Model-based Testing of Practical Distributed Systems in Actor Model

Ilya Kokorin
kokorin.ilya.1998@gmail.com

Evgeny Chernatsky
chernatskiy2001@gmail.com

It began with ~~the forging of Great Rings~~ distributed consensus implementation

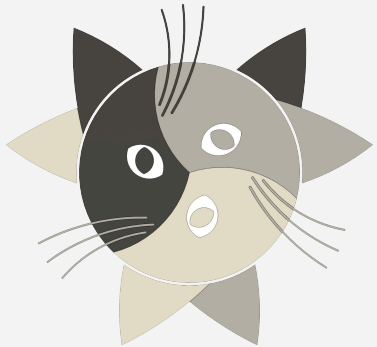
To help VK.com build strongly-consistent replicated databases

www.youtube.com/watch?v=V033-W6Xxk0

- Only available in Russian
- Auto-generated captions may help though

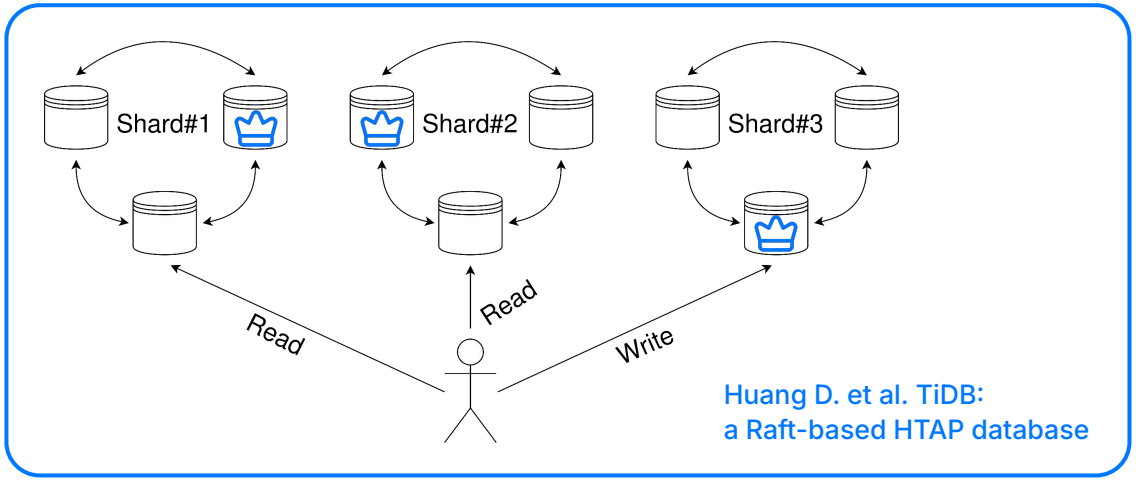
BARSiC

A commonly-used cat name in Russia



BARSiC Architecture

- Database is sharded
- Each shard is replicated and has a single leader
- Leader is re-elected on death
- Write queries are processed by the corresponding shard leader
- Read queries are processed by any node of the shard



Viewstamped Replication

One of the (many) papers describing strongly-consistent replication algorithms

- Paxos
- Raft
- Zookeeper Atomic Broadcast
- Bizur
- etc as another examples

Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems

Brian M. Oki
Barbara H. Liskov

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, MA 02139

Viewstamped Replication Revisited

Barbara Liskov and James Cowling
MIT Computer Science and Artificial Intelligence Laboratory
liskov@csail.mit.edu, cowling@csail.mit.edu

Viewstamped Replication

In VK.com mere implementation of the paper led us to production-ready strongly consistent replication engine

Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems

Brian M. Oki
Barbara H. Liskov

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, MA 02139

Viewstamped Replication Revisited

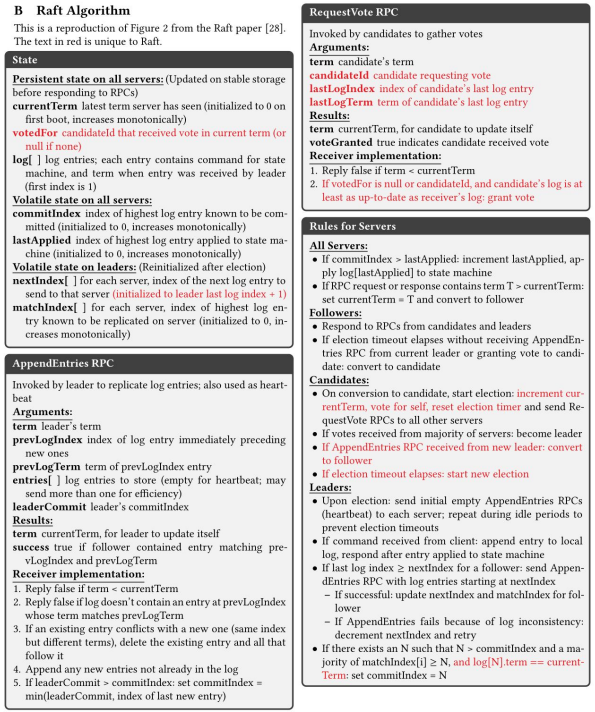
Barbara Liskov and James Cowling
MIT Computer Science and Artificial Intelligence Laboratory
liskov@csail.mit.edu, cowling@csail.mit.edu

Viewstamped Replication

In VK.com mere implementation of the paper led us to production-ready strongly consistent replication engine

A difficult path to the production-ready replicator

Implementing an algorithm from a paper is simple, but...



B Raft Algorithm

This is a reproduction of Figure 2 from the Raft paper [28]. The text in red is unique to Raft.

State

Persistent state on all servers: (Updated on stable storage before responding to RPCs)
currentTerm latest term server has seen (initialized to 0 on first boot, increases monotonically)
votedFor candidateId that received vote in current term (or null if none)
log[ ] log entries; each entry contains command for state machine, and term when entry was received by leader (first index is 1)
Volatile state on all servers:
commitIndex index of highest log entry known to be committed (initialized to 0, increases monotonically)
lastApplied index of highest log entry applied to state machine (initialized to 0, increases monotonically)
Volatile state on leaders: (Reinitialized after election)
nextIndex[ ] for each server, index of the next log entry to send to that server (initialized to leader last log index + 1)
matchIndex[ ] for each server, index of highest log entry known to be replicated on server (initialized to 0, increases monotonically)

AppendEntries RPC

Invoked by leader to replicate log entries; also used as heartbeat
Arguments:
term leader's term
prevLogIndex index of log entry immediately preceding new ones
prevLogTerm term of prevLogIndex entry
entries[ ] log entries to store (empty for heartbeat; may send more than one for efficiency)
leaderCommit leader's commitIndex
Results:
term currentTerm, for leader to update itself
success true if follower contained entry matching prevLogIndex and prevLogTerm
Receiver implementation:
1. Reply false if term < currentTerm
2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm
3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it
4. Append any new entries not already in the log
5. If leaderCommit > commitIndex: set commitIndex = min(leaderCommit, index of last new entry)

RequestVote RPC

Invoked by candidates to gather votes
Arguments:
term candidate's term
candidateId candidate requesting vote
lastLogIndex index of candidate's last log entry
lastLogTerm term of candidate's last log entry
Results:
term currentTerm, for candidate to update itself
voteGranted true indicates candidate received vote
Receiver implementation:
1. Reply false if term < currentTerm
2. If votedFor is null or candidateId, and candidate's log is at least as up-to-date as receiver's log: grant vote

Rules for Servers

All Servers:
- If commitIndex > lastApplied: increment lastApplied, apply log[lastApplied] to state machine
- If RPC request or response contains term T > currentTerm: set currentTerm = T and convert to follower

Followers:
- Respond to RPCs from candidates and leaders
- If election timeout elapses without receiving AppendEntries RPC from current leader or granting vote to candidate: convert to candidate

