

Practical TLA⁺ for Concurrent and Distributed Systems

Calvin Loncaric

Oracle, Inc.

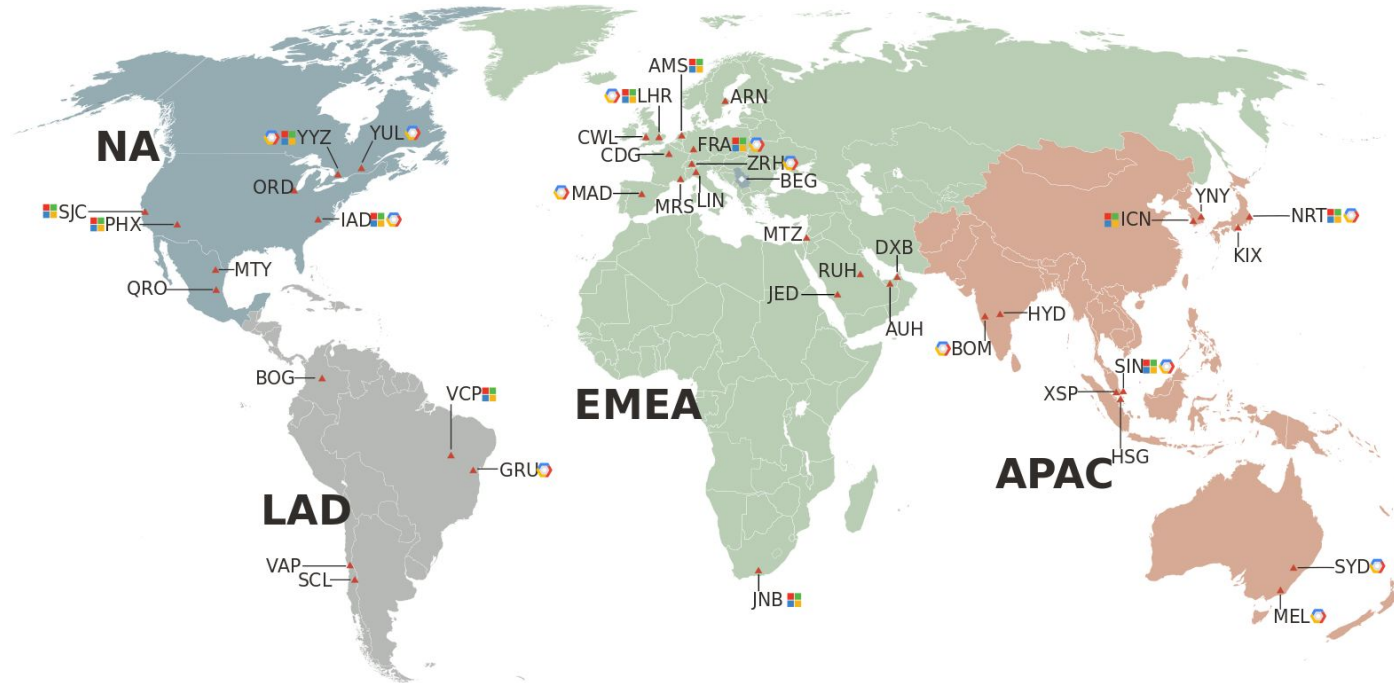
Who am I?

- 2013–2018: PHD at UW
- 2018: Started work at Oracle, doing distributed systems verification
 - Very unique position!
 - Real systems, real customers
 - Coding? Rarely!
 - Publishing? Rarely!
 - Formal methods? Always!
- 2022: Became a maintainer of the open-source TLA⁺ tools
- Today: Championing formal methods at Oracle, working (slowly) to broaden adoption

Formal Methods at Oracle Cloud Infrastructure

- 2011: Amazon starts using TLA⁺
 - Somewhat secretive!
- 2014: Chris Newcombe et. al. publish “Why Amazon Chose TLA⁺”
- 2014: Oracle hires Chris to help architect their new cloud
- 2017: Chris establishes a “Verification Team” to do formal methods full-time
- 2018: They hire me!

Oracle Cloud



<https://docs.oracle.com/en-us/iaas/Content/GSG/Concepts/concepts-physical.htm>

Why does a cloud provider want formal methods?

- Downtime is costly
- Data loss can be *irreversible*

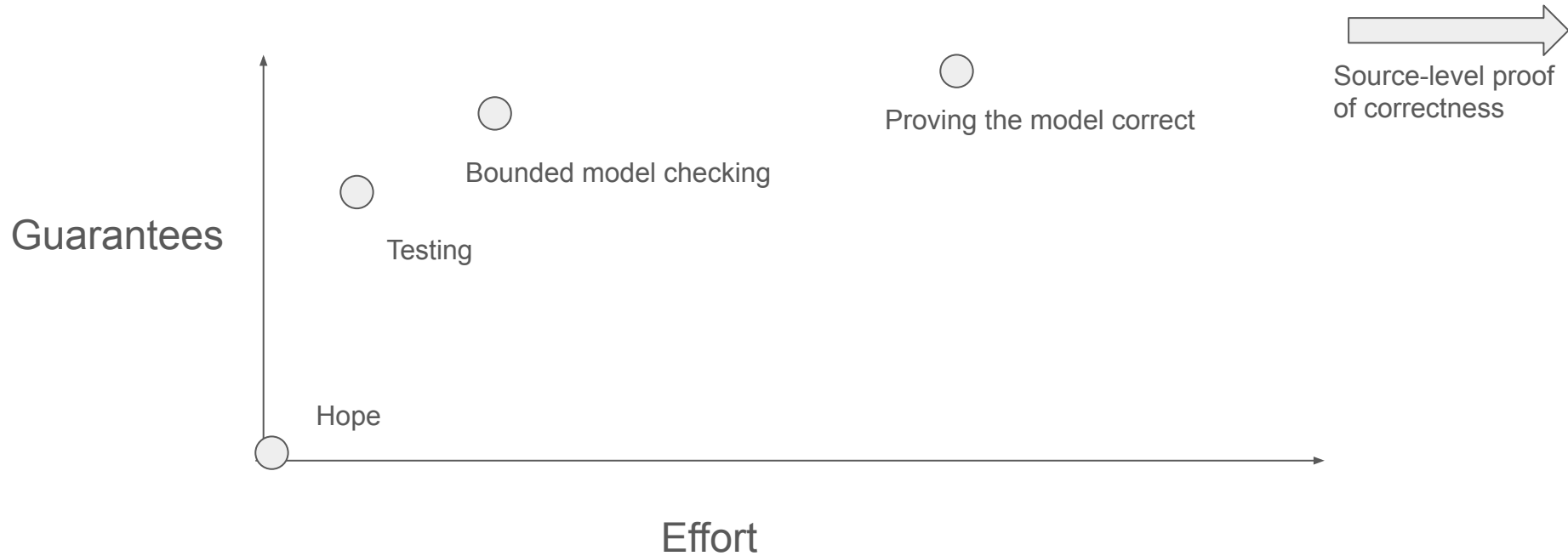
- Testing is great! ...but there are gaps:
 - Network anomalies
 - Power outages
 - Underspecified behaviors
 - Human intervention

