Synchronizer: a recipe for building correct algorithms under partial synchrony

Alexey Gotsman

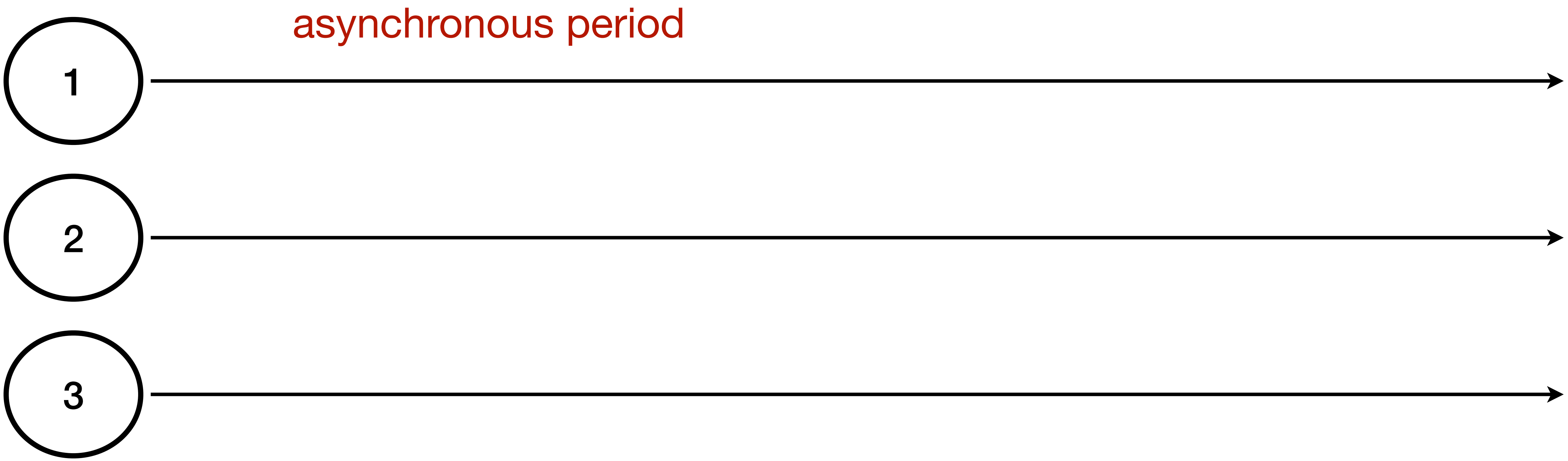
IMDEA Software Institute, Madrid, Spain

Joint work with Manuel Bravo (Informal Systems), Gregory Chockler (University of Surrey), and Alejandro Naser Pastoriza (IMDEA)

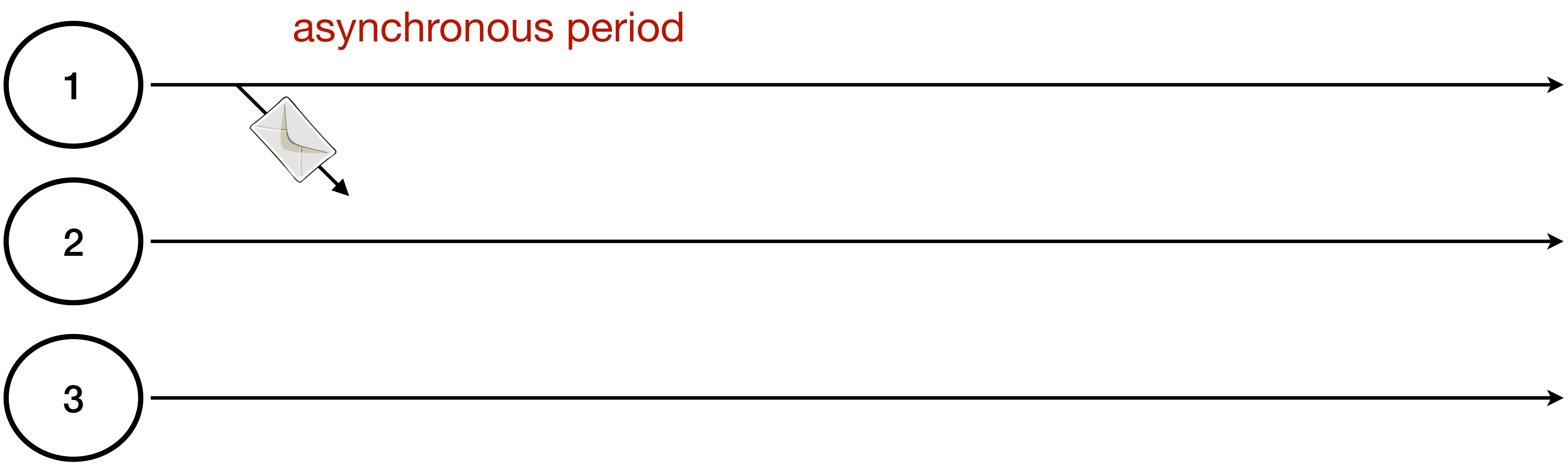
Fault-tolerant distributed computing

- Many distributed computing problems are unsolvable under asynchrony and failures
- Consensus and state-machine replication [FLP85]
- Compromise: provide safety always and liveness only under synchrony

Partial synchrony [DLS88]

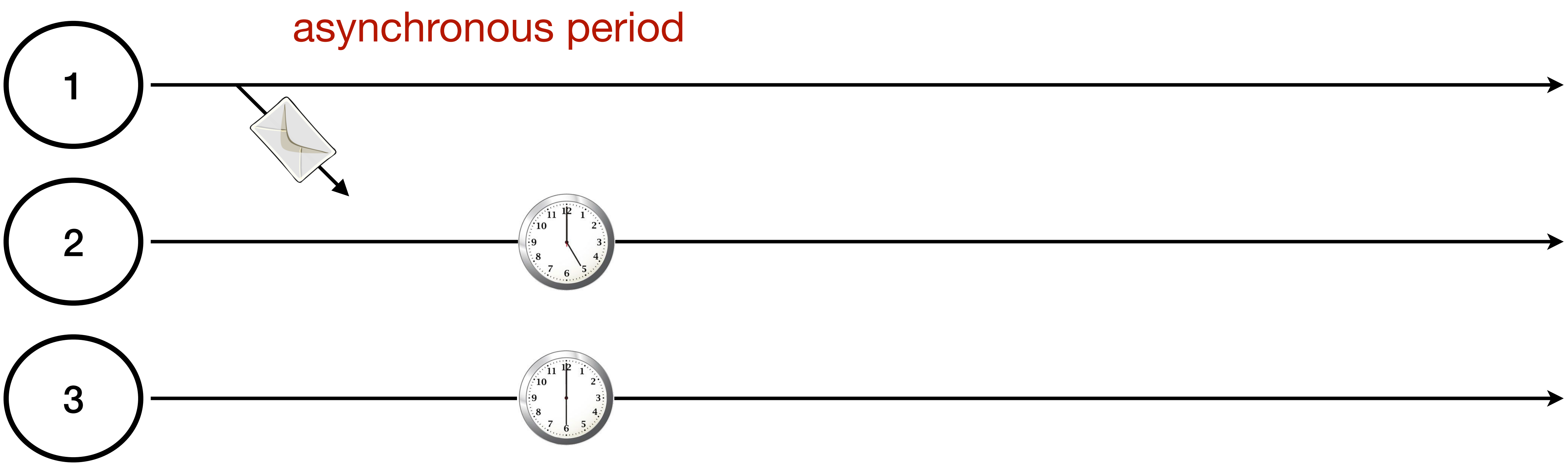


Partial synchrony [DLS88]



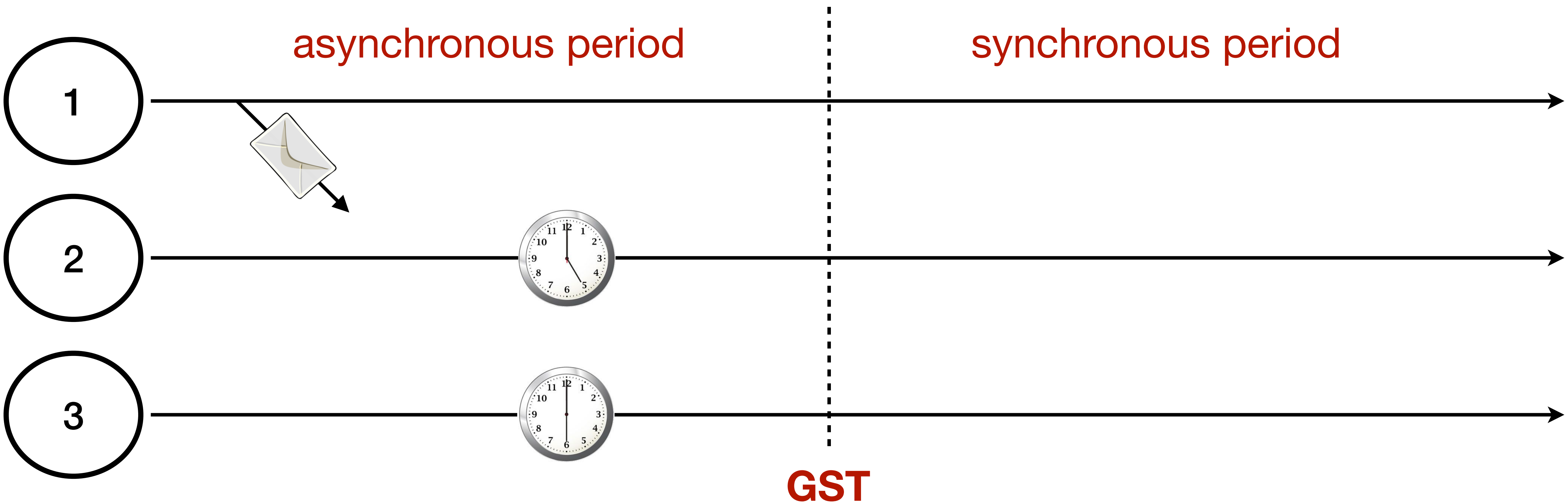
- Messages delayed or lost

Partial synchrony [DLS88]



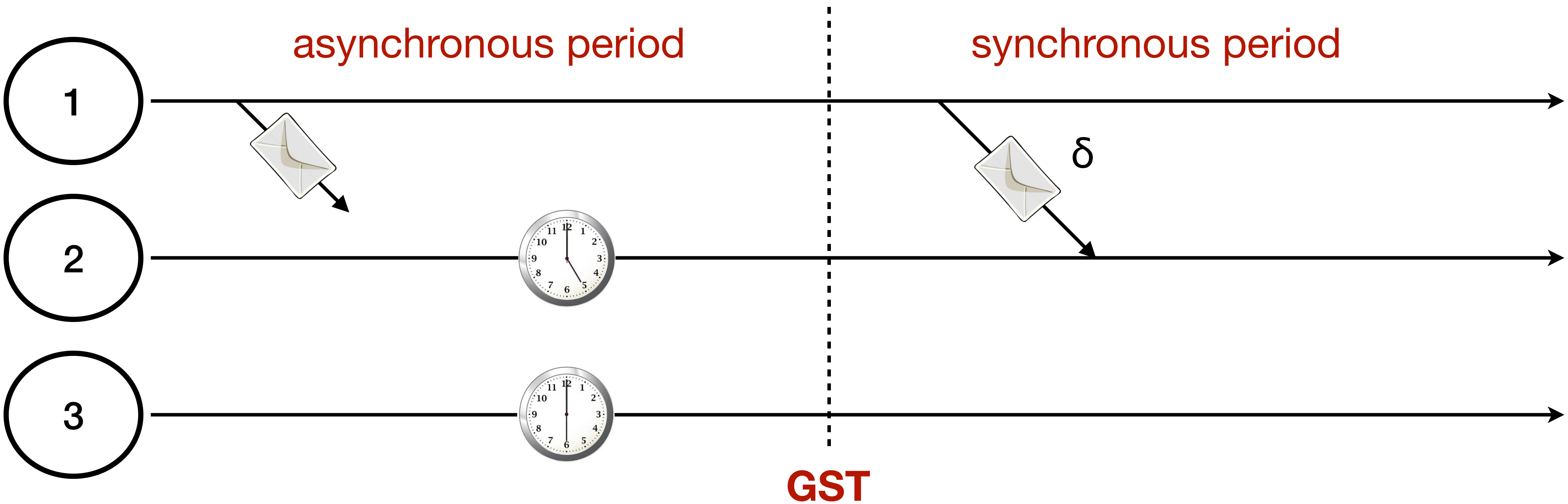
- Messages delayed or lost
- Process clocks out of sync

Partial synchrony [DLS88]



- Messages delayed or lost
- Process clocks out of sync

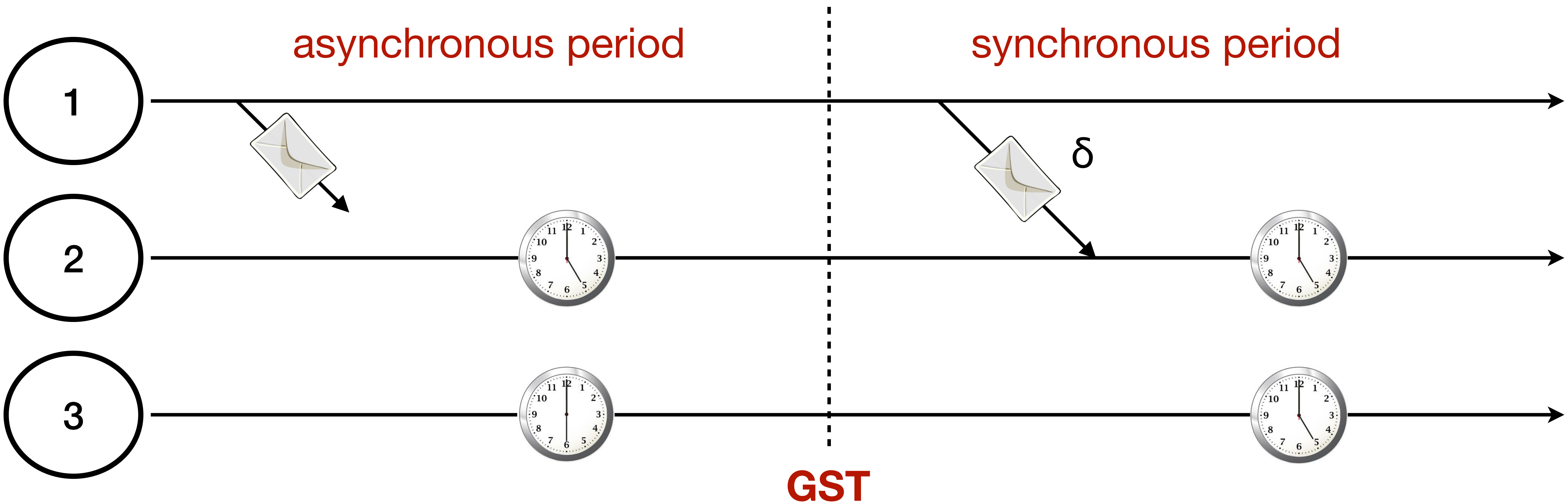
Partial synchrony [DLS88]



- Messages delayed or lost
- Process clocks out of sync

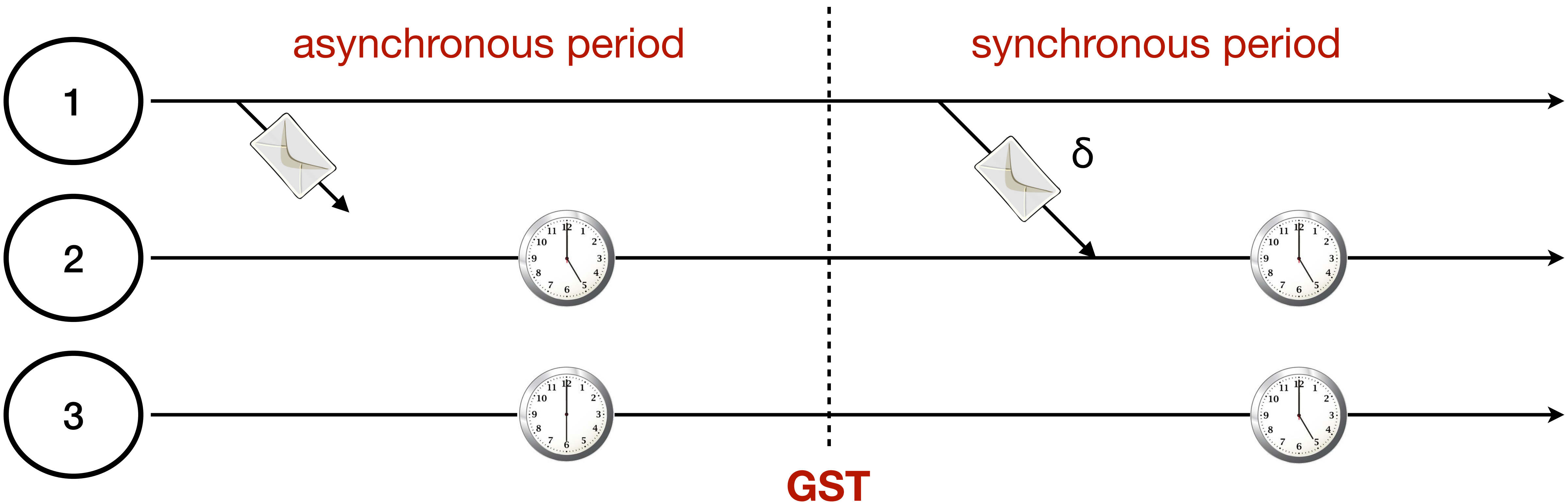
- Messages through correct channels delivered within an unknown time δ

Partial synchrony [DLS88]



- Messages delayed or lost
- Process clocks out of sync
- Messages through correct channels delivered within an unknown time δ
- Process clocks track real time

Partial synchrony [DLS88]



- Messages delayed or lost
- Process clocks out of sync

- Messages through correct channels delivered within an unknown time δ
- Process clocks track real time

Byzantine or crash failures

It ain't simple

[SOSP'07, Best Paper Award]

Zyzzyva: Speculative Byzantine Fault Tolerance

Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong
Dept. of Computer Sciences
University of Texas at Austin
{kotla,lorenzo,dahlin,aclement,elwong}@cs.utexas.edu

ABSTRACT

We present Zyzzyva, a protocol that uses speculation to reduce the cost and simplify the design of Byzantine fault tolerant state machine replication. In Zyzzyva, replicas respond to a client's request without first running an expensive three-phase commit protocol to reach agreement on the order in which the request must be processed. Instead, they optimistically adopt the order proposed by the primary and respond immediately to the client. Replicas can thus become temporarily inconsistent with one another, but clients detect inconsistencies, help correct replicas converge on a single total ordering of requests, and only rely on responses that are consistent with this total order. This approach allows Zyzzyva to reduce replication overheads to near their theoretical minima.

Categories and Subject Descriptors

D.4.5 [**Operating Systems**]: Reliability—*Fault-tolerance*; D.4.7 [**Operating Systems**]: Organization and Design—*Distributed systems*; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Distributed systems*

General Terms

Performance, Reliability

non-fail-stop behavior in real systems [2, 5, 6, 27, 30, 32, 36, 39, 40] suggest that BFT may yield significant benefits even without resorting to n-version programming [4, 15, 33]. Third, improvements to the state of the art in BFT replication techniques [3, 9, 10, 18, 33, 41] make BFT replication increasingly practical by narrowing the gap between BFT replication costs and costs already being paid for non-BFT replication. For example, by default, the Google file system uses 3-way replication of storage, which is roughly the cost of BFT replication for $f = 1$ failures with 4 agreement nodes and 3 execution nodes [41].

This paper presents Zyzzyva[1], a new protocol that uses *speculation* to reduce the cost and simplify the design of BFT state machine replication [19, 35]. Like traditional state machine replication protocols [9, 33, 41], a primary proposes an order on client requests to the other replicas. In Zyzzyva, unlike in traditional protocols, replicas speculatively execute requests without running an expensive agreement protocol to definitively establish the order. As a result, correct replicas' states may diverge, and replicas may send different responses to clients. Nonetheless, applications at clients observe the traditional and powerful abstraction of a replicated state machine that executes requests in a linearizable [13] order because replies carry with them sufficient history information for clients to determine if the replies and history are *stable* and guaranteed to be eventually committed. If a speculative reply and history are stable, the client uses the

It ain't simple

[SOSP'07, Best Paper Award]

Zyzzyva: Speculative Byzantine Fault Tolerance

Revisiting Fast Practical Byzantine Fault Tolerance

Ittai Abraham, Guy Gueta, Dahlia Malkhi
VMware Research

with:
Lorenzo Alvisi (Cornell),
Rama Kotla (Amazon),
Jean-Philippe Martin (Verily)

December 6, 2017

Abstract

In this note, we observe a safety violation in Zyzzyva [7, 9, 8] and a liveness violation in FaB [14, 15]. To demonstrate these issues, we require relatively simple scenarios, involving only four replicas, and one or two view changes. In all of them, the problem is manifested already in the first log slot.

1 Introduction

A landmark solution in achieving replication with Byzantine fault tolerance has been the Practical Byzantine Fault Tolerance (PBFT) work by Castro and Liskov [3, 4]. Since the PBFT publication, there has been a stream of works aiming to improve the efficiency of PBFT protocols. One strand of these works revolves

It ain't simple

[SOSP'07, Best Paper Award]

Zyzzyva: Speculative Byzantine Fault Tolerance

ABS
We p
duce
tolera
spond
three-
der in
optim
respo
come
detec
single
that a
lows
theor

Cate
D.4.5
D.4.7
Distr
triev

Gen
Perfo

Revisiting Fast Practical Byzantine Fault Tolerance

Ittai Abraham, Guy Gueta, Dahlia Malkhi
VMware Research

with:
Lorenzo Alvisi (Cornell),
Rama Kotla (Amazon),
Jean-Philippe Martin (Verily)

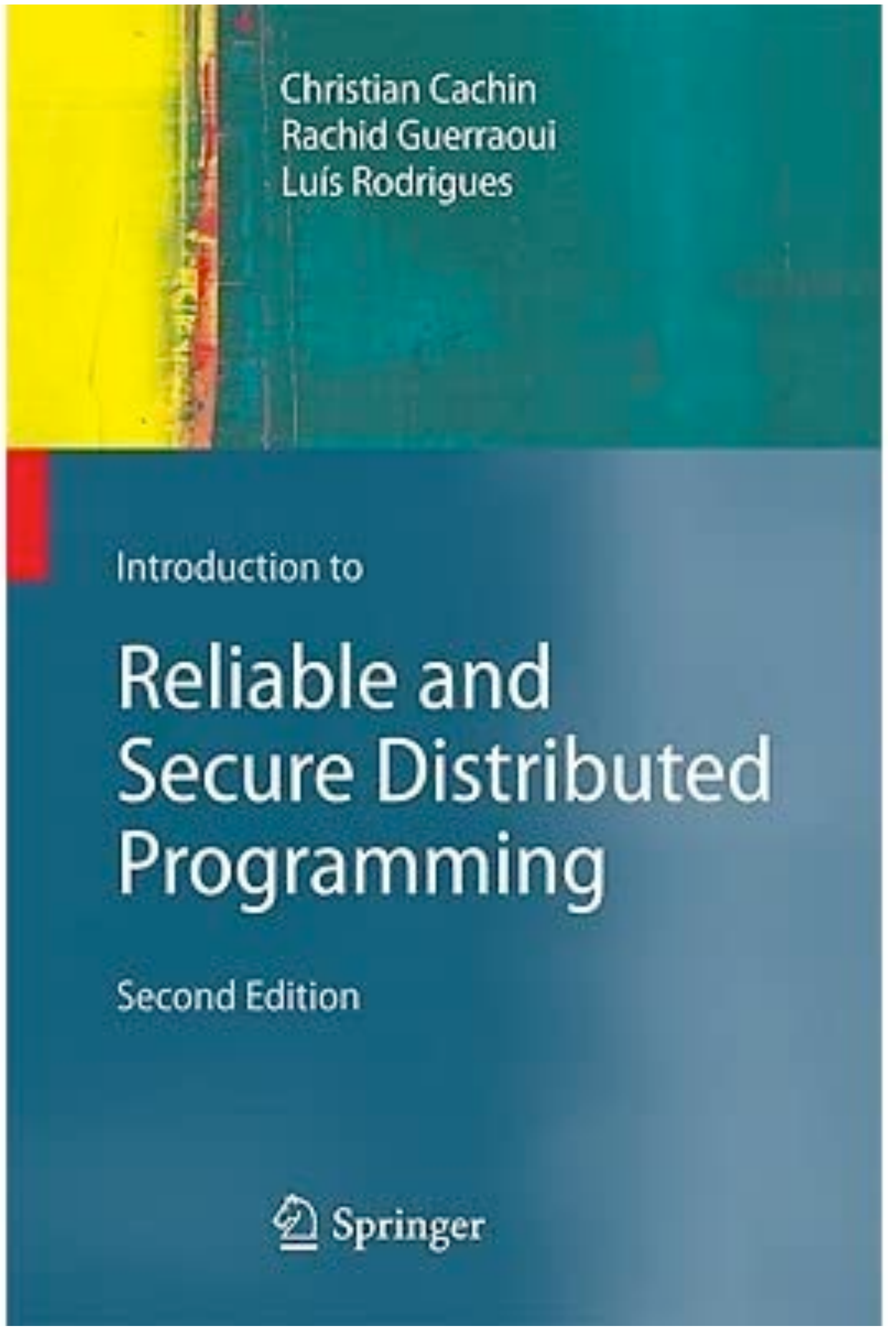
Formal Verification of Blockchain Byzantine Fault Tolerance