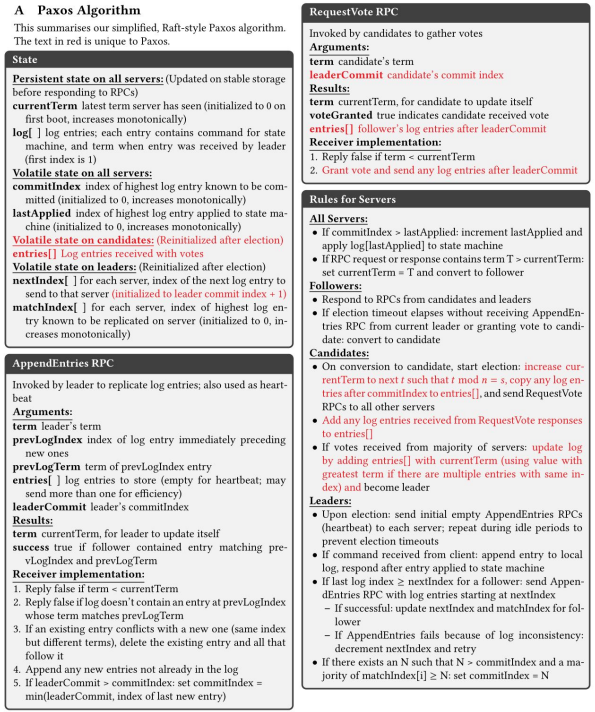
Candidates:
- On conversion to candidate, start election: increment currentTerm, vote for self, reset election timer and send RequestVote RPCs to all other servers
- If votes received from majority of servers: become leader
- If AppendEntries RPC received from new leader: convert to follower
- If election timeout elapses: start new election

Leaders:
- Upon election: send initial empty AppendEntries RPCs (heartbeat) to each server; repeat during idle periods to prevent election timeouts
- If command received from client: append entry to local log, respond after entry applied to state machine
- If last log index ≥ nextIndex for a follower: send AppendEntries RPC with log entries starting at nextIndex
  - If successful: update nextIndex and matchIndex for follower
  - If AppendEntries fails because of log inconsistency: decrement nextIndex and retry
- If there exists an N such that N > commitIndex and a majority of matchIndex[i] ≥ N, and log[N].term == currentTerm: set commitIndex = N

Howard H., Mortier R. Paxos vs Raft: Have we reached consensus on distributed consensus?

A difficult path to the production-ready replicator

Implementing an algorithm from a paper is simple, but...



A Paxos Algorithm

This summarises our simplified, Raft-style Paxos algorithm. The text in red is unique to Paxos.

State

Persistent state on all servers: (Updated on stable storage before responding to RPCs)
currentTerm latest term server has seen (initialized to 0 on first boot, increases monotonically)
log[ ] log entries; each entry contains command for state machine, and term when entry was received by leader (first index is 1)
Volatile state on all servers:
commitIndex index of highest log entry known to be committed (initialized to 0, increases monotonically)
lastApplied index of highest log entry applied to state machine (initialized to 0, increases monotonically)
Volatile state on candidates: (Reinitialized after election)
entries[] Log entries received with votes
Volatile state on leaders: (Reinitialized after election)
nextIndex[ ] for each server, index of the next log entry to send to that server (initialized to leader commit index + 1)
matchIndex[ ] for each server, index of highest log entry known to be replicated on server (initialized to 0, increases monotonically)

AppendEntries RPC

Invoked by leader to replicate log entries; also used as heartbeat
Arguments:
term leader's term
prevLogIndex index of log entry immediately preceding new ones
prevLogTerm term of prevLogIndex entry
entries[ ] log entries to store (empty for heartbeat; may send more than one for efficiency)
leaderCommit leader's commitIndex
Results:
term currentTerm, for leader to update itself
success true if follower contained entry matching prevLogIndex and prevLogTerm
Receiver implementation:
1. Reply false if term < currentTerm
2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm
3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it
4. Append any new entries not already in the log
5. If leaderCommit > commitIndex: set commitIndex = min(leaderCommit, index of last new entry)

RequestVote RPC

Invoked by candidates to gather votes
Arguments:
term candidate's term
leaderCommit candidate's commit index
Results:
term currentTerm, for candidate to update itself
voteGranted true indicates candidate received vote
entries[] follower's log entries after leaderCommit
Receiver implementation:
1. Reply false if term < currentTerm
2. Grant vote and send any log entries after leaderCommit

Rules for Servers

All Servers:
- If commitIndex > lastApplied: increment lastApplied and apply log[lastApplied] to state machine
- If RPC request or response contains term T > currentTerm: set currentTerm = T and convert to follower

Followers:
- Respond to RPCs from candidates and leaders
- If election timeout elapses without receiving AppendEntries RPC from current leader or granting vote to candidate: convert to candidate

Candidates:
- On conversion to candidate, start election: increase currentTerm to next t such that $t \bmod n = s$, copy any log entries after commitIndex to entries[], and send RequestVote RPCs to all other servers
- Add any log entries received from RequestVote responses to entries[]
- If votes received from majority of servers: update log by adding entries[] with currentTerm (using value with greatest term if there are multiple entries with same index) and become leader

Leaders:
- Upon election: send initial empty AppendEntries RPCs (heartbeat) to each server; repeat during idle periods to prevent election timeouts
- If command received from client: append entry to local log, respond after entry applied to state machine
- If last log index ≥ nextIndex for a follower: send AppendEntries RPC with log entries starting at nextIndex
  - If successful: update nextIndex and matchIndex for follower
  - If AppendEntries fails because of log inconsistency: decrement nextIndex and retry
- If there exists an N such that N > commitIndex and a majority of matchIndex[i] ≥ N: set commitIndex = N

Howard H., Mortier R. Paxos vs Raft: Have we reached consensus on distributed consensus?

Viewstamped Replication

... algorithm from the paper can be impractical

Viewstamped Replication requires sending the whole append-only log to other nodes periodically

We have to optimize the algorithm

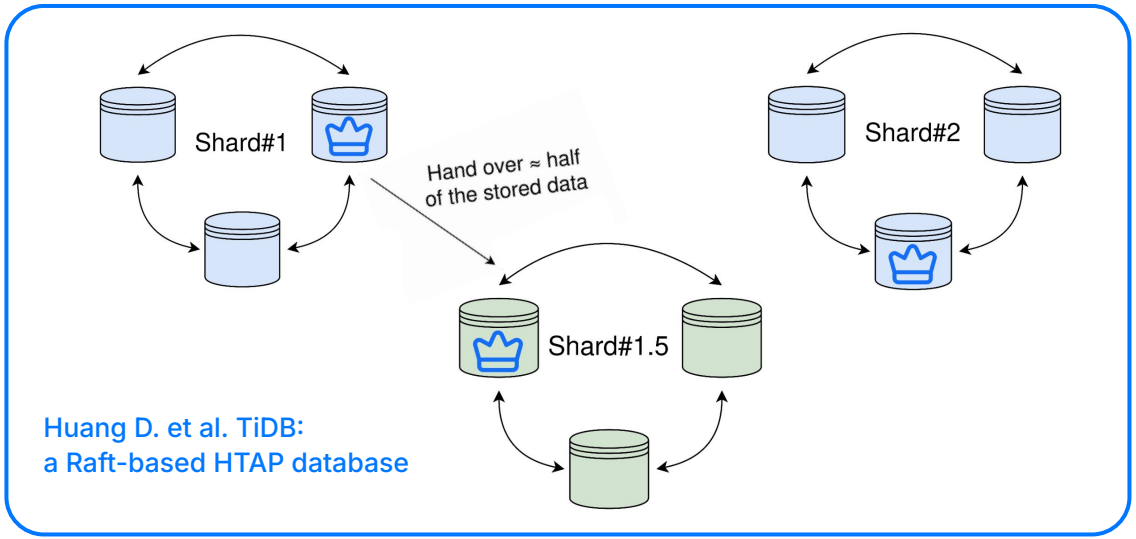
2. When replica i receives STARTVIEWCHANGE messages for its *view-number* from f other replicas, it sends a ⟨DOVIEWCHANGE *v, l, v', n, k, i*⟩ message to the node that will be the primary in the new view. Here v is its *view-number*, l is its log, v' is the view number of the latest view in which its status was *normal*, n is the *op-number*, and k is the *commit-number*.

