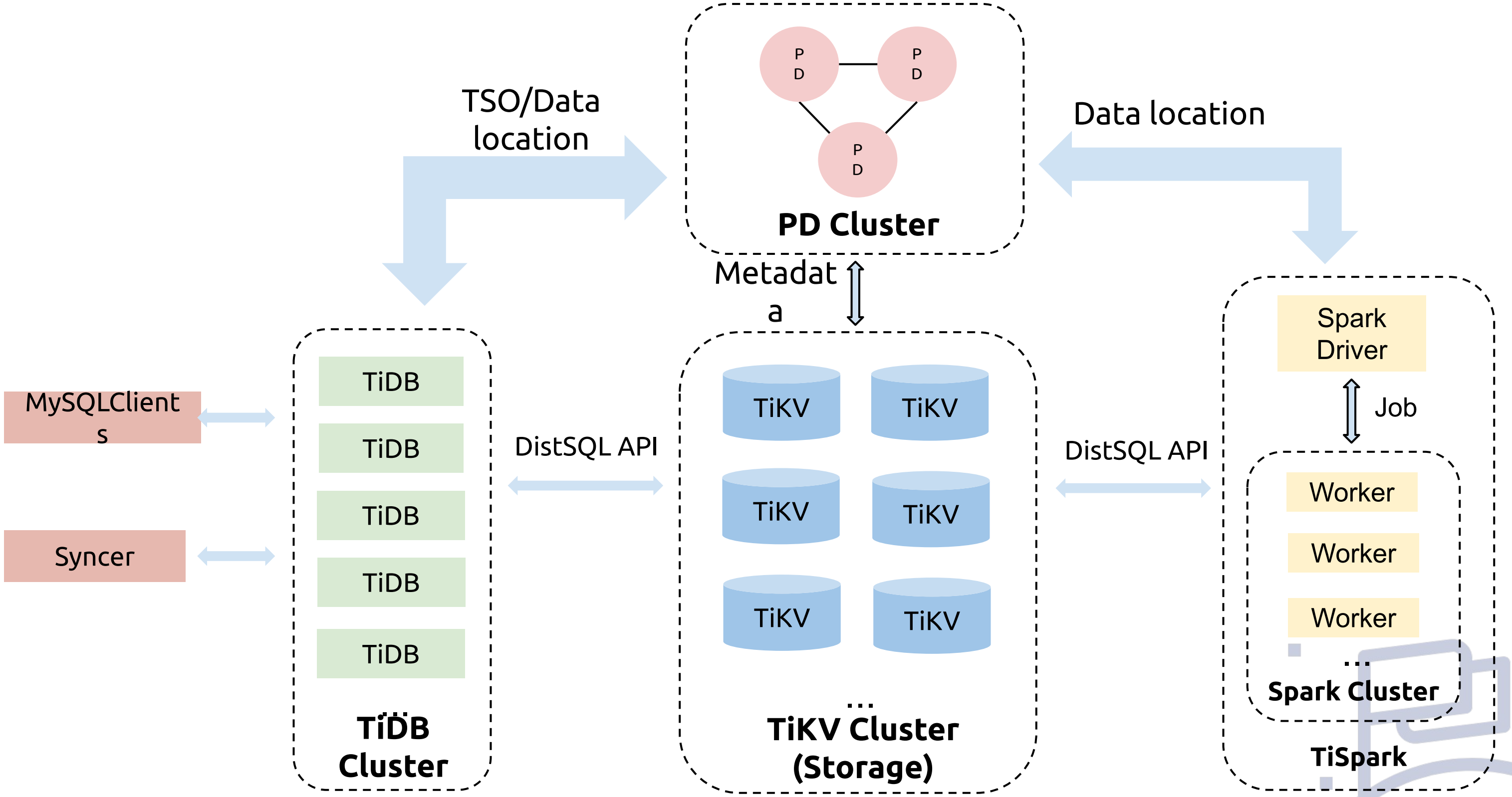
  ```
  executor.poll(f);
  ```

Part III - gRPC

A high-performance, open-source universal RPC framework

https://grpc.io/

gRPC in TiDB?



