Lecture #3: Solving SMR with Crash Faults and Synchrony

COMS 4995-001:
The Science of Blockchains

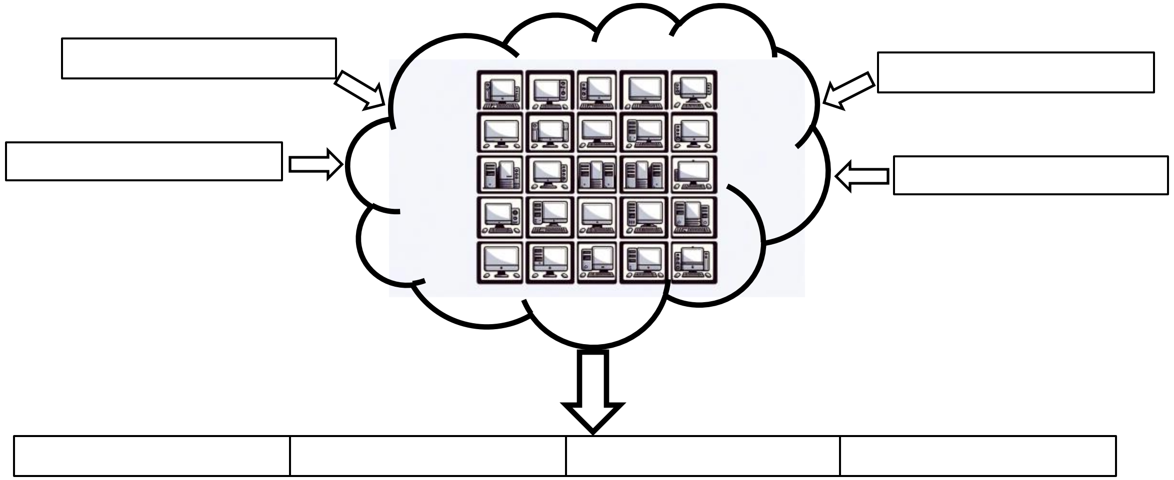
URL: https://timroughgarden.org/s25/

Tim Roughgarden

State Machine Replication (SMR)

SMR: version of consensus appropriate for a blockchain protocol.

- “state machine” = for us, current state of virtual machine
- “replication” = all validators perform same state transitions
- “clients” submit transactions (“txs”) to validators
- each validator maintains an append-only list of finalized txs (a.k.a. “log” or “history”)



State Machine Replication (SMR)

SMR: version of consensus appropriate for a blockchain protocol.

- "state machine" = for us, current state of virtual machine
- "replication" = all validators perform same state transitions
- "clients" submit transactions ("txs") to validators
- each validator maintains an append-only list of finalized txs (a.k.a. "log" or "history")

Goal: a protocol that satisfies consistency and liveness.

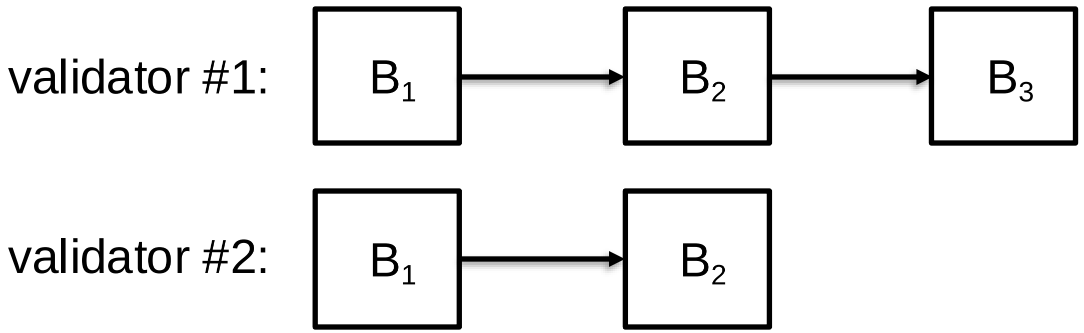
Consistency and Liveness

Goal: a protocol that satisfies consistency and liveness.

Consistency: all validators agree on a transaction sequence.

- ok if some lag behind, but no disagreements allowed!

acceptable:



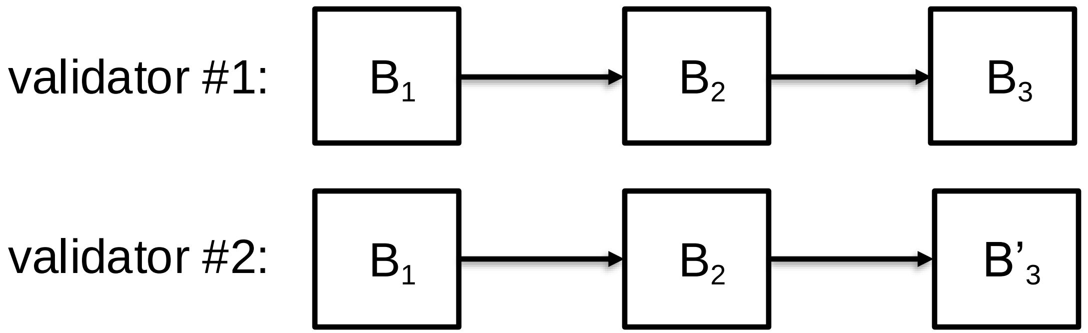
Consistency and Liveness

Goal: a protocol that satisfies consistency and liveness.

Consistency: all validators agree on a transaction sequence.