- The *op-number* assigned to the most recently received request, initially 0.
- The *log*. This is an array containing *op-number* entries. The entries contain the requests that have been received so far in their assigned order.

Viewstamped Replication

... algorithm from the paper lacks features required for production-ready service

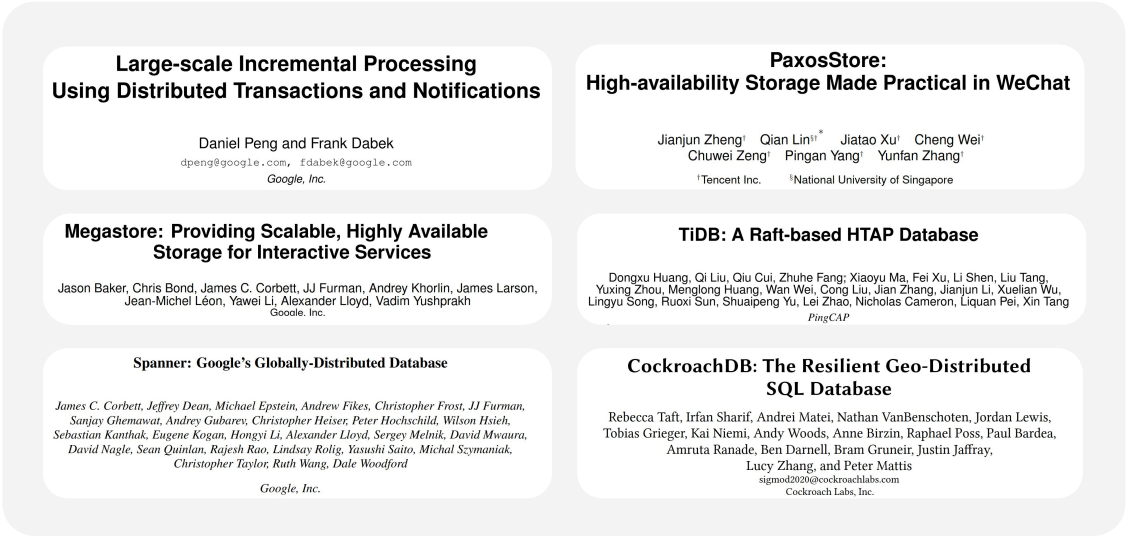
- Split overloaded shards into sub-shards
- Guarantee that no more than one replica is running garbage collection concurrently



Viewstamped Replication

... major changes should be introduced to the algorithm from the paper on the path to the production-ready system

Is it just our problem?



Large-scale Incremental Processing Using Distributed Transactions and Notifications

Daniel Peng and Frank Dabek
dpeng@google.com, fdabek@google.com
Google, Inc.

PaxosStore: High-availability Storage Made Practical in WeChat

Jianjun Zheng[†] Qian Lin[§†*] Jiatao Xu[†] Cheng Wei[†]
Chuwei Zeng[†] Pingan Yang[†] Yunfan Zhang[†]
[†]Tencent Inc. [§]National University of Singapore

Megastore: Providing Scalable, Highly Available Storage for Interactive Services

Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Léon, Yawei Li, Alexander Lloyd, Vadim Yushprakh
Google. Inc.

TiDB: A Raft-based HTAP Database

Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang; Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, Xin Tang
PingCAP

Spanner: Google's Globally-Distributed Database

James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, Dale Woodford

Google, Inc.

CockroachDB: The Resilient Geo-Distributed SQL Database

Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis
sigmod2020@cockroachlabs.com
Cockroach Labs, Inc.

Viewstamped Replication

... major changes should be introduced to the algorithm from the paper on the path to the production-ready system

Is it just our problem? No, it is common for the whole industry

9 Summary and open problems

We have described our implementation of a fault-tolerant database, based on the Paxos consensus algorithm. Despite the large body of literature in the field, algorithms dating back more then 15 years, and experience of our team (one of us has designed a similar system before and the others have built other types of complex systems in the past), it was significantly harder to build this system then originally anticipated. We attribute this to several shortcomings in the field:

- There are significant gaps between the description of the Paxos algorithm and the needs of a real-world system. In order to build a real-world system, an expert needs to use numerous ideas scattered in the literature and make several relatively small protocol extensions. The cumulative effort will be substantial and the final system will be based on an unproven protocol.

Chandra T. D., Griesemer R., Redstone J. Paxos made live: an engineering perspective

Testing distributed systems: Randomized approach

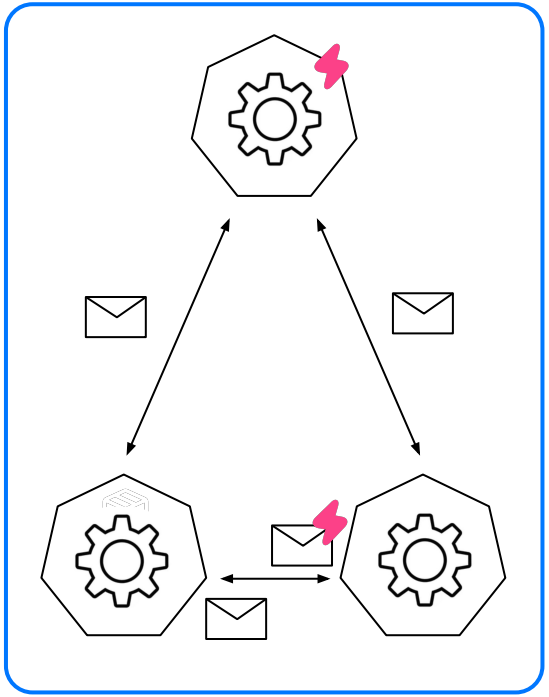
Run the system as a set of distinct processes

Introduce random errors to check fault tolerance

- Drop packets via iptables -j DROP
- Kill participants via kill -9

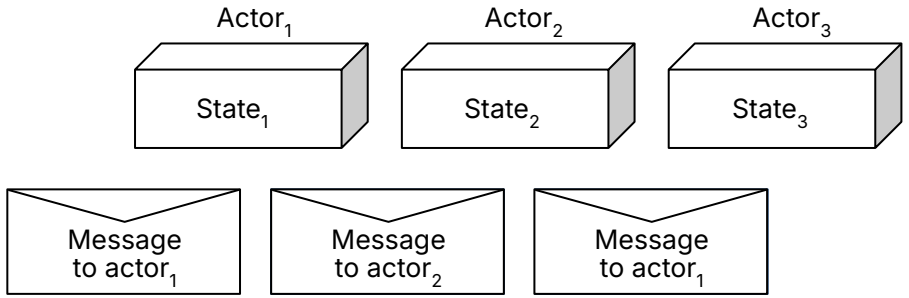
Not reproducible

Cannot force parallel processes repeat their actions in the exactly same order



Actor systems

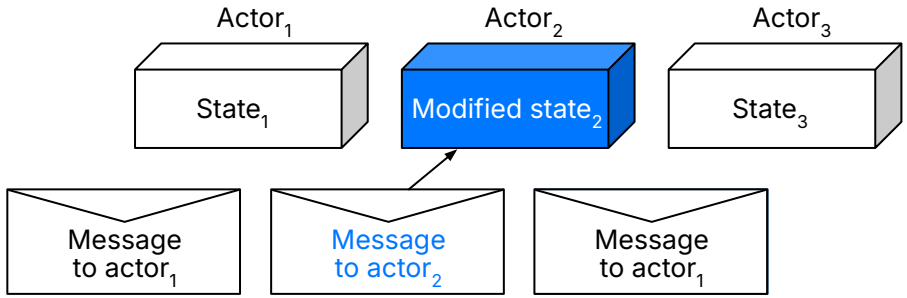
- To allow reproducible testing, we implement the system in the actor model
- Actor contains private state which cannot be accesses and modified by other actors and can process events
- The system consists of a collection of actors and a set of unprocessed events



