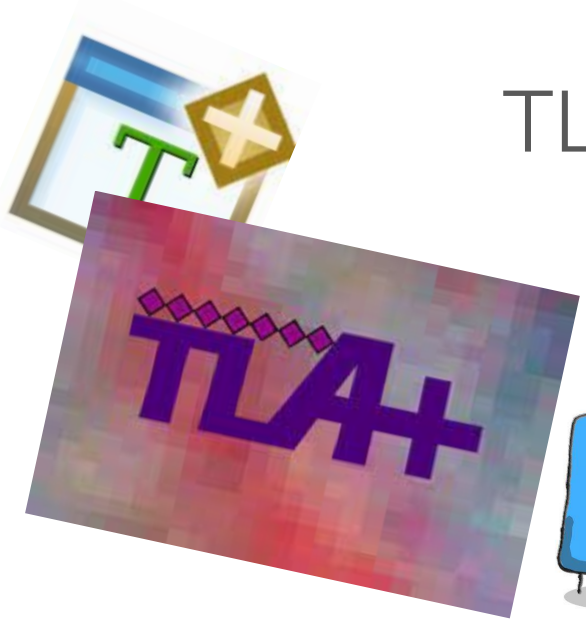
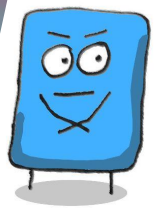
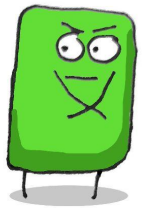
Distributed Systems Showdown

TLA+ vs Real Code







... and why we model

Jack Vanlightly

www.jack-vanlightly.com
@vanlightly

Principal Software Engineer

splunk>

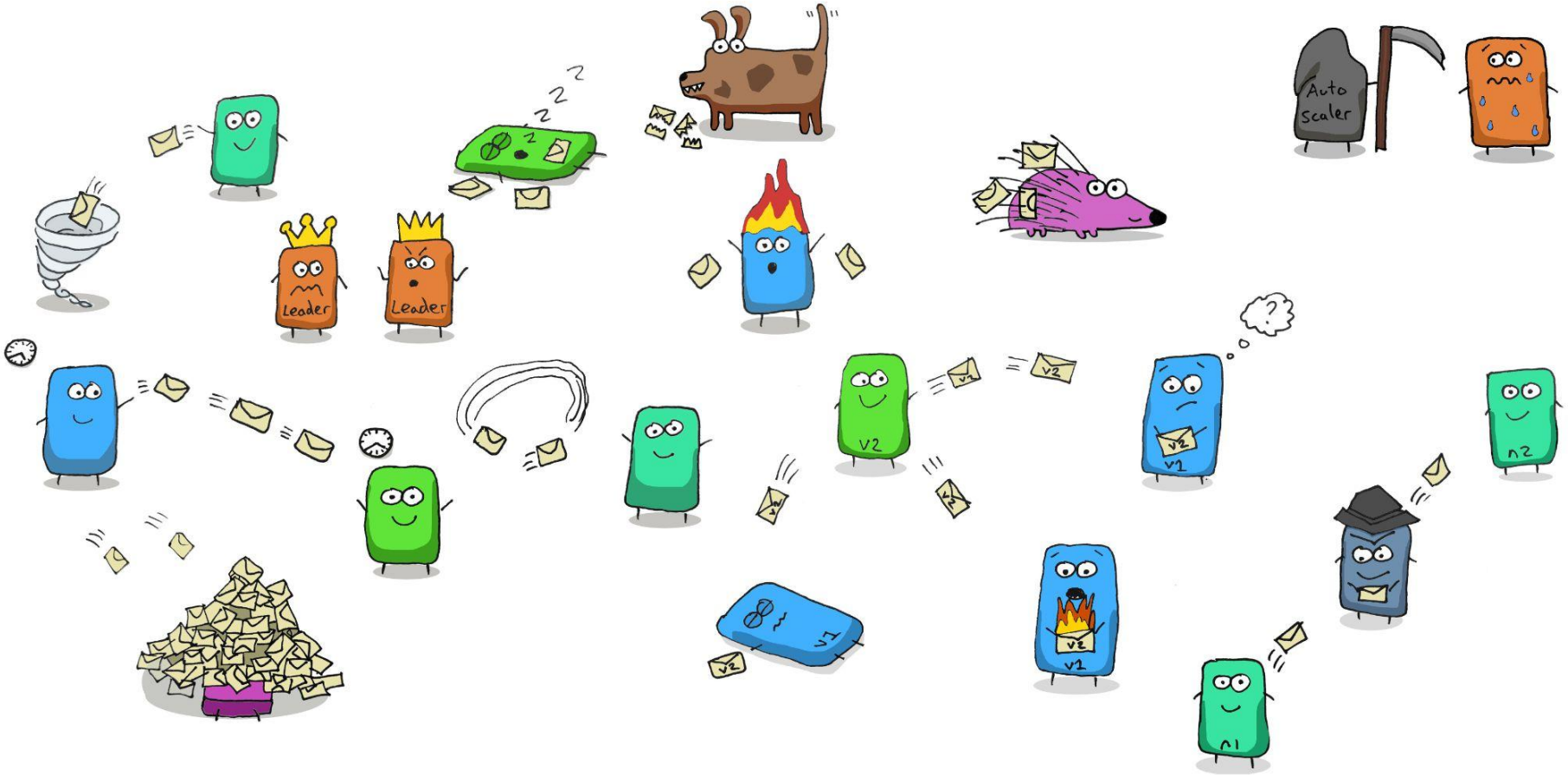
Apache BookKeeper™

PULSAR

What we'll cover...

- Why do modelling at all?
- Quick look at TLA+
- Quick look at Maelstrom/Jepsen
- Case study to compare them
  - Modelling a distributed log storage system

Why do modelling and verification anyway?



PIP-31: Transactional Streaming

Motivation

This document outlines a proposal for supporting transactional messaging at Apache Pulsar. Transactions are used for strengthening the message delivery semantics of Apache Pulsar and processing guarantees at Pulsar Functions.

The highest message delivery guarantee that Apache Pulsar currently provides is `exactly-once` producing at one single partition via idempotent Producer . Users are guaranteed that every message produced to one single partition via an idempotent Producer will be persisted exactly once, without data loss. There is no `atomicity` when producing attempts to produce messages to multiple partitions. For instance, a publish failure can occur on the broker published and if the producer doesn't retry or has exhausted its retry count, the messages will not be written to pulsar. On the consumer side, the known acknowledgment cursor may be a best effort operation, which will result in message redelivery here the consumer will receive duplicate messages. Pulsar only guarantees at-least-once consumption for consumers.

Similarly, Pulsar Functions only guarantees exactly-once processing on single event for an idempotent function. It can't guarantee processing multiple events or producing multiple results can happen exactly. For example, if a function accepts multiple events and produces a result (e.g. windowing functions), the function can fail between producing the result and acknowledging the incoming messages, or even between acknowledging individual events. This will cause all (or some) incoming messages being re-delivered and reprocessed, and a new result is generated.

Users of Pulsar and Pulsar Functions will greatly benefit from transactional semantic support. Every message written or processed will be happening exactly once, without duplicates and without data loss - even in the event of broker or individual failures. A transactional messaging system that only makes writing applications using Pulsar or Pulsar Functions easier, it expands the scope which Pulsar can provide.

Use Cases

Transaction Flow

All transaction implementations can be shaped using these key components/concepts described in the above sections.

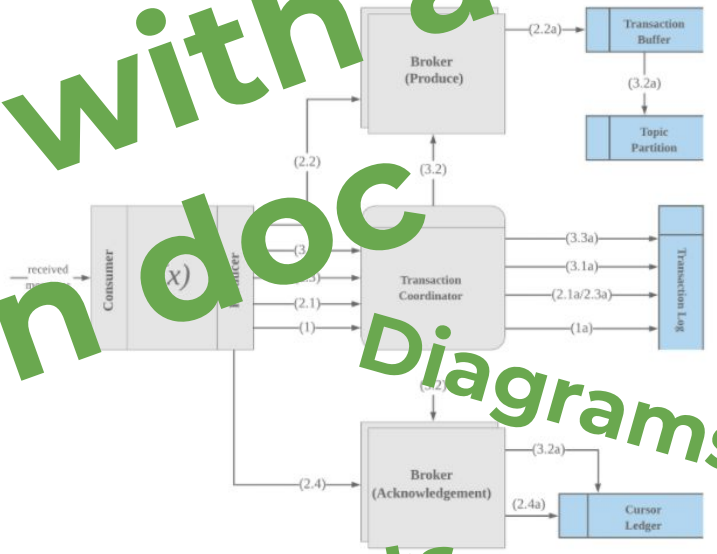


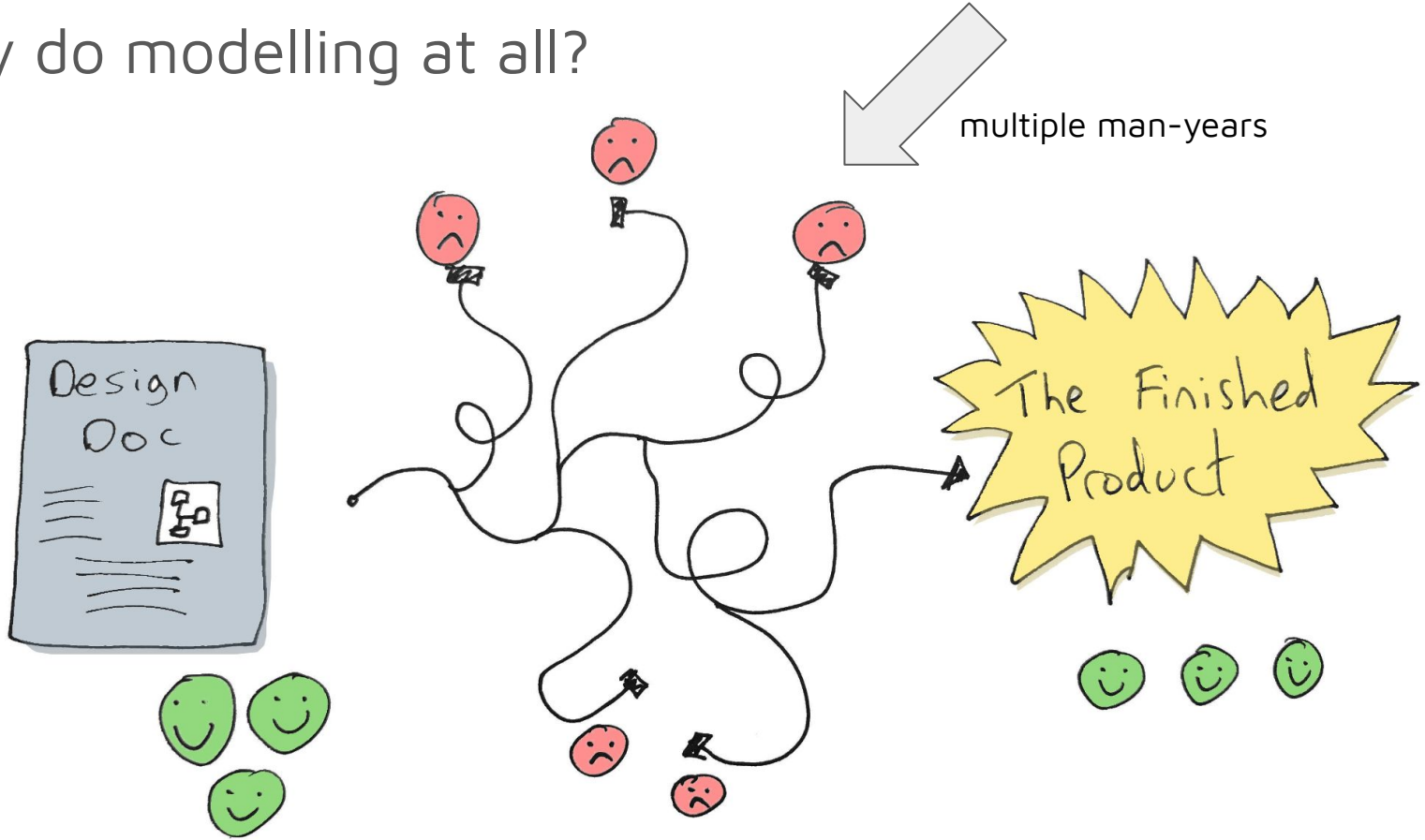
Figure 1 Transaction Flow

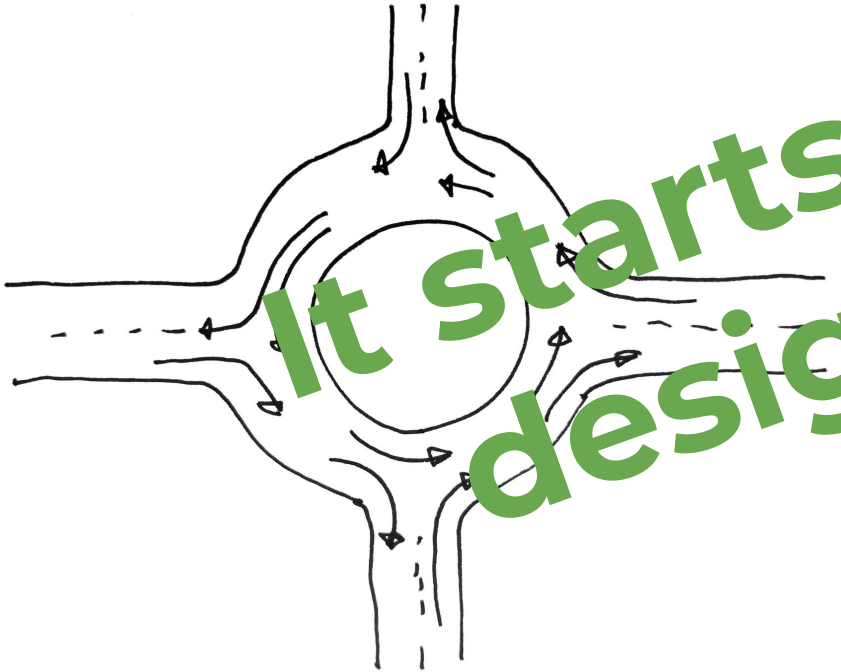
Figure 1 Transaction Flow:

- The gray square boxes represent distinct brokers.
- The gray rounded boxes represent logical components which can be running inside a broker or as a separated service (e.g. like how we run function worker as part of broker)
- All the blue boxes represent logs. The logs can be a pulsar topic, a bookkeeper ledger, or a managed ledger.

Motivation
Background
Context
It starts with a design doc
Diagrams
Components
Interactions

Why do modelling at all?





Roundabout Design Document

Motivation

Intersections with traffic lights block the free flow of traffic which causes unnecessary extra wait time and also more pollution due to their stop/start nature.

We propose a new form of intersection which is more free flowing, causing less pollution and less accidents.

The Roundabout

A **roundabout** is a proposed type of circular intersection or junction in which road traffic is permitted to flow in one direction around a central island, and priority is typically given to traffic already in the junction.

Compared to stop signs, traffic signals, roundabouts will reduce the likelihood and severity of collisions greatly by reducing traffic speeds and minimizing T-bone and head-on collisions. Variations on the basic concept include integration with tram or train lines, two-way flow, higher speeds and many others.