The cost of failure

<real incidents>

Use of TLA⁺ in Cloud Computing

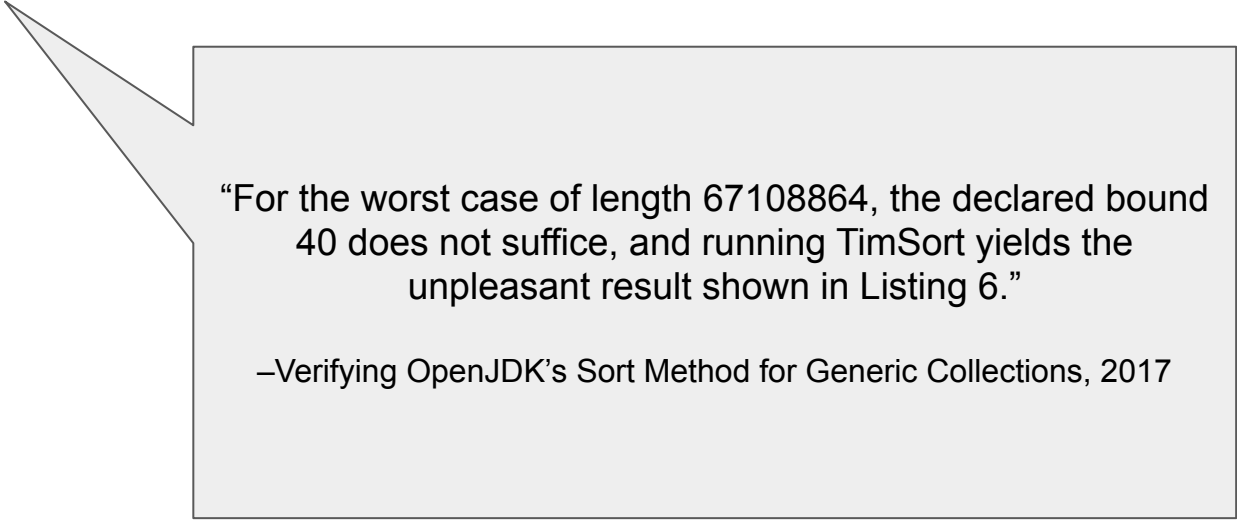
Choices for software verification



Model checking: unreasonably effective

“Small scope hypothesis” or “small model hypothesis”:

- Almost all bugs can be triggered by small inputs / exhibited by short traces



“For the worst case of length 67108864, the declared bound 40 does not suffice, and running TimSort yields the unpleasant result shown in Listing 6.”

–Verifying OpenJDK’s Sort Method for Generic Collections, 2017

From Testing to Model Checking

What does model checking add?

- Much easier to describe environment actions (power outages, process crashes, human operators, etc.)
- Lack of bias: cover *every* interleaving, not just those you can induce from the Linux scheduler

From Model Checking to Proof

What does a proof add?

- True exhaustiveness: not just $\forall x \in \text{BoundedSet}$, but actually $\forall x$
- Slightly higher confidence: it's never clear how big your model needs to be!
 - Is 3 actors enough? 5? 100?

From Proof_{model} to Proof_{code}

What does a proof on the source code add?

- A lot!
- Direct connection from spec to implementation
- ...but prohibitively difficult for all but the most critical lines of code

So what exactly does Oracle do with TLA⁺?

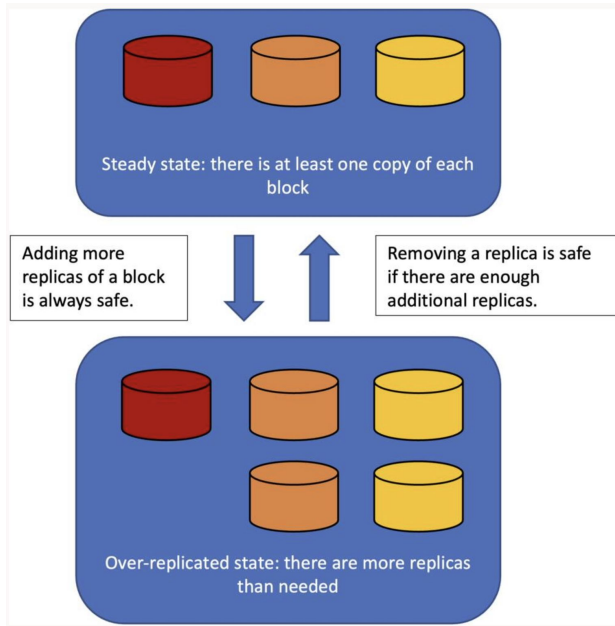
[details redacted]

Broadly:

- *Specification*: hammering out exactly what behaviors customers can expect
- *Formal modeling*: constructing simplified models of systems and protocols
- *Model checking*: exhaustively testing the models to find bugs
- *Machine-checked proof* (rarely!): when model checking isn't enough (and importance is high enough)

Example 1/2: drive management

<https://blogs.oracle.com/cloud-infrastructure/post/sleeping-soundly-with-the-help-of-tla>



Example 1/2: drive management

<https://blogs.oracle.com/cloud-infrastructure/post/sleeping-soundly-with-the-help-of-tla>

“A removal request can be issued whenever there are enough additional copies of the data on other drives.”

Dangerous!

=> Allows inadvertent removal of all data using multiple concurrent requests

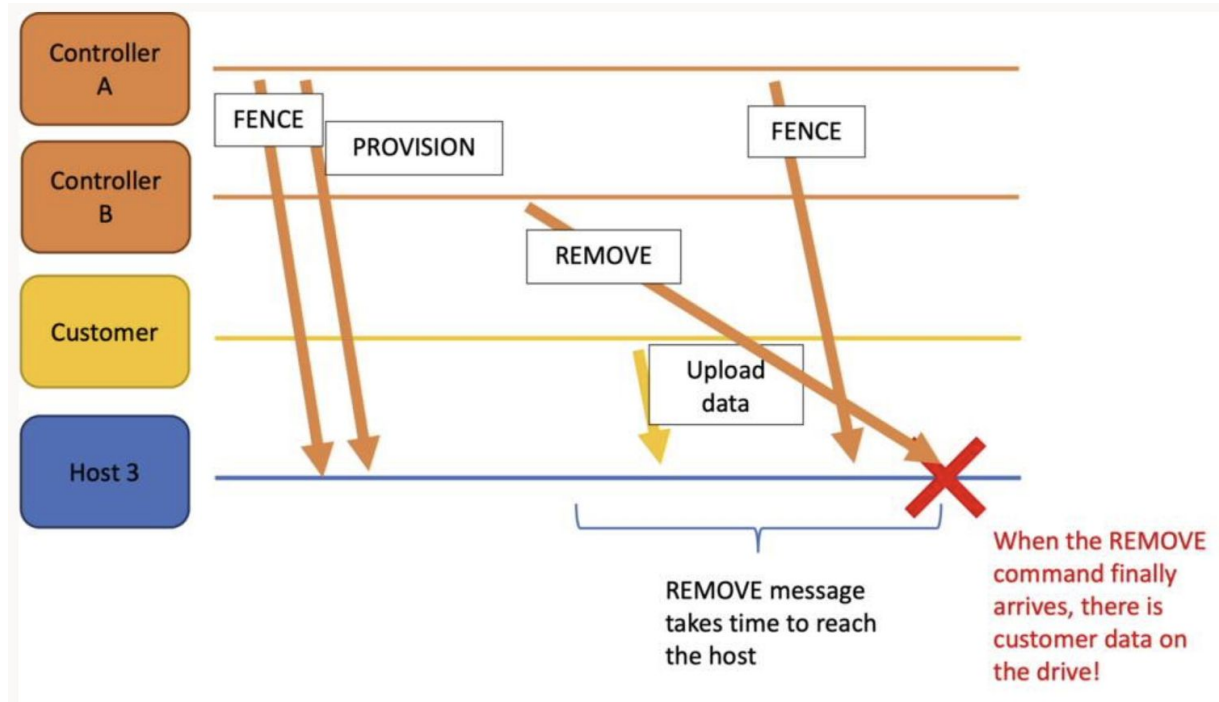
“Also, there are no other in-progress removals.”

Still not enough!

=> Impossible to enforce
=> ...and not enough, even if you could!

