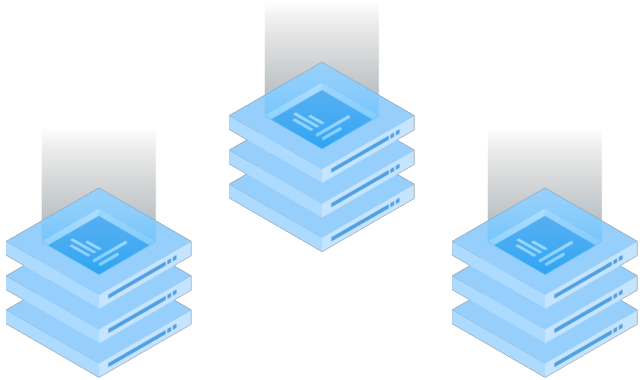
TiKV Raw API Execution Process

Presented by Zhen Liu

PingCAP TiDB



PingCAP

Part I – Call RPC Service

Raw API Overview

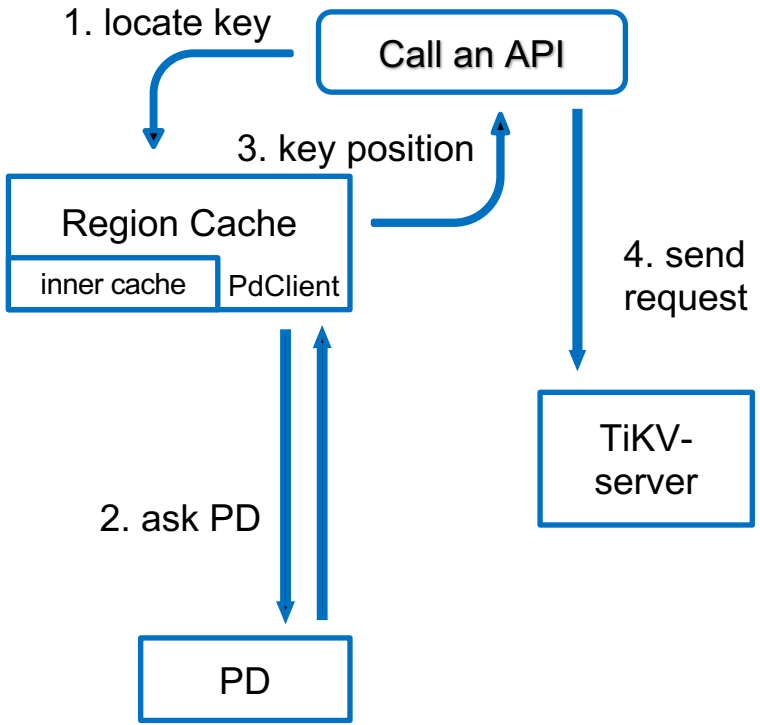
- `func (c *RawKVClient) ClusterID() uint64`
- `func (c *RawKVClient) Delete(key []byte) error`
- `func (c *RawKVClient) Get(key []byte) ([]byte, error)`
- `func (c *RawKVClient) Put(key, value []byte) error`
- `func (c *RawKVClient) Scan(startKey, endKey []byte, limit int) (keys [][]byte, values [][]byte, err error)`

Raw API Example

```
cli, err := tikv.NewRawKVClient
([]string{"192.168.199.113:2379"}, config.Security{})
defer cli.Close()
key := []byte("Company")
val := []byte("PingCAP")
// put key into tikv
err = cli.Put(key, val)
// get key from tikv
val, err = cli.Get(key)
// delete key from tikv
err = cli.Delete(key)
// get key again from tikv
val, err = cli.Get(key)
```

Ti-Client Flow Chart

- New a Client
- Construct a RPC request
- Send request to TiKV-server
    - Locate the key position(regionID,store_id ..)
        - 1. search cache
        - 2. update cache(missed)
    - RPC client send the request and get response
- Get response



