Model checking of distributed algorithms:

from classics towards Tendermint blockchain

Igor Konnov

VMCAI winter school, January 16-18, 2020

INTERCHAIN
FOUNDATION

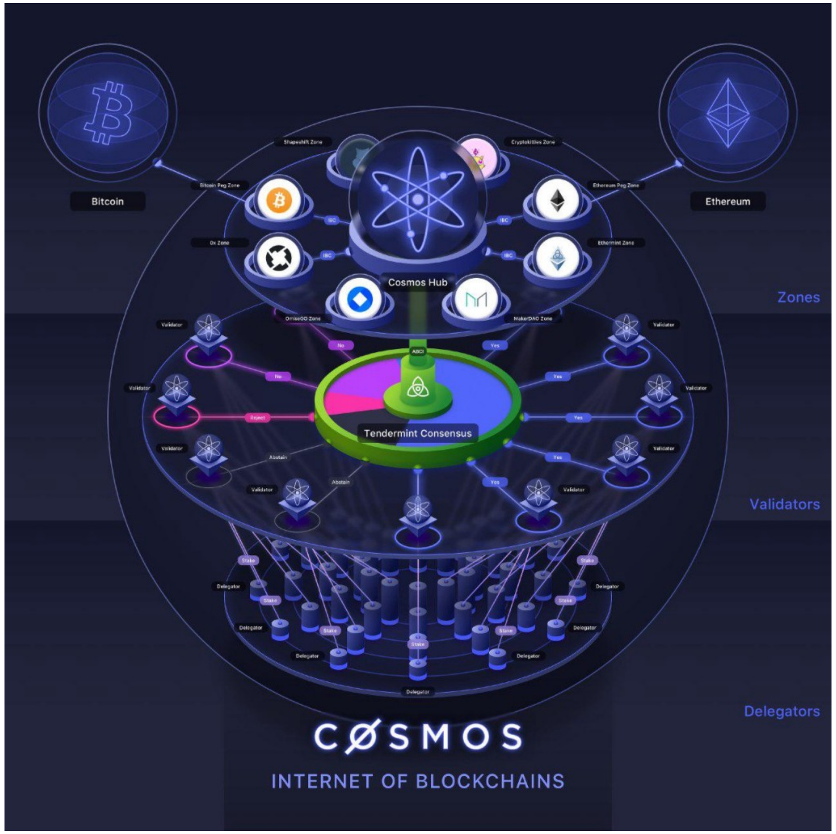
Swiss non-profit foundation

Supports R&D of applications that are:

- secure and scalable
- decentralized

Main focus:

- the Cosmos Network
- Tendermint consensus



Cosmos

A decentralized network of independent blockchains

Blockchains are powered by BFT consensus like Tendermint

They communicate over Inter-Blockchain Communication protocol

`[cosmos.network/ecosystem]`

Tendermint

Byzantine fault-tolerant State Machine Replication middleware

Consensus protocol adapts DLS & PBFT for blockchains:

- wide area network
- hundreds of validators and thousands of nodes
- communication via gossip

efficient and **open source**

Theory: [arxiv.org/abs/1807.04938]

Verification-Driven Development of Tendermint:

1. PODC-style specifications in English

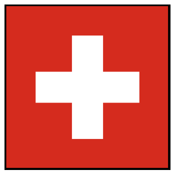
2. TLA^+ specifications (make English formal / fix it)

   - model checking for finding bugs in TLA^+ specs

3. Implementation in Rust

   - model-based testing of the implementation using TLA^+ specs

4. Automated verification of TLA^+ specs

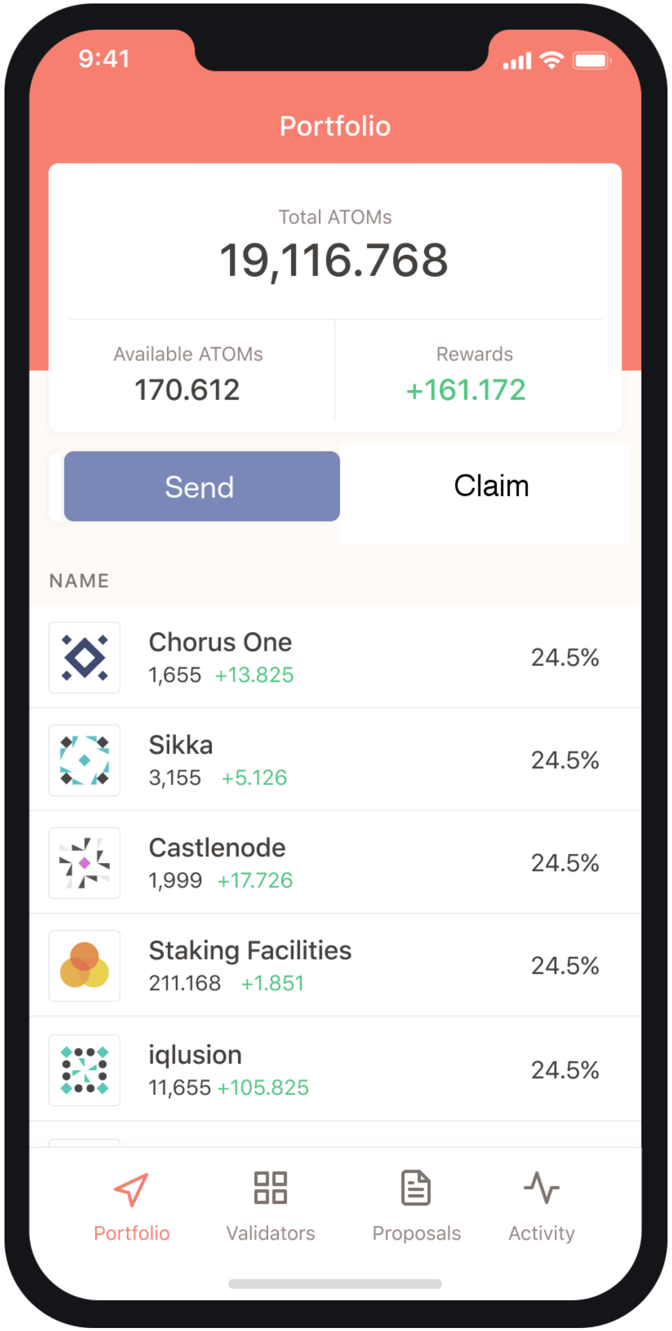


Timeline

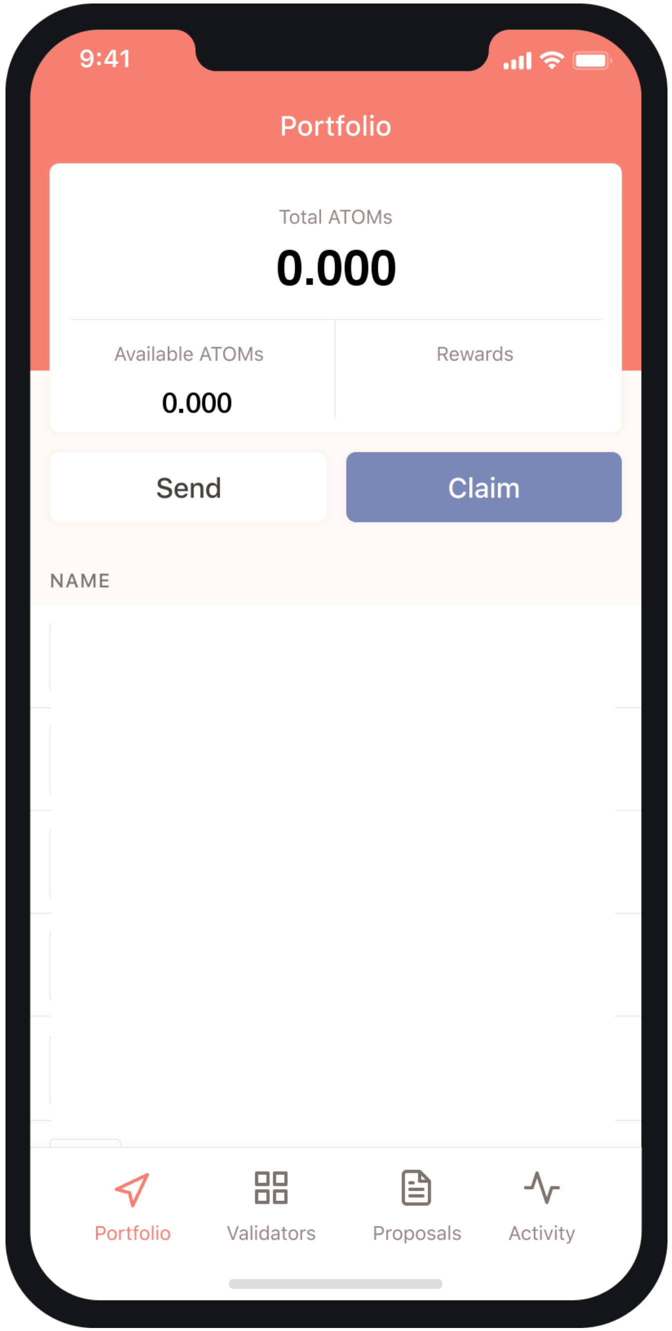
- Introduction to **fault-tolerant** distributed algorithms
- Verifying **synchronous** threshold-guarded algorithms
- Verifying **asynchronous** threshold-guarded algorithms
- Can we verify **Tendermint consensus?**

Please send me some money

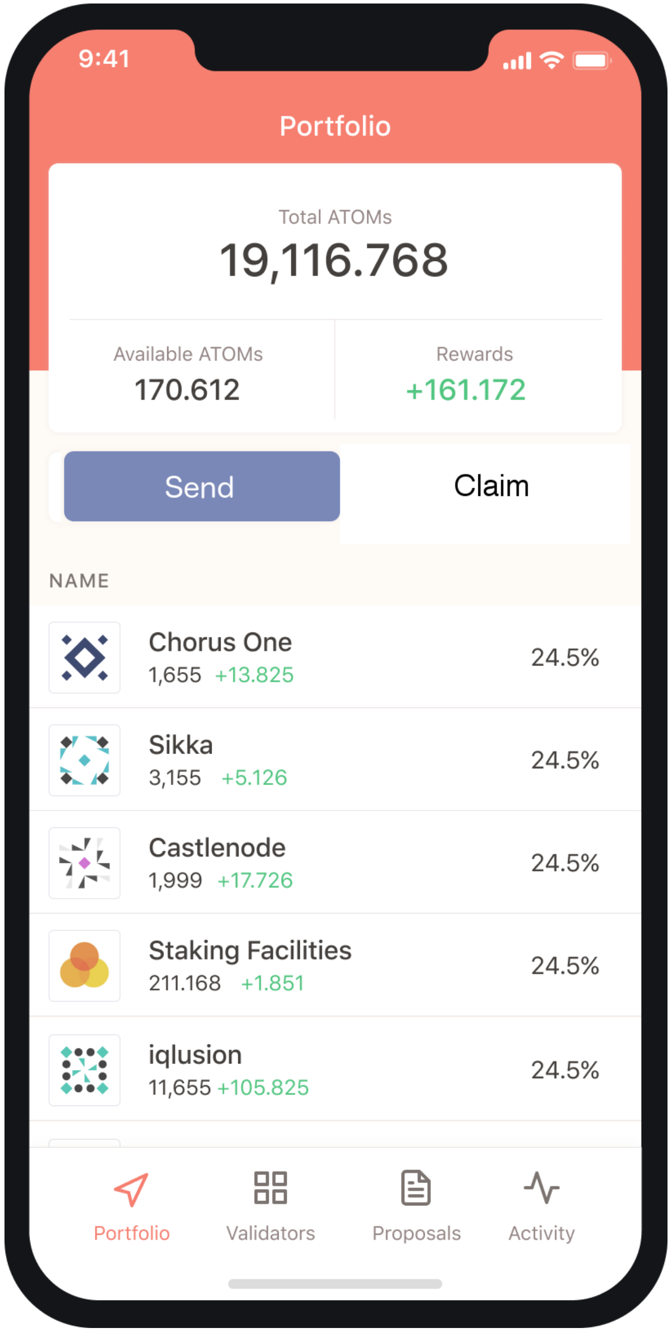
I will transfer you 100 atoms



lunie.io



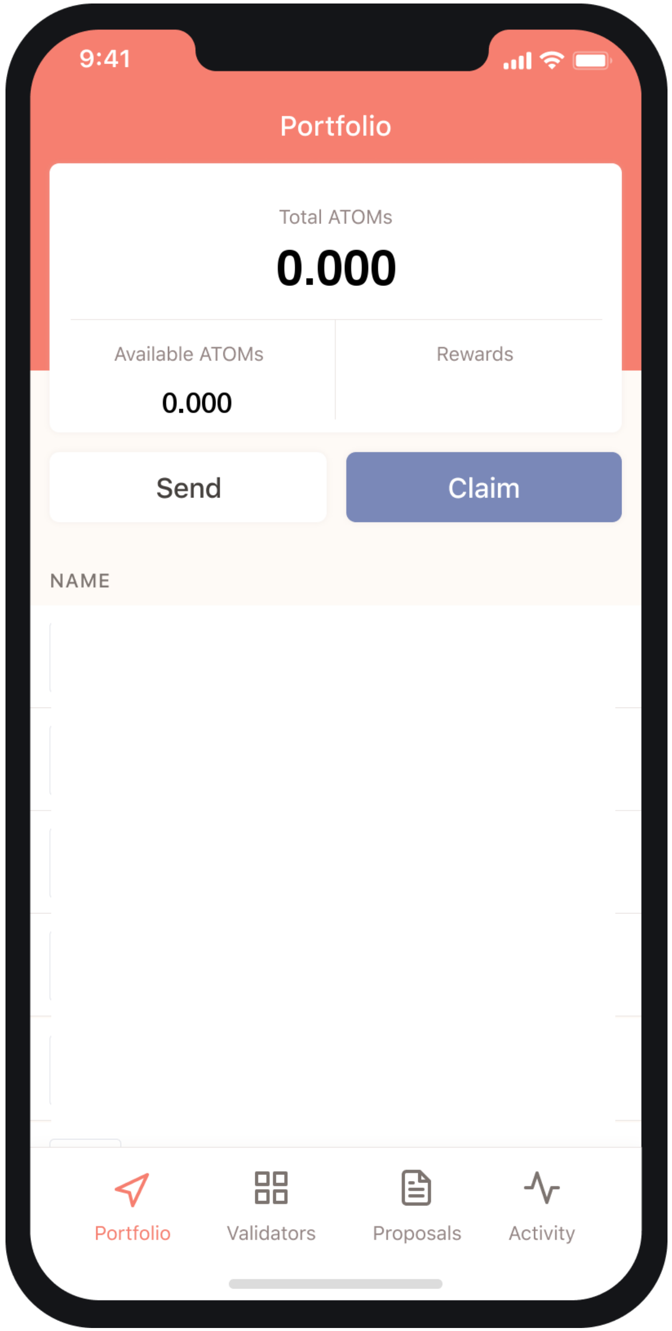
lunie.io



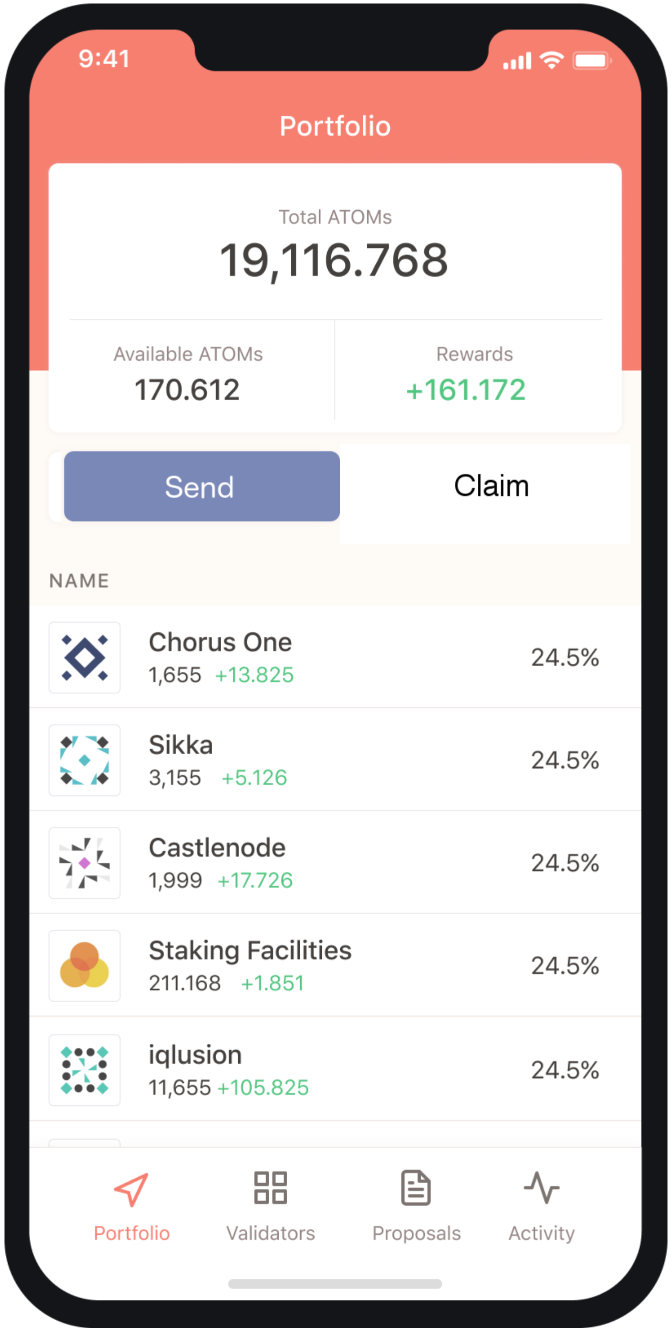
lunie.io

Send 100 ATOMs to

`cosmos1wze...`



lunie.io

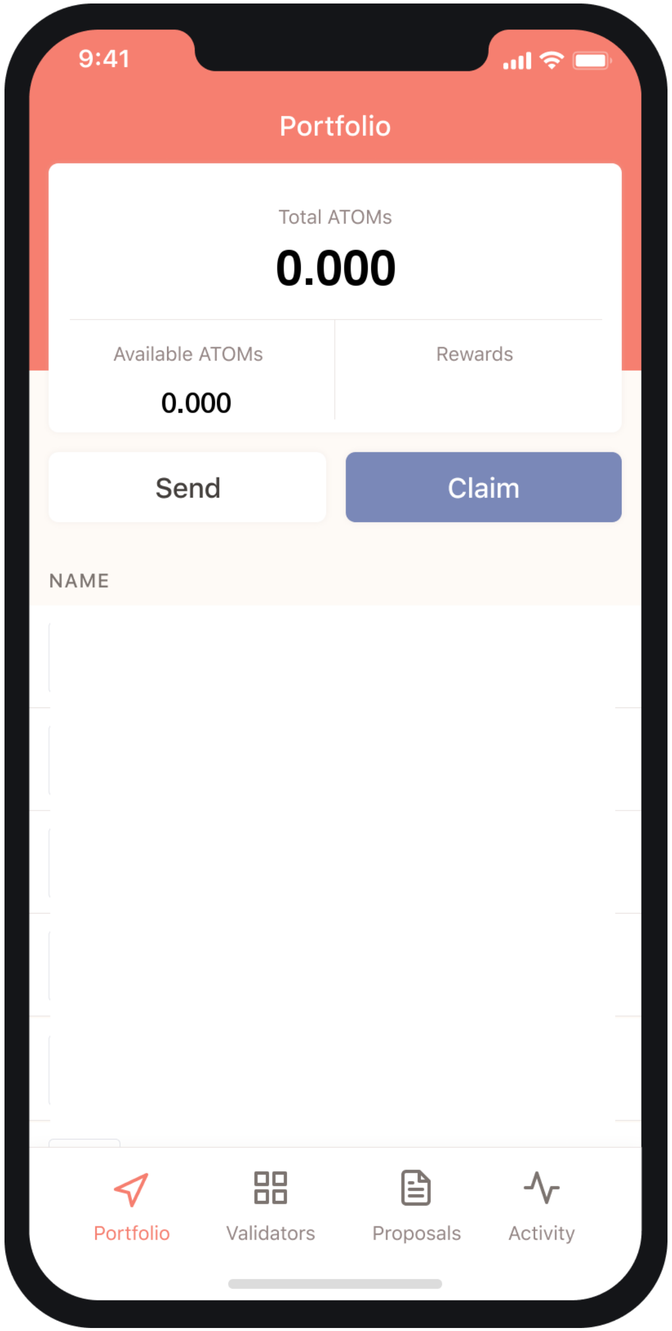


lunie.io

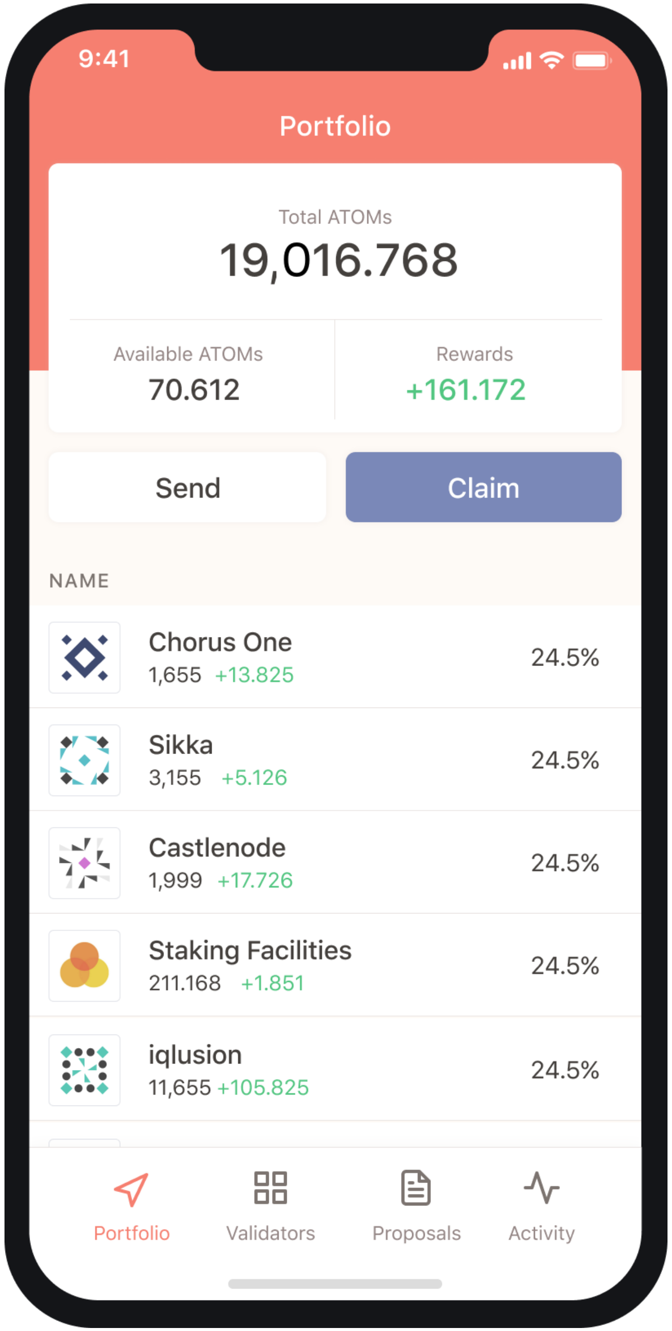
Send 100 ATOMs to

`cosmos1wze...`

COSMOS

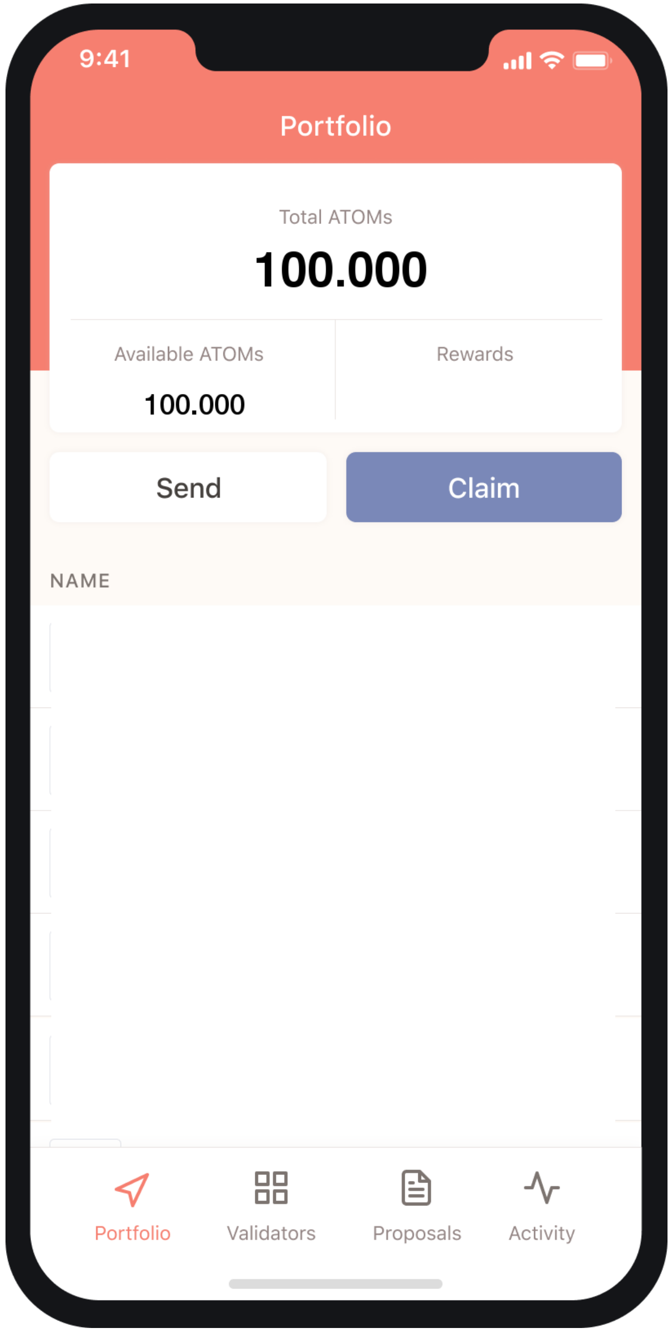


lunie.io



lunie.io

CØSMOS



lunie.io

Features of the system

Distributed

logically and geographically

Fault-tolerant

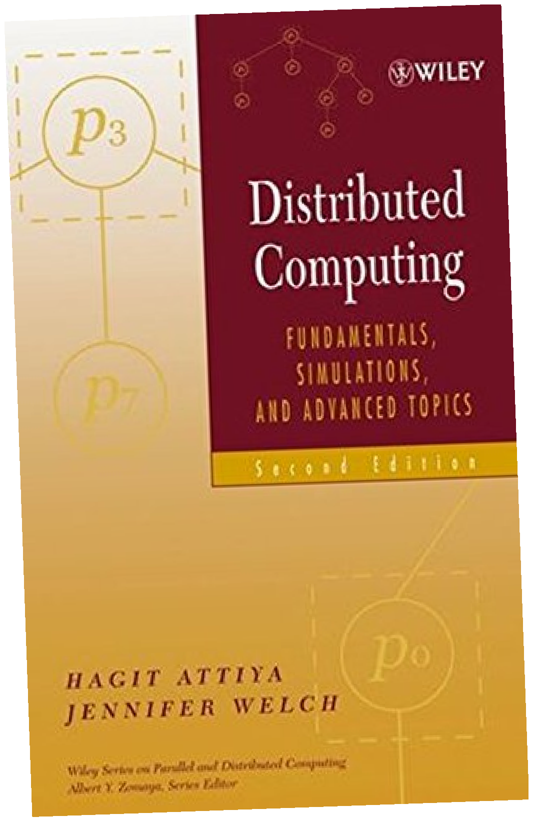
individual machines may crash and even act malicious

Safe and live

e.g., no double spending

every transaction is eventually committed

How to build such a system?