Traffic exiting the roundabout comes from one direction, instead of three, simplifying the pedestrian's visual environment. Traffic moves slowly enough to allow visual engagement with pedestrians, encouraging deference towards them. Other benefits include reduced driver confusion associated with perpendicular junctions and reduced queuing associated with traffic lights. They allow U-turns within the normal flow of traffic, which often are not possible at other forms of junction. Moreover, since vehicles that run on gasoline averagely spend less time idling at roundabouts than at signalled intersections, using a roundabout potentially leads to less pollution. When entering vehicles only need to give way, they do not always perform a full stop; as a result, by keeping a part of their momentum, the engine will produce less work to regain the initial speed, resulting in lower emissions. Research has also shown that slow-moving traffic in roundabouts makes less noise than traffic that must stop and start, speed up and brake.

It starts with a design doc

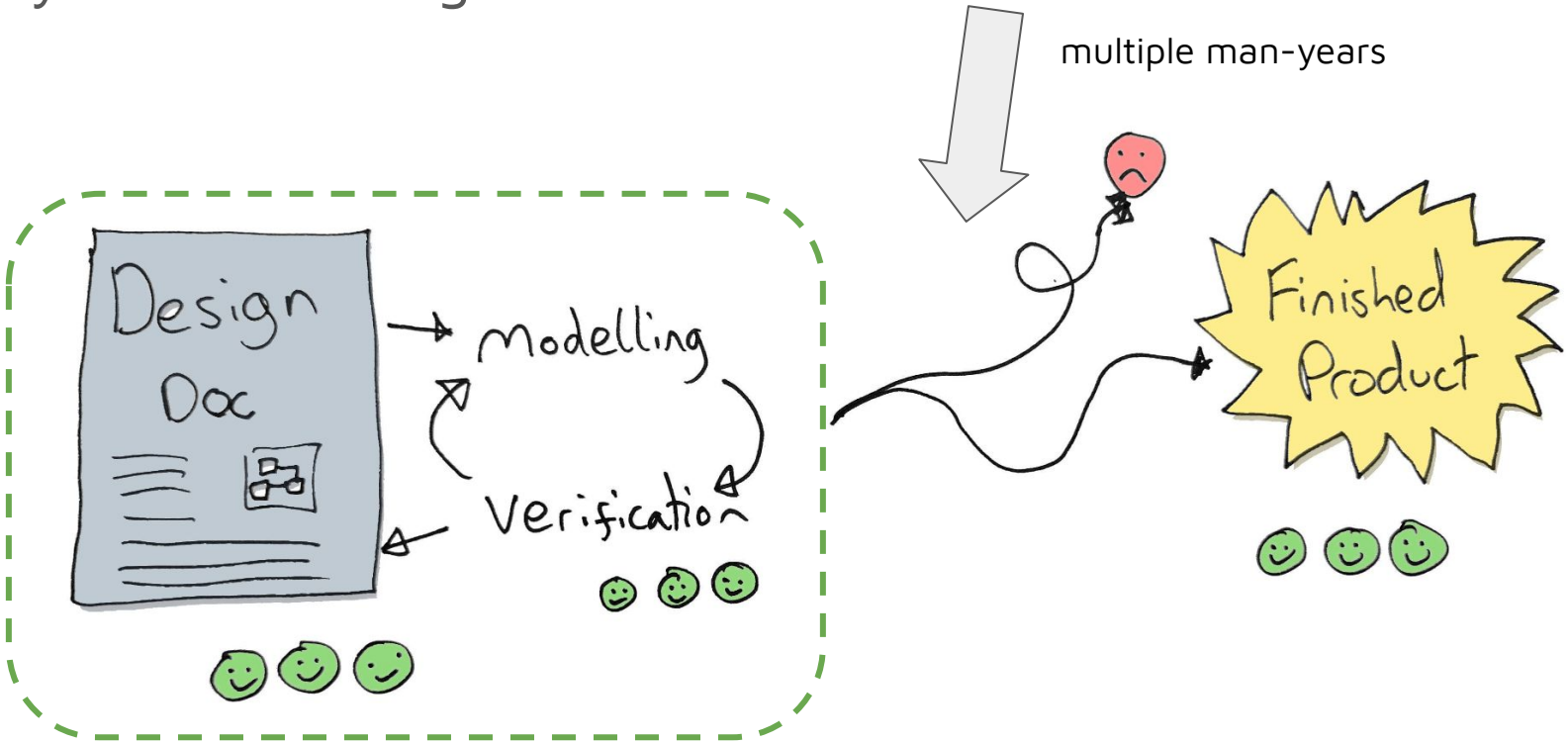

Layton Highway
Chester Street
Cypress Bridge
Ashton White Bridge
Layton Highway
Layton Highway
Ashton White Street
Cypress Bridge
Rosemary Highway


Ashton White Street
Layton Highway
Hunter Bridge
Cypress Street
Ashton White Bridge
Layton Highway
Hunter Bridge
Ashton White Street
Layton Highway
Layton Highway
Ashton White Street
Layton Highway


Leyton Highway
Barlow Bridge
Leyton Highway


Cedar Street
Chapman Highway

Why do modelling at all?



What properties should our models have?

Small
Malleable

Reduced to core behaviour

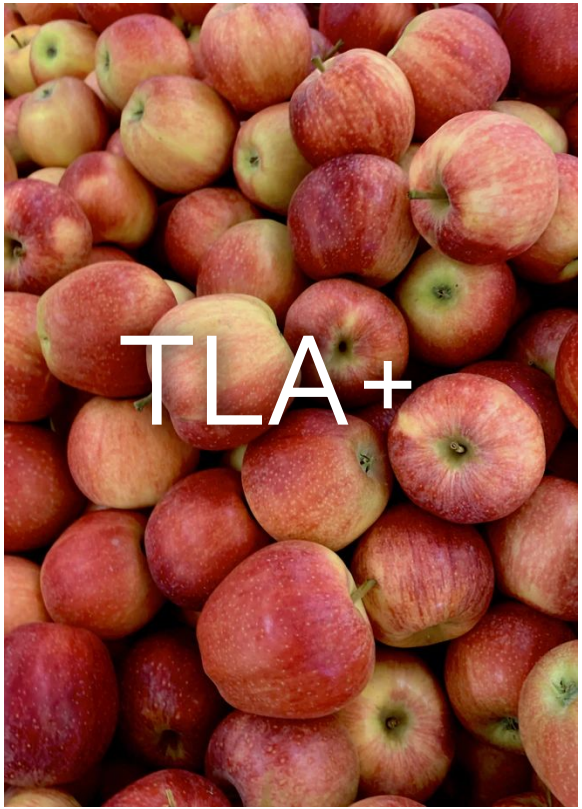
“Easy” to internalize

Free of extraneous clutter

Verifiable

“Living” doc?

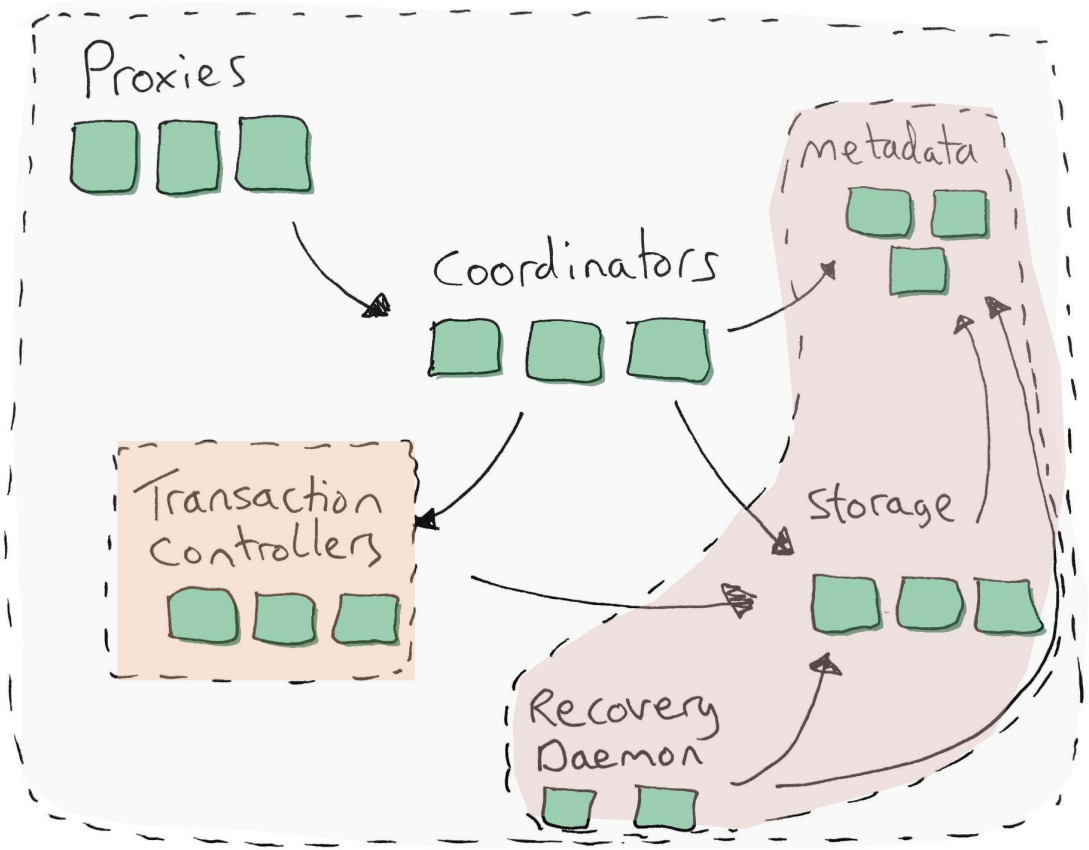
Modelling and Verification with Two Different Tools



A Quick Look at TLA+

TLA+ Specifications

TLA+ Arbitrary Levels of Abstraction

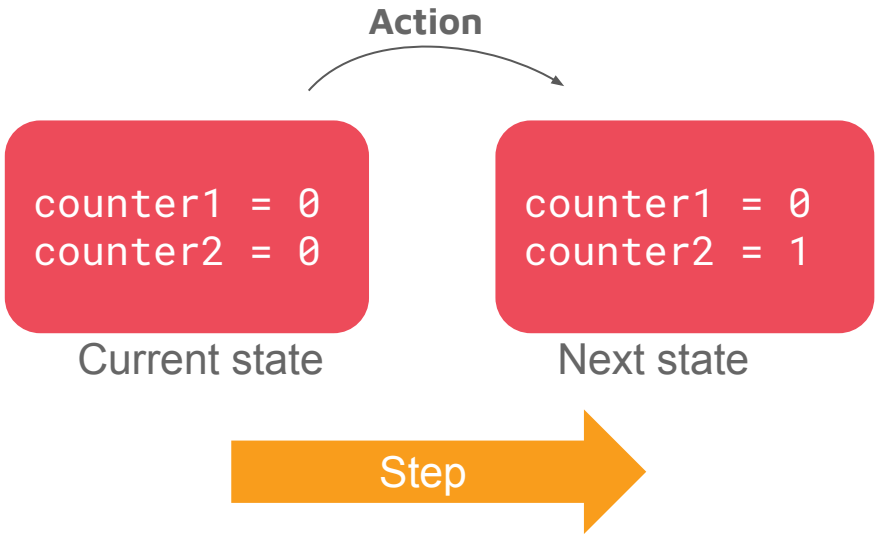


TLA+ Algorithmic Thinking

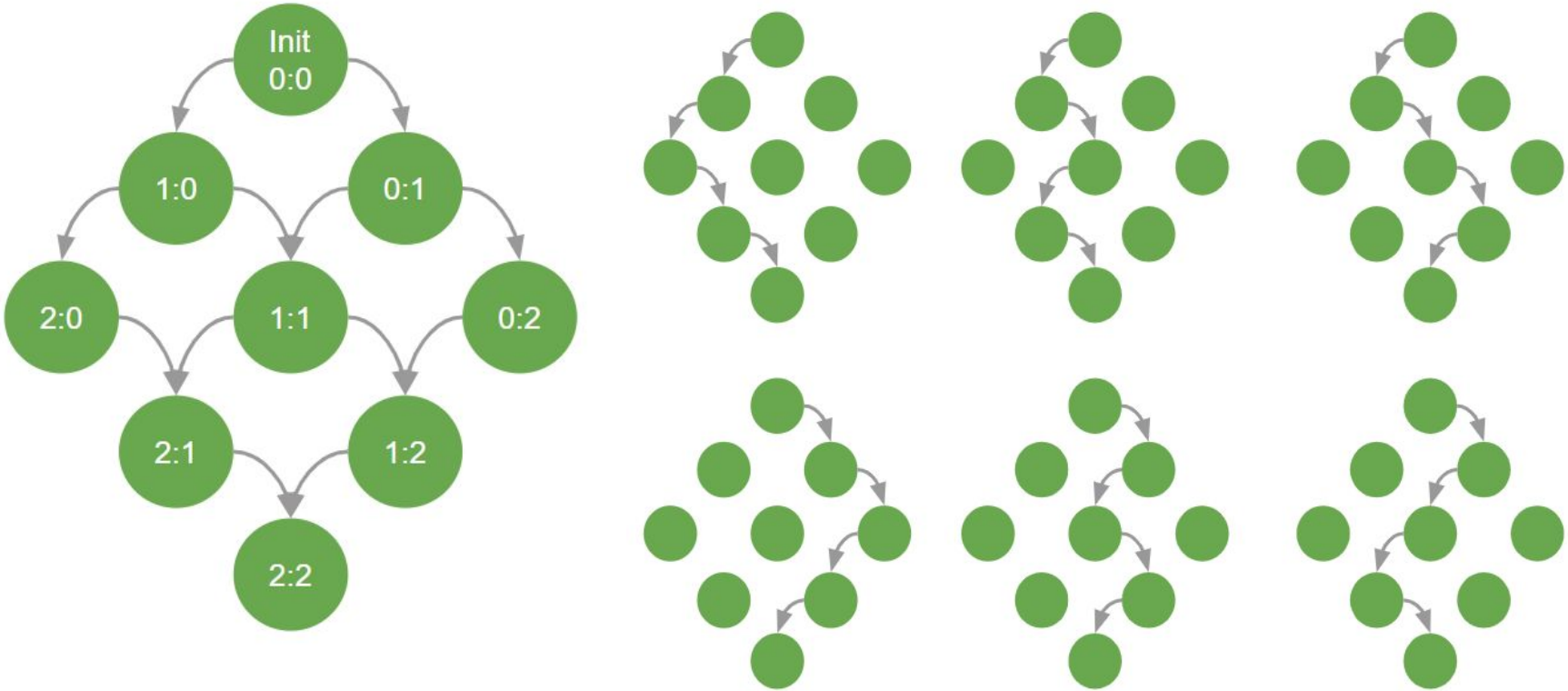
- Describe what not how
  - Free from low-level programming considerations
  - Imagine when drawing a design on a whiteboard having to describe threading models, error handling, network buffers, memory management...