New a RawkvClient

RawKVClient contains
1. clusterID to identify itself
2. regionCache to store and get region info
3. pdClient to connect with PD,rpcClient to connect with TiKV-server

```
func NewRawKVClient(pdAddrs []string, security config.Security) (*RawKVClient,
error) {
    pdCli, err := pd.NewClient(…)
    return &RawKVClient{
        clusterID: pdCli.GetClusterID(context.TODO()),
        regionCache: NewRegionCache(pdCli),
        pdClient: pdCli,
        rpcClient: newRPCClient(security),
    }, nil
…
}
```

PingCAP

Construct a RPC request

Get interface main process:

```
func (c *RawKVClient) Get(key []byte) ([]byte, error) {
    req := &tikvrpc.Request{
        Type: tikvrpc.CmdRawGet,
        RawGet: &kvrpcpb.RawGetRequest{
            Key: key,
        },
    }
    resp, _, err := c.sendReq(key, req)
}
```

Prepare to Send Request to TiKV-server

1. First to use regionCache to locate key position
2. Second to use RegionRequestSender to send request

```
func (c *RawKVClient) sendReq(key []byte, req *tikvrpc.Request)
(*tikvrpc.Response, *KeyLocation, error) {
…
    sender := NewRegionRequestSender(c.regionCache, c.rpcClient)
    for {
        loc, err := c.regionCache.LocateKey(bo, key)
        if err != nil {
            return nil, nil, errors.Trace(err)
        }
        resp, err := sender.SendReq(bo, req, loc.Region,
readTimeoutShort)
…
}
```

PingCAP

Region Cache

RegionCache maintain some datas:
1. regions : store region indexed by RegionVerID
2. sorted : store key range of regions sorted by star key
3. stores : record store information

```
type RegionCache struct {
    pdClient pd.Client
    mu struct {
        sync.RWMutex
        regions map[RegionVerID]*CachedRegion
        sorted *btree.BTree
    }
    storeMu struct {
        sync.RWMutex
        stores map[uint64]*Store
    }
}
```

PingCAP

Locate key

1. Region contains a sorted region tree sorted by start key
2. When cache is missed(no found or out of date), ask pd for new information and update regionCache.

```
func (c *RegionCache) LocateKey(bo *Backoffer, key []byte)
(*KeyLocation, error) {
     r := c.searchCachedRegion(key)
    r, err := c.loadRegion(bo, key)
    if err != nil {
        return nil, errors.Trace(err)
    }
    defer c.mu.Unlock()
    c.insertRegionToCache(r)
}
```

Send a RPC request

Send a RPC request according to the region information

```
func (s *RegionRequestSender) sendReqToRegion(ctx *RPCContext, req
*tikvrpc.Request…) (resp *tikvrpc.Response, retry bool, err error)
{
    if e := tikvrpc.SetContext(req, ctx.Meta, ctx.Peer); e != nil {
        return nil, false, errors.Trace(e)
    }
    resp, err = s.client.SendRequest(bo.ctx, ctx.Addr, req, timeout)
…
}
```

PingCAP

PingCAP

Part II - Raw Get Implementation

RPC Service

Call storage module to handle the request

```
fn raw_get(&self, ctx: RpcContext, mut req: RawGetRequest, sink:
UnarySink<RawGetResponse>) {
    let future = self
        .storage
        .async_raw_get(req.take_context(), req.take_cf(), req.take_key())
        .then(|v| {
            let mut resp = RawGetResponse::new();
      …
            sink.success(resp).map_err(Error::from)
        })

    ctx.spawn(future);
}
```

src/server/service/kv.rs

Storage module handler

1. Use read_pool to async execute function
2. Call fuction “async_snapshot” to get a snapshot and read value from it

```
pub fn async_raw_get(&self,ctx: Context,cf: String,key: Vec<u8>,
) -> impl Future<Item = Option<Vec<u8>>, Error = Error> {
let res = self.read_pool.future_execute(priority, move |ctxd| {
        Self::async_snapshot(engine, &ctx)
            .and_then(move |snapshot: E::Snap| {
snapshot.get_cf(cf, &Key::from_encoded(key))
                    // map storage::engine::Error -> storage::Error
                    .map(|r| {
                        if let Some(ref value) = r {
          ...
                        }

}
```

