

# Practical TLA<sup>+</sup> for Concurrent and Distributed Systems

Calvin Loncaric

Oracle, Inc.

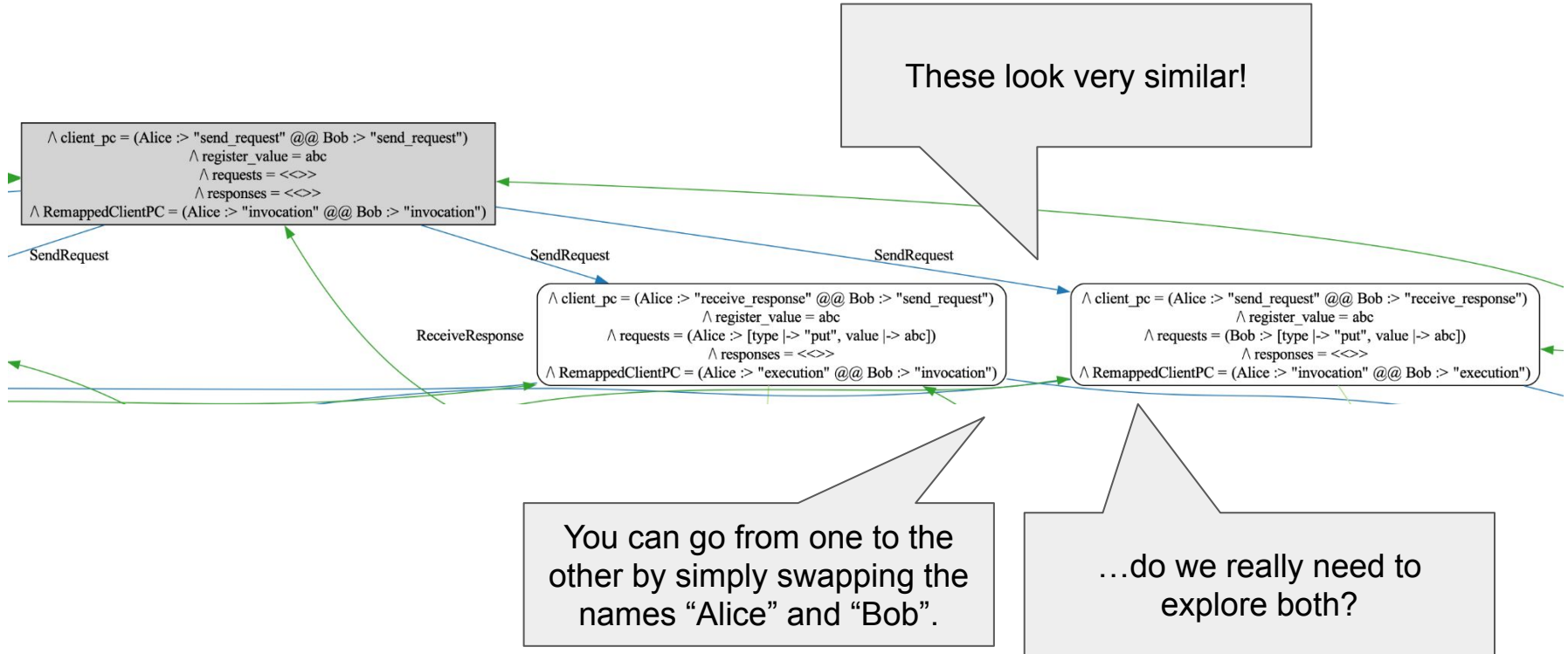
# Today

- Symmetry reduction
- Filesystems
- Exercise!
- Liveness and “Eventual consistency”
- Exercise!