TLA+ States and Actions

- State: A snapshot in (virtual) time of the variables
- Action: Takes us from one state to another (state transition)

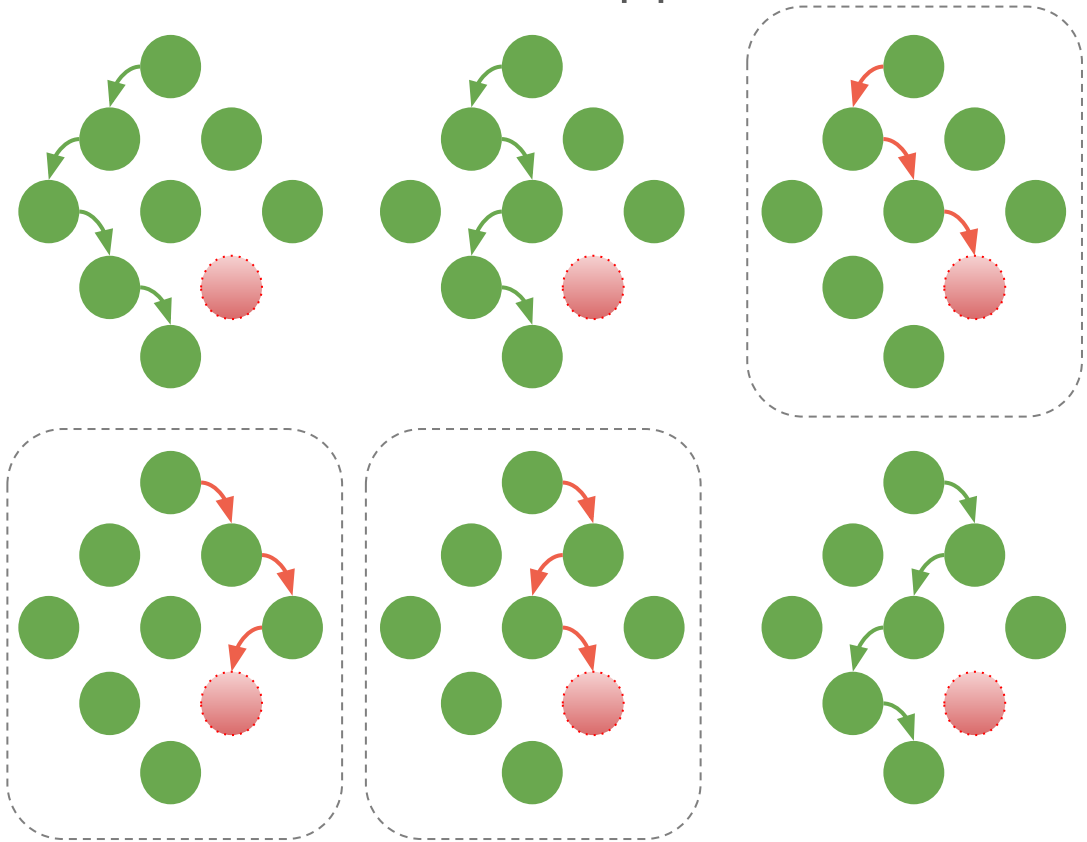


State transitions and behaviours in TLA+



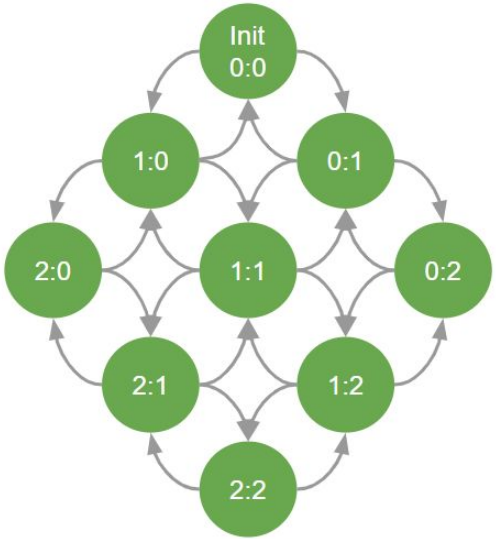
Invariants (something bad that must not happen)

Invariants map to states.

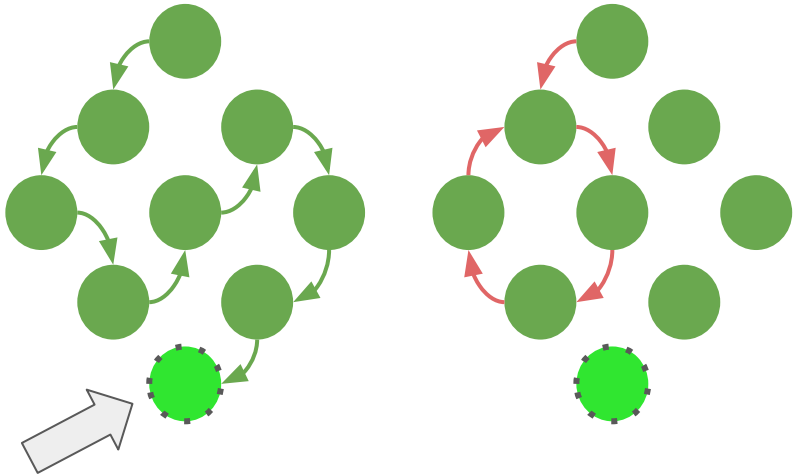


Liveness (something good that should eventually happen)

Liveness maps to behaviours.



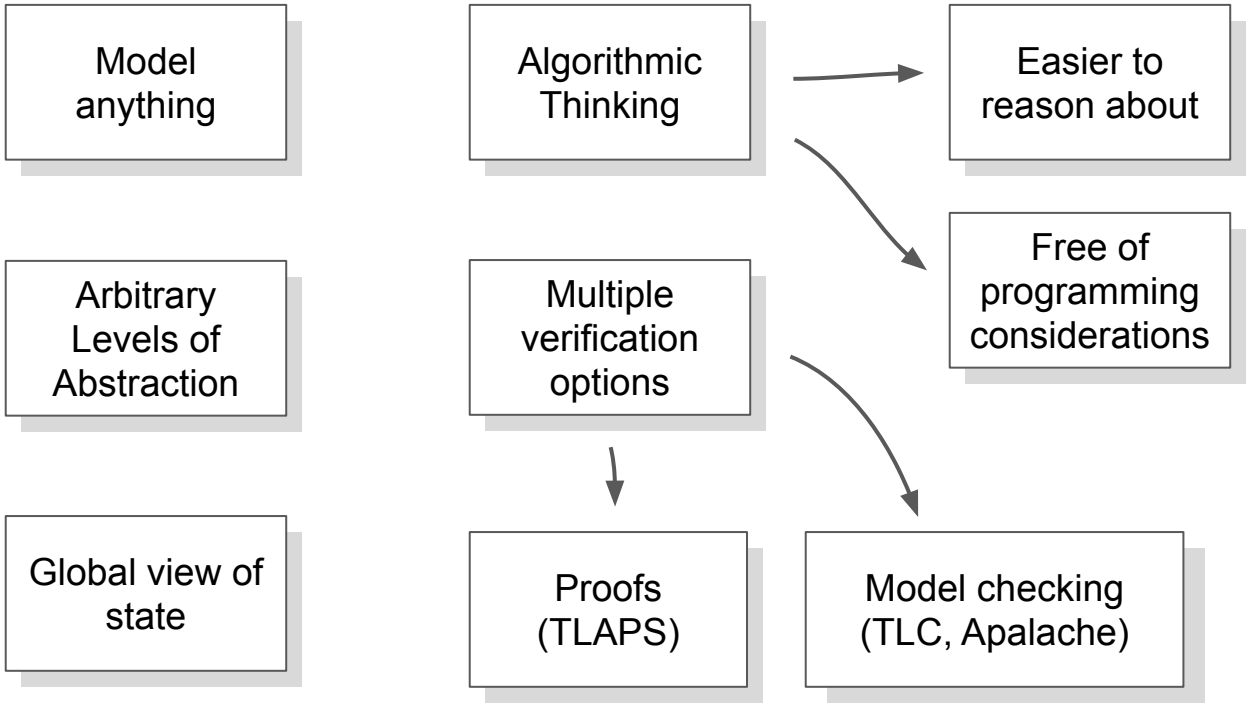
Counters can now decrement as well



Something we want

Liveness property: Eventually both counters reach 2

Summary TLA+

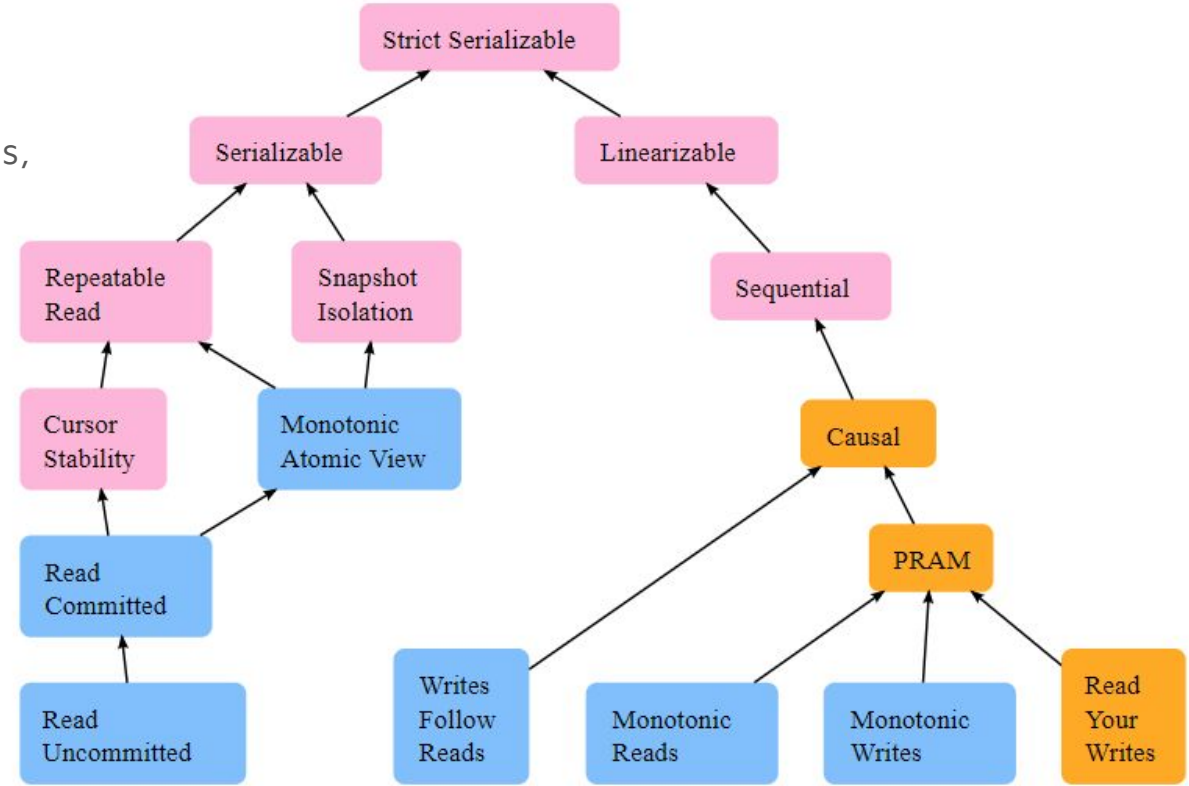

Model anything
Arbitrary Levels of Abstraction
Global view of state
Algorithmic Thinking
Multiple verification options
Proofs (TLAPS)
Model checking (TLC, Apalache)
Easier to reason about
Free of programming considerations

A Quick Look at Jepsen And Maelstrom

https://jepsen.io
https://github.com/jepsen-io/jepsen
https://github.com/jepsen-io/maelstrom

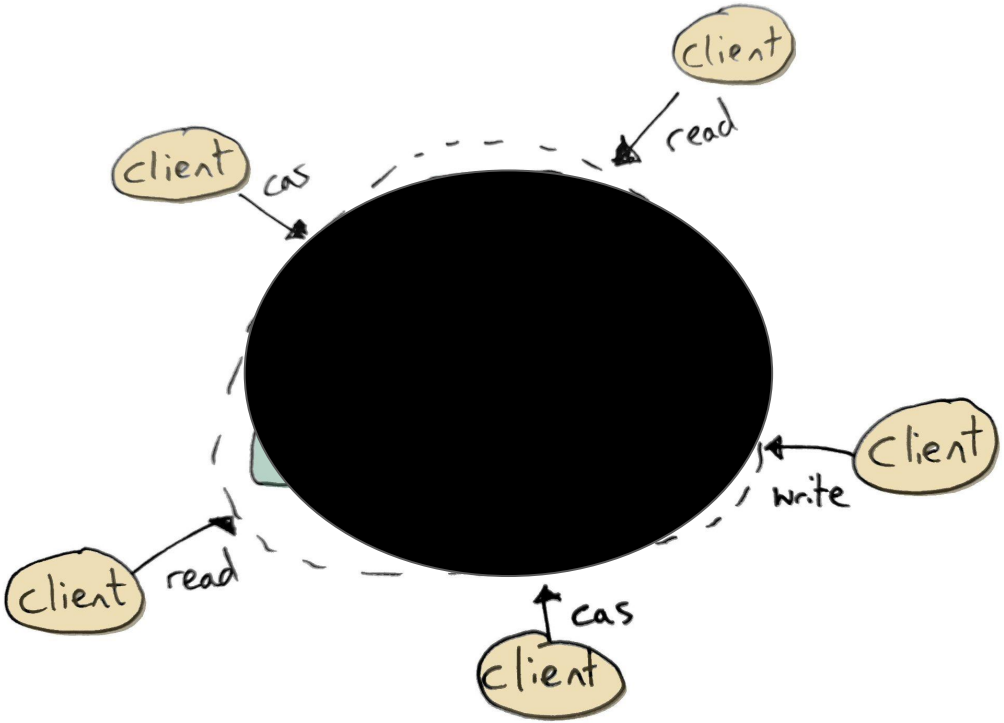
Jepsen Tests

- Test distributed data systems, checking against specific transaction isolation and consistency levels