Tholoniat & Gramoli, FRIDA'19

Table 1: Consensus algorithms that experienced liveness or safety limitations

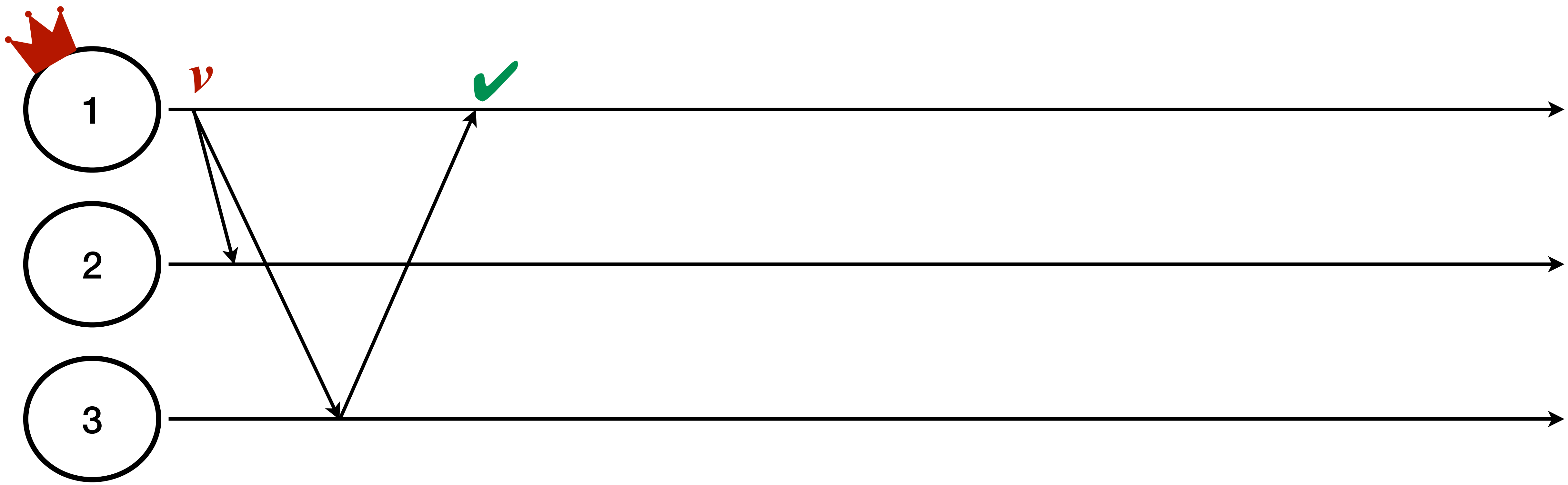
Algorithms	Ref.	Limitation	Counter-example	Alternative	Blockchain
Randomized consensus	[41]	liveness	[new]	[42]	HoneyBadger [40]
Casper	[13]	liveness	[new]	[52]	Ethereum v2.0 [26]
Ripple consensus	[47]	safety	[5]	[18]	xRapid [11]
Tendermint consensus	[12]	safety	[4]	[3]	Tendermint [36]
Zyzzyva	[35]	safety	[1]	[6]	SBFT [27]
IBFT	[38]	liveness	[46]	[46]	Quorum [19]

It ain't simple



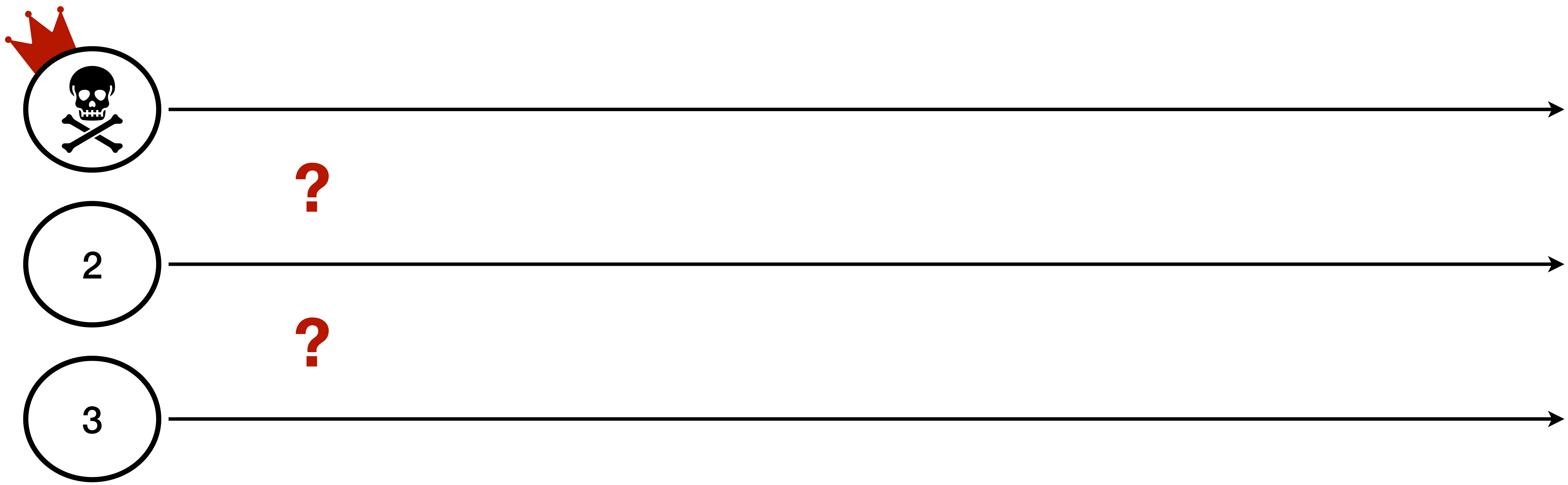
- Byzantine Consensus protocol via a modular decomposition
- Found a liveness bug
- Fixable, but the intermediate abstractions will remain broken: too strong to be implementable

Leader-driven consensus



- The leader proposes a value to vote on
- The processes can vote to accept the value
- Consensus is reached when enough processes vote to accept the value

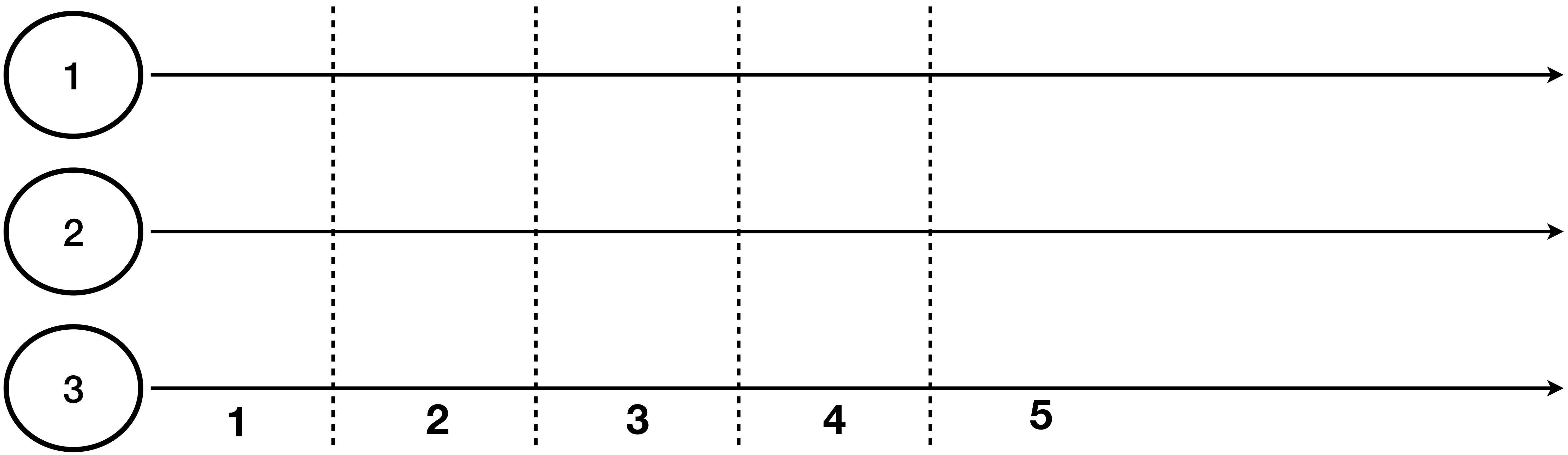
Failed votes



- Votes may fail: faulty leader or asynchronous network

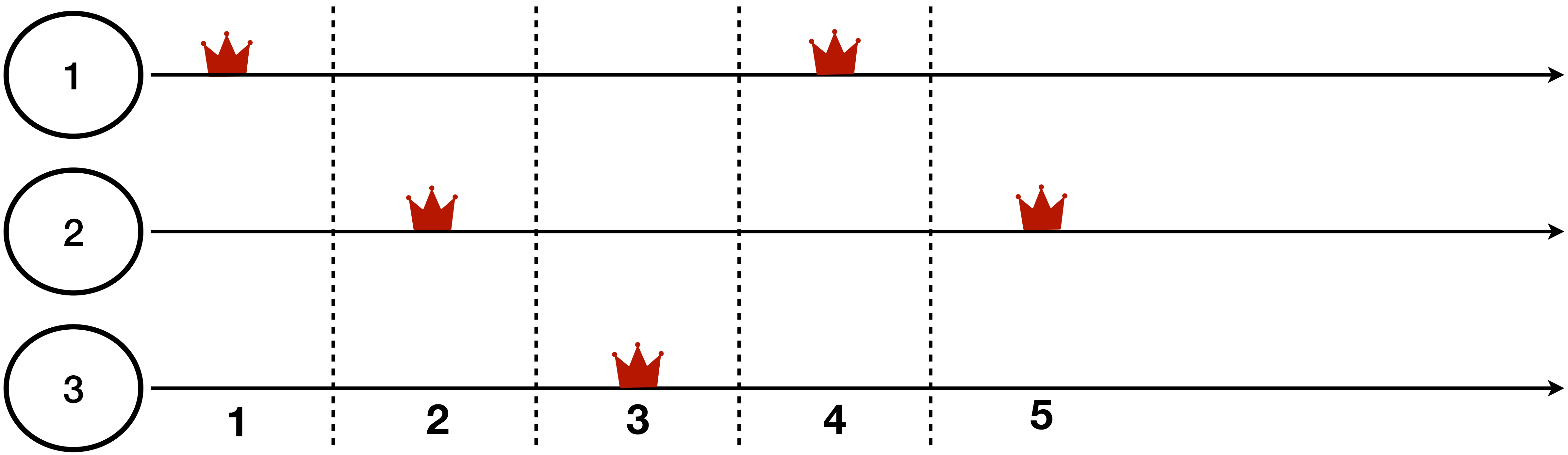
- May need to change leaders

Views



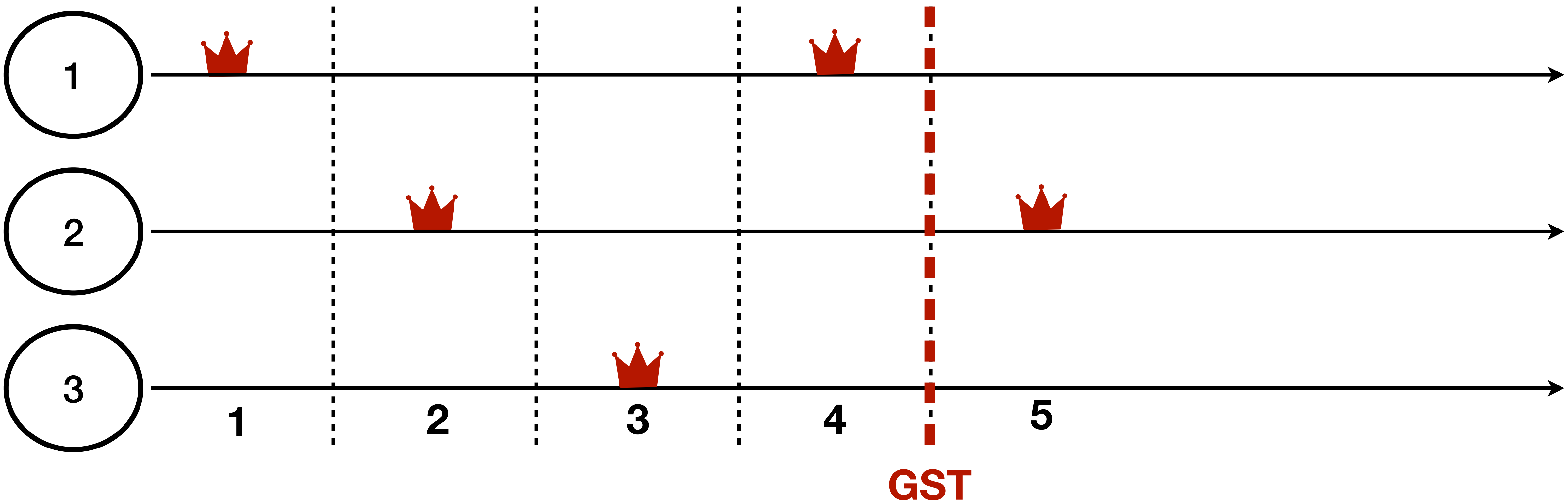
- Divide the execution into views (aka rounds), each with a fixed leader: view mod n

Views



- Divide the execution into views (aka rounds), each with a fixed leader: view mod n

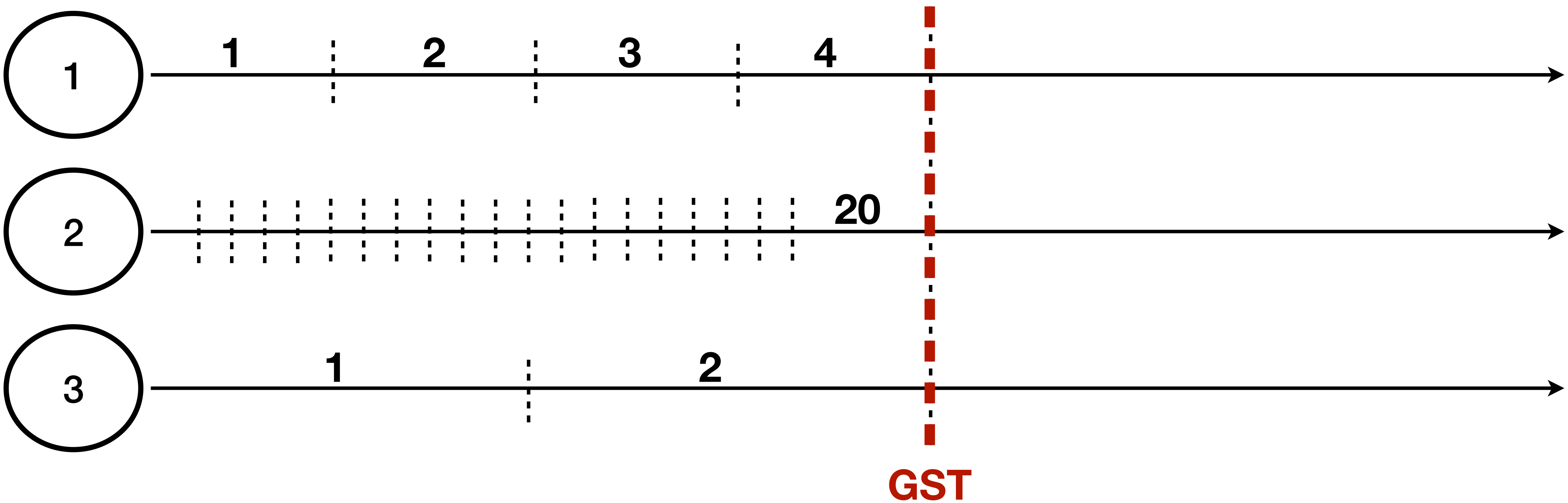
Views



- Divide the execution into views (aka rounds), each with a fixed leader: view mod n

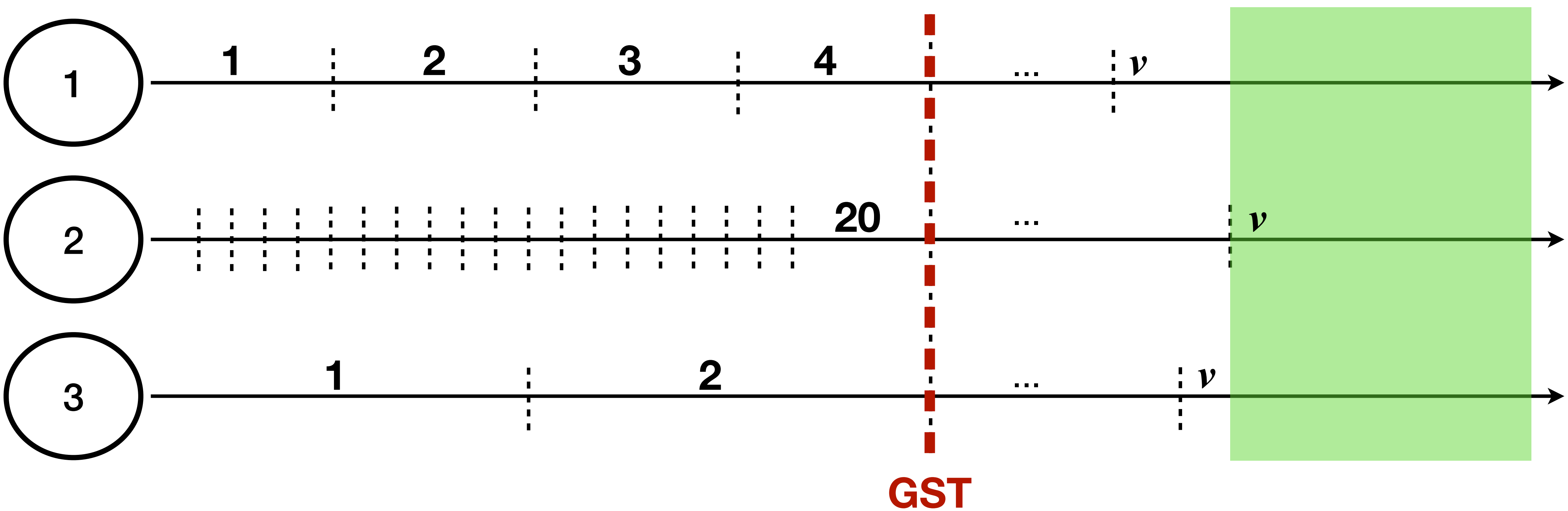
- Will hit a good leader after GST $\Longrightarrow$ will decide

View synchronization



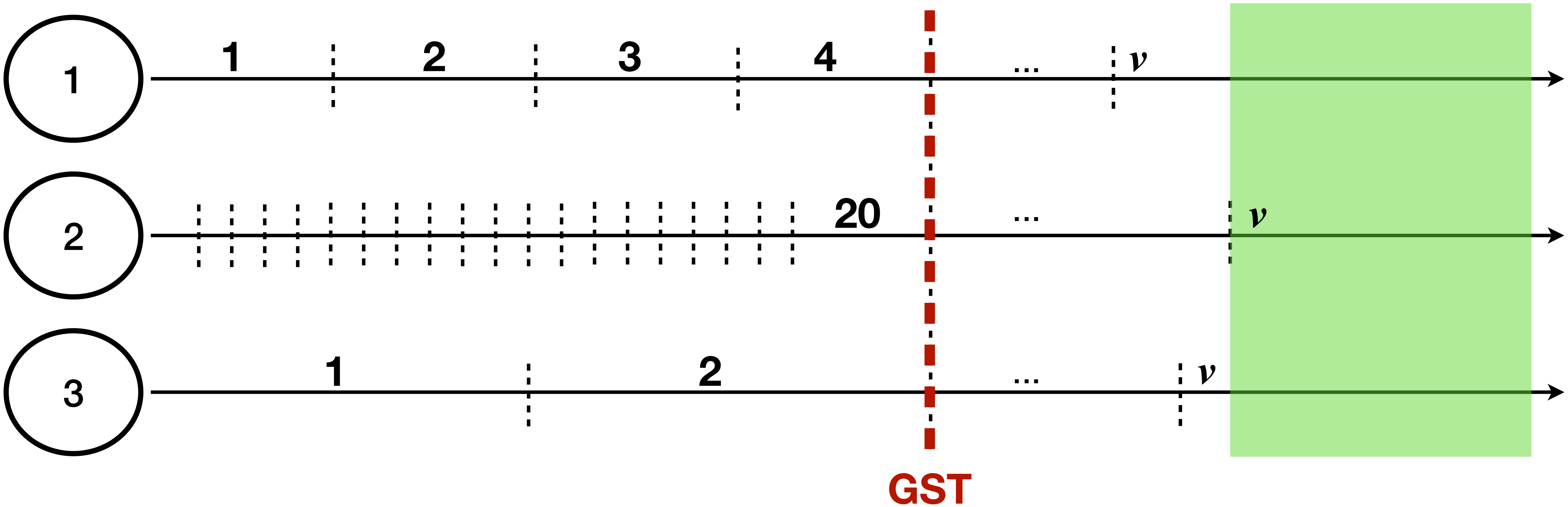
- Before GST: clocks out of sync, messages delayed or lost

View synchronization



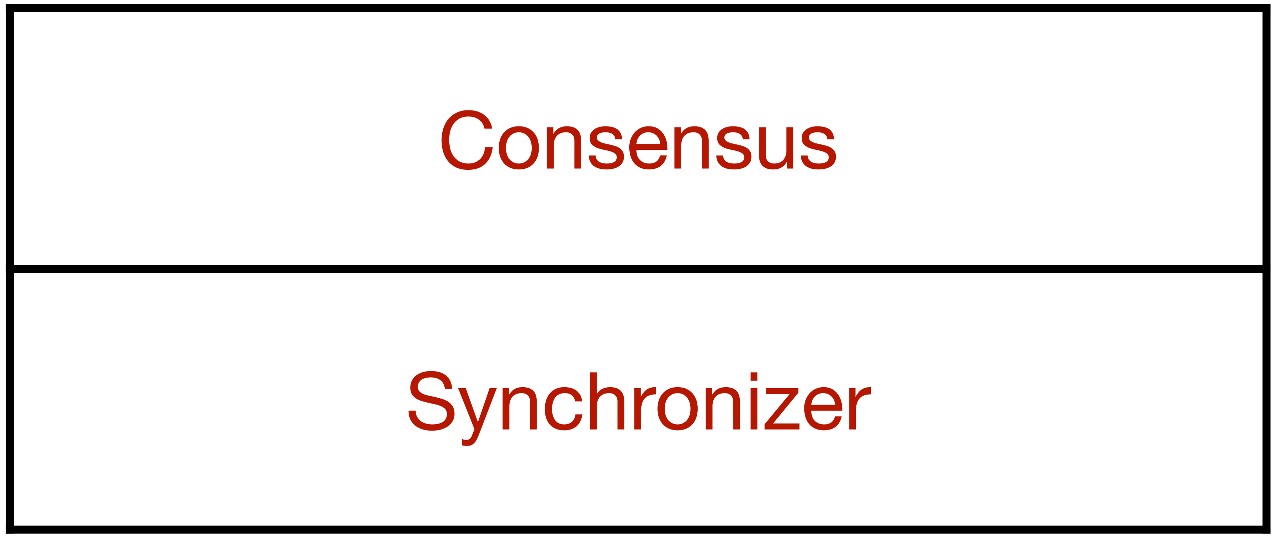
- Before GST: clocks out of sync, messages delayed or lost
- After GST: need to bring all non-faulty processes into the same view

View synchronization



- Before GST: clocks out of sync, messages delayed or lost
- After GST: need to bring all non-faulty processes into the same view
- Integrating the liveness mechanisms complicates the protocol

Synchronizer



- Synchronizer tells the processes when to switch views [DLS88], [HotStuff, PODC'19], [Naor&Keidar, DISC'20]...
- Reused across different protocols $\Longrightarrow$ more systematic design, modular proofs
- White-box optimizations can be done for each protocol

Our contributions

- Precise specification of the synchronizer abstraction
- Synchronizer implementations under partial synchrony
- Case studies of implementing live consensus and state-machine replication

Our contributions

- Precise specification of the synchronizer abstraction
- Synchronizer implementations under partial synchrony
- Case studies of implementing live consensus and state-machine replication
- Different failure models:
  - Byzantine faults [DISC'20, DISC'22]
  - Crash faults and classical partial synchrony model
  - Crash faults and intermitted connectivity [arxiv]

Crashes ain't simple either

[OSDI'18]

An Analysis of Network-Partitioning Failures in Cloud Systems

Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta, Samer Al-Kiswany
University of Waterloo, Canada

Abstract

We present a comprehensive study of 136 system failures attributed to network-partitioning faults from 25 widely used distributed systems. We found that the majority of the failures led to catastrophic effects, such as data loss, reappearance of deleted data, broken locks, and system crashes. The majority of the failures can easily manifest once a network partition occurs: They require little to no client input, can be triggered by isolating a single node, and are deterministic. However, the number of test cases that one must consider is extremely large. Fortunately, we identify ordering, timing, and network fault characteristics that significantly simplify testing. Furthermore, we found that a significant number of the failures are due to design flaws in core system mechanisms.

We found that the majority of the failures could have been avoided by design reviews, and could have been discovered by testing with network-partitioning fault injection. We built NEAT, a testing framework that simplifies the coordination of multiple clients and can inject different types of network-partitioning faults

production networks, network-partitioning faults occur as frequently as once a week and take from tens of minutes to hours to repair.

Given that network-partitioning fault tolerance is a well-studied problem [13, 14, 17, 20], this raises questions about *how these faults sill lead to system failures. What is the impact of these failures? What are the characteristics of the sequence of events that lead to a system failure? What are the characteristics of the network-partitioning faults? And, foremost, how can we improve system resilience to these faults?*

To help answer these questions, we conducted a thorough study of 136 network-partitioning failures[1] from 25 widely used distributed systems. The systems we selected are popular and diverse, including key-value systems and databases (MongoDB, VoltDB, Redis, Riak, RethinkDB, HBase, Aerospike, Cassandra, Geode, Infinispan, and Ignite), file systems (HDFS and MooseFS), an object store (Ceph), a coordination service (ZooKeeper), messaging systems (Kafka, ActiveMQ, and RabbitMQ), a data-processing framework (Hadoop MapReduce), a search engine

Crashes ain't simple either

[OSDI'18]

An Analysis of Network-Partitioning Failures in Cloud Systems

Ahmed Alquraan, Hatem Takruri, Mohammed A
University of Waterloo, Ca

Abstract

We present a comprehensive study of 136 system failures attributed to network-partitioning faults from 25 widely used distributed systems. We found that the majority of the failures led to catastrophic effects, such as data loss, reappearance of deleted data, broken locks, and system crashes. The majority of the failures can easily manifest once a network partition occurs: They require little to no client input, can be triggered by isolating a single node, and are deterministic. However, the number of test cases that one must consider is extremely large. Fortunately, we identify ordering, timing, and network fault characteristics that significantly simplify testing. Furthermore, we found that a significant number of the failures are due to design flaws in core system mechanisms.