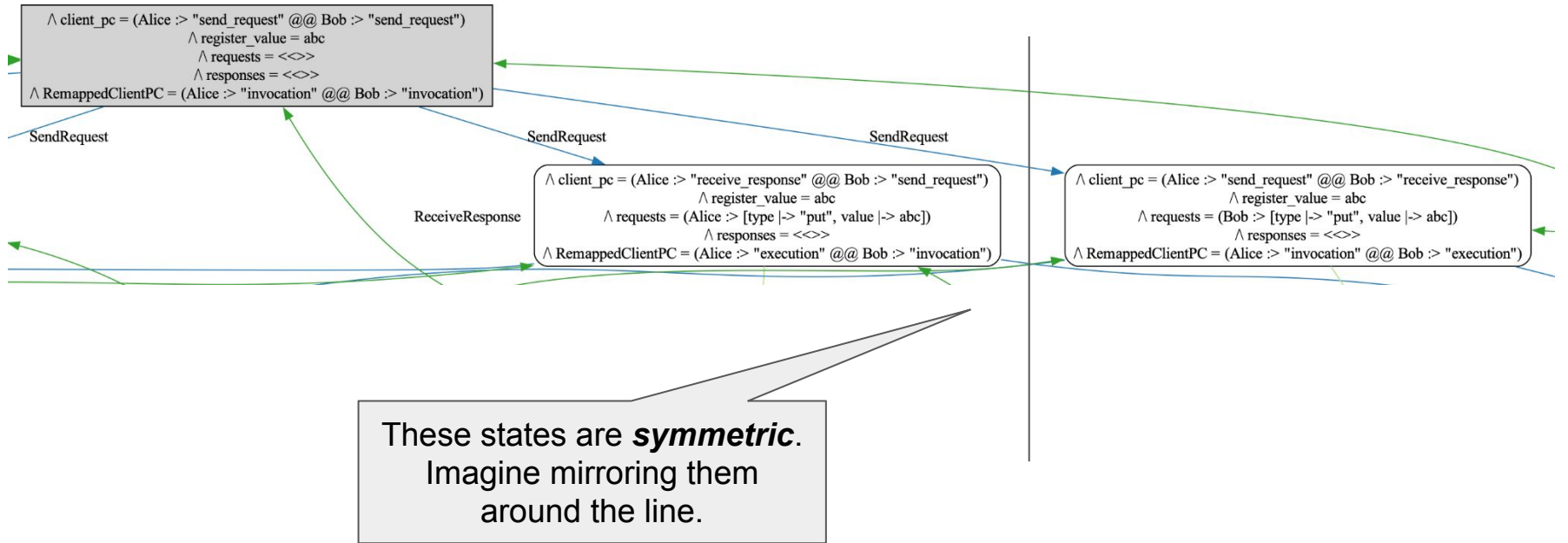
# State Space Explosion

<demo>

# Hmm...



# Symmetry



# Permutations

A **permutation** of a set  $S$  is a bijection from  $S$  to itself

Available in TLA<sup>+</sup>:

**Permutations** ( $S$ ) ==  
 $\{f \in [S \rightarrow S] : \forall w \in S : \exists v \in S : f[v]=w\}$

# Exploiting symmetries



When visiting either of these states, TLC will use any declared permutations to convert it to a single canonical example

How is this better?

**No need to visit successors if the canonical example has already been explored!  
=> Potentially exponential savings!**

## A bit more formal

We can now define what symmetry means. A specification *Spec* is *symmetric with respect to* a permutation  $\pi$  iff the following condition holds: for any behavior  $\sigma$ , formula *Spec* is satisfied by  $\sigma$  iff it is satisfied by  $\sigma^\pi$ .

Specifying Systems, ch14



**Today, TLC's simple symmetry-breaking does *not* work for liveness properties! Only for safety.**

# So when should I use symmetry sets in TLC?

- The set only contains “model values”
- Your spec must be symmetric with respect to permutations of the set
- Quick litmus test: no “distinguished” elements
  - i.e. no use of `CHOOSE x \in S`
- Only checking safety properties

# “View” symmetry

(There exists a more general symmetry mechanism not covered here.)

# Recap: the networked register

Contract: linearizability

\*\* Reminder: we described the write to disk as a single atomic action.

- Is that realistic?