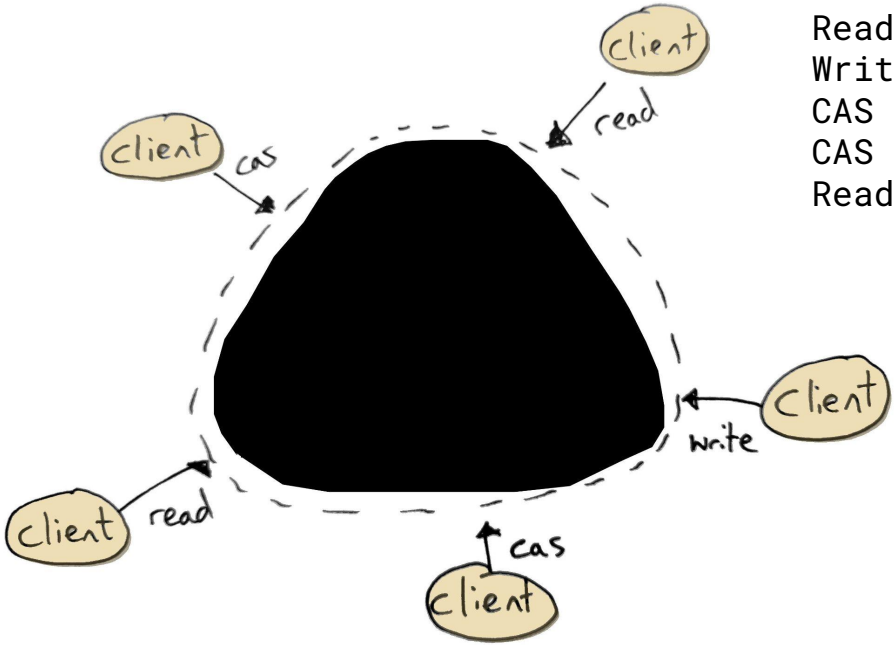


https://jepsen.io/consistency

Jepsen Tests



Jepsen Tests



Checker: Linearizable KV

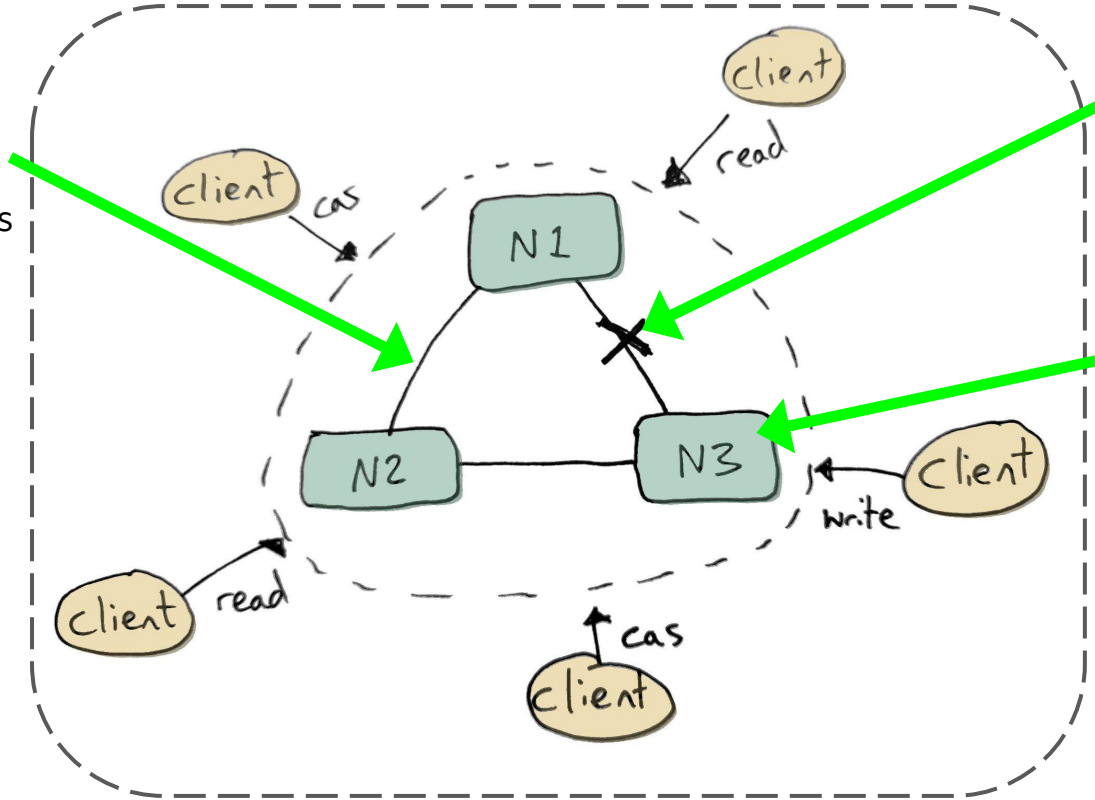
Op	Reg	Val	Result
Write	1	1	OK
Read	1		1
Write	1	3	OK
CAS	1	2, 3	Fail (correct)
CAS	1	3, 4	OK
Read	1		3 (wrong)

Not a linearizable history

Maelstrom

Network in/out
is stdin/stdout

Maelstrom forwards
messages between
nodes.



Maelstrom perturbs
the system (nemesis)

Your code (Python,
Ruby, Java, C++,
Rust, Go etc)

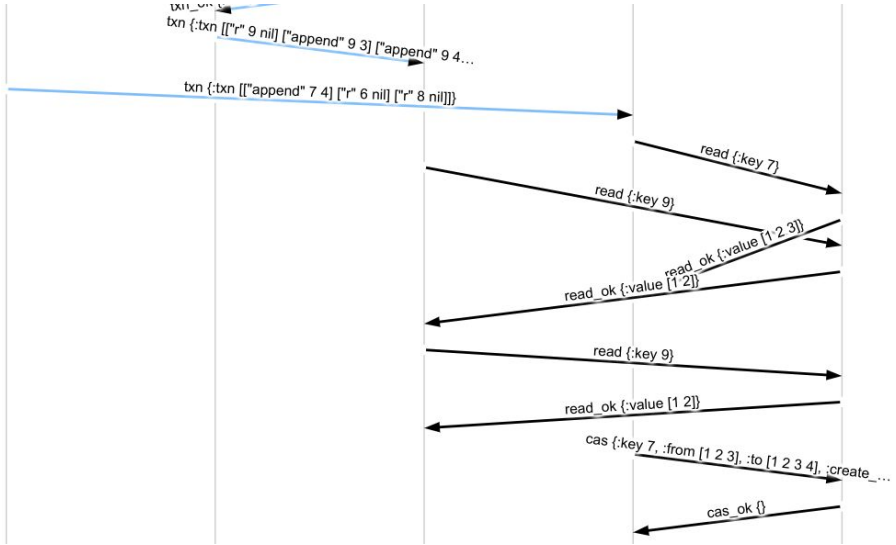
Binary run as a
sub-process per
desired node.

Maelstrom Workloads and Services

- Workloads:
    - **G-counter** (eventually consistent counter)
    - **G-set** (grow-only set)
    - **Lin-kv** (linearizable kv store)
    - **Pn-counter** (eventually consistent counter)
    - **Txn-list-append** (transactional)
- Services:
    - **Lin-kv** (linearizable KV store)
    - **Seq-kv** (sequentially consistent KV store)
    - **Lww-kv** (Last-write-wins KV store)
    - **Lin-tso** (linearizable timestamp oracle)

Maelstrom Results

- Pass/Fail
- Log of message passing
- Visualization of message passing
- Maelstrom logs
- Node logs
- Statistics



https://github.com/jepsen-io/maelstrom/blob/main/doc/05-datomic/02-shared-state.md

Maelstrom Demos

- Ruby, Python, Clojure
- Systems:
    - Raft
    - Datomic
    - CRDTs
- Languages:
    - Raft, Python: 1 file, 593 lines
    - Raft, Ruby: 1 file, 683 lines
    - Datomic list append, Ruby: 1 file, 610 lines

Summary Maelstrom/Jepsen

Oriented towards distributed data systems

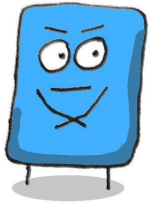
Runs your code (any language)

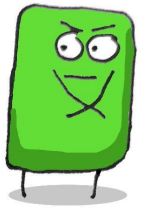
Verification via input vs output

Network stdin/stdout Json messages.

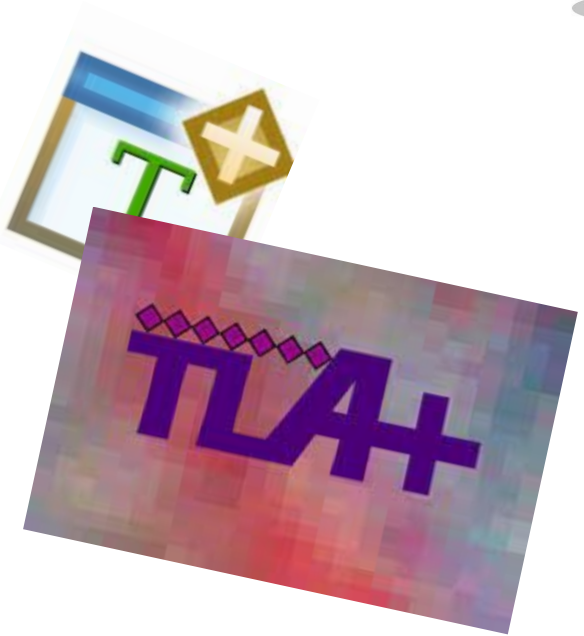
Workloads and checkers

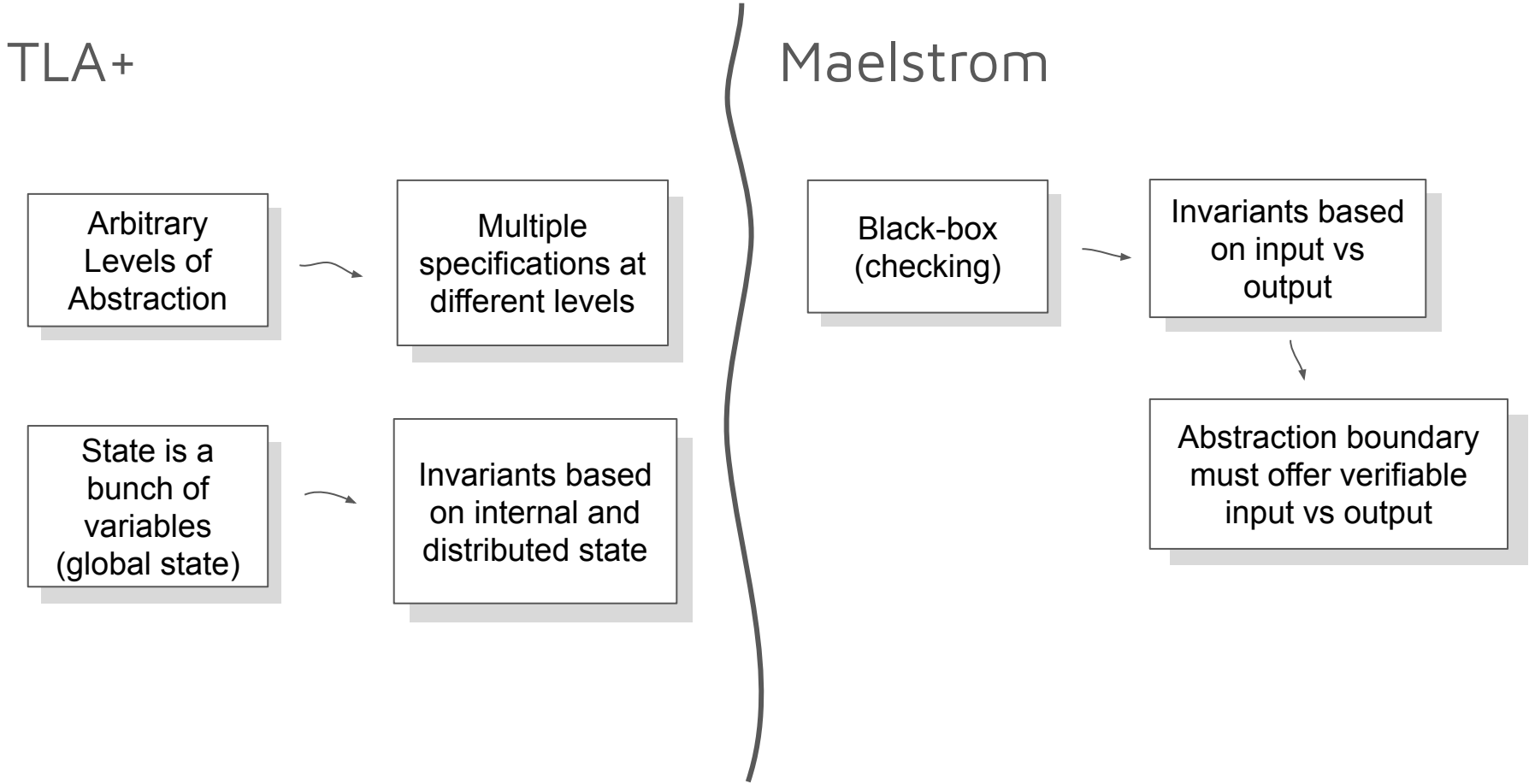
Verification via simulation with perturbations