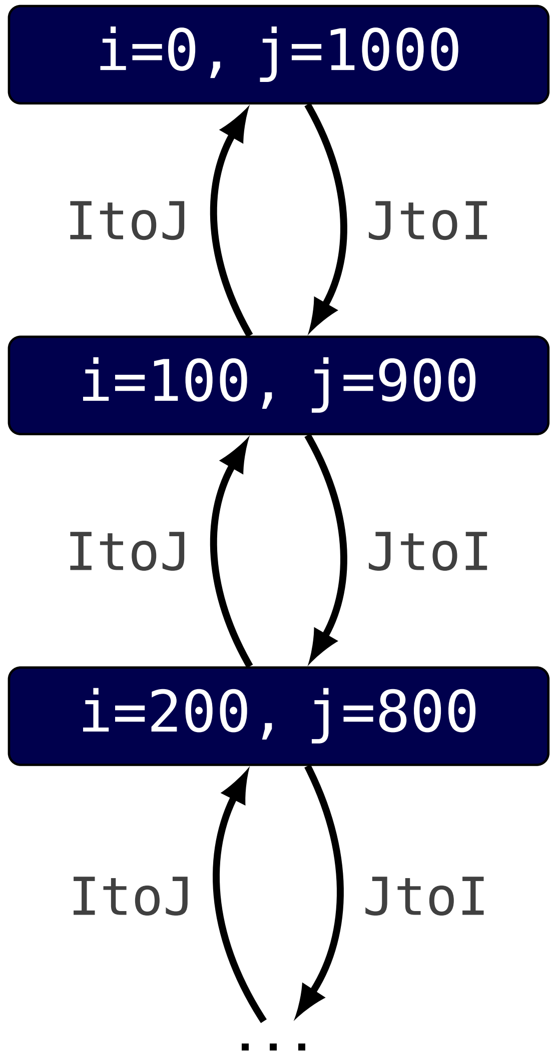
sequential code:

```
int i = 0, j = 1000;

while (true) {
  begin_tx();

  if (recv(ItoJ))
    { i -= 100; j += 100; }

  if (recv(JtoI))
    { i += 100; j -= 100; }

  if (i < 0 || j < 0)
    abort_tx();
  else
    commit_tx();
}
```

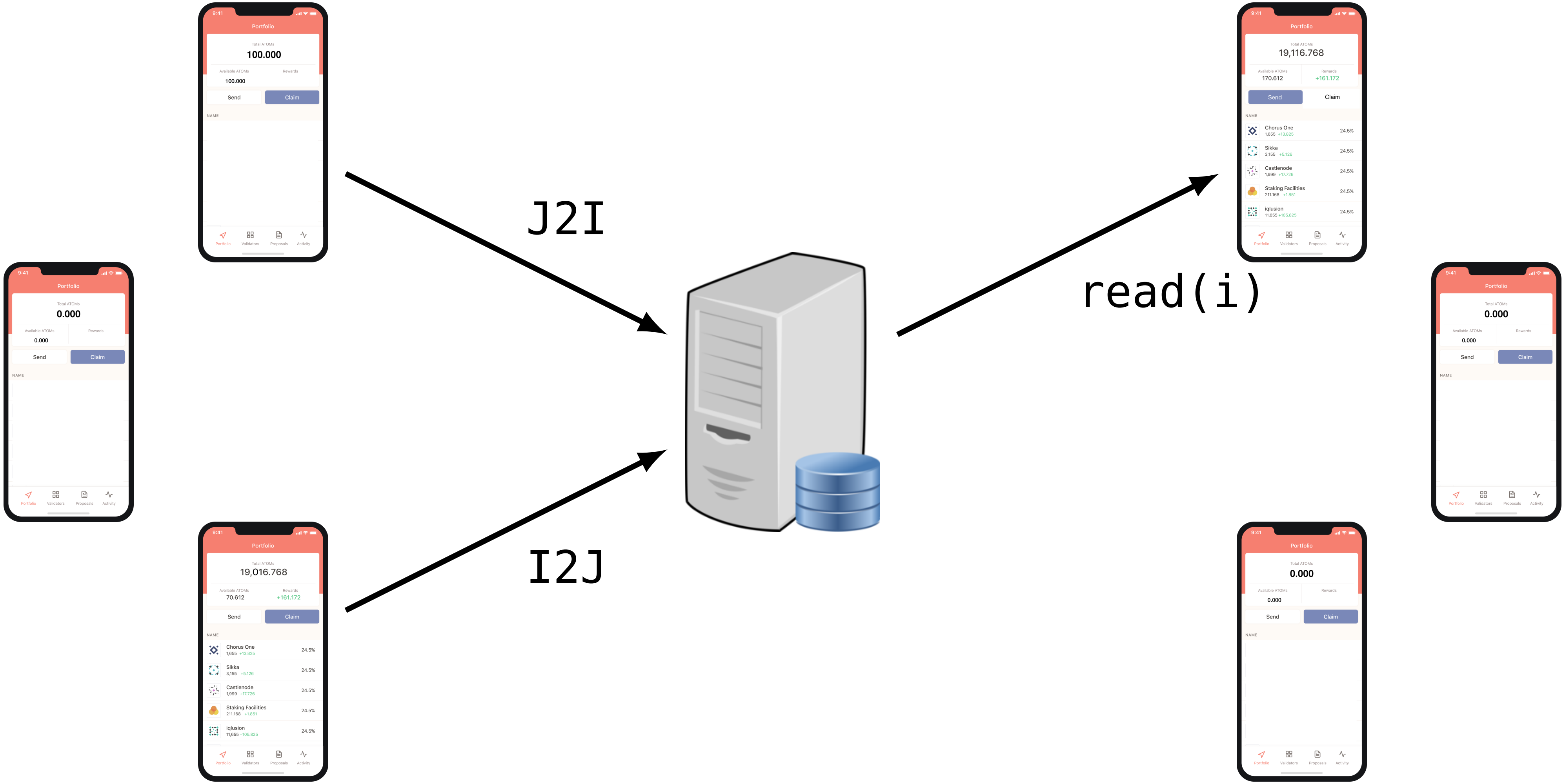
sequential code:

```
int i = 0, j = 1000;

while (true) {
  begin_tx();

  if (recv(ItoJ))
    { i -= 100; j += 100; }

  if (recv(JtoI))
    { i += 100; j -= 100; }

  if (i < 0 || j < 0)
    abort_tx();
  else
    commit_tx();
}
```

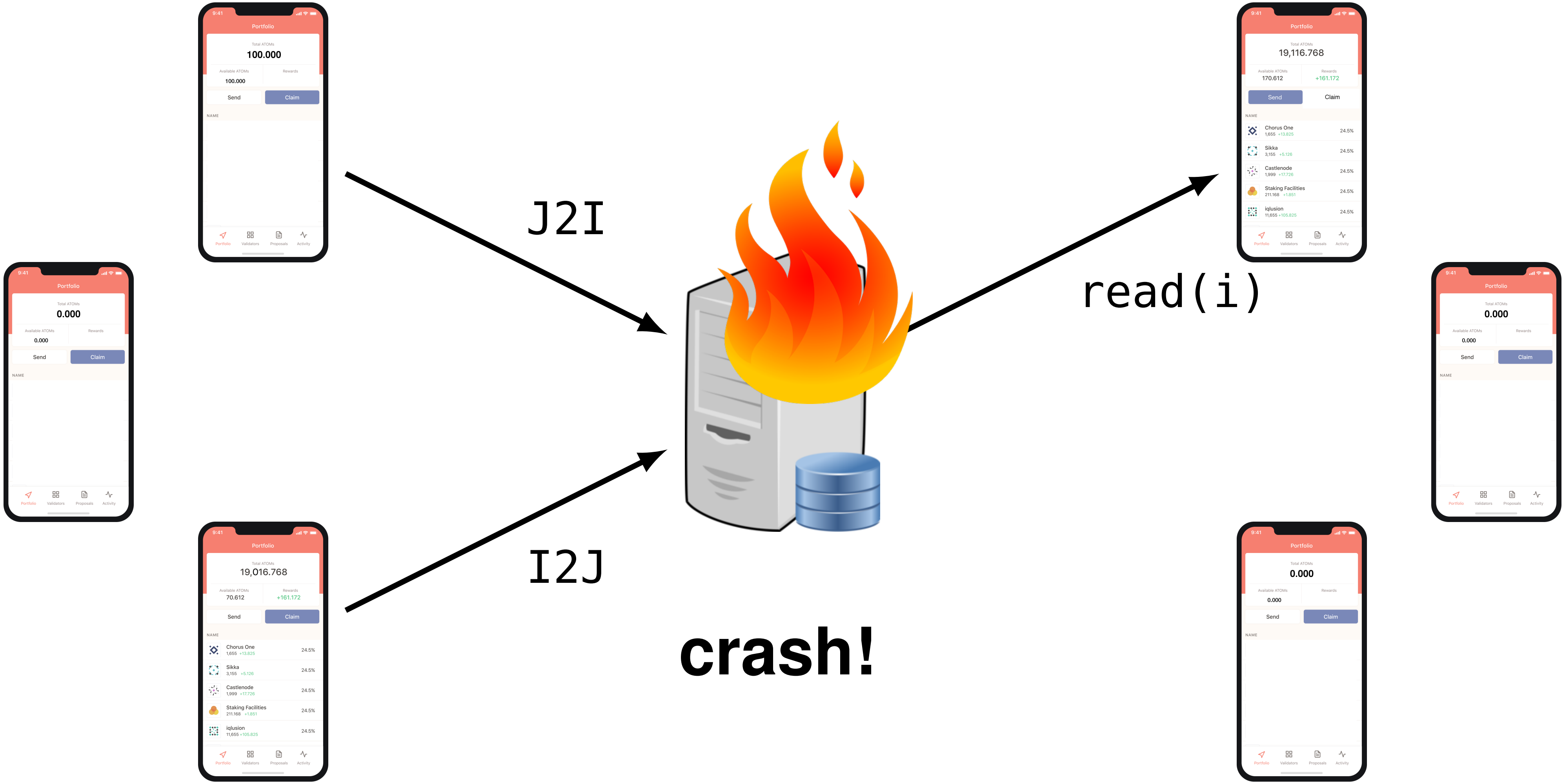
state machine:



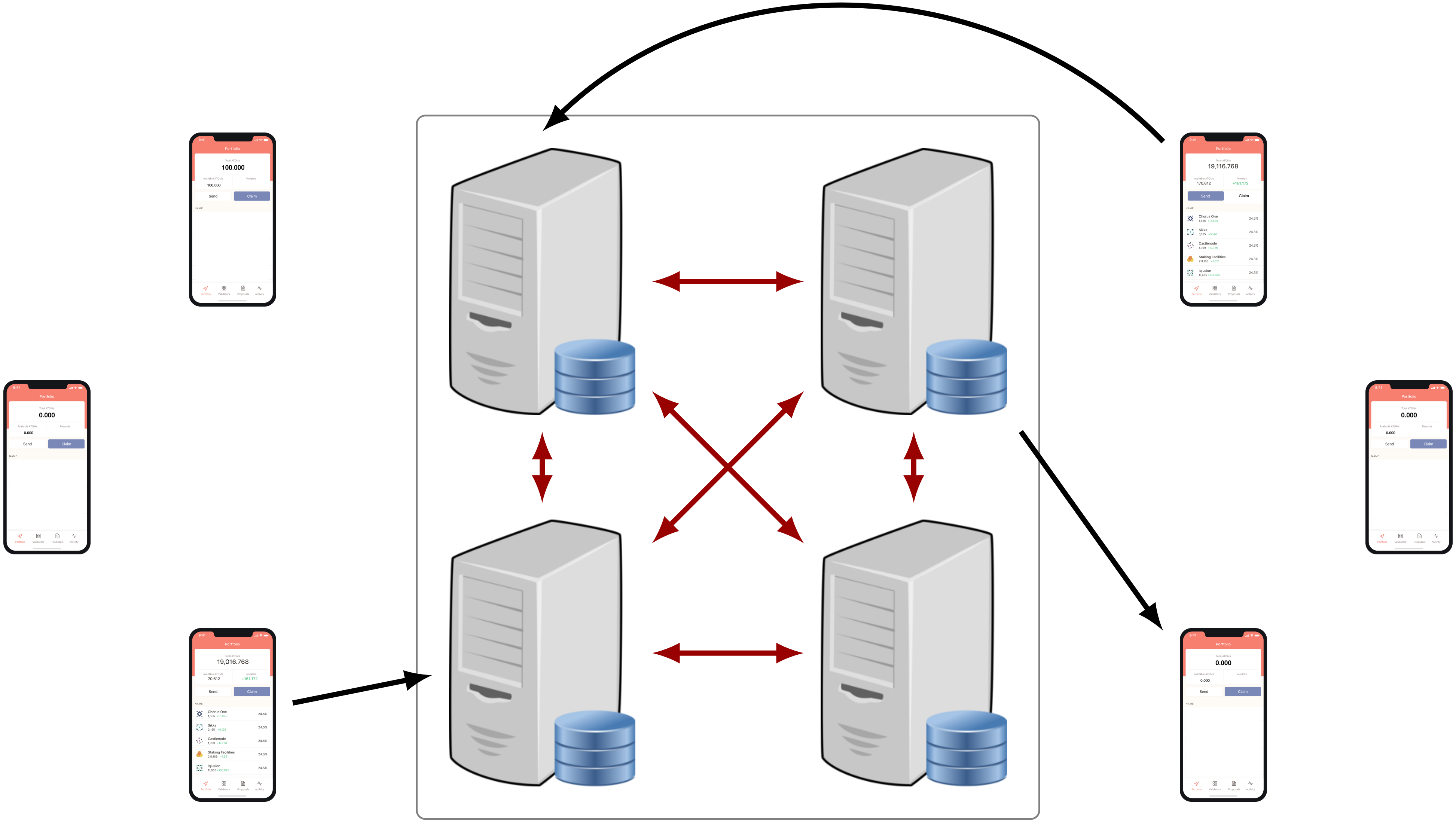
Central server



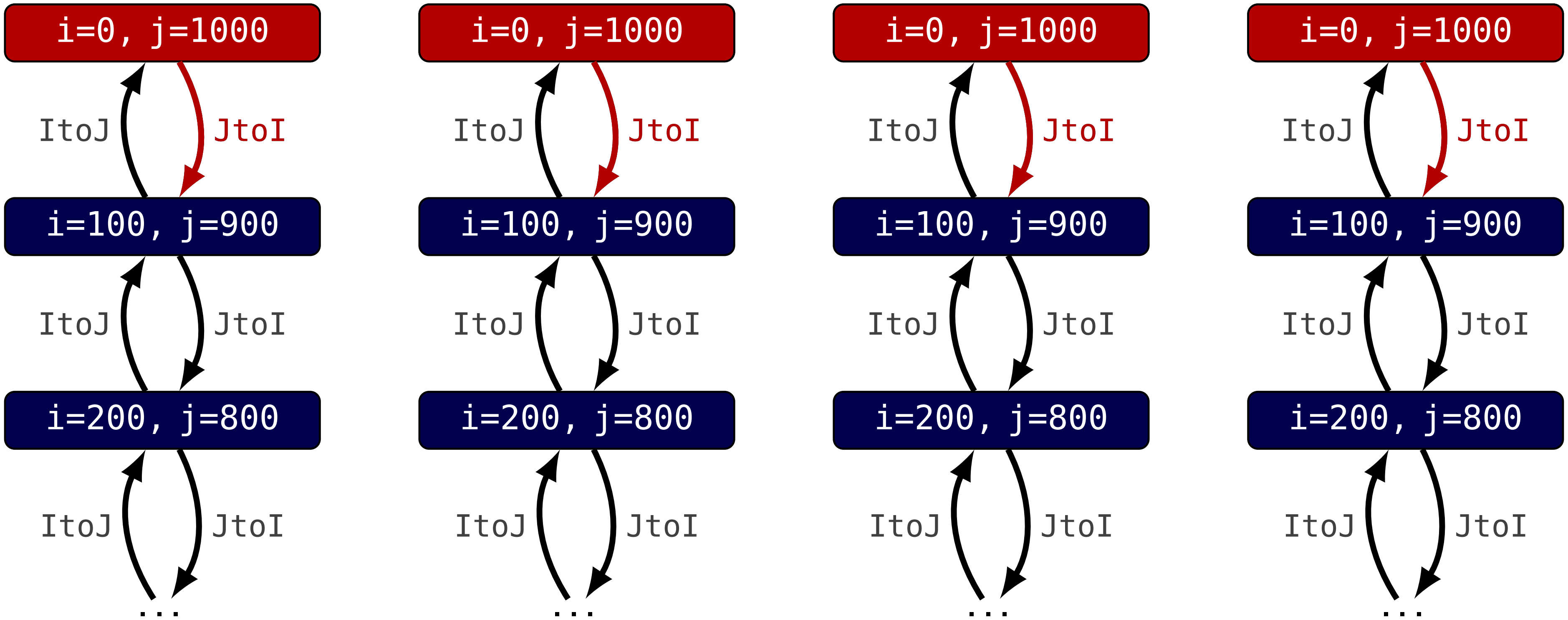
Central server



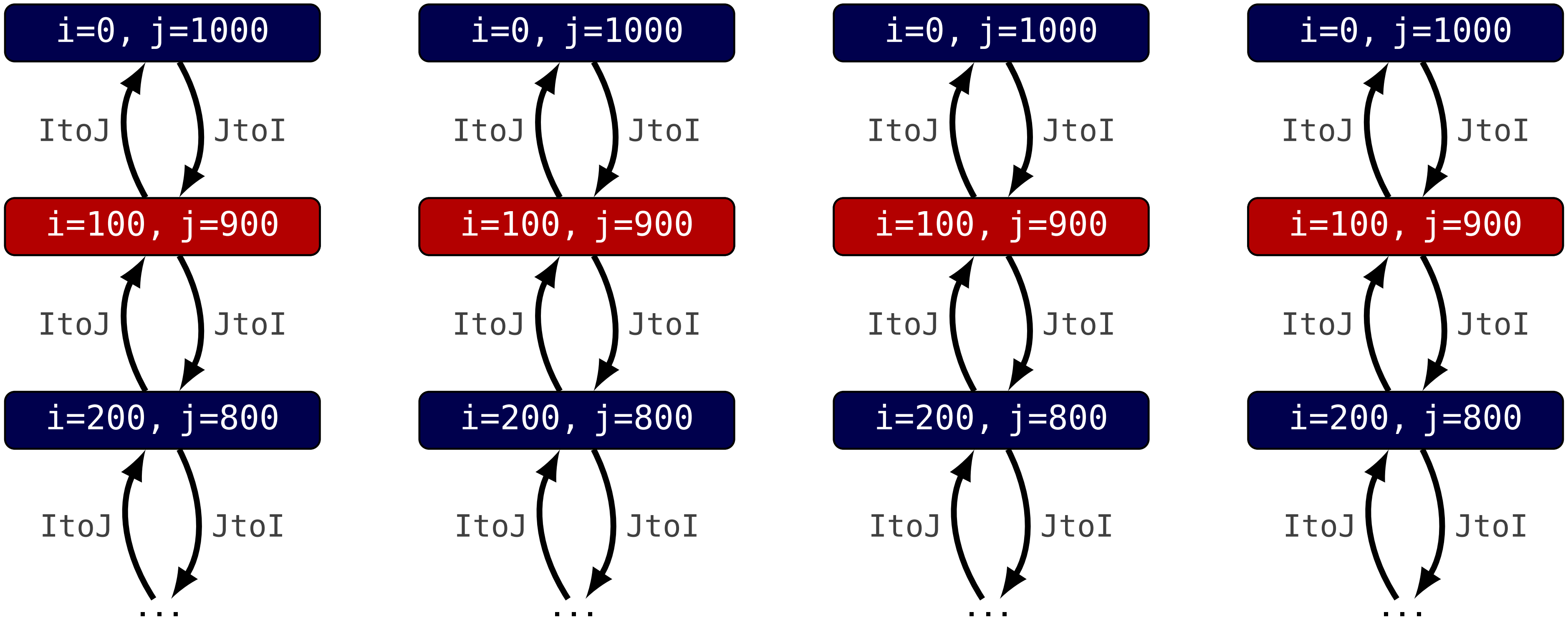
Replication is the solution



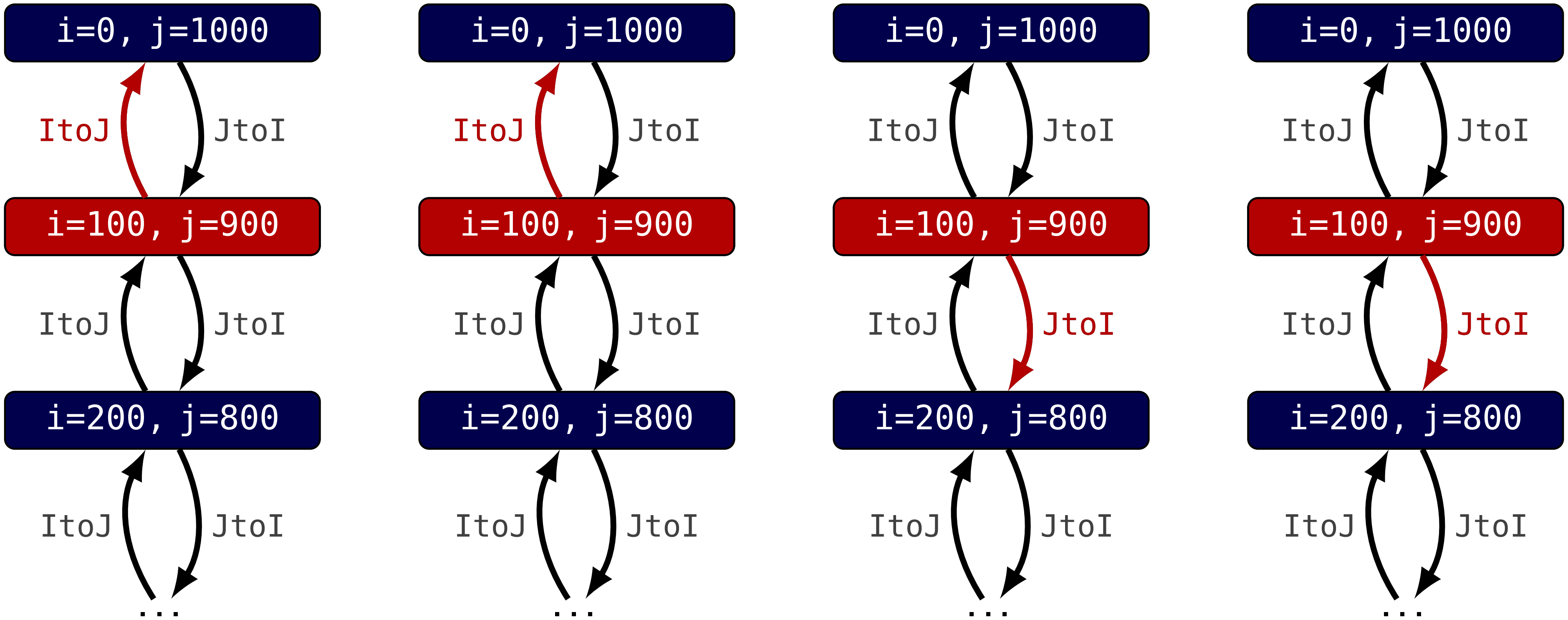
Replicated state machine



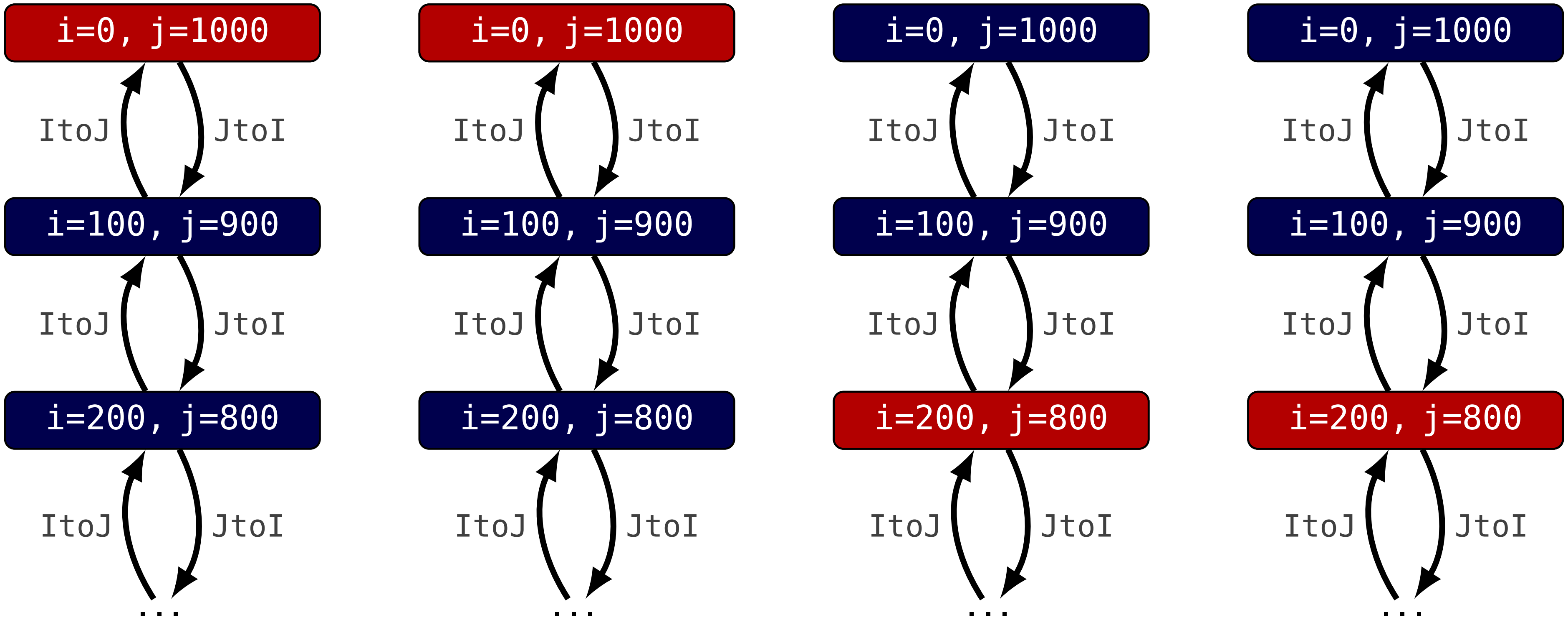
Replicated state machine



Replicated state machine



Replicated state machine



How to coordinate them?

Two-phase commit

Transaction manager:

```
send <INIT, txid> to ALL
ncommits = 0
while ncommits < N {
 on <ABORT> from i {
   send <ABORT> to ALL;
   break
 }

 on <COMMIT> from i
   ncommits++

  if ncommits == N
   send <COMMIT> to ALL
}
```

Replica *i* of *N*:

```
on <INIT, txid> from mgr {
 begin_tx(txid)
 /* processing... */
 if error()
 send <ABORT> to mgr
 else send <COMMIT> to mgr

 receive m from mgr

 if m == <ABORT>
  abort_tx(txid)
 else
  commit_tx(txid)
}
```

if there are crashes?

Two-phase commit

Transaction manager:

```
send <INIT, txid> to ALL
ncommits = 0
while ncommits < N {
 on <ABORT> from i {
   send <ABORT> to ALL;
   break
 }

 on <COMMIT> from i
   ncommits++

  if ncommits == N
   send <COMMIT> to ALL
}
```

Replica *i* of *N*:

```
on <INIT, txid> from mgr {
 begin_tx(txid)
 /* processing... */
 if error()
 send <ABORT> to mgr
 else send <COMMIT> to mgr

 receive m from mgr

 if m == <ABORT>
  abort_tx(txid)
 else
  commit_tx(txid)
}
```