- ok if some lag behind, but no disagreements allowed!

unacceptable:

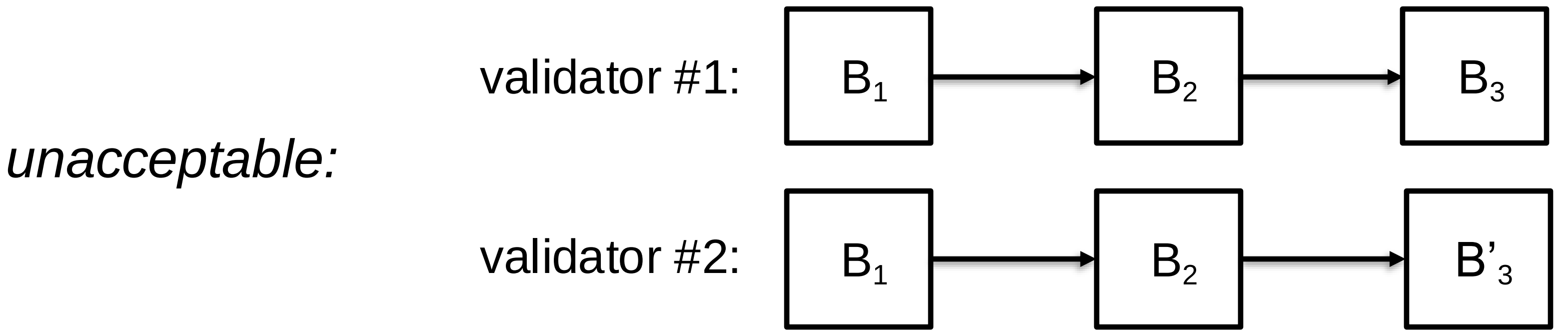


Consistency and Liveness

Goal: a protocol that satisfies consistency and liveness.

Consistency: all validators agree on a transaction sequence.

- ok if some lag behind, but no disagreements allowed!



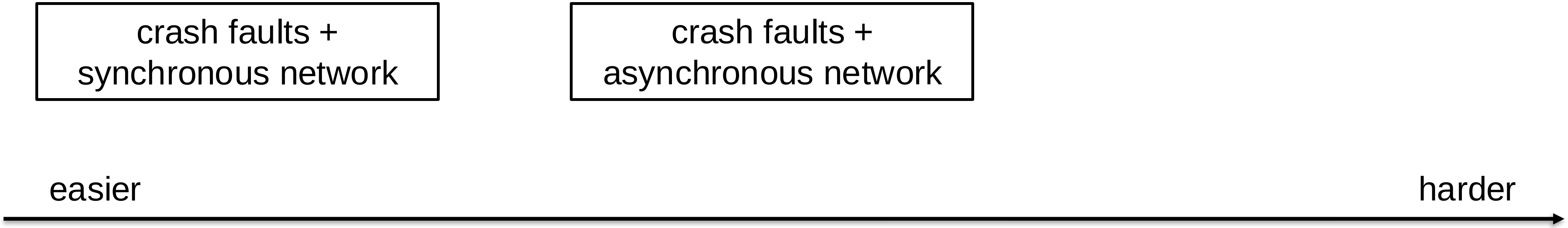
Liveness: every valid transaction submitted by a client eventually added to validators' local histories/chains.

A Road Map to Practical SMR Protocols

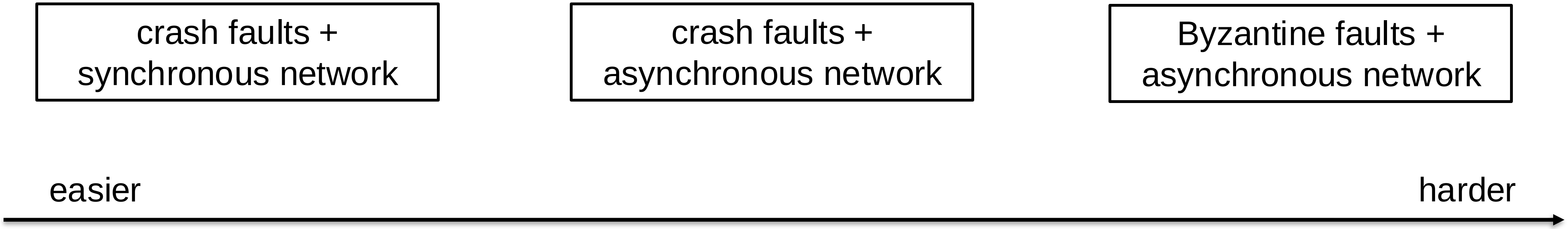
easier harder

A Road Map to Practical SMR Protocols

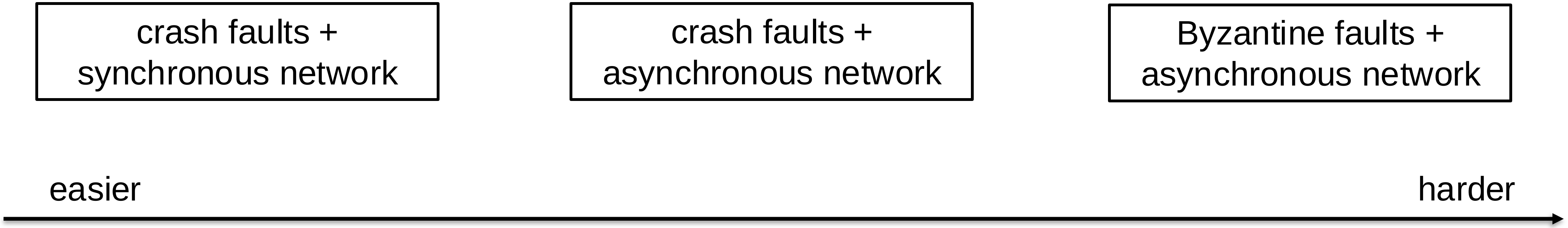
A Road Map to Practical SMR Protocols



A Road Map to Practical SMR Protocols



A Road Map to Practical SMR Protocols



Expectations:

1. More positive results (i.e., good SMR protocols) toward the left.
2. More impossibility results (i.e., SMR unsolvable) toward the right.
3. Simpler protocols toward the left, more complex toward the right.