We found that the majority of the failures could have been avoided by design reviews, and could have been discovered by testing with network-partitioning fault injection. We built NEAT, a testing framework that simplifies the coordination of multiple clients and can inject different types of network-partitioning faults

production as frequen minutes to

Given well-studie questions *failures. W the charac a system f network-pa improve sy*

To hel thorough from 25 w we selecte value syst Redis, Rial Geode, Inf MooseFS), service (ActiveMQ framework (Hadoop MapReduce), a search engine

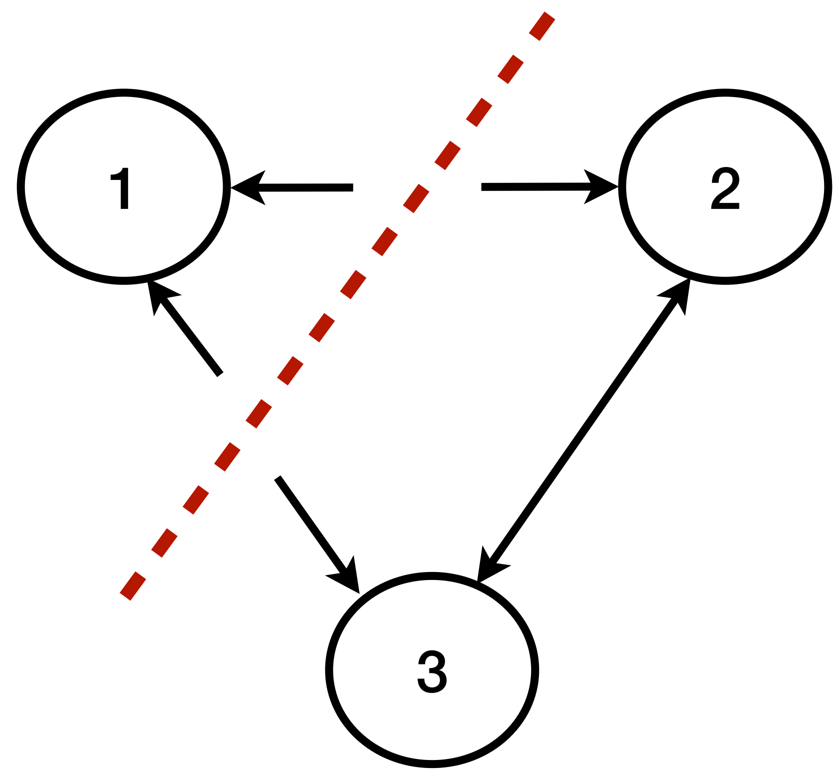
Table 1. List of studied system. The table shows systems' consistency model, number of failures, and number of catastrophic failures. Highlighted rows indicate systems we tested using NEAT, and the number of failures we found.

System	Consistency Model	Failures	
		Total	Catastrophic
MongoDB [31]	Strong	19	11
VoltDB [33]	Strong	4	4
RethinkDB [52]	Strong	3	3
HBase [56]	Strong	5	3
Riak [57]	Strong/Eventual	1	1
Cassandra [58]	Strong	4	4
Aerospike [59]	Eventual	3	3
Geode [60]	Strong	2	2
Redis [32]	Eventual	3	2
Hazelcast [29]	Best Effort	7	5
Elasticsearch [28]	Eventual	22	21
ZooKeeper [61]	Strong	3	3
HDFS [1]	Custom	4	2
Kafka [30]	-	5	3
RabbitMQ [62]	-	7	4
MapReduce [4]	-	6	2
Chronos [63]	-	2	1
Mesos [64]	-	4	0
Infinispan [42]	Strong	1	1
Ignite [39]	Strong	15	13
Terracotta [40]	Strong	9	9
Ceph [37]	Strong	2	2
MooseFS [43]	Eventual	2	2
ActiveMQ [38]	-	2	2
DKron [41]	-	1	1
Total	-	**136**	**104**

CAP theorem

Can't get all of:

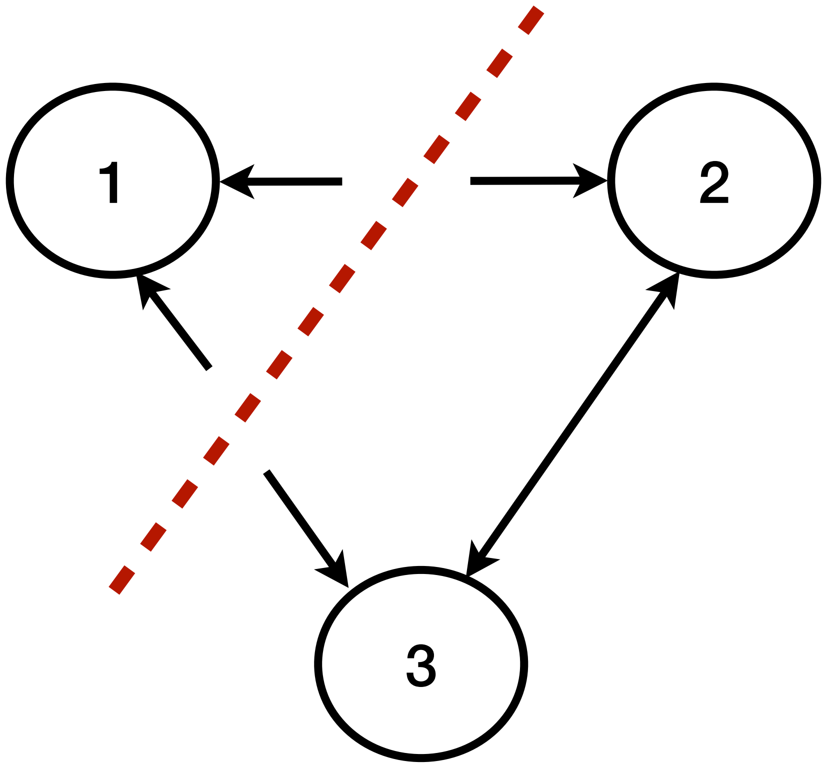
- strong **C**onsistency
- **A**vailability
- **P**artition-tolerance



CAP theorem

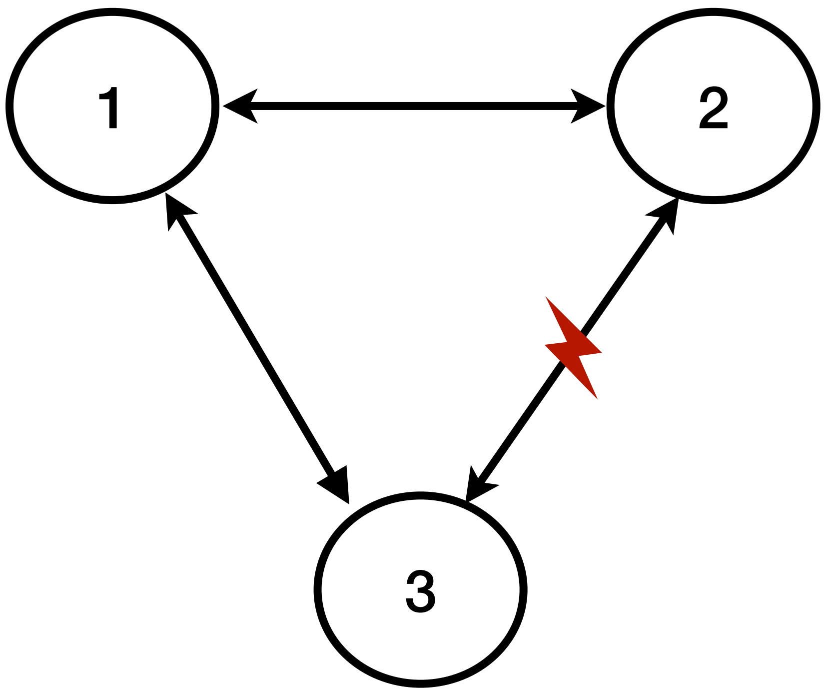
Can't get all of:

- strong **C**onsistency
- **A**vailability
- **P**artition-tolerance



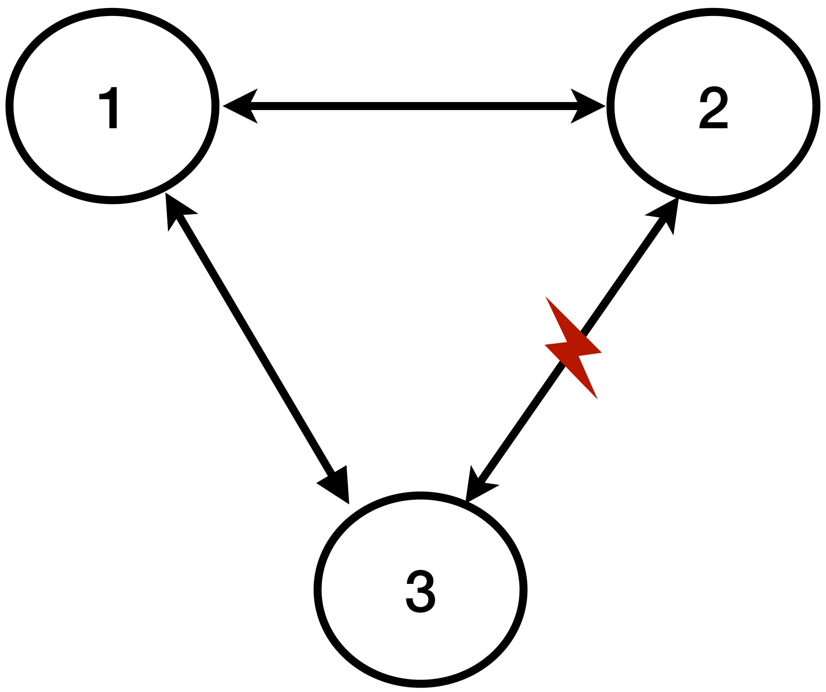
Doesn't preclude availability in parts of the system:
can run Paxos at the majority side of a partition

Types of network failures

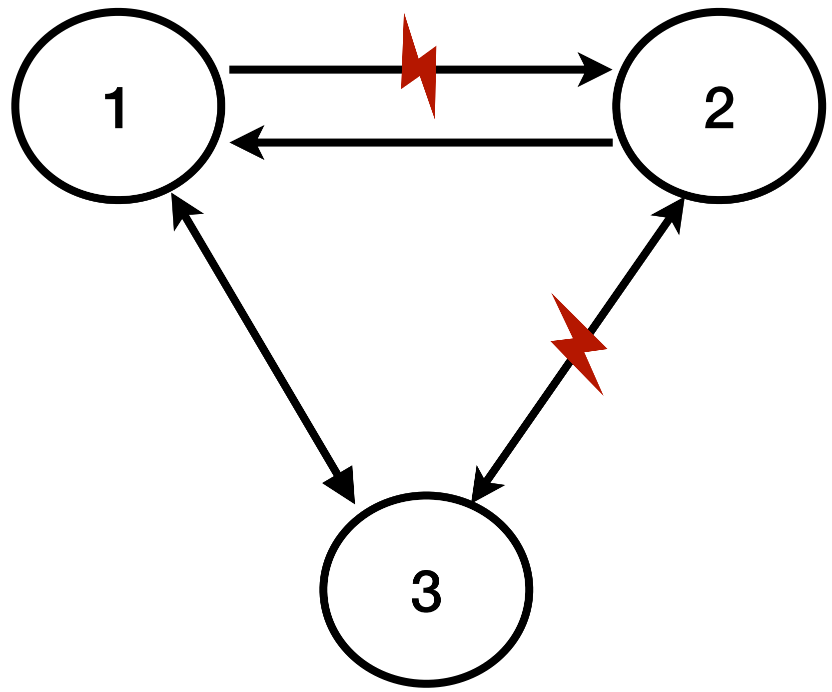


indirect connectivity
/ partial partitions

Types of network failures

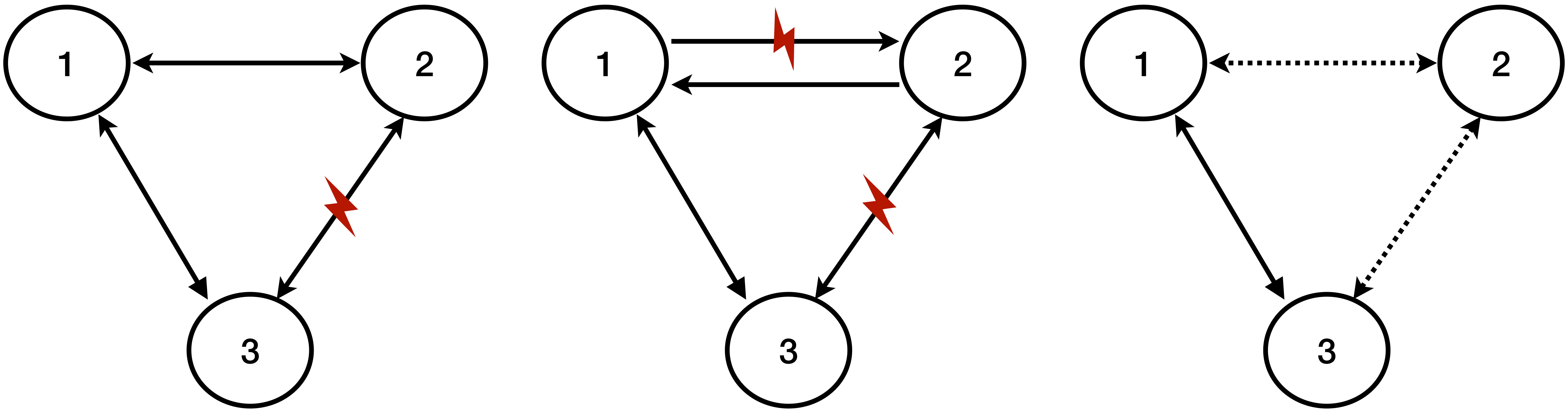


indirect connectivity
/ partial partitions



asymmetric connectivity

Types of network failures

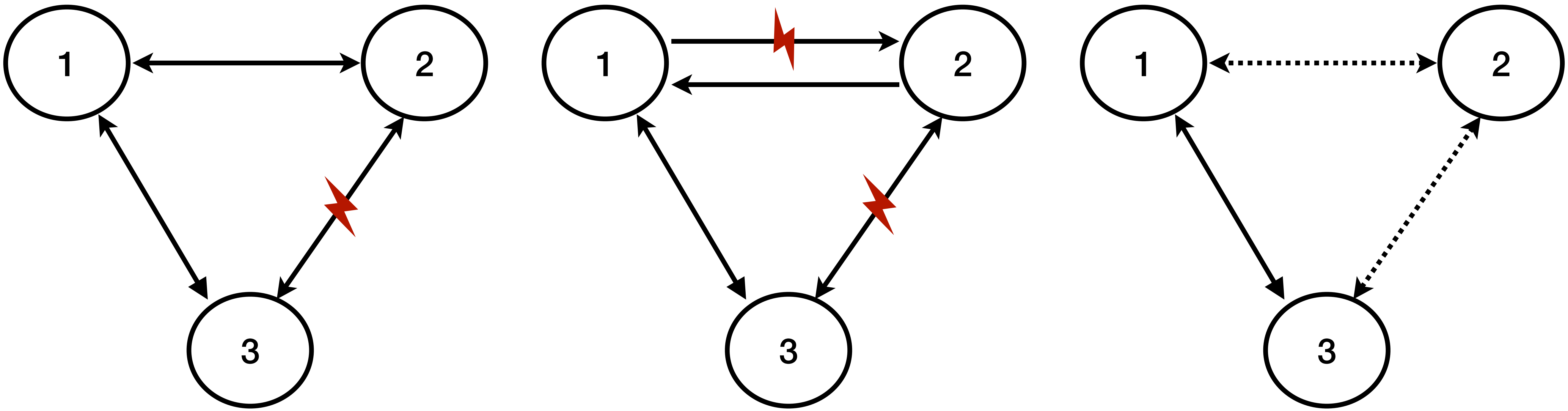


indirect connectivity / partial partitions

asymmetric connectivity

intermittent connectivity

Types of network failures



indirect connectivity / partial partitions

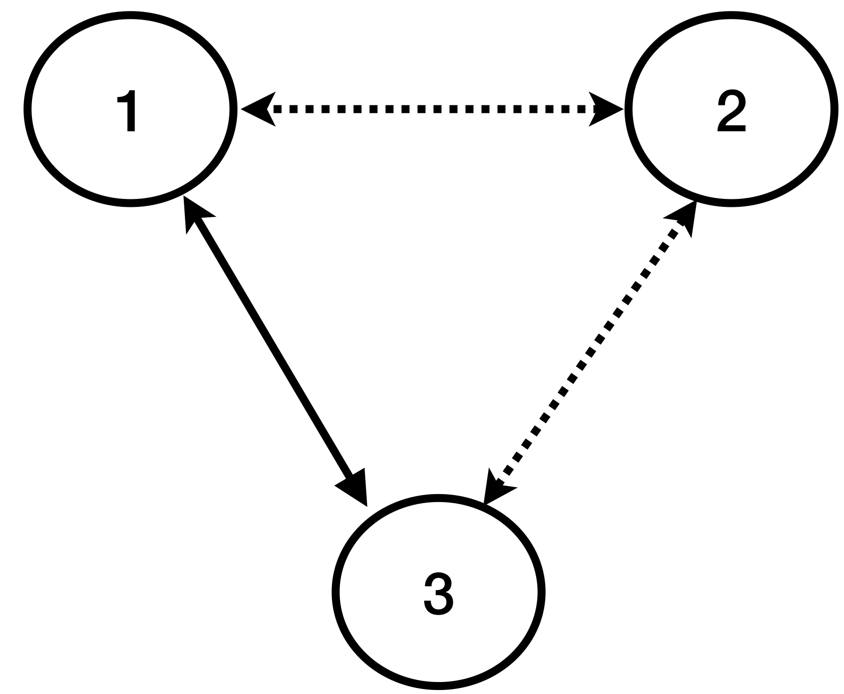
asymmetric connectivity

intermittent connectivity

Often consequence of Byzantine router failures

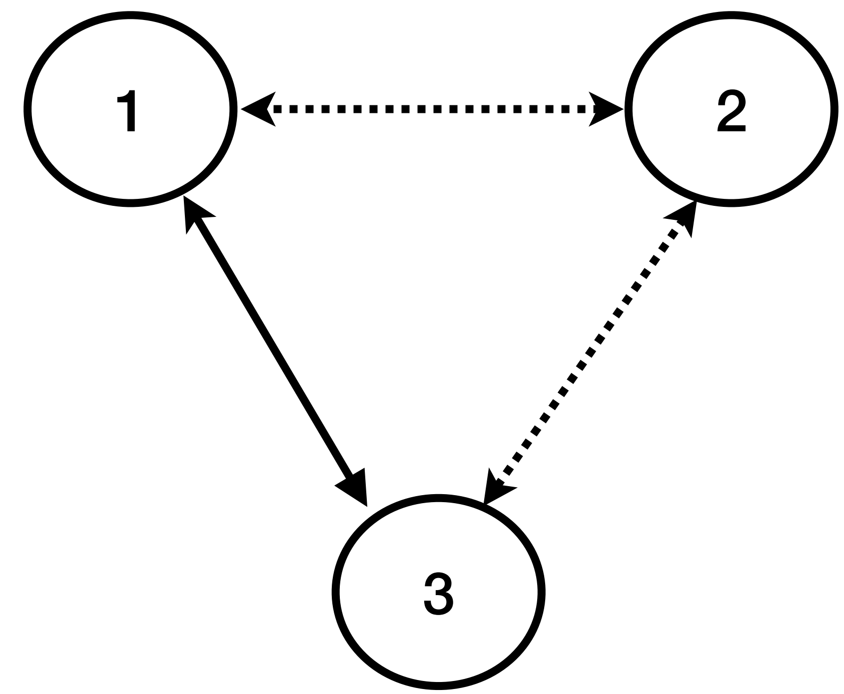
Flaky channels

- Can drop an arbitrary subset of messages sent through them



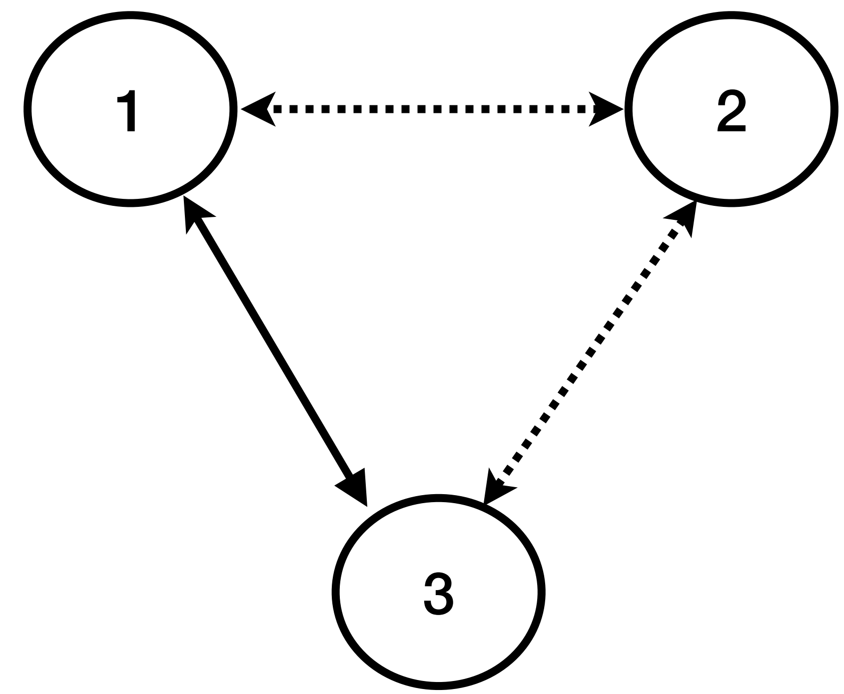
Flaky channels

- Can drop an arbitrary subset of messages sent through them
- Capture indirect, asymmetric and intermittent connectivity, selective omission...