PingCAP

src/storage/mod.rs

Call Engine to get a snapshot

Use raftkv engine the get snapshot

```
fn async_snapshot(engine: E, ctx: &Context) -> impl Future<Item =
E::Snap, Error = Error> {
    let (callback, future) = util::future::paired_future_callback();
    let val = engine.async_snapshot(ctx, callback);
    future::result(val)
        .and_then(|_| future.map_err(|cancel|
EngineError::Other(box_err!(cancel))))
        .and_then(|(_ctx, result)| result)
        // map storage::engine::Error -> storage::txn::Error ->
storage::Error
        .map_err(txn::Error::from)
        .map_err(Error::from)
}
```

src/storage/mod.rs

RaftKV Engine Execute Read request

RaftKV engine represent the raftkv level

```
fn async_snapshot(&self, ctx: &Context, cb: Callback<Self::Snap>) ->
engine::Result<()> {
let mut req = Request::new();
    req.set_cmd_type(CmdType::Snap);
self.exec_read_requests(ctx, vec![req], box move |(cb_ctx, res)| match
res {
        Ok(CmdRes::Resp(r)) => cb((
            cb_ctx,
            Err(invalid_resp_type(CmdType::Snap,
r[0].get_cmd_type()).into()),
        )),
        Ok(CmdRes::Snap(s)) => {
         cb((cb_ctx, Ok(s)))
        }
}
```

PingCAP

src/storage/engine/raftkv.rs

Use RaftRouter Send Raft Command

RaftRouter realizes transport interface of raft group：send ..

```
fn exec_read_requests(&self, ctx: &Context,reqs: Vec<Request>,cb:
Callback<CmdRes>,) -> Result<()> {
let mut cmd = RaftCmdRequest::new();
    cmd.set_header(header);
    cmd.set_requests(RepeatedField::from_vec(reqs));
    self.router.send_command(
            cmd,
            StoreCallback::Read(box move |resp| {
                let (cb_ctx, res) = on_read_result(resp, len);
                cb((cb_ctx, res.map_err(Error::into)));
            }),
        )
        .map_err(From::from)
}
```

src/storage/engine/raftkv.rs

Worker will Check Task

RafterRouter push the worker to do actually task, and the worker will check task type and do different work.

```
fn send_command(&self, req: RaftCmdRequest, cb: Callback) ->
RaftStoreResult<()> {
    self.try_send(StoreMsg::new_raft_cmd(req, cb))
}

fn try_send(&self, msg: StoreMsg) -> RaftStoreResult<()> {
    if ReadTask::acceptable(&msg) {
        self.local_reader_ch
            .schedule(ReadTask::read(msg))
            .map_err(|e| box_err!(e))
    } else {
        self.ch.try_send(msg).map_err(RaftStoreError::Transport)
    }
}
```

PingCAP

src/server/transport.rs

Worker will Check Task

```
Task accepts `Mag`s that contain Get/Snap requests.
Returns `true`, it can be saftly sent to localreader,
Returns `false`, it must not be sent to localreader.

pub fn acceptable(msg: &StoreMsg) -> bool {
    match *msg {
        StoreMsg::RaftCmd { ref request, .. } => {
            if request.has_admin_request()
                 ||request.has_status_request() {
                false
            } else {
                for r in request.get_requests() {
                    match r.get_cmd_type() {
                        CmdType::Get | CmdType::Snap => (),
         }
                true
    }
```


src/raftstore/store/worker/read.rs

Read Locally or Send to Peers(1)

Peers need to handle read request

```
fn handle_read(&mut self, req: RaftCmdRequest, check_epoch: bool) ->
ReadResponse {
        let mut resp = ReadExecutor::new(
            self.engines.kv.clone(),
            check_epoch,
            false, /* we don't need snapshot time */
        ).execute(&req, self.region());

        cmd_resp::bind_term(&mut resp.response, self.term());
        resp
}
```

src/raftstore/store/peer.rs

Read Locally or Send to Peers(2)

Leader can read from local data

```
 pub fn execute(&mut self, msg: &RaftCmdRequest, region: &metapb::Region) ->
ReadResponse {
…
for req in requests {
        let cmd_type = req.get_cmd_type();
        let mut resp = match cmd_type {
            CmdType::Get => match self.do_get(req, region) {
…
}
 fn do_get(&self, req: &Request, region: &metapb::Region) ->
Result<Response> {
let mut resp = Response::new();
…
    let snapshot = self.snapshot.as_ref().unwrap();
…}
```