if there are crashes? 🔥

Distributed consensus

Idea of consensus

A distributed algorithm for N replicas

every replica proposes a value $w \in V$

Termination

every correct replica eventually decides on a value $v \in V$

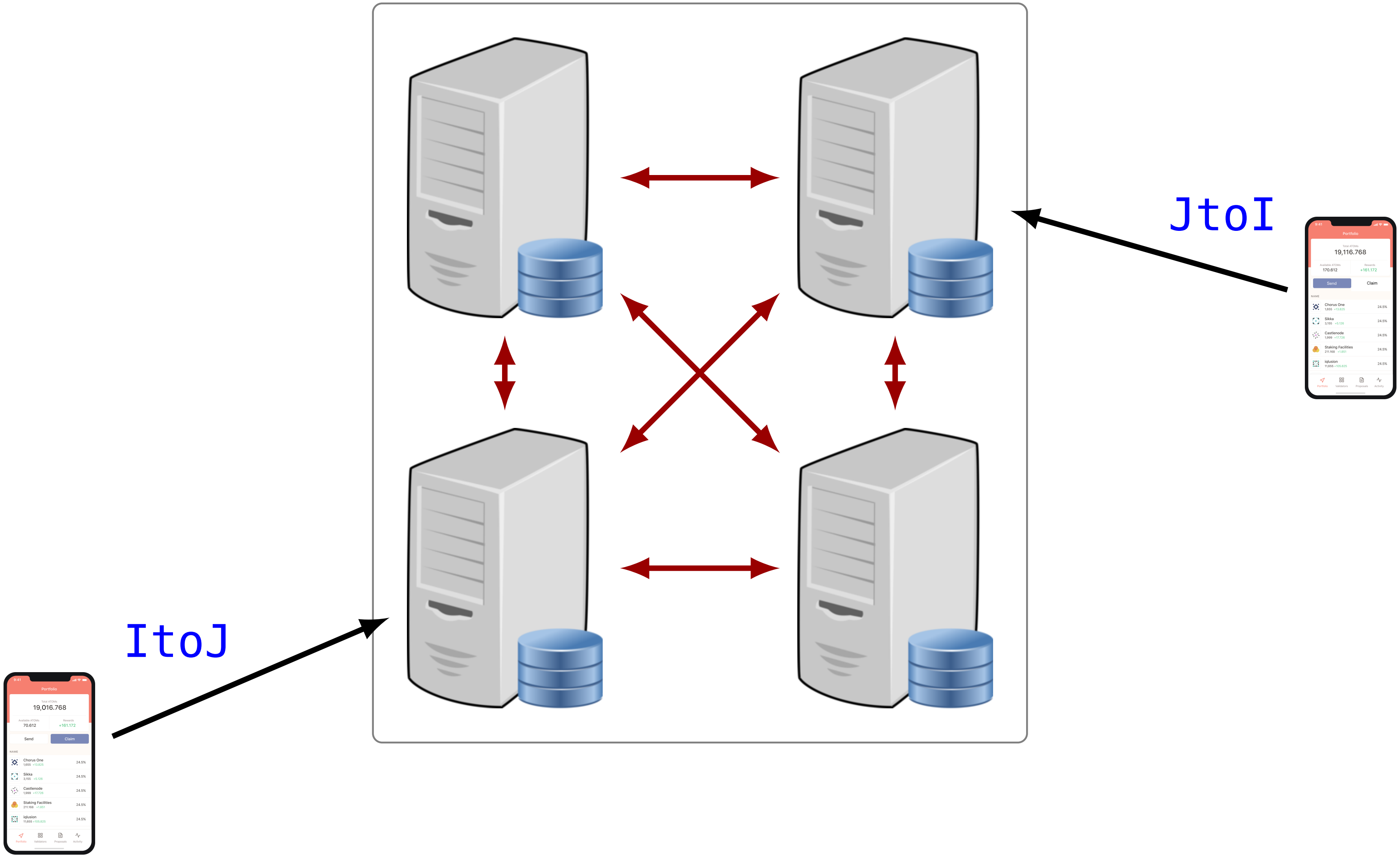
Agreement

if a replica decides on v, no replica decides on $V \setminus \{v\}$

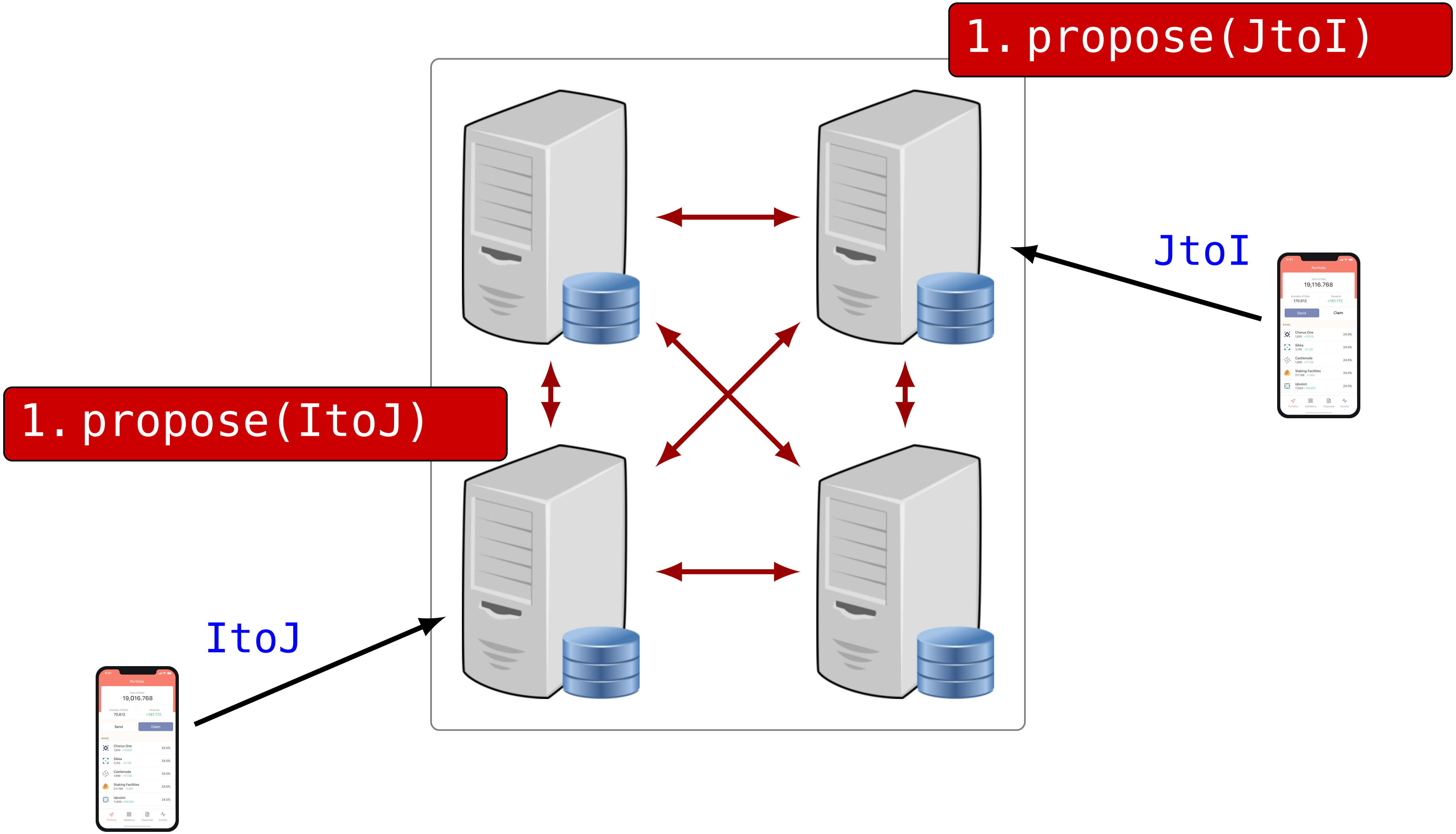
Validity

if a replica decides on v, the value v was proposed earlier

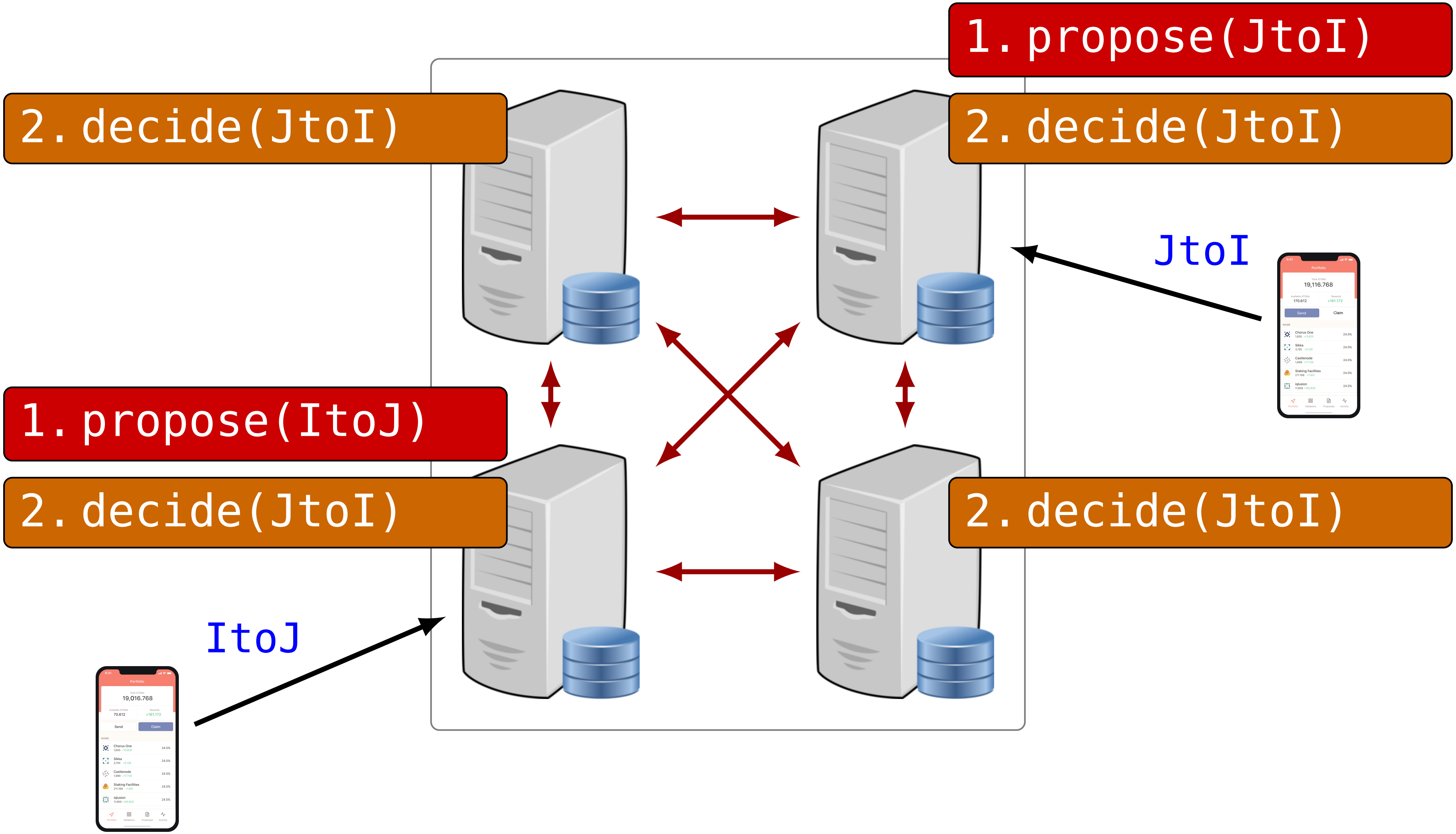
How is consensus useful?



How is consensus useful?



How is consensus useful?



Blockchain with classical consensus

Block 1	Block 2	Block 3	Block 4	...
`ItoJ`	`JtoI`	`Coffee`	`Tea`	...

In practice, multiple user transactions are packed together

Consensus decides on block hashes

Let’s write some algorithms

~~Termination~~

every replica eventually decides on a value $v \in V$

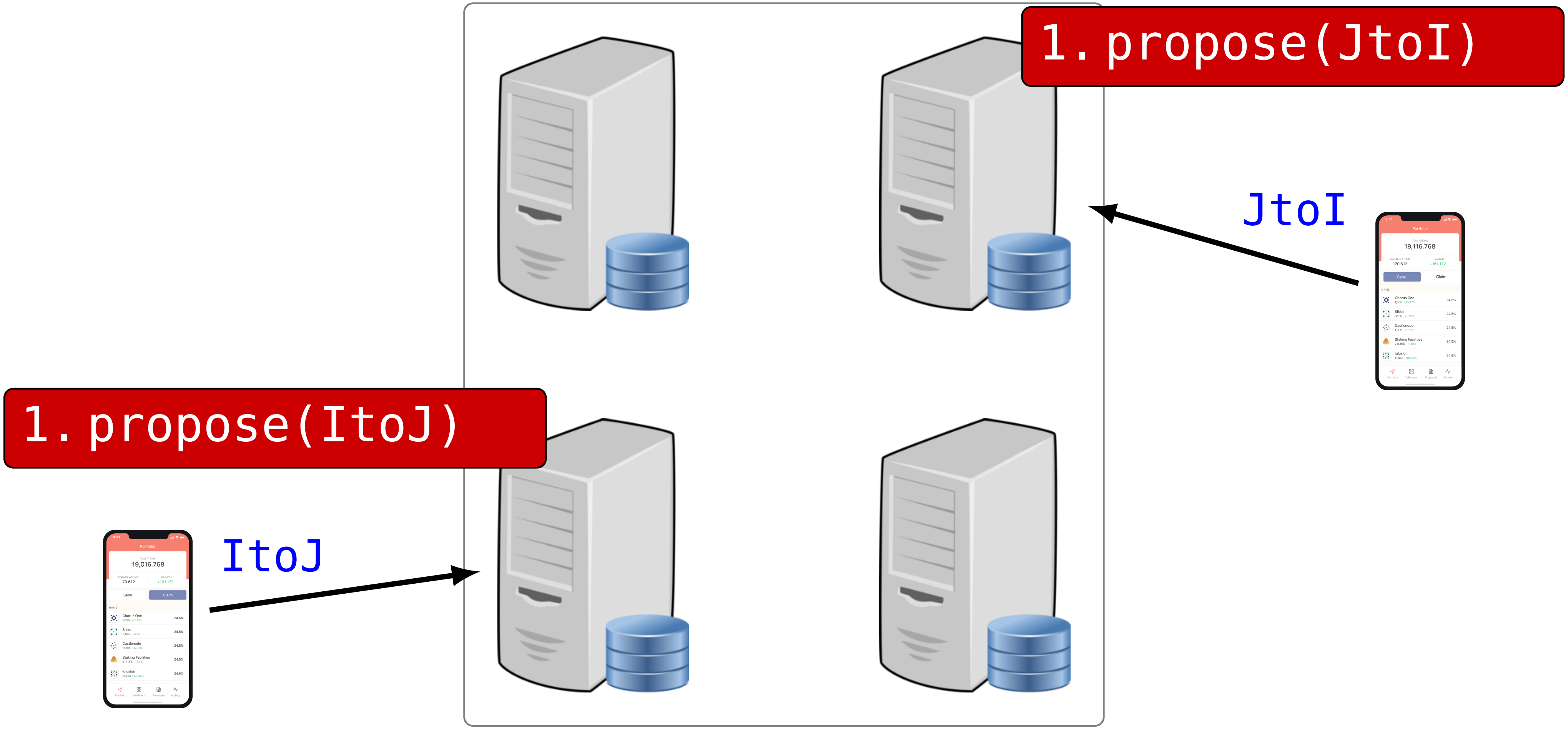
Agreement

if a replica decides on v, no replica decides on $V \setminus \{v\}$

Validity

if a replica decides on v, the value v was proposed earlier

The algorithm: do nothing!



Termination

every replica eventually decides on a value $v \in V$

~~**Agreement**~~

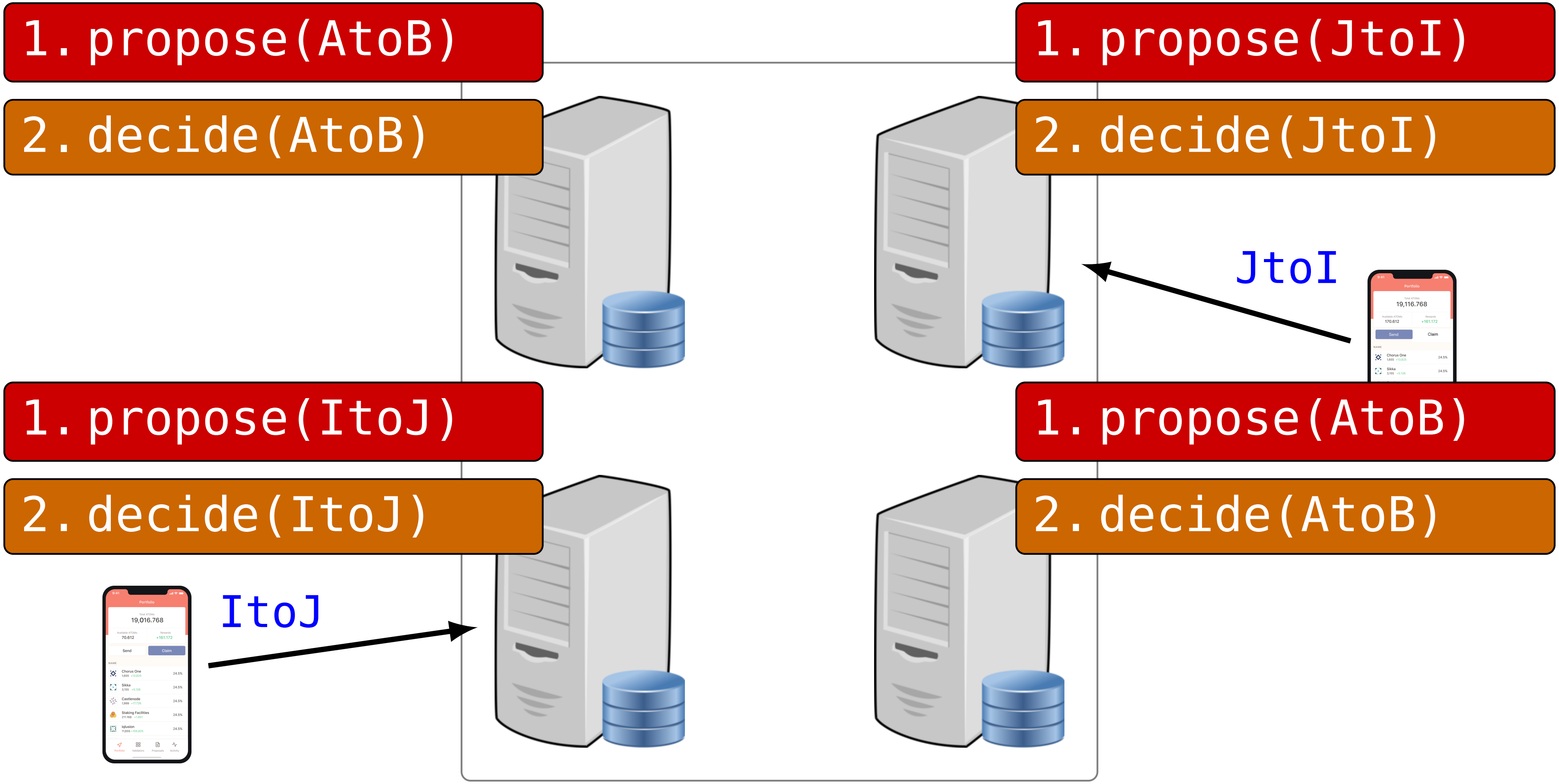
if a replica decides on v, no replica decides on $V \setminus \{v\}$

Validity

if a replica decides on v, the value v was proposed earlier

Consensus without agreement

The algorithm: decide on own value!



Termination

every replica eventually decides on a value $v \in V$

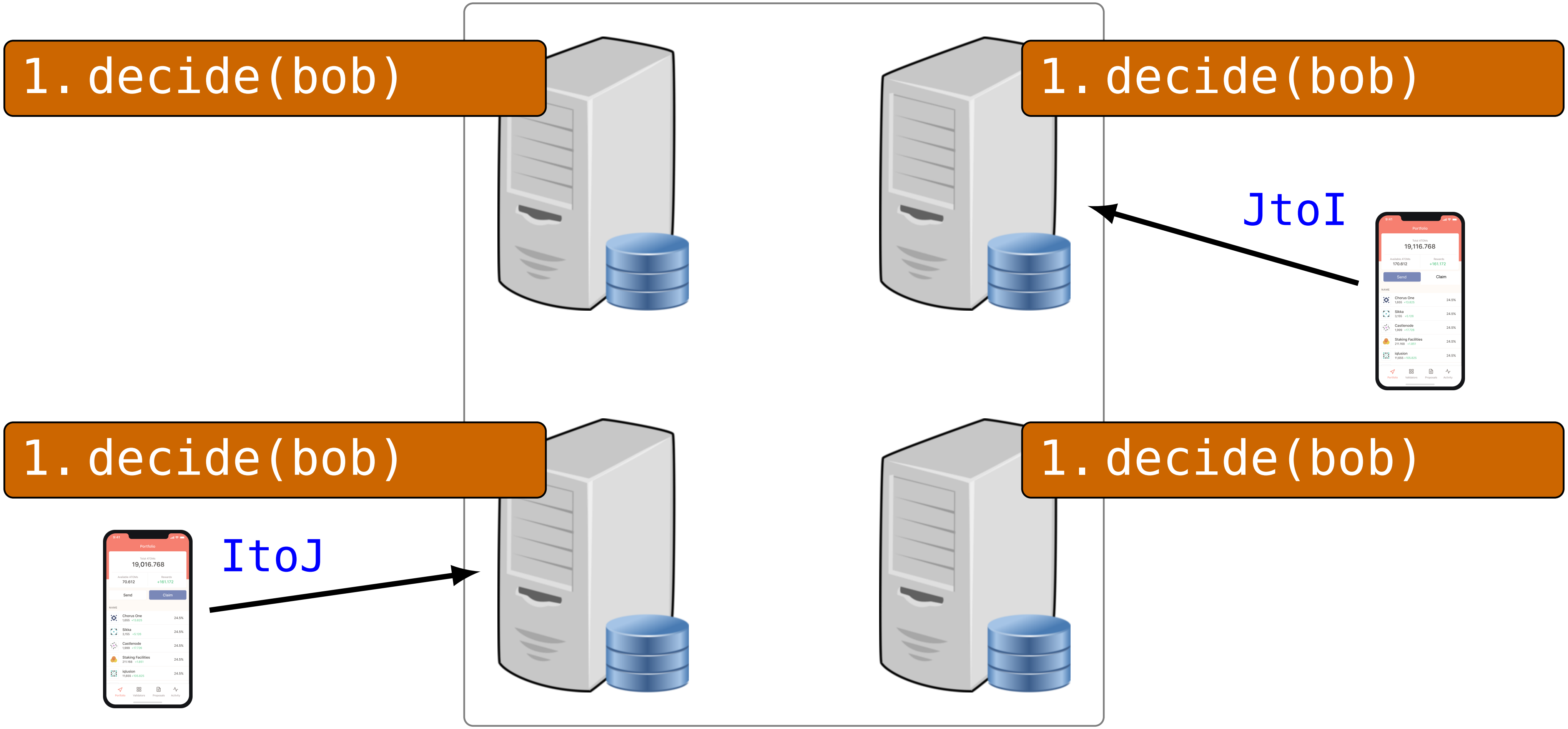
Agreement

if a replica decides on v, no replica decides on $V \setminus \{v\}$

~~**Validity**~~

if a replica decides on v, the value v was proposed earlier

The algorithm: decide on a fixed value!



Termination

every replica eventually decides on a value $v \in V$

Agreement

if a replica decides on v, no replica decides on $V \setminus \{v\}$

Validity

if a replica decides on v, the value v was proposed earlier

is there an algorithm?

Synchronous distributed consensus

Synchronous rounds

a) send post on Monday, receive post on Thursday, and
compute on Friday

b) DHL delivers the post in 48 hours

	Round 1	**Round 2**	...
Replica 1:	send/receive/compute	send/receive/compute	...
Replica 2:	send/receive/compute	send/receive/compute	...
Replica 3:	send/receive/compute	send/receive/compute	...
Replica 4:	send/receive/compute	send/receive/compute	...

a) in every round, a replica executes send/receive/compute

b) every message sent in round k is received in round k

Synchronous rounds

a) send post on Monday, receive post on Thursday, and compute on Friday

b) DHL delivers the post in 48 hours

	Round 1	Round 2	...
Replica 1:	send/receive/compute	send/receive/compute	...
Replica 2:	send/receive/compute	send/receive/compute	...
Replica 3:	send/receive/compute	send/receive/compute	...
Replica 4:	send/receive/compute	send/receive/compute	...

a) in every round, a replica executes send/receive/compute

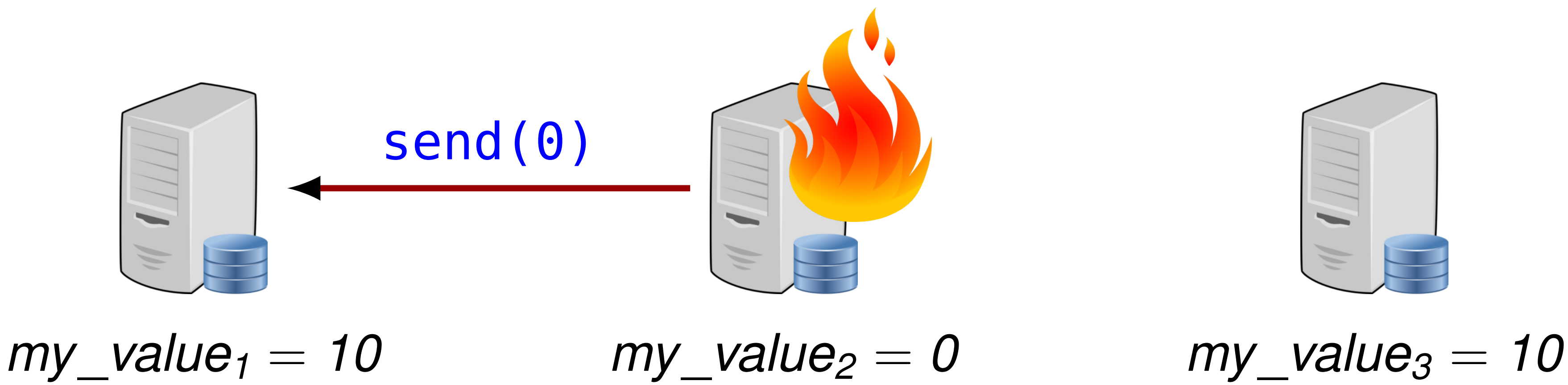
b) every message sent in round k is received in round k

Naïve algorithm

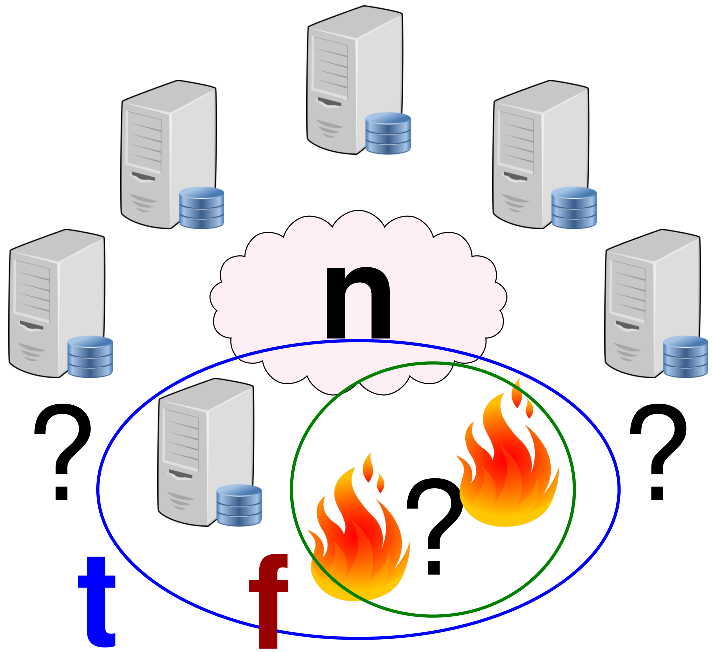
$round_1$:
  send $\{my\_value_i\}$ **to** ALL
  receive S_j **from** r_j: $1 \leq j \leq N$
  V_i := $\bigcup_{1 \leq j \leq N} S_j$
  decide(min(V_i))

Naïve algorithm

```
round_1 :
    send {my_value_i} to ALL
    receive S_j from r_j: 1 ≤ j ≤ N
    V_i := ⋃_{1≤j≤N} S_j
    decide(min(V_i))
```



Assumptions about faults



f replicas crash (unknown)

t $<$ ***n*** is an upper bound on ***f*** (known)

FloodMin [Chaudhuri, Herlihy, Lynch, Tuttle, JACM 2000]

Every replica r_i for $i \in \{1, \ldots, N\}$ executes the algorithm:

```
init:
    best_i := my_value_i

round_k: 1 ≤ k ≤ t + 1
    send best_i to ALL
    receive b_j from r_j: 1 ≤ j ≤ N
    best_i := min {b_1, ..., b_N}
    if k = t + 1 then decide(best_i)
```

FloodMin [Chaudhuri, Herlihy, Lynch, Tuttle, JACM 2000]

Every replica r_i for $i \in \{1, \ldots, N\}$ executes the algorithm:

```
init:
    best_i := my_value_i

round_k: 1 ≤ k ≤ t + 1
    send best_i to ALL
    receive b_j from r_j: 1 ≤ j ≤ N
    best_i := min {b_1, ..., b_N}
    if k = t + 1 then decide(best_i)
```

Termination ✔

FloodMin [Chaudhuri, Herlihy, Lynch, Tuttle, JACM 2000]

Every replica r_i for $i \in \{1, \ldots, N\}$ executes the algorithm:

```
init:
    best_i := my_value_i

round_k: 1 ≤ k ≤ t + 1
    send best_i to ALL
    receive b_j from r_j: 1 ≤ j ≤ N
    best_i := min {b_1, ..., b_N}
    if k = t + 1 then decide(best_i)
```

Termination ✓ **Validity** ✓

$$best_i \in \bigcup_{1 \leq j \leq N} \{my\_value_j\}$$

FloodMin [Chaudhuri, Herlihy, Lynch, Tuttle, JACM 2000]

Every replica r_i for $i \in \{1, \ldots, N\}$ executes the algorithm:

```
init:
    best_i := my_value_i

round_k:  1 ≤ k ≤ t + 1
    send best_i to ALL
    receive b_j from r_j:  1 ≤ j ≤ N
    best_i := min {b_1, ..., b_N}
    if k = t + 1 then decide(best_i)
```

Termination ✓ **Validity** ✓ **Agreement** ?

$$best_i \in \bigcup_{1 \le j \le N} \{my\_value_j\}$$

Proving agreement (pencil & paper)

```
4  round_k: 1 ≤ k ≤ t + 1
5      send best_i to ALL
6      receive b_j from r_j: 1 ≤ j ≤ N
7      best_i := min {b_1, ..., b_N}
8      if k = t + 1 then decide(best_i)
```

Assume **agreement** is violated:

- *Two replicas r_i and r_j call decide(v_i) and decide(v_j) in line 8*
- *assume $v_i < v_j$*
- *r_j never received v_i in line 6*
- *by assumption, there are most t crashes*
- *hence, no crashes happen in some round $m \leq t + 1$*
- *each replica receives $best_1, \ldots, best_N$ in round m (lines 5–7)*
- *hence, if r_i received v_i, then r_j received v_i in round m*

Proving agreement (pencil & paper)

```
4  round_k: 1 ≤ k ≤ t + 1
5     send best_i to ALL
6     receive b_j from r_j: 1 ≤ j ≤ N
7     best_i := min {b_1, ..., b_N}
8     if k = t + 1 then decide(best_i)
```

Assume **agreement** is violated:

- *Two replicas r_i and r_j call decide(v_i) and decide(v_j) in line 8*
- *assume $v_i < v_j$*
- *r_j never received v_i in line 6*
- *by assumption, there are most t crashes*
- *hence, no crashes happen in some round $m \leq t + 1$*
- *each replica receives $best_1, \ldots, best_N$ in round m (lines 5–7)*
- *hence, if r_i received v_i, then r_j received v_i in round m* ✓

fewer constraints?

Asynchronous systems

r_1 sends/receives on Monday/Thursday, computes on Friday

r_2 sends/receives/computes once a month

r_3 went for a two-month vacation

r_4 left job without notice

r_1 uses DHL, r_2 uses LA POSTE, r_3 uses Post

Consensus in asynchronous systems

Various processor speeds

Various message delays, unbounded but finite

Consensus is not solvable [Fischer, Lynch, Paterson, 1985]

Practical consensus algorithms:

- termination is the engineering problem, **Paxos**
- or restrict asynchrony, **DLS88, Tendermint**
- or prove almost-sure termination **Ben-Or**

Consensus in asynchronous systems

Various processor speeds

Various message delays, unbounded but finite

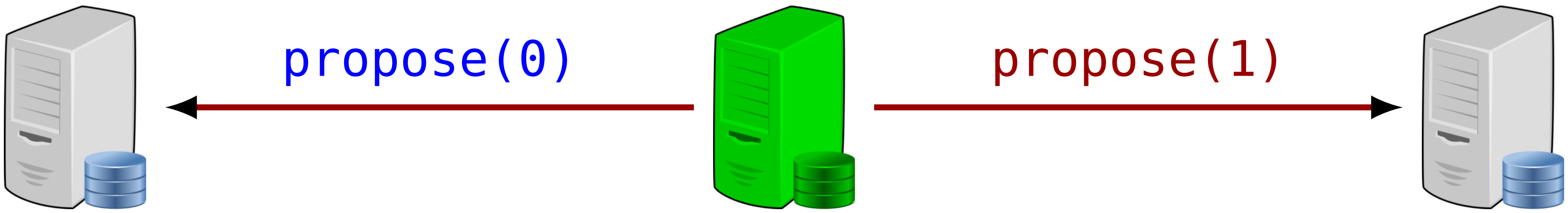
Consensus is not solvable [Fischer, Lynch, Paterson, 1985]

Practical consensus algorithms:

- termination is the engineering problem, **Paxos**
- or restrict asynchrony, **DLS88, Tendermint**
- or prove almost-sure termination **Ben-Or**

Beyond crashes

What if some replicas lie?



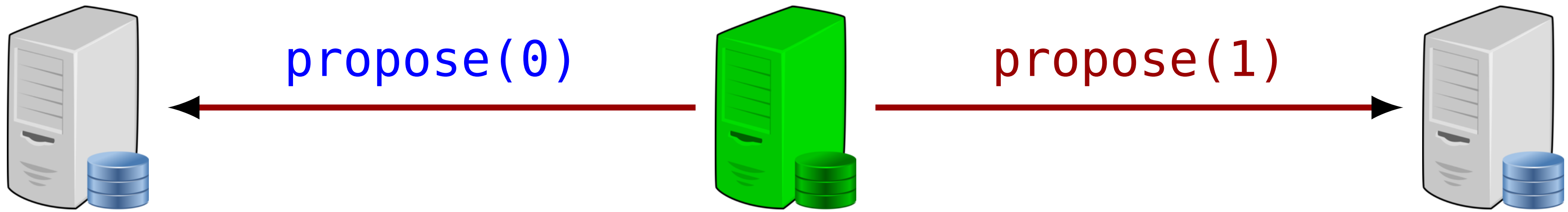
This is **Byzantine** behavior [Lamport, Shostak, Pease, 1982]

More than two-thirds must be correct: $n > 3t$

e.g., Tendermint

Beyond crashes

What if some replicas lie?



This is **Byzantine** behavior [Lamport, Shostak, Pease, 1982]

More than two-thirds must be correct: $n > 3t$

e.g., Tendermint

Conclusions for Part I

Distributed consensus provides fault tolerance

Interaction of multiple peers, fraction of them faulty

Various assumptions about computations

Are the fault-tolerant algorithms bug-free?