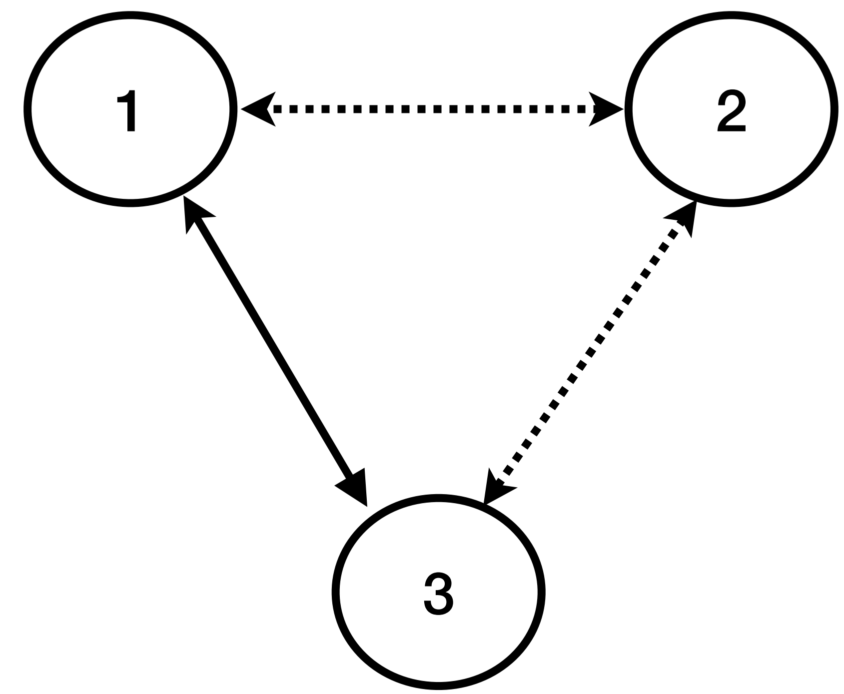
Flaky channels

- Can drop an arbitrary subset of messages sent through them
- Capture indirect, asymmetric and intermittent connectivity, selective omission...
- Flaky channels strictly weaker than fair-lossy ones



Failure detectors don’t work

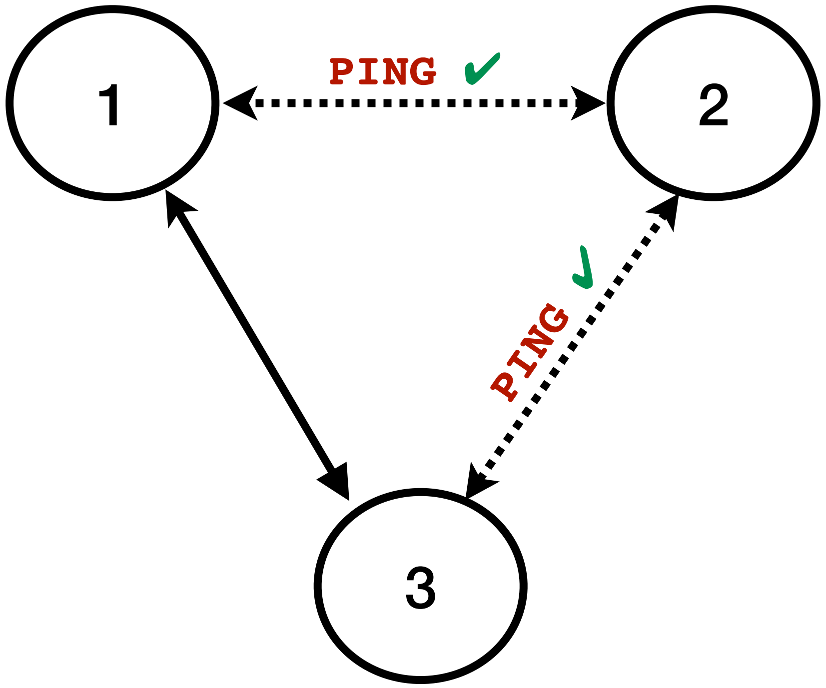
- Can't implement consensus by first implementing Ω



Failure detectors don’t work

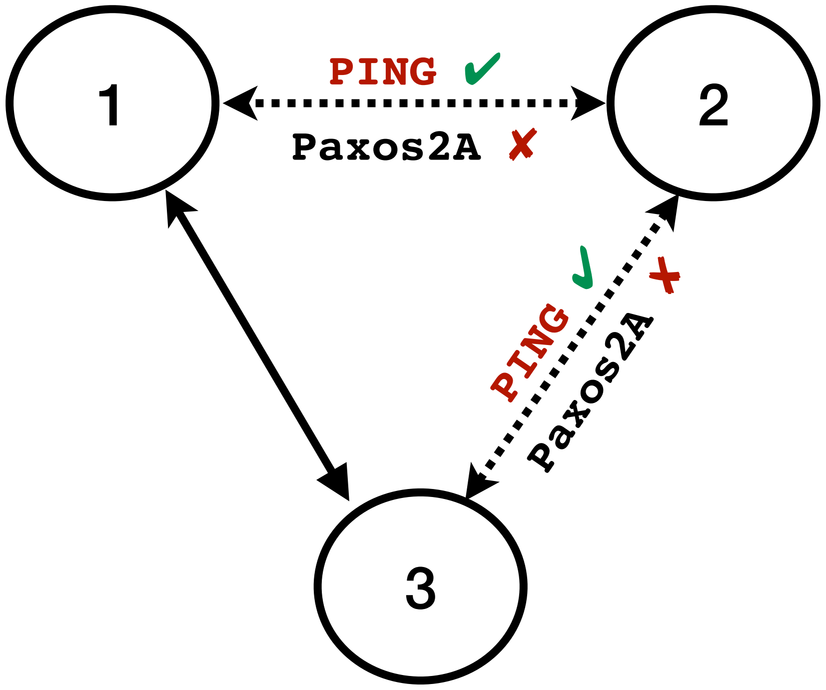
- Can't implement consensus by first implementing Ω

- Flaky channels can deliver Ω messages



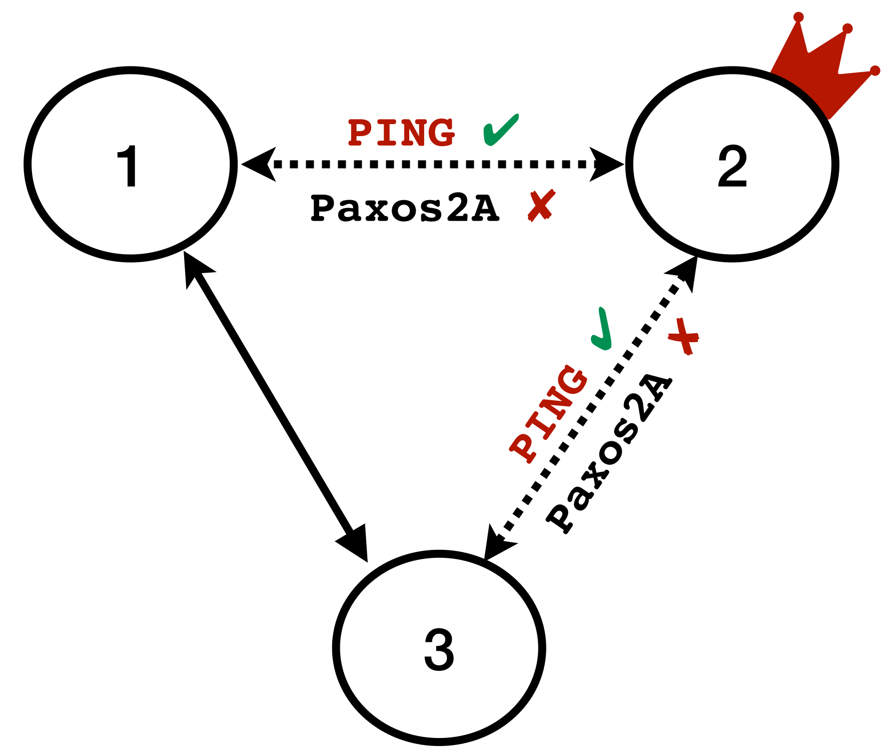
Failure detectors don't work

- Can't implement consensus by first implementing Ω
- Flaky channels can deliver Ω messages
- But drop all other messages

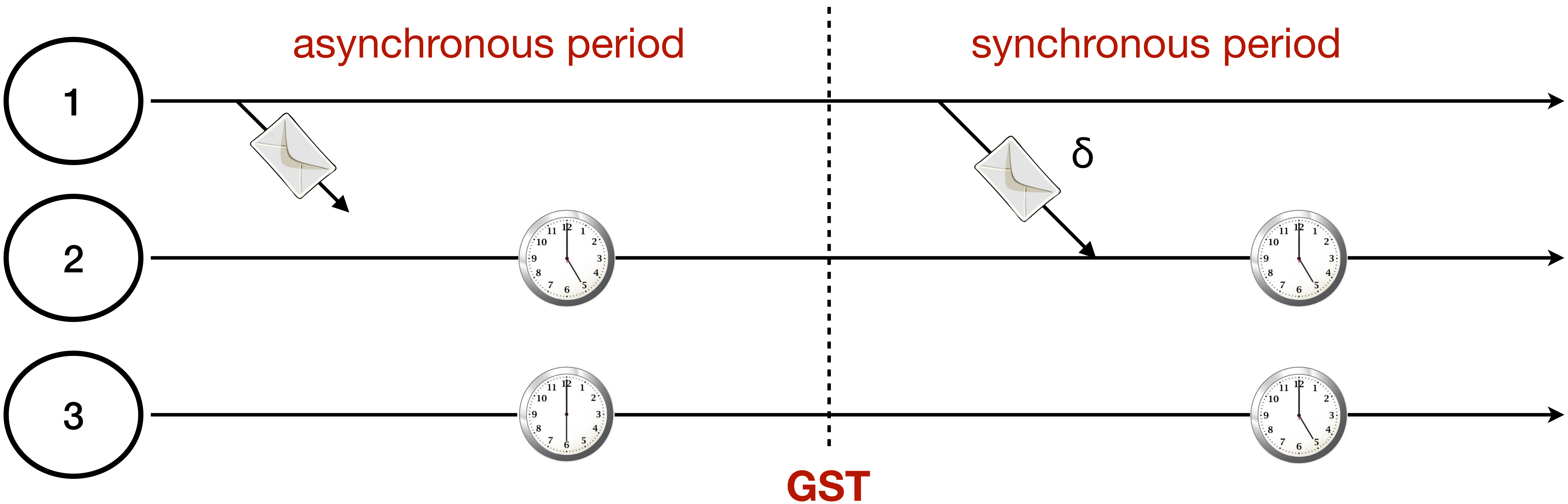


Failure detectors don’t work

- Can't implement consensus by first implementing Ω
- Flaky channels can deliver Ω messages
- But drop all other messages
- So Ω elects a leader with bad connectivity



System model

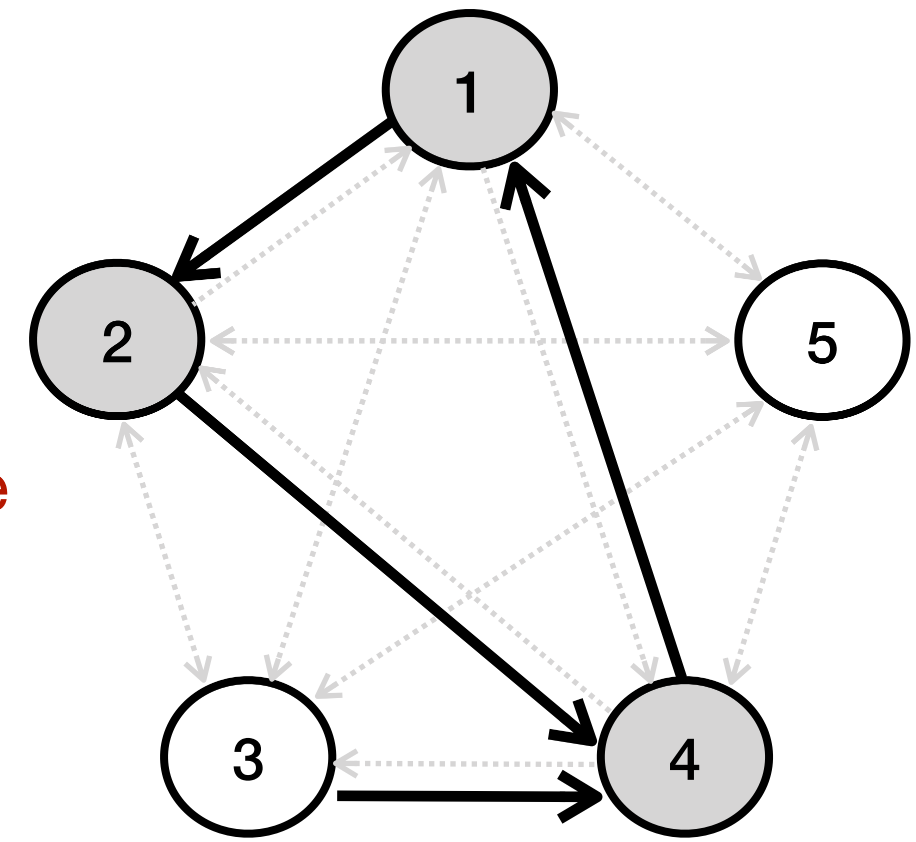


Partial synchrony where:

- Processes can fail by crashing
- Channels between correct processes are either eventually timely or flaky

Our results

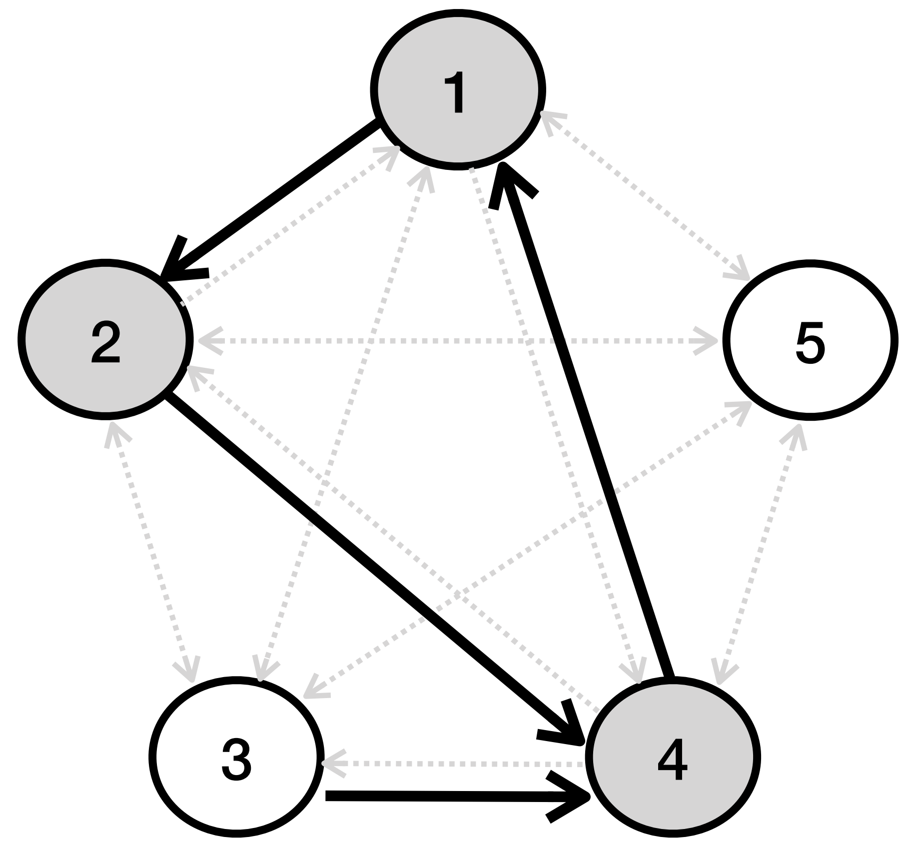
- Upper bound:
  - Can implement consensus if
    at most a minority of processes crash, and
    a majority of correct processes are strongly
    connected by correct channels: connected core



core = {1,2,4}

Our results

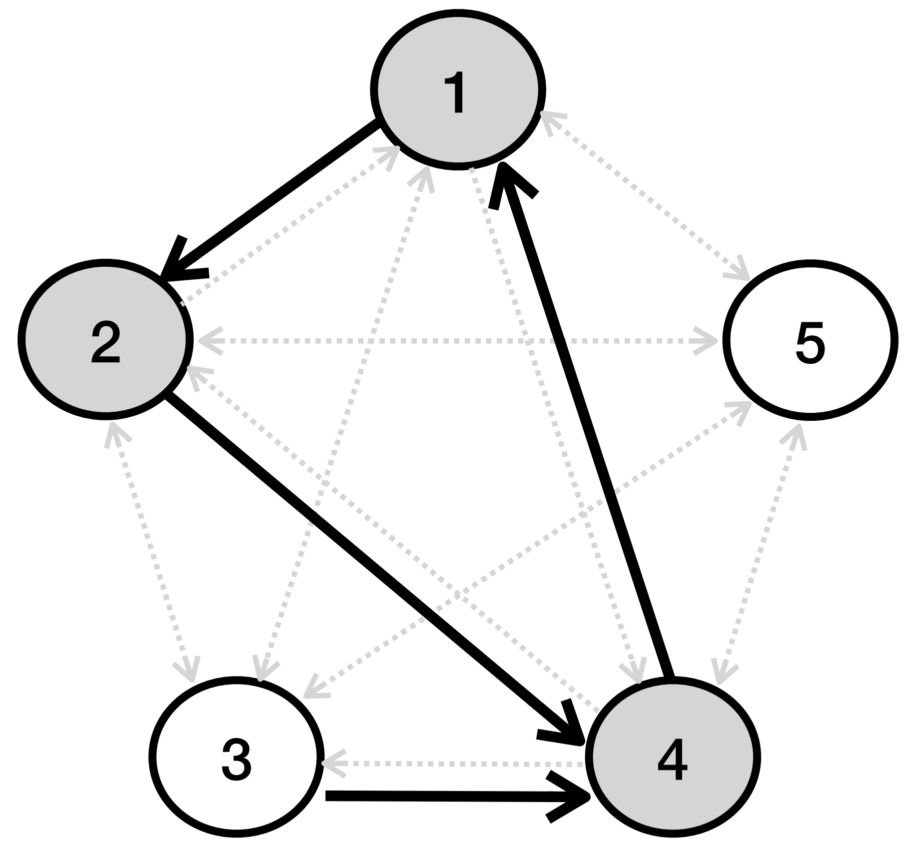
- Upper bound:
  - Can implement consensus if at most a minority of processes crash, and a majority of correct processes are strongly connected by correct channels: connected core
  - Get availability only within the connected core



core = {1,2,4}

Our results

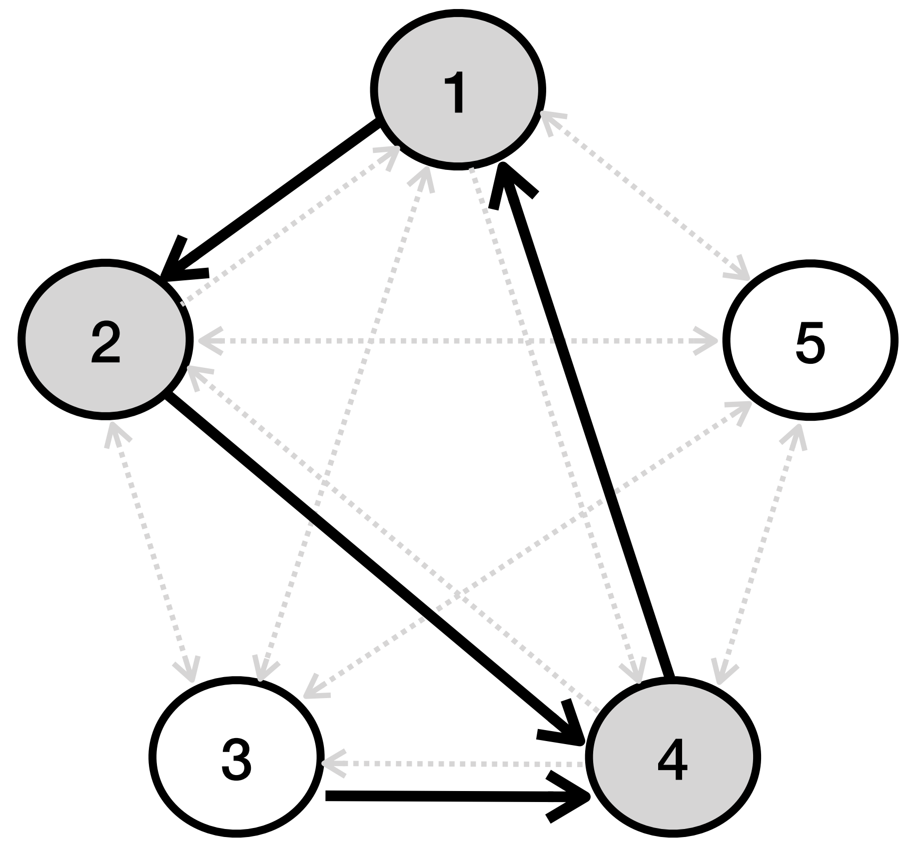
- Upper bound:
  - Can implement consensus if
    at most a minority of processes crash, and
    a majority of correct processes are strongly
    connected by correct channels: connected core
  - Get availability only within the connected core
  - Constructed using a synchronizer



core = {1,2,4}

Our results

- Upper bound:
  - Can implement consensus if at most a minority of processes crash, and a majority of correct processes are strongly connected by correct channels: connected core
  - Get availability only within the connected core
  - Constructed using a synchronizer

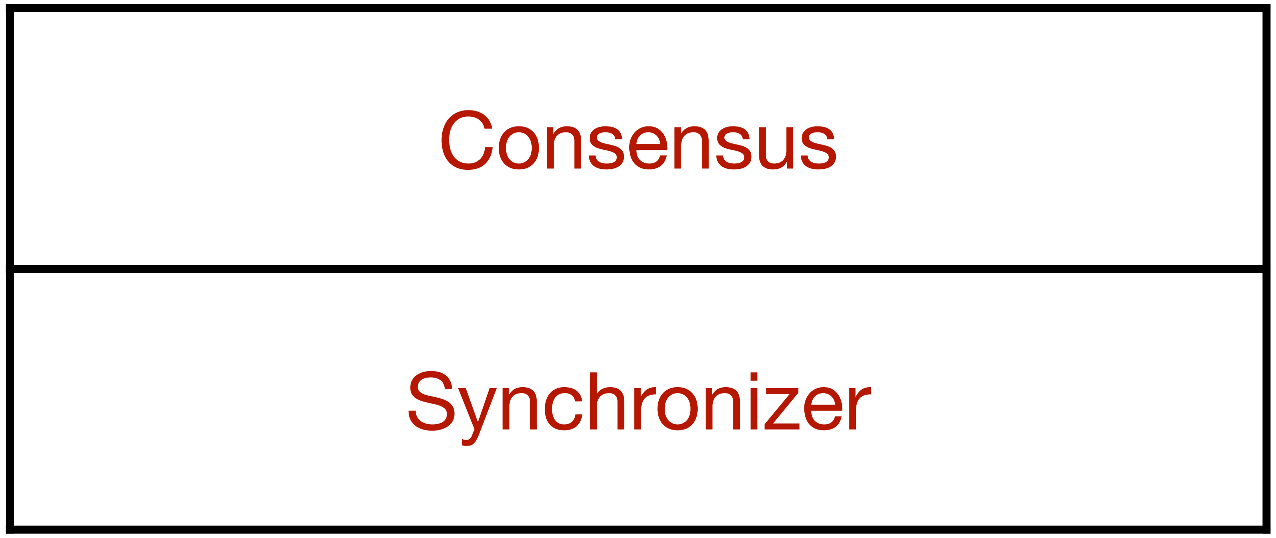


core = {1,2,4}

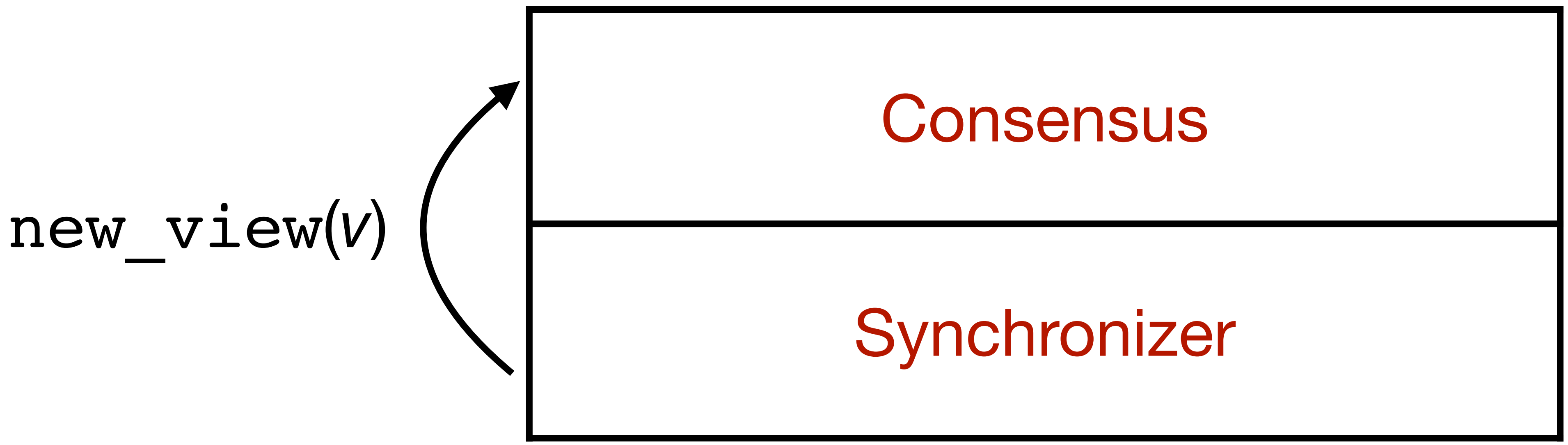
- Lower bound: our connectivity assumption is optimal

Upper bound

Synchronizer API [Naor&Keidar, 2020]

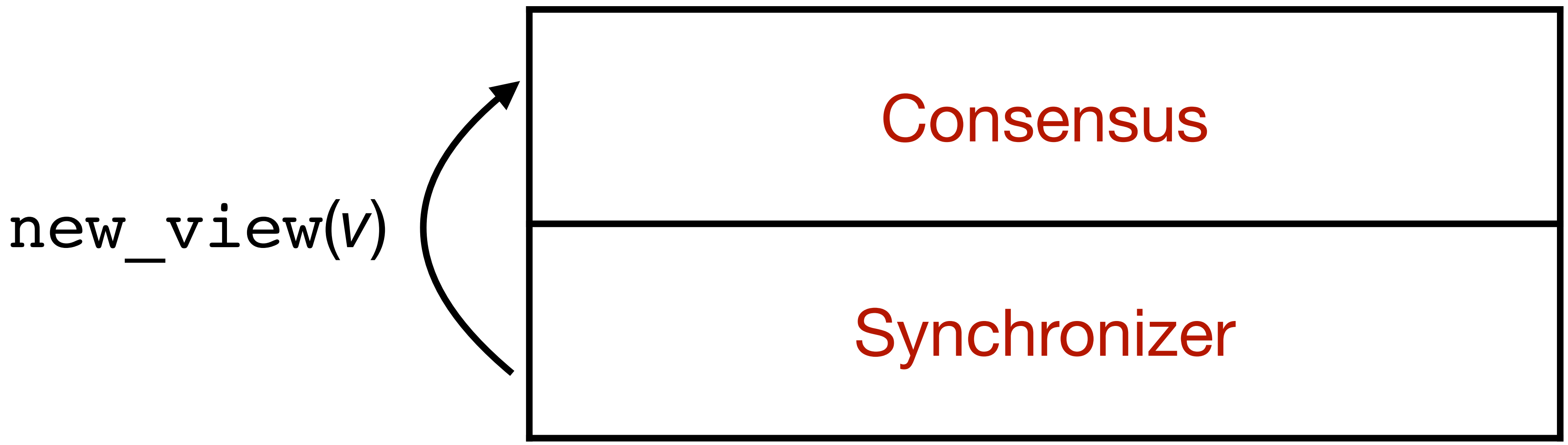


Synchronizer API [Naor&Keidar, 2020]



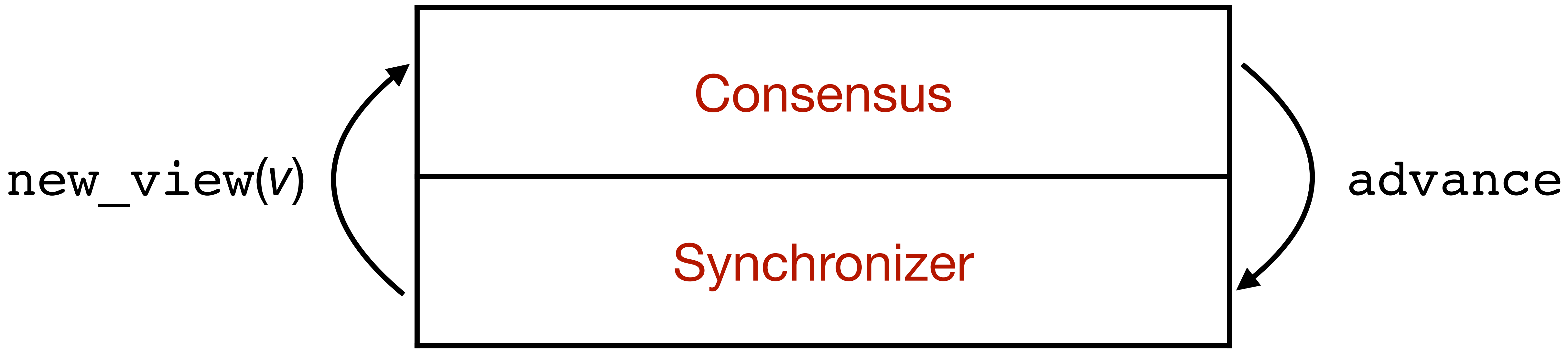
- Synchronizer tells the processes to enter a view *v* via `new_view`(*v*)