TSO/Data location
Data location
PD
PD
PD
PD Cluster
Metadata
MySQLClients
Syncer
TiDB
TiDB
TiDB
TiDB
TiDB
...
TiDB Cluster
DistSQL API
TiKV
TiKV
TiKV
TiKV
TiKV
TiKV
...
TiKV Cluster (Storage)
DistSQL API
Spark Driver
Job
Worker
Worker
Worker
...
Spark Cluster
TiSpark

Why

- Protobuf
- Widely used
- Supported by many languages
- Benefit from HTTP/2
- Rich interfaces

HTTP/2

- Binary protocol
- Multiplexing
- Priority
- Flow Control
- HPACK

C gRPC Key Concept

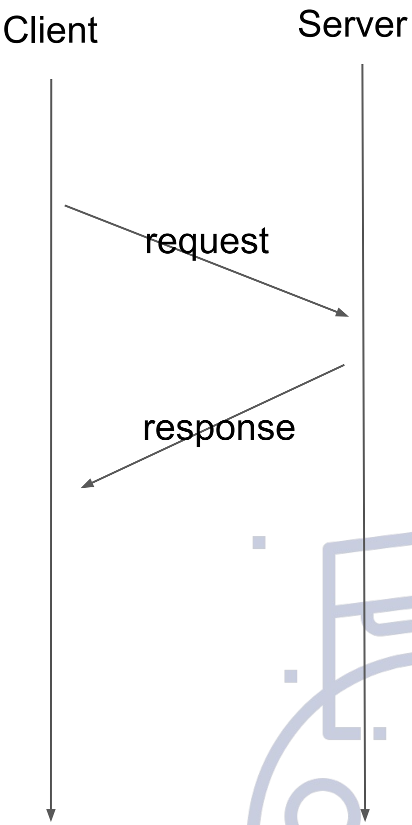
- Call: RPC call, Unary, Client, Server and Duplex streaming
- Channel: Connection
- Server: Sever to register the service
- Completion Queue: Drive RPC

Workflow

- Create completion queue
- Create client channel
- Create a call from the channel
- Start operations in batch of the call with a **tag**
- **Poll** completion queue to perform the call and return the event contains a **tag**
- Use tag to do something...

Unary

- Client sends request
- Server replies response

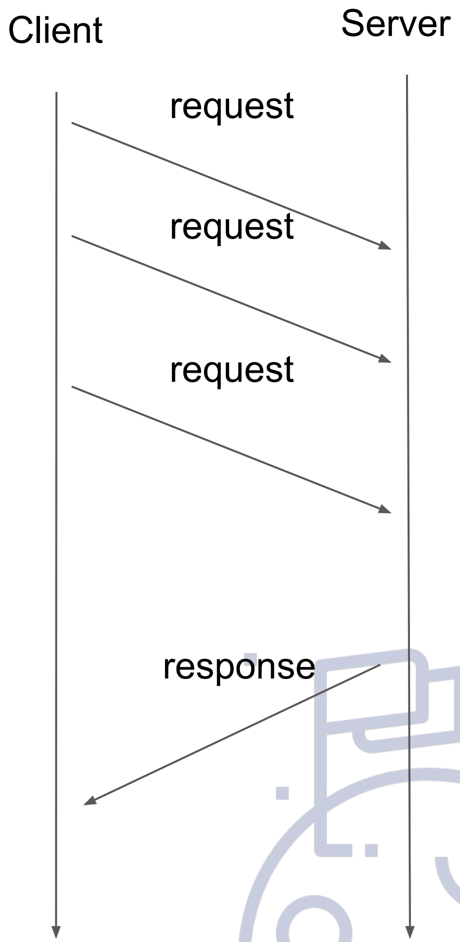


Unary: Pseudo Flow

```
> Client
let future = unary(service, method, request);
let response = future.wait();
> Server
fn on_unary(context, request, response_sink) {
    context.spawn(|| {
        // do something with request
        response_sink.send(response)
    });
}
```