Example 1/2: drive management

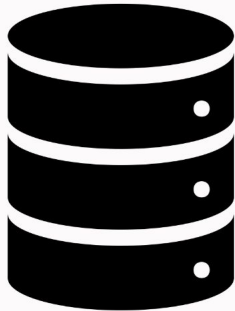
<https://blogs.oracle.com/cloud-infrastructure/post/sleeping-soundly-with-the-help-of-tla>



Example 2/2: Automatic Password Rotation

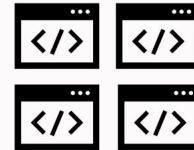
<https://conf.tlapl.us/2024/>

Ultra-High-Level Intuition



Account A

Clients connect using Account A



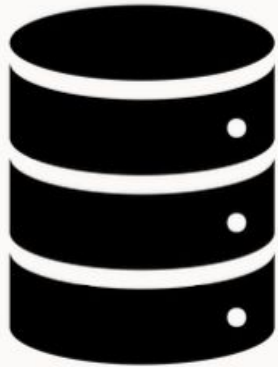
Account B

It is safe to change Account B's password without disrupting new connections

Example 2/2: Automatic Password Rotation

<https://conf.tlapl.us/2024/>

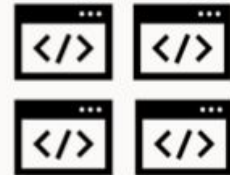
Ultra-High-Level Intuition



It is safe to change Account A's password without disrupting new connections

Account A

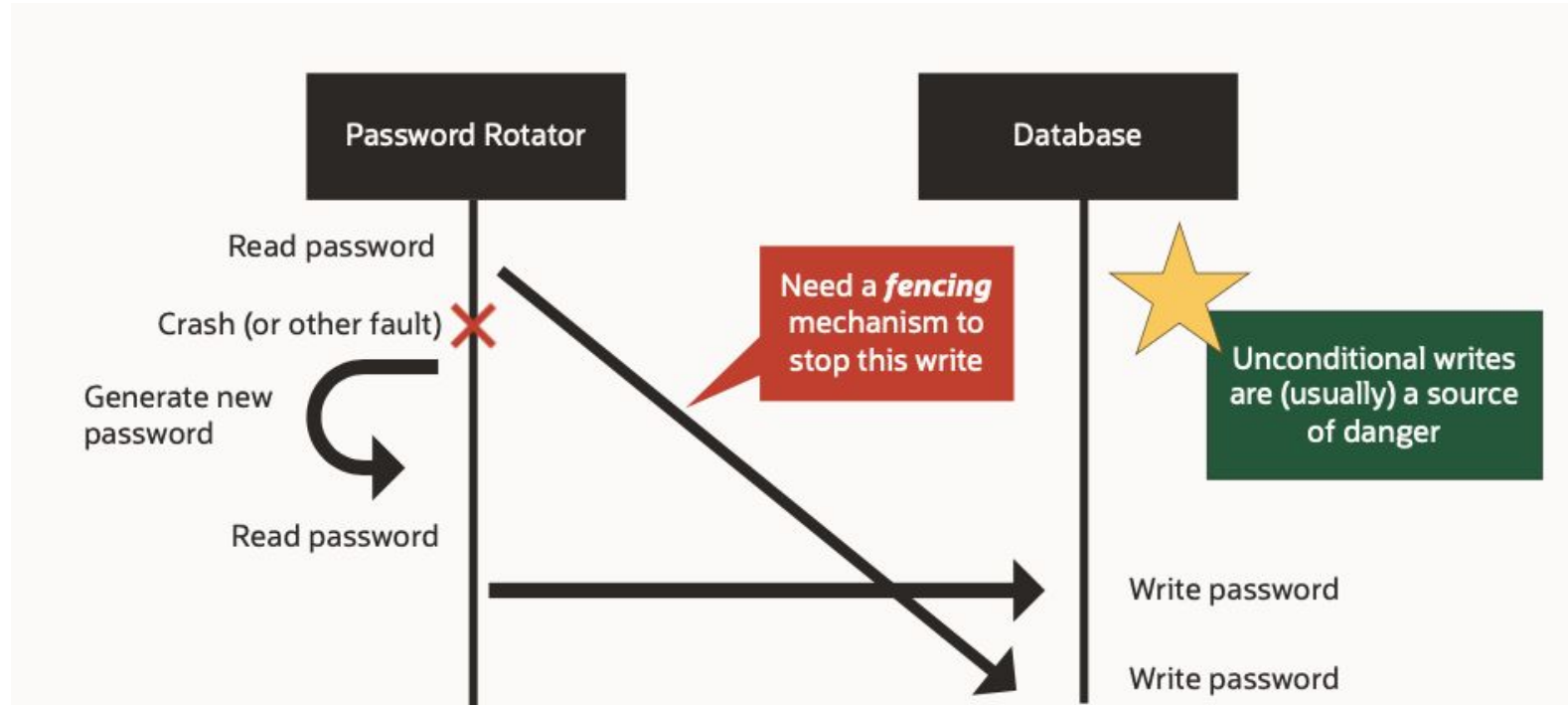
Account B



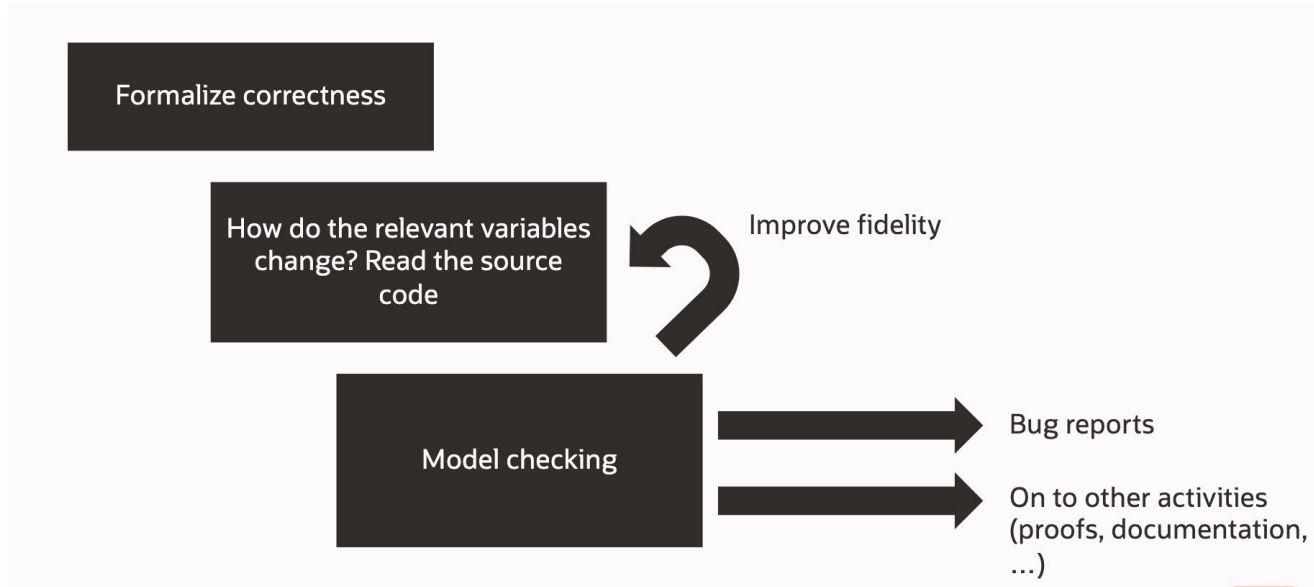
Clients connect using Account B

Example 2/2: Automatic Password Rotation

<https://conf.tlapl.us/2024/>



In general: the consultancy model



Segue: storage systems!

- Deeply interesting
- Critically important
- You use them every day

“I’m never going to write a storage system. Is this relevant to me?”

YES!

Even the *purest* programs use them:

- Compilers write output to disk
- Browsers have on-disk caches
- etc.