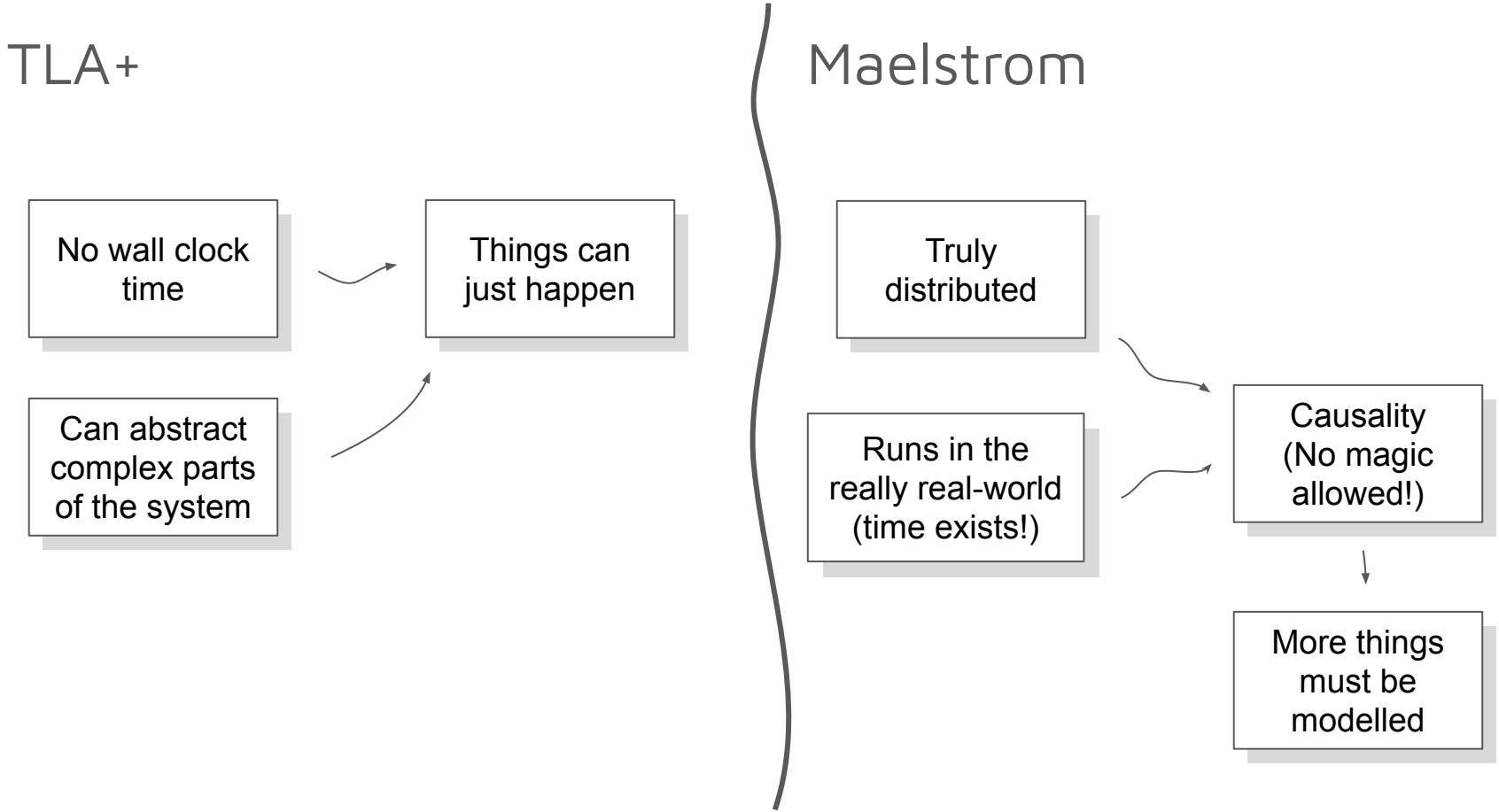




A First Take



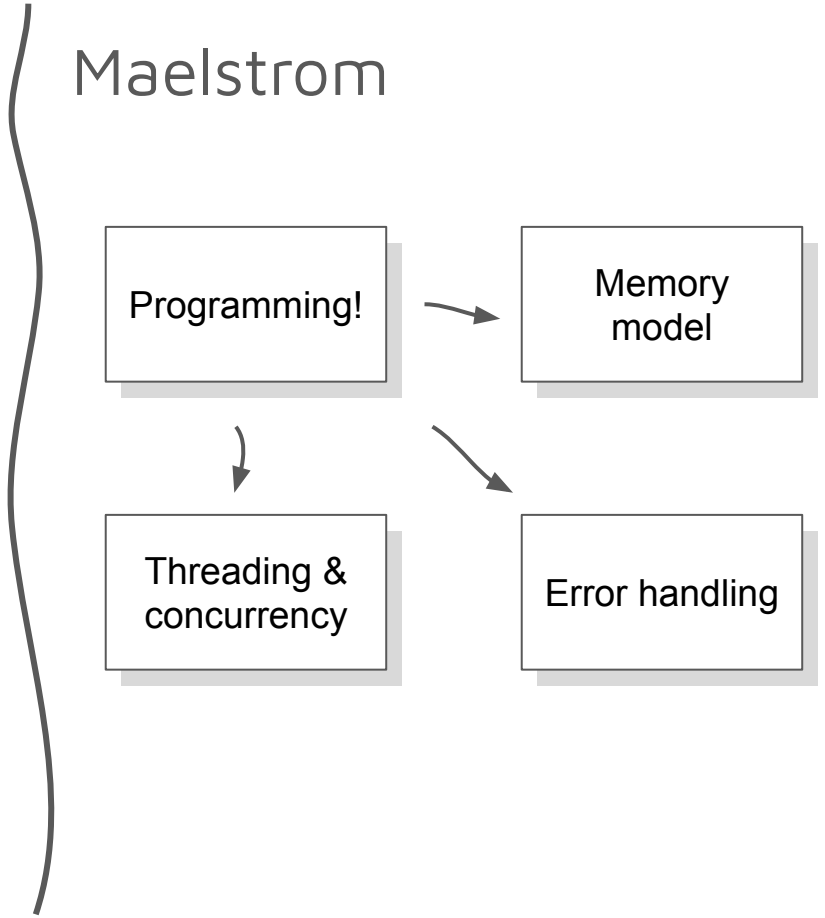

TLA+
Arbitrary Levels of Abstraction
Multiple specifications at different levels
State is a bunch of variables (global state)
Invariants based on internal and distributed state
Maelstrom
Black-box (checking)
Invariants based on input vs output
Abstraction boundary must offer verifiable input vs output

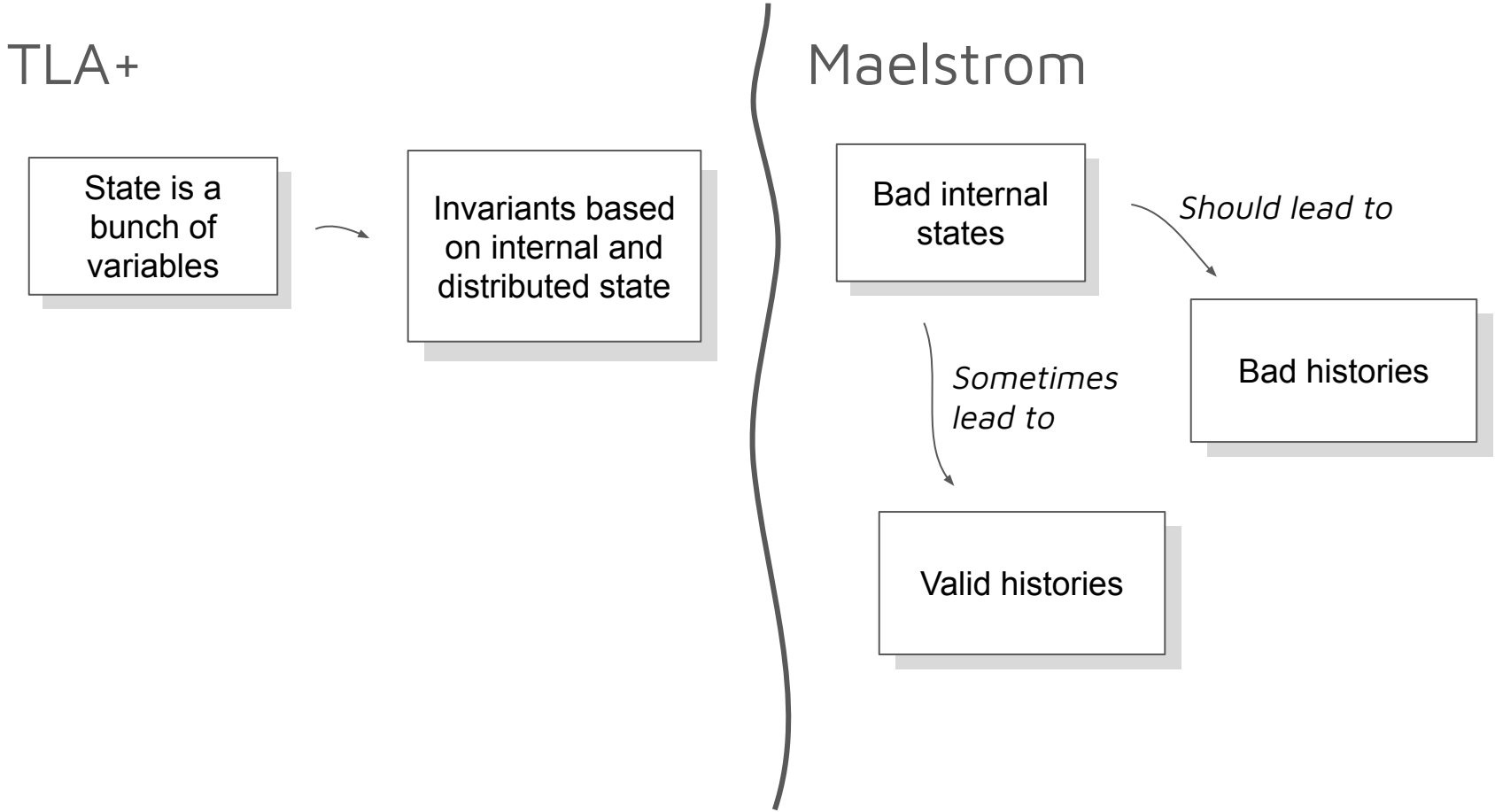

TLA+
No wall clock time
Can abstract complex parts of the system
Things can just happen
Maelstrom
Truly distributed
Runs in the really real-world (time exists!)
Causality (No magic allowed!)
More things must be modelled

TLA+

Algorithmic thinking

Maelstrom



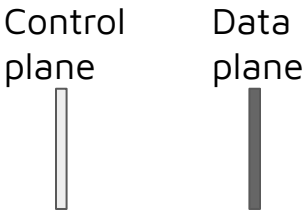

TLA+
State is a bunch of variables
Invariants based on internal and distributed state
Maelstrom
Bad internal states
Should lead to
Bad histories
Sometimes lead to
Valid histories

State enumeration vs simulation

- Categorize actions into:
  - Control plane (leader elections etc)
  - Data plane (steady state of replication)
- TLC explores state space, every possible sequence of actions explored (within constraints of state space size)
- Maelstrom uses simulation and perturbations

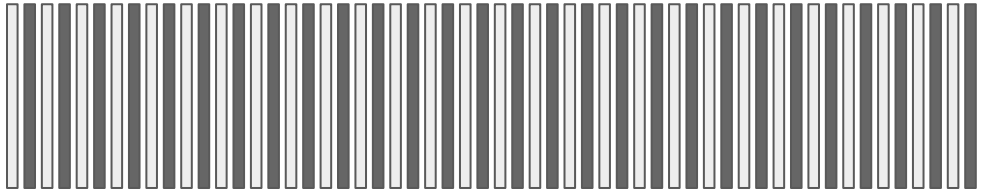
State enumeration vs simulation

Distribution of actions is not equal



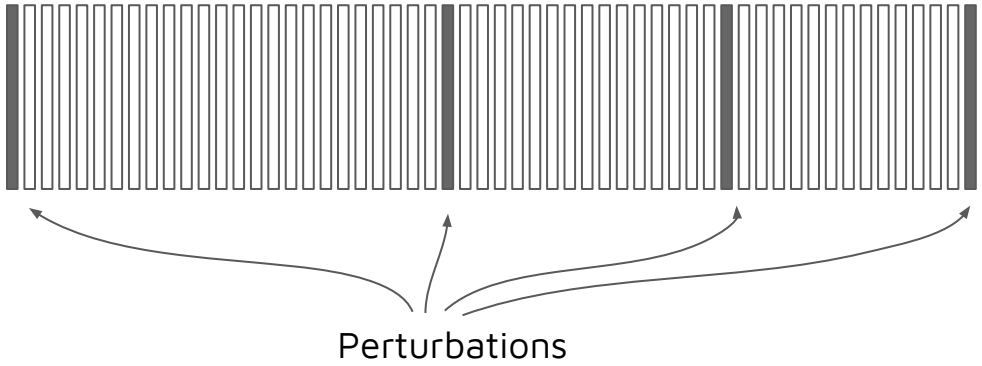
TLA+ with TLC

- State space explored
- Things can happen at any time
- One sequence equal to any other



Maelstrom

- Simulation
- Things happen for a reason
- Slow to explore all possible control-plane sequences

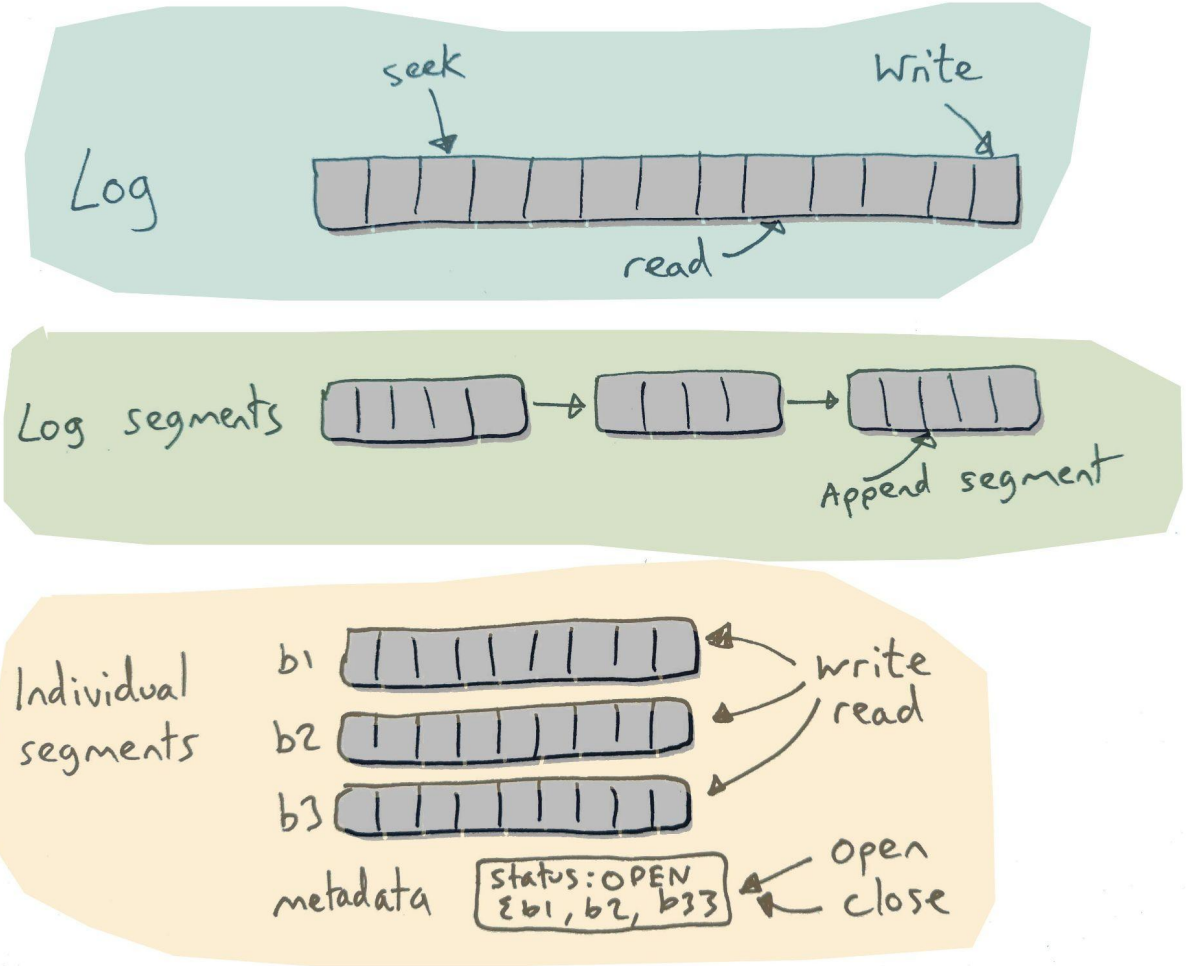


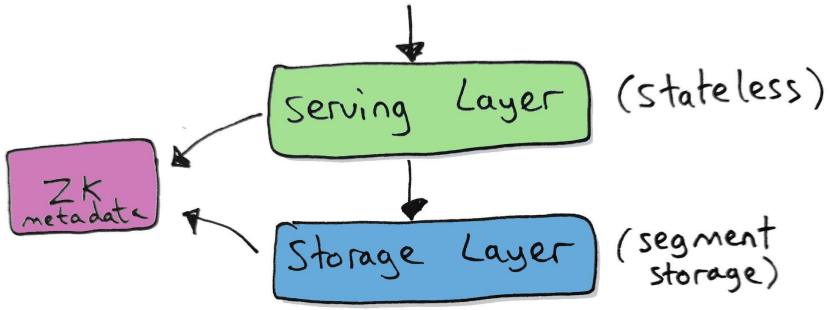
My Experiment

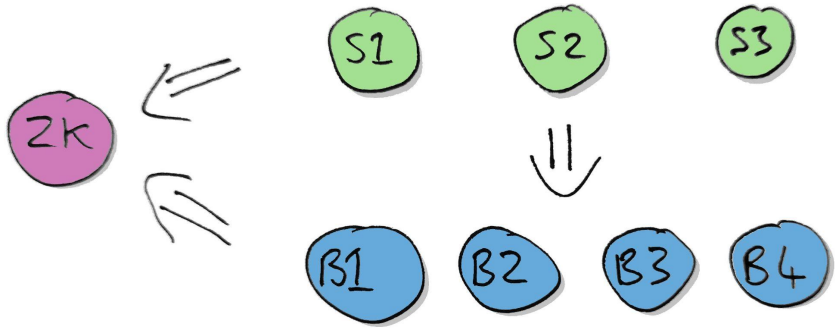
The Distributed Log Storage System

(aka  Apache BookKeeper™)

Abstractions

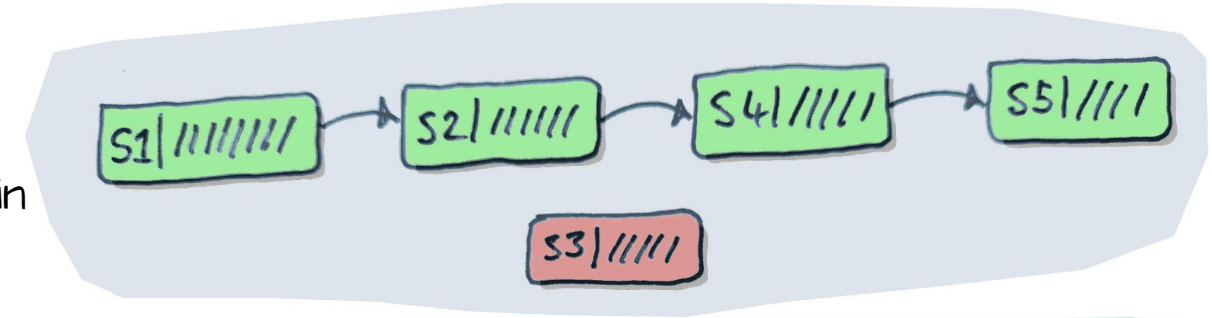



Serving Layer
(stateless)
ZK
metadata
Storage Layer
(segment
storage)