Model checking of distributed algorithms:

from classics towards Tendermint blockchain

part II

Igor Konnov

VMCAI winter school, January 16-18, 2020

informal

INTERCHAIN FOUNDATION

Timeline

Introduction to **fault-tolerant** distributed algorithms

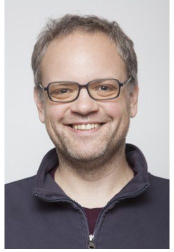
Verifying **synchronous** threshold-guarded algorithms

Verifying **asynchronous** threshold-guarded algorithms

Can we verify **Tendermint consensus?**

Verifying **synchronous** threshold-guarded distributed algorithms

[Stoilkovska, K., Widder, Zuleger. TACAS 2019]

Recall FloodMin:

init **:**
  $best_i := my\_value_i$

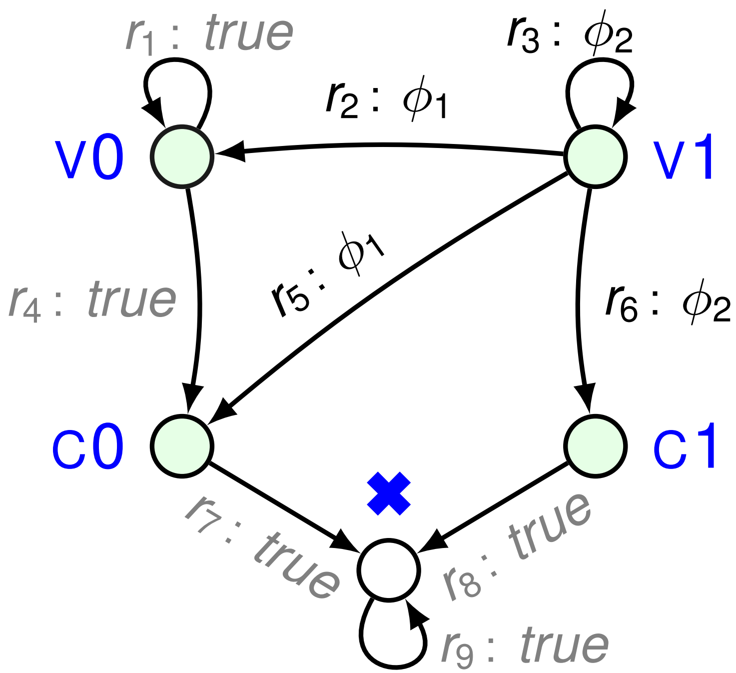
round$_k$ **:** $1 \leq k \leq t+1$
  send $best_i$ **to** `ALL`
  receive b_j **from** r_j**:** $1 \leq j \leq N$
  $best_i$ `:=` **`min`** $\{b_1, \ldots, b_N\}$
  if `k = t + 1` **then** `decide(`$best_i$`)`

Formalizing pseudo-code with threshold automata

Recall FloodMin:

init **:**
  $best_i := my\_value_i$

$round_k$ **:** $1 \leq k \leq t+1$
  send $best_i$ **to** `ALL`
  receive b_j **from** r_j**:** $1 \leq j \leq N$
  $best_i$ `:=` **`min`** $\{b_1, \ldots, b_N\}$
  `if` `k = t + 1` **then** `decide(`$best_i$`)`



$\phi_1 \equiv \#\{\text{v0}, \text{c0}\} > 0$

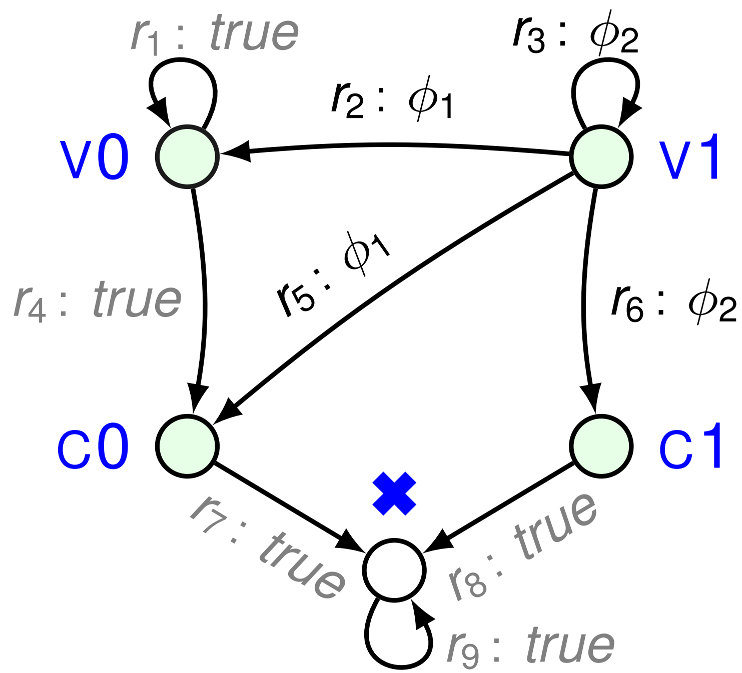
$\phi_2 \equiv \#\{\text{v0}\} = 0$

Formalizing pseudo-code with threshold automata

Recall FloodMin:

init **:**
  $best_i := my\_value_i$

$round_k$ **:** $1 \le k \le t+1$
  send $best_i$ **to** `ALL`
  receive b_j **from** r_j **:** $1 \le j \le N$
  $best_i$ `:=` **min** $\{b_1, \dots, b_N\}$
  if `k = t + 1` **then** `decide(`$best_i$`)`

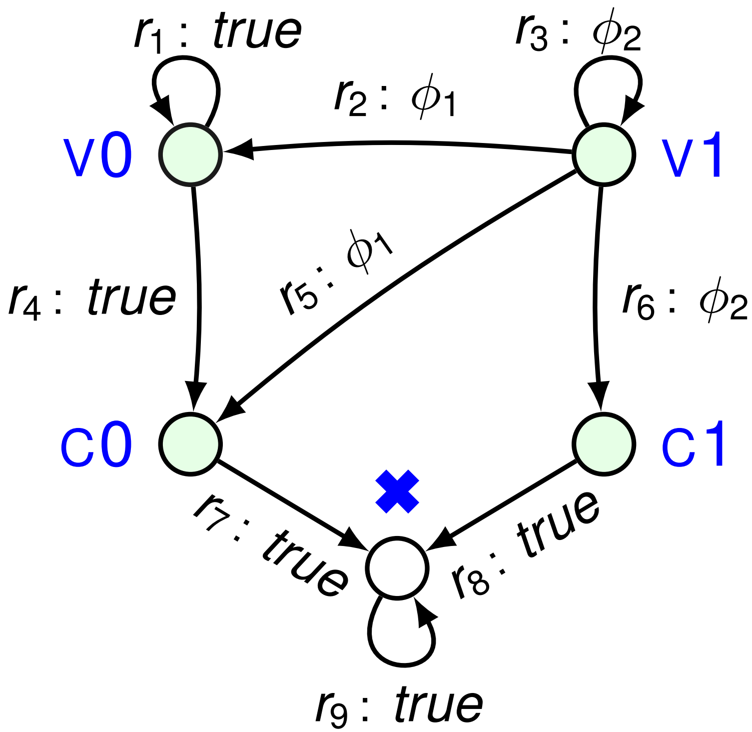


$\{v0, c0\}$ send 0
$\{v1, c1\}$ send 1

$\phi_1 \equiv \#$

$\phi_2 \equiv \#\{v0\} = 0$

Semantics of synchronous threshold automata

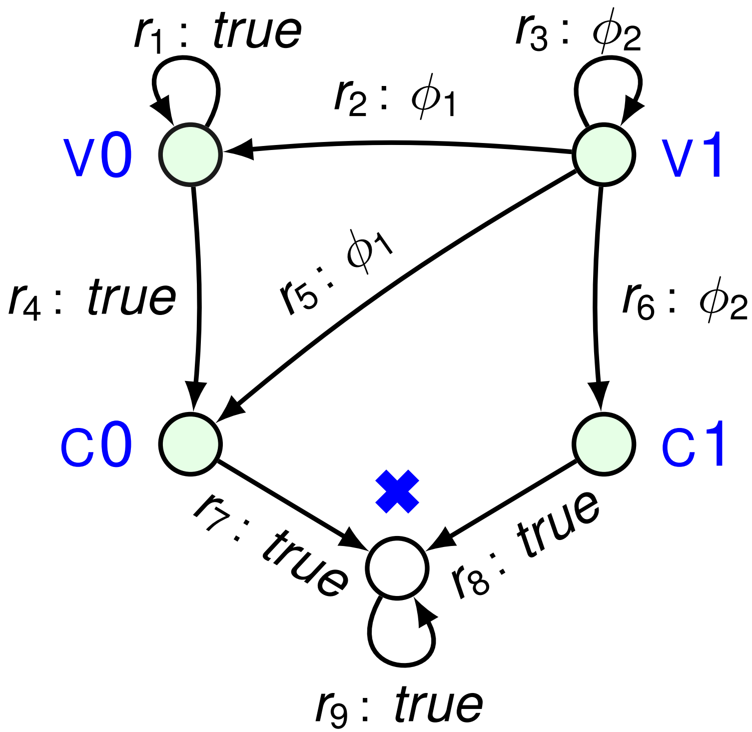


ϕ_1 is $\#\{\text{v0}, \text{c0}\} > 0$

ϕ_2 is $\#\{\text{v0}\} = 0$

Counter system: (Σ, I, T)

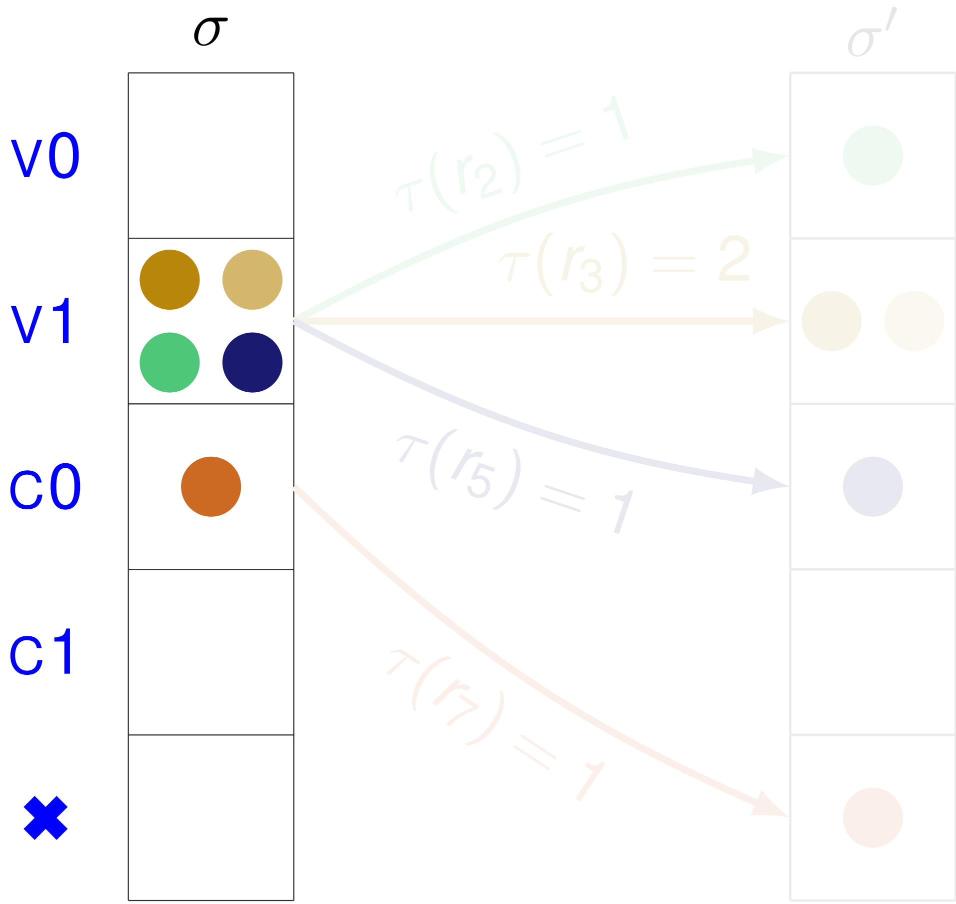
Semantics of synchronous threshold automata



ϕ_1 is $\#\{\text{v0}, \text{c0}\} > 0$

ϕ_2 is $\#\{\text{v0}\} = 0$

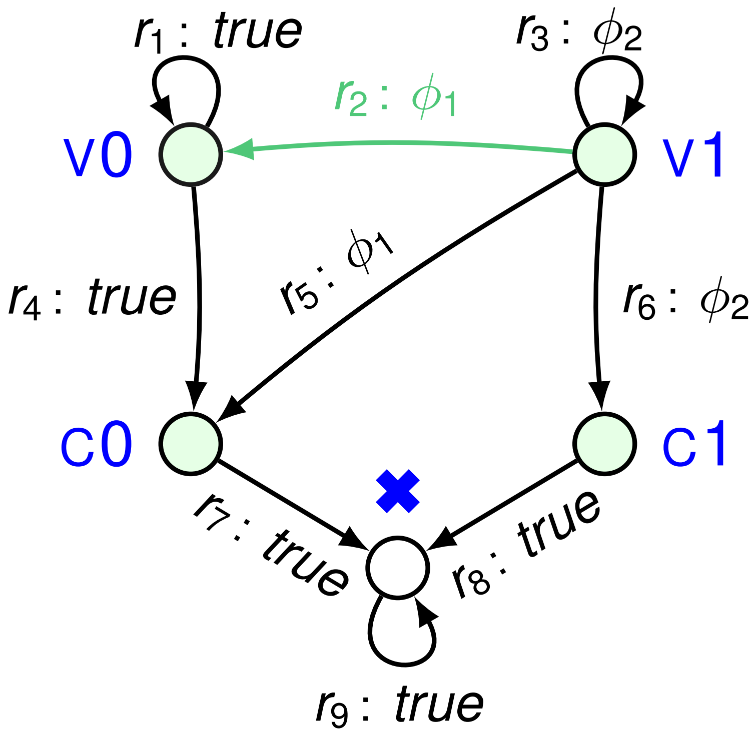
$n = 5, \quad t = 2, \quad f = 2$

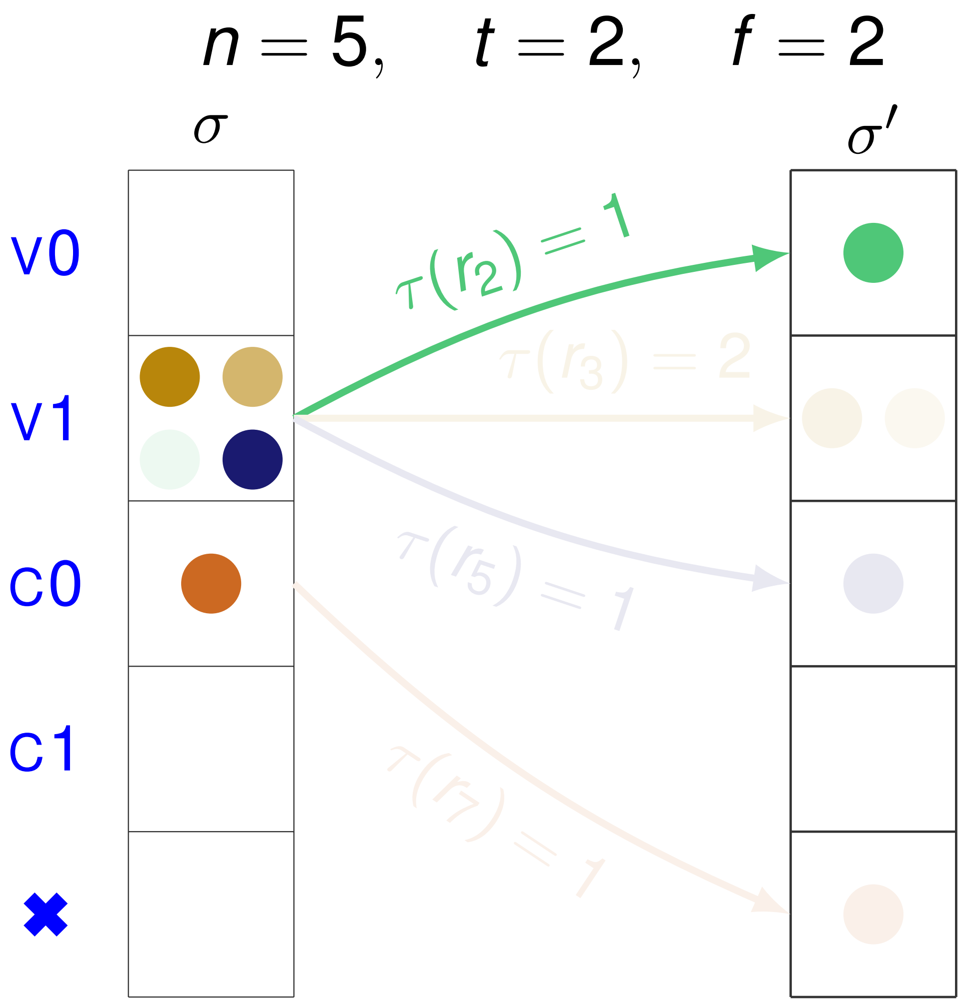


$\tau(r_1) + \cdots + \tau(r_9) = n$

Counter system: (Σ, I, T)

Semantics of synchronous threshold automata



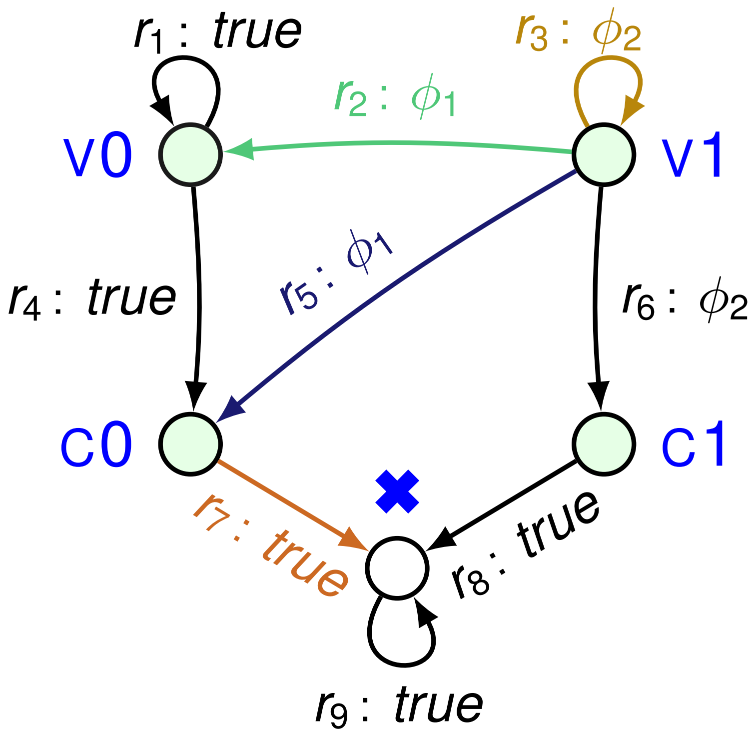


ϕ_1 is $\#\{\text{v0}, \text{c0}\} > 0$
ϕ_2 is $\#\{\text{v0}\} = 0$

$\tau(r_1) + \cdots + \tau(r_9) = n$

Counter system: (Σ, I, T)

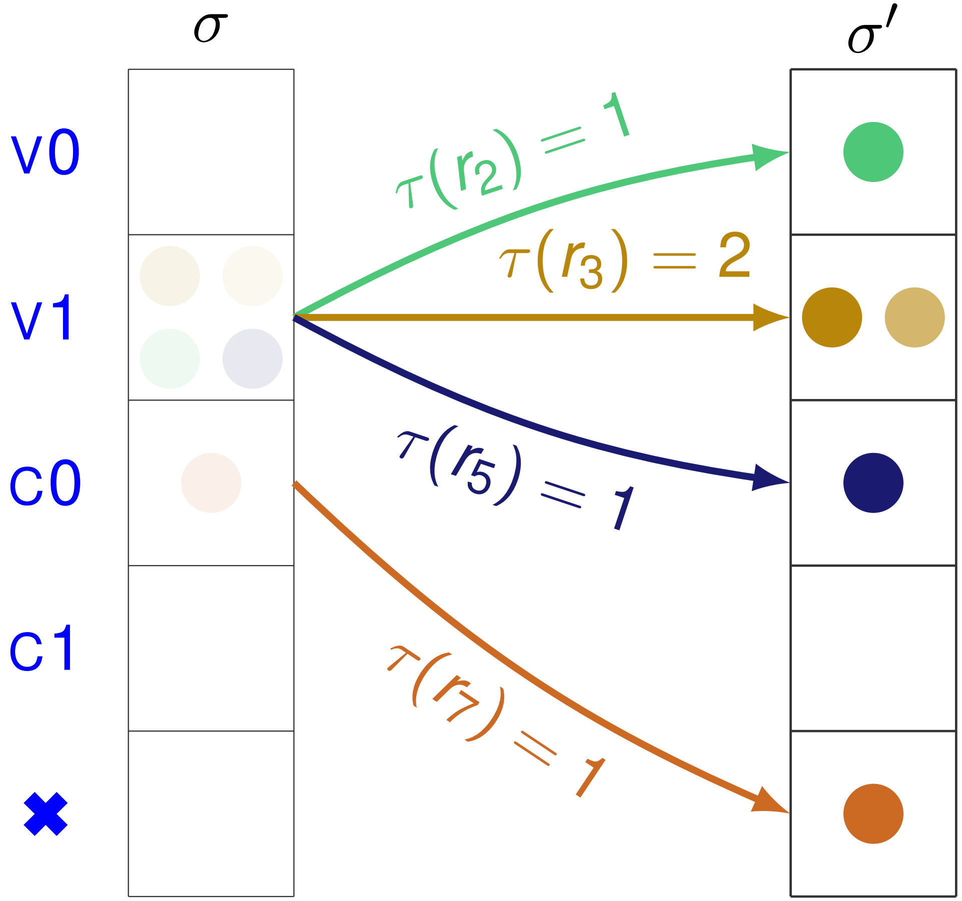
Semantics of synchronous threshold automata



ϕ_1 is $\#\{v0, c0\} > 0$
ϕ_2 is $\#\{v0\} = 0$

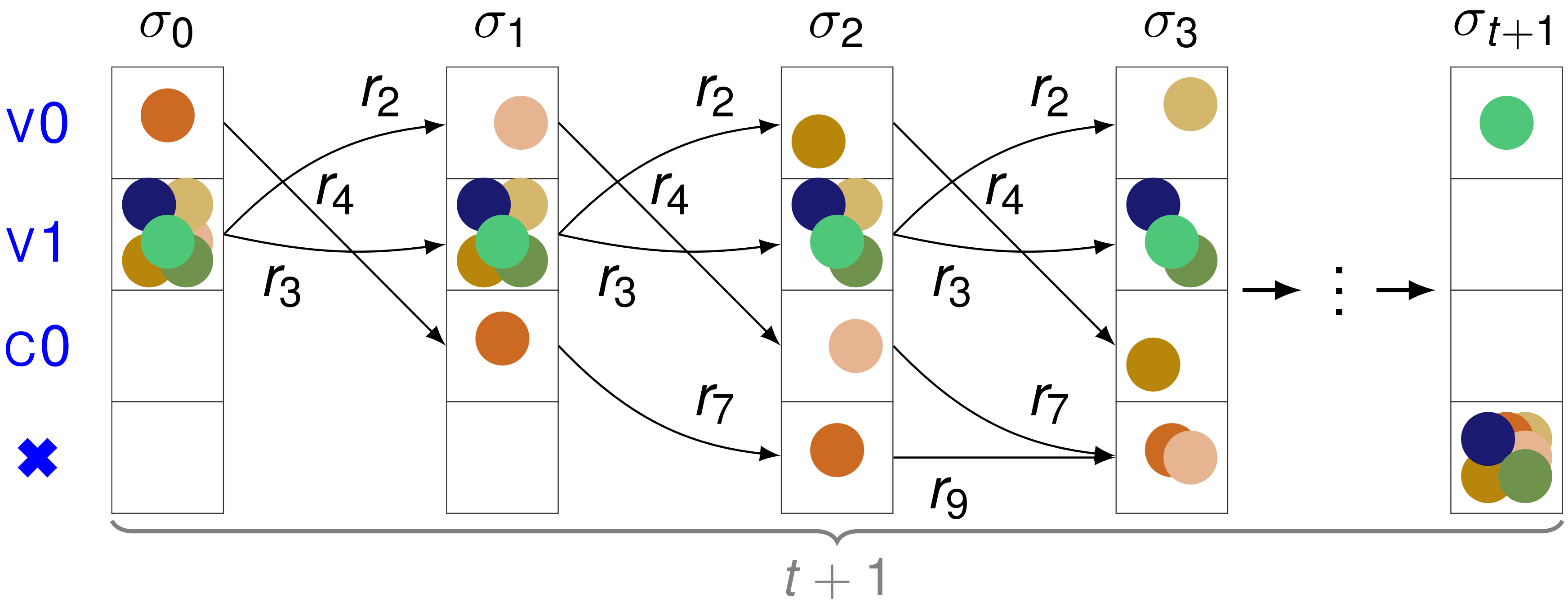
Counter system: (Σ, I, T)

$n = 5, \quad t = 2, \quad f = 2$



$\tau(r_1) + \cdots + \tau(r_9) = n$

An execution of the counter system

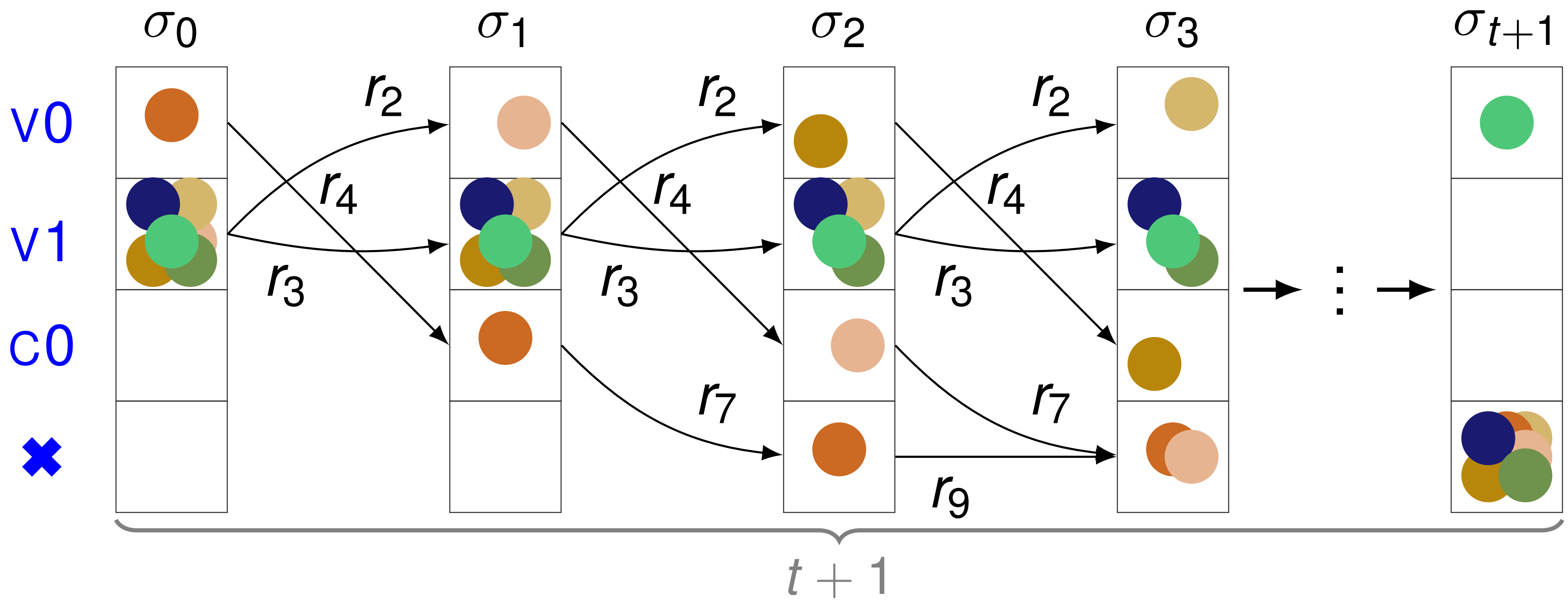


A configuration is a tuple of counters $\kappa_{V0}, \kappa_{V1}, \kappa_{SE}, \kappa_{AC}$

An execution is a sequence of configurations

(related by transitions)

An execution of the counter system



A configuration is a tuple of counters $\boldsymbol{\kappa}_{\mathrm{V0}}, \boldsymbol{\kappa}_{\mathrm{V1}}, \boldsymbol{\kappa}_{\mathrm{SE}}, \boldsymbol{\kappa}_{\mathrm{AC}}$

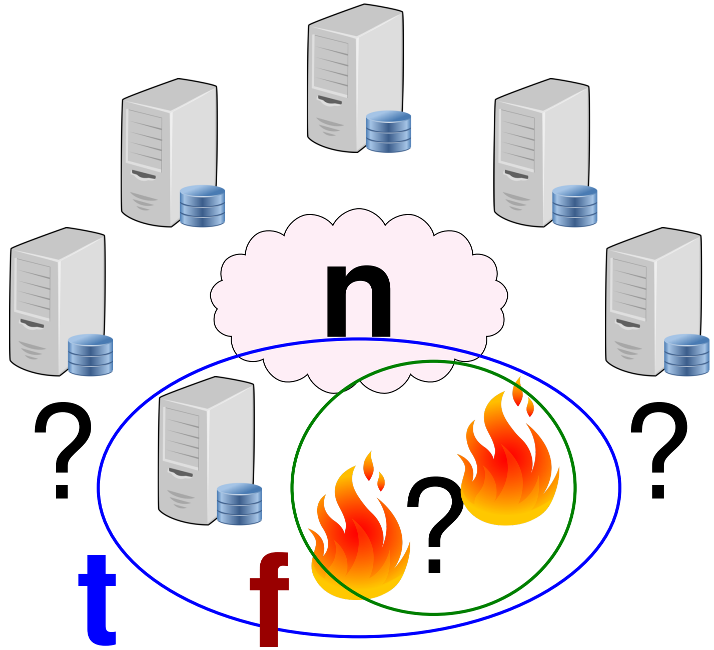
An execution is a sequence of configurations