Synchronizer API [Naor&Keidar, 2020]



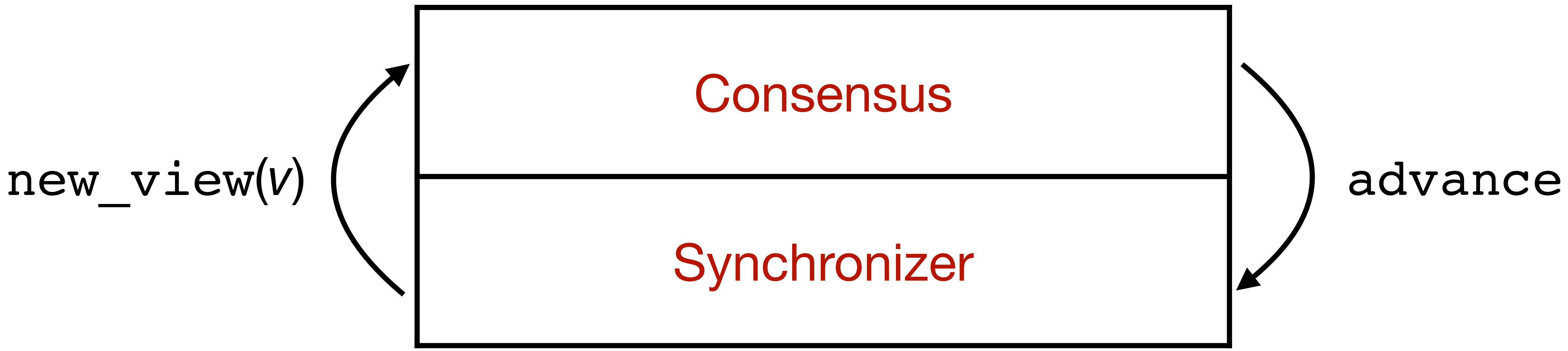
- Synchronizer tells the processes to enter a view *v* via `new_view`(*v*)
- Rules for when to switch views are protocol-specific

Synchronizer API [Naor&Keidar, 2020]



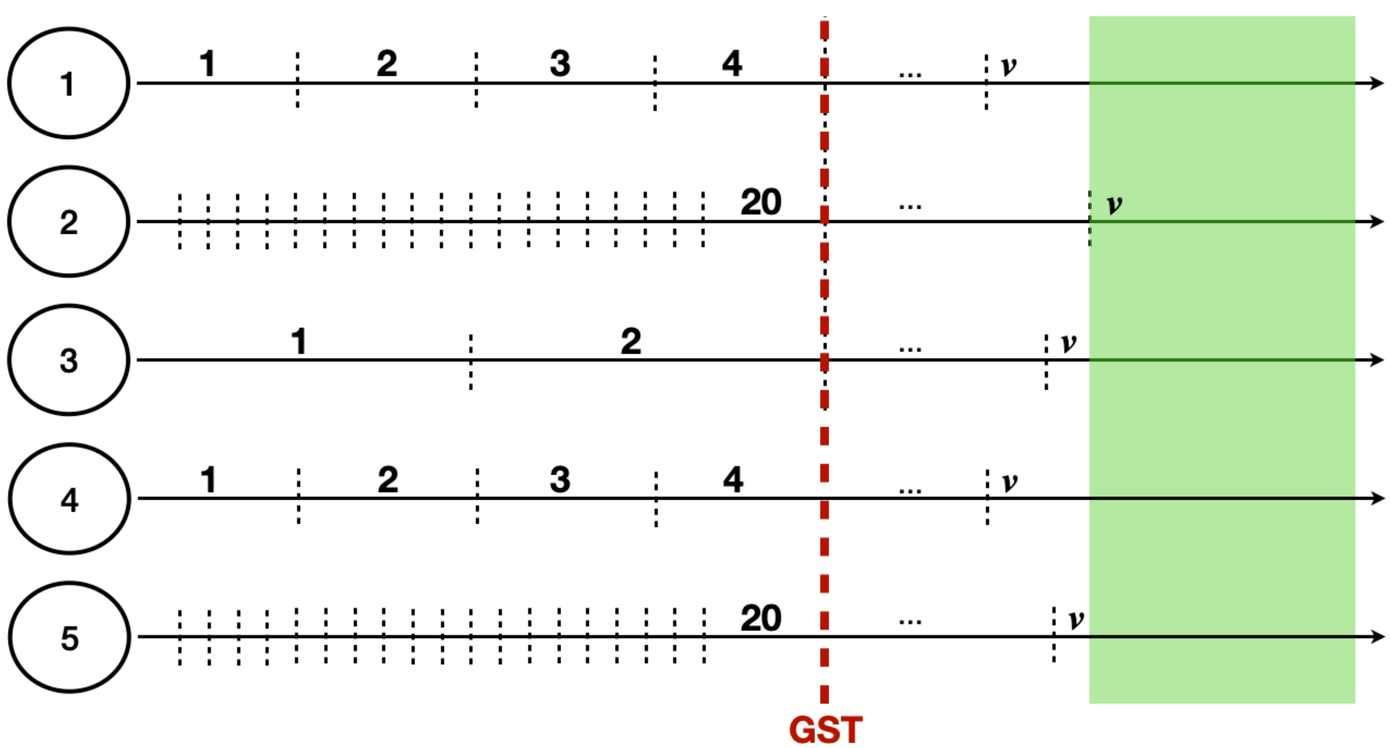
- Synchronizer tells the processes to enter a view *v* via `new_view`(*v*)
- Rules for when to switch views are protocol-specific
- A process requests a switch via `advance`

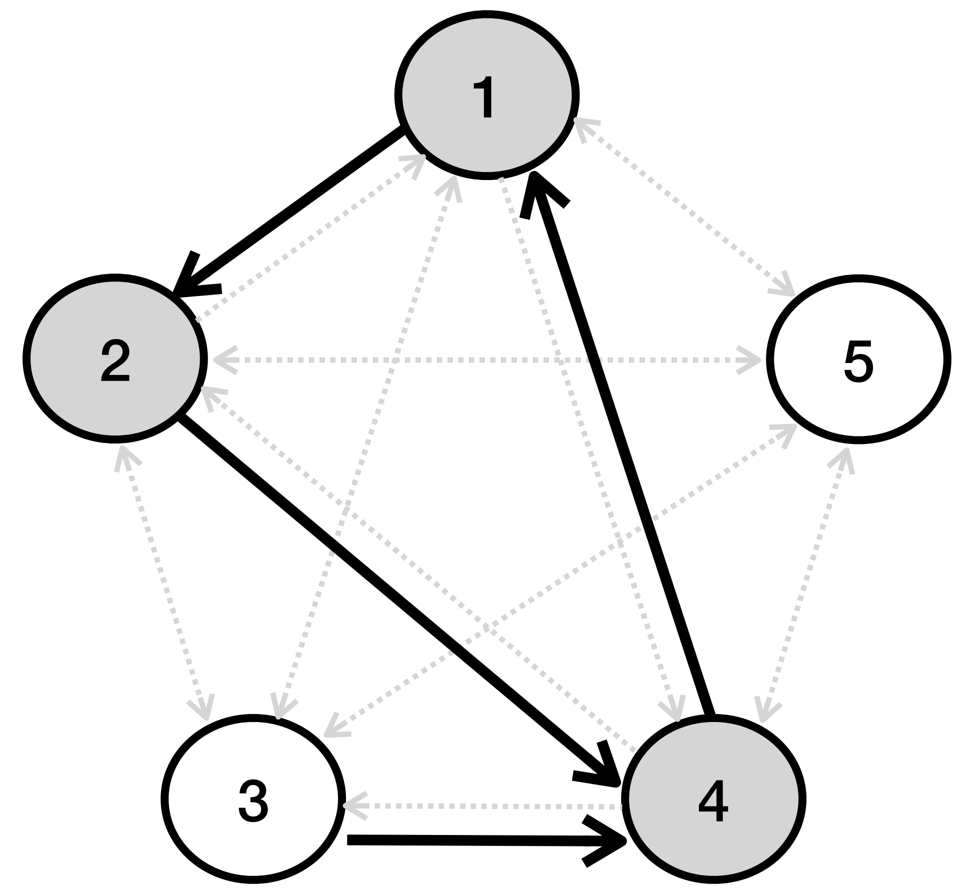
Synchronizer specification



- A balance between implementability and usability:
  - Implementable in our model
  - Can be used to implement consensus

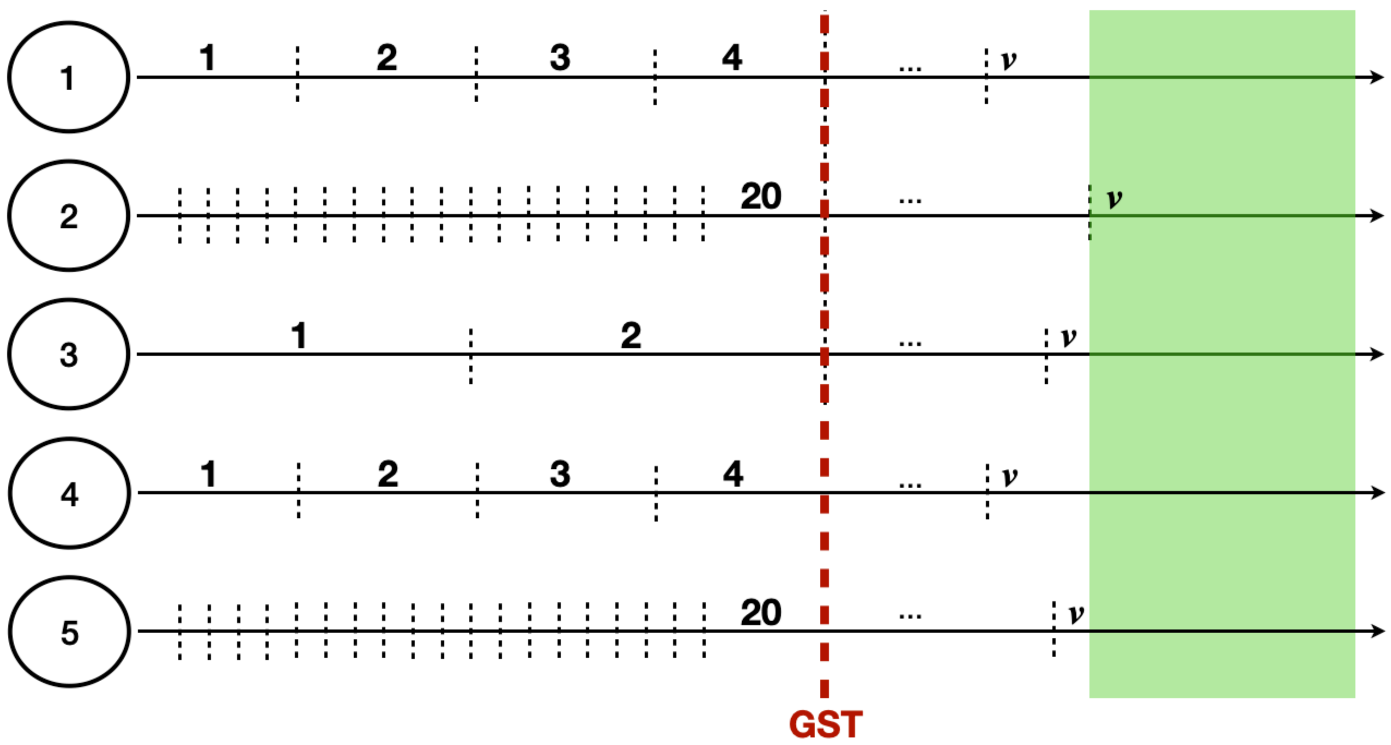
Validity

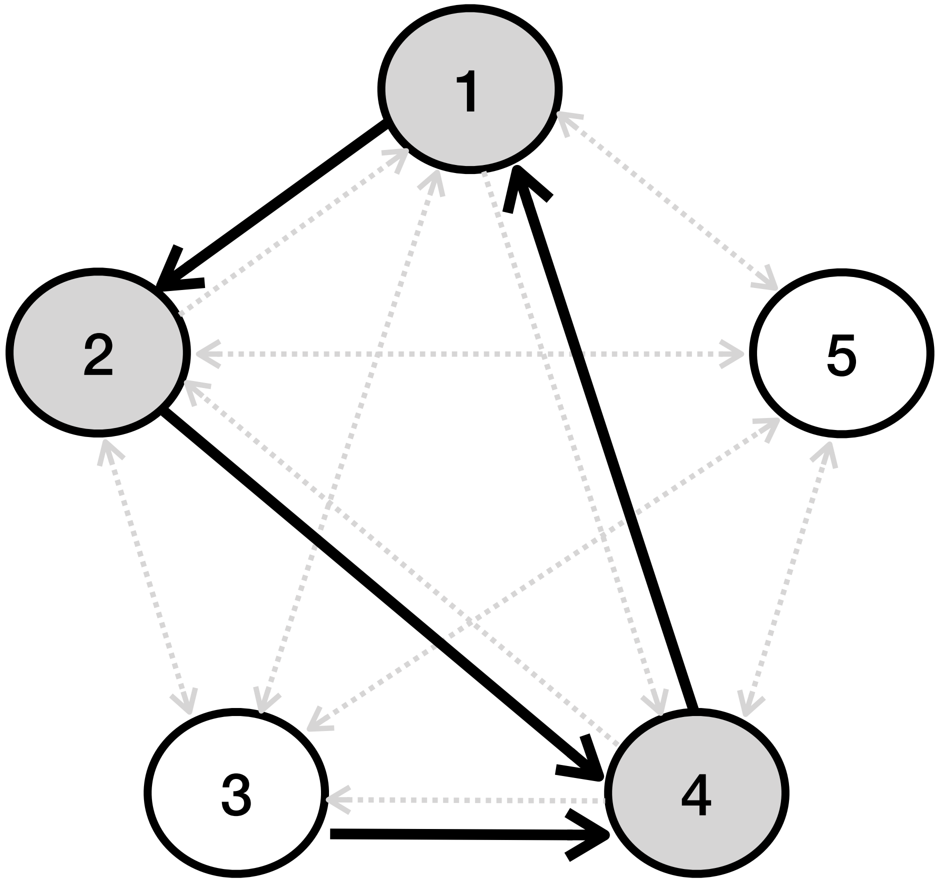




- A process can enter $v + 1$ only if some process from the core has invoked `advance` in v

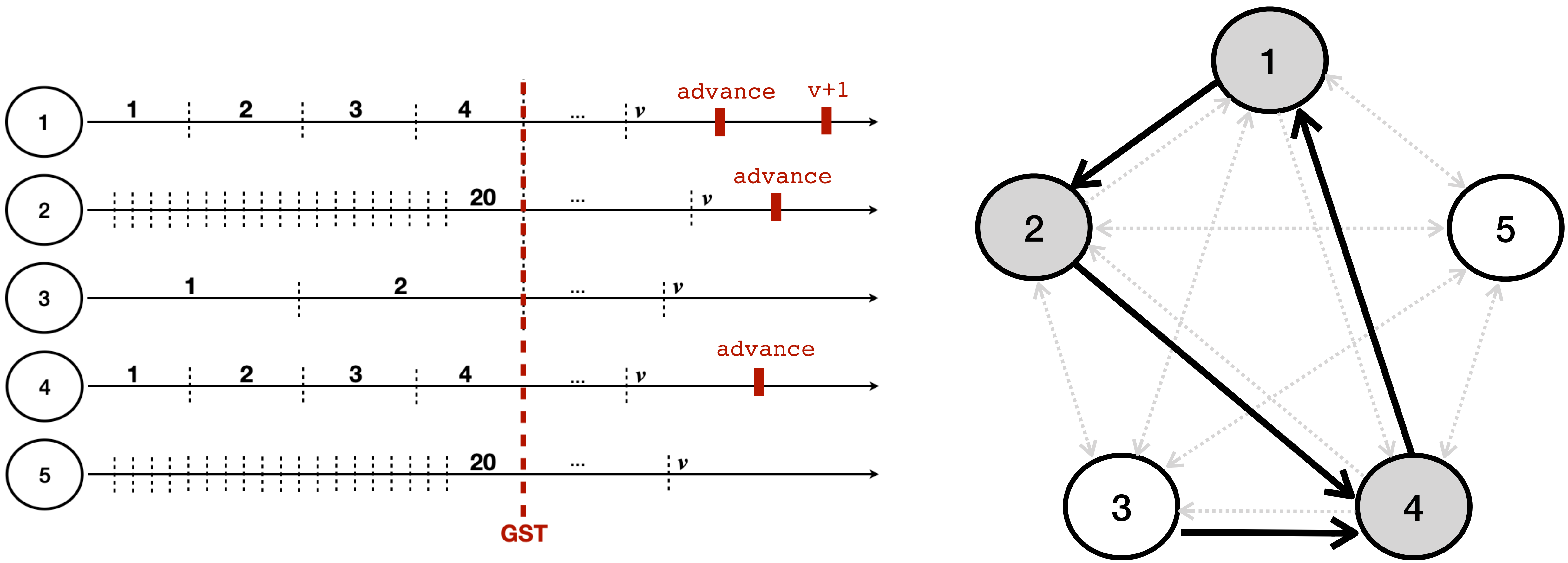
Validity





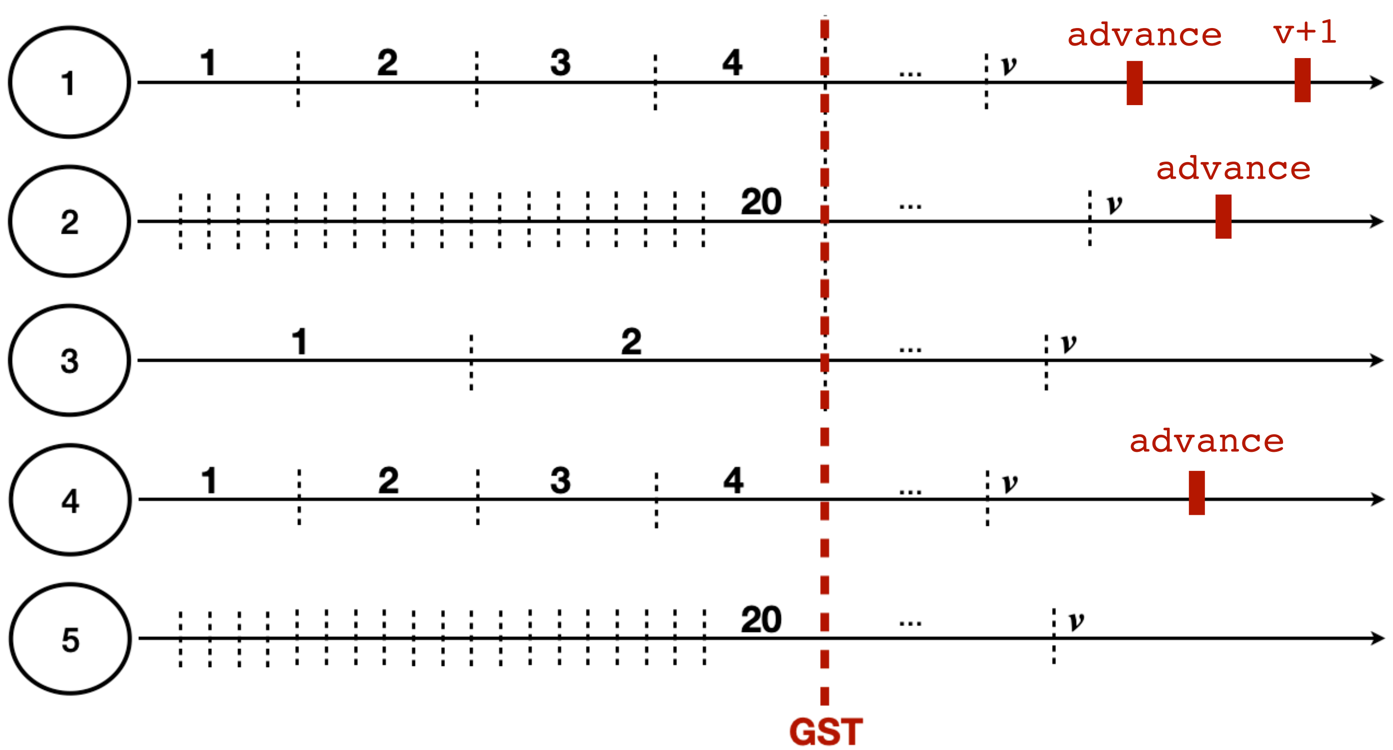
- A process can enter $v + 1$ only if some process from the core has invoked `advance` in v
- Ensures the system won't leave a view that all processes from the core are happy with

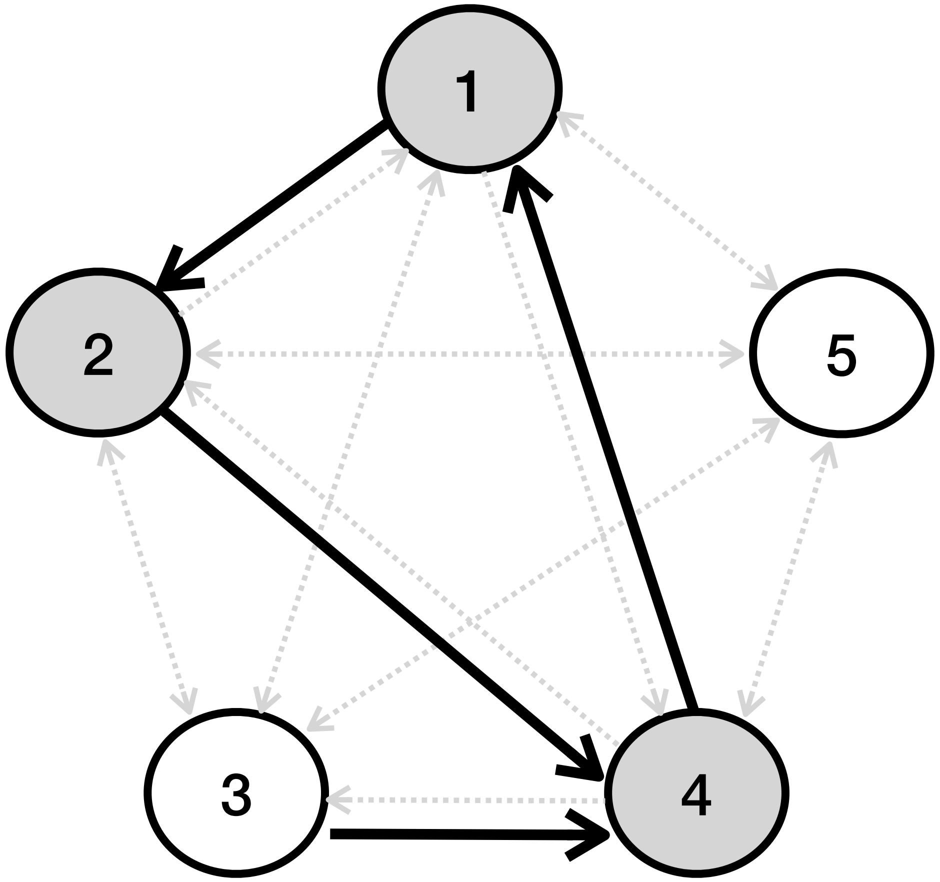
Progress (simplified)



- Some process from the core will enter $v + 1$ if $>n/2$ processes from the core invoke `advance` in v

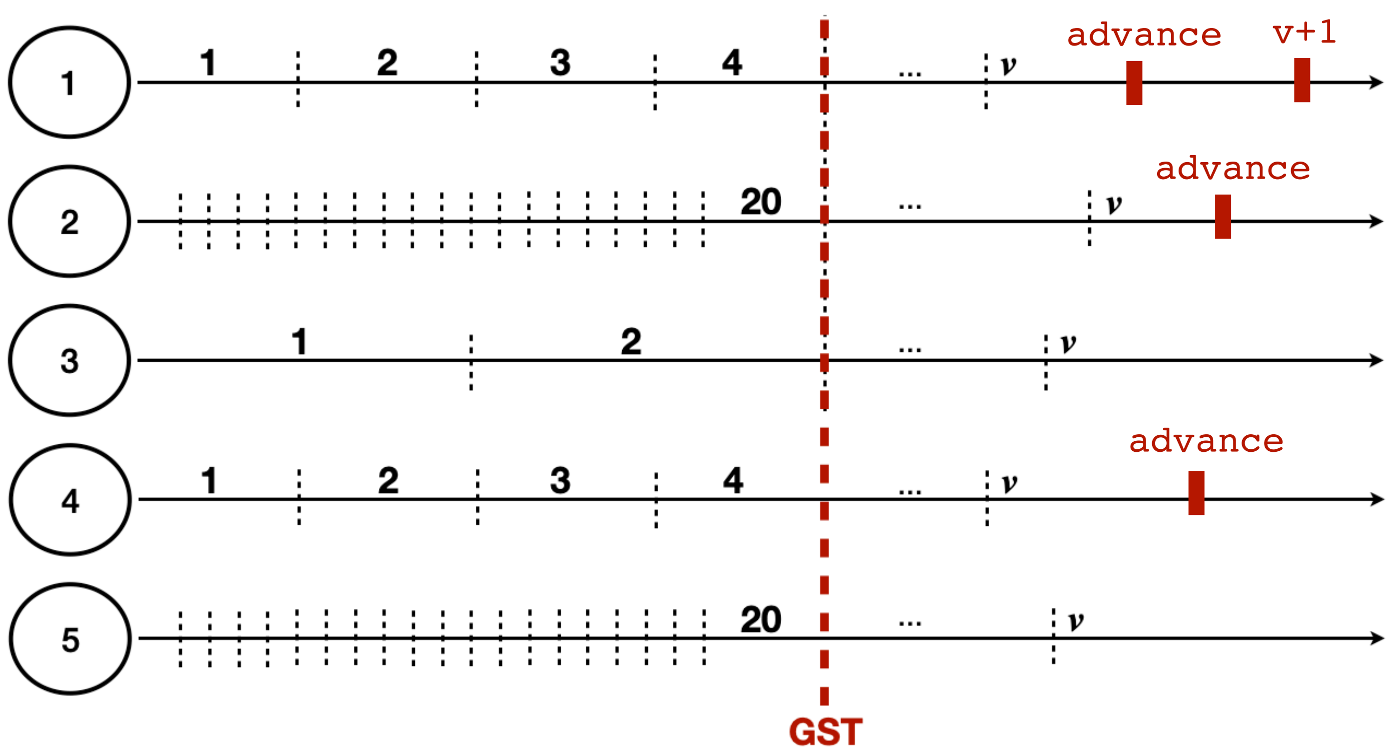
Progress (simplified)