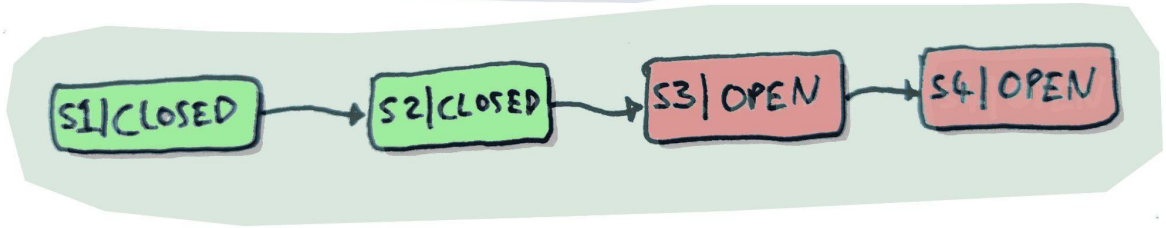

S1
S2
S3
ZK
B1
B2
B3
B4

Segment Chaining Invariants

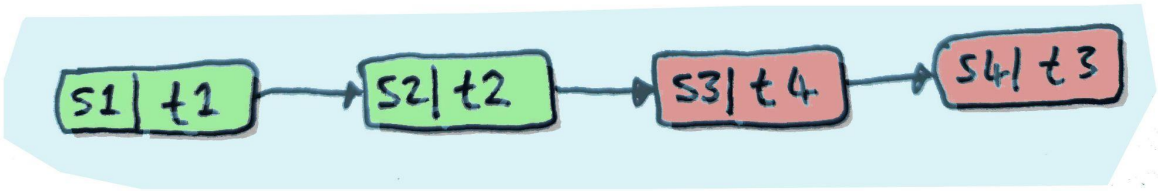
Segment with data outside of segment chain



More than one open segment in the chain

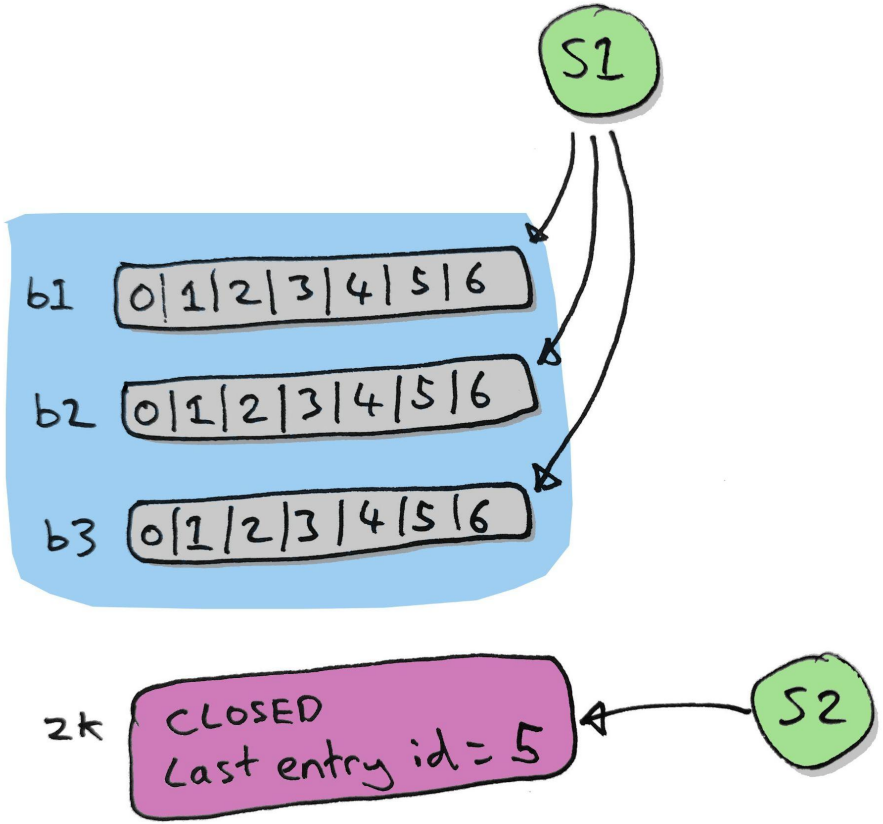


Temporal ordering



Segment Invariants

Metadata and bookies cannot diverge (segment truncation)



A First Look at the Two Models

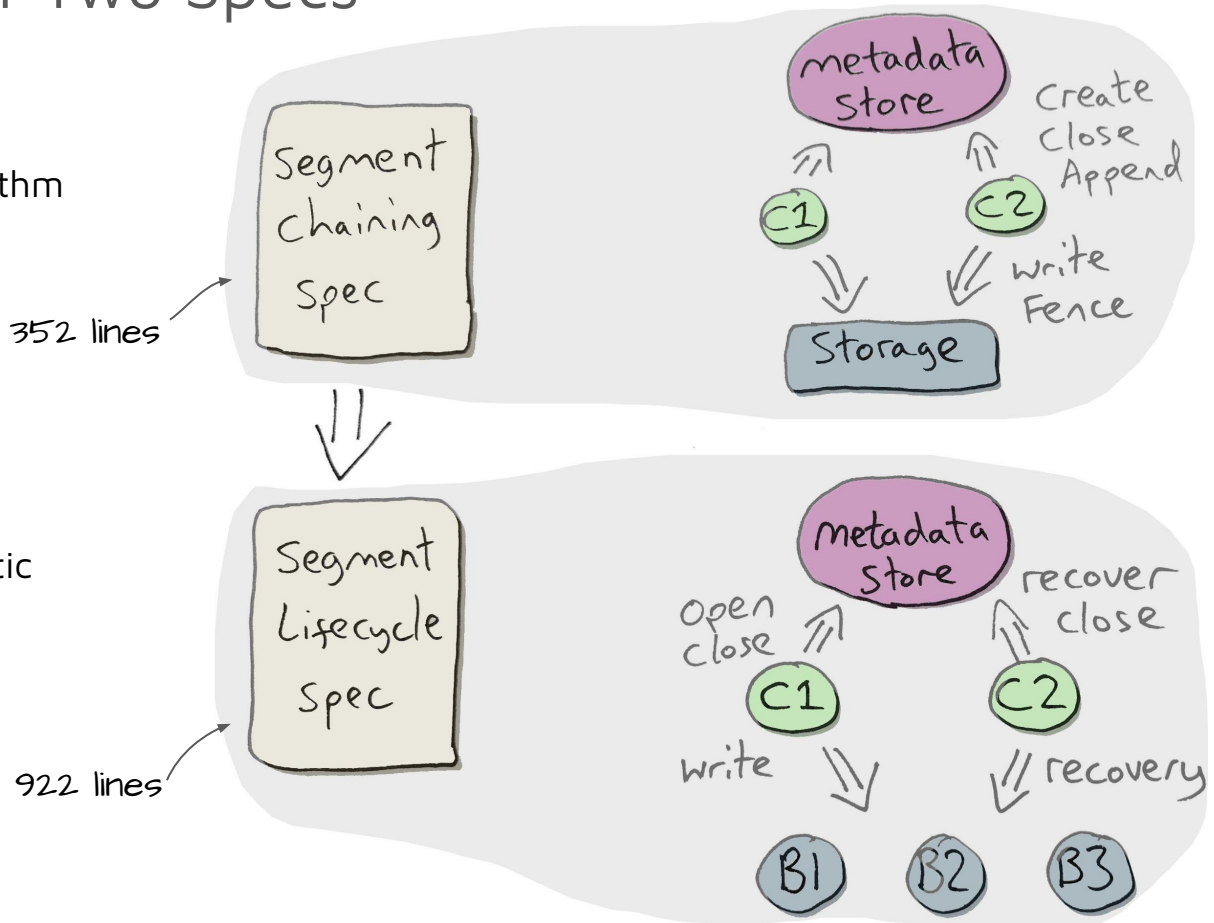
TLA+... A Tale of Two Specs

Things I didn't model:

- Discovery
- Leader election algorithm
- Failure detection
- Reads

Things I did model:

- Leader changes
- Message passing
  - Non-deterministic ordering
  - Message loss



TLA+ Segment Chaining

LedgerChaining.tla > {} LedgerChaining > Next

```
261
262
263 Next ==
264     \E c \in Clients :
265         \/ LeaderChosen(c)
266         \/ BecomeLeader(c)
267         \/ Abdicate(c)
268         \/ GetLastLedger(c)
269         \/ LedgerAlreadyClosed(c)
270         \/ CloseLastLedgerSuccess(c)
271         \/ CloseLastLedgerBadVersion(c)
272         \/ CreateLedger(c)
273         \/ AppendLedgerSuccess(c)
274         \/ AppendLedgerBadVersion(c)
275         \/ WriteToLedger(c)
276         \/ CloseOwnLedgerSuccess(c)
277         \/ CloseOwnLedgerBadVersion(c)
278
279 (*
280     Types
281 *)
282
283 ClientStatuses == {
284             WAITING,
285             GET_MD_FOR_CLOSING,
286             CLOSE_LAST_LEDGER,
287             PENDING_CREATE_LEDGER,
288             PENDING_APPEND_LEDGER,
```

Status

Checking LedgerChaining.tla / LedgerChaining.cfg

Success : Fingerprint collision probability: 3.7E-8

Start: 02:51:35 (May 22), end: 02:51:59 (May 22)

States

Time	Diameter	Found	Distinct	Queue
00:00:00	0	1	1	1
00:00:03	18	182 949	49 270	16 199
00:00:23	40	2 207 914	375 963	0

Coverage

Module	Action	Total	Distinct
LedgerChaining	Init	1	1
LedgerChaining	LeaderChosen	751 926	98 150
LedgerChaining	BecomeLeader	105 957	60 233
LedgerChaining	Abdicate	540 012	84 673
LedgerChaining	GetLastLedger	317 871	45 893
LedgerChaining	LedgerAlreadyClosed	91 287	43 376
LedgerChaining	CloseLastLedgerSuccess	23 184	12 303
LedgerChaining	CloseLastLedgerBadVersion	65 952	8 236
LedgerChaining	CreateLedger	233 514	4 610
LedgerChaining	AppendLedgerSuccess	6 174	3 496
LedgerChaining	AppendLedgerBadVersion	9 612	1 186
LedgerChaining	WriteToLedger	10 116	5 083
LedgerChaining	CloseOwnLedgerSuccess	10 116	5 117
LedgerChaining	CloseOwnLedgerBadVersion	42 192	3 606

TLA+ Segment Lifecycle

```
Next ==
    \* Bookies
    \/ BookieSendsAddConfirmedResponse
    \/ BookieSendsAddFencedResponse
    \/ BookieSendsFencingReadLacResponse
    \/ BookieSendsReadResponse
    \* W1
    \/ W1CreatesLedger
    \/ W1SendsAddEntryRequests
    \/ W1ReceivesAddConfirmedResponse
    \/ W1ReceivesAddFencedResponse
    \/ W1ChangesEnsemble
    \/ W1TriesInvalidEnsembleChange
    \/ W1SendsPendingAddOp
    \/ W1CloseLedgerSuccess
    \/ W1CloseLedgerFail
    \* W2
    \/ W2PlaceInRecovery
    \/ W2ReceivesFencingReadLacResponse
    \/ W2SendsReadRequests
    \/ W2ReceivesNonFinalRead
    \/ W2CompletesReadSuccessfully
    \/ W2CompletesReadWithNoSuchEntry
    \/ W2WritesBackEntry
    \/ W2ReceivesAddConfirmedResponse
    \/ W2ChangesEnsemble
    \/ W2TriesInvalidEnsembleChange
    \/ W2SendsPendingAddOp
    \/ W2ClosesLedger
```

TLA+ Segment Lifecycle - State Space

Model params
Rep factor 3
4 nodes
1 entry

Hardware
12 workers
64 GB RAM
124 GB Storage

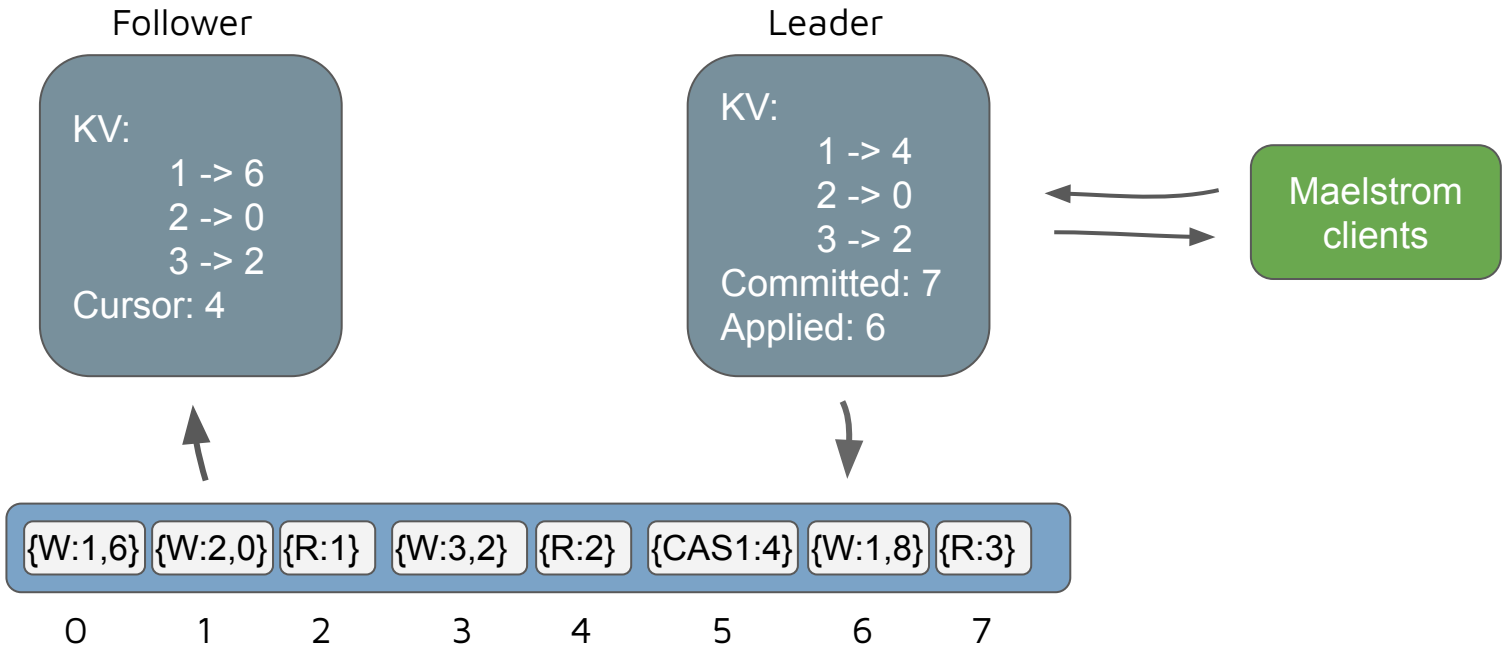
Performance
250000 states/s
67M states
14M unique states
4 ½ hour running