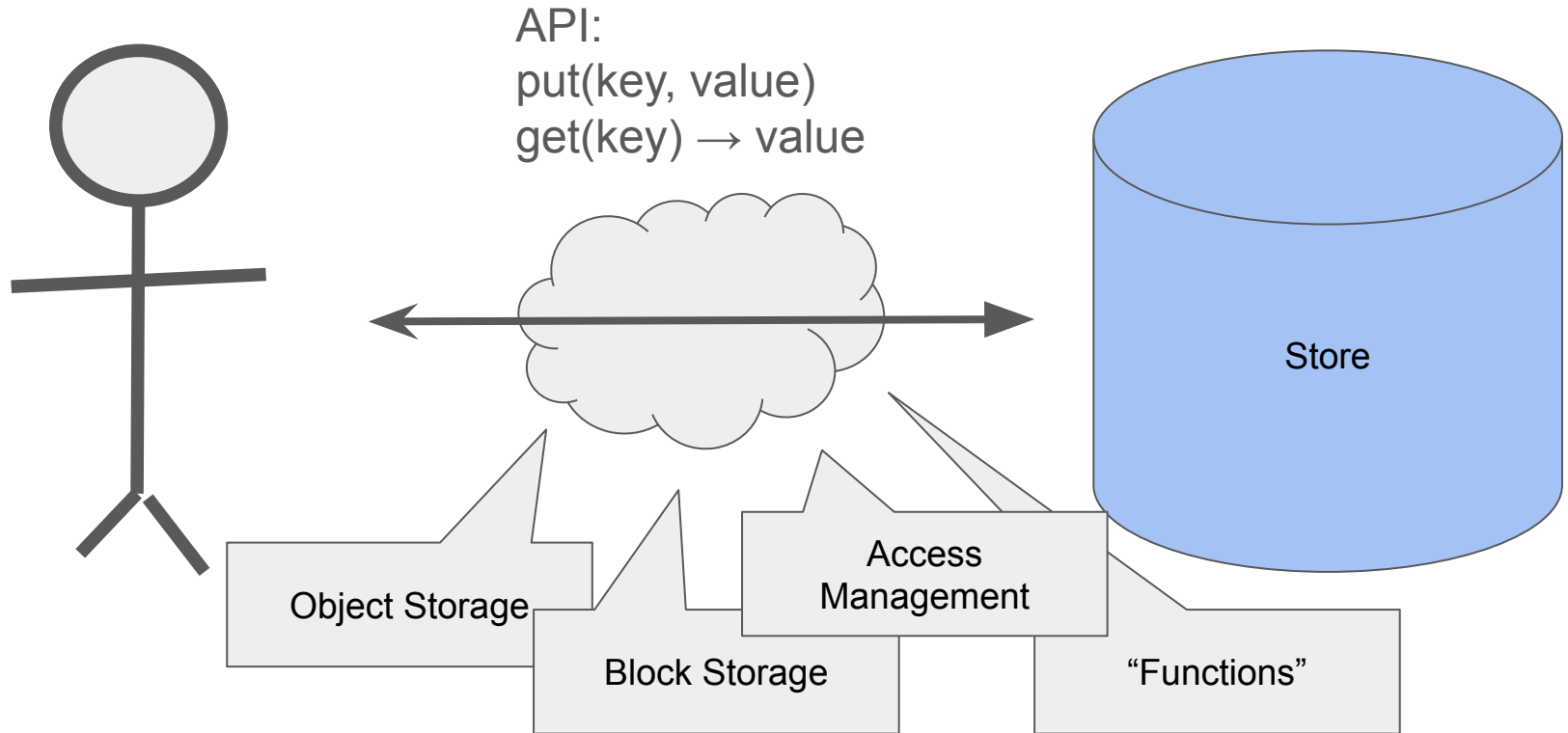
What to expect

- Focus on stateful systems (networks, disks, ...)
 - Not very cloud-specific—but biased toward things I have personally been exposed to in my work
- Formalization in TLA⁺
- Big exercises writing TLA⁺
- Emphasis on model checking instead of proof
 - See small model hypothesis slide from earlier: proof is *obviously a stronger result*, but model checking is *much quicker* and still provides great benefit

Today

- Strong consistency/linearizability
- Exercise!
- State machine refinement
- Exercise!
- Exercise!

Motivating Example: a networked key-value store



Motivating Example: a networked ~~key-value store~~ single register

Good advice I once received from Zach Tatlock (paraphrased):

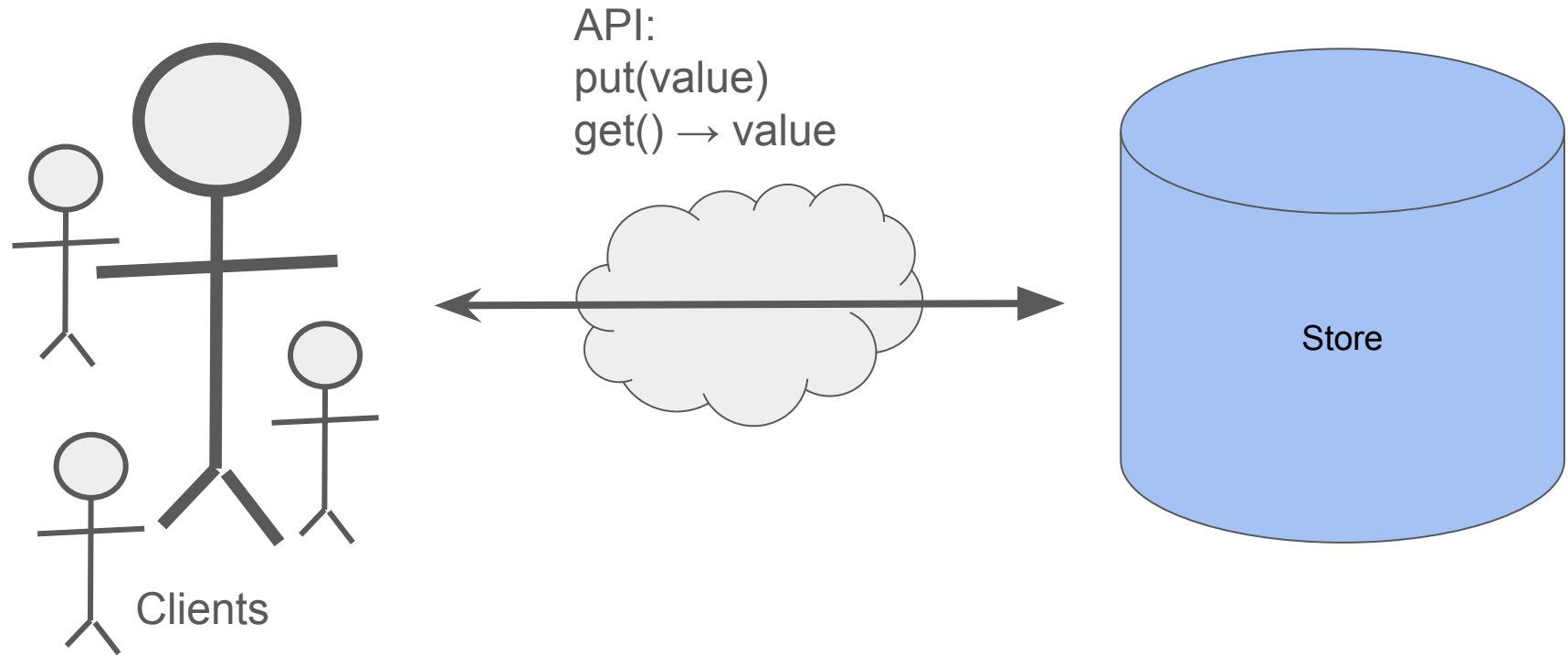
“Start with the simplest thing you can possibly imagine.”