Hewitt C. The actor model

Actor systems

- Testing is performed step-by-step sequentially
- On each step a single event is chosen and given for processing by the corresponding actor
- The actor modifies its state
- All other actor states are intact

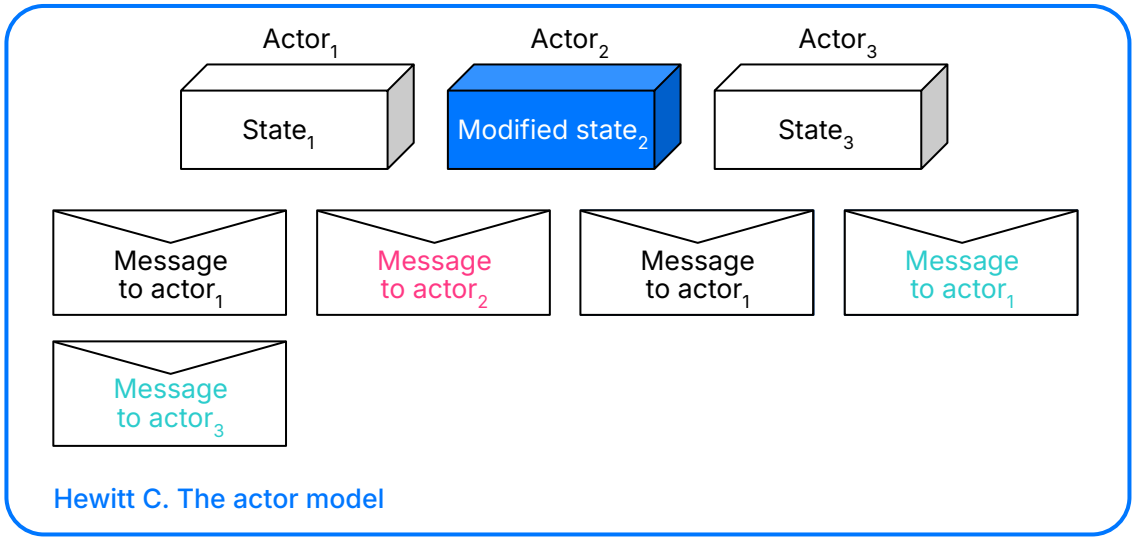


Hewitt C. The actor model

Actor systems

The actor adds new events to the set of uprocessed events during processing

- e.g., requests to send a couple of new messages



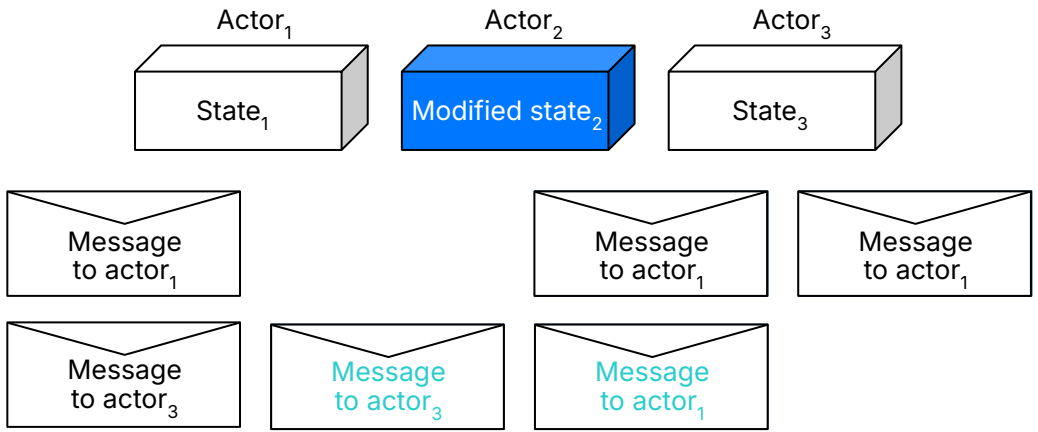
Actor systems

- The processed event is removed from the set
- Proceed to the next step, choose the next event

- In production actors are executed in parallel on multiple nodes

C++ Actor Framework

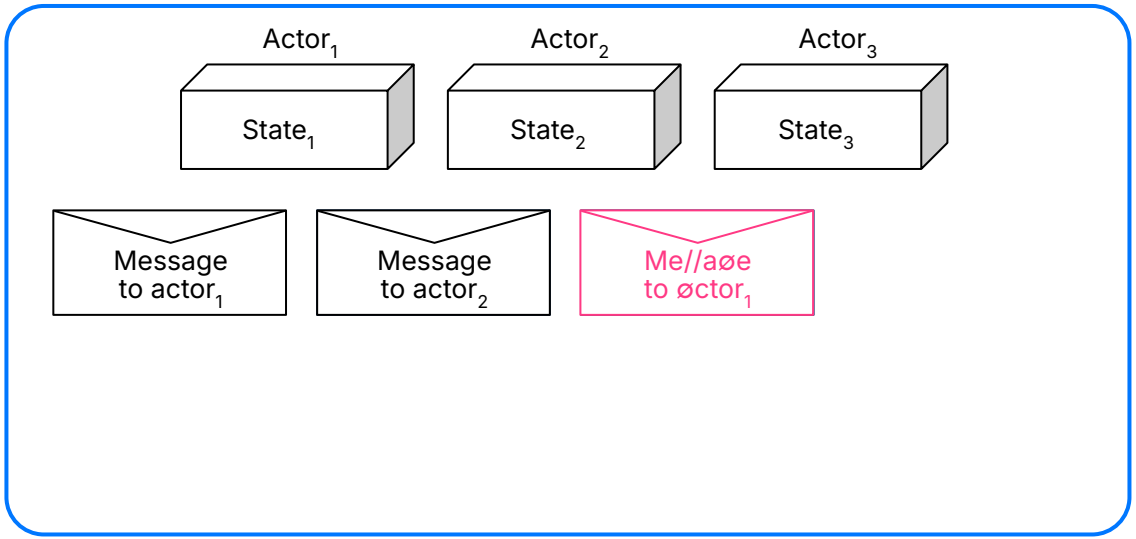
Akka



Hewitt C. The actor model

Deterministic errors

- Corrupt a message
- Stop delivering messages to a particular actor for K steps (Network partition)
- Drop a message (Unreliable network)
- Recreate an actor with a blanc stat (Crash & recovery)



TLA+/TLC to the rescue

Test is a sequence of processed events and introduced erors

- Process message M_1 by actor A_i
- Drop message M_2
- Corrupt message M_3
- Process message M_3 by actor A_j
- ...