Client Streaming

- Client sends request 1
- ……
- Client sends request N
- Server replies response

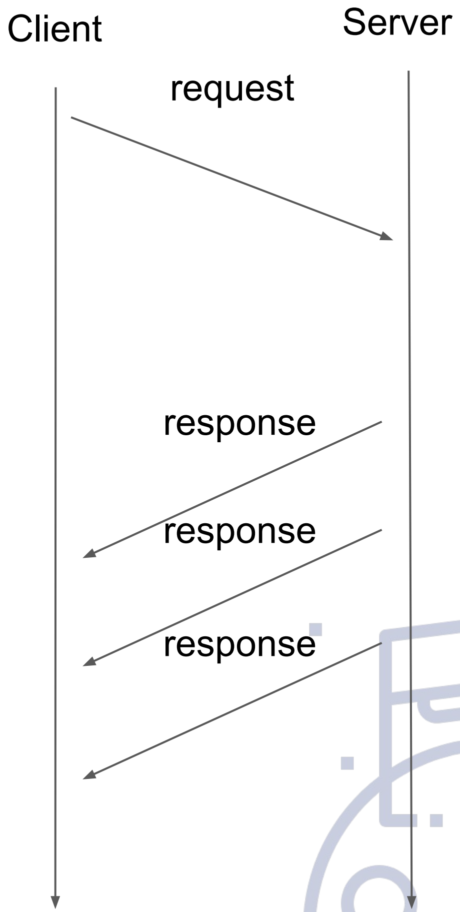


Client Streaming: Pseudo Flow

```
> Client
let (mut sink, resp) = client_streaming(service, method);
loop { sink = sink.send(request).wait().unwrap(); }
let response = resp.wait();
> Server
fn on_client_streaming(context, request_stream, response_sink) {
   context.spawn(request_stream
       .for_each(|request| { /* .. */ })
       .and_then(|res| { response_sink.send(res) }));
}
```

Server Streaming

- Client sends request 1
- Server replies response 1
- ……
- Server replies response N

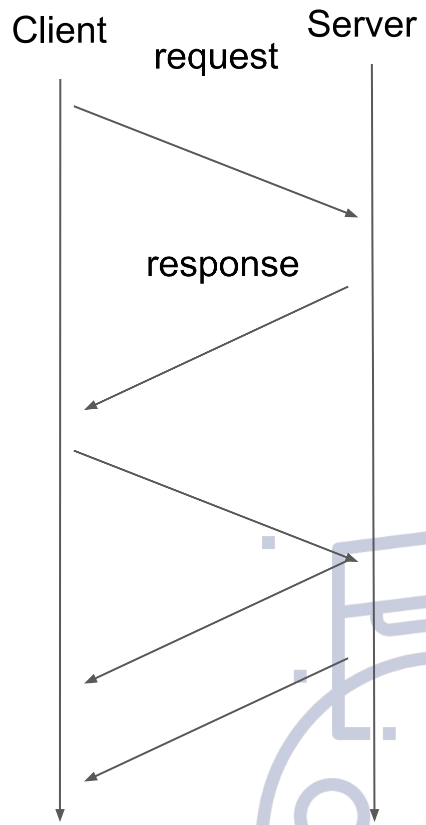


Server Streaming: Pseudo Flow

```
> Client
let resp_stream = server_streaming(service, method, request);
resp_stream.for_each(|response| { /* */ }).wait();
> Server
fn on_server_streaming(context, request, response_sink) {
    let future = response_sink.send_all(responses);
    context.spawn(future);
}
```

Duplex Streaming

- Client sends request 1
- Server replies response 1
- ……
- Client sends request N
- Server replies response N



Duplex Streaming: Pseudo Flow

```
> Client
let (sink, stream) = duplex_streaming(service, method);
sink.send_all(requests);
stream.for_each(|response| { /* */ });
> Server
fn on_duplex_streaming(context, request_stream, response_sink) {
    context.spawn(response_sink.send_all(request_stream))
}
```

Part V - Unary Implementation

Client Unary

```
let call = channel.create_call();
let (resp_future, tag) = CallTag::batch_pair();
// create a tag and let gRPC manages its lifetime
let tag_box = Box:new(tag);
let tag_ptr = Box::into_raw(tag_box) as _;
channel.start_batch(call, tag_ptr);
```

Unary Future

```
fn poll(&mut self) -> Poll<T, Error> {
    let guard = self.inner.lock();
    if let Some(res) = guard.result.take() {
        let r = try!(res);
        return Ok(Async::Ready(r));
    }
    // Has not been finished yet, Add notification hook
    if guard.task.is_none()
        || !guard.task.as_ref().unwrap().will_notify_current() {
        guard.task = task::current();
    }
    Ok(Async::NotReady)
}
```

Resolve Future

```
> Completion Queue:
let e = cq.next();
// Get the tag from gRPC again
let tag: Box<CallTag> = unsafe {
Box::from_raw(e.tag as _) } ;
tag.resolve(&cq, e.success != 0);
```

```
> Resolve:
let task = {
   let mut guard = self.inner.lock();
   guard.set_result(res)
   gurad.task.take()
};
task.map(|t| t.notify());
```

TiDB https://github.com/pingcap/tidb

TiKV https://github.com/tikv/tikv/

GRPC https://github.com/pingcap/grpc-rs

PingCAP

Thank You !