**Server** ==

```
\E client \in DOMAIN requests:
/\ LET request == requests[client] IN
  IF request.type = "get" THEN
    /\ ...
  ELSE \* request.type = "put"
    /\ register_value' = request.value
    /\ responses' = MapPut(responses, client, [type |-> "put"])
  /\ requests' = MapDel(requests, client)
  /\ UNCHANGED <<client_pc>>
```

# Disks and Durability

Basic idea: what you write to disk will stay there unchanged until either

- You overwrite it
- ...or the disk fails

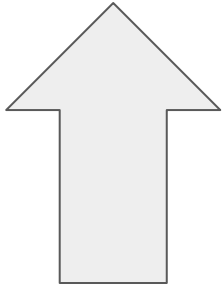
*Crucially:* Data survives the death of your process, e.g. due to

- Bugs in your program
- Power outages
- The OOM killer
- Operator intervention (`$ kill -KILL <PID>`)

# Interacting with disks

Linux, Mac, Windows all work like this:

Userspace  $\xrightarrow{\text{[Filesystem API]}}$  Kernel  $\xrightarrow{\text{[iSCSI]}}$  Disk



# Working with files

Point to POSIX API

open() / close()



read() / write()

fsync()

## NAME



fsync — synchronize changes to a file

## SYNOPSIS

```
[FSC]  #include <unistd.h>  
  
int fsync(int fil-des); 
```

## DESCRIPTION

The *fsync()* function shall request that all data for the open file descriptor named by *fil-des* is to be transferred to the storage device associated with the file described by *fil-des*. The nature of the transfer is implementation-defined. The *fsync()* function shall not return until the system has completed that action or until an error is detected.

[SIO]  If `_POSIX_SYNCHRONIZED_IO` is defined, the *fsync()* function shall force all currently queued I/O operations associated with the file indicated by file descriptor *fil-des* to the synchronized I/O completion state. All I/O operations shall be completed as defined for synchronized I/O file integrity completion. 

# Working with files: Example

```
int f = open("file", O_WRONLY | O_CREAT);  
write(f, "hello", 5);  
close(f);
```

Error handling  
omitted for clarity

I will omit most  
arguments going  
forward

# Up next: getting our feet wet by understanding the pitfalls

Repeatedly:

- Look at an example
- I'll point out why it's wrong
- We'll correct our understanding and continue

Reminder: we're building toward a robust understanding of the filesystem API so we can

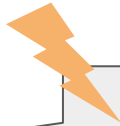
- Describe it formally
- Prove something about a system that uses it!

# Example 1/?

```
open()
```

```
write(data)
```

```
close()
```



After power outage here, file is empty.

Where is my data??

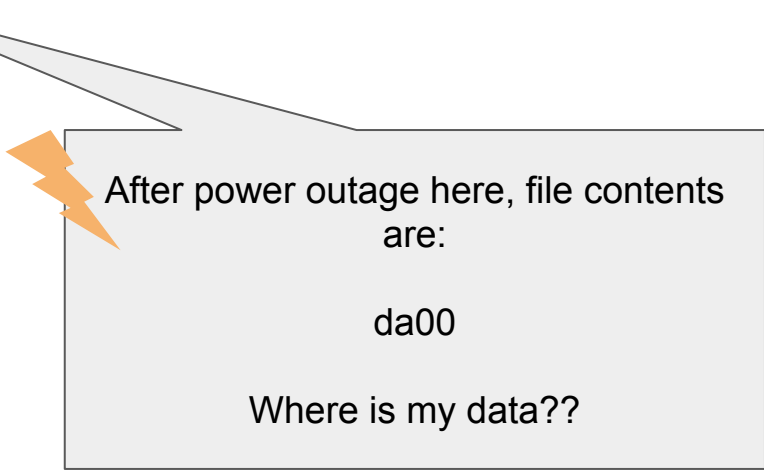
# Example 2/?

```
open()
```

```
write(data)
```

```
fsync()
```

```
close()
```



After power outage here, file contents  
are:

da00

Where is my data??

# Example 3/?

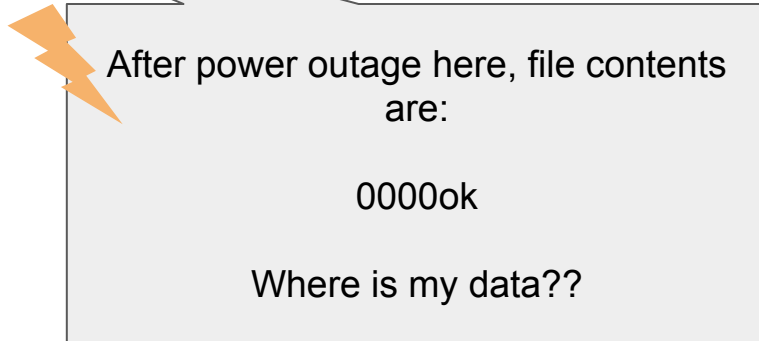
```
open()
```

```
write(data)
```

```
write(ok) ← requires collaboration with read path: on read, check OK bit
```

```
fsync()
```

```
close()
```



After power outage here, file contents are:

0000ok

Where is my data??