Motivating Example: a networked ~~key-value store~~ single register

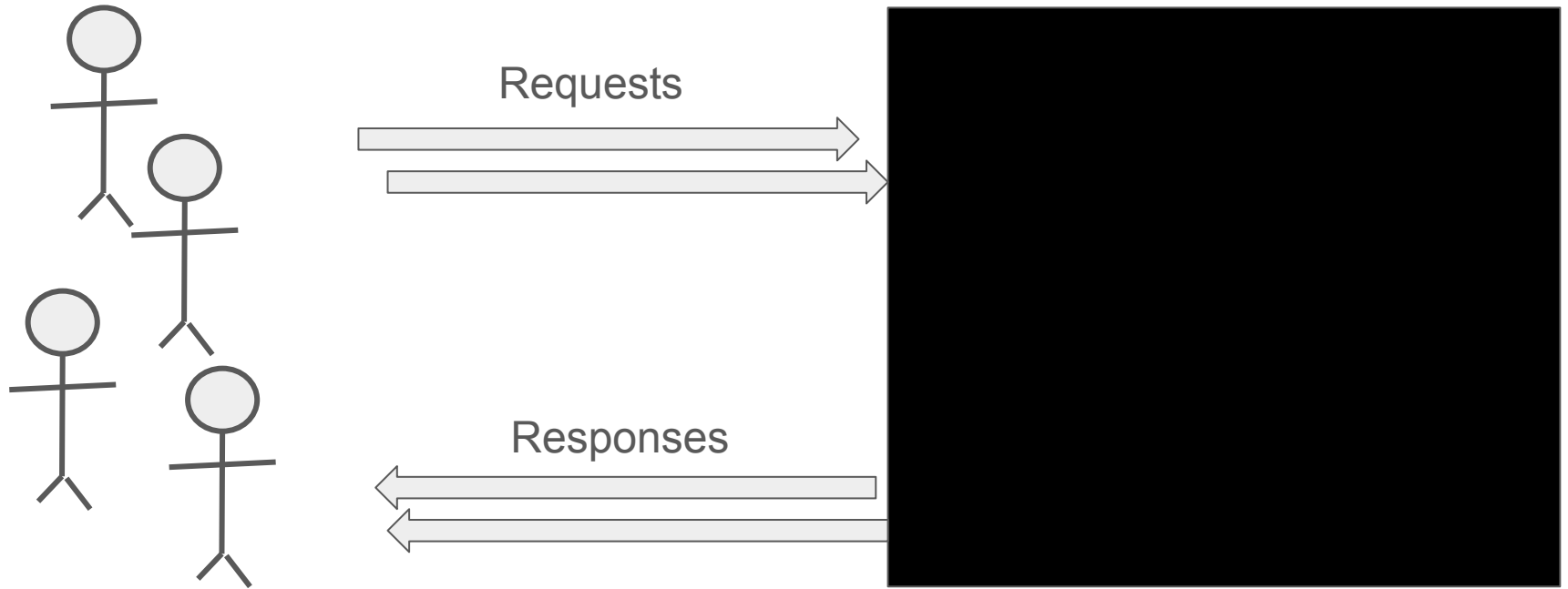
Plenty of interesting stuff to explore here

- Today:
 - Specifying a contract
 - Writing a model implementation
 - Bounded proof of correctness :)
- Friday:
 - Durable disk storage
 - Crash safety/liveness
 - Use in a larger system
- Even more, not covered this week:
 - secondary backups, 2-phase commit, Paxos, ...

Motivating Example: a networked register



Picture of the world



To specify a contract

Hoare-style pre- and post-conditions:

{TRUE}

put(value)

{get() = value}

A crack in the wall

Concurrent actor invalidates postcondition!

{TRUE}

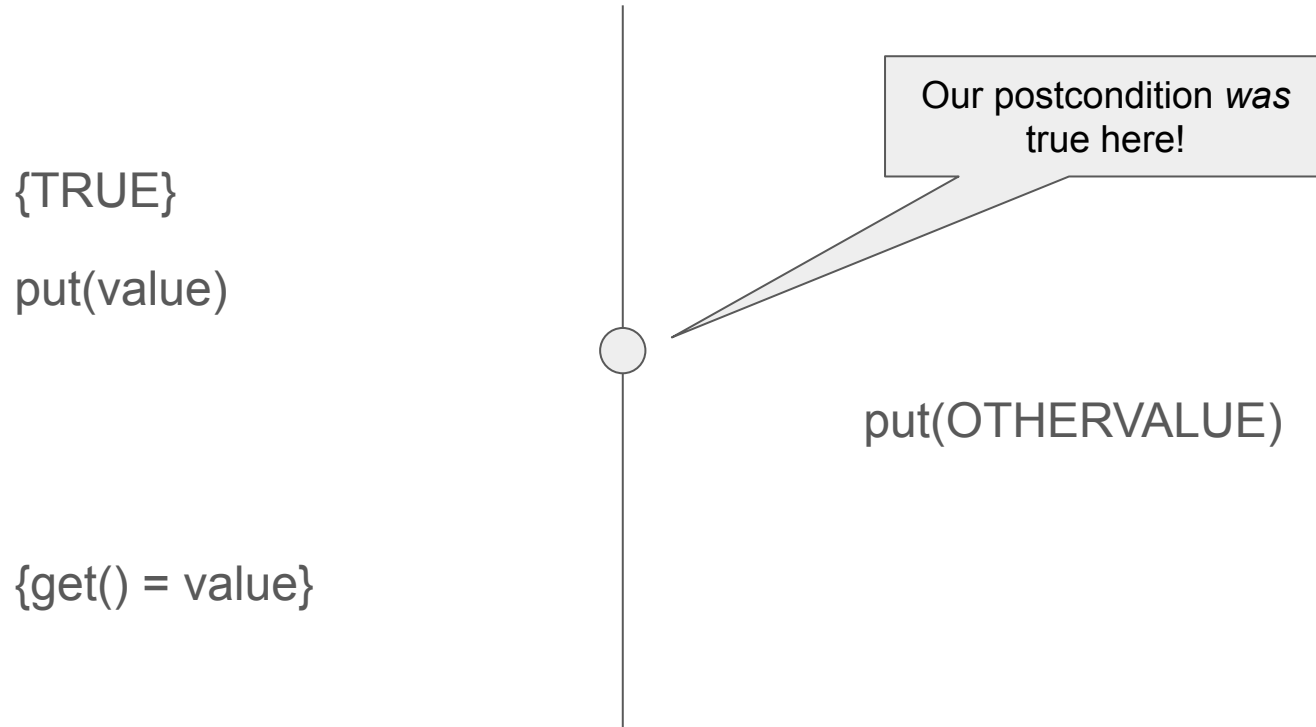
put(value)

← put(OTHERVALUE)

{get() = value}

Not true! Now it is
OTHERVALUE!

Observation: the postcondition *was* true at some point!



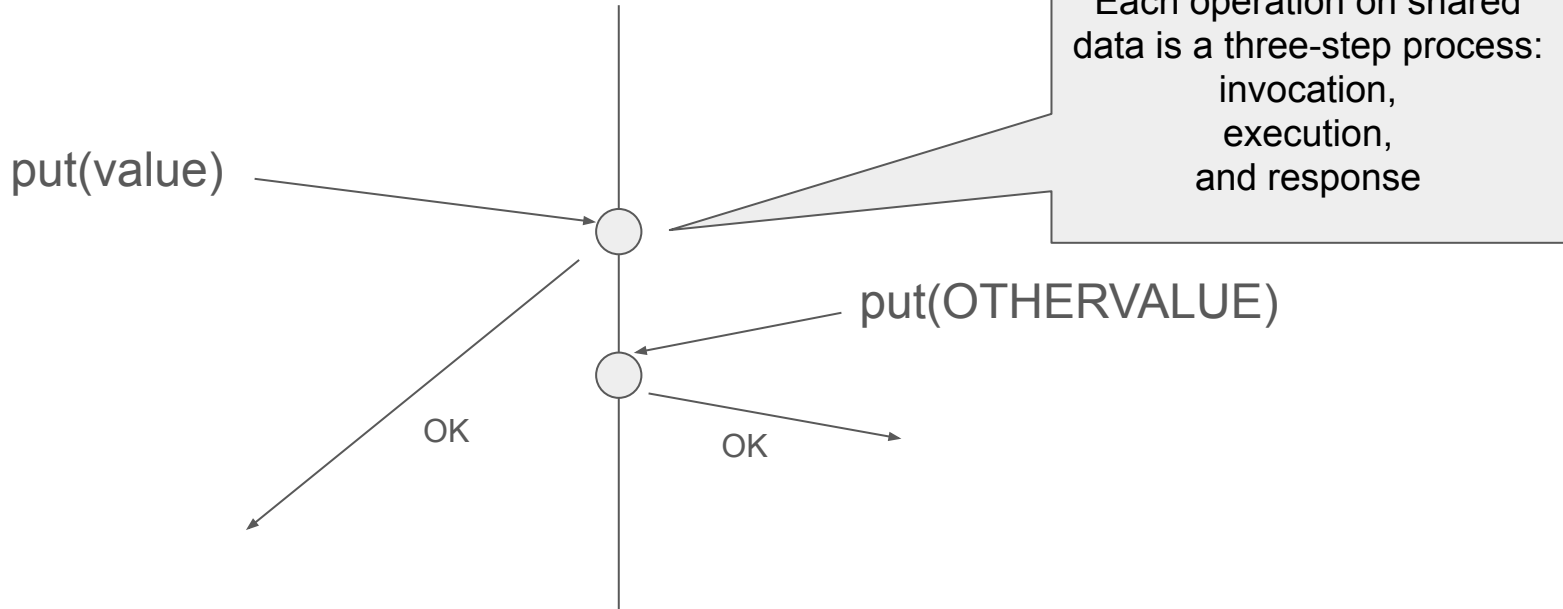
Linearizability (“Strong Consistency”)