Test can be replayed deterministically for debugging purposes

TLA+/TLC can help us test all possible execution scenarios within the given constraints

We can use fuzzing to generate execution scenarios

TLA+/TLC to the rescue

- Implement a model of the system
- The model consists of an array of actors and a set of unprocessed event

```
Init ==
    // Each replica contains an initial state
    /\ replicaStates = [ replica \ in Replicas |-> [
        status |-> "Normal",
        log |-> <<>>,
        viewNumber |-> 0,
        commitNumber |-> 0,
        downloadReplica |-> replica,
        catchupPos |-> 0,
        phase2 |-> FALSE
    ]]
    /\ queriesCount = 0 // No user has issued a request to our system
    /\ unprocessedEvents = {} // The set of unprocessed events is initially empty
```

TLA+/TLC to the rescue

- Implement a model of the system
- Transition corresponds to a processing of a single event
- Modifies the processing actor state
- Inserts new events to the unprocessed set

```
TimeoutStartViewChange(replica) ==
    // The transition is possblie only if the number of master election does not
    // exceed the threshould
    /\ ViewNumber(replica) < MaxViewNumber
    // Replica that started the master election process changes its state
    /\ replicaStates' = [replicaStates EXCEPT
        ![replica].status = "ViewChange",
        ![replica].viewNumber = @ + 1,
        ![replica].downloadReplica = "None",
        ![replica].catchupPos = 0.
        ![replica].phase2 = FALSE ]
    // Replica sends StartViewChange messages to all other replicas
    /\ unprocessedEvent' = unprocessedEvent ∪ { [
            type |-> "StartViewChange",
            viewNumber |-> ViewNumber(replica) + 1,
            to |-> otherReplica
        ] : otherReplica \in Replicas }
    // The number of user requests does not change
    /\ UNCHANGED << queriesCount >>
```

TLA+/TLC to the rescue

- Implement a model of the system
- Specify invariants that must hold during all executions

```
PrefixLogConsistency ==
    \A r1, r2 \in Replicas:
        \/ IsPrefixOf(SubSeq(Log(r1), 1, CommitNumber(r1)),
                      SubSeq(Log(r2), 1, CommitNumber(r2)))
        \/ IsPrefixOf(SubSeq(Log(r2), 1, CommitNumber(r2)),
                      SubSeq(Log(r1), 1, CommitNumber(r1)))

LogCommitProgress ==
    []<> \A r \in Replicas:
            /\ Status(r) = "Normal"
            /\ LogQueryCount(r) = MaxLogQueryCount
            /\ CommitNumber(r) = LogLength(r)
```

Implementation approach

We use neither PlusCal nor any other tools that transpile imperative code to the TLA+

We use “specification-first” approach

— First TLA+-specification is designed and checked
— Then the imperative code is written based on the specification
— To modify an existing code we first modify the specification, and then the imperative code

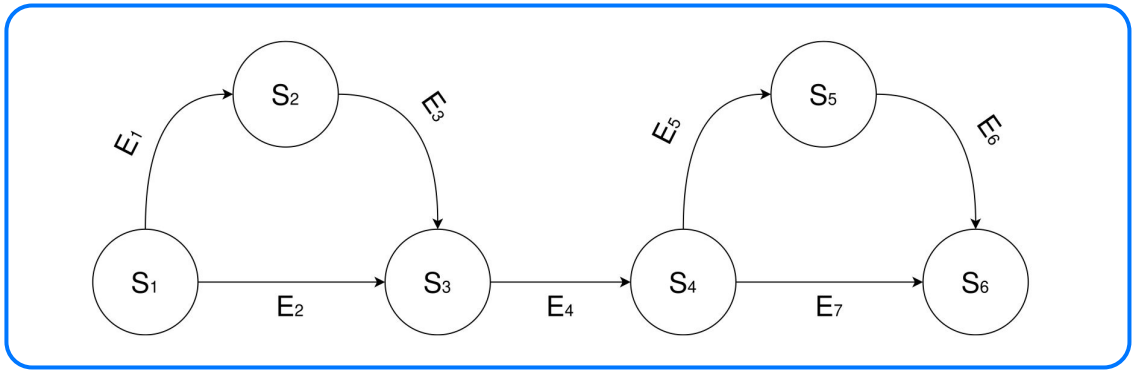
Bringing the gap between the model and the code

TLC builds the finite transition graph of the actor system

- States are possible states of the system, edges are processed events
- TLC checks that all invariants hold fot that graph

Is the code behaviour consistent with the model?

The task is to generate a set of tests to verify it



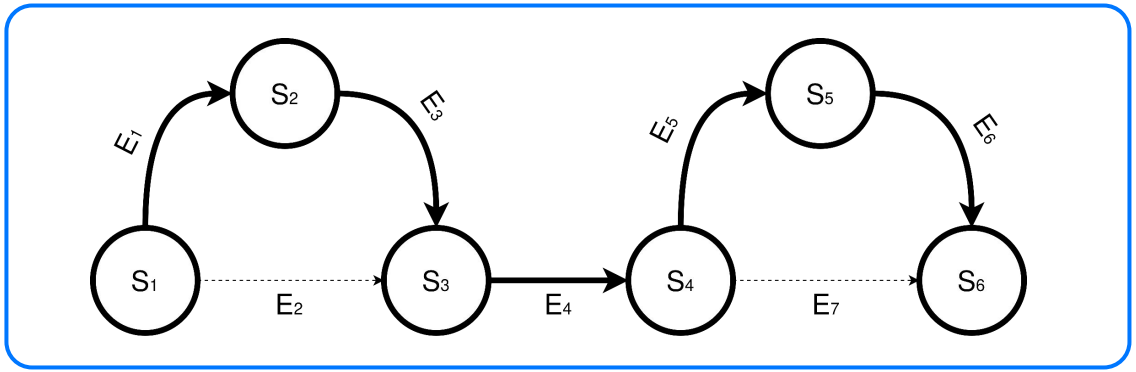
Bringing the gap between the model and the code

Each test is a path starting in the initial state (graph source node)

- The path corresponds to a sequence of processed events

Exhaustive test suite must cover each edge at least once

- Covering each node is not sufficient
- We must verify that the processing of each event is consistent between the model and the code

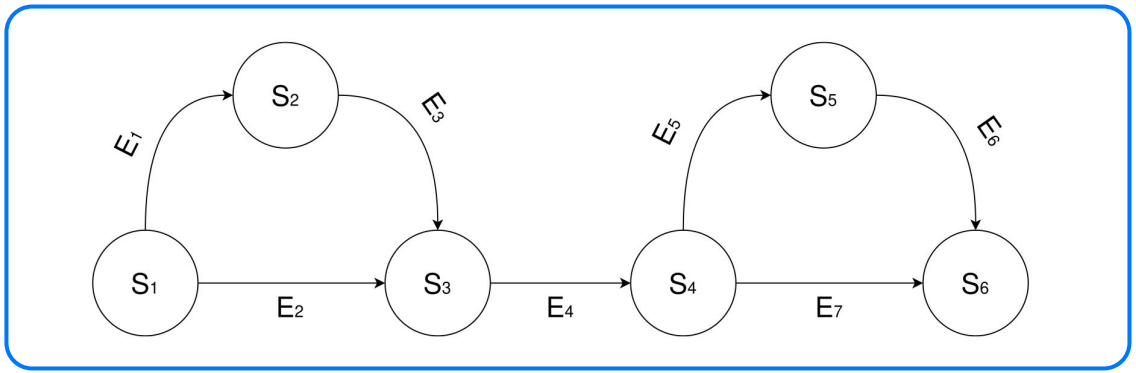


Bringing the gap between the model and the code

Two tests are required for the exhaustive test suite:

- $E_1 \rightarrow E_3 \rightarrow E_4 \rightarrow E_5 \rightarrow E_6$
- $E_2 \rightarrow E_4 \rightarrow E_7$

We want to make this exhausive test suite as small as possible



Another soultions: Modelator

- **Supports only Python programs**

  Does not support our primary languages: C++ and Go

- **Generates only random tests**

  Does not support exhaustive testing

- **Supports only small subset of the TLA+ language**

Another soultions: Kayfabe

— **Supports only C# programs**

No support for C++ or Go

Dorminey S. Kayfabe: model-based program testing with TLA+/TLC

— **Reduces test generation problem to a Travelling Salesman Problem**

Which requires either solving an NP-hard task in exponential time or using approximate solvers

Our solution supports graphs with tens of millions of states

«Since the route solver is NP-hard, models are constrained to only a few hundred states to remain feasible»

— **Visits each node at lest once, does not guarantee visiting all the edges**

Another soultions: a tool by Wang et. al.