PingCAP

src/raftstore/store/peer.rs

PingCAP

Part III – Raw Put Implementation

Raw Put Implementation

```
Similar to raw get process
pub fn async_raw_put(&self,ctx: Context,cf: String,key: Vec<u8>,value:
Vec<u8>,callback: Callback<()>,
) -> Result<()> {
self.engine.async_write(
        &ctx,
        vec![Modify::Put(
            Self::rawkv_cf(&cf)?,
            Key::from_encoded(key),
            value,
        )],
        box |(_, res): (_, engine::Result<_>)|
callback(res.map_err(Error::from)),
    )?;
Ok(())
}
```

PingCAP

src/storage/mod.rs

Call RaftKV Engine to Write

```
Convert Vec<Modify> to Request,call other function
fn async_write(&self,ctx: &Context,modifies: Vec<Modify>,
cb: Callback<()>,) -> engine::Result<()> {
for m in modifies {
        let mut req = Request::new();
        match m {
           Modify::Put(cf, k, v) => {
                let mut put = PutRequest::new();
…
                req.set_put(put);
            }
}
        reqs.push(req);
    }
self.exec_write_requests(ctx, reqs, box move |(cb_ctx, res)| match
res}
```

PingCAP

src/storage/engine/raftkv.rs

Alse Use RaftRouter to send command

```
fn exec_write_requests(&self,ctx: &Context,reqs: Vec<Request>,cb:
Callback<CmdRes>,) -> Result<()> {
let header = self.new_request_header(ctx);
…
    let mut cmd = RaftCmdRequest::new();
    cmd.set_header(header);
    cmd.set_requests(RepeatedField::from_vec(reqs));
    self.router
        .send_command(cmd,
            StoreCallback::Write(box move |resp| {
                let (cb_ctx, res) = on_write_result(resp, len);
                cb((cb_ctx, res.map_err(Error::into)));
            }),
        )…
}
```

PingCAP

src/storage/engine/raftkv.rs

Alse Use RaftRouter to send command

```rust
pub ch: SendCh<StoreMsg>,
pub type SendCh<T> = RetryableSendCh<T, mio::Sender<T>>;
```

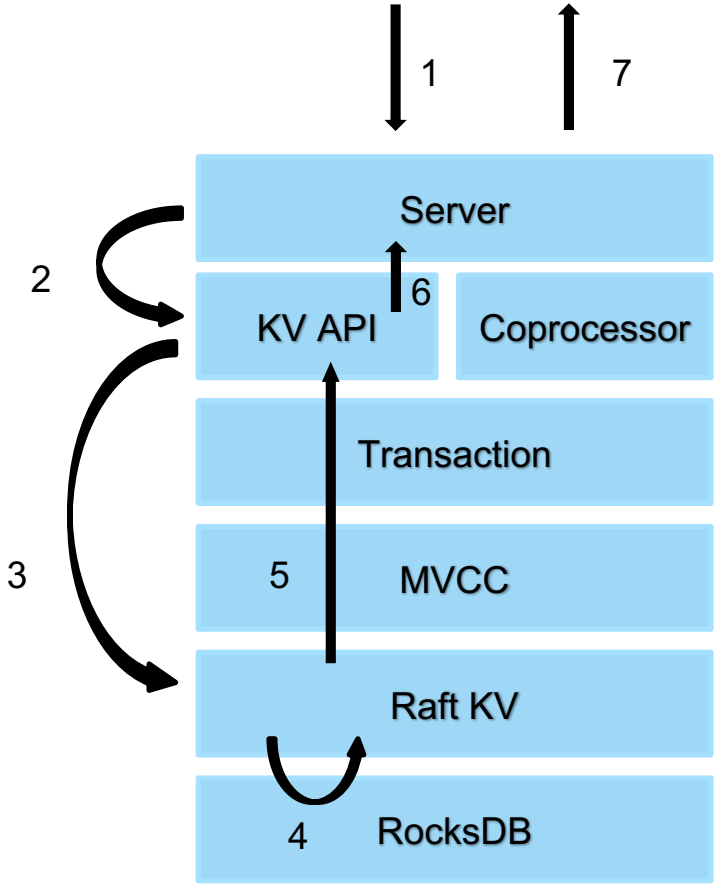
src/util/transport.rs

```rust
pub ch: SendCh<StoreMsg>,
fn try_send(&self, msg: StoreMsg) -> RaftStoreResult<()> {
    if ReadTask::acceptable(&msg) {
        self.local_reader_ch
            .schedule(ReadTask::read(msg))
            .map_err(|e| box_err!(e))
    } else {
        self.ch.try_send(msg).map_err(RaftStoreError::Transport)
    }
}
```