“each operation applied by concurrent processes takes effect instantaneously at some point between its invocation and its response”

– Herlihy and Wing ‘90

Linearizability (“Strong Consistency”)

“each operation applied by concurrent processes takes effect instantaneously at some point between its invocation and its response”



Linearizability for our networked register, take 1

```
VARIABLE register_value  
  
Put(new_value) ==  
    register_value' = new_value  
  
Next == \E x \in Values: Put(x)
```

Problem: what about reads?

Problem: *who drives it?* The system cannot be allowed to change the value on its own; only in response to client requests

Solution: *model the client*

Linearizability for our networked register, take 2

Adding a simple description of the client

- Program counter (PC)
- Communication channel

VARIABLES

`client_pc,`
`requests,`
`responses`

Three actions

- Invocation
- Execution
- Response

“each operation applied by concurrent processes takes effect instantaneously at some point between its invocation and its response”

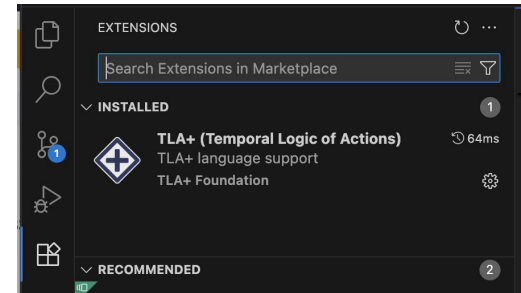
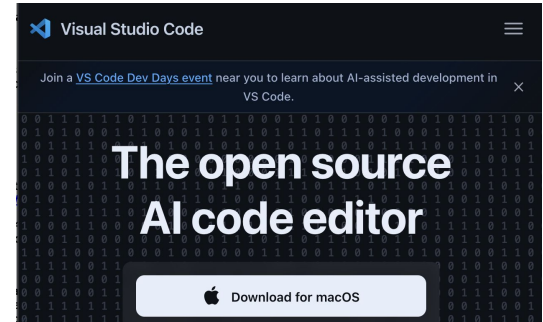
Exercise 1: specifying a networked register

1. Install VSCode
 - a. <https://code.visualstudio.com/>
2. Install extension “TLA+” from “TLA+ Foundation”
3. Download:
 - a. <https://calvin.loncaric.us/resources/vtsa2025.tar.gz>
4. Extract
5. Open folder with VSCode

Boilerplate has been provided! TODO:

- Fill in actions
- Visualize the state space with 1 client (to confirm your intuition)

<https://lampert.azurewebsites.net/tla/summary-standalone.pdf>



So we specified it. So what?

Reminder of where we're going: a verified implementation!

You can't verify an implementation without a contract!

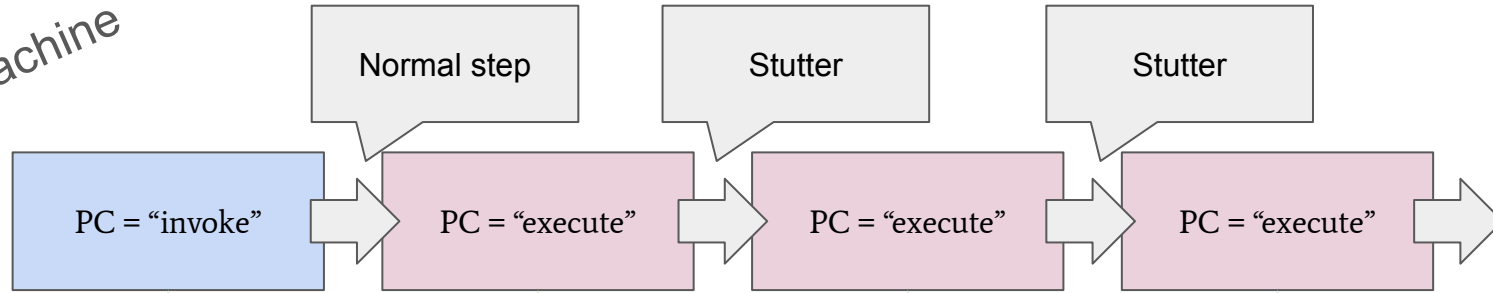
...But as a description of *reality*, this is very unsatisfying!

Most of our variables make intuitive sense, but...

Where can I find `client_pc` in the real world?

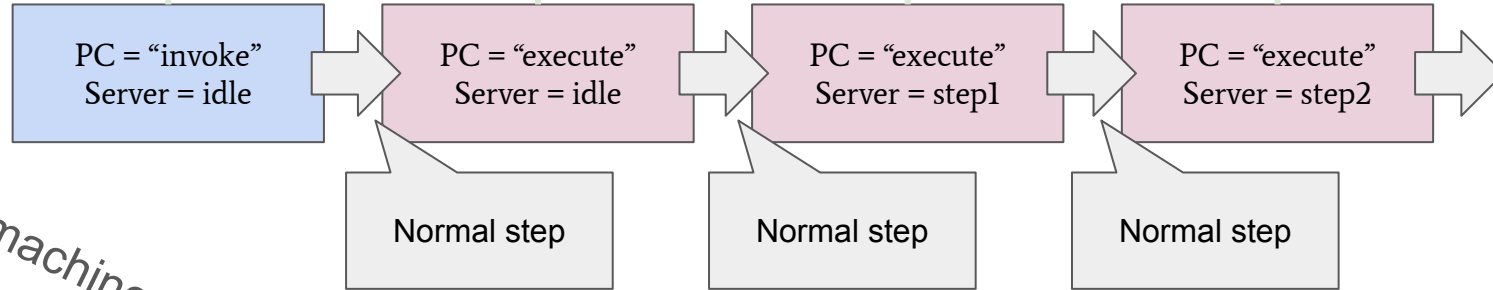
State Machine Refinement

High-level state machine



Refinement mapping:
Low-level state → High-level state

Low-level state machine



Setting up a refinement mapping

```
---- MODULE Impl ----
```

```
VARIABLES ...
```

```
I == INSTANCE LinearizableRegister WITH
```

```
  SomeConstant <- ...,
```

```
  Some_variable <- ...,
```

```
Safety == I!Spec
```

```
=====
```

Refinement mapping (or “abstraction function”) –
In terms of concrete state,
what is the abstract state?

Syntactic convenience:
 $x \leftarrow x$
can be omitted

All definitions (e.g. `Spec`) in
`HighLevelSpec.tla` now
available (e.g. `I!Spec`)

What should we describe in the low-level state machine?