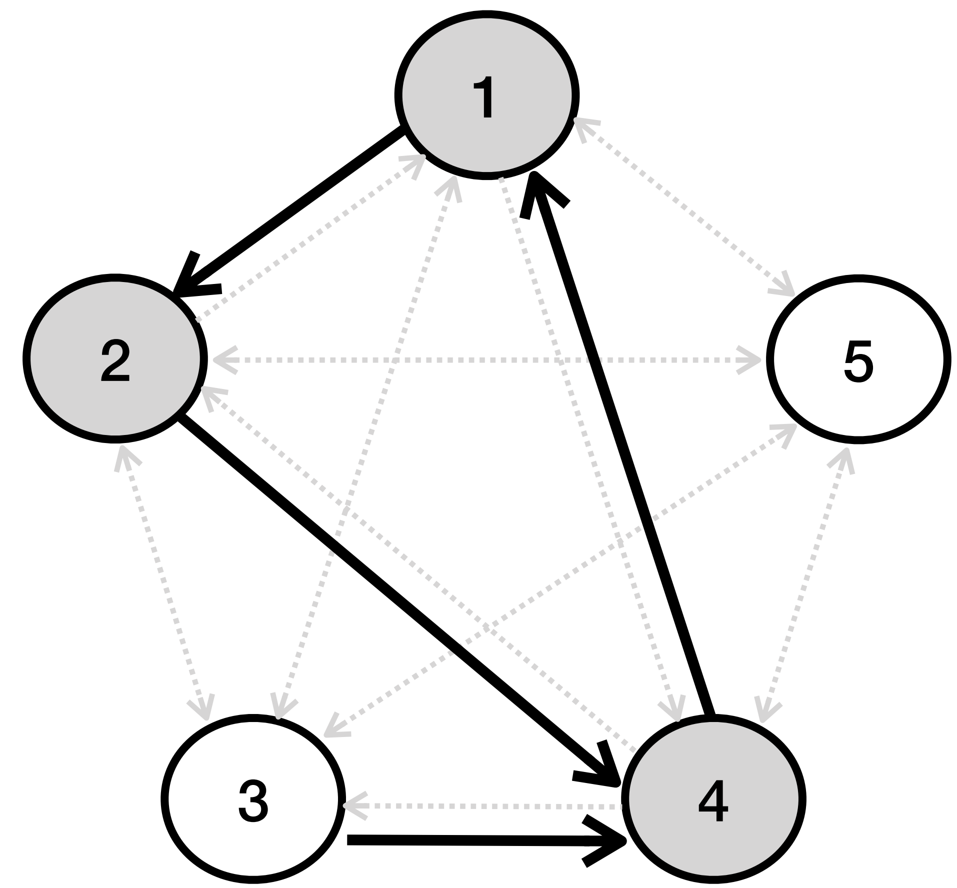




- Some process from the core will enter v + 1 if >n/2 processes from the core invoke `advance` in v
- Allows iterating over views in search of a correct well-connected leader

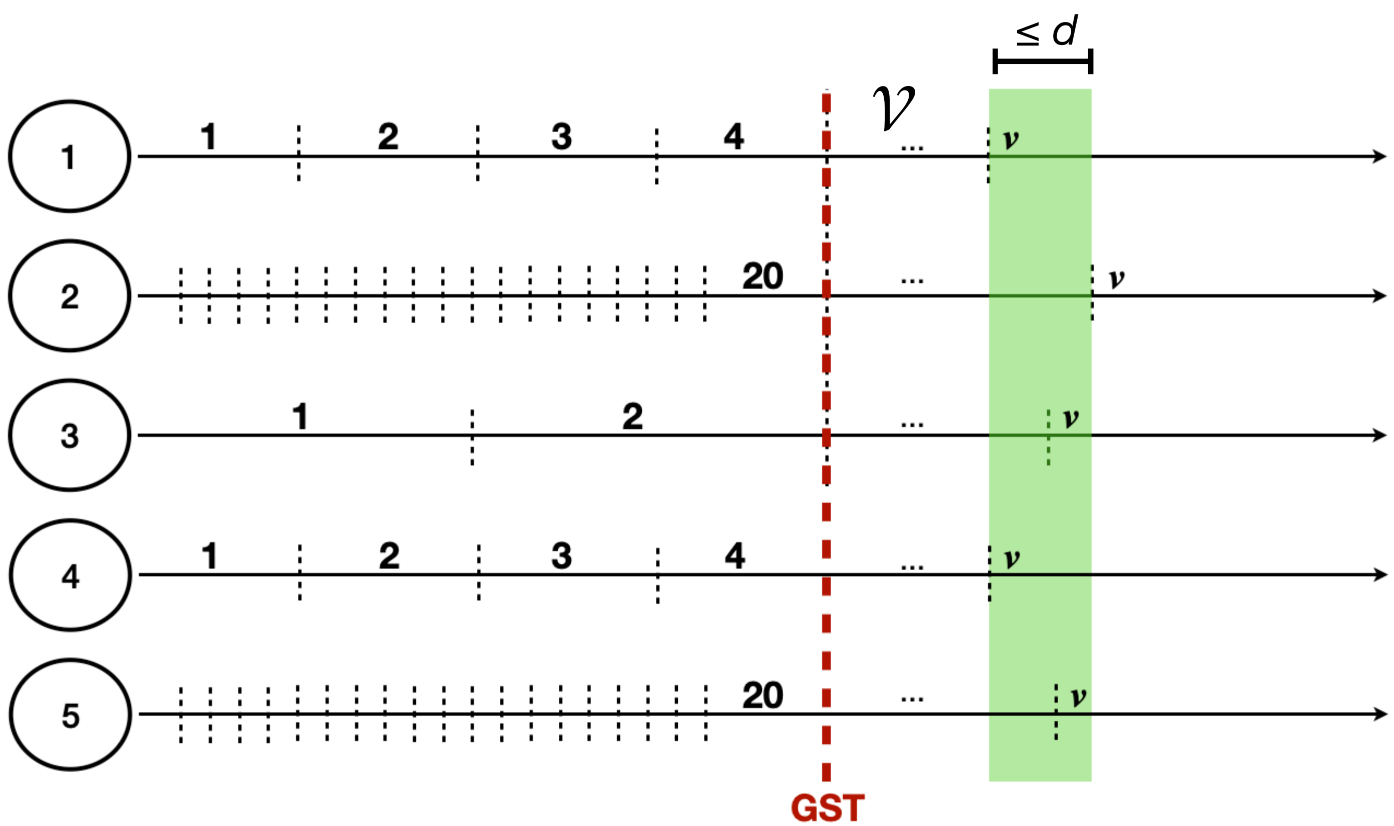
Progress (simplified)

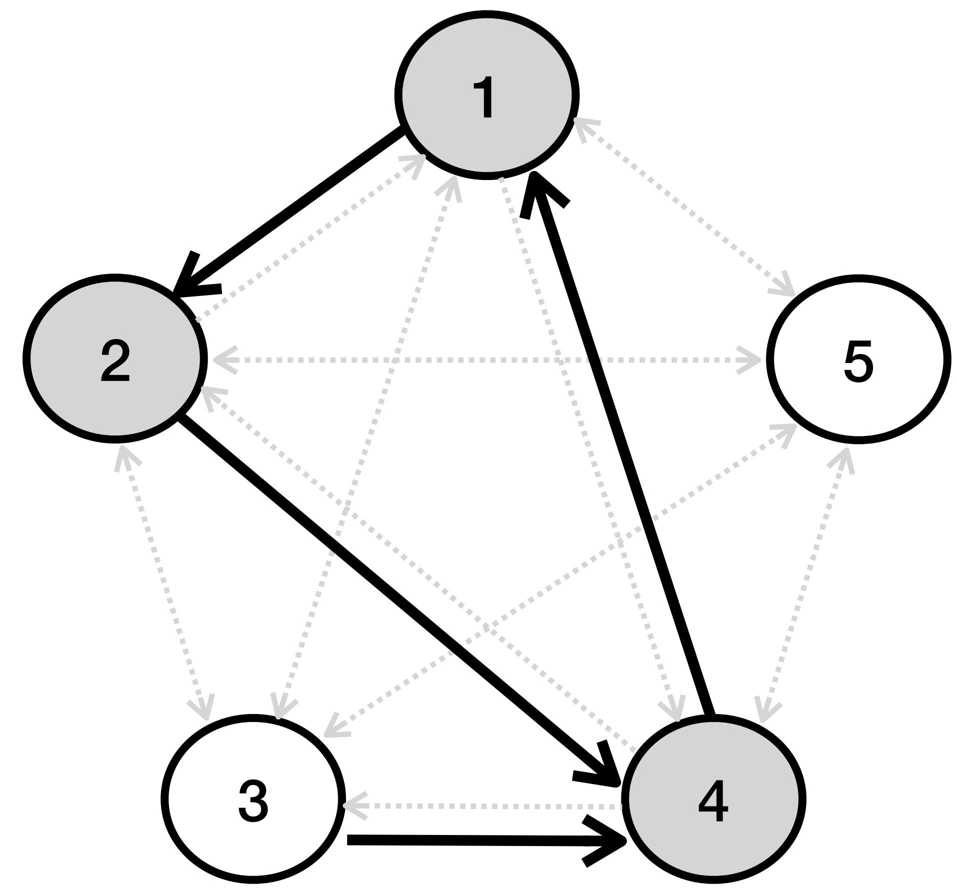




- Some process from the core will enter $v + 1$ if $>n/2$ processes from the core invoke `advance` in v
- Allows iterating over views in search of a correct well-connected leader
- $>n/2$ `advance` calls instead of 1: needed for implementability

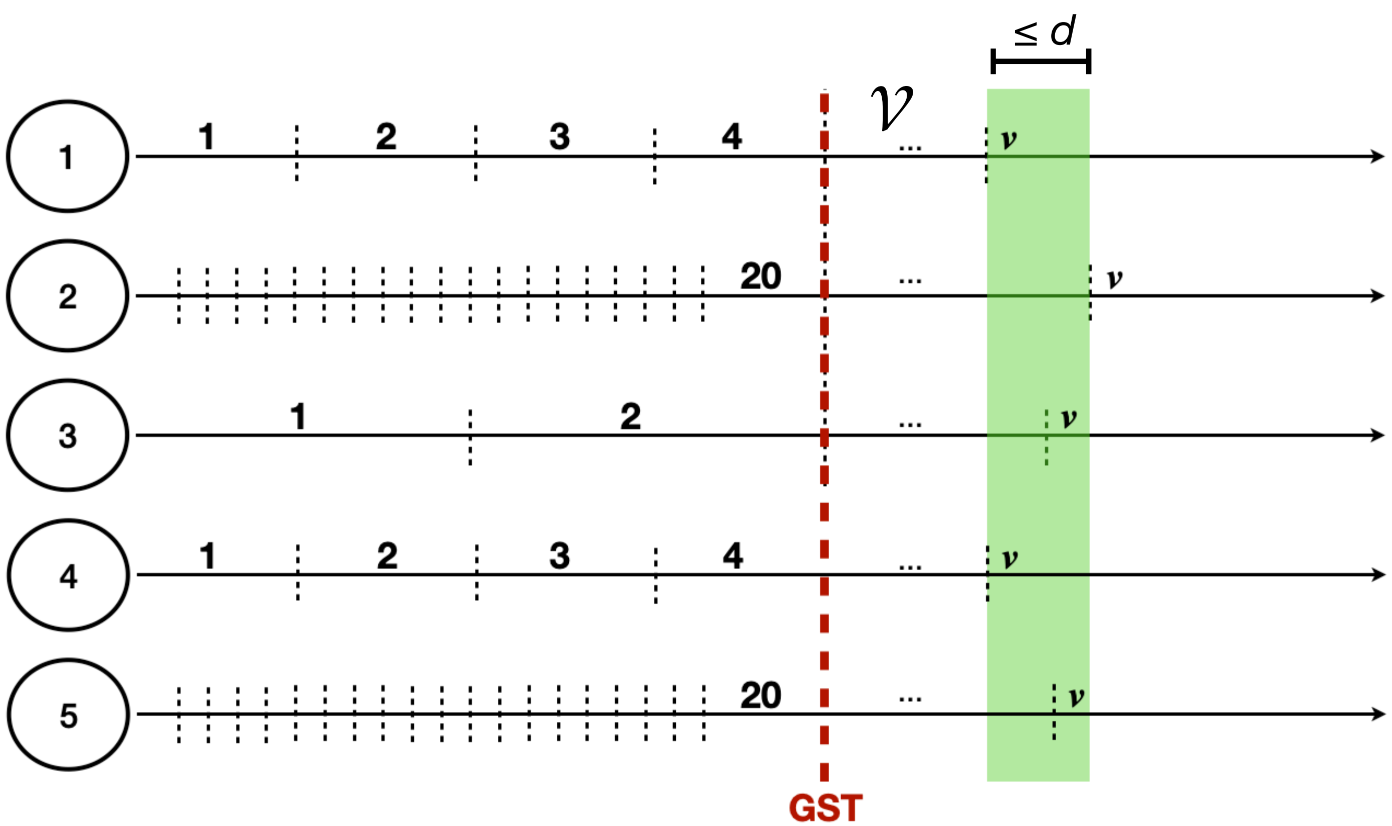
Bounded entry

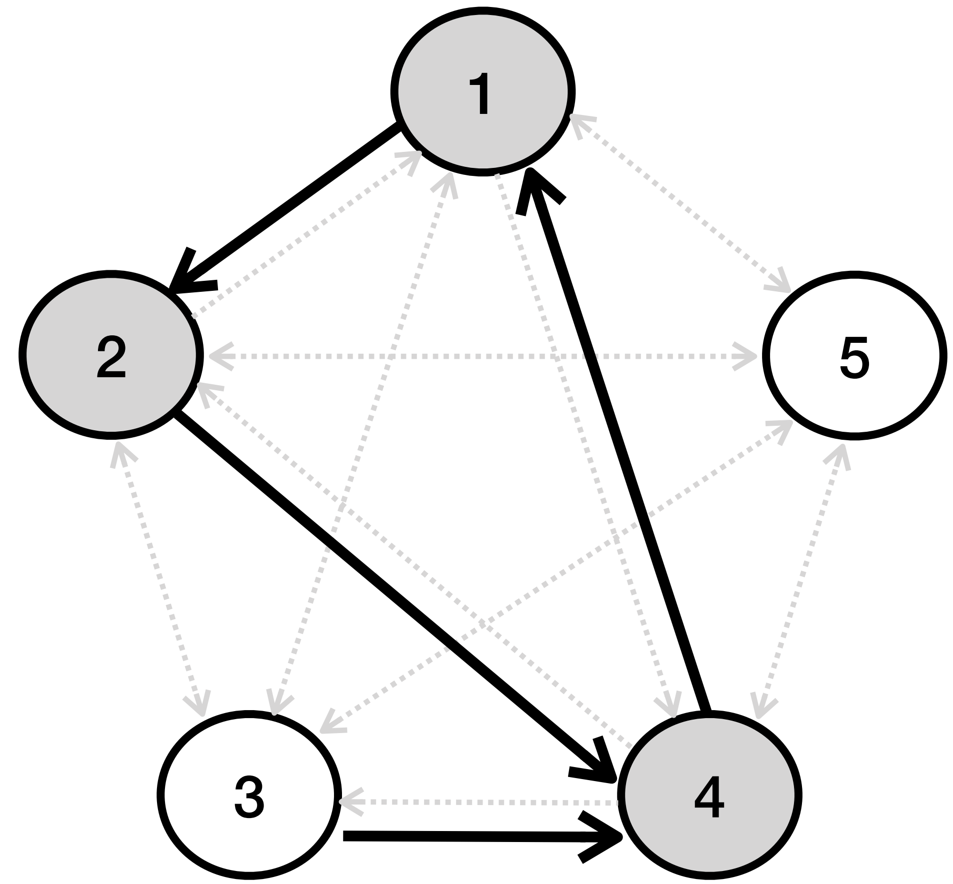




- If a process from the core enters *v*, then all processes from the core will enter *v* within *d* (e.g., *δ* * diameter(core))

Bounded entry

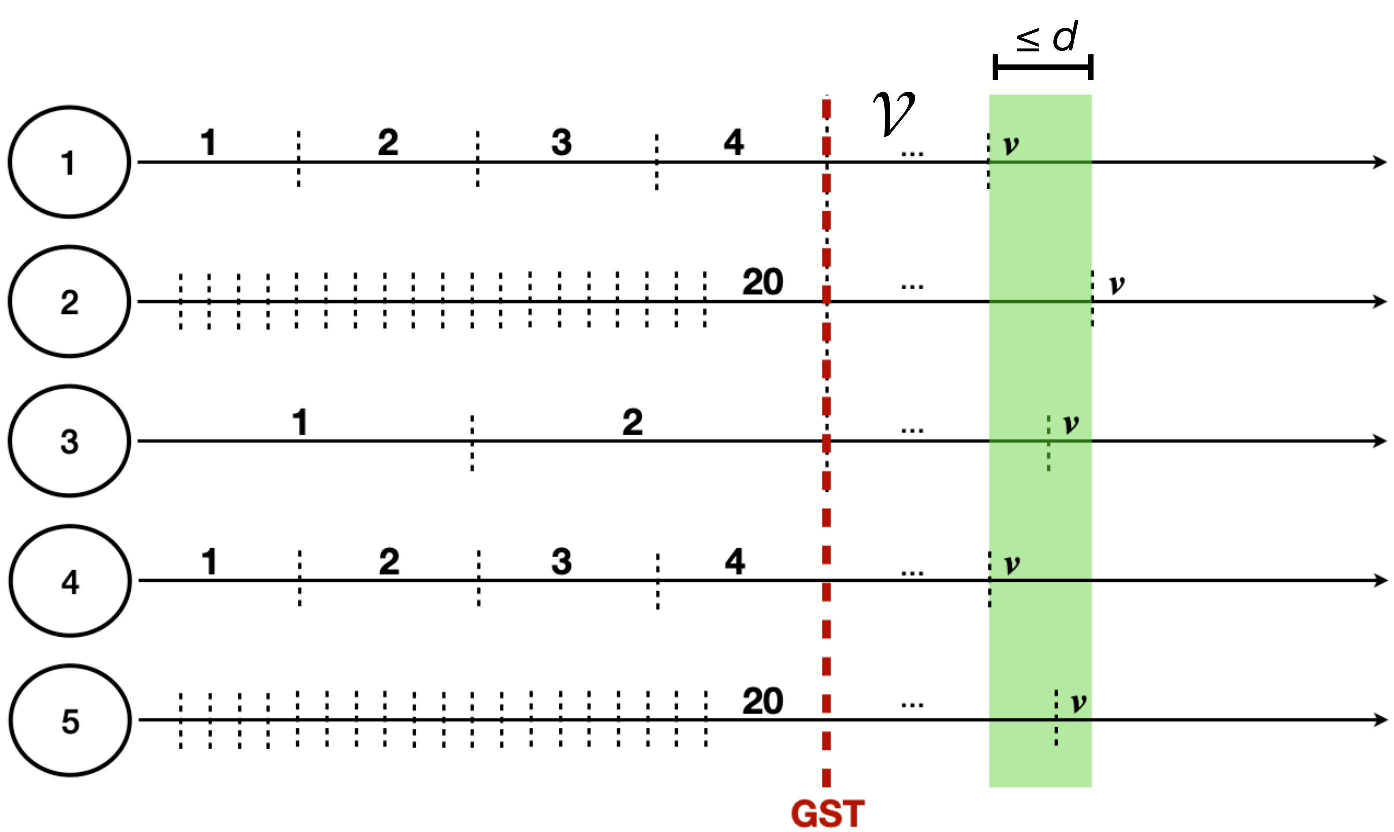


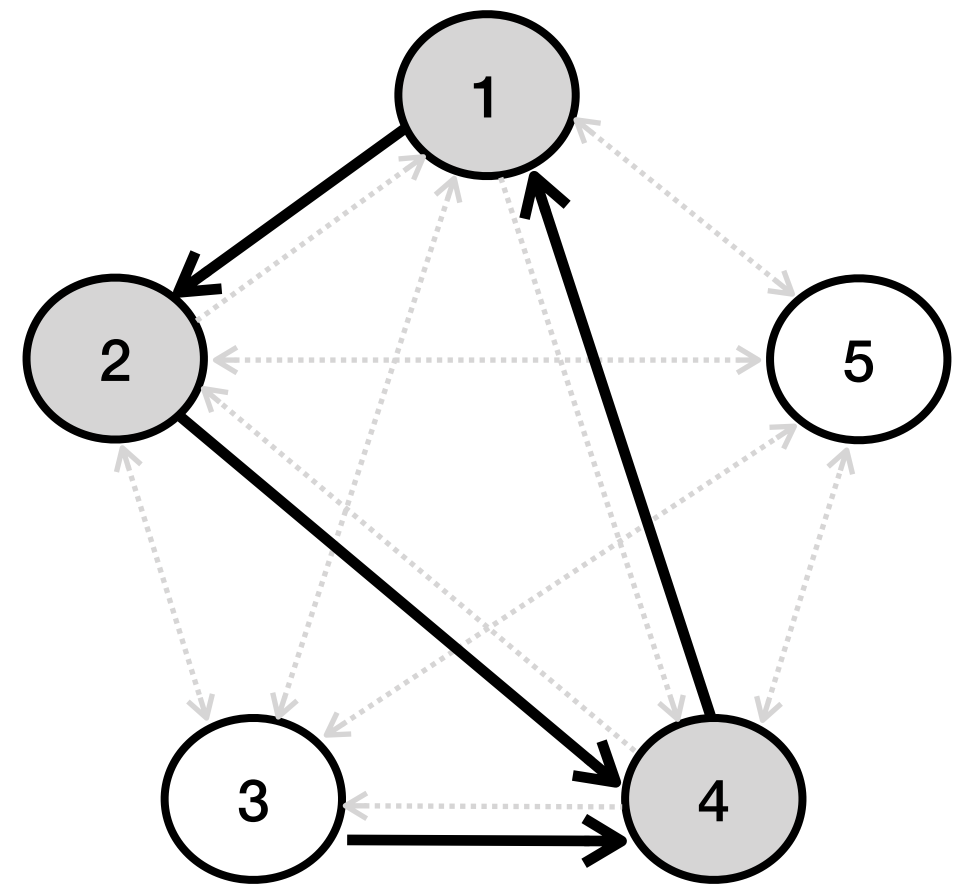


- If a process from the core enters *v*, then all processes from the core will enter *v* within *d* (e.g., δ * diameter(core)), provided $v \geq \mathcal{V}$