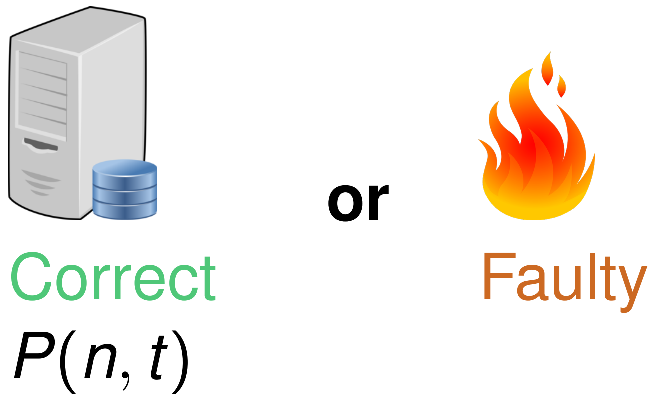
(related by transitions)

Can we verify safety?

e.g., agreement

Parameterized model checking





$\forall n, t, f$ satisfying the resilience condition (e.g., $n > t$)

$$\underbrace{P(n,t) \parallel P(n,t) \parallel \ldots \parallel P(n,t)}_{n-f \text{ correct}} \parallel \underbrace{\text{Faulty} \parallel \ldots \parallel \text{Faulty}}_{f \text{ faulty}} \models \varphi$$

Parameterized reachability

Input:

- synchronous threshold automaton TA
- Boolean formula ϕ over counter equalities $\sum_{\ell \in L} \boldsymbol{\kappa}[\ell] \geq \mathbf{a} \cdot \mathbf{p} + \mathbf{b}$

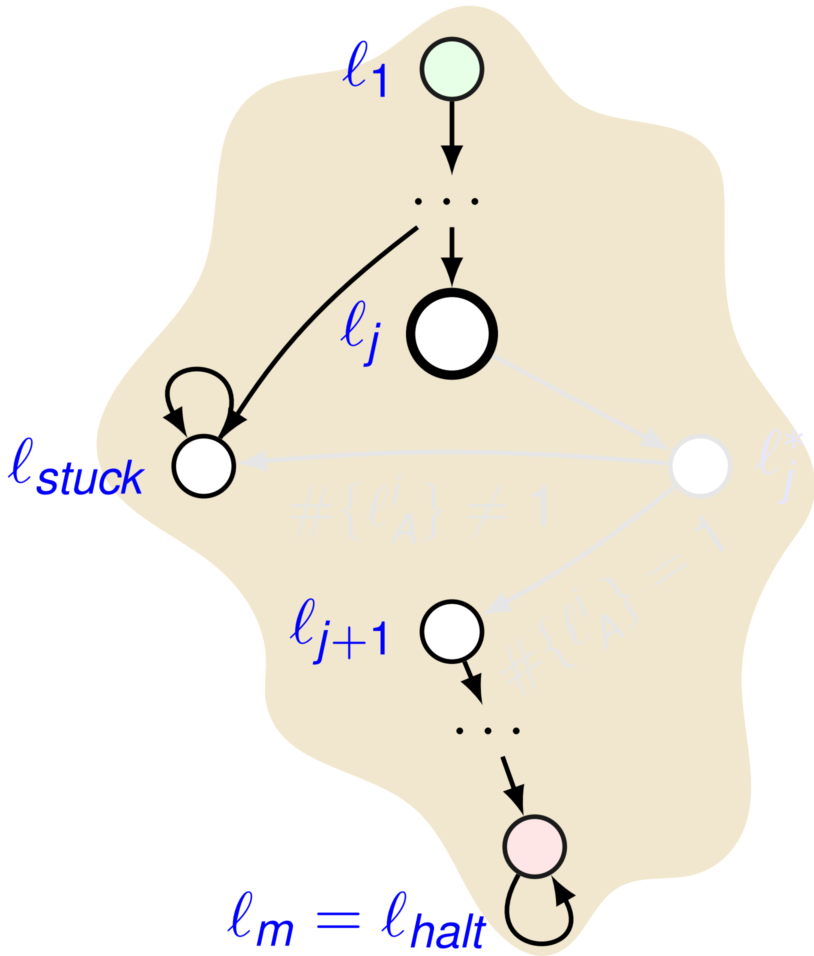
Problem:

- find an initial configuration σ_{init} and a final configuration σ_{fin}
- there is an exection from σ_{init} to σ_{fin}
- formula ϕ holds in σ_{fin}

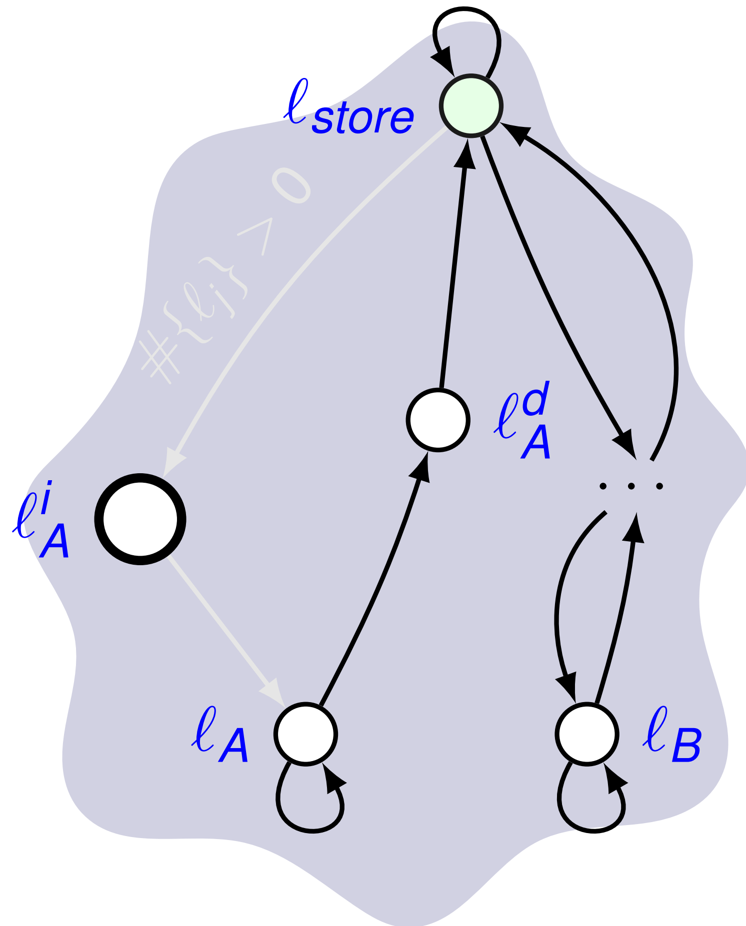
Parameterized reachability for STA is undecidable

Reduction to non-halting of a two-counter machine

control flow
processes



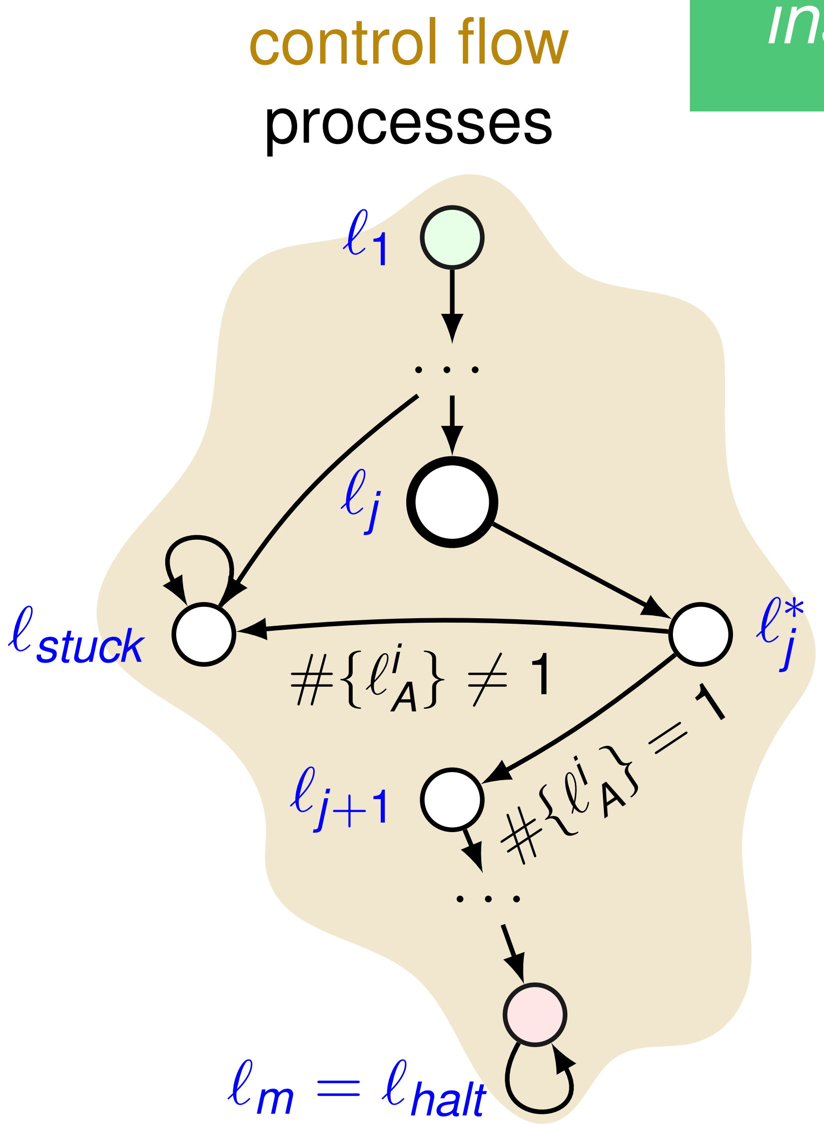
register
processes

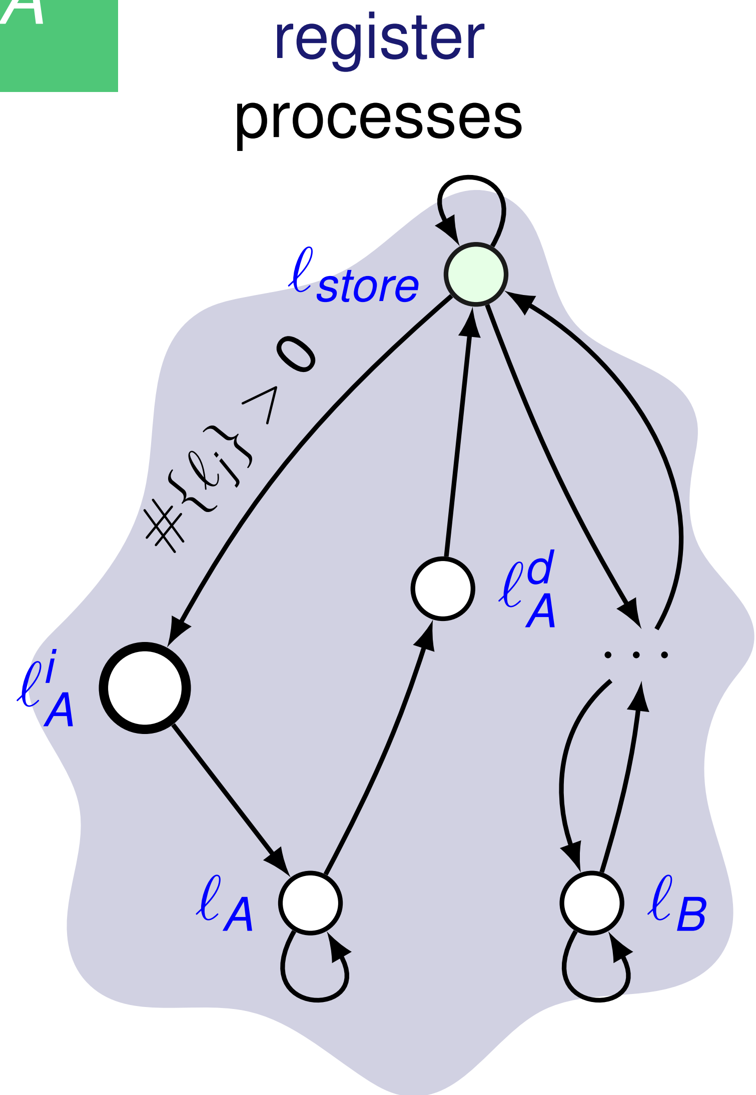


Parameterized reachability for STA is undecidable

Reduction to non-halting of a two-counter machine

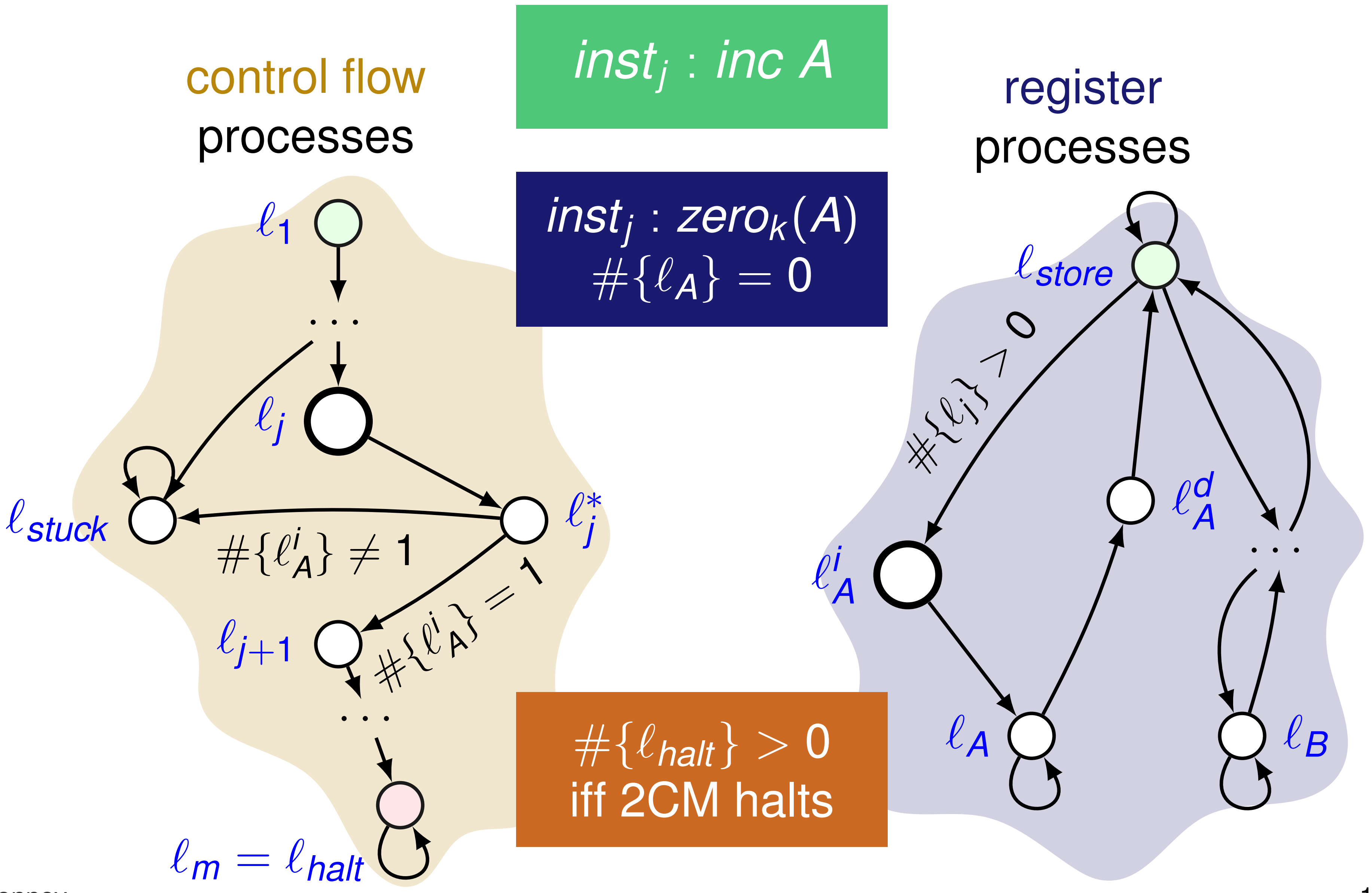
$inst_j : inc\ A$





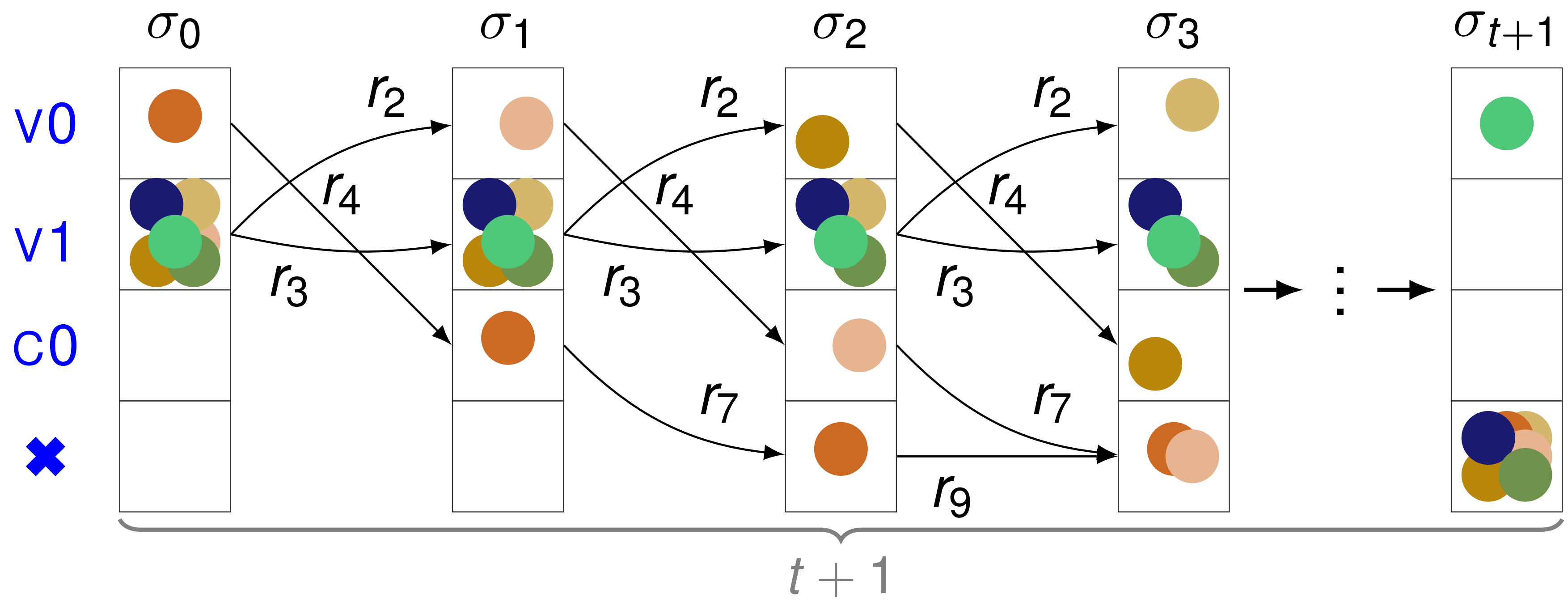
Parameterized reachability for STA is undecidable

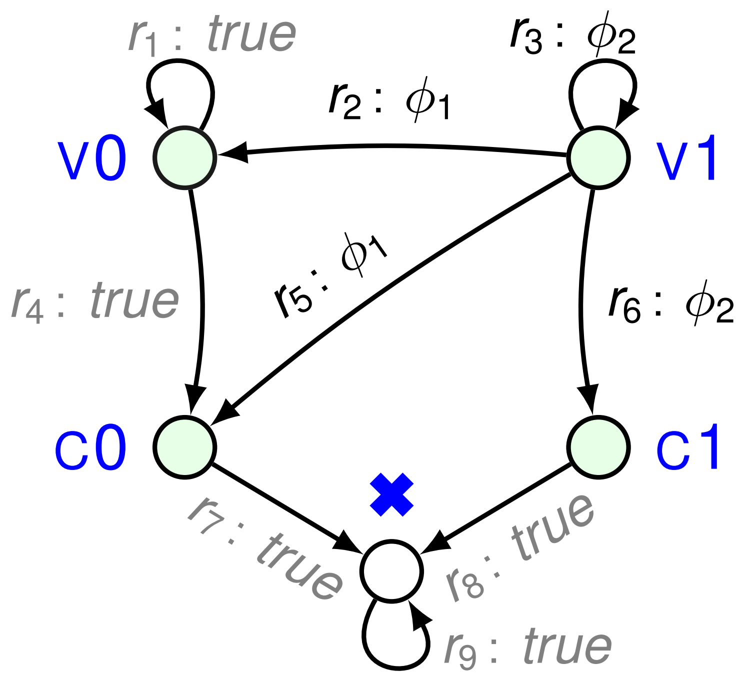
Reduction to non-halting of a two-counter machine



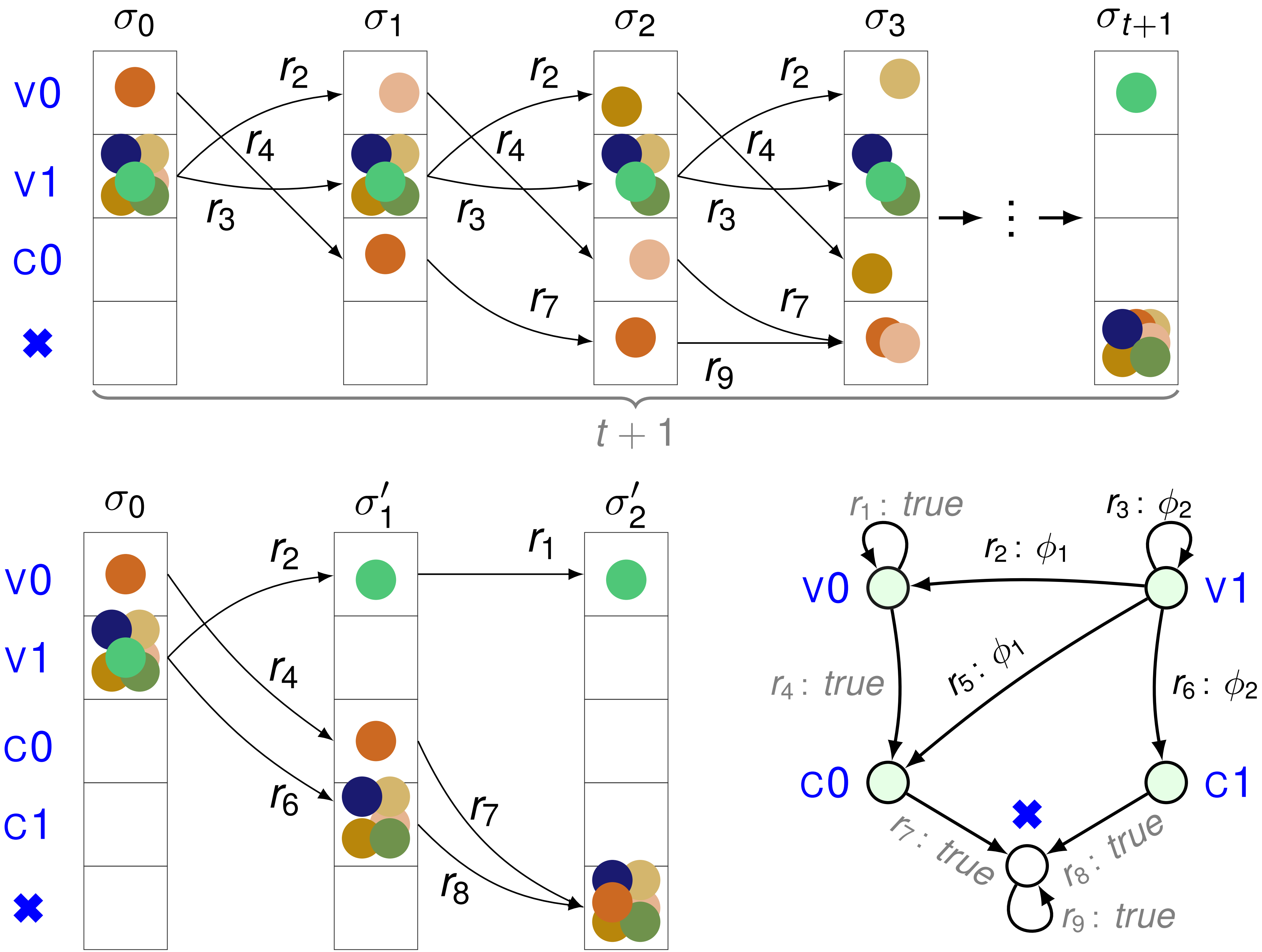
Semi-decision procedure

Long vs. short executions

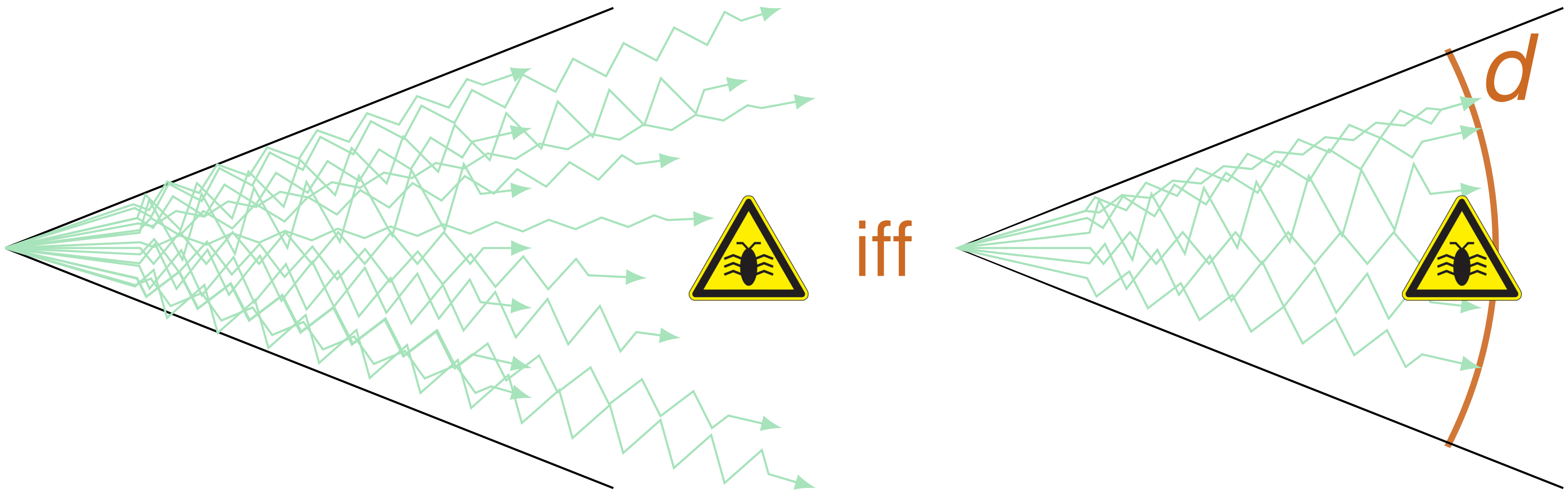




Long vs. short executions



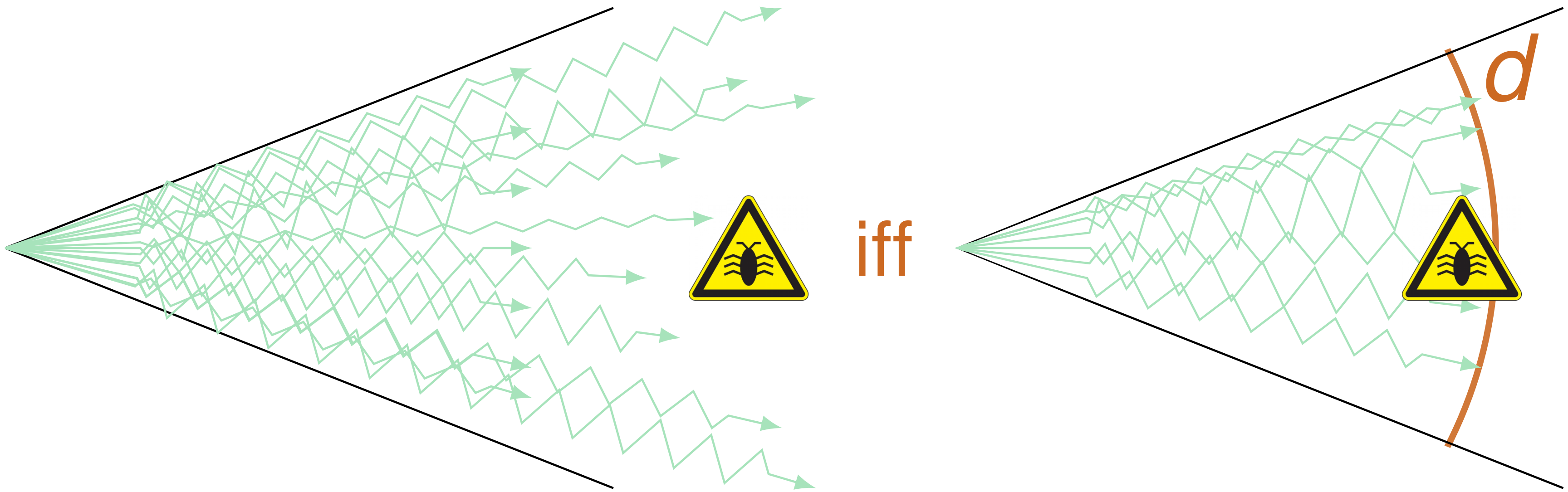
Bounded executions for reachability



Is there a number d such that we can always shorten executions to executions of length $\leq d$?

Yes, for several textbook algorithms

Bounded executions for reachability



Is there a number d such that we can always shorten executions to executions of length $\leq d$?

