database/sql

There and back and again

Intro

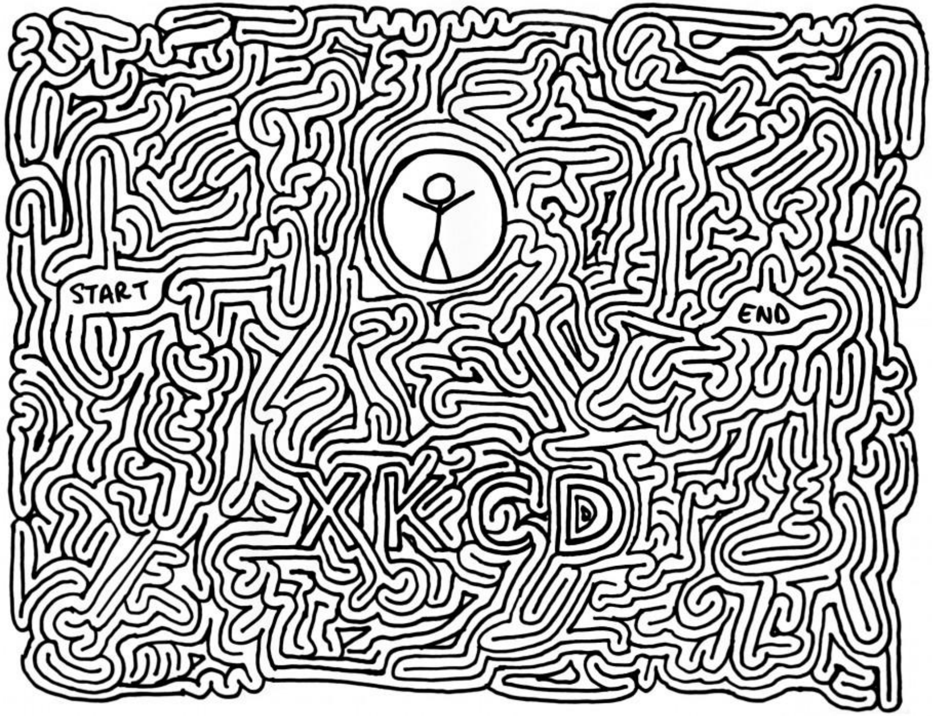
- Huu Khiem (Mark), currently at Viki (viki.com)

- Motivation for talk: need to understand how Go + SQL works better, to:
  - Optimize code (medium/large services PostgreSQL, 8M+ users, lots of touch point with DB)
  - Capacity planning (understand behavior at average & peak load)
  - Failure handling (know when and how database code fails)
  - Write less (lots of boilerplate code to delete)

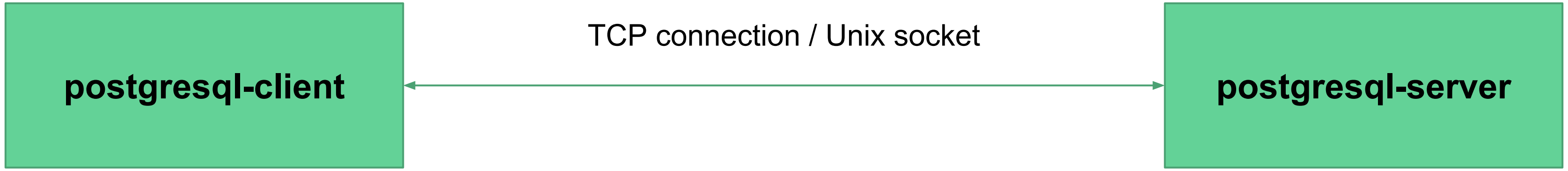
- Talk based on Go 1.7. Lots of change in Go 1.8!