Before GST may not be able to exchange messages needed to synchronise processes

Bounded entry

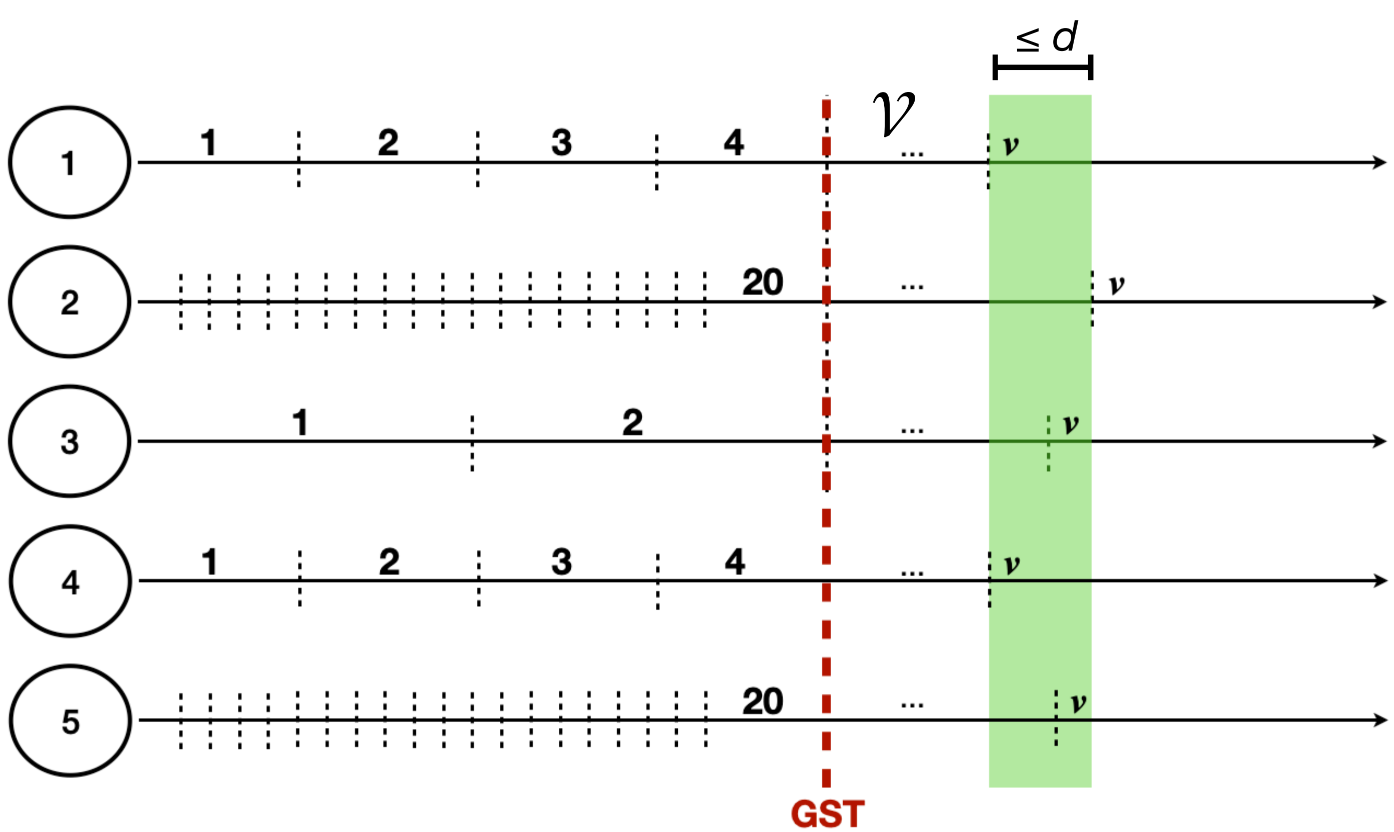


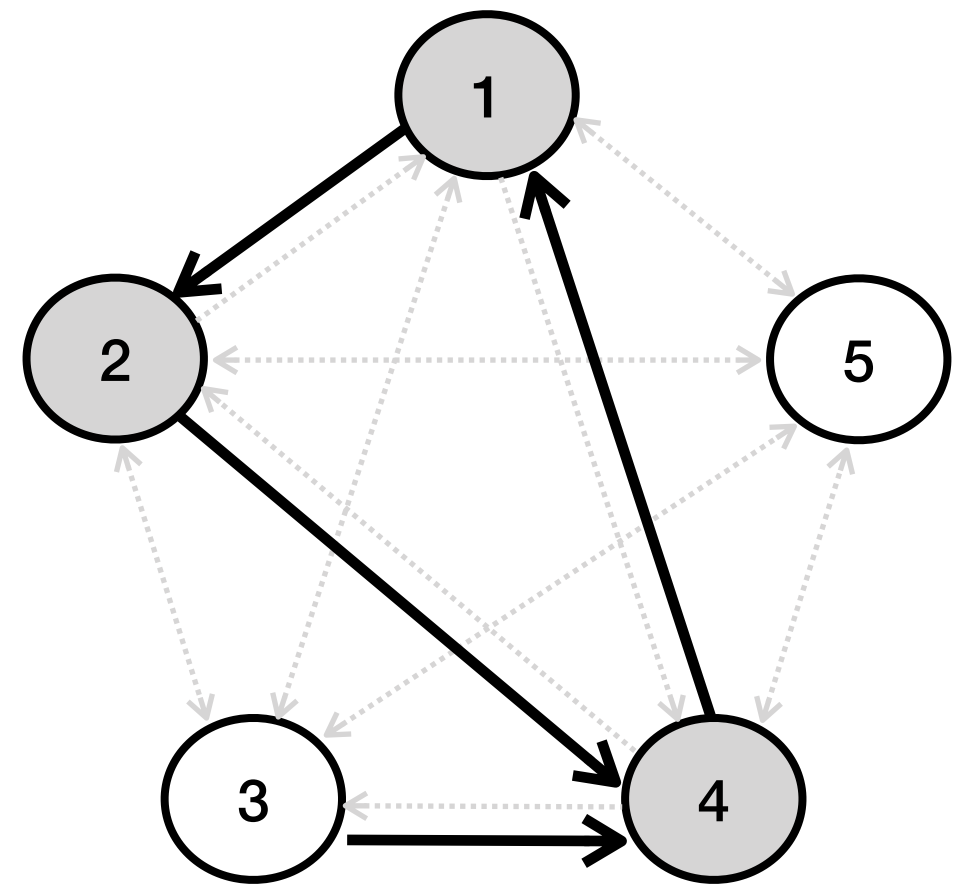


- If a process from the core enters *v*, then all processes from the core will enter *v* within *d* (e.g., δ * diameter(core)),
  provided $v \geq \mathcal{V}$ and no process from the core attempts to advance to a higher view within *d*

> If a process calls `advance` from *v*, then some processes may skip *v* and enter *v*+1 directly

Bounded entry



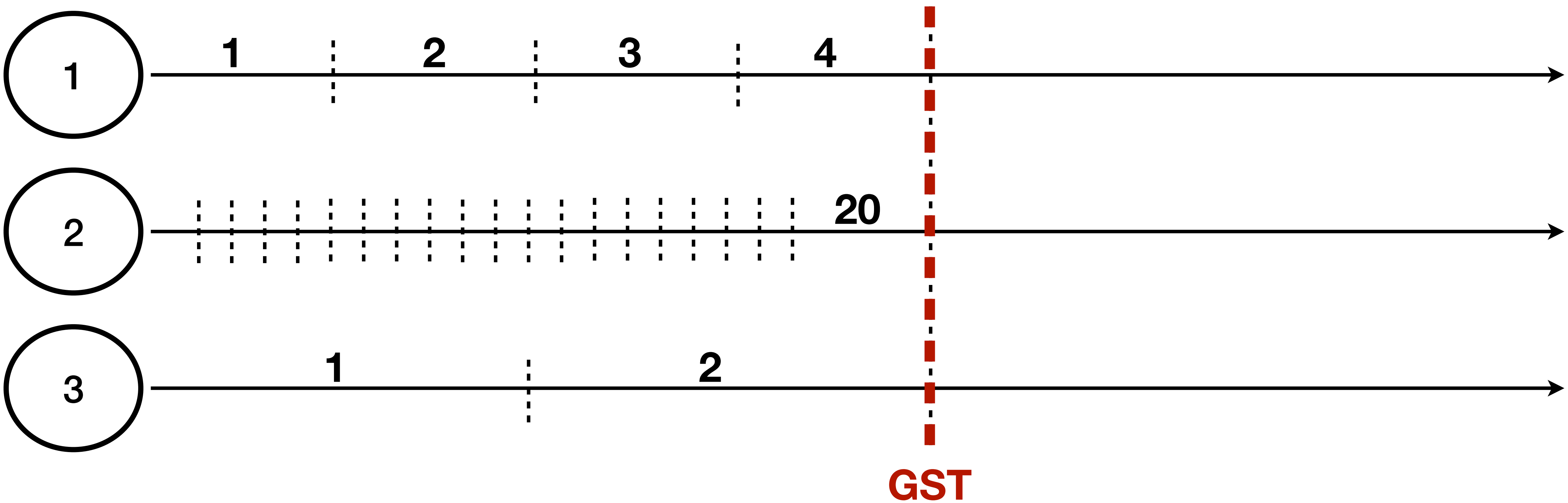


- If a process from the core enters *v*, then all processes from the core will enter *v* within *d* (e.g., δ * diameter(core)),
  provided $v \geq \mathcal{V}$ and no process from the core attempts to advance to a higher view within *d*
- Allows promptly bringing the core into the same view

Synchronizer specification

- Progress: allows iterating over views in search for a leader from the core
- Bounded entry: ensures all process from the core enter the same view
- Validity: ensures processes from the core stay in a good view

Implementation



- Don't just enter a new view once somebody calls `advance`: processes need to communicate first

Implementation

- When a process calls `advance` in a view v,
  it broadcasts `WISH`(v + 1), saying it wants to enter $v + 1$

Implementation

- When a process calls `advance` in a view v,
  it broadcasts `WISH`(v + 1), saying it wants to enter v + 1

- A process enters view v + 1 when a majority of processes
  express a similar wish: e.g. `WISH`(v + 1)

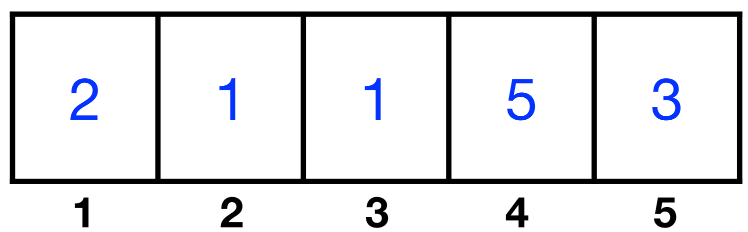
Implementation

- When a process calls `advance` in a view v,
  it broadcasts `WISH`(v + 1), saying it wants to enter $v + 1$

- A process enters view $v + 1$ when a majority of processes express a similar wish: e.g. `WISH`(v + 1)

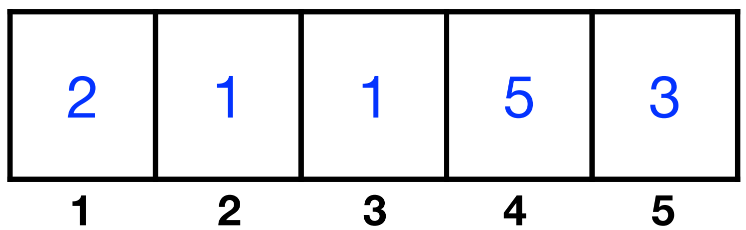
- Requiring majority guards against disruptions by badly connected processes

Implementation



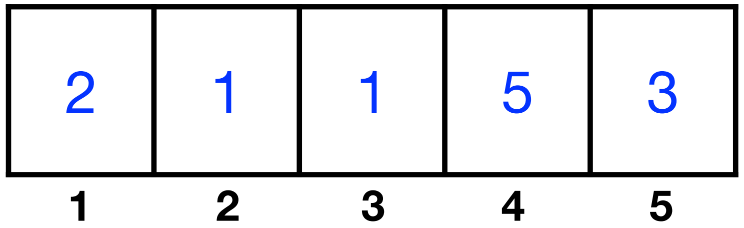
- Maintain an array with the highest `WISH` received from each process: run in bounded space

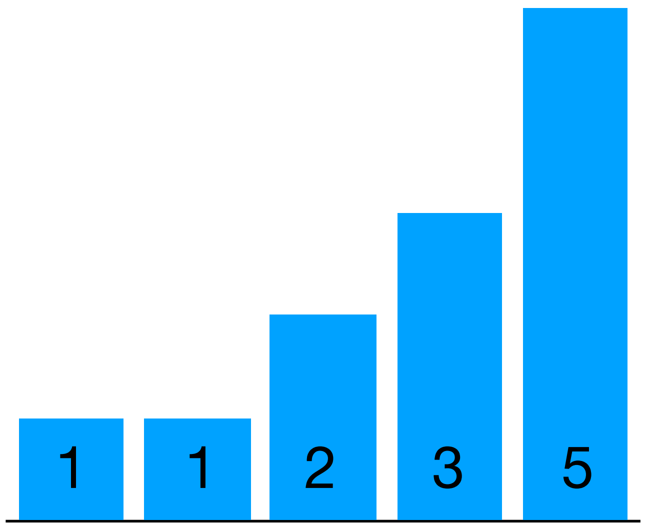
Implementation



- Maintain an array with the highest `WISH` received from each process: run in bounded space
- When received $n/2 + 1$ `WISH`es for views > yours, enter the minimal view in them

Implementation

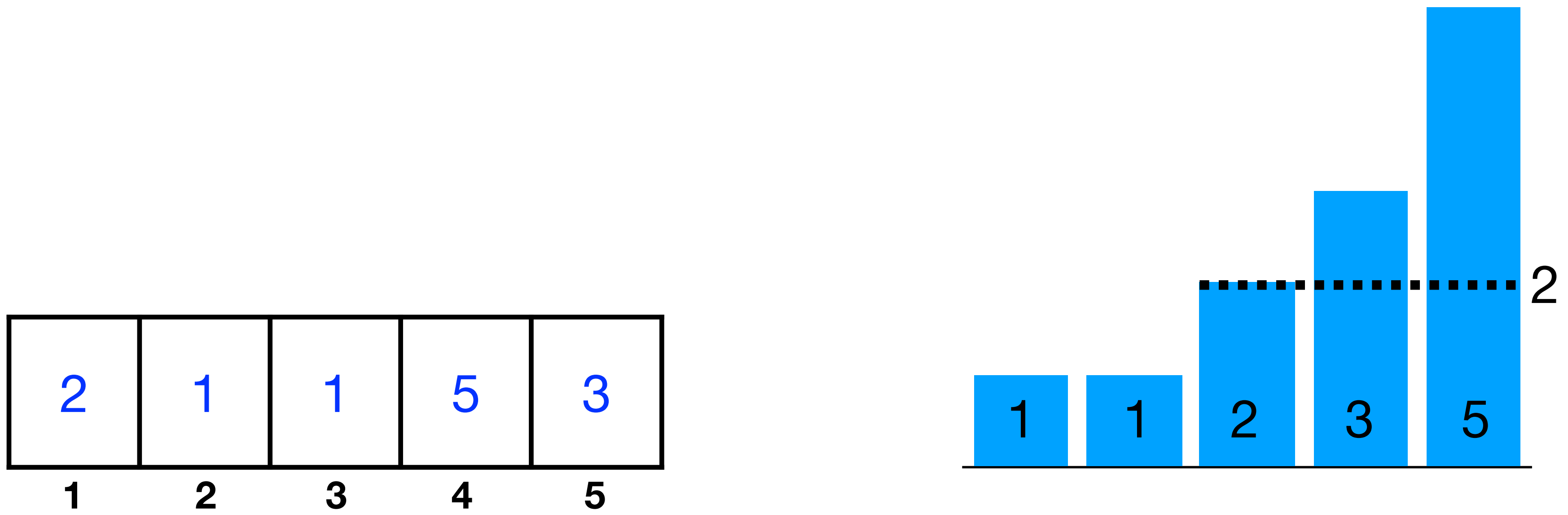




- Maintain an array with the highest `WISH` received from each process: run in bounded space

- When received $n/2 + 1$ `WISH`es for views > yours, enter the minimal view in them

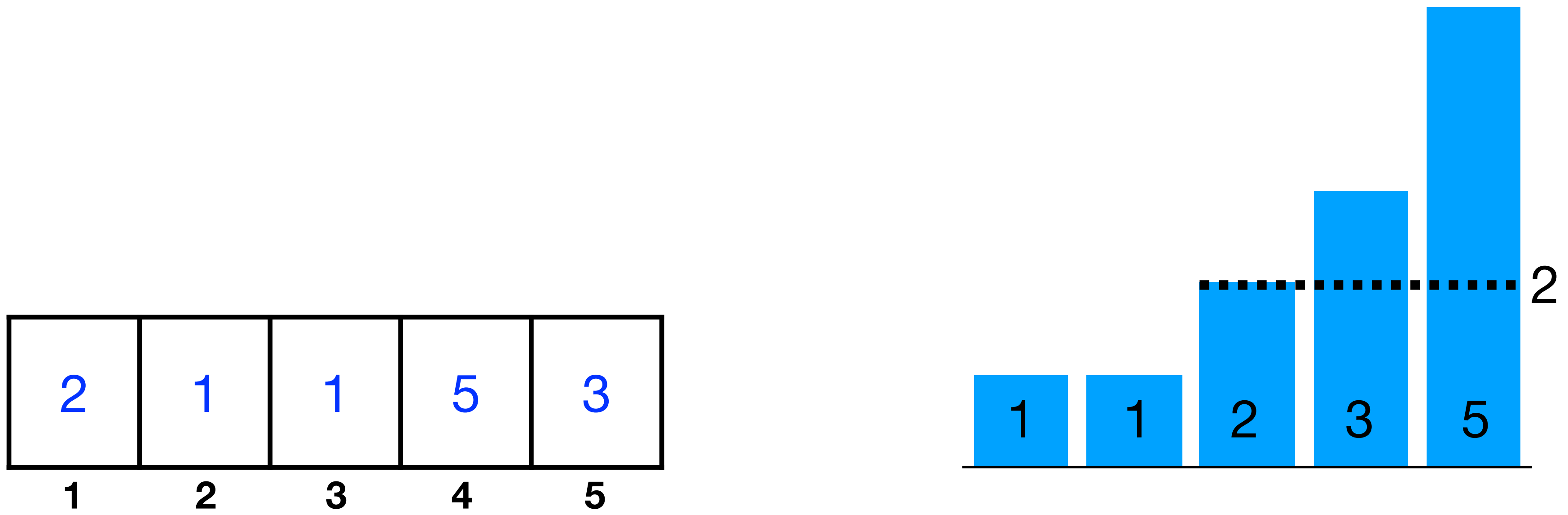
Implementation



- Maintain an array with the highest `WISH` received from each process: run in bounded space

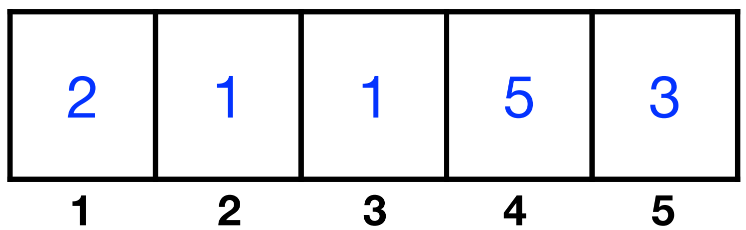
- When received *n*/2 + 1 `WISH`es for views > yours, enter the minimal view in them

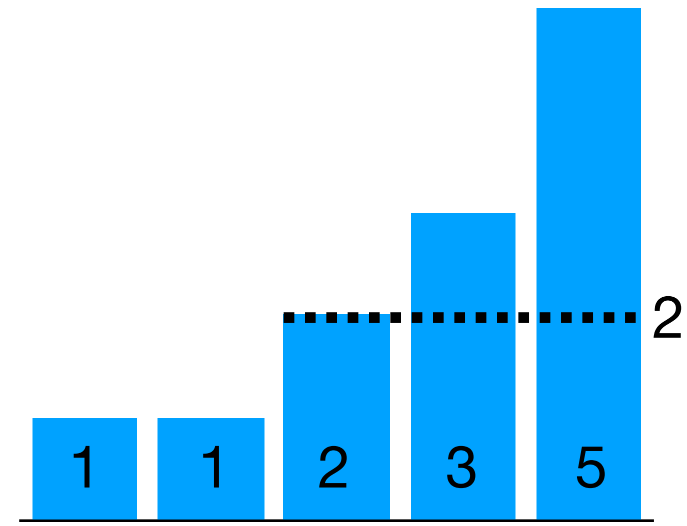
Implementation



- Maintain an array with the highest `WISH` received from each process: run in bounded space
- When received $n/2 + 1$ `WISH`es for views > yours, enter the minimal view in them
- Switch regardless of whether you called `advance`: allows lagging processes to catch up

Implementation





- Maintain an array with the highest `WISH` received from each process: run in bounded space

- When received $n/2 + 1$ `WISH`es for views > yours, enter the minimal view in them

- Switch regardless of whether you called `advance`: allows lagging processes to catch up

- Messages can get lost before GST and we have to cope with indirect connectivity: periodically resend the array with `WISH`es

Synchronizer correctness

- Proved correctness wrt our specification
- View synchronization mechanics hidden under the spec

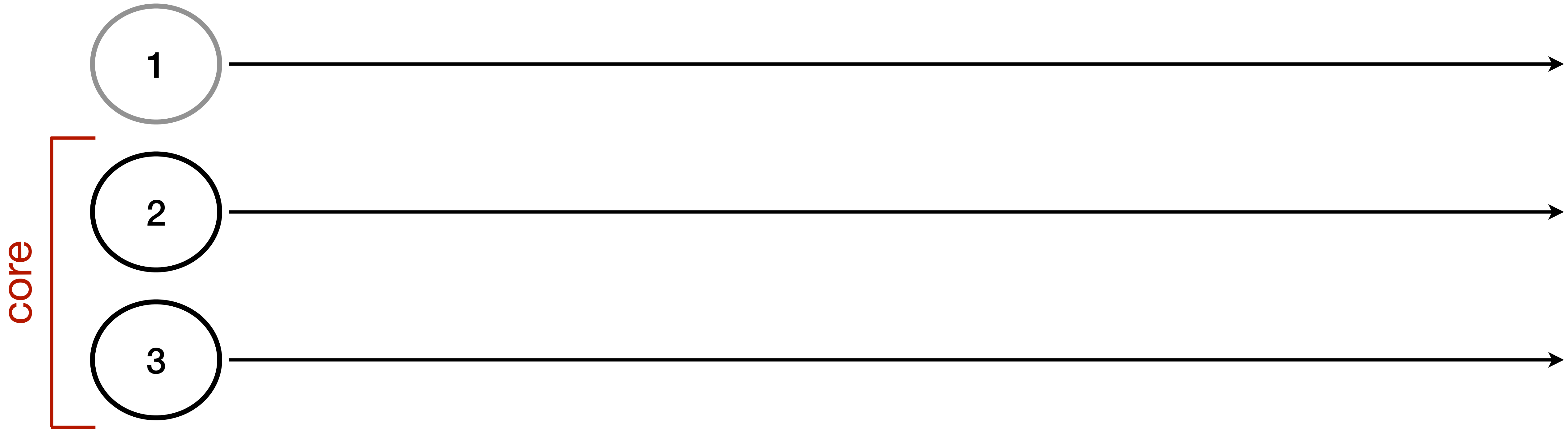
Consensus liveness

- Liveness property: any `propose()` invocation by a process in the connected core eventually returns
  - Can't guarantee liveness outside the connected core

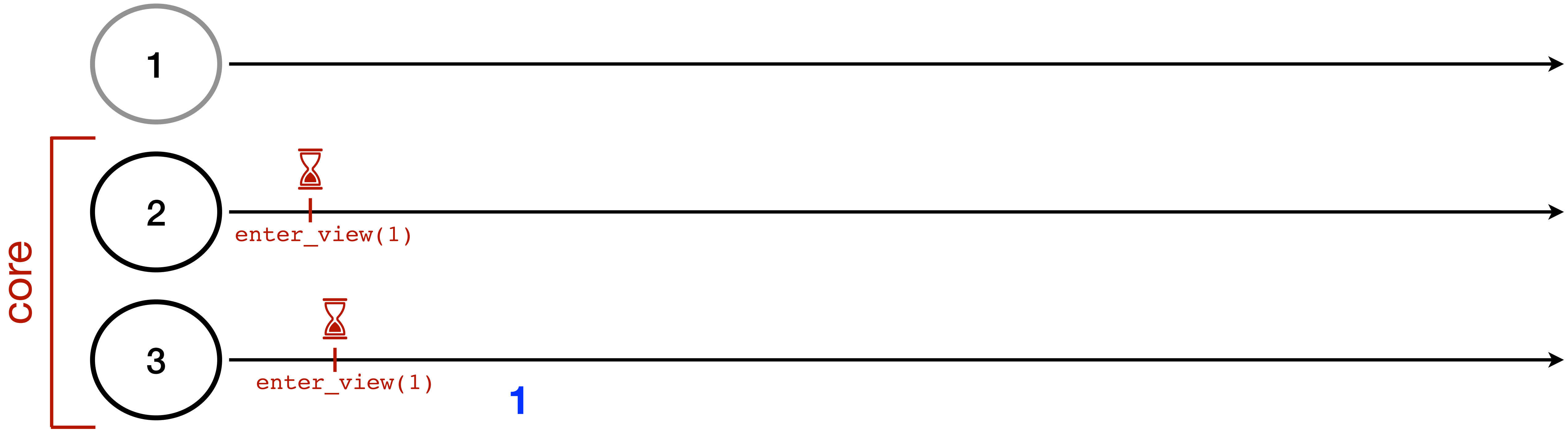
Consensus liveness

- Liveness property: any `propose()` invocation by a process in the connected core eventually returns
  - Can't guarantee liveness outside the connected core
- Implementation:
  - Single-decree Paxos on top of the view synchronizer
  - Leaders rotate round-robin: leader = view mod n
  - Processes monitor the leader behaviour and call `advance` if they suspect it's faulty or has a bad connectivity

Consensus

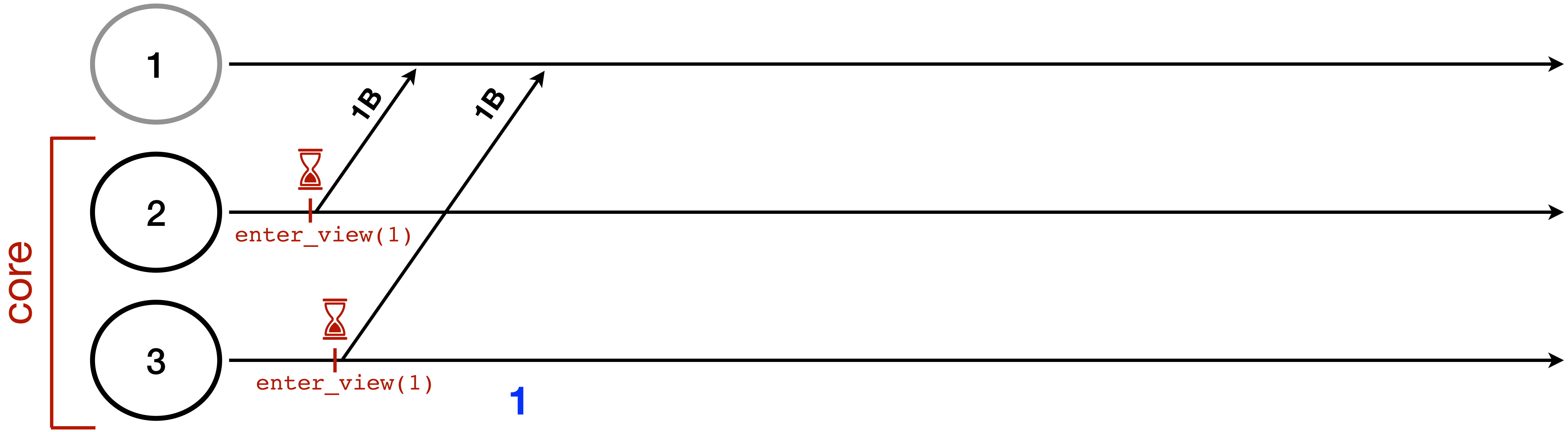


Consensus



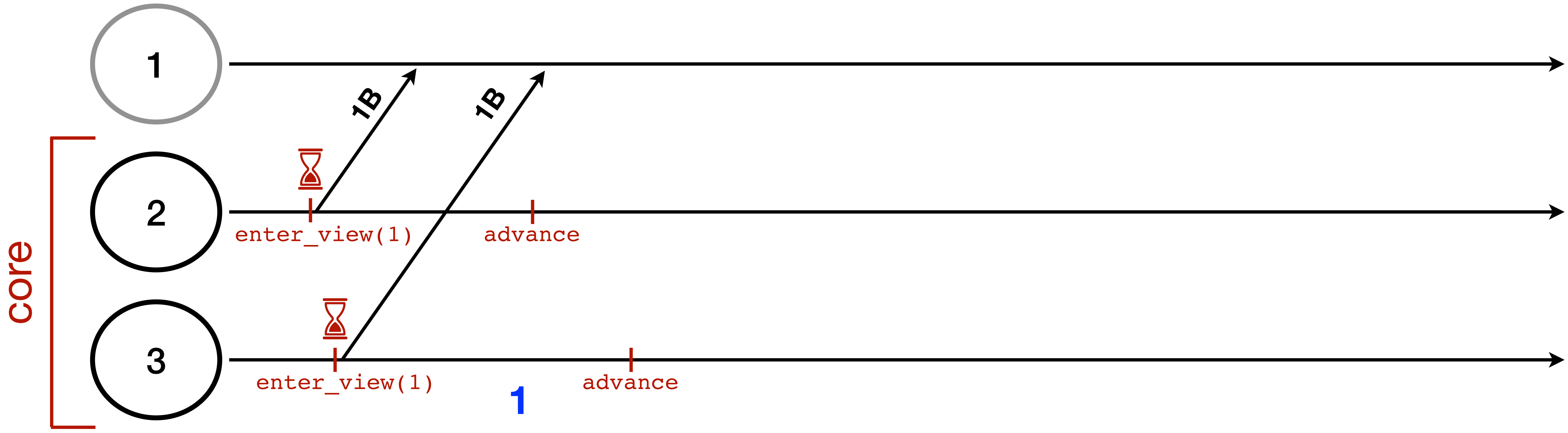
- When entering a view, send your value to the leader and set the timer for the expected decision delay: 3*diameter(core)*δ. If it expires, call `advance`