Many choices for how it should work

- Single-server
- Primary-secondary via 2-phase commit
- Consensus-based replication

All the way down to

- Storage media: embedded database, raw block device, ...

Key challenge: identifying the “linearization point”

<diagram of abstract states init/exec/finish + lower-level states “establish connection to secondary”, “write”, “get response”, “write to local db”, ...>

Implementing the networked register

Today: let's keep it simple

...but connected to reality

Same system, but over the network!

- Need to specify the structure of network packets and our assumptions about how the network behaves.

Computer networks (application developer's view)

UDP

- `send(packet) / recv()` → packet
- All-or-nothing (no corrupted packets (assuming no bad actors))
- Maximum packet size :(

Computer networks (application developer's view)

TCP

- listen() / connect()
- send(bytes) / recv() → bytes
- “Stream-oriented”
- ...but actually, we usually just use it to send big packets (“messages”)
 - HTTP: request/response
 - gRPC: request/response
 - Java RMI ([JRMP](#)): request/response

Aside: TCP might not do what you think

commonly referred to as TCP/IP. TCP provides **reliable**

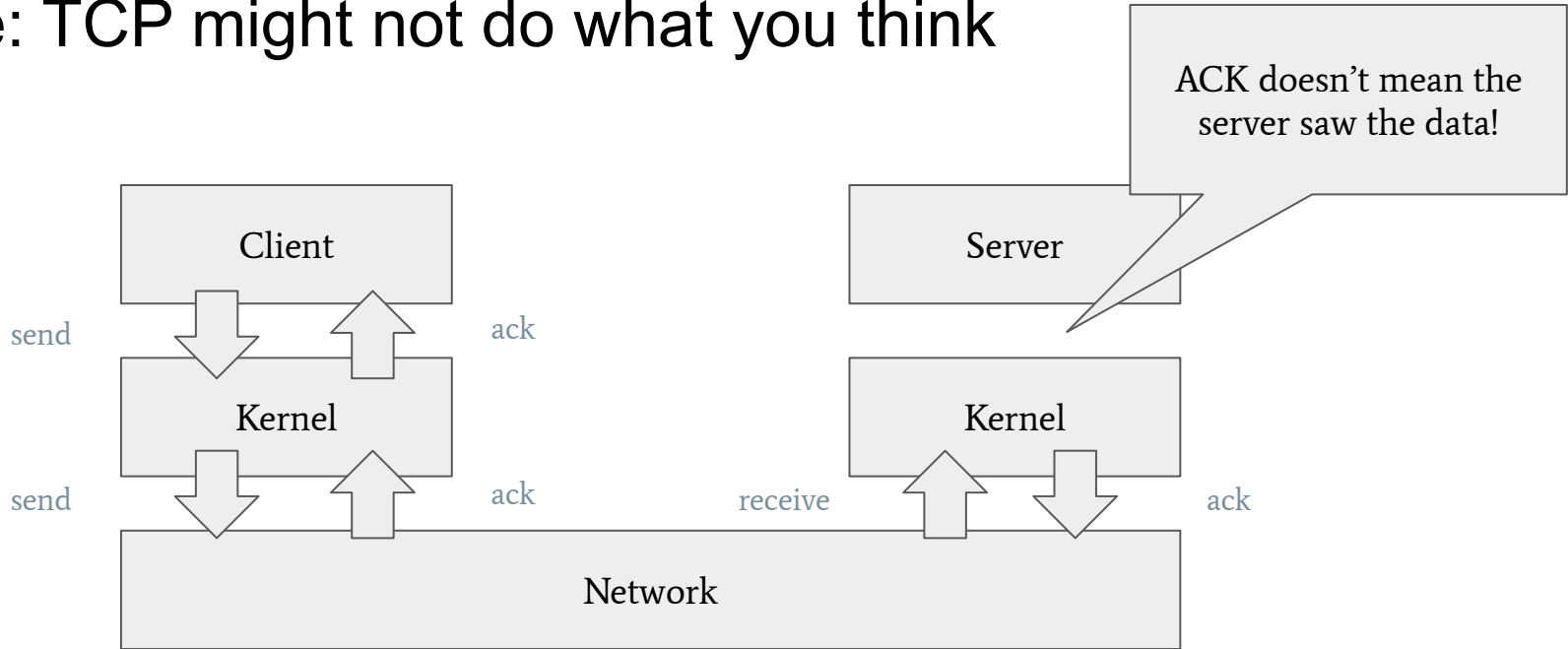
In computer networking, a **reliable** protocol is a communication protocol that notifies the sender whether or not the delivery of data to intended recipients was successful. Reliability is a synonym for **assurance**, which is the term used by the ITU and ATM Forum, and leads to **fault-tolerant messaging**.

Very easy to misunderstand!

“Reliable” is *not* “all data will be delivered”

“Reliable” is *not even* “all ack’d data will be delivered”

Aside: TCP might not do what you think



Aside: TCP might not do what you think

commonly referred to as [TCP/IP](#). TCP provides [reliable](#),

[reliable](#)
ajoi
and
s of
enc
tro
ctic

In computer networking, a **reliable** protocol is a communication protocol that notifies the sender whether or not the delivery of data to intended recipients was successful. Reliability is a synonym for **assurance**, which is the term used by the ITU and ATM Forum, and leads to **fault-tolerant messaging**.

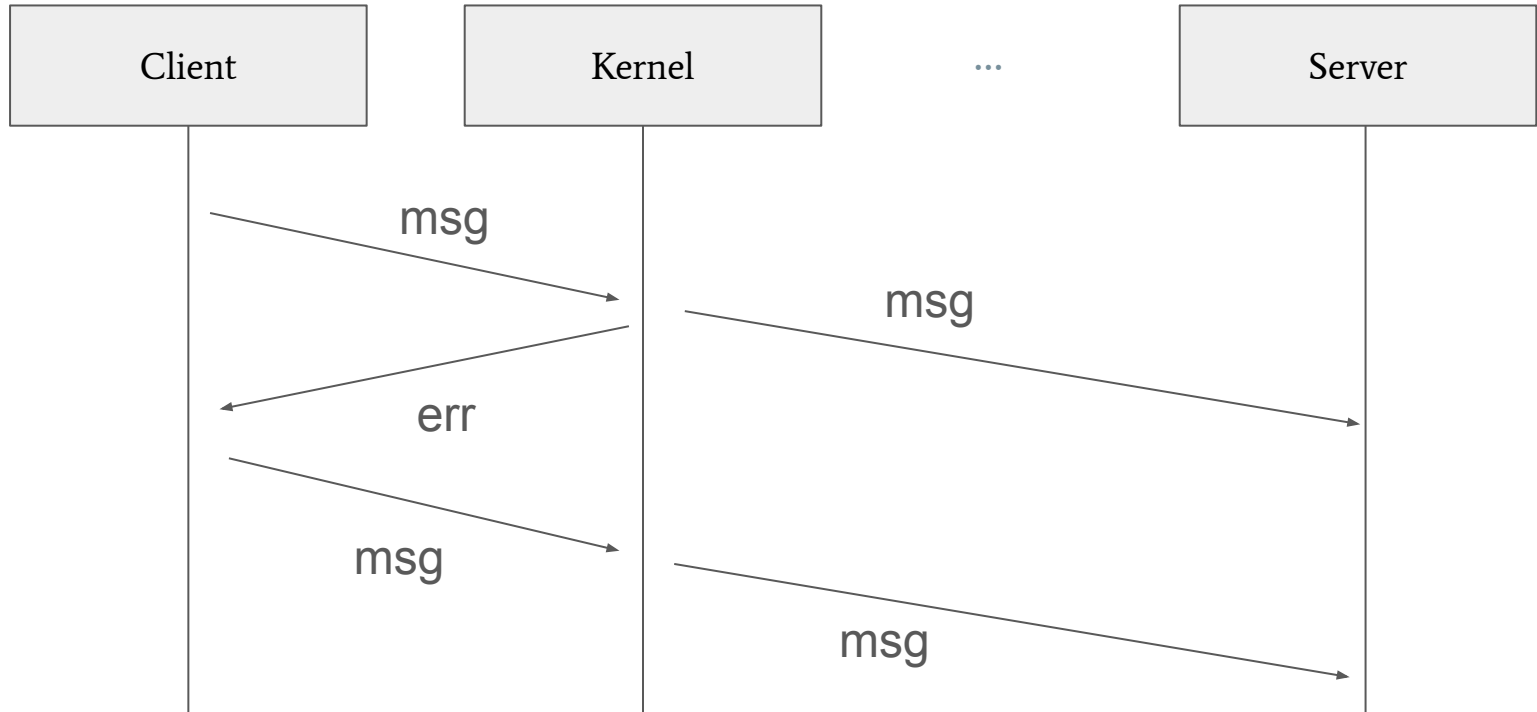
bli

Surely a reliable protocol protects me from duplication and reordering?