Statistics

State space progress (click column header for graph)

Time	Diameter	States Found	Distinct States	Queue Size
04:34:50	44	67,489,121	13,902,117	0
04:34:11	42	67,352,439	13,884,287	49,388
04:33:11	42	67,112,666	13,847,390	103,737
04:32:11	42	66,861,608	13,810,072	152,153
04:31:11	42	66,623,232	13,770,314	192,803
04:30:11	41	66,401,660	13,731,173	230,816
04:29:11	41	66,143,750	13,694,102	275,458
04:28:11	41	65,900,260	13,657,520	304,088
04:27:11	41	65,664,396	13,616,891	335,555
04:26:11	41	65,430,993	13,578,257	361,370
04:25:11	41	65,210,851	13,540,252	382,242
04:24:11	40	64,981,871	13,499,089	423,459
04:23:11	40	64,721,050	13,462,658	474,543
04:22:11	40	64,472,754	13,423,369	494,788
04:21:11	40	64,228,815	13,383,489	511,752
04:20:11	40	63,996,126	13,351,003	533,832
04:19:11	40	63,759,021	13,310,841	558,358
04:18:11	40	63,528,157	13,270,583	576,412

Maelstrom - A Linearizable KV Store



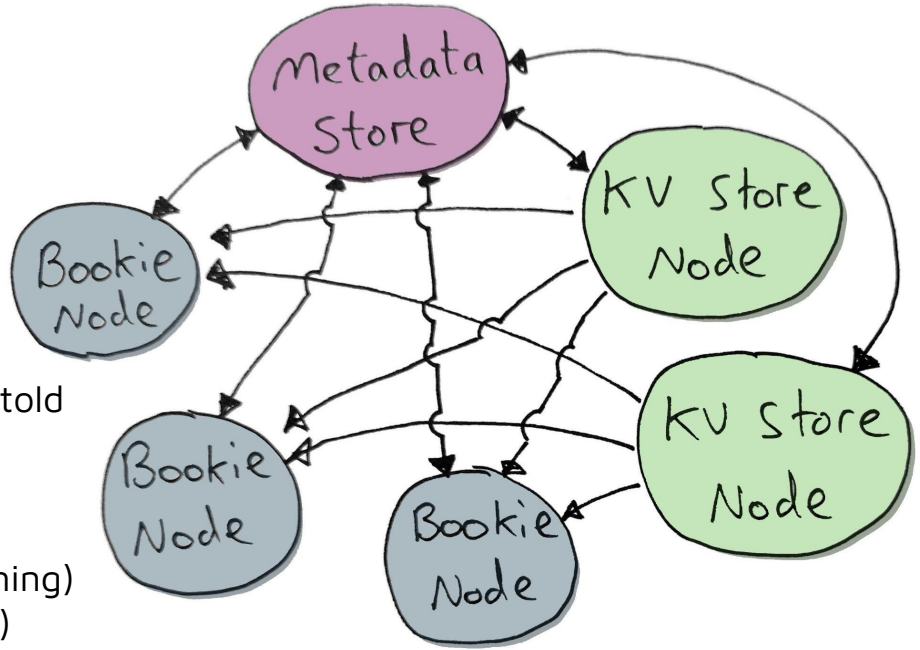
Maelstrom - One Model to Rule Them All!

Things I didn't model:

- ...?

Things I did model:

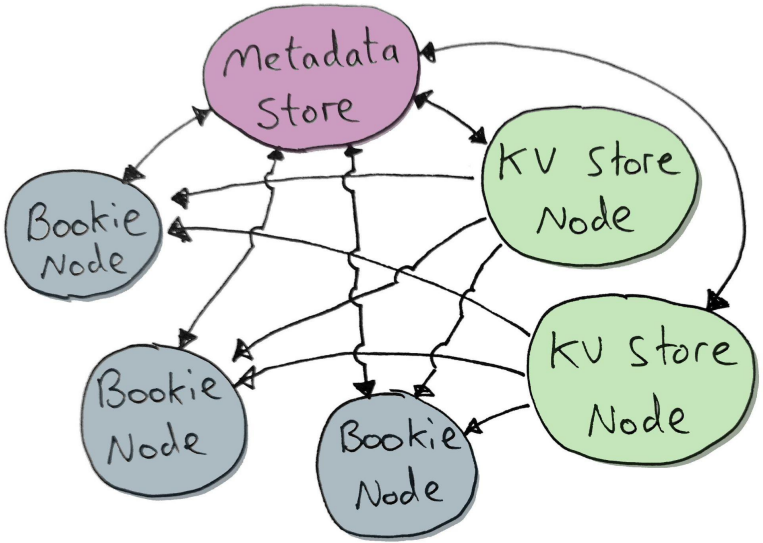
- Metadata store (not distributed)
  - Session management
  - Discovery
  - Leader election
  - Failure detection
- Bookie nodes
  - Store and retrieve what they are told
  - Fencing
- KV Store node
  - KV store
  - Log reader/writer (segment chaining)
  - Ledger handle (segment lifecycle)
  - Projects log into linearizable KV store
    - Replicates reads & writes



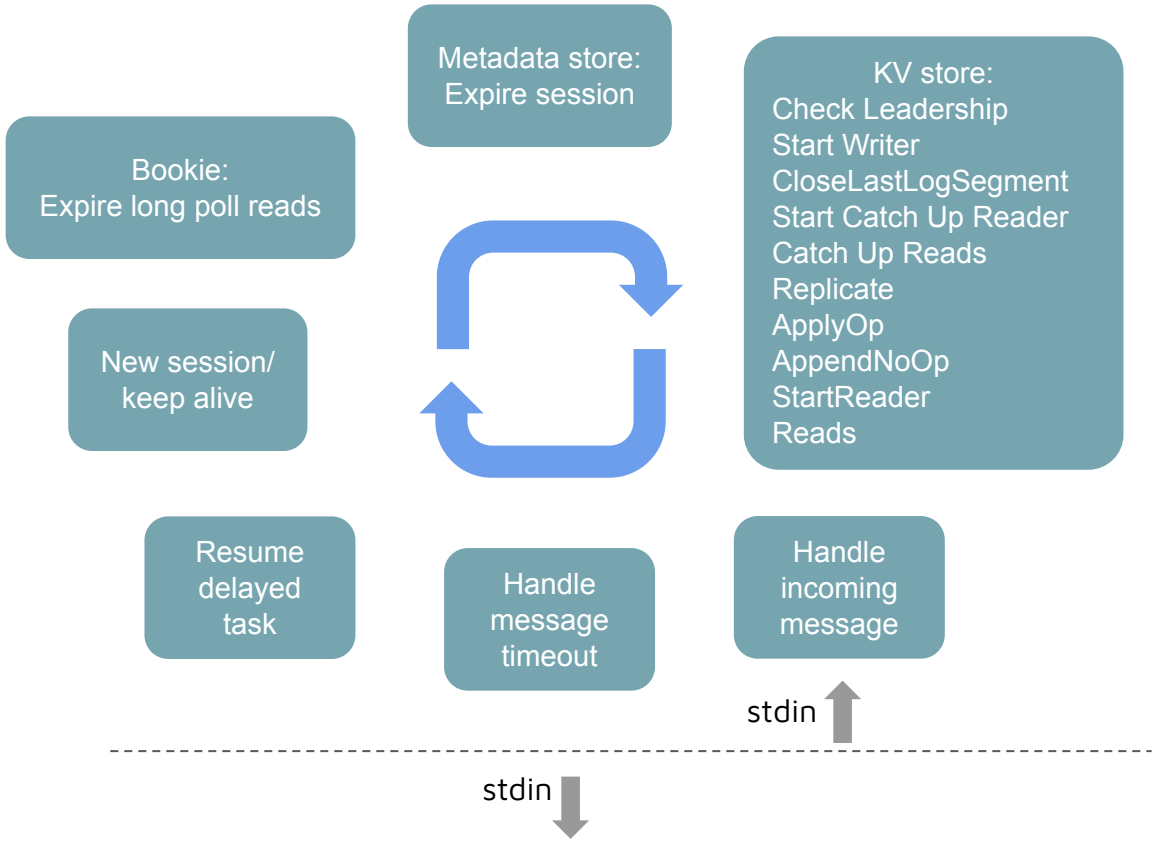
Maelstrom - One Model to Rule Them All!

How big?

- ~6000 lines of Java
- 52 files
- Utility code = 779 lines
  - Futures
  - Logging
  - Delays
  - Etc
  - Timeouts
  - Field names
  - Return codes
- Nodes
  - Shared: 491 lines
    - send, receive, shared data model, proxying
  - Session management: 183 lines
  - Bookie node: 482 lines
  - Metadata store node: 482 lines
  - KV Store Node: 717
    - BK client (segment lifecycle):1543 lines
    - Log reader/writer: (segment chaining): 1148 lines
    - KV store: 445



Maelstrom - Single-Threaded Event Loop



KV Store Node

```
@Override
public void initialize(JsonNode initMsg) { sendInitOk(initMsg); }

@Override
public boolean roleSpecificAction() {
    return sessionManager.maintainSession()
            || checkLeadership()
            || initiateNewWriterSequence()
            || closeLastLogSegment()
            || startCatchUpReader()
            || keepCatchingUp()
            || startWriter()
            || writerSegmentNoLongerOpen()
            || replicate()
            || applyOp()
            || appendNoOp()
            || startReader()
            || keepReading();
}
```

Bookie Node

```
@Override
public boolean roleSpecificAction() {
    return sessionManager.maintainSession()
            || expireLongPollLacReads();
}
```

Metadata Store Node

```
@Override
public boolean roleSpecificAction() {
    return sessionsExpired();
}
```

```
public void handleRequest(JsonNode request) {
    if (mayBeRedirect(request)) {
        return;
    }

    try {
        String type = request.get(Fields.BODY).get(Fields.MSG_TYPE).asText();
        switch (type) {
            case Commands.PRINT_STATE:
                printState();
                break;
            case Commands.Metadata.SESSION_NEW:
                handleNewSession(request);
                break;
            case Commands.Metadata.SESSION_KEEP_ALIVE:
                handleKeepAlive(request);
                break;
            case Commands.Metadata.GET_LEADER_ID:
                handleGetLeaderId(request);
                break;
            case Commands.Metadata.GET_LEDGER_ID:
                handleGetLedgerId(request);
                break;
            case Commands.Metadata.GET_LEDGER_LIST:
                handleGetLedgerList(request);
                break;
```

```
case Commands.Metadata.LEDGER_READ:
    handleReadLedger(request);
    break;
case Commands.Metadata.LEDGER_UPDATE:
    handleUpdateLedger(request);
    break;
case Commands.Metadata.LEDGER_CREATE:
    handleCreateLedger(request);
    break;
case Commands.Metadata.LEDGER_LIST_UPDATE:
    handleLedgerListUpdate(request);
    break;
```

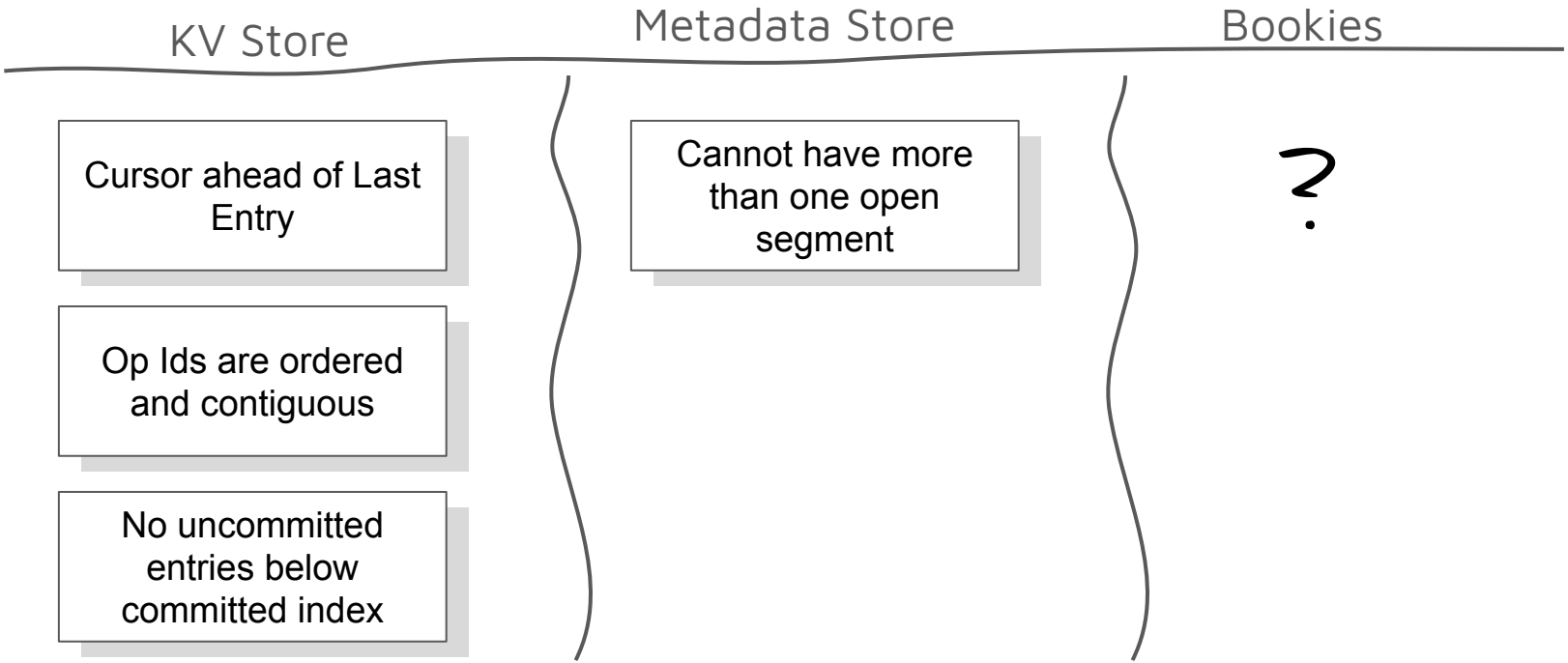
No Blocking Code

```
private CompletableFuture<Void> createWritableLedgerHandle() {
    CompletableFuture<Void> future = new CompletableFuture<>();

    ledgerManager.getAvailableBookies() CompletableFuture<List<String>>
            .thenApply(this::checkForCancellation)
            .thenCompose((List<String> availableBookies) -> createLedgerMetadata(availableBookies))
            .thenApply(this::checkForCancellation)
            .thenCompose((Versioned<LedgerMetadata> vlm) -> appendToLedgerList(vlm))
            .thenApply(this::checkForCancellation)
            .whenComplete((Versioned<LedgerMetadata> vlm, Throwable t) -> {
                if (t == null) {
                    writeHandle = new LedgerWriteHandle(ledgerManager, messageSender, vlm);
                    logger.logDebug( text: "Created new ledger handle for writer");
                    writeHandle.printState();
                    future.complete( value: null);
                } else if (isError(t)) {
                    future.completeExceptionally(t);
                } else {
                    future.complete( value: null);
                }
            });
    return future;
}
```

Local Invariants - Looking inside the box again

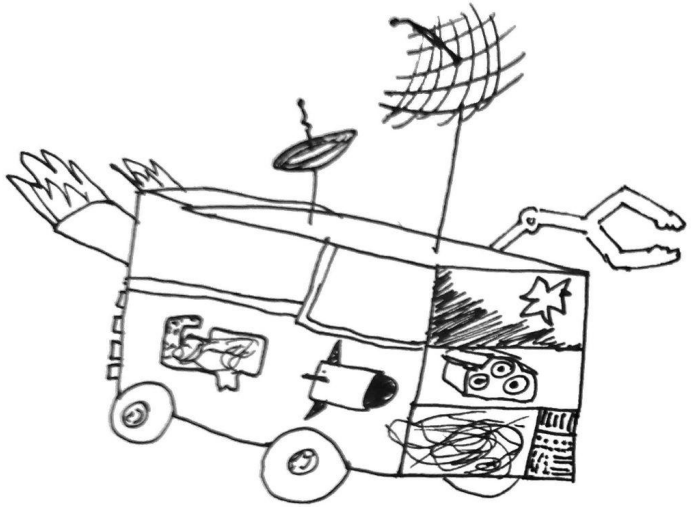
Local invariants -> Crash the node



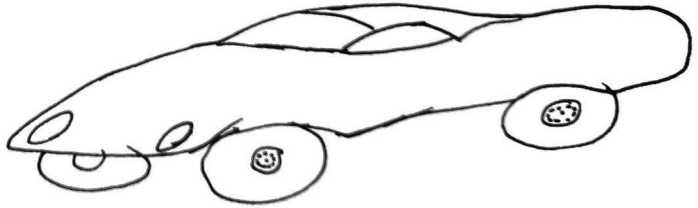
My experiences, mine!