It’s harder than it looks

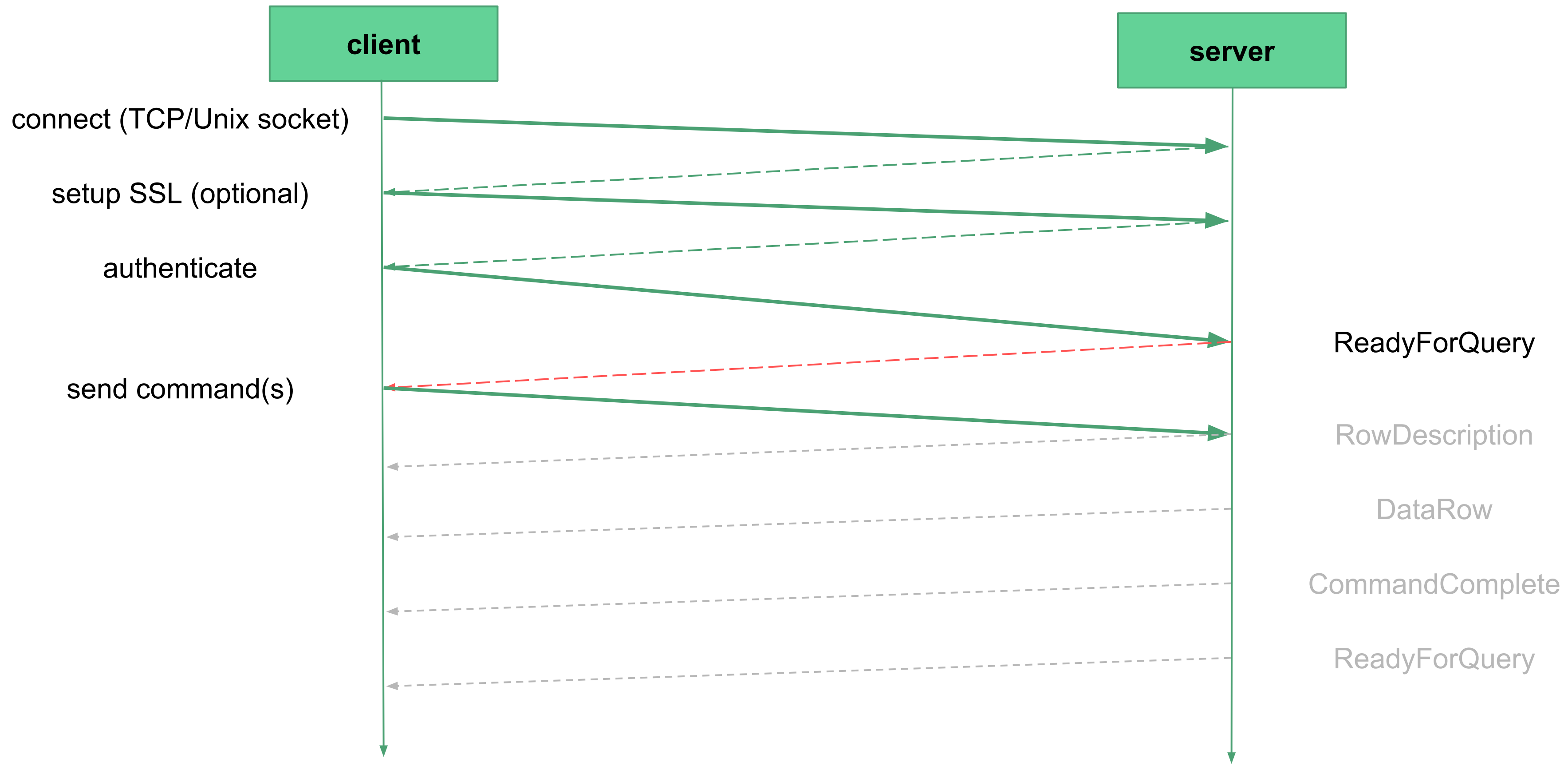
```
db, err := sql.Open(“postgres”, “…”)
if err != nil {
     // handle error
}

// ...
rows, err := db.Query(query, args…)
if err != nil {
     // handle error
}
defer row.Close()
for rows.Next() {
     // scan and parse
}
```