Consensus



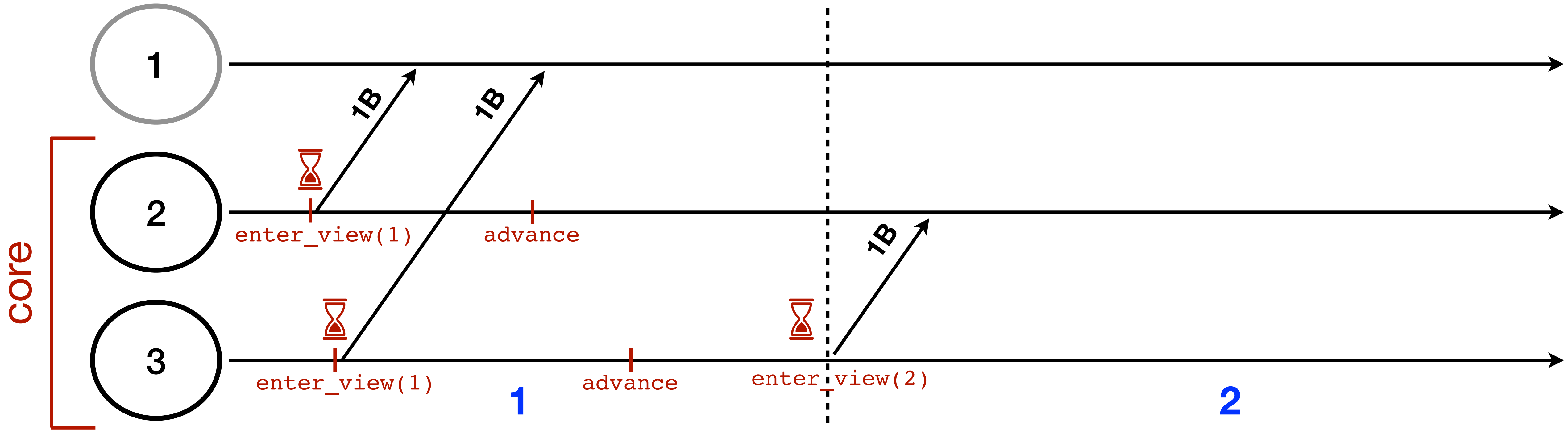
- When entering a view, send your value to the leader and set the timer for the expected decision delay: 3*diameter(core)*δ. If it expires, call `advance`

Consensus



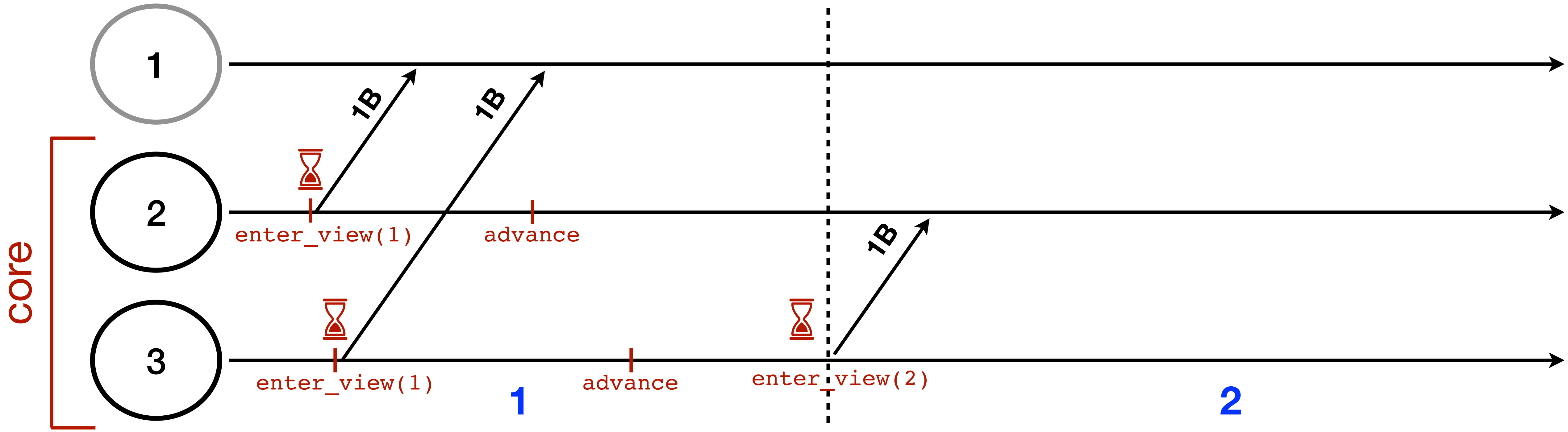
- When entering a view, send your value to the leader and set the timer for the expected decision delay: 3*diameter(core)*δ. If it expires, call `advance`
- Process 1 isn't connected ⟹ the timers eventually expire and {2,3} call `advance`

Consensus



- When entering a view, send your value to the leader and set the timer for the expected decision delay: 3*diameter(core)*δ. If it expires, call `advance`
- Process 1 isn't connected ⟹ the timers eventually expire and {2,3} call `advance`
- By Progress, some process from {2,3} enters view **2**

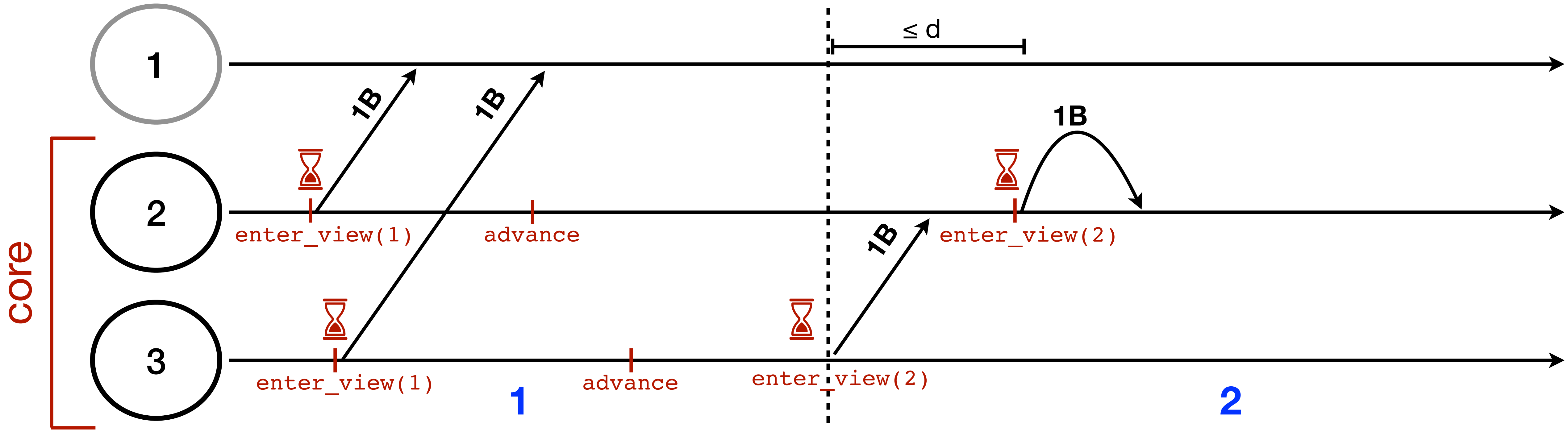
Consensus



- When entering a view, send your value to the leader and set the timer for the expected decision delay: 3*diameter(core)*δ. If it expires, call `advance`
- Process 1 isn't connected ⟹ the timers eventually expire and {2,3} call `advance`
- By Progress, some process from {2,3} enters view **2**

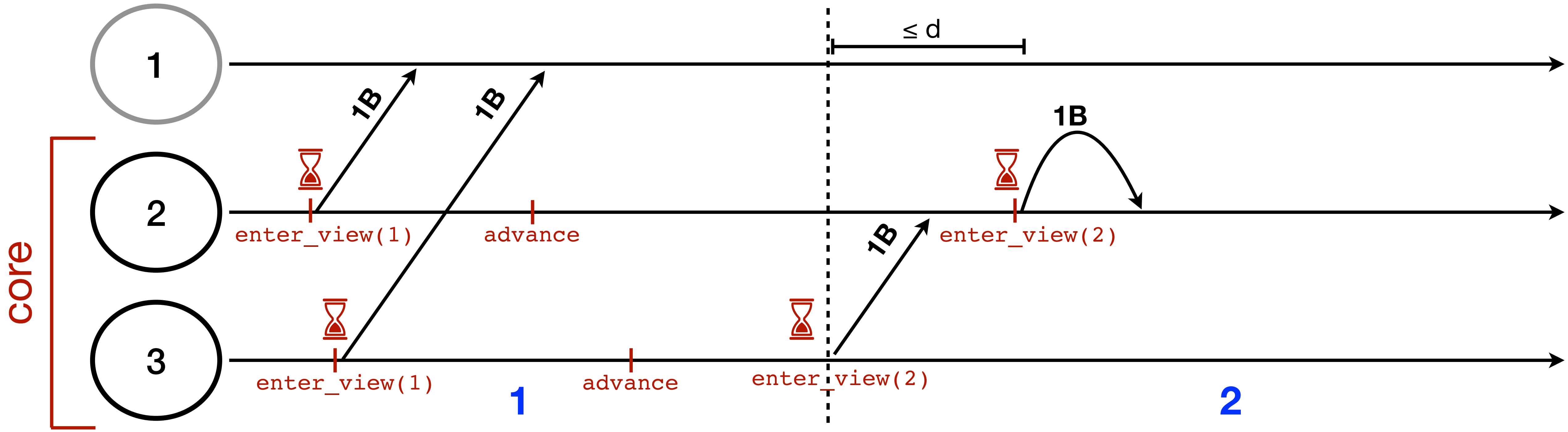
> Some process from the core will enter *v* + 1 if more than *n*/2 processes from the core invoke `advance` in *v*

Consensus



- When entering a view, send your value to the leader and set the timer for the expected decision delay: 3*diameter(core)*δ. If it expires, call `advance`
- Process 1 isn't connected ⟹ the timers eventually expire and {2,3} call `advance`
- By Progress, some process from {2,3} enters view **2**
- By Bounded entry, process 2 will promptly enter view **2** within d = δ * diameter(core)

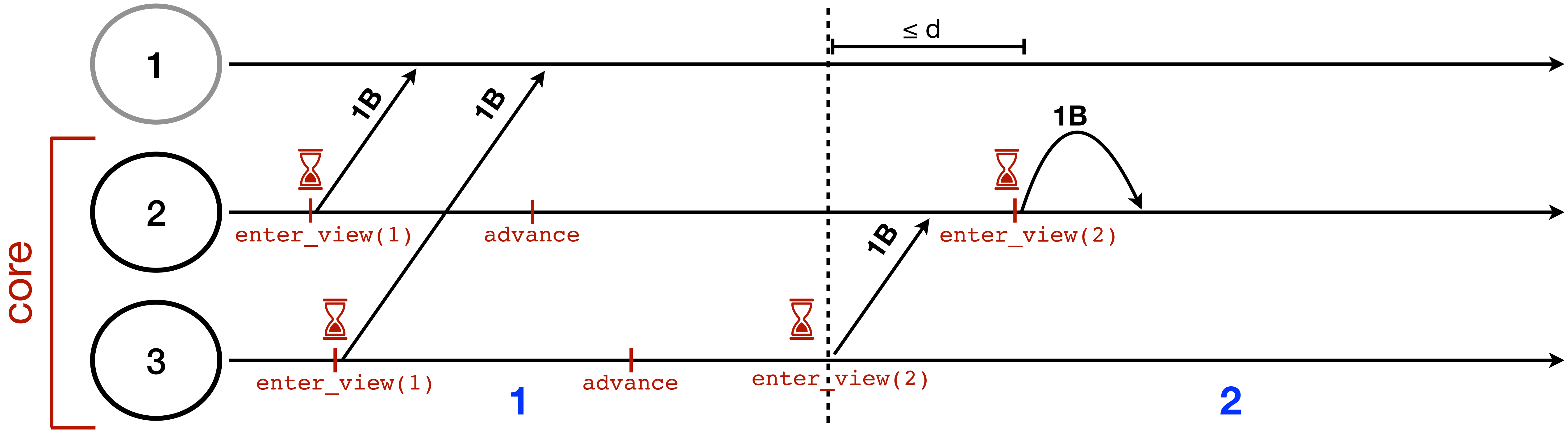
Consensus



- When entering a view, send your value to the leader and set the timer for the expected decision delay: 3*diameter(core)*δ. If it expires, call `advance`
- Process 1 isn't connected ⟹ the timers eventually expire and {2,3} call `advance`
- By Progress, some process from {2,3} enters view **2**
- By Bounded entry, process 2 will promptly enter view **2** within d = δ * diameter(core)

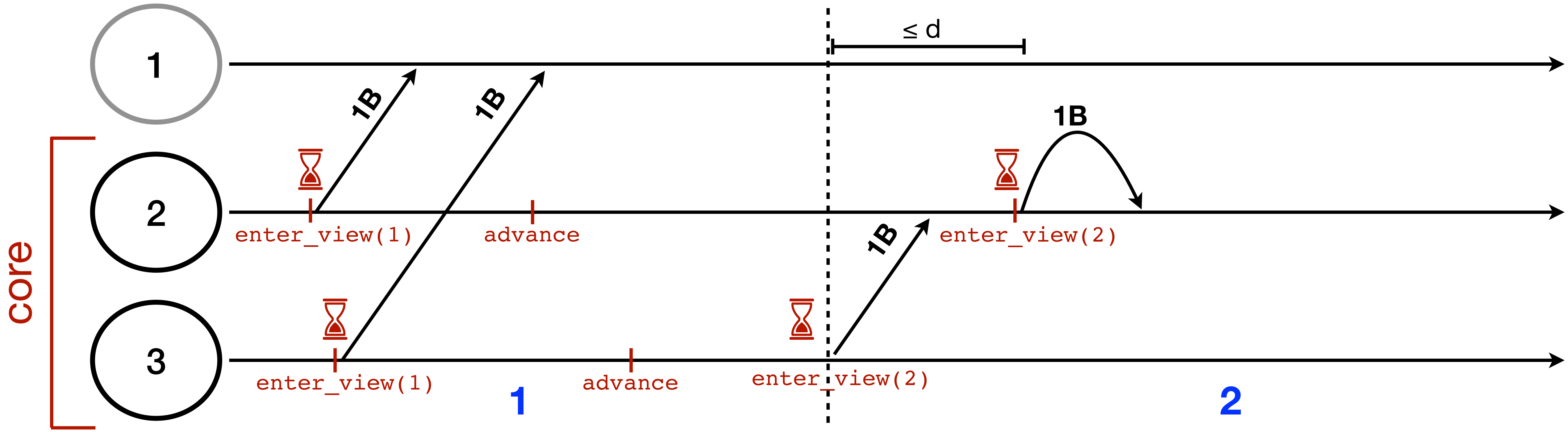
If a process from the core enters *v*, then all processes from the core will enter *v* within *d*, provided $v \geq \mathcal{V}$ and no process from the core attempts to advance to a higher view within d

Consensus



- When entering a view, send your value to the leader and set the timer for the expected decision delay: 3*diameter(core)*δ. If it expires, call `advance`
- Process 1 isn't connected ⟹ the timers eventually expire and {2,3} call `advance`
- By Progress, some process from {2,3} enters view **2**
- By Bounded entry, process 2 will promptly enter view **2** within d = δ * diameter(core)
- Nobody calls `advance` until a timer expires ⟹ by Validity, {2,3} stay in view **2** until this

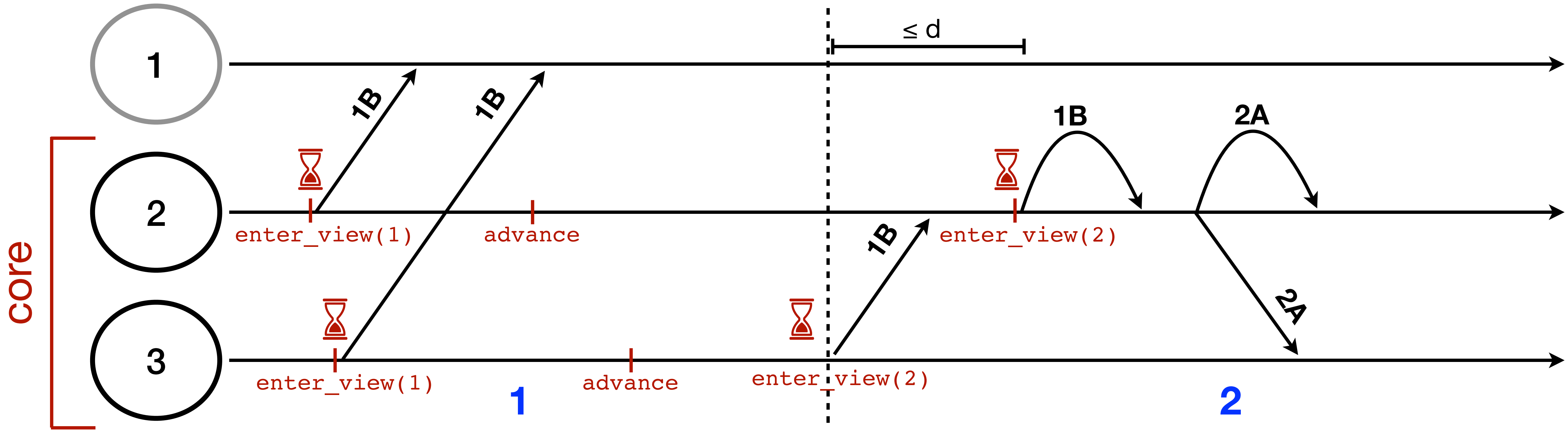
Consensus



- When entering a view, send your value to the leader and set the timer for the expected decision delay: 3*diameter(core)*δ. If it expires, call `advance`
- Process 1 isn't connected ⟹ the timers eventually expire and {2,3} call `advance`
- By Progress, some process from {2,3} enters view **2**
- By Bounded entry, process 2 will promptly enter view **2** within d = δ * diameter(core)
- Nobody calls `advance` until a timer expires ⟹ by Validity, {2,3} stay in view **2** until this

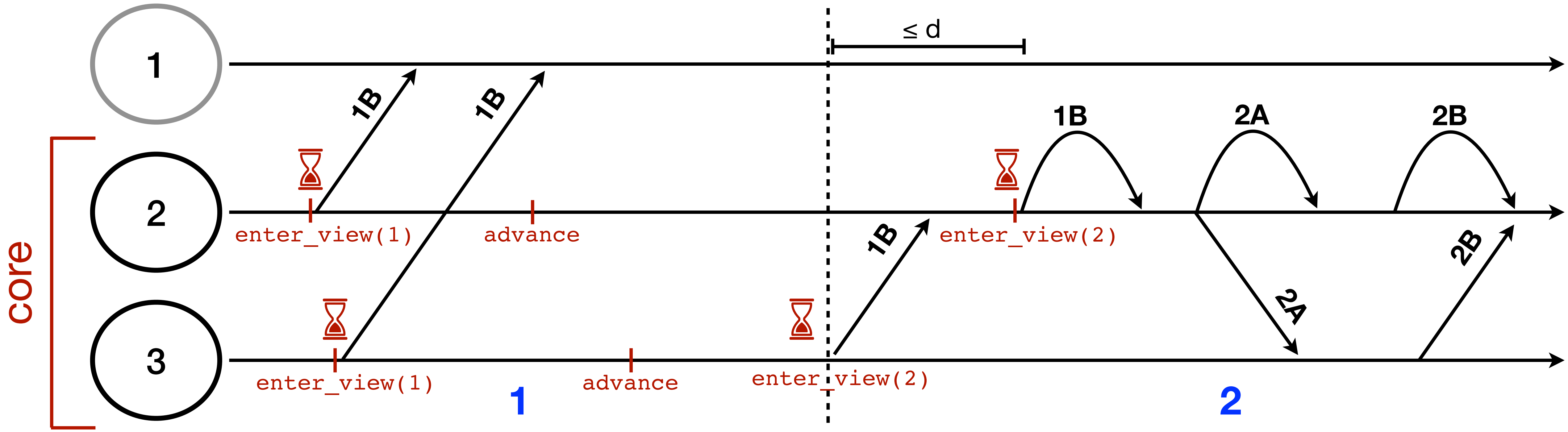
> A process can enter $v + 1$ only if some process from the core has invoked `advance` in v

Consensus



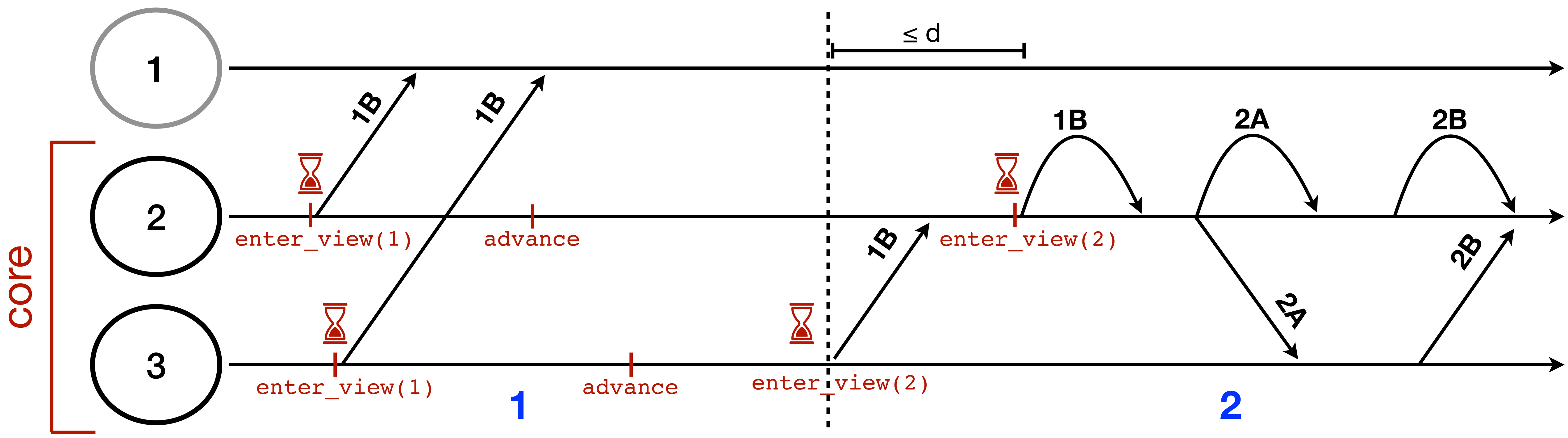
- When entering a view, send your value to the leader and set the timer for the expected decision delay: 3*diameter(core)*δ. If it expires, call `advance`
- Process 1 isn't connected ⟹ the timers eventually expire and {2,3} call `advance`
- By Progress, some process from {2,3} enters view **2**
- By Bounded entry, process 2 will promptly enter view **2** within d = δ * diameter(core)
- Nobody calls `advance` until a timer expires ⟹ by Validity, {2,3} stay in view **2** until this

Consensus



- When entering a view, send your value to the leader and set the timer for the expected decision delay: 3*diameter(core)*δ. If it expires, call `advance`
- Process 1 isn't connected ⟹ the timers eventually expire and {2,3} call `advance`
- By Progress, some process from {2,3} enters view **2**
- By Bounded entry, process 2 will promptly enter view **2** within d = δ * diameter(core)
- Nobody calls `advance` until a timer expires ⟹ by Validity, {2,3} stay in view **2** until this

Consensus



- When entering a view, send your value to the leader and set the timer for the expected decision delay: 3*diameter(core)*δ. If it expires, call `advance`
- Process 1 isn't connected ⟹ the timers eventually expire and {2,3} call `advance`
- By Progress, some process from {2,3} enters view **2**
- By Bounded entry, process 2 will promptly enter view **2** within d = δ * diameter(core)
- Nobody calls `advance` until a timer expires ⟹ by Validity, {2,3} stay in view **2** until this
- Processes in the core will decide before any timer expires

Proving liveness

- Don't know $\delta \Longrightarrow$ increase timeouts when calling `advance`
- Proof - interplay between the properties of the consensus protocol and the synchronizer
- Top-level protocol proofs are simple, synchronizer proofs more complex
- The structure is reused for proofs of different protocols: in the Byzantine context, have given the first proof of liveness to PBFT

Conclusion

- Separating liveness from safety simplifies the design and proofs of consensus protocols
- Synchronizers are widely applicable, from crash to Byzantine failures
- CAP is not everything. Now working on generalizing lower bounds to non-cardinality based failure patterns