Goals for Lecture #3

1. The challenge of crash faults.
   – simple, but already messes up Protocol A from last time

2. Solving SMR with crash faults and a synchronous network.
   – already forces us to introduce some important design principles
   – good warm-up for more challenging and blockchain-relevant settings

3. Asynchrony: challenges and compromises.
   – an impossibility result motivates a “sweet spot” synchronous-asynchronous hybrid model

SMR: Crash Faults and Synchrony

Crash faults: every validator correctly executes the protocol except it may crash (forever) at some point.

SMR: Crash Faults and Synchrony

Crash faults: every validator correctly executes the protocol except it may crash (forever) at some point.

Synchronous network: for known parameter Δ, every msg delivered in ≤ Δ time steps

SMR: Crash Faults and Synchrony

Crash faults: every validator correctly executes the protocol except it may crash (forever) at some point.

Synchronous network: for known parameter Δ, every msg delivered in $\leq \Delta$ time steps

Recall: Protocol A [code run by every validator]

SMR: Crash Faults and Synchrony

Crash faults: every validator correctly executes the protocol except it may crash (forever) at some point.

Synchronous network: for known parameter Δ, every msg delivered in $\leq \Delta$ time steps

Recall: Protocol A [code run by every validator]

- define "view" = Δ consecutive timesteps

SMR: Crash Faults and Synchrony

Crash faults: every validator correctly executes the protocol except it may crash (forever) at some point.

Synchronous network: for known parameter Δ, every msg delivered in ≤ Δ time steps

Recall: Protocol A [code run by every validator]

- define "view" = Δ consecutive timesteps
- validators take turns as leader (round-robin, one per view)
  - plays the role of a temporary dictator (to coordinate others)
  - recall assumptions of known validator set, shared global clock

SMR: Crash Faults and Synchrony

Recall: Protocol A [code run by every validator]

- define view = Δ consecutive timesteps
- validators take turns as leader (round-robin, one per view)

SMR: Crash Faults and Synchrony

Recall: Protocol A [code run by every validator]

- define view = Δ consecutive timesteps
- validators take turns as leader (round-robin, one per view)
- at time $\Delta \cdot v$: [i.e., at beginning of view v]
  - leader assembles block B = all not-yet-included valid txs it knows about
  - leader sends B to all other validators

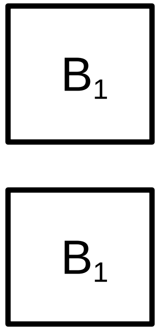
SMR: Crash Faults and Synchrony

Recall: Protocol A [code run by every validator]

- define view = Δ consecutive timesteps
- validators take turns as leader (round-robin, one per view)
- at time $\Delta \cdot v$: [i.e., at beginning of view v]
  - leader assembles block B = all not-yet-included valid txs it knows about
  - leader sends B to all other validators
- at time $\Delta \cdot v + \Delta$: [i.e., at end of view v, before view v+1]
  - if validator i received a block B from the leader by this time:
    - validator i appends B to its local history

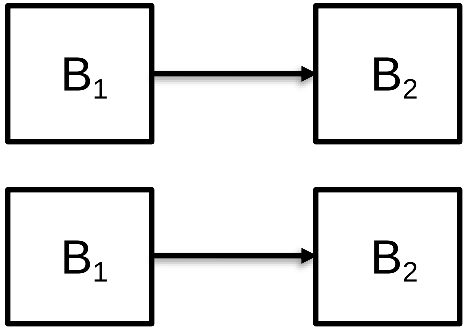
Why Protocol A Can't Handle Crashes

Problem: leader might crash after sending B to some but not all validators [➔ could lead to a consistency violation].



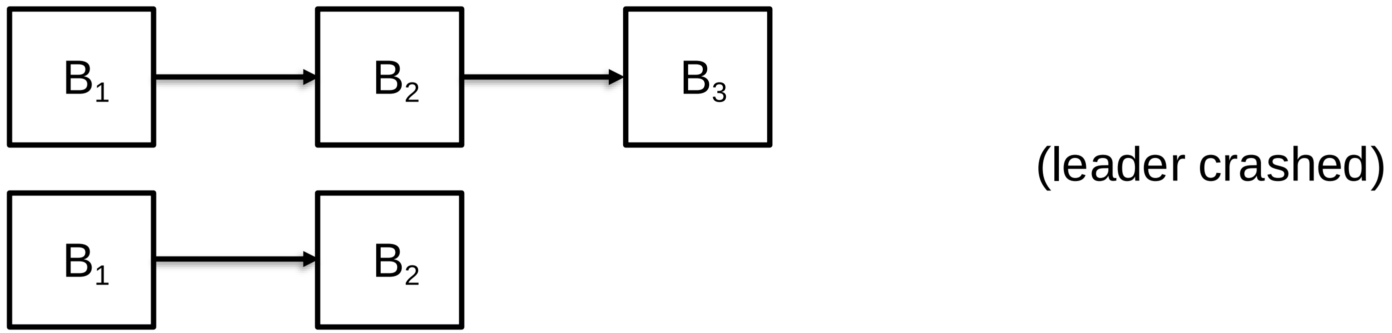
Why Protocol A Can't Handle Crashes

Problem: leader might crash after sending B to some but not all validators [➔ could lead to a consistency violation].