- Much slower than our solution
- We generate and execute thousands

Column Time in Table 3 shows the time of executing test cases generated applying both EC and POR. On average, a ZooKeeper test case takes about 10 seconds. Xraft and Raft-java take 7 seconds and 5 seconds to execute a test case, respectively. It is reasonable since ZooKeeper is more complex than the other two distributed systems.

Table 3. Testing Effort

System	State	Path_{EC}	$\text{Path}_{\text{EC+POR}}$	Time
Xraft	91,532	296,154	39,047	75 h
Raft-java	23,911	85,976	9,829	13 h
ZooKeeper	105,054	342,770	44,361	123 h

Wang D., Dou W., Gao Y., et al. Model checking guided testing for distributed systems

Minimal cost circulation problem

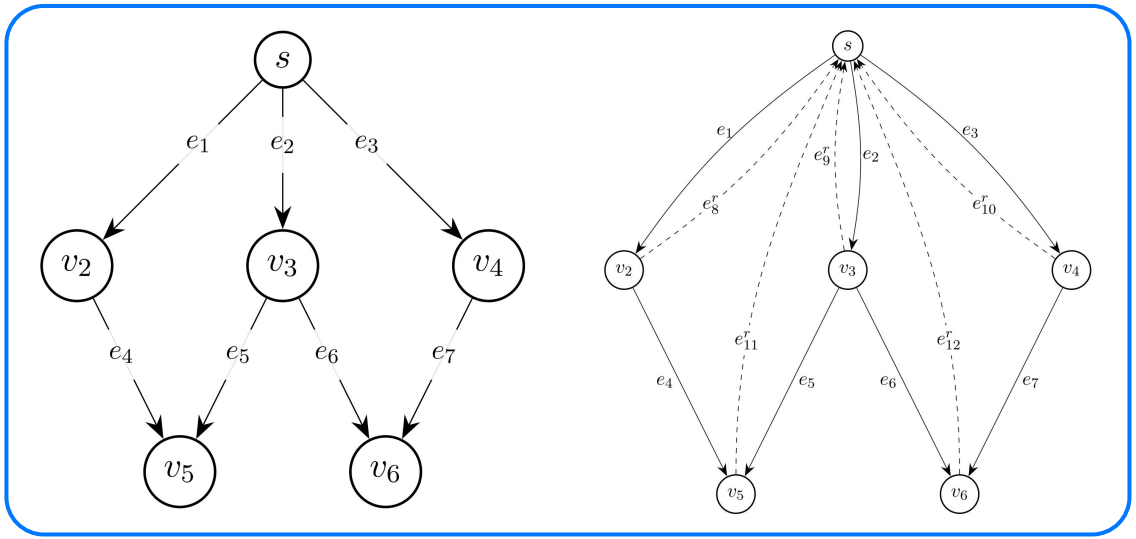
- **Given a multigraph G = (V, E)**
- **Given three functions L, U, C : E → $\mathbb{N}$**
- **Find a function F : E → $\mathbb{N}$ such that**
  - $\forall e \in E : L(e) \leq F(e) \leq U(e)$
  - $\forall v \in V : \sum_{u \to v \in E} F(u \to v) = \sum_{v \to w \in E} F(v \to w)$
  - From all matching functions, find the one that minimizes $\sum_{e \in E} C(e) \cdot F(e)$
- **Can be solved in**

$$O(|V| \cdot |E| \cdot \log |V| \cdot \min \left(|E| \cdot \log |V|, |V| \cdot \max_{e \in E} C(e) \right)).$$

Goldberg A.V., Tarjan R.E. Finding minimum-cost circulations by canceling negative cycles

Finding minimal exhaustive test suite

— Given the transition multigraph G = (V, E), add reverse edge from each node to the source node, obtaining multigraph $G_R = (V, E \cup E_R)$

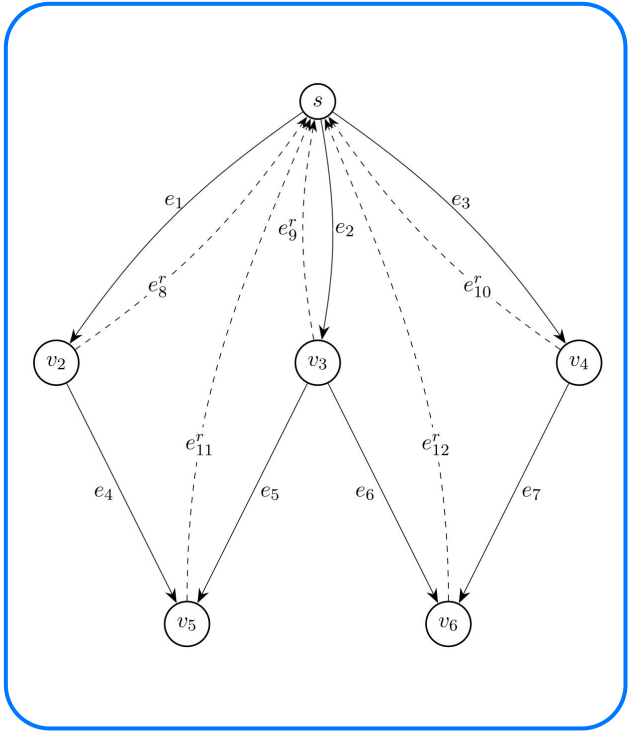


Finding minimal exhaustive test suite

- For reverse edges, set $\forall e \in E_R : L(e) = 0$
- For original edges set $\forall e \in E : L(e) = 1$

  Each original edge must have non-zero flow
- $\forall e \in E \cup E_R : U(e) = +\infty$

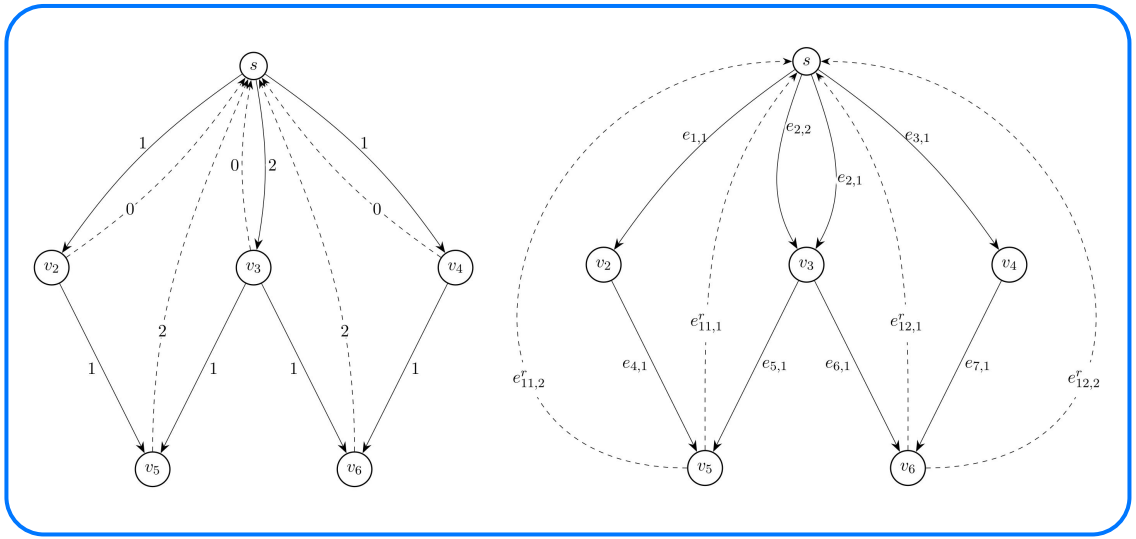
  No flow upper bound
- Solve the circulation problem