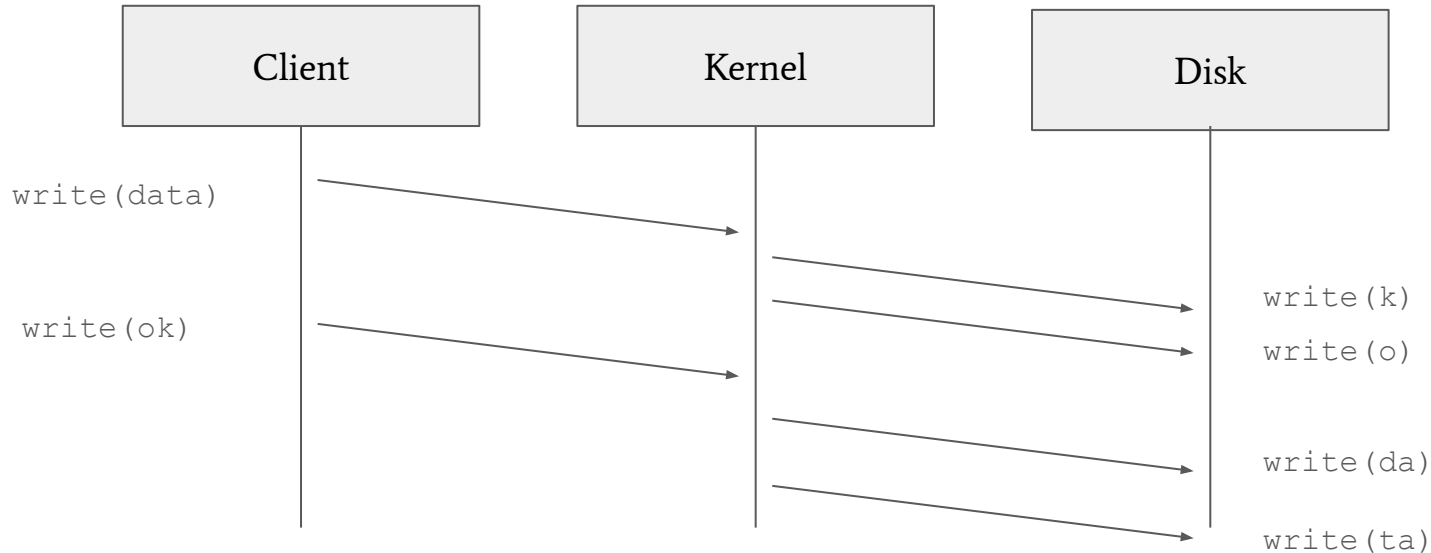
# Write Reordering

“Specifying and Checking File System  
Crash-Consistency Models”

<https://sandcat.cs.washington.edu/ferrite/ferrite-asplos16.pdf>

Did you think write() was linearizable?

Think again!



# Example 4/?

```
open()
```

```
write(data)
```

```
fsync()
```

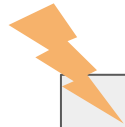
```
write(ok)
```

```
fsync()
```

```
close()
```

Working toward a global invariant:

File contains “ok” => File contains “data”



After power outage here, file doesn't even exist!

The directory is empty!

# Directory entries and operations

`open()` is a modification to the *containing directory*

New things to worry about:

- `rename()`
- `unlink()` ← aka ``rm``
- `fsync()` ← it works on directories too!

# Example 5/?

```
open(file)
```

```
write(data)
```

```
fsync()
```

```
write(ok)
```

```
fsync()
```

```
close(file)
```

```
open(dir)
```

```
fsync()
```

```
close(dir)
```

Suppose the Kernel decides to send the  
create-file writes to disk HERE.

BUT: a disk error occurs.

Who gets the error? -> Nobody!

It affected Postgres!

“errors could easily be lost so that  
no application would ever see  
them”

<https://lwn.net/Articles/752063/>

Algorithm runs to completion **error-free**,  
but the directory is still missing the file!

Where is my data?