src/server/transport.rs

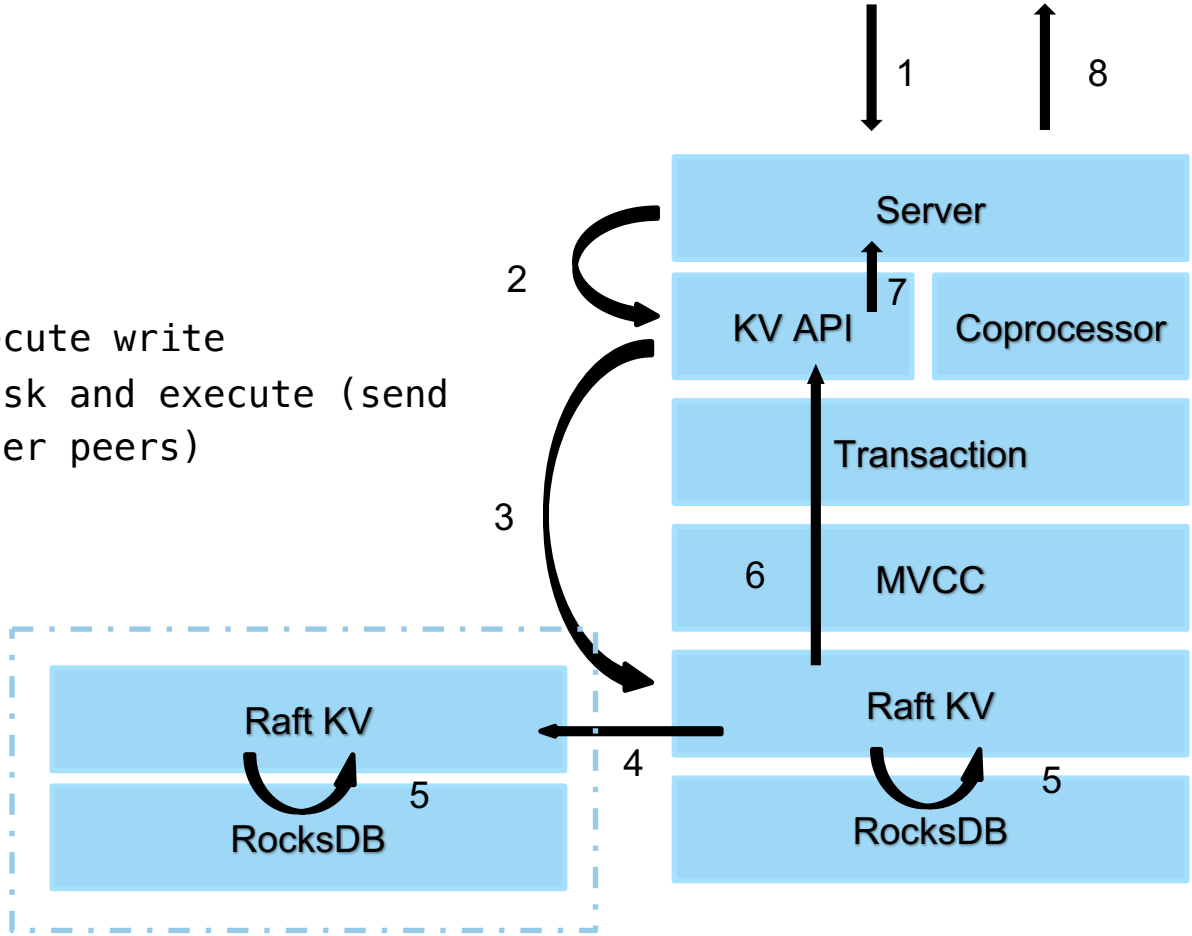
Raw Get Flow

1. Recv read request
2. Dispatch to KV API
3. Put into read pool and get raftkv snapshot
4. Raft worker check task and execute (read locally or send to peers)
5. Return snapshot
6. Use snapshot read and return result
7. Finished read