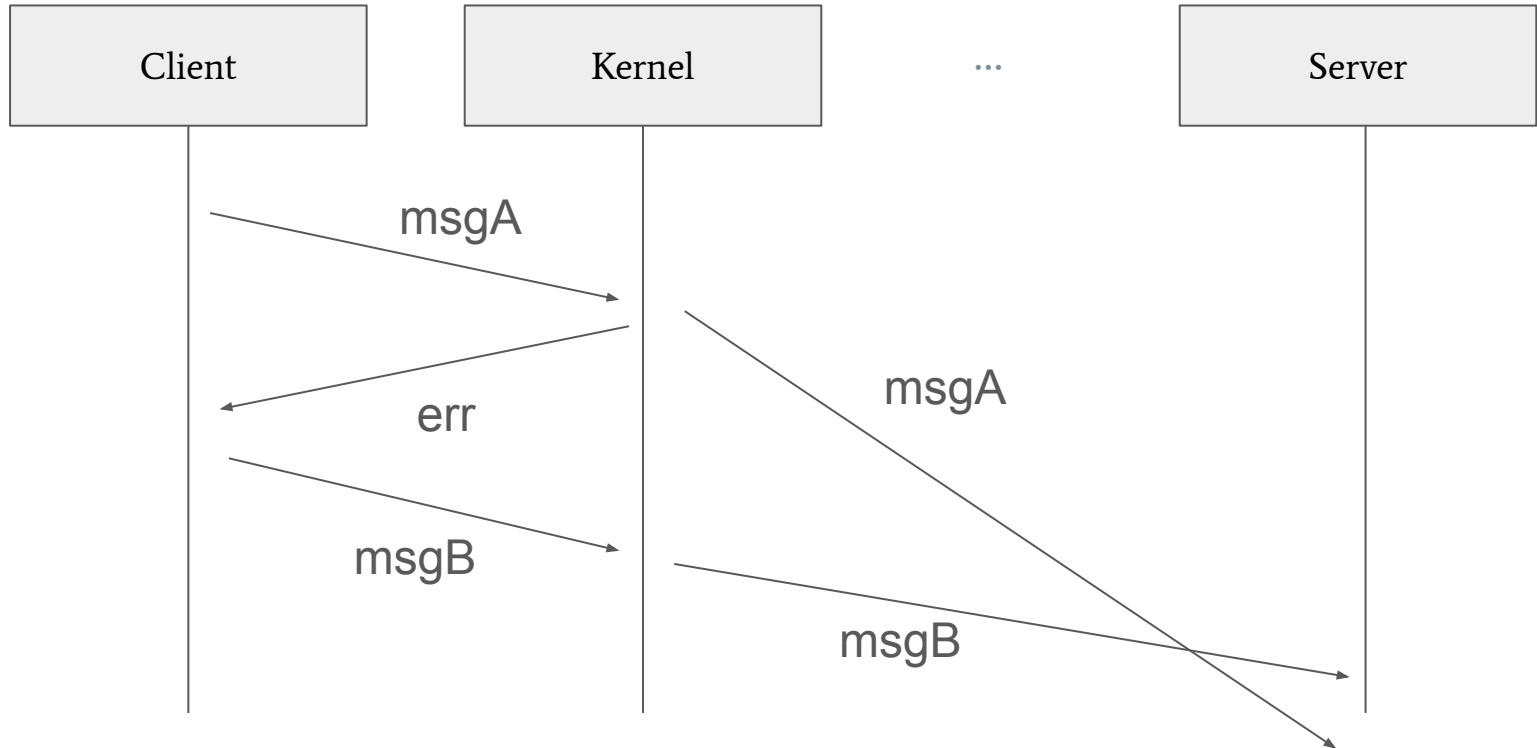
It does!

...but only if you are willing to let your application deadlock!

Aside: TCP might not do what you think



Aside: TCP might not do what you think



Hiding the Details: How best to model networks in TLA⁺?

Simplest model: bag-of-messages

- Generalizes UDP and TCP (using message-oriented protocol)
- Not well suited to request-response interaction

```
VARIABLE messages
```

```
Send(msg) ==  
  messages' = messages \union {msg}
```

```
Recv(msg) ==  
  msg \in messages
```

Hiding the Details: How best to model networks in TLA⁺?

Simplest model: bag-of-messages

- Generalizes UDP and TCP (using message-oriented protocol)
- Not well suited to request-response interaction
- Elegantly describes reordered/duplicated messages
- ...and also dropped messages!

Drop (msg) ==
messages' = messages \ {msg}

Not necessary!

Remember: TLA+ requires every spec to allow stutter steps—so we are already describing behaviors where the message simply isn't delivered, even without an explicit Drop action!

Hiding the Details: How best to model networks in TLA⁺?

More advanced model: communication “channels”

- Specifying Systems, ch3

```
VARIABLES requests, responses

SendRequest(client, req) ==
  requests' =
    [requests
     EXCEPT ![client] = @ \o <<req>>]

RecvRequest(client, req) ==
  /\ Requests[client] /= <<>>
  /\ requests[client][1] = req
  /\ requests' = [requests EXCEPT ![client] = Tail(@)]
```

Client can queue up multiple requests over the same channel (e.g. HTTP pipelining)

...but in practice the client will usually wait for a response

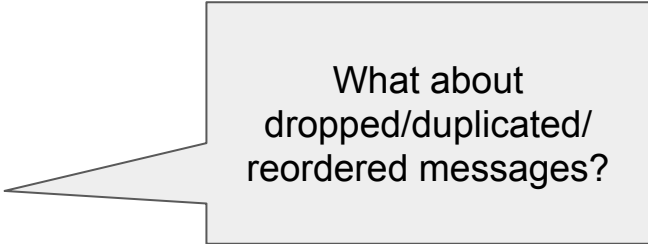
Hiding the Details: How best to model networks in TLA⁺?

We've already been using something like channels, but with at most one outstanding message:

```
VARIABLES requests, responses

SendRequest(client, req) ==
  /\ client \notin DOMAIN requests
  /\ requests' = (client :> req) @@ requests

RecvRequest(client, req) ==
  /\ client \in DOMAIN requests
  /\ requests[client] = req
  /\ requests' = [c \in (Clients \ {client}) |-> requests[c]]
```



What about
dropped/duplicated/
reordered messages?

Exercise 2: toward an implementation

A bit more realism:

- Clients only have two PC values:
 - “send_request”
 - “receive_response”
- Execution of the request is handled by the server

Need a clever refinement mapping!

Exercise 3: closer toward an implementation!

Let's go even farther:

- In reality, the server can't update `register_value` *and* send a response in one atomic action!
- Let's split the server into multiple steps.

Need a *very* clever refinement mapping!

Summary

What we've done today is incredibly cool

- Full spec of a real(ish) system
- Strong evidence (via model checking) that we can implement the design using TCP

Next time:

- Filesystems
- Eventual consistency
- Crashes
- Liveness