Finding minimal exhaustive test suite

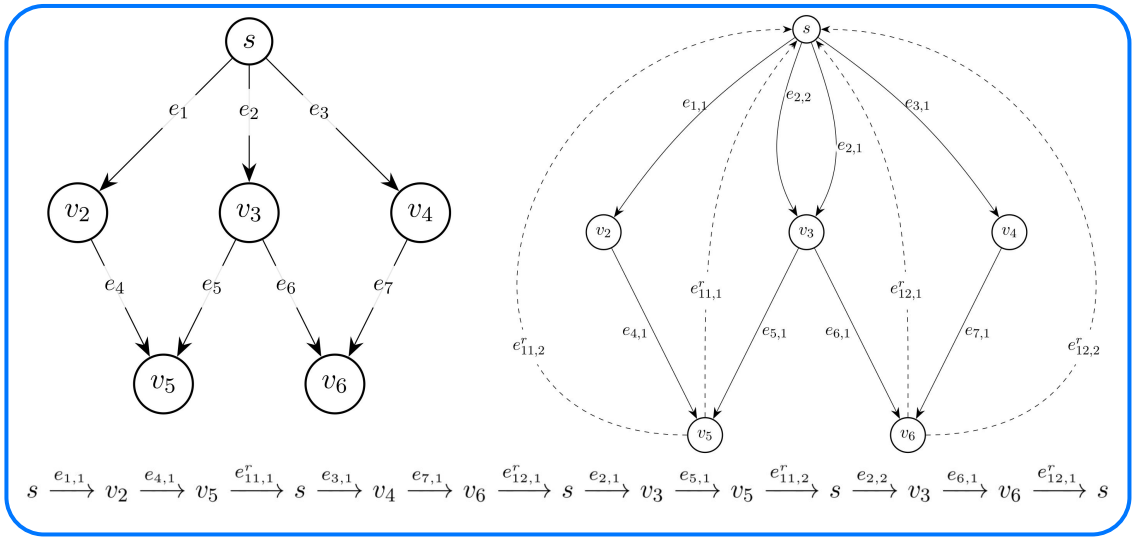
— Create a multigraph G_F by repearing each edge e of G_R F(e) times

  Each original edge e ∈ E exist in GF at least once



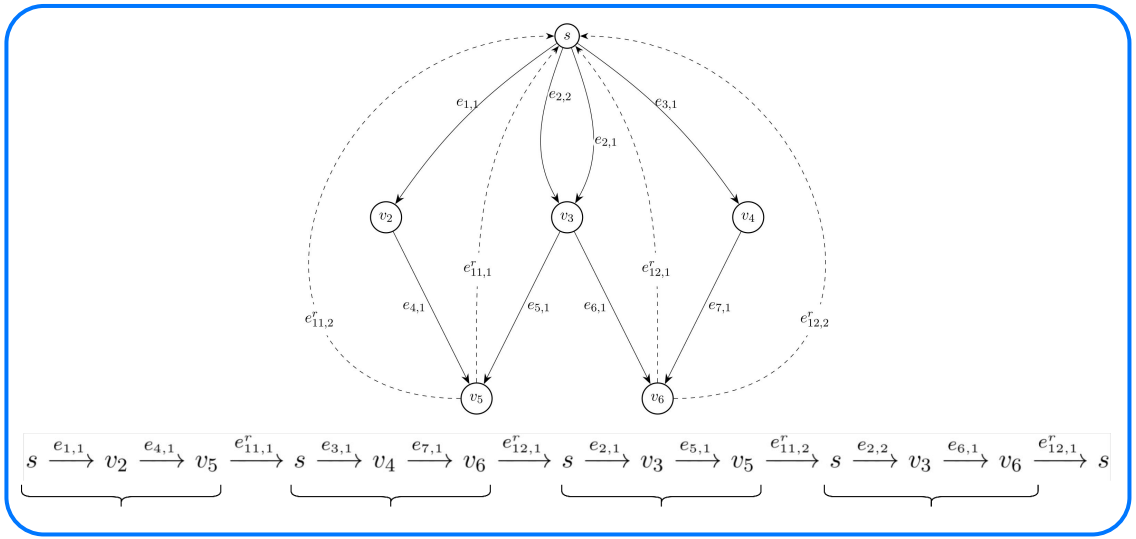
Finding minimal exhaustive test suite

- G_F has en Eulerian cycle
- Covers each original edge from E at least once



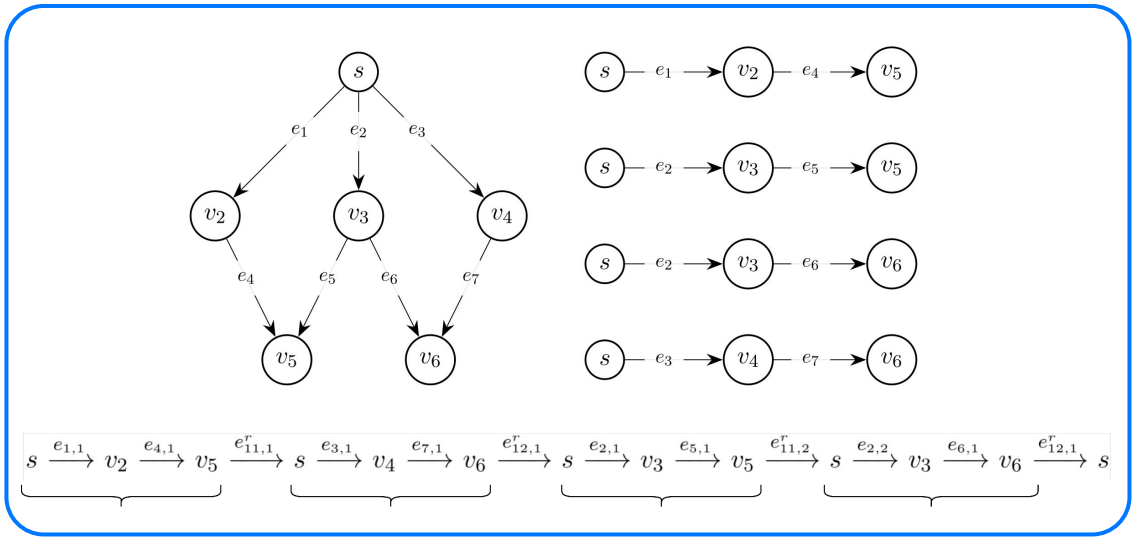
Finding minimal exhaustive test suite

- Traverse the Eulerian cycle
- Each time we follow the reverse edge to the source node, we start a new test