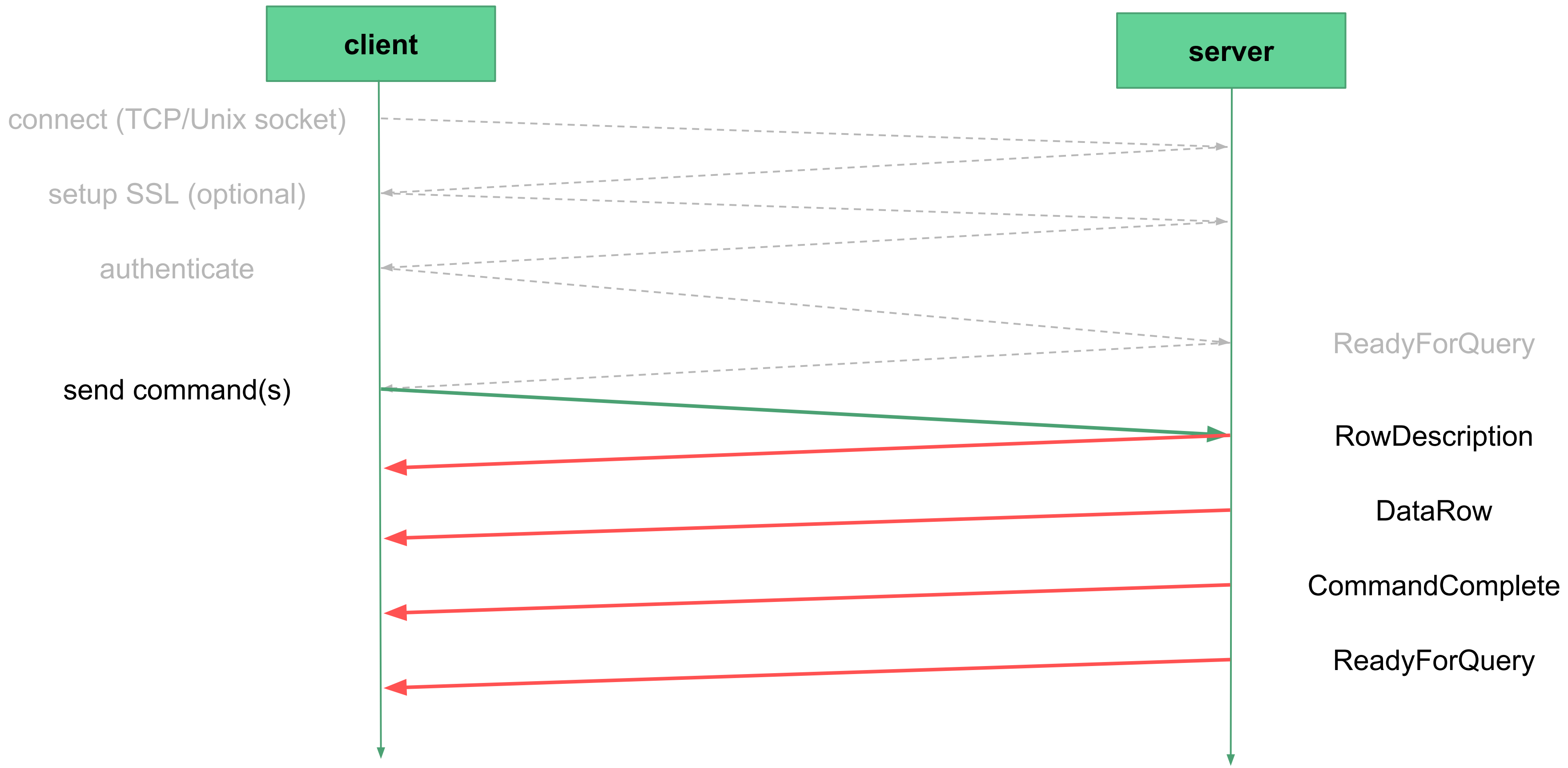
Talking to PostgreSQL, high level



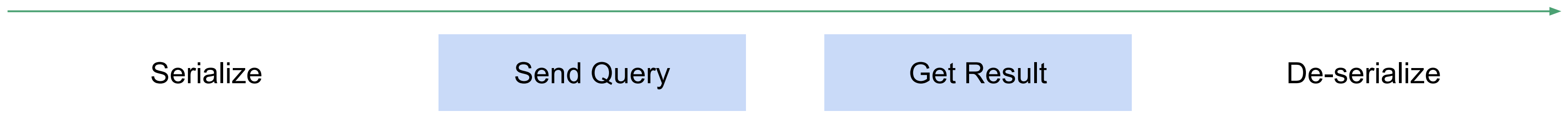
Establish connection



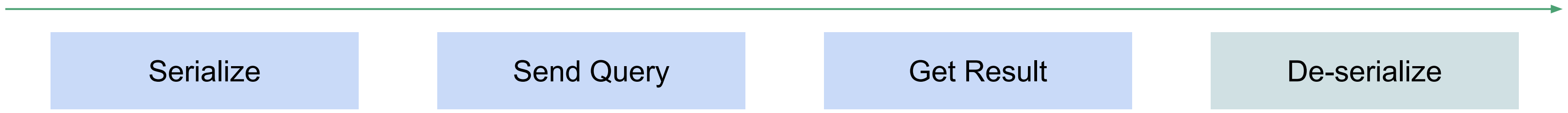
Establish connection (cont.)



Tracing the calls: serialize



Tracing the calls: deserialize



So far...

Saw how data flows within **one** connection

How about **concurrent** connections?

How is a connection created?

```go
type DB struct { // link
    // ...
    openerCh chan struct{}
    // ...
}

func (db *DB) ConnectionOpener() { // link
    for range db.openerCh {
        db.openNewConnection()
    }
}
```

How is a connection created? (cont.)

```
func (db *DB) maybeOpenNewConnections() { // link
    // …
    for numRequests > 0 {
        db.numOpen++ // optimistically
        numRequests--
        if db.closed {
            return
        }
        db.openerCh <- struct{}{}
    }
}
```

How is a connection created? (visualize)

- Modify `database/sql` to write logs
- Aggregate events and get counts

- *NOTE: this experiment is not a proper benchmark*

So far...

- database/sql provide app-level connection pool
- pool size is dynamic, but can be controlled by:
  - SetMaxOpenConns()