Yes, for several textbook algorithms

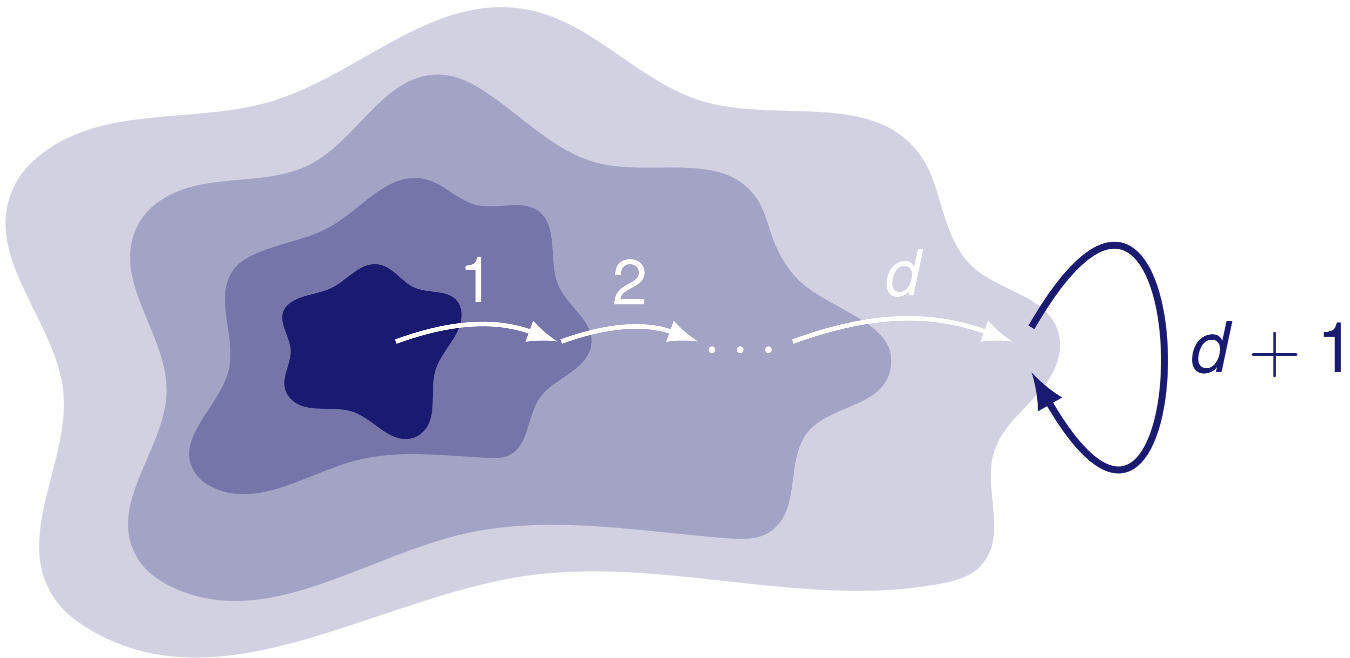
Diameters computed with SMT

algorithm	loca-tions	resilience condition	d	z3 sec.	cvc4 sec.
rb	4	$n > 3t$	**2**	0.27	0.99
rb_hybrid	8	$n > 3b + 2s$	**2**	1.16	37.6
rb_omit	8	$n > 2t$	**2**	0.43	2.47
fair_cons	11	$n > t$	**2**	0.97	10.9
floodmin, $k = 1$	5	$n > t$	**2**	0.21	0.86
floodmin, $k = 2$	7	$n > t$	**2**	0.53	7.43
floodset	7	$n > t$	**2**	0.36	3.01
kset_omit, $k = 1$	4	$n > t$	**1**	0.08	0.09
kset_omit, $k = 2$	6	$n > t$	**1**	0.17	0.27
phase_king	34	$n > 3t$	**4**	12.9	50.5
phase_queen	24	$n > 4t$	**3**	1.78	17.7

Byzantine, Send Omission, Crash

Computing the diameter d

Reach every configuration in a predefined number of steps?



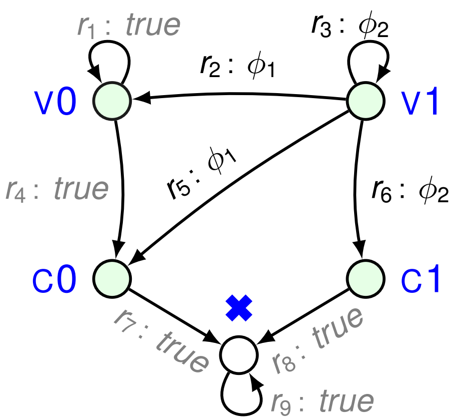
d is the diameter of the system

Safety of synchronous fault-tolerant algorithms

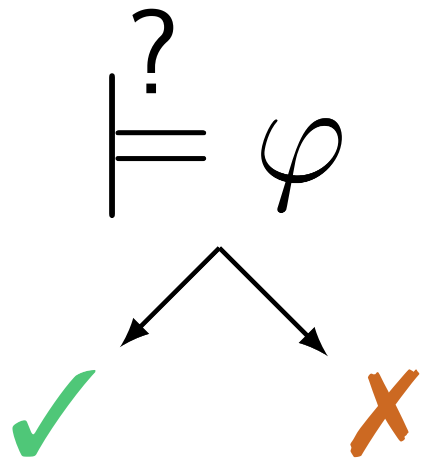
Input STA

Compute diameter

Use BMC



using SMT (Z3)



SMT encoding

d is the diameter bound iff $\Phi(d)$ holds true:

$\forall n, t, f.\ \forall \sigma_0, \ldots, \sigma_{d+1}.\ \exists \sigma'_0, \ldots, \sigma'_d.$

parameterized
+
quantifier alternation

$$\sigma_0 \xrightarrow{\tau_1} \cdots \xrightarrow{\tau_{d+1}} \sigma_{d+1} \Rightarrow$$

$$(\sigma_0 = \sigma'_0) \wedge \sigma'_0 \xrightarrow{\tau'_1} \cdots \xrightarrow{\tau'_d} \sigma'_d \wedge \bigvee_{i=0}^{d} \sigma'_i = \sigma_{d+1}$$

1. initialize d to 1
2. check if $\neg\Phi(d)$ is unsatisfiable
3. if yes, output d and terminate
4. if no, increment d, jump to step 2

SMT encoding

d is the diameter bound iff $\Phi(d)$ holds true:

parameterized
+
quantifier alternation

$$\forall n, t, f.\ \forall \sigma_0, \dots, \sigma_{d+1}.\ \exists \sigma'_0, \dots, \sigma'_d.$$

$$\sigma_0 \xrightarrow{\tau_1} \cdots \xrightarrow{\tau_{d+1}} \sigma_{d+1} \quad \Rightarrow$$

$$(\sigma_0 = \sigma'_0) \wedge \sigma'_0 \xrightarrow{\tau'_1} \cdots \xrightarrow{\tau'_d} \sigma'_d \wedge \bigvee_{i=0}^{d} \sigma'_i = \sigma_{d+1}$$

---

1. initialize d to 1
2. check if $\neg\Phi(d)$ is unsatisfiable
3. if yes, output d and terminate
4. if no, increment d, jump to step 2

SMT encoding

d is the diameter bound iff $\Phi(d)$ holds true:

parameterized
+
quantifier alternation

$$\forall n, t, f.\ \forall \sigma_0, \dots, \sigma_{d+1}.\ \exists \sigma'_0, \dots, \sigma'_d.$$

$$\begin{array}{ccccccc}
 & \sigma_0 & \xrightarrow{\tau_1} & \cdots & \xrightarrow{\tau_{d+1}} & \sigma_{d+1} & \Rightarrow \\
 & \parallel & & & & \parallel & \\
(\sigma_0 = \sigma'_0) \wedge & \sigma'_0 & \xrightarrow{\tau'_1} & \cdots & \xrightarrow{\tau'_d} & \sigma'_d & \wedge \bigvee_{i=0}^{d} \sigma'_i = \sigma_{d+1}
\end{array}$$

---

1. initialize d to 1
2. check if $\neg\Phi(d)$ is unsatisfiable
3. if yes, output d and terminate
4. if no, increment d, jump to step 2

SMT encoding

d is the diameter bound iff $\Phi(d)$ holds true:

parameterized + quantifier alternation

$$\forall n, t, f.\ \forall \sigma_0, \dots, \sigma_{d+1}.\ \exists \sigma'_0, \dots, \sigma'_d.$$

$$\begin{array}{ccccccc} & \sigma_0 & \xrightarrow{\tau_1} & \cdots & \xrightarrow{\tau_{d+1}} & \sigma_{d+1} & \Rightarrow \\ & \| & & & & \| & \\ (\sigma_0 = \sigma'_0) \wedge & \sigma'_0 & \xrightarrow{\tau'_1} & \cdots & \xrightarrow{\tau'_d} & \sigma'_d & \wedge \bigvee_{i=0}^{d} \sigma'_i = \sigma_{d+1} \end{array}$$

---

1. initialize d to 1
2. check if $\neg\Phi(d)$ is unsatisfiable
3. if yes, output d and terminate
4. if no, increment d, jump to step 2

LIA

Bounded model checking with SMT

algorithm	locations	RC	z3 sec.	cvc sec.
rb	4	$n > 3t$	0.08	0.08
rb_hybrid	8	$n > 3b + 2s$	0.09	0.15
rb_omit	8	$n > 2t$	0.09	0.14
fair_cons	11	$n > t$	0.27	0.47
floodmin, $k = 1$	5	$n > t$	0.18	0.29
floodmin, $k = 2$	7	$n > t$	0.22	0.52
floodset	7	$n > t$	0.21	0.49
kset_omit, $k = 1$	4	$n > t$	0.04	0.03
kset_omit, $k = 2$	6	$n > t$	0.04	0.07
phase_king	34	$n > 3t$	1.41	5.12
phase_queen	24	$n > 4t$	0.36	1.92

Byzantine, Send Omission, Crash

Actual bug in [BGP89a], corrected in [BGP89b]

```
for k := 1 to t+1 begin
                (* universal exchange
    send(V);
    for j := 0 to 1 do
        C[j] := the number of recei
                (* universal exchange 2 *)
    for j := 0 to 1 do begin
        send(C[j]≥n−t);
        D[j] := the number of received 1's;
    end;
    V := D[1]>t;
                (* King's broadcast *)
    if k = p then send(V);
    if D[V] < n−t then
        V := the received message;
end;
```

$C[j] \geq n - t$

Fig. 2. The *Phase King* protocol: code for processor *i*.

al exchanges are needed to achieve this.

2: *Phase King* solves the Distributed Consensus problem ounds and two-bit messages (or $4(t+1)$ rounds and single-b $> 3t$.

1. Our technique reported a counterexample

Actual bug in [BGP89a], corrected in [BGP89b]

```
V := v_i; (* i's initial value *)
for m := 1 to t+1 begin
              (* Exchange 1 *)
    send(V);
    V := 2;
    for k := 0 to 1 do begin
        C(k) := the number of received k's;
        if C(k) ≥ n−t then V := k
    end;
              (* Exchange 2 *)
    send(V);
    for k := 2 downto 0 do begin
        D(k) := the number of received k's;
        if D(k) > t then V := k
    end;
              (* Exchange 3 *)
    if m = i then
        send(V);
    if V = 2 or D(V) < n−t then
        V := MIN(1,received message);
end;
```

Fig. 4. The *Phase King* protocol: code for processor *i*.

$C(k) \geq n-t$

1. Our technique reported a counterexample

2. Corrected by changing inequality to $\geq$

Conclusions for Part II

Synchronous threshold automata to model the algorithms

Bounded model checking of counter systems

Completeness due to the diameter bounds

Diameters are not always bounded undecidability

Model checking of distributed algorithms:

from classics towards Tendermint blockchain

part III

Igor Konnov

VMCAI winter school, January 16-18, 2020

informal

INTERCHAIN FOUNDATION

Timeline

Introduction to **fault-tolerant** distributed algorithms

Verifying **synchronous** threshold-guarded algorithms

Verifying **asynchronous** threshold-guarded algorithms

Can we verify **Tendermint consensus?**

Verifying **asynchronous** threshold-guarded distributed algorithms

[K., Veith, Widder. CAV'15]
[K., Lazić, Veith, Widder. POPL'17]
[K., Lazić, Veith, Widder. FMSD'17]
[K., Widder. ISoLA'18]
...

Asynchronous systems

r_1 sends/receives on Monday/Thursday, computes on Friday

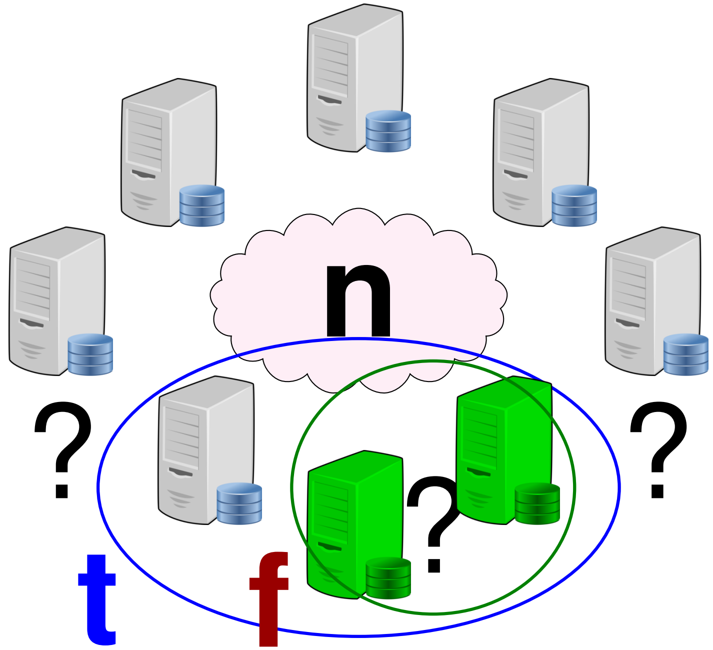
r_2 sends/receives/computes once a month

r_3 went for a two-month vacation

r_4 left job without notice

r_1 uses DHL, r_2 uses LA POSTE, r_3 uses Post

Fault-tolerant distributed algorithms



n processes send messages **asynchronously**

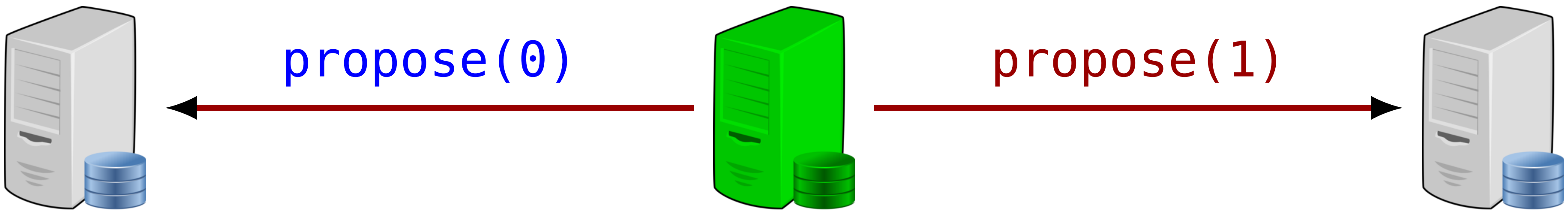
f processes are faulty (unknown)

t is an upper bound on ***f*** (known)

resilience condition on n, t, and f, e.g., $n > 3t \wedge t \geq f \geq 0$

Faults and communication

Byzantine behavior: [Lamport, Shostak, Pease, 1982]



More than two-thirds must be correct: $n > 3t$ (resilience)

Communication is **reliable**:

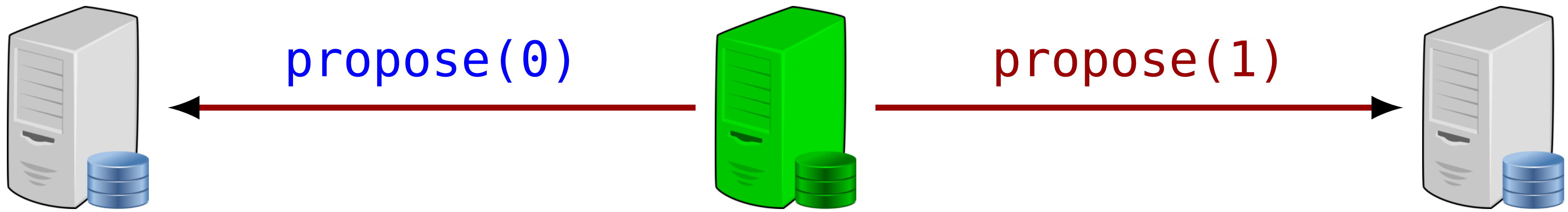
if a correct process sends a message m,

m is eventually delivered to all correct processes

[Fischer, Lynch, Paterson, 1985]

Faults and communication

Byzantine behavior: [Lamport, Shostak, Pease, 1982]



More than two-thirds must be correct: $n > 3t$ (resilience)

Communication is **reliable**:

if a correct process sends a message *m*,

m is eventually delivered to all correct processes

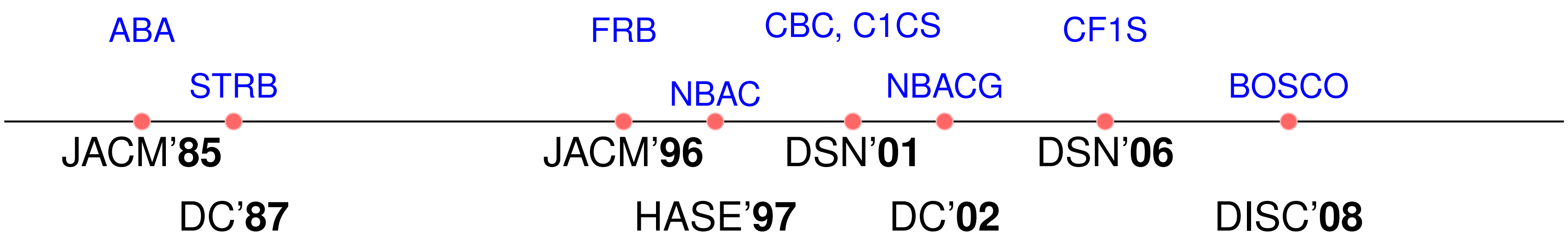
[Fischer, Lynch, Paterson, 1985]

Byzantine model checker

[forsyte.at/software/bymc]

(source code, benchmarks, virtual machines, etc.)

10 parameterized fault-tolerant distributed algorithms:



An example

One-step Byzantine asynchronous consensus

every process starts with a value $v_i \in \{0, 1\}$

agreement: no two processes decide differently

validity: if a correct process decides on *v*,
then *v* was the initial value of at least one process

unanimity: if all correct processes are initialized with *v*,
every deciding correct process must decide on *v*

termination: all correct processes eventually decide

---

decide in one communication step,

when there are "not too many faults"

One-step Byzantine asynchronous consensus

every process starts with a value $v_i \in \{0, 1\}$

agreement: no two processes decide differently

validity: if a correct process decides on v,
then v was the initial value of at least one process

unanimity: if all correct processes are initialized with v,
every deciding correct process must decide on v

termination: all correct processes eventually decide

---

decide in one communication step,

when there are "not too many faults"

BOSCO

[Song & van Renesse, DISC 2008]

```
input v_p
send ⟨VOTE, v_p⟩ to all processors;

wait until n − t VOTE messages have been received;

if more than (n+3t)/2 VOTE messages contain the same value v
then DECIDE(v);

if more than (n−t)/2 VOTE messages contain the same value v,
    and there is only one such value v
then v_p ← v;

call Underlying-Consensus(v_p);
```

resilience: of $n > 3t$ processes, $f \leq t$ processes are Byzantine

fast termination: when $n > 5t$ and $f = 0$ and $n > 7t$

Formalizing pseudo-code

Many ways to formalize distributed algorithms

General languages

for instance, TLA$^+$

model checking is hard

Parametric Promela

relatively easy to understand

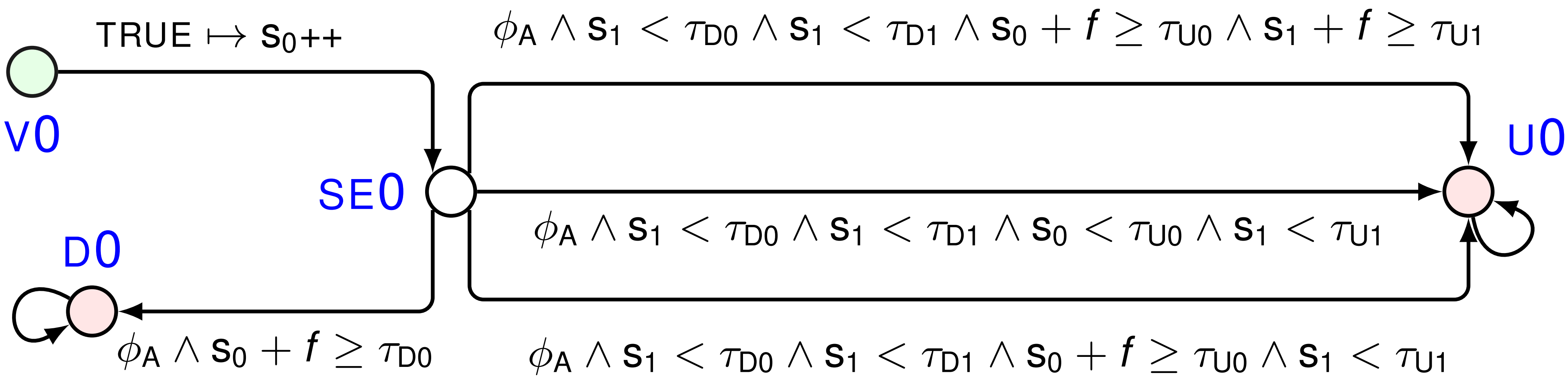
supported by ByMC via abstraction

Threshold automata

special input for ByMC

efficient model checking with SMT

(Asynchronous) threshold automata



(similar for V1, SE1, D1, U1, . . .)

threshold guards, e.g., ϕ_A is defined as $s_0 + s_1 + f \geq n - t$

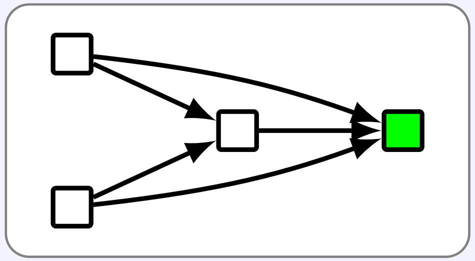
increments of shared variables, e.g., s_{0++}

run $n - f$ copies provided that there are $f \leq t$ Byzantine faults and $n > 3t$

Verifying the asynchronous algorithms

Verifying these algorithms?

Parameterized verification problem:

$\forall n, f.\quad n - f$ copies of  $\models \varphi$

Our approach:

(I) Counting processes,

(II) Acceleration,

(III) Bounded model checking, and

(IV) Schemas

(I) Counting processes

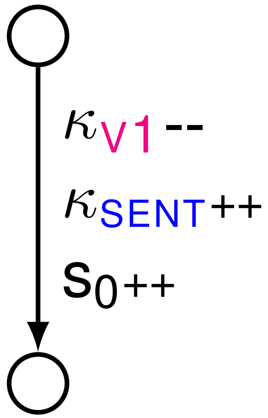
Threshold guards (e.g., $s_0 + s_1 + f \geq n - t$) do not use process ids

A transition by a single process:

$$\left\{\kappa_{V1} = 4 \wedge \kappa_{SENT} = 1 \wedge s_0 = 1\right\}$$

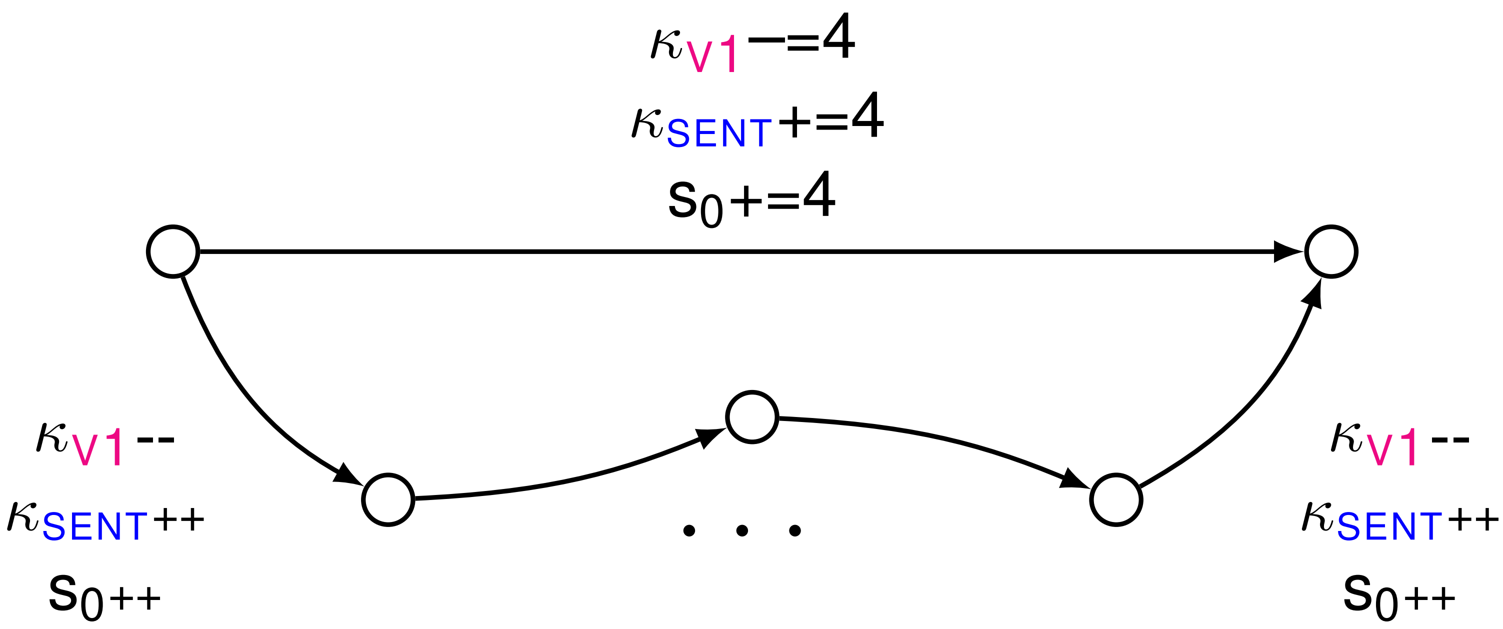
$$\kappa_{V1}\text{--} \; ; \; \kappa_{SENT}\text{++} \; ; \; s_0\text{++} \; ;$$

$$\left\{\kappa_{V1} = 3 \wedge \kappa_{SENT} = 2 \wedge s_0 = 2\right\}$$



(II) Acceleration

The same transition by unboundedly many processes in one step:

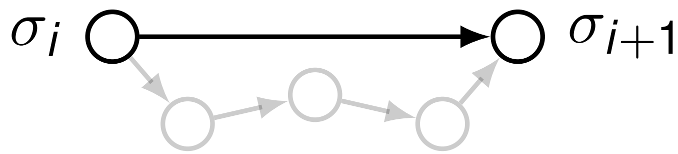


Acceleration factor can be any natural number δ

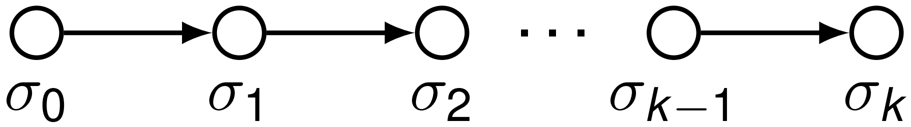
(III) Bounded model checking with SMT

A transition by δ_i processes (in linear integer arithmetic):

$$T(\sigma_i, \sigma_{i+1}, \delta_i) = \begin{bmatrix} \kappa^{i+1}_{\mathrm{V1}} = \kappa^{i}_{\mathrm{V1}} - \delta_i \wedge \\ \kappa^{i+1}_{\mathrm{SENT}} = \kappa^{i}_{\mathrm{SENT}} + \delta_i \wedge \\ \mathbf{s_0}^{i+1} = \mathbf{s_0}^{i} + \delta_i \end{bmatrix}$$



Execution:

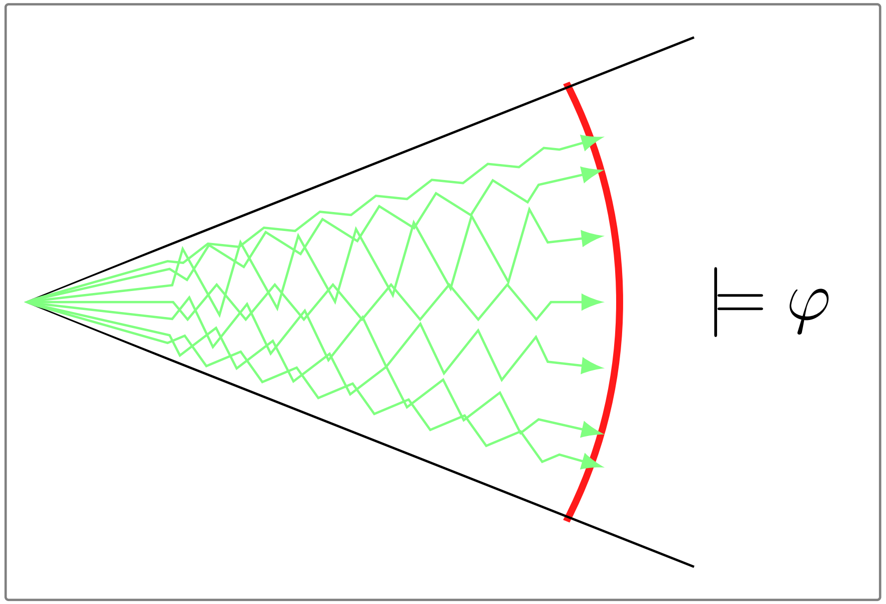


SMT formula: $T(\sigma_0, \sigma_1, \delta_0) \wedge T(\sigma_1, \sigma_2, \delta_1) \wedge \cdots \wedge T(\sigma_{k-1}, \sigma_k, \delta_{k-1}) \wedge \text{Spec}$

how long should the executions be?