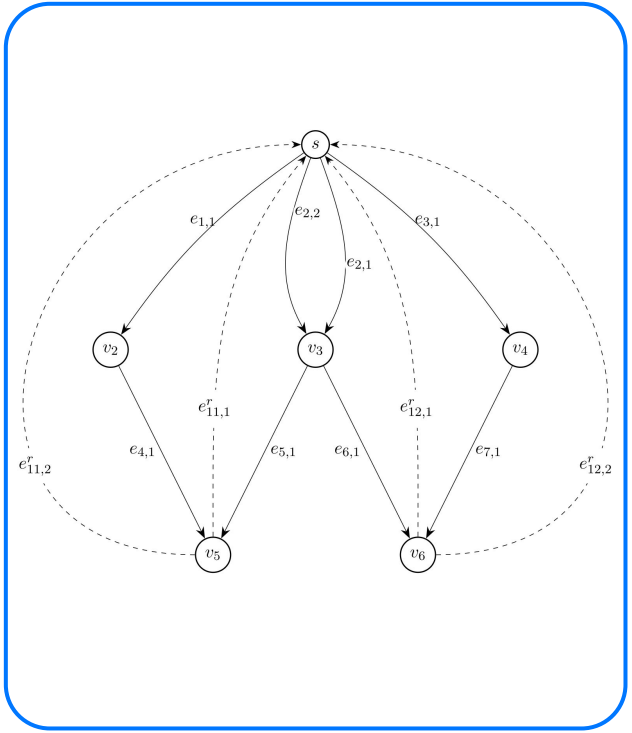
Finding minimal exhaustive test suite

- Each path is transformed into a sequence of events and executed as a test case



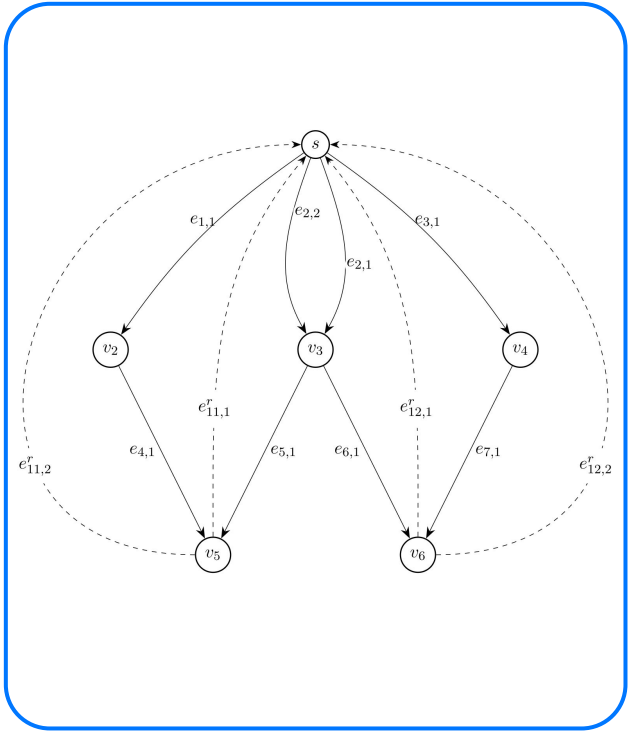
Finding minimal exhaustive test suite

- To minimize test cases count:
  1. for reverse edges, set $\forall e \in E_R : C(e) = 1$
  2. for original edges set $\forall e \in E : C(e) = 0$
- Minimizing $\sum_{e \in E} C(e) \cdot F(e)$ will lead to minimizing the flow over revese edges from E_R
- Thus minimizing the number of test cases



Finding minimal exhaustive test suite

- To minimize the total processed events count:
  1. for reverse edges, set $\forall e \in E_R : C(e) = 0$
  2. for original edges set $\forall e \in E : C(e) = 1$
- Minimizing $\sum_{e \in E} C(e) \cdot F(e)$ will lead to minimizing the flow over original edges from E
- Thus minimizing the total number of processing events



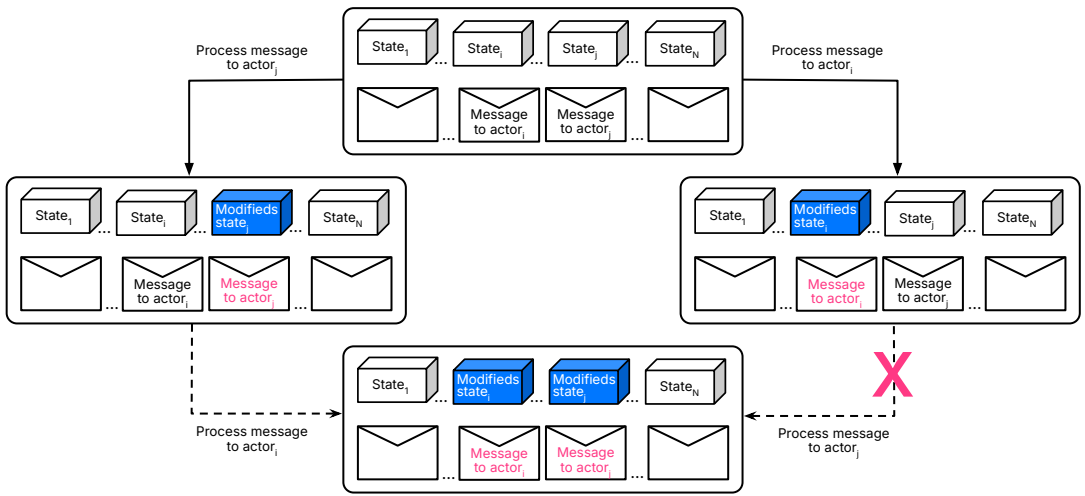
Test suite execution

- Each system actor must implement .ToModel() method that represent actor state as a set of TLA+ variables
- At each step of the test we verify that ∀ i : Actors[i].ToModel() equals the state of i-th TLA+ actor
- Compare status, log, viewNumber, commitNumber, dowloadReplica, catchupPos and phase2 fields

```
/\ replicaStates = [ replica \ in Replicas |-> [
    status |-> "Normal",
    log |-> <<>>,
    viewNumber |-> 0,
    commitNumber |-> 0,
    downloadReplica |-> replica,
    catchupPos |-> 0,
    phase2 |-> FALSE
]]
```

Graph prunning

If messages M_1 and M_2 are destined for different actors and exist at the unprocessed set at the same step, processing M_1, M_2 leads to the same state as processing M_2, M_1



Fischer M. J., Lynch N. A., Paterson M. S. Impossibility of distributed consensus with one faulty process

Practical results:
Test suite generation

Model constarints	Nodes count |V|	Edges count |E|	Graph diameter D	Tests count	Graph building time	Tests generation time
OldConfig ← {A,B,C} NewConfig ←{A,C,D} Witnesses ← {A} OldMaxView← 1 NewMaxView ← 1	2 476 101	23 404 512	52	10 298 526	14 min.	11 min.
OldConfig ← {A,B,C} NewConfig ←{A,C,D} Witnesses ← {A} OldMaxView← 2 NewMaxView ← 1	14 394 163	129 133 639	59	64 025 854	93 min.	69 min.
OldConfig ← {A} NewConfig ←{A,C,D} Witnesses ← {C} OldMaxView← 2 NewMaxView ← 1	2 207 933	16 340 753	54	9 847 523	7 min.	195 sec.

Practical results: Tests execution

Model constarints	Graph diameter D	Tests count	Testing time, one core	Testing time, 16 cores	Speed up
OldConfig ← {A,B,C} NewConfig ←{A,C,D} Witnesses ← {A} OldMaxView← 1 NewMaxView ← 1	52	10 298 526	3min. 46 sec.	16 sec.	x14.1
OldConfig ← {A,B,C} NewConfig ←{A,C,D} Witnesses ← {A} OldMaxView← 2 NewMaxView ← 1	59	64 025 854	23 min. 39 sec.	1 min. 38 sec.	x14.5
OldConfig ← {A} NewConfig ←{A,C,D} Witnesses ← {C} OldMaxView← 2 NewMaxView ← 1	54	9 847 523	3 min. 15 sec.	14 sec.	x13.9

Conclusion

The tool generates a minimal-size test suite that altogether covers all edges of the transition graph

Executes each test and checks that each transition is consistent between the model and the code

1

The longest testing time was 93 minutes to generate transition graph, 69 minutes to generate tests and 24 minutes to execute ≈ 64 millions test cases on a single core

- On a relatively big graph with 14 millions of nodes and 129 millions of edges
- Our tool is capable of generating and executing 5700 test cases per second on a single core, it is much faster than our counterparts

2

Future work

We are experimenting with advanced graph prunning techniques

1

On big graphs generating random execution paths instead of checking the whole state space seems like a prominent approach

Dynamic partial order reduction must be applied to skip execution scenarios that do not bear any additional information, compared to already traversed paths

2

We are looking for more sophisticated minimizatin algorithms

Capable of, for example, minimizing maxinal test length while keeping the total number of operations below a threshold

3

Thanks for your attention