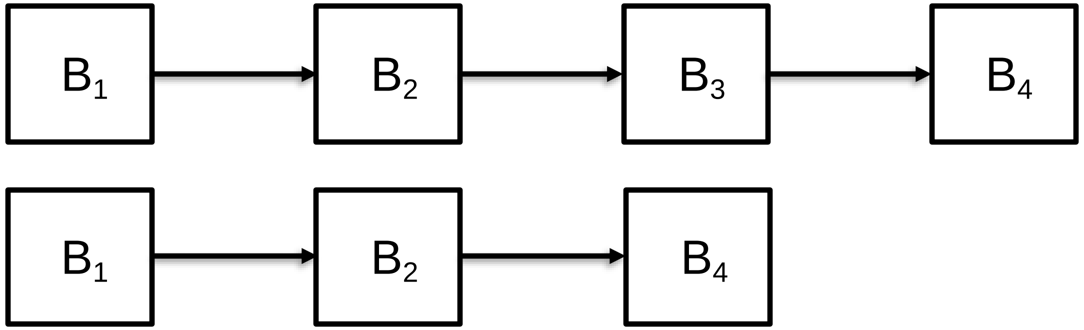
Why Protocol A Can't Handle Crashes

Problem: leader might crash after sending B to some but not all validators [➔ could lead to a consistency violation].



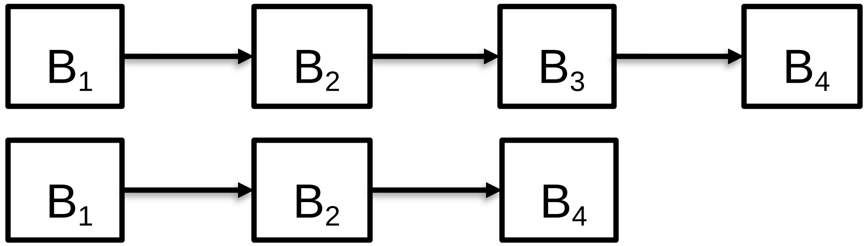
Why Protocol A Can't Handle Crashes

Problem: leader might crash after sending B to some but not all validators [➔ could lead to a consistency violation].



Why Protocol A Can't Handle Crashes

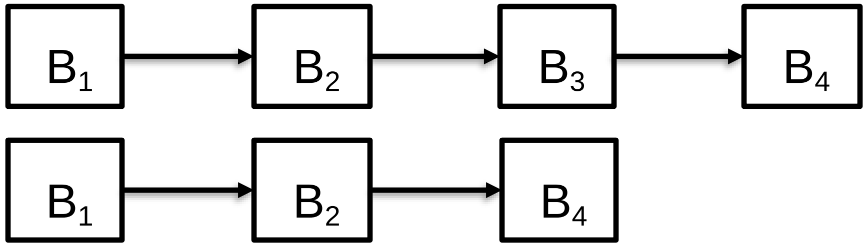
Problem: leader might crash after sending B to some but not all validators [➔ could lead to a consistency violation].



Fix:

Why Protocol A Can't Handle Crashes

Problem: leader might crash after sending B to some but not all validators [➔ could lead to a consistency violation].

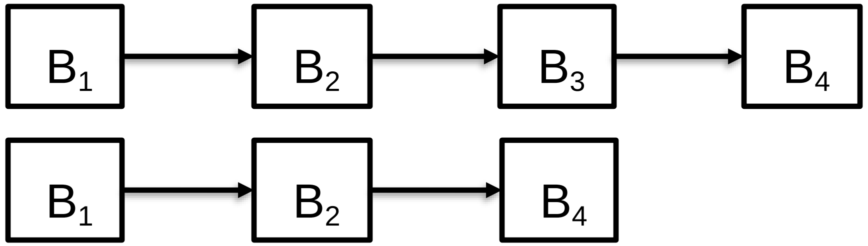


Fix:

1. validators update next leader as to their current history
   – to make sure leader is up-to-date before proposing

Why Protocol A Can't Handle Crashes

Problem: leader might crash after sending B to some but not all validators [➔ could lead to a consistency violation].



Fix:

1. validators update next leader as to their current history
   – to make sure leader is up-to-date before proposing
2. send entire history/chain, not just latest block
   – crashes ➔ validator may learn about many new blocks at same time
   – will make more practical using commitments in Part II

SMR: Crash Faults and Synchrony

Protocol B [code run by every validator]

SMR: Crash Faults and Synchrony

Protocol B [code run by every validator]

- define view = 2Δ consecutive timesteps
- validator i maintains local chain C_i (i.e., sequence of blocks)
- validators take turns as leader (round-robin, one per view)

SMR: Crash Faults and Synchrony

Protocol B [code run by every validator]

- define view = 2Δ consecutive timesteps
- validator i maintains local chain C_i (i.e., sequence of blocks)
- validators take turns as leader (round-robin, one per view)
- at time $\Delta \cdot v$: [i.e., at beginning of view v]
  - each validator i sends current chain C_i to v's leader ℓ

SMR: Crash Faults and Synchrony

Protocol B [code run by every validator]

- define view = 2Δ consecutive timesteps
- validator i maintains local chain C_i (i.e., sequence of blocks)
- validators take turns as leader (round-robin, one per view)

• at time $\Delta \cdot v$: [i.e., at beginning of view v]

- each validator i sends current chain C_i to v's leader ℓ

• at time $\Delta \cdot v + \Delta$:

- let C = longest chain received by ℓ in this view
- ℓ assembles B := all not-yet-included (in C) valid txs it knows about
- ℓ sends C^* := (C,B) to all other validators

SMR: Crash Faults and Synchrony

Protocol B [code run by every validator]