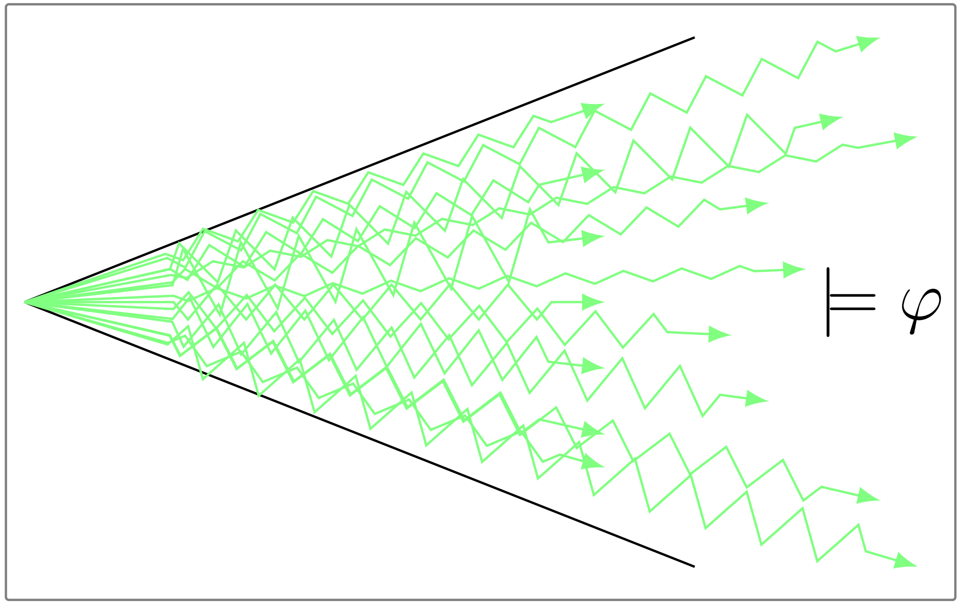
Completeness of bounded model checking

What we **can** do:



iff

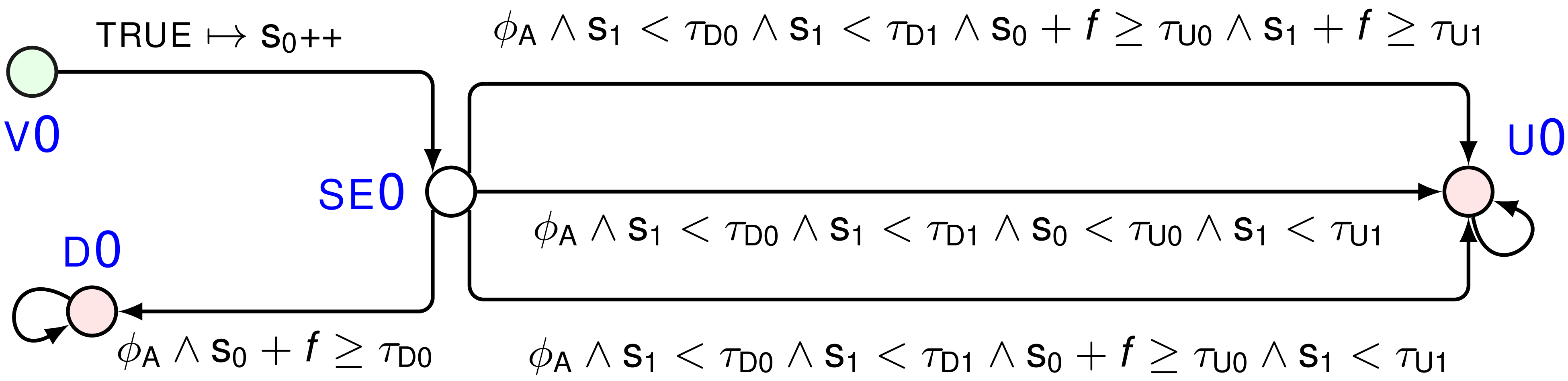
What we **want** to do:



Complete and efficient BMC for:

- reachability [K., Veith, Widder: CAV'15]
- safety and liveness [K., Lazić, Veith, Widder: POPL'17]

(Asynchronous) threshold automata



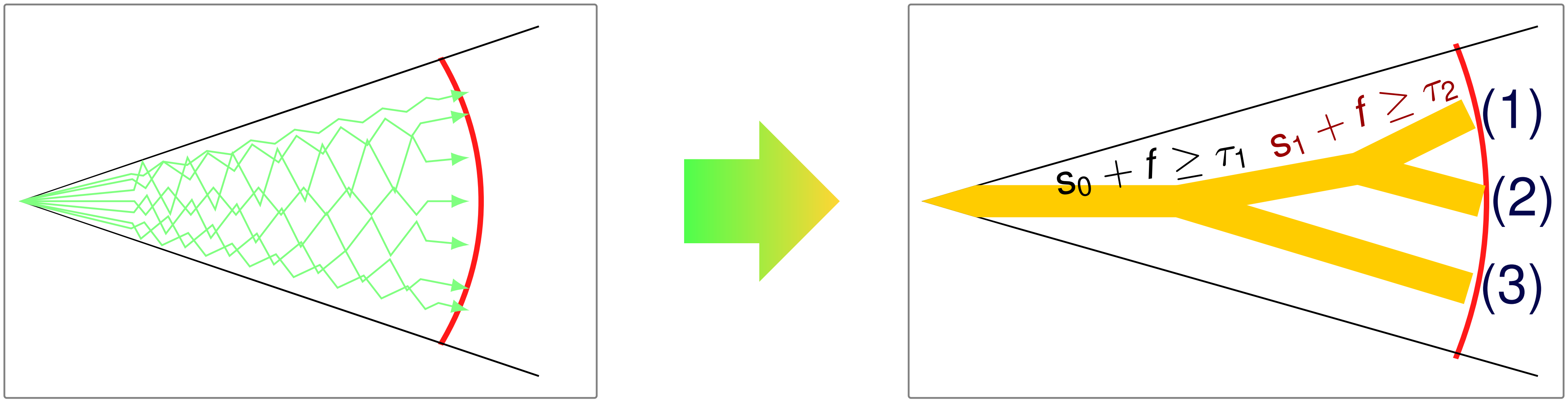
(similar for V1, SE1, D1, U1, . . .)

threshold guards, e.g., ϕ_A is defined as $s_0 + s_1 + f \geq n - t$

increments of shared variables, e.g., s_{0}++

run $n - f$ copies provided that there are $f \leq t$ Byzantine faults and $n > 3t$

Exploring all bounded executions is inefficient



The argument contains:

- reordering: $s_0{+}{+};\ s_1{+}{+};\ s_0{+}{+}$ becomes $s_0{+}{+};\ s_0{+}{+};\ s_1{+}{+}$
- acceleration $s_0{+}{+};\ s_0{+}{+};\ s_1{+}{+}$ becomes $s_0 \mathrel{+}= 2;\ s_1{+}{+}$

(IV) Schemas — encoding representatives

Schema: $\{pre_1\}$ $actions_1$ $\{post_1\}$ ... $\{pre_k\}$ $actions_k$ $\{post_k\}$

Example:

$\{\}$ $(\mathsf{V0} \to \mathsf{SE0})^{\delta_1}$ $\{s_0 + f \geq \tau_{D0}\}$ $(\mathsf{V1} \to \mathsf{SE1})^{\delta_2}$ $\{\dots, s_1 + f \geq \tau_{D1}\}$

$(\mathsf{V0} \to \mathsf{SE0})^{\delta_3}, (\mathsf{V1} \to \mathsf{SE1})^{\delta_4}$ $\{\dots, \phi_A\}$ $(\mathsf{SE0} \to \mathsf{D0})^{\delta_5}, (\mathsf{SE1} \to \mathsf{D1})^{\delta_6}$

$\{\kappa^6_{D0} \neq 0 \land \kappa^6_{D1} \neq 0\}$

SMT solver tries to find: parameters n, t, f,
acceleration factors $\delta(1), \dots, \delta(6)$,
counters $\kappa^i_{D0}, \kappa^i_{D1}, \dots$

(a) the schema does not violate the property (**UNSAT**), or
(b) there is a counterexample (**SAT**)

(IV) Schemas — encoding representatives

Schema: $\{pre_1\}$ $actions_1$ $\{post_1\}$ ... $\{pre_k\}$ $actions_k$ $\{post_k\}$

Example:

$\{\}$ $(\mathsf{V0} \to \mathsf{SE0})^{\delta_1}$ $\{s_0 + f \geq \tau_{\mathsf{D0}}\}$ $(\mathsf{V1} \to \mathsf{SE1})^{\delta_2}$ $\{\dots, s_1 + f \geq \tau_{\mathsf{D1}}\}$

$(\mathsf{V0} \to \mathsf{SE0})^{\delta_3}, (\mathsf{V1} \to \mathsf{SE1})^{\delta_4}$ $\{\dots, \phi_{\mathsf{A}}\}$ $(\mathsf{SE0} \to \mathsf{D0})^{\delta_5}, (\mathsf{SE1} \to \mathsf{D1})^{\delta_6}$

$\{\boldsymbol{\kappa}^6_{\mathsf{D0}} \neq 0 \land \boldsymbol{\kappa}^6_{\mathsf{D1}} \neq 0\}$

SMT solver tries to find: parameters n, t, f,
acceleration factors $\delta(1), \dots, \delta(6)$,
counters $\boldsymbol{\kappa}^i_{\mathsf{D0}}, \boldsymbol{\kappa}^i_{\mathsf{D1}}, \dots$

(a) the schema does not violate the property (**UNSAT**), or
(b) there is a counterexample (**SAT**)

(IV) Schemas — encoding representatives

Schema: $\{pre_1\}$ $actions_1$ $\{post_1\}$ $\ldots$ $\{pre_k\}$ $actions_k$ $\{post_k\}$

Example:

$\{\}$ $(\mathsf{V0} \to \mathsf{SE0})^{\delta_1}$ $\{s_0 + f \geq \tau_{\mathsf{D0}}\}$ $(\mathsf{V1} \to \mathsf{SE1})^{\delta_2}$ $\{\ldots, s_1 + f \geq \tau_{\mathsf{D1}}\}$
$(\mathsf{V0} \to \mathsf{SE0})^{\delta_3}, (\mathsf{V1} \to \mathsf{SE1})^{\delta_4}$ $\{\ldots, \phi_{\mathsf{A}}\}$ $(\mathsf{SE0} \to \mathsf{D0})^{\delta_5}, (\mathsf{SE1} \to \mathsf{D1})^{\delta_6}$

$\{\kappa^6_{\mathsf{D0}} \neq 0 \wedge \kappa^6_{\mathsf{D1}} \neq 0\}$

SMT solver tries to find: parameters n, t, f,
acceleration factors $\delta(1), \ldots, \delta(6)$,
counters $\kappa^i_{\mathsf{D0}}, \kappa^i_{\mathsf{D1}}, \ldots$

(a) the schema does not violate the property (**UNSAT**), or
(b) there is a counterexample (**SAT**)

(IV) Schemas — encoding representatives

Schema: $\{pre_1\}$ $actions_1$ $\{post_1\}$ ... $\{pre_k\}$ $actions_k$ $\{post_k\}$

Example:

$\{\}$ $(\mathsf{V0} \to \mathsf{SE0})^{\delta_1}$ $\{s_0 + f \geq \tau_{\mathsf{D0}}\}$ $(\mathsf{V1} \to \mathsf{SE1})^{\delta_2}$ $\{\dots, s_1 + f \geq \tau_{\mathsf{D1}}\}$
$(\mathsf{V0} \to \mathsf{SE0})^{\delta_3}, (\mathsf{V1} \to \mathsf{SE1})^{\delta_4}$ $\{\dots, \phi_{\mathsf{A}}\}$ $(\mathsf{SE0} \to \mathsf{D0})^{\delta_5}, (\mathsf{SE1} \to \mathsf{D1})^{\delta_6}$

$\{\boldsymbol{\kappa}^6_{\mathsf{D0}} \neq 0 \wedge \boldsymbol{\kappa}^6_{\mathsf{D1}} \neq 0\}$

SMT solver tries to find: parameters n, t, f,
acceleration factors $\delta(1), \dots, \delta(6)$,
counters $\boldsymbol{\kappa}^i_{\mathsf{D0}}, \boldsymbol{\kappa}^i_{\mathsf{D1}}, \dots$

(a) the schema does not violate the property (**UNSAT**), or
(b) there is a counterexample (**SAT**)

(IV) Schemas — encoding representatives

Schema: $\{pre_1\}$ $actions_1$ $\{post_1\}$ $\dots$ $\{pre_k\}$ $actions_k$ $\{post_k\}$

Example:

$\{\}$ $(\mathsf{V0} \to \mathsf{SE0})^{\delta_1}$ $\{s_0 + f \geq \tau_{\mathsf{D0}}\}$ $(\mathsf{V1} \to \mathsf{SE1})^{\delta_2}$ $\{\dots, s_1 + f \geq \tau_{\mathsf{D1}}\}$
$(\mathsf{V0} \to \mathsf{SE0})^{\delta_3}, (\mathsf{V1} \to \mathsf{SE1})^{\delta_4}$ $\{\dots, \phi_{\mathsf{A}}\}$ $(\mathsf{SE0} \to \mathsf{D0})^{\delta_5}, (\mathsf{SE1} \to \mathsf{D1})^{\delta_6}$

$\{\boldsymbol{\kappa}^6_{\mathsf{D0}} \neq 0 \wedge \boldsymbol{\kappa}^6_{\mathsf{D1}} \neq 0\}$

SMT solver tries to find: parameters n, t, f,
acceleration factors $\delta(1), \dots, \delta(6)$,
counters $\boldsymbol{\kappa}^i_{\mathsf{D0}}, \boldsymbol{\kappa}^i_{\mathsf{D1}}, \dots$

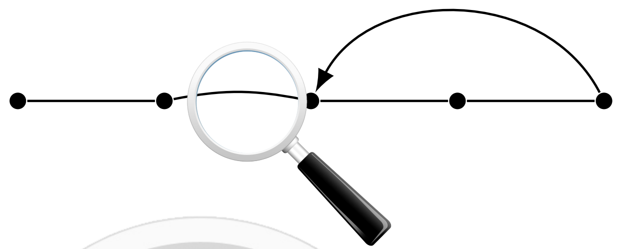
(a) the schema does not violate the property (**UNSAT**), or
(b) there is a counterexample (**SAT**)

From reachability to safety & liveness

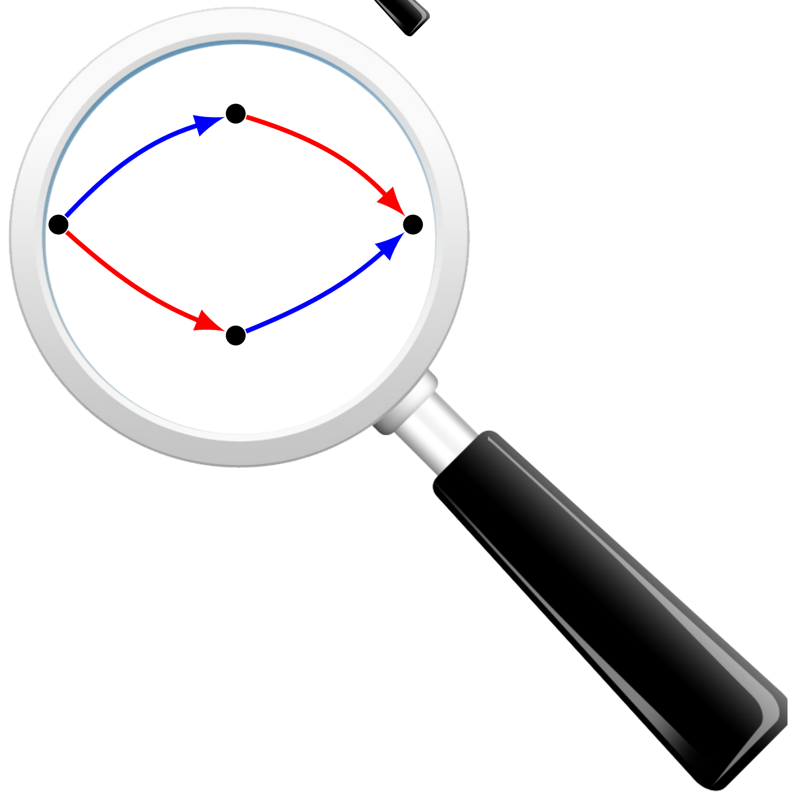
A) A temporal logic for bad executions

$$\mathbf{E}\,(\varphi_1 \wedge \Diamond \Box (\varphi_2 \vee \varphi_3))$$

B) Enumerating shapes of counterexamples



C) Property specific mover analysis



Details in [K., Lazić, Veith, Widder. POPL'17]

Overview of the verification algorithm

Threshold automaton $\longrightarrow$ schemas $\{S_1, \ldots, S_k\}$

$Z3 \models S_1$ sat

$Z3 \models S_2$ counterexample

$\ldots$

$Z3 \models S_k$

unsat?

Overview of the verification algorithm

Threshold automaton $\longrightarrow$ schemas $\{S_1, \ldots, S_k\}$

$Z3 \models S_1$

$Z3 \models S_2$

$\ldots$

$Z3 \models S_k$

sat

counterexample

unsat?

Overview of the verification algorithm

Threshold automaton $\longrightarrow$ schemas $\{S_1, \ldots, S_k\}$

$Z3 \models S_1$

$Z3 \models S_2$

$\ldots$

$Z3 \models S_k$

sat

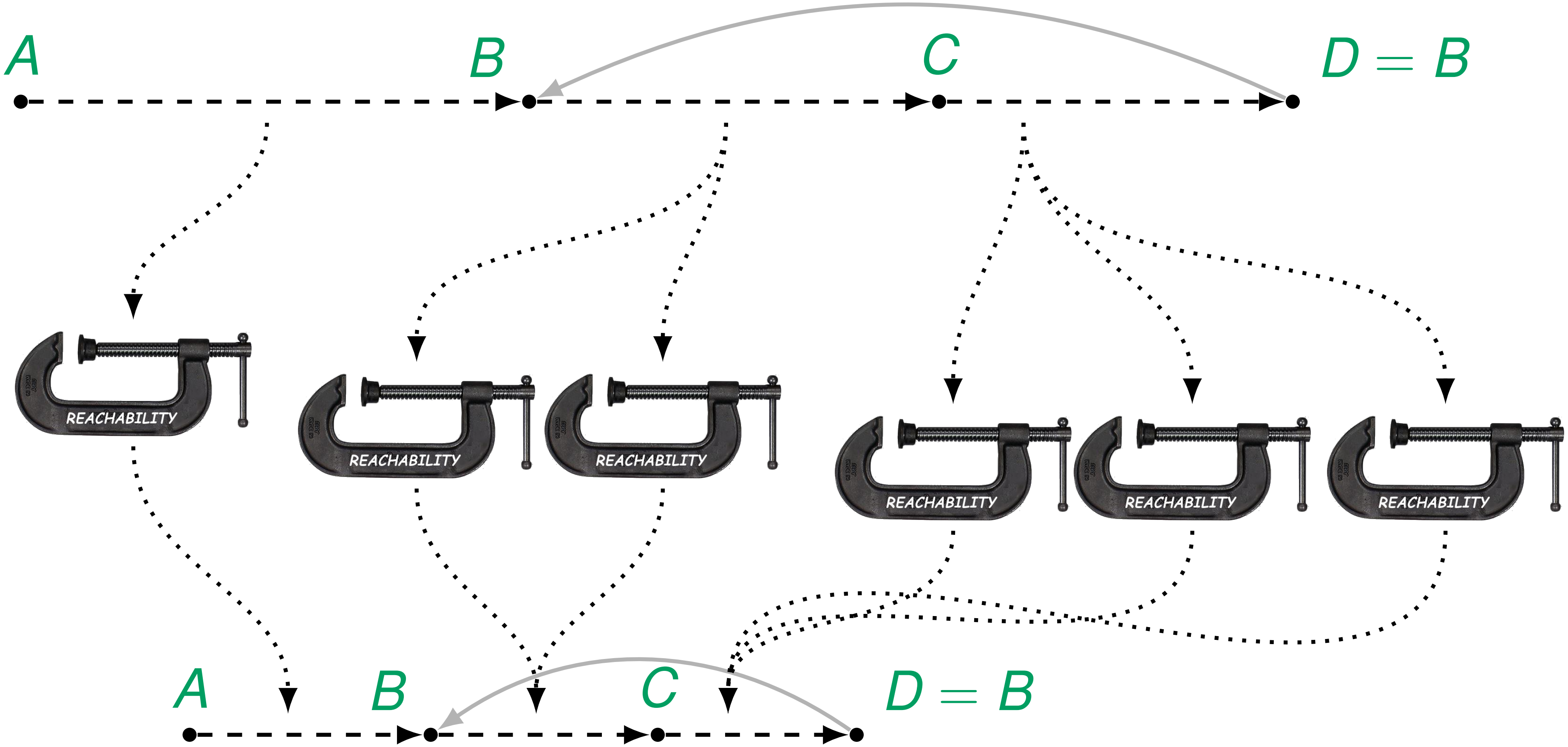
counterexample

unsat?

©VSC / Claudia Blaas-Schenner

Vienna Scientific Cluster

Short counterexamples for safety or liveness



Safety & liveness (POPL'17)

Every lasso can be transformed into a bounded one. The bound depends on the process code and the specification, not the parameters.

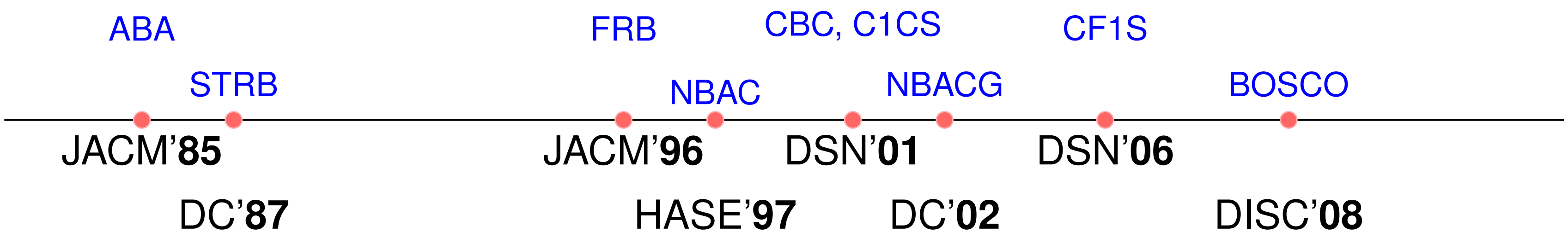
Experiments

Byzantine model checker

[forsyte.at/software/bymc]

(source code, benchmarks, virtual machines, etc.)

10 parameterized fault-tolerant distributed algorithms:



More threshold guards...

Reliable broadcast	$x \geq t + 1$ $x \geq n - t$	[Srikanth, Toueg'86]
Hybrid broadcast	$x \geq t_b + 1$ $x \geq n - t_b - t_c$	[Widder, Schmid'07]
Byzantine agreement	$x \geq \lceil \frac{n}{2} \rceil + 1$	[Bracha, Toueg'85]
Non-blocking atomic commitment	$x \geq n$	[Raynal'97], [Guerraoui'01]
Condition-based consensus	$x \geq n - t$ $x \geq \lceil \frac{n}{2} \rceil + 1$	[Mostéfaoui, Mourgaya, Parvedy, Raynal'03]
Consensus in one communication step	$x \geq n - t$ $x \geq n - 2t$	[Brasileiro, Greve, Mostéfaoui, Raynal'03]
Byzantine one-step consensus	$x \geq \lceil \frac{n+3t}{2} \rceil + 1$	[Song, van Renesse'08]

In general, there is a resilience condition, e.g., $n > 3t$, $n > 7t$

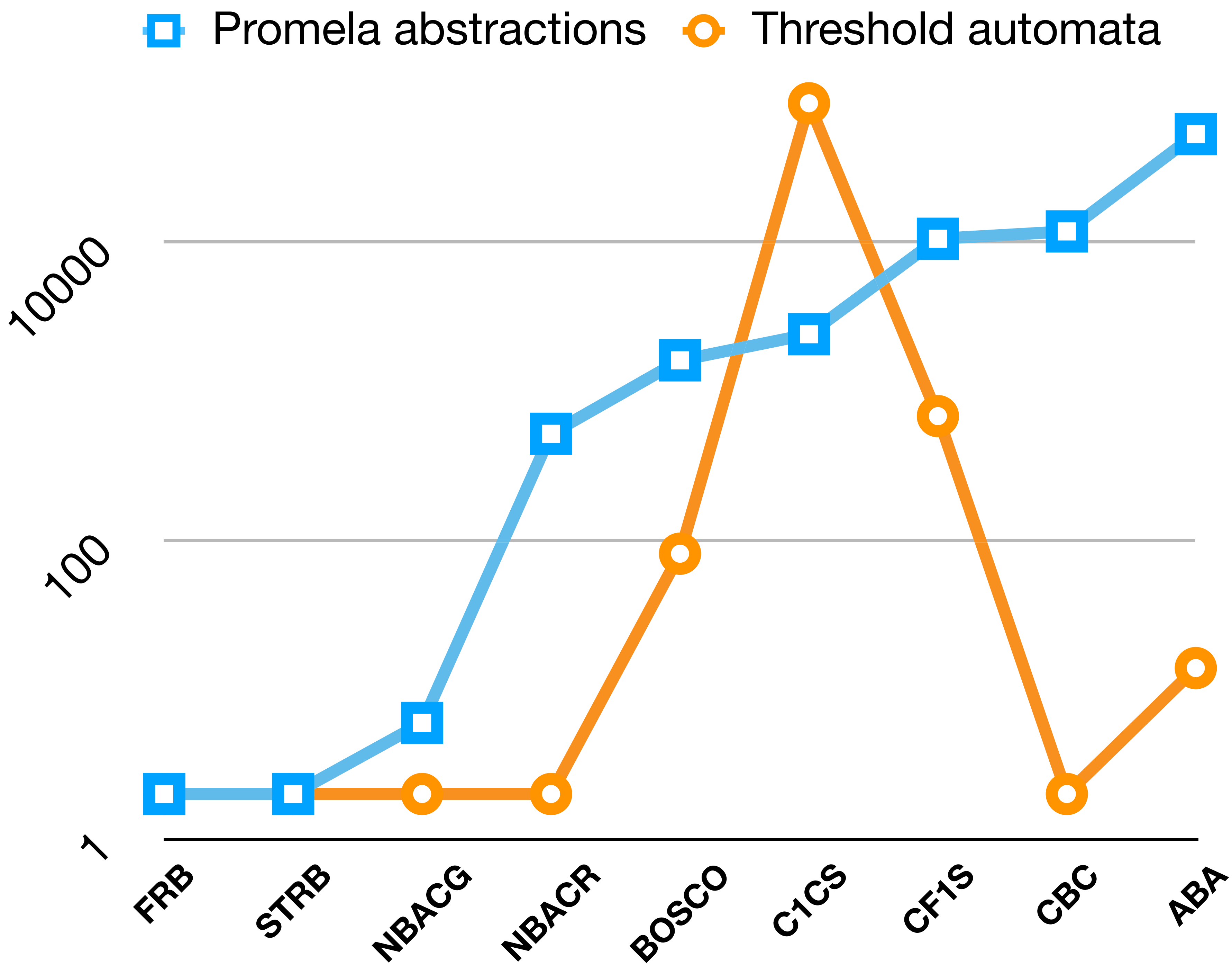
Benchmarks

Each benchmark has two versions:

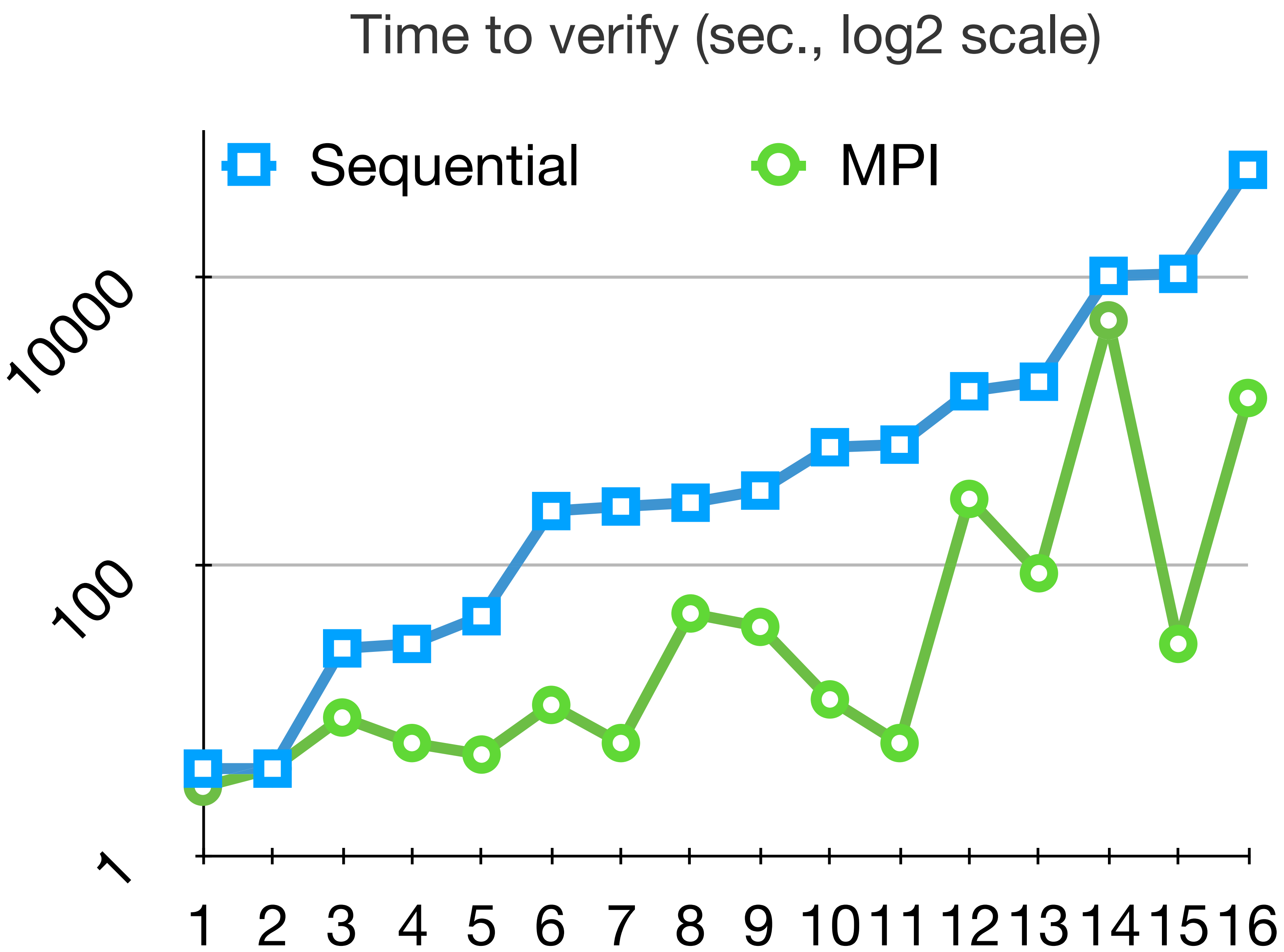
1. Threshold automaton *hand-written*
2. Promela code *automatic abstraction*

Condition-based consensus	Consensus in one comm. step
One-step consensus	BOSCO
Non-blocking atomic commitment (2 versions)	
Reliable broadcast	Folklore broadcast
	Asynchronous Byzantine agreement