# Example 6/?

```
open (dir)
```

```
open (file)
```

```
fsync (dir)
```

```
close (dir)
```

```
write (data)
```

```
fsync ()
```

```
write (ok)
```

```
fsync ()
```

```
close ()
```

# Example 7/?

```
open(tmp)
```

```
write(data)
```

```
fsync()
```

```
write(ok)
```

```
fsync()
```

```
close()
```

```
open(dir)
```

```
rename(tmp, file)
```

```
fsync()
```

```
close(dir)
```

## Example 8/?

```
open(tmp)
```

```
write(data)
```

```
fsync()
```

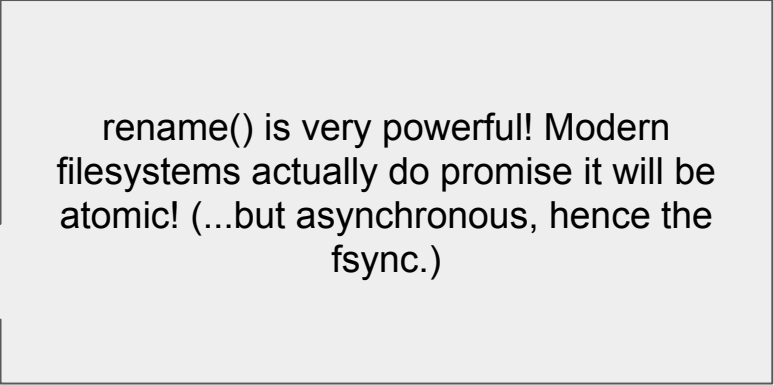
```
close(file)
```

```
open(dir)
```

```
rename(tmp, file)
```

```
fsync()
```

```
close(dir)
```



**rename() is very powerful! Modern filesystems actually do promise it will be atomic! (...but asynchronous, hence the fsync.)**

# Formalizing in TLA<sup>+</sup>

```
VARIABLE file
```

```
TypeOK == file \in [  
  kernel_cache: Seq(Byte),  
  on_disk:      Seq(Byte)]
```

open()/read()/write()/rename()/unlink() interact with `file.kernel_cache` only!

`fsync()` is our *only* tool for working with `file.on_disk`:

- Kernel “synchronizes” cached and durable state by pushing writes to disk (in whatever order it sees fit)

also:

- Kernel is free to synchronize *early*! It can push writes to `durable_state` whenever it wants.

# Modeling write()

```
Write(data) ==  
    file' = [file EXCEPT  
            !.kernel_cache = data]
```

# Modeling fsync()

**FlushOneByte** ==

```
\E index \in DOMAIN file.kernel_cache:  
\ \ index \in DOMAIN file.on_disk  
\ \ file' = [file EXCEPT  
    !.on_disk[index] = file.kernel_cache[index]]
```

**FSync** ==

\ pc = "fsync"  
\ file.on\_disk = file.kernel\_cache  
\ pc' = ...

# Modeling Power Outages

```
PowerOutage ==  
    file' = [file EXCEPT  
            !.kernel_cache = file.on_disk]
```

# Exercise 4: implementing the networked register using filesystem writes

 **Extremely Hard** 

Two big options:

1. Expand the provided scaffolding to include directories and use the `rename()` approach, or
2. Devise a way to keep everything in one file so you do not need to use slow `rename()` shenanigans.