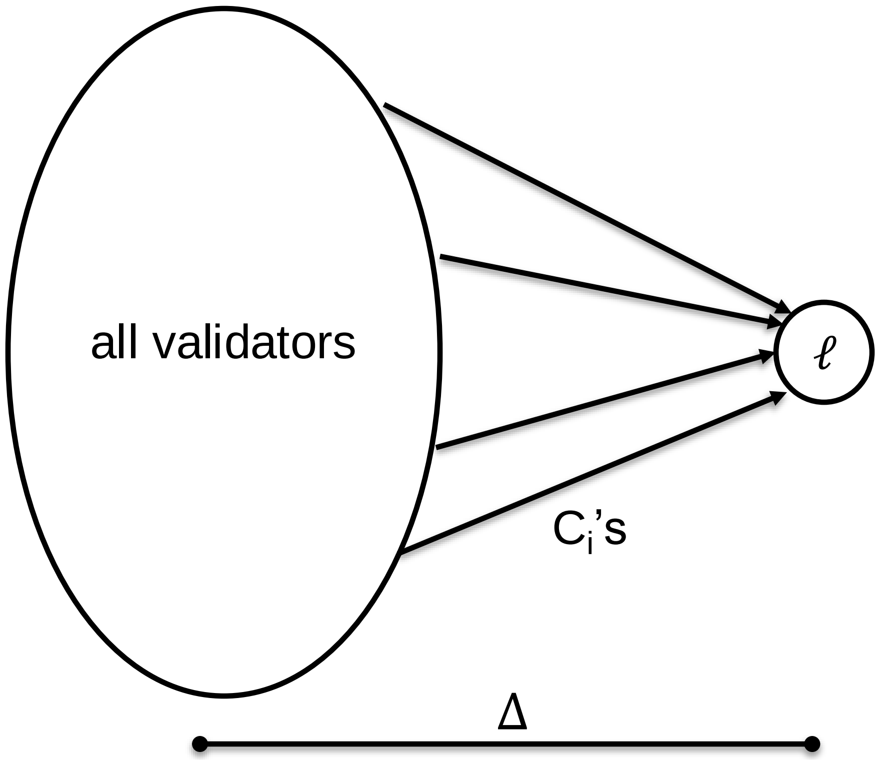
- at time $\Delta \cdot v$: [i.e., at beginning of view v]
  - each validator i sends current chain C_i to v's leader ℓ
- at time $\Delta \cdot v + \Delta$:
  - let C = longest chain received by ℓ in this view
  - ℓ assembles B := all not-yet-included (in C) valid txs it knows about
  - ℓ sends C^* := (C,B) to all other validators

SMR: Crash Faults and Synchrony

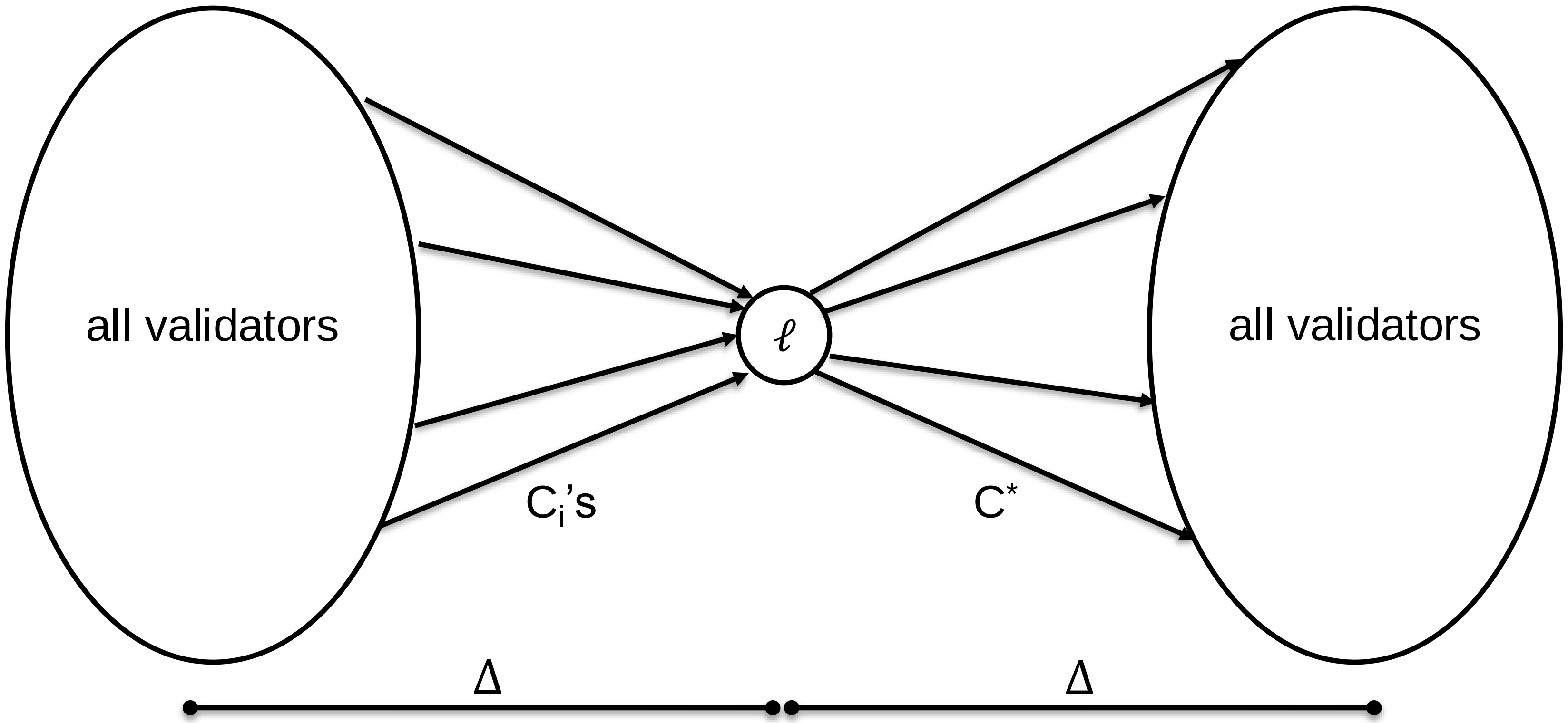
Protocol B [code run by every validator]