- Maelstrom
  - Spent more time than I'd like on getting all the asynchronous code to work correctly:
    - Each node single-threaded to avoid complexity of multi-threading within a single node
    - Uses an event loop to trigger actions, respond to replies, timeout requests, implement non-blocking delays
    - Chaining non-blocking calls
    - Handling, propagating errors correctly
    - Building in timeouts, delays into the event loop

Choice of Language



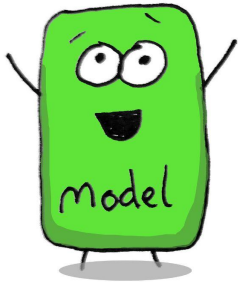
Java, C++



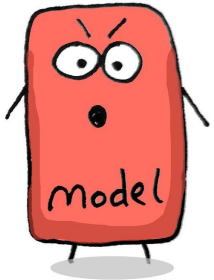
Ruby, Python

Statically typed, fast, good for distributed data system

Elegant, Good for modelling



Model Checking Wins and Fails



TLA+ Finds Real Protocol Defect! Model Checking Win

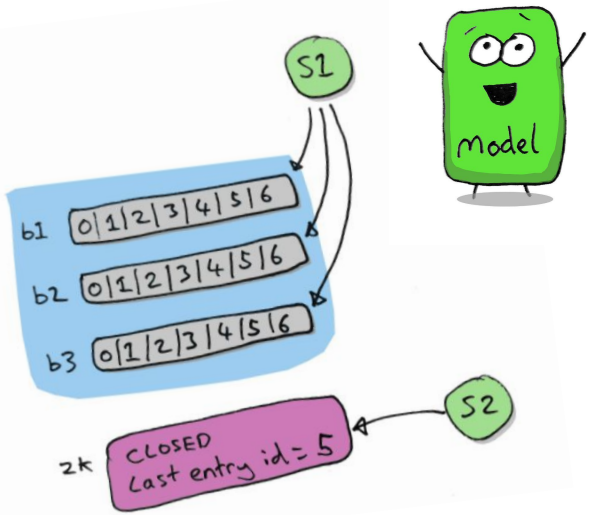
TLC Model Checker

TLA+ Segment Lifecycle Invariants

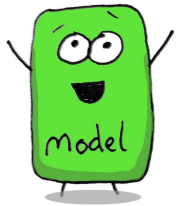
```
(***************************************************************************)
(* Invariant: No Divergence Between Writer And MetaData                    *)
(*                                                                         *)
(* This invariant is violated if, once a ledger is closed, the writer has  *)
(* an entry acknowledged (by Qa bookies) that has a higher entry id than   *)
(* the endpoint of the ledger as stored in the metadata store.             *)
(* This is divergence and data loss.                                       *)
(***************************************************************************)
NoDivergenceBetweenWriterAndMetaData ==
    IF meta_status # STATUS_CLOSED
    THEN TRUE
    ELSE \A id \in 1..wl.lac :
            id <= meta_last_entry
```

Segment Invariants

Metadata and bookies cannot diverge (segment truncation)



TLA+ Real-life Defect - Model Checking Win



28 CPU threads, 100GB RAM, NVMe SSD

```
Invariant NoDivergenceBetweenWriterAndMetaData is violated.
```

Rep factor 3, 3 Bookies, 1 entry

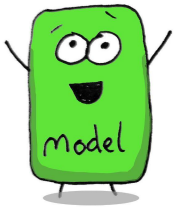
State space progress (click column header for graph)

Time	Diameter	States Found	Distinct States	Queue Size
00:00:22	22	1,577,889	376,929	42,957
00:00:14	16	156,524	52,618	24,514
00:00:11	0	1	1	1

TLA+ Model Checking Win

Error trace makes it clear:

- What invariant got violated
- The specific sequence of state steps that leads to the violation with the states involved



Invariant NoDivergenceBetweenWriterAndMetaData is violated.

Error-Trace Exploration

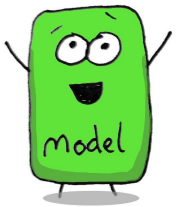
Error-Trace

Name	Value
<Initial predicate>	State (num = 1)
b_entries	(B1 :> { } @@ B2 :> { })
b_fenced	(B1 :> FALSE @@ B2 :> FALSE)
b_lac	(B1 :> 0 @@ B2 :> 0)
messages	<< >>
meta_fragments	<< >>
meta_last_entry	0
meta_status	Nil
meta_version	0
w1	[meta_version |-> Nil, status |-> Nil, curr_fragment |-> Nil, pending_add_ops |-> { }, lap |->..
w2	[meta_version |-> Nil, status |-> Nil, curr_fragment |-> Nil, pending_add_ops |-> { }, lap |->..
<W1CreatesLedger line 161, col 5 t	State (num = 2)
b_entries	(B1 :> { } @@ B2 :> { })
b_fenced	(B1 :> FALSE @@ B2 :> FALSE)
b_lac	(B1 :> 0 @@ B2 :> 0)
messages	<< >>
meta_fragments	<<[id |-> 1, ensemble |-> {B1, B2}, first_entry_id |-> 1]>>
meta_last_entry	0
meta_status	STATUS_OPEN
meta_version	1
w1	[meta_version |-> 1, status |-> STATUS_OPEN, curr_fragment |-> [id |-> 1, ensemble |-> ...
w2	[meta_version |-> Nil, status |-> Nil, curr_fragment |-> Nil, pending_add_ops |-> { }, lap |->..
<W1SendsAddEntryRequests line 1	State (num = 3)
b_entries	(B1 :> { } @@ B2 :> { })

Maelstrom - Checking Win

I made mistake after mistake after mistake during the implementation...

Maelstrom usually found them in under 5 minutes, sometimes an hour.



Showed lower level mistakes that the higher level TLA+ specification could not flag.

Great insight into the kinds of mistakes that could get implemented in the real implementation.

Mistake!

Mistake!

Mistake!

Mistake!

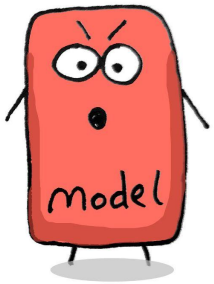
Mistake!

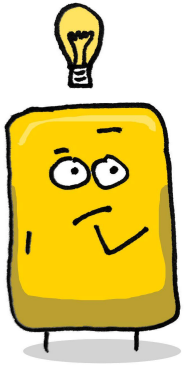
Mistake!

Maelstrom - Checking Fail? Or Jack Fail...

Couldn't start checking until whole system modelled.

A long time passed until I could start getting confidence via the checking (then huge volume of mistakes to fix).





Could have started with a non distributed KV Store, then slowly add components as I went, checking along the way.

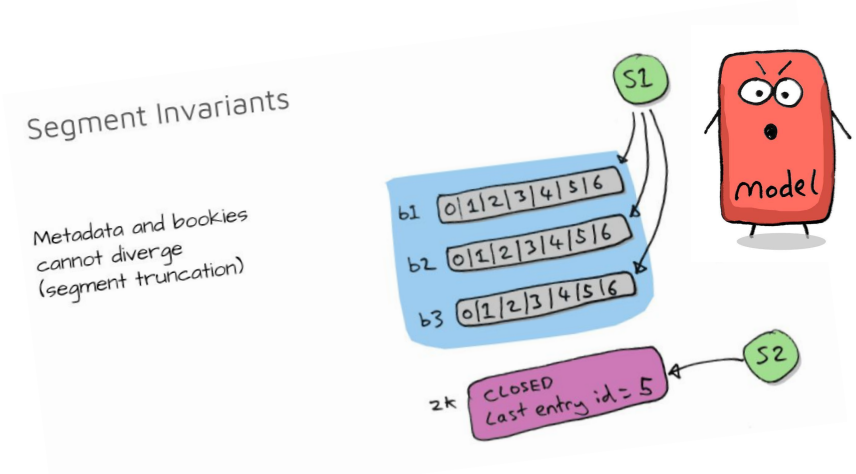
Maelstrom - Checking Fail

Ran it for 10 days... did not find this defect.

Hard-coded losing a key message on EVERY SINGLE leader failover... nope.

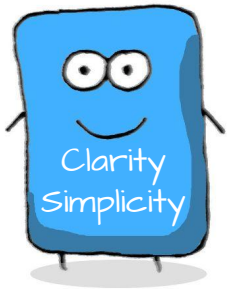
Hard-coded dropping session keep-alives after 3 seconds in order to trigger leader failover every 3 seconds ... nope.

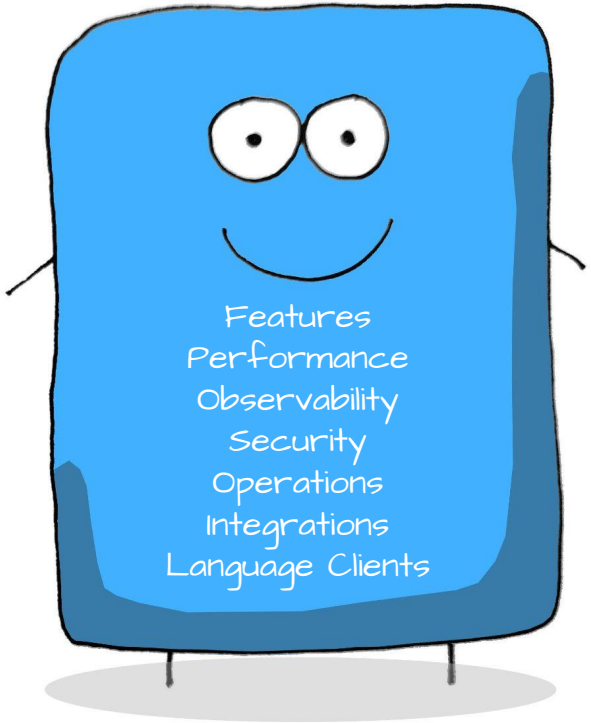
Hard-coded 100 ms delay between message sends to each bookie to increase probability of overlap... Yes! After only 1 hour! (local invariant not jepsen)



Turned off local invariant checking to see if Jepsen would detect it... Yes! After 5 days.

Re-enabled local invariants, removed hard-coded delay and used Maelstrom random network latency of 10 ms...
... ran for 5 days and finally yes.

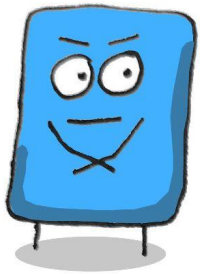


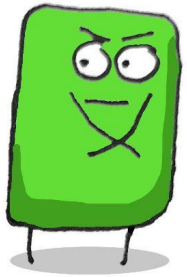


From model to implementation

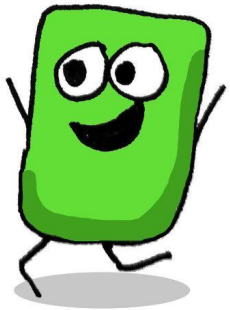
You've already taken the first steps when using Maelstrom

- Shows you what you need to log
  - Good logging is a necessity not an afterthought!
- The power of the network shim
  - Not sure I'd give up the convenience of Maelstrom, even with my implementation.
- The model likely has shown you insights into real mistakes that could be made in the implementation.
- But ...
  - Simplicity vs performance - can you reuse the model code at all?
  - Does the model and implementation even have the same language?





Final Thoughts



The Good Parts

TLA+

Like sketching
Free flowing

One file,
can keep it in
my head

Model checker
found defects
fast

Error traces
relatively easy
to parse

Maelstrom

It's just coding
in my chosen
language!

Easy to
inspect the
network

Simple to run
(no servers,
no k8s etc)

Checking found
most defects within
5 minutes to an
hour

The Challenges

TLA+

New to TLA+? Expect a steep learning curve

Battling the state space

Success! Or is it?

Maelstrom

Analysis is time consuming!

Custom workloads require Clojure (learning curve)

Boilerplate!

Can be larger, more complex

Less free flowing

Hard to keep in my head

Use them for their strengths

TLA+

TLA+ is abstract, and that is its strength

Focus on what not how

Free of clutter

Maelstrom

Maelstrom is truly distributed in true time

Blurs the line of model and prototype

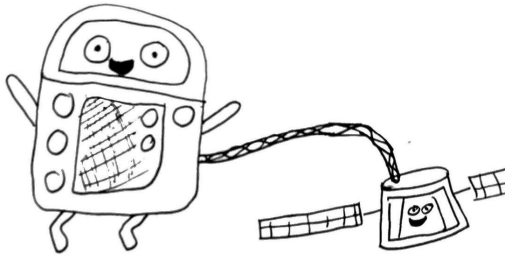
Insights directly related to coding the solution

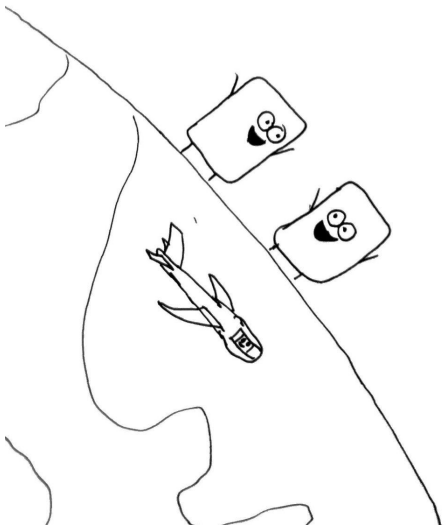
Hopefully makes you do better logging

Possibly useful beyond model/prototype stage

Design doc -> TLA+ -> Maelstrom prototype -> Impl

Modelling is not a Silver Bullet!





Thanks!

https://jepsen.io
https://github.com/jepsen-io/jepsen
https://github.com/jepsen-io/maelstrom
Check out the Maelstrom demos!

BookKeeper TLA+ Specifications
https://github.com/Vanlightly/bookkeeper-tlaplus

My Distributed Log Maelstrom Model
https://github.com/Vanlightly/maelstrom-playground

Art Director and Illustrator: My son!

Attend the TLA+ workshop!!!