  - SetMaxIdleConns()
- Usually: set maximum open conn
  - less chance for runaway connections
  - helps with capacity planning
  - Though it cause other effects (might block requests => increased latency)
- Collect data from db.Stats() / source code to plan for capacity

Cleaner code with database/sql/driver

```
func (c CountryType) Value() (driver.Value, error) {
    return c.String(), nil
}

func (c *CountryType) Scan(v interface{}) error {
    val, ok := v.([]byte)
    if !ok {
        return fmt.Errorf("invalid data for CountryType: %#v", v)
    }

    *c = pgCountryTypeToEnum[string(val)]
    return nil
}
```

ORM or not ORM?

Basic problems:

- Marshaling data (solved)
- Generating queries (same same, but different)

- Generating queries == generating SQL
- Hard (SQL is expressi) & limit the patterns you can use

What we have not covered

- Performance: prepared query vs one-time
- In-depth connection pooling
- Failure handling

More

- https://github.com/bradfitz/go-sql-test
- https://github.com/DATA-DOG/go-sqlmock
- https://docs.google.com/document/d/1F778e7ZSNiSmbju3jsEWzShcb8lIO4kDyfKDNm4PNd8/edit#
- https://www.postgresql.org/docs/9.6/static/protocol.html
- <rtfsc> :D