- at time $\Delta \cdot v$: [i.e., at beginning of view v]
  - each validator i sends current chain C_i to v's leader ℓ
- at time $\Delta \cdot v + \Delta$:
  - let C = longest chain received by ℓ in this view
  - ℓ assembles B := all not-yet-included (in C) valid txs it knows about
  - ℓ sends C^* := (C,B) to all other validators
- at time $\Delta \cdot v + 2\Delta$: [i.e., at end of view v, before view v+1]
  - if validator i receives a new chain C^* from ℓ by this time:
    - validator i updates $C_i := C^*$

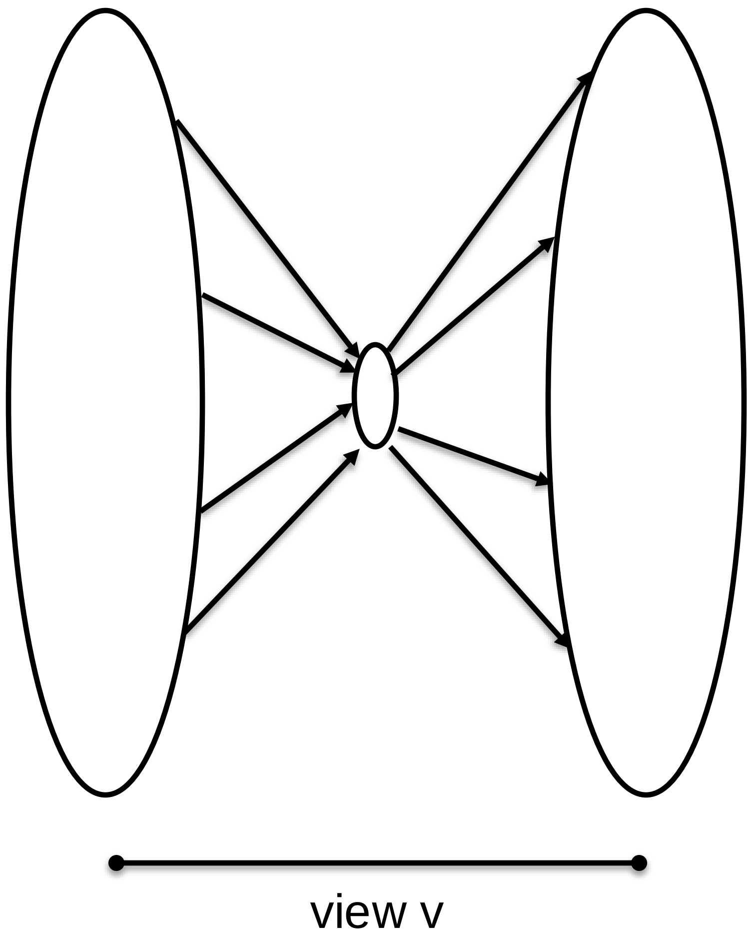
Picture of One View



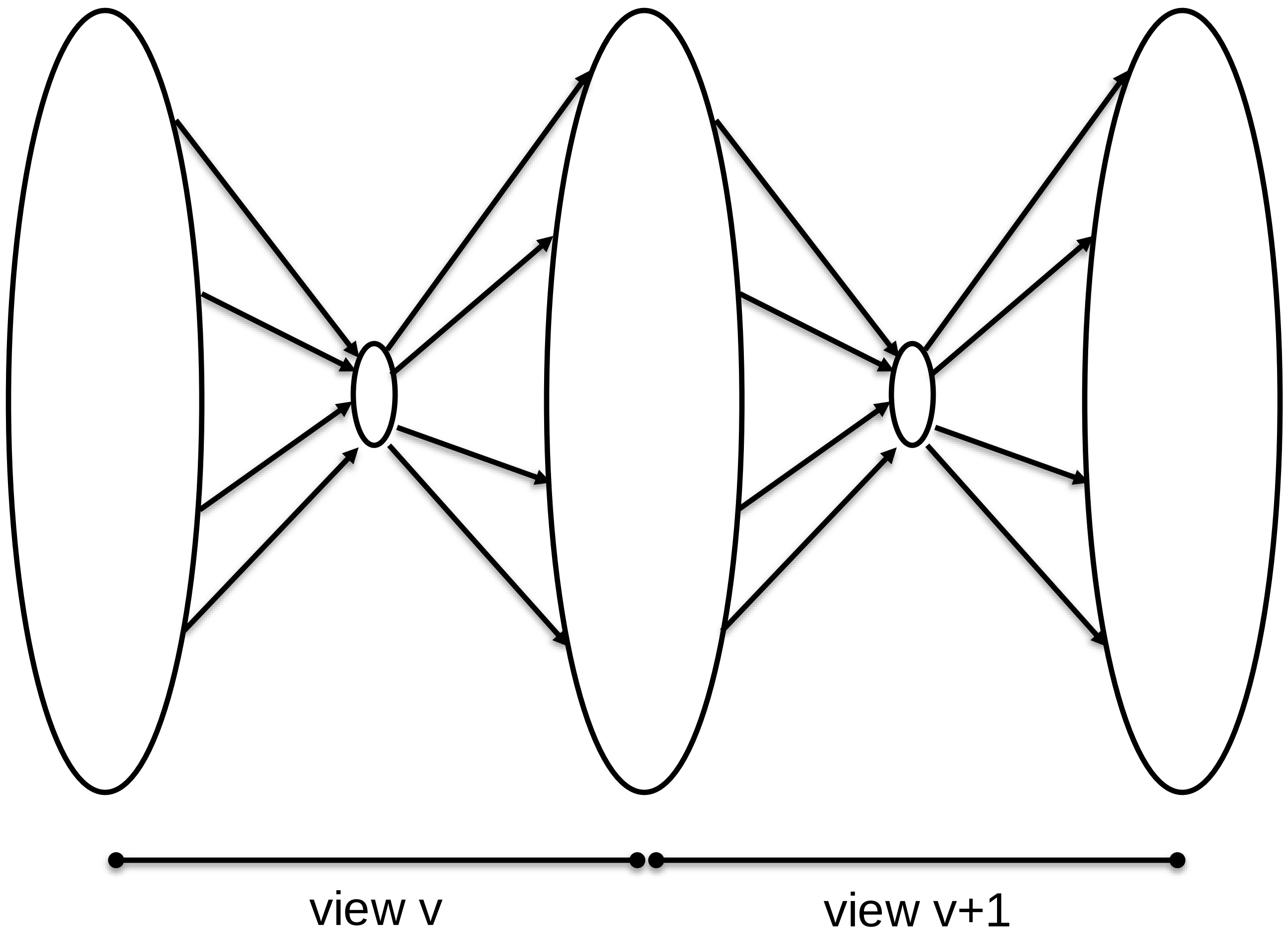
Picture of One View



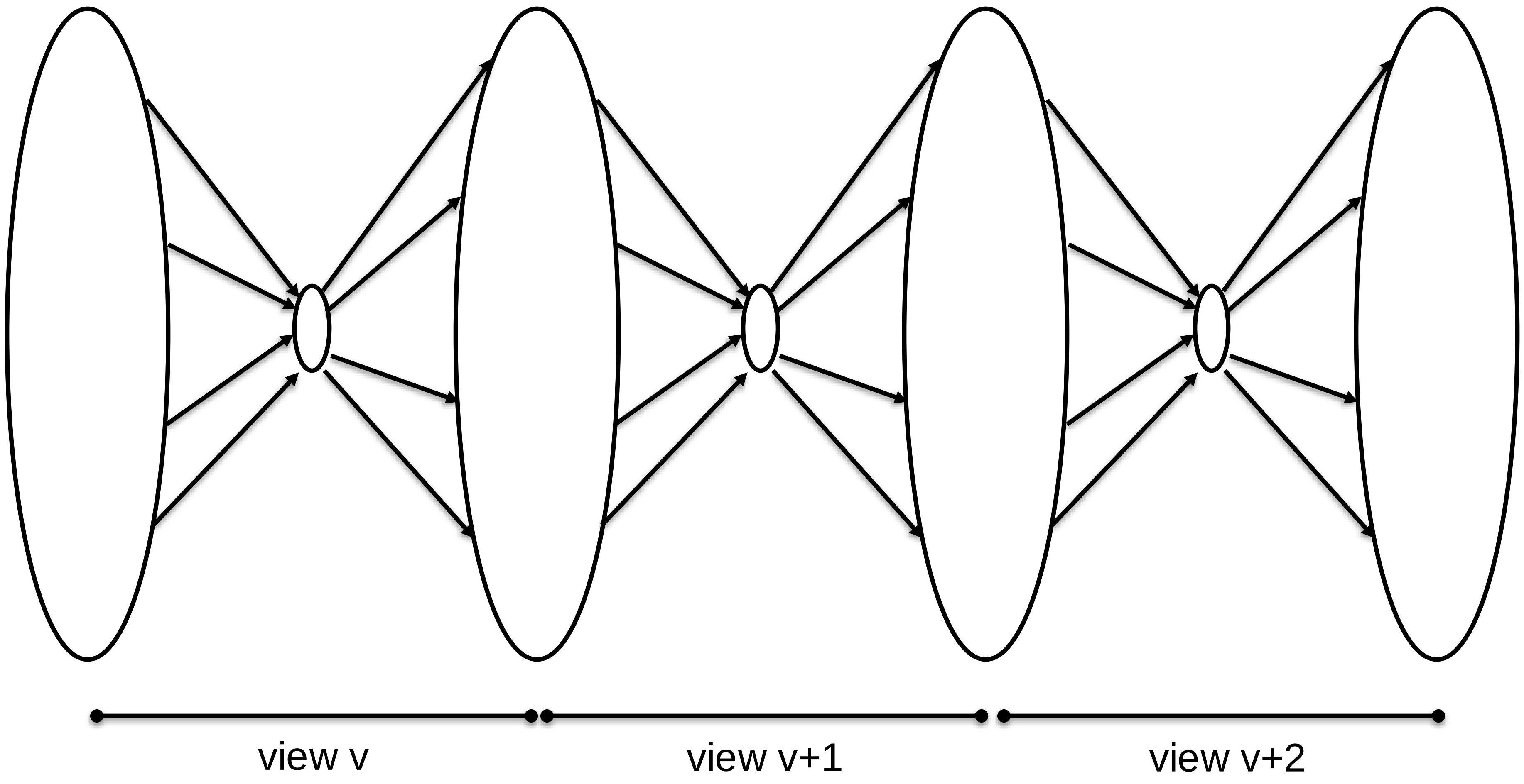