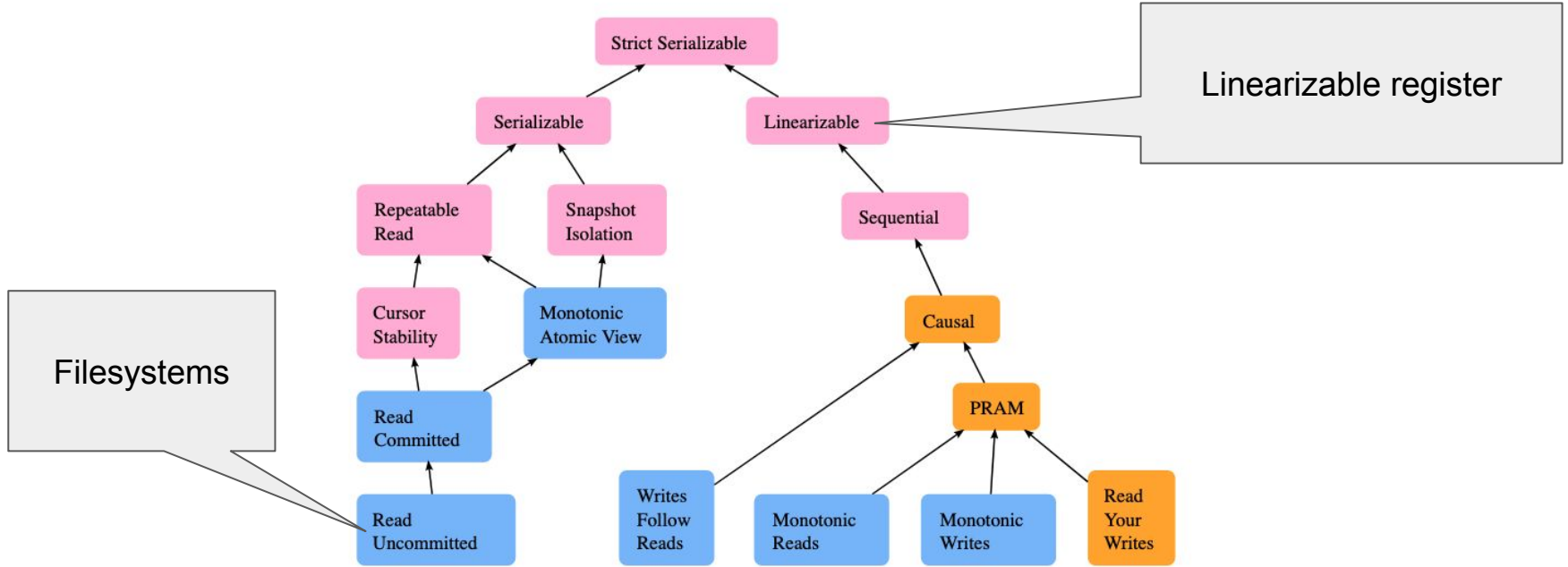
Write a refinement mapping and show using TLC that the system is still linearizable.

Tricky, because the linearization point will be during an `fsync()` call!