Raw Put

1. Recv put request
2. Dispatch to KV API
3. Call raft engine execute write
4. Raft worker check task and execute (send write command to other peers)
5. Write to RocksDB
6. Return result
8. Finished put

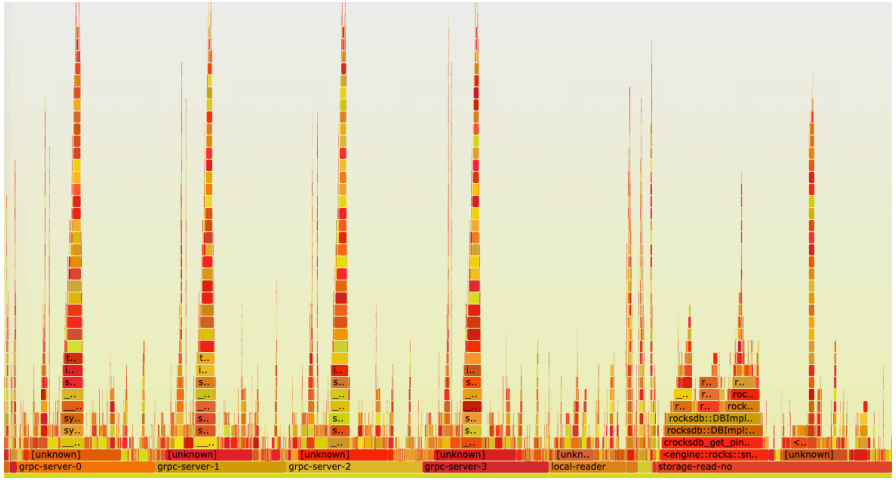


PingCAP

Part IV – Advices

Advices

- Problem：RPC consumes most of the computer resources

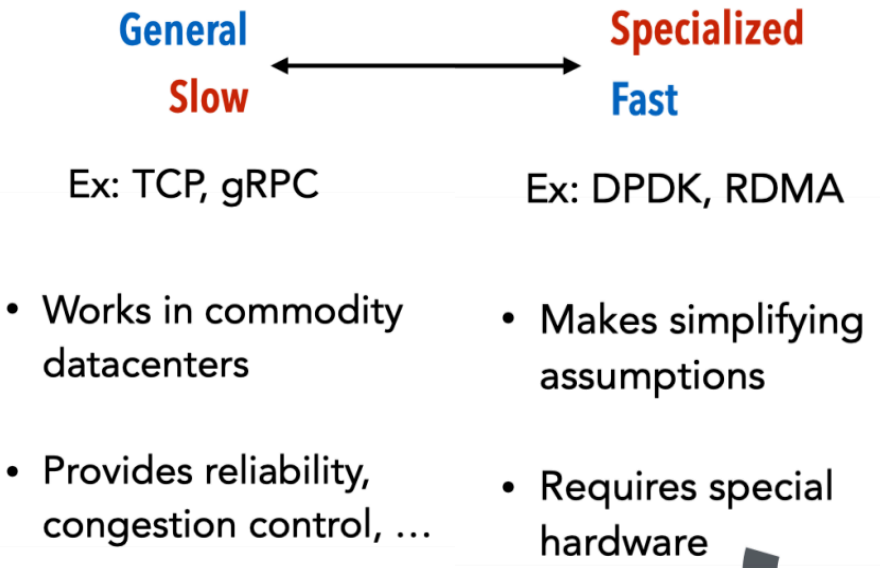


PingCAP

Advices

More efficient RPC：eRPC

NSDI‘19 Datacenter RPCs can be General and Fast



PingCAP

Thank You !