Time to check the algorithms

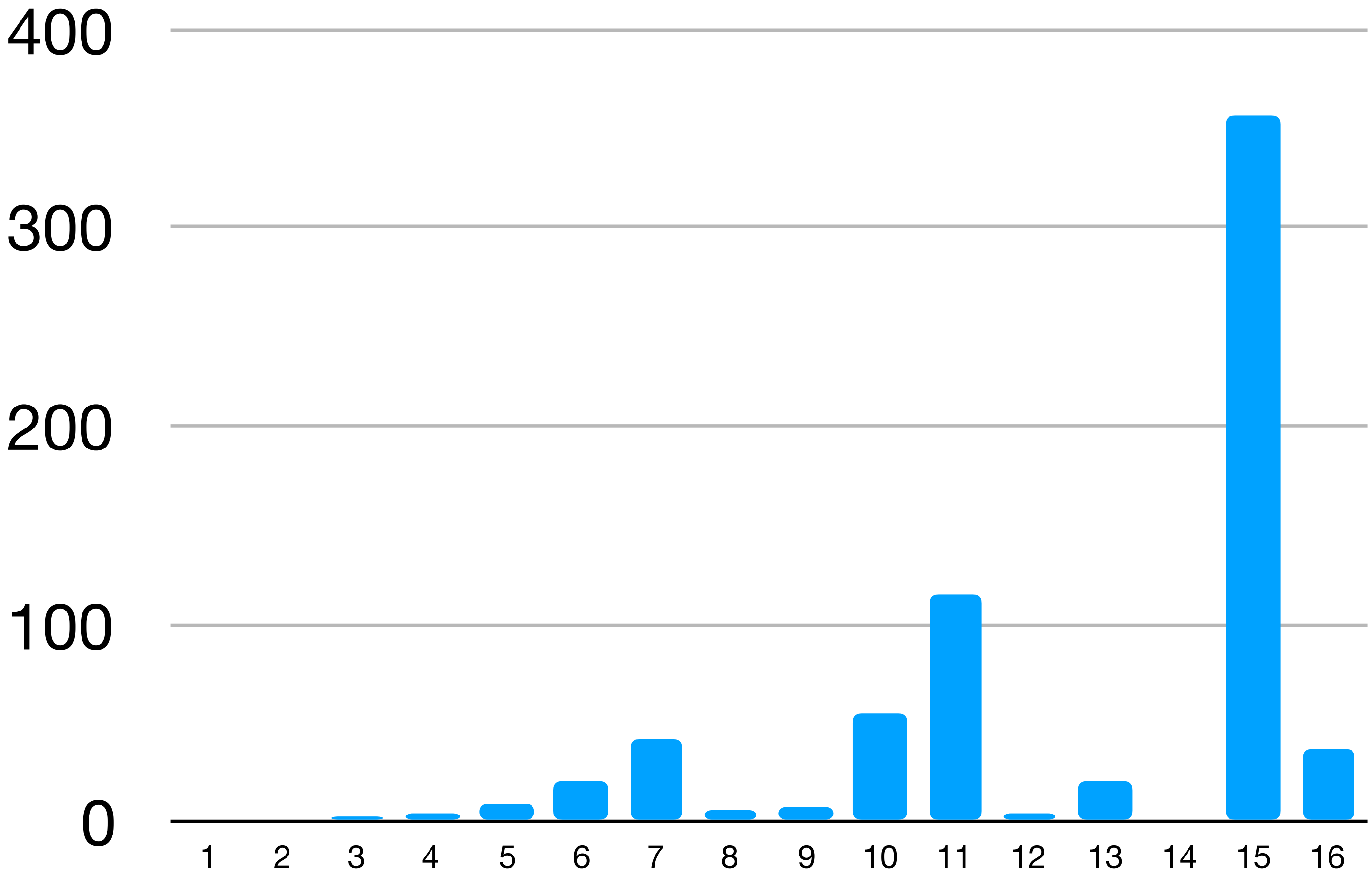


Sequential vs. parallel (256 MPI cores)

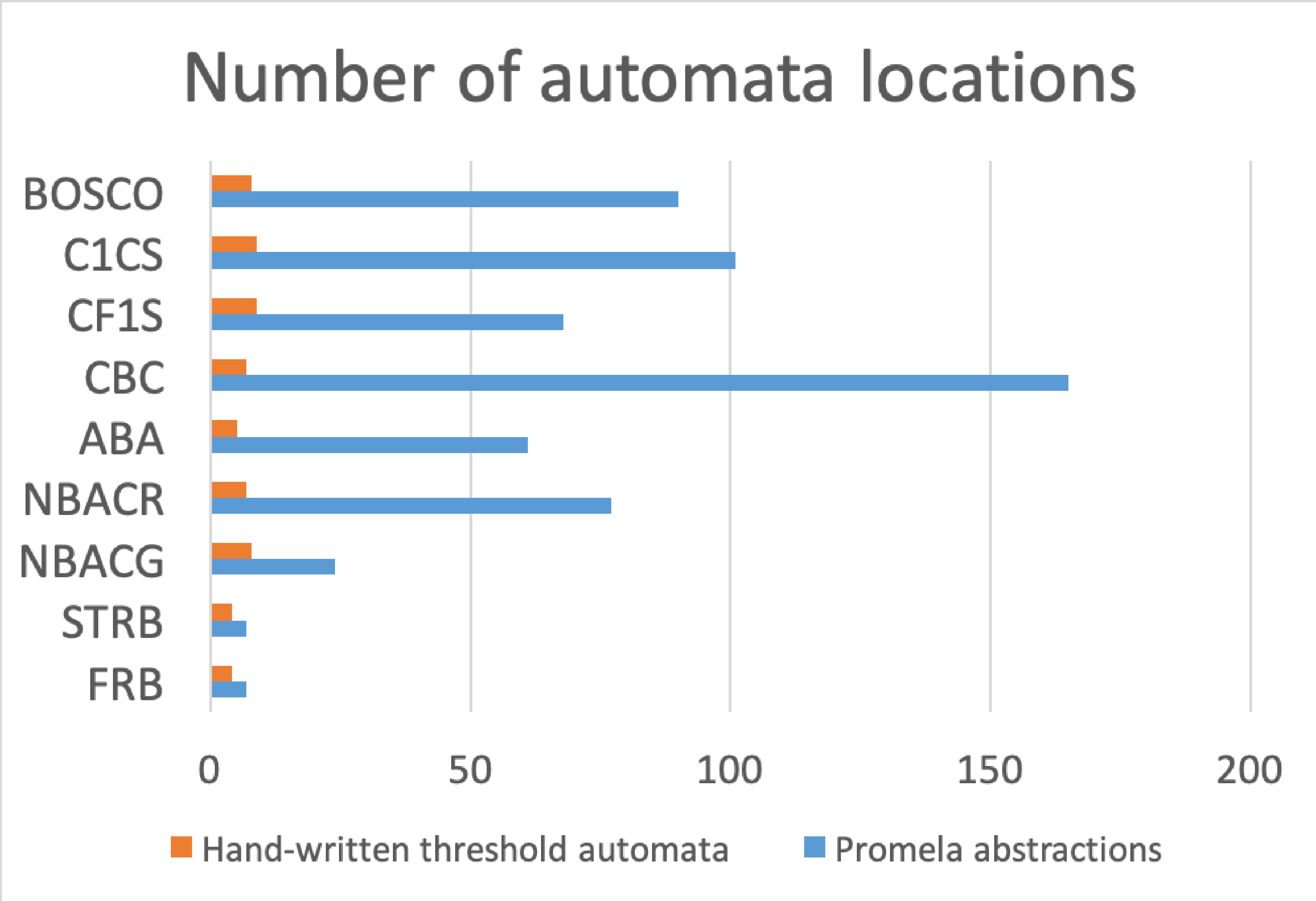


Speedup

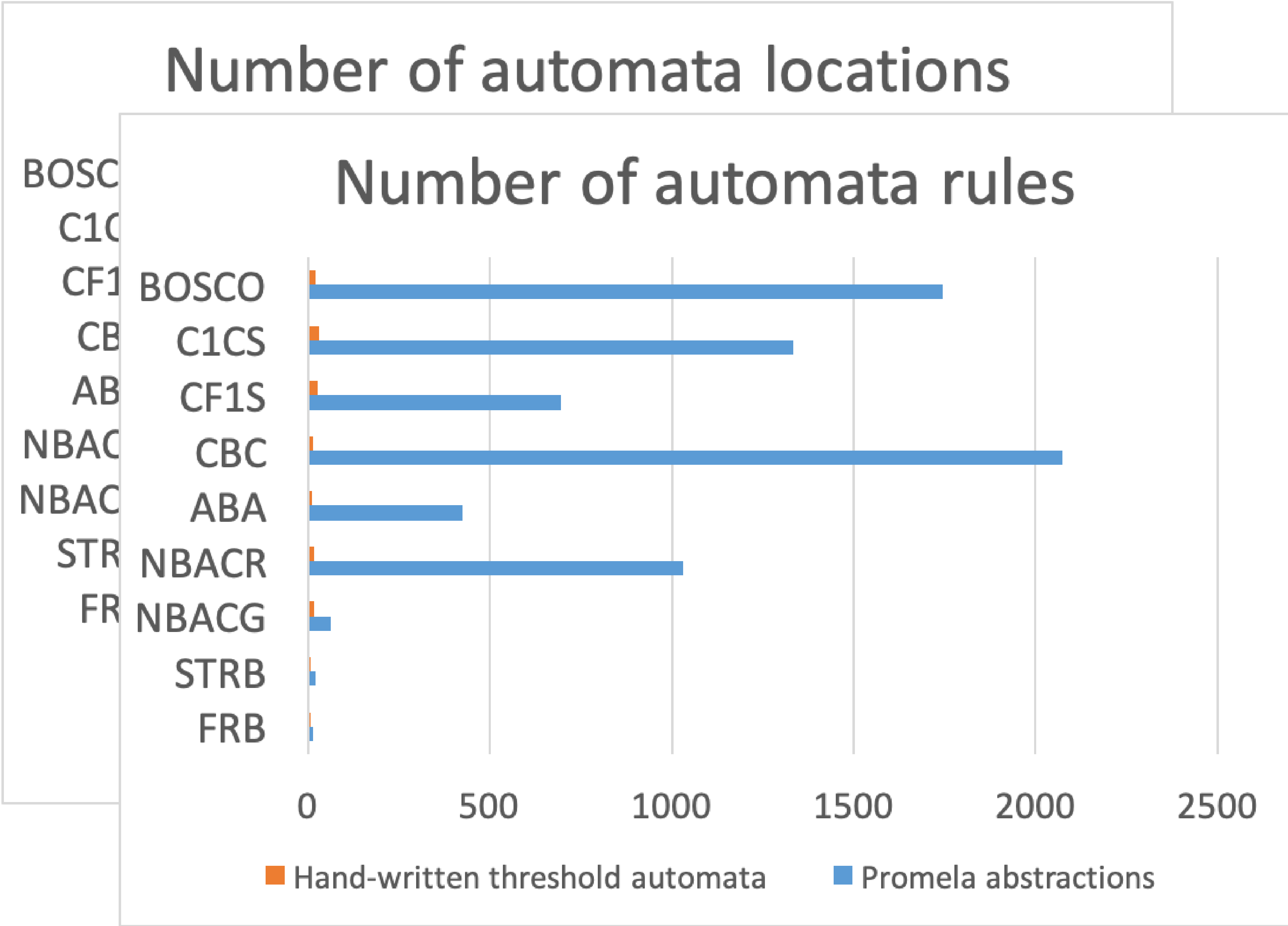
sometimes, the number of schemas is smaller than the number of cores (256)



Promela vs. threshold automata: input



Promela vs. threshold automata: input



Conclusions for Part III

Threshold automata to model asynchronous algorithms

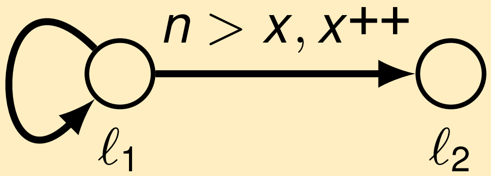
Bounded model checking of counter systems

Completeness due to the bounds

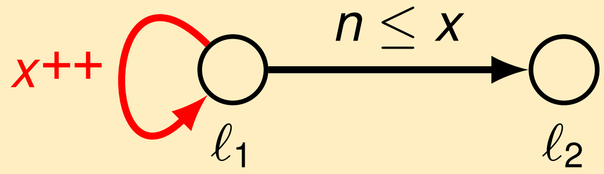
. . . for safety and liveness

Extending threshold automata

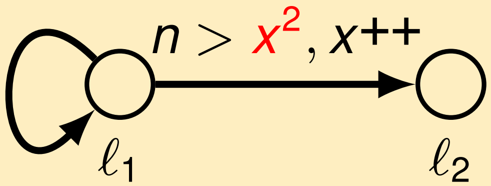
standard TA



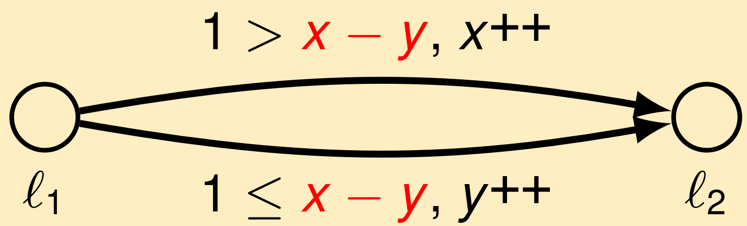
increments in loops (NCTA)



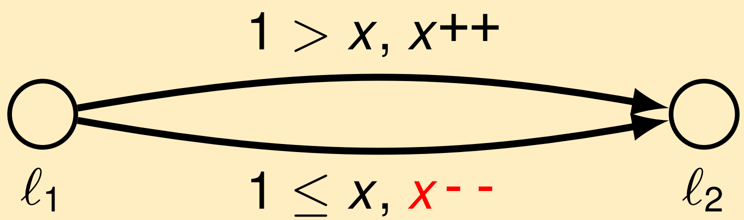
piecewise monotone (PMTA)



bounded difference (BDTA)



reversible (RTA)



reversal bounded (RBTA)

Like reversible automata, but increments and decrements of variables may alternate a bounded number of times.

All flavors of threshold automata

Level	Reversals	Canonical	Bounded diameter	Flattable	Decidable reachability	Fragment
x	0	✓	✓	✓	✓	TA
p.m. $f(x)$	0	✓	✓	✓	✓	PMTA
x	$\leq k$	✓	✓	✓	✓	RBTA
x	0	✗	✗	✓	✓	NCTA
$x - y$	0	✓	✗	✗	✗	BDTA
x	∞	✓	✗	✗	✗	RTA

Jure Kukovec

I.K.

Josef Widder

Randomized consensus algorithm Ben-Or

```
bool v := input_value({0, 1});
int r := 1;
while (true) do
 send (R,r,v) to all;
 wait for n - t messages (R,r,∗);

 if received (n + t) / 2 messages (R,r,w)
 then send (P,r,w,D) to all;
 else send (P,r,?) to all;
 wait for n - t messages (P,r,∗);

 if received at least t + 1
    messages (P,r,w,D) then {
  v := w;                  /∗ enough support −> update estimate ∗/
  if received at least (n + t) / 2
   messages (P,r,w,D)
  then decide w;                  /∗ strong majority −> decide ∗/
 } else v := random({0,1});       /∗ unclear −> coin toss       ∗/
 r := r + 1;
od
```

[Ben-Or, PODC 1983]

Randomized consensus algorithm Ben-Or

```
bool v := input_value({0, 1});
int r := 1;
while (true) do
 send (R,r,v) to all;
 wait for n - t messages (R,r,*);

 if received (n + t) / 2 messages (R,r,w)
 then send (P,r,w,D) to all;
 else send (P,r,?) to all;
 wait for n - t messages (P,r,*);

 if received at least t + 1
   messages (P,r,w,D) then {
  v := w;                  /* enough support -> update estimate */
  if received at least (n + t) / 2
   messages (P,r,w,D)
  then decide w;                  /* strong majority -> decide */
 } else v := random({0,1});      /* unclear -> coin toss        */
 r := r + 1;
od
```

[Ben-Or, PODC 1983]

Randomized consensus algorithm Ben-Or

```
bool v := input_value({0, 1});
int r := 1;
while (true) do
  send (R,r,v) to all;
  wait for n - t messages (R,r,*);

  if received (n + t) / 2 messages (R,r,w)
  then send (P,r,w,D) to all;
  else send (P,r,?) to all;
  wait for n - t messages (P,r,*);

  if received at least t + 1
      messages (P,r,w,D) then {
    v := w;                      /* enough support -> update estimate */
    if received at least (n + t) / 2
     messages (P,r,w,D)
    then decide w;               /* strong majority -> decide */
  } else v := random({0,1});     /* unclear -> coin toss          */
  r := r + 1;
od
```

[Ben-Or, PODC 1983]

Randomized consensus algorithm Ben-Or

```
bool v := input_value({0, 1});
int r := 1;
while (true) do
 send (R,r,v) to all;
 wait for n - t messages (R,r,*);

 if received (n + t) / 2 messages (R,r,w)
 then send (P,r,w,D) to all;
 else send (P,r,?) to all;
 wait for n - t messages (P,r,*);

 if received at least t + 1
     messages (P,r,w,D) then {
   v := w;                    /* enough support -> update estimate */
   if received at least (n + t) / 2
    messages (P,r,w,D)
   then decide w;                 /* strong majority -> decide */
 } else v := random({0,1});     /* unclear -> coin toss          */
 r := r + 1;
od
```

[Ben-Or, PODC 1983]

No consensus algorithm for asynchronous systems (FLP'85)

Coin toss to break ties: $\mathit{value} := \mathit{random}(\{0, 1\})$

Ben-Or's, Bracha's consensus, RS-Bosco, k-set agreement

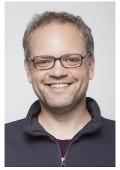
Compositional reasoning and reduction for multiple rounds

ByMC to reason about a single round

Nathalie Bertrand

I.K.

Marijana Lazić



Josef Widder

Model checking of distributed algorithms:

from classics towards Tendermint blockchain

part IV

Igor Konnov

VMCAI winter school, January 16-18, 2020

informal

INTERCHAIN FOUNDATION

Timeline

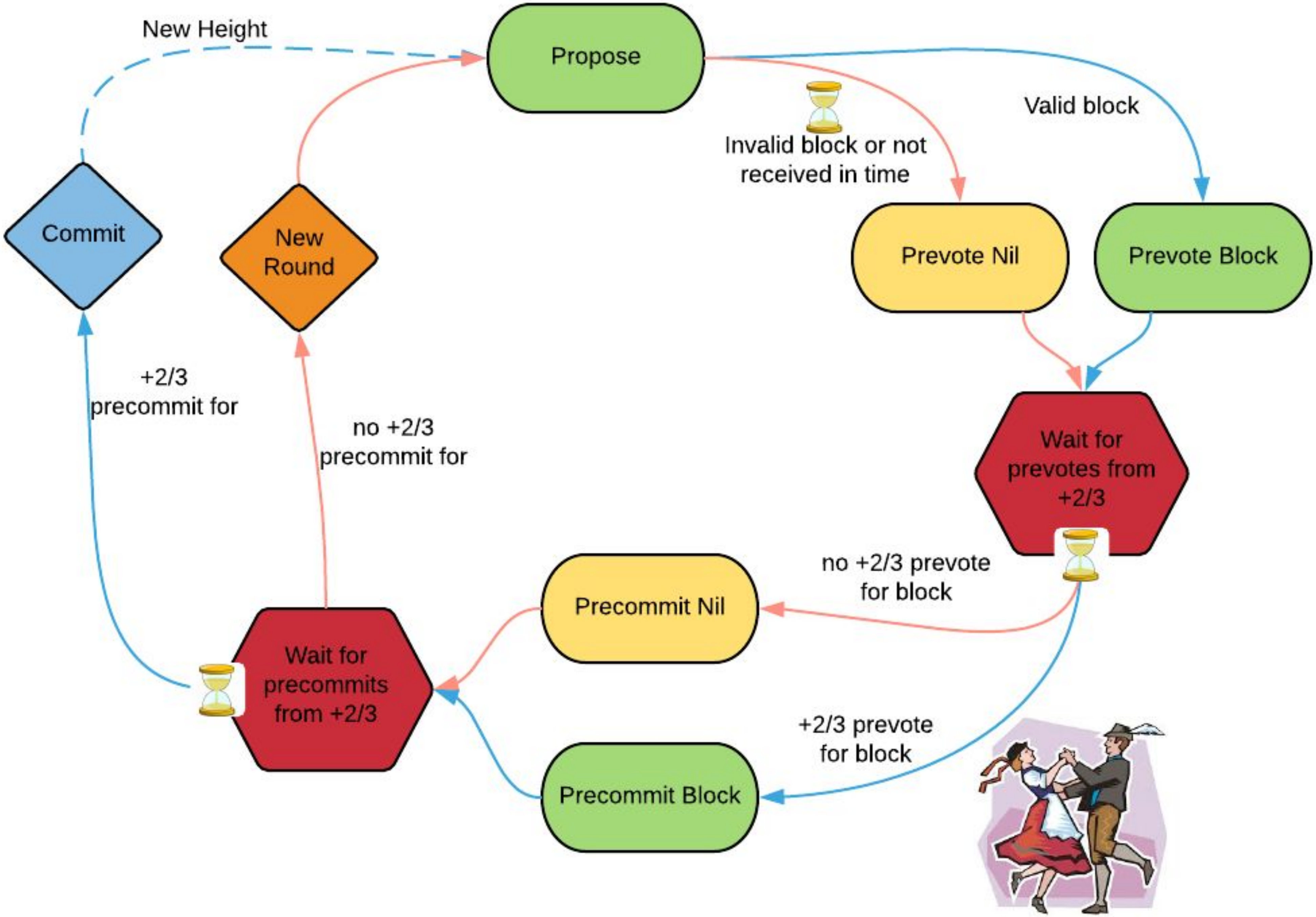
Introduction to **fault-tolerant** distributed algorithms

Verifying **synchronous** threshold-guarded algorithms

Verifying **asynchronous** threshold-guarded algorithms

Can we verify **Tendermint consensus?**

Tendermint consensus



Algorithm 1 Tendermint consensus algorithm

Initialization:
  $h_p := 0$ /* current height, or consensus instance we are currently executing */
  $round_p := 0$ /* current round number */
  $step_p \in \{propose, prevote, precommit\}$
  $decision_p[] := nil$
  $lockedValue_p := nil$
  $lockedRound_p := -1$
  $validValue_p := nil$
  $validRound_p := -1$

upon start **do** $StartRound(0)$

Function $StartRound(round)$ **:**
  $round_p \leftarrow round$
  $step_p \leftarrow propose$
  if $\mathsf{proposer}(h_p, round_p) = p$ **then**
    if $validValue_p \neq nil$ **then**
      $proposal \leftarrow validValue_p$
    else
      $proposal \leftarrow getValue()$
    broadcast $\langle \mathsf{PROPOSAL}, h_p, round_p, proposal, validRound_p \rangle$
  else
    schedule $OnTimeoutPropose(h_p, round_p)$ to be executed **after** $timeoutPropose(round_p)$

upon $\langle \mathsf{PROPOSAL}, h_p, round_p, v, -1 \rangle$ **from** $\mathsf{proposer}(h_p, round_p)$ **while** $step_p = propose$ **do**
  if $valid(v) \wedge (lockedRound_p = -1 \vee lockedValue_p = v)$ **then**
    broadcast $\langle \mathsf{PREVOTE}, h_p, round_p, id(v) \rangle$
  else
    broadcast $\langle \mathsf{PREVOTE}, h_p, round_p, nil \rangle$
  $step_p \leftarrow prevote$

upon $\langle \mathsf{PROPOSAL}, h_p, round_p, v, vr \rangle$ **from** $\mathsf{proposer}(h_p, round_p)$ **AND** $2f+1$ $\langle \mathsf{PREVOTE}, h_p, vr, id(v) \rangle$ **while** $step_p = propose \wedge (vr \geq 0 \wedge vr < round_p)$ **do**
  if $valid(v) \wedge (lockedRound_p \leq vr \vee lockedValue_p = v)$ **then**
    broadcast $\langle \mathsf{PREVOTE}, h_p, round_p, id(v) \rangle$
  else
    broadcast $\langle \mathsf{PREVOTE}, h_p, round_p, nil \rangle$
  $step_p \leftarrow prevote$

upon $2f+1$ $\langle \mathsf{PREVOTE}, h_p, round_p, * \rangle$ **while** $step_p = prevote$ for the first time **do**
  schedule $OnTimeoutPrevote(h_p, round_p)$ to be executed **after** $timeoutPrevote(round_p)$

upon $\langle \mathsf{PROPOSAL}, h_p, round_p, v, * \rangle$ **from** $\mathsf{proposer}(h_p, round_p)$ **AND** $2f+1$ $\langle \mathsf{PREVOTE}, h_p, round_p, id(v) \rangle$ **while** $valid(v) \wedge step_p \geq prevote$ for the first time **do**
  if $step_p = prevote$ **then**
    $lockedValue_p \leftarrow v$
    $lockedRound_p \leftarrow round_p$
    broadcast $\langle \mathsf{PRECOMMIT}, h_p, round_p, id(v)) \rangle$
    $step_p \leftarrow precommit$
  $validValue_p \leftarrow v$
  $validRound_p \leftarrow round_p$

upon $2f+1$ $\langle \mathsf{PREVOTE}, h_p, round_p, nil \rangle$ **while** $step_p = prevote$ **do**
  broadcast $\langle \mathsf{PRECOMMIT}, h_p, round_p, nil \rangle$
  $step_p \leftarrow precommit$

upon $2f+1$ $\langle \mathsf{PRECOMMIT}, h_p, round_p, * \rangle$ for the first time **do**
  schedule $OnTimeoutPrecommit(h_p, round_p)$ to be executed **after** $timeoutPrecommit(round_p)$

upon $\langle \mathsf{PROPOSAL}, h_p, r, v, * \rangle$ **from** $\mathsf{proposer}(h_p, r)$ **AND** $2f+1$ $\langle \mathsf{PRECOMMIT}, h_p, r, id(v) \rangle$ **while** $decision_p[h_p] = nil$ **do**
  if $valid(v)$ **then**
    $decision_p[h_p] = v$
    $h_p \leftarrow h_p + 1$
    reset $lockedRound_p$, $lockedValue_p$, $validRound_p$ and $validValue_p$ to initial values and empty message log
    $StartRound(0)$

upon $f+1$ $\langle *, h_p, round, *, * \rangle$ **with** $round > round_p$ **do**
  $StartRound(round)$

Function $OnTimeoutPropose(height, round)$ **:**
  if $height = h_p \wedge round = round_p \wedge step_p = propose$ **then**

Challenges for ByMC

Unbounded height of the blockchain

Unbounded number of rounds within one height

Rotating coordinator, breaking symmetry

Partial synchrony to guarantee liveness

Correct processes have more than 2/3 of voting power

Can we help?

I read that paper about **Byzantine Model Checker**

Model the algorithm as a threshold automaton

Verify safety and liveness for all $n, t, f : n > 3t \wedge t \geq f \geq 0$

I have heard this talk by Leslie Lamport

Let's write it in TLA$^+$

Run the **TLC model checker** for fixed parameters

TLC takes forever...

Run **APALACHE** for fixed parameters

Can we help?

I read that paper about **Byzantine Model Checker**

Model the algorithm as a threshold automaton

Verify safety and liveness for all $n, t, f : n > 3t \wedge t \geq f \geq 0$

I have heard this talk by Leslie Lamport

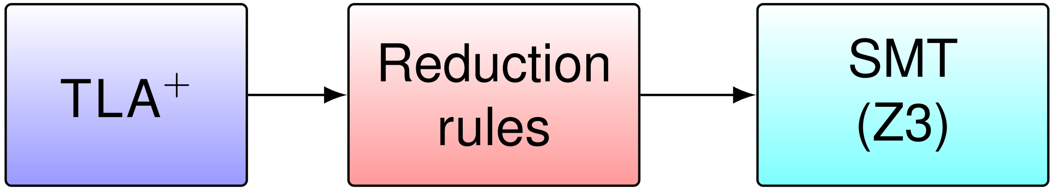
Let's write it in TLA$^+$

Run the **TLC model checker** for fixed parameters

TLC takes forever...

Run **APALACHE** for fixed parameters

Symbolic model checker for TLA$^+$ [OOPSLA'19]



Focus on distributed algorithms

- ☑ Invariants
- ☑ Inductive invariants
- ⊕ Fixed parameters, bounded executions
- ⊕ Fixed parameters

`[forsyte.at/research/apalache/]`

What we were doing in the last months...

Specifying several Tendermint protocols in TLA$^+$:

- fast synchronization
- light client
- consensus, tuned for fork detection

[github.com/interchainio/verification]

Medium DAILY DIGEST

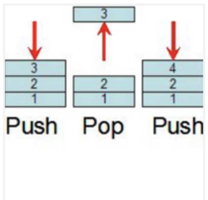
Stories for Igor Konnov

Today’s highlights

Functional Programming features in Scala

I’ve been exploring functional programming with Scala and its eco system for the past few months.

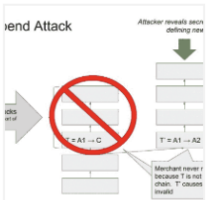
Kevin Lawrence in Towards Data Science ★ 6 min read



How to understand your program’s memory

When coding in a language like C or C++ you can interact with your memory in a more low-level way. Sometimes…

Tiago Antunes in freeCodeCamp.org 6 min read



Ethereum Classic (ETC) is currently being 51% attacked

On 1/5/2019, Coinbase detected a deep reorg of the Ethereum Classic blockchain that included a double spend…

Mark Nesbitt in The Coinbase Blog 7 min read

Fork accountability

Detect the peers that caused a fork — violation of agreement

Ran Apalache: 4 peers, 2 faults, fault threshold is 1:

- ✓ found **equivocation**, 2 hours
- ✓ found **amnesia**, 2 hours
- ⊕ no other scenarios up to 15 steps, 7 CPU cores, 6.5 hours

---

Proving that no other scenarios exist? . . . for all parameters?

Fork accountability

Detect the peers that caused a fork — violation of agreement

Ran Apalache: 4 peers, 2 faults, fault threshold is 1:

- ✓ found **equivocation**, 2 hours
- ✓ found **amnesia**, 2 hours
- ➕ no other scenarios up to 15 steps, 7 CPU cores, 6.5 hours

---

Proving that no other scenarios exist? ... for all parameters?

Conclusions

Reasoning about fault-tolerant algorithms is hard

. . . but fun!

Practical algorithms are even harder

Threshold guards are everywhere

Specialized tools for narrow classes, e.g., ByMC

vs.

General tools for broader classes, e.g., Apalache

Future

Supporting as many features as in TLC

TLA$^+$ users specify industrial-scale distributed protocols

all kinds of Paxos, Raft, key-value stores, group membership

These are large and complex specifications [Newcombe et al.'14]

Amazon used 80 CPU cores to find a trace of 35 steps

Semi-automated techniques that would get help from the user

Reduction arguments, abstractions, etc.