# Takeaway

Difficulty (and necessity!) of building stronger guarantees on a weaker system

# Consistency: A Spectrum

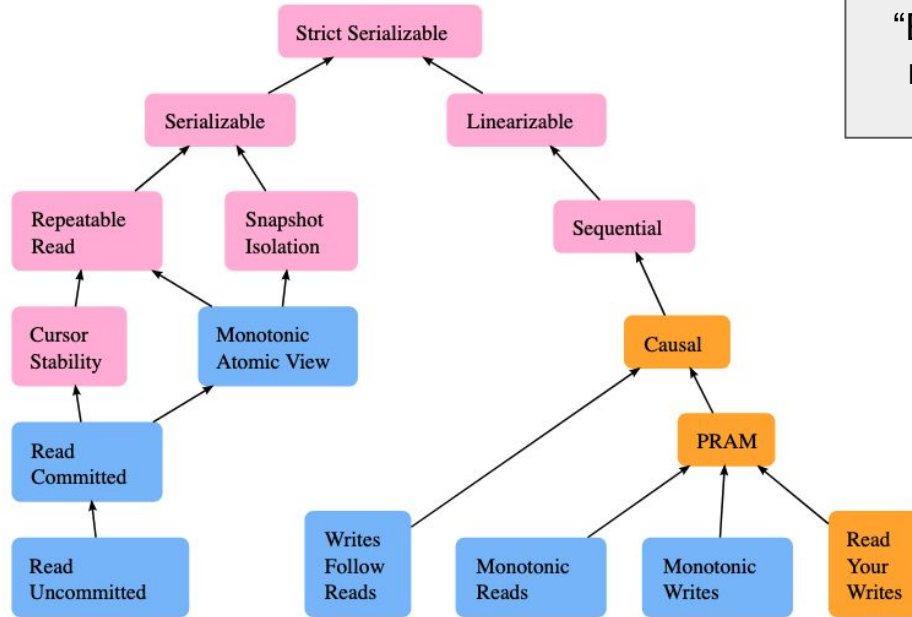


<https://jepsen.io/consistency/models>

# “Eventual Consistency”

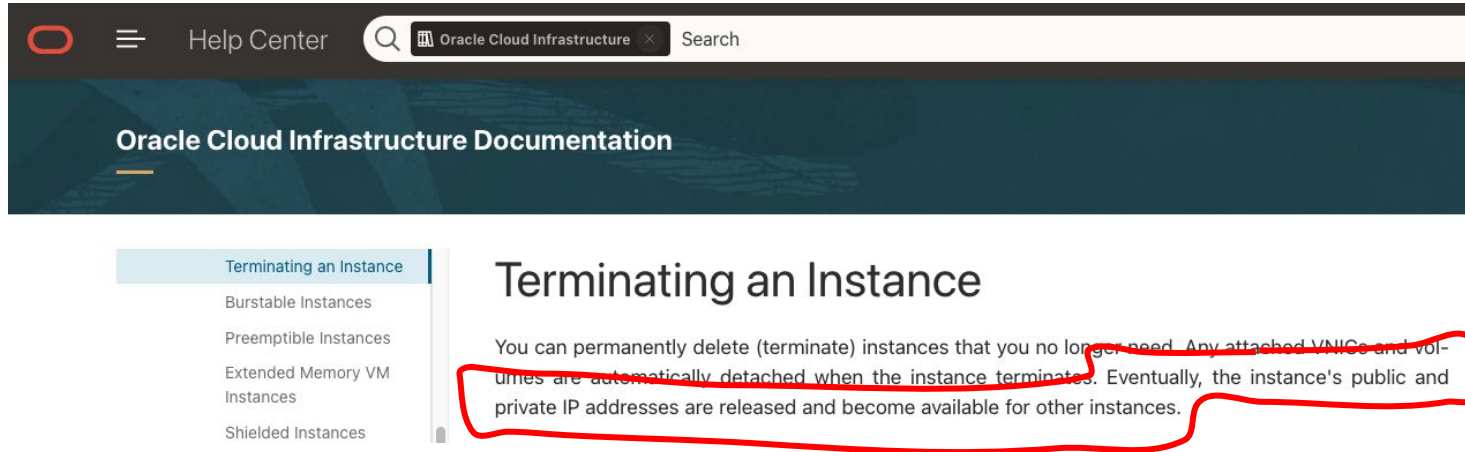
Imprecise!

“Eventual consistency”  
refers to a *big* family!



<https://jepsen.io/consistency/models>

# Eventual consistency in the cloud



The screenshot shows the Oracle Cloud Infrastructure documentation page for "Terminating an Instance". The page header includes the Oracle logo, a menu icon, "Help Center", a search bar with "Oracle Cloud Infrastructure" and a search icon, and the text "Oracle Cloud Infrastructure Documentation". The left sidebar lists navigation options: "Terminating an Instance" (highlighted), "Burstable Instances", "Preemptible Instances", "Extended Memory VM Instances", and "Shielded Instances". The main content area has the title "Terminating an Instance" and a paragraph: "You can permanently delete (terminate) instances that you no longer need. Any attached VNICs and volumes are automatically detached when the instance terminates. Eventually, the instance's public and private IP addresses are released and become available for other instances." A red hand-drawn circle highlights the sentence: "Eventually, the instance's public and private IP addresses are released and become available for other instances."

<https://docs.oracle.com/en-us/iaas/Content/Compute/Tasks/terminatinginstance.htm>

# Termination Workflow in TLA<sup>+</sup>

```
/\ instance_exists  
  /\ ip_allocated
```

```
/\ ~instance_exists  
  /\ ip_allocated
```

```
/\ ~instance_exists  
  /\ ~ip_allocated
```

# Why the delay?

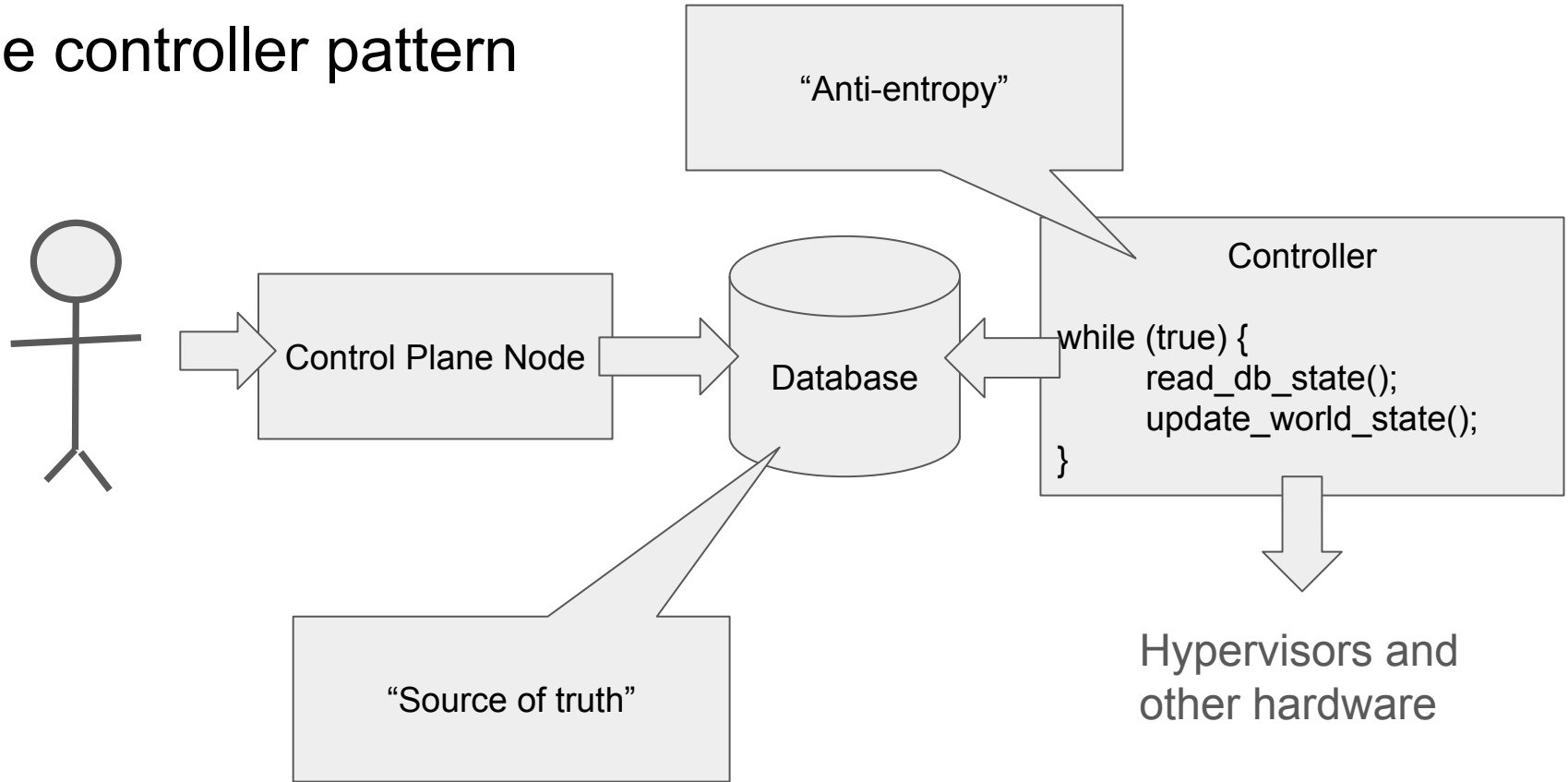
Termination is (at least) 2 steps:

1. Tell hypervisor to delete VM
2. Reconfigure routing tables now that it no longer owns IP

These steps cannot be done atomically! – fundamental limitation of the systems we're building on (hypervisor + routers)

...so what happens if the server executing this workflow crashes after step 1?

# The controller pattern



# Exercise 5: Eventual Consistency

Scaffolding provided for you:

- Spec we described earlier (with WF condition!)
- Client, hypervisor, and smartnic

Implement:

- API server + DB (recv loop)
- Controller (while true do something)

\*\* reminder: fairness in pluscal

# What does fairness really *mean*?

We had to add fairness conditions, but...

What does that mean for an organization running this software in their datacenter?

=> you can let it go down, but *not for very long*

=> need observability on how long IPs wait to be reusable

^ Lots of complexity!

# Call to Action

We are all building on quicksand.

Go push the state of the art so we can verify everything!

## Get involved with TLA<sup>+</sup>!

- <https://www.tlapl.us>
- <https://github.com/tlaplus>