Zooming Out



Zooming Out



Zooming Out



Protocol B: An Example Execution

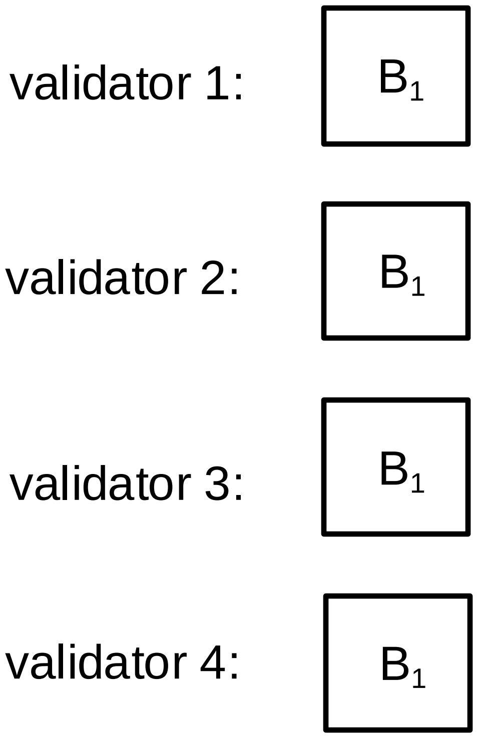
validator 1:

validator 2:

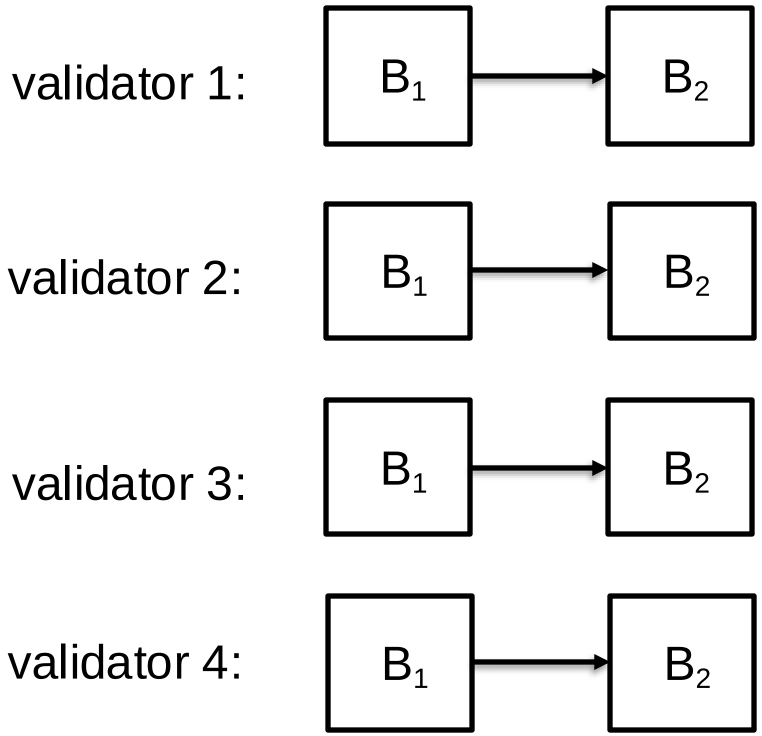
validator 3:

validator 4:

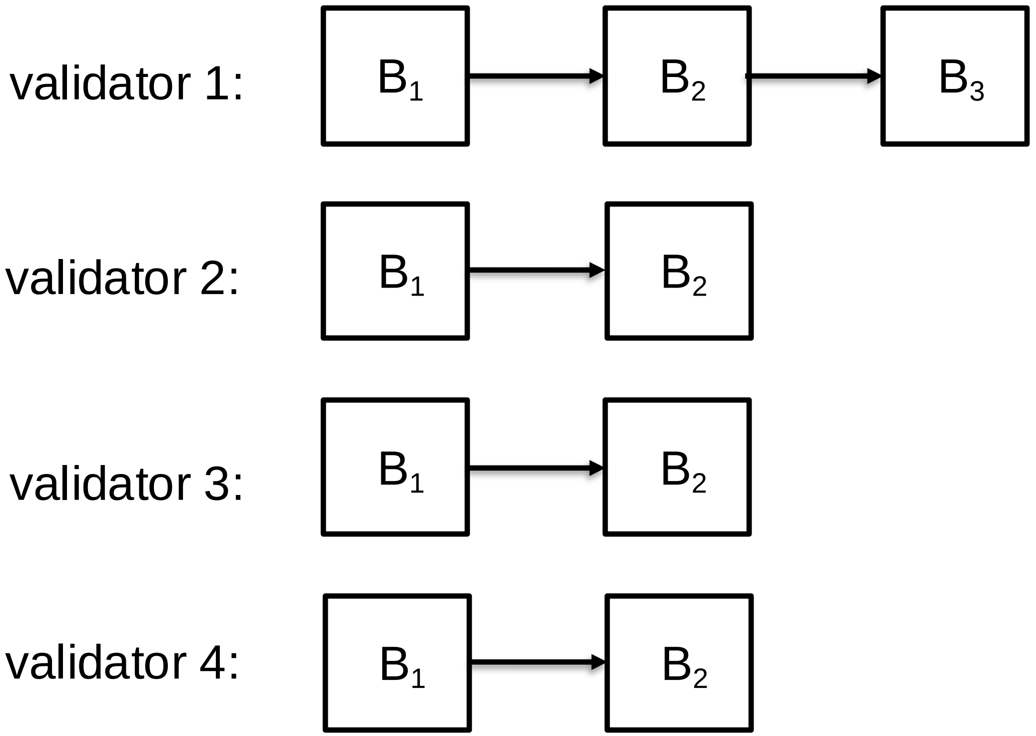
Protocol B: An Example Execution



Protocol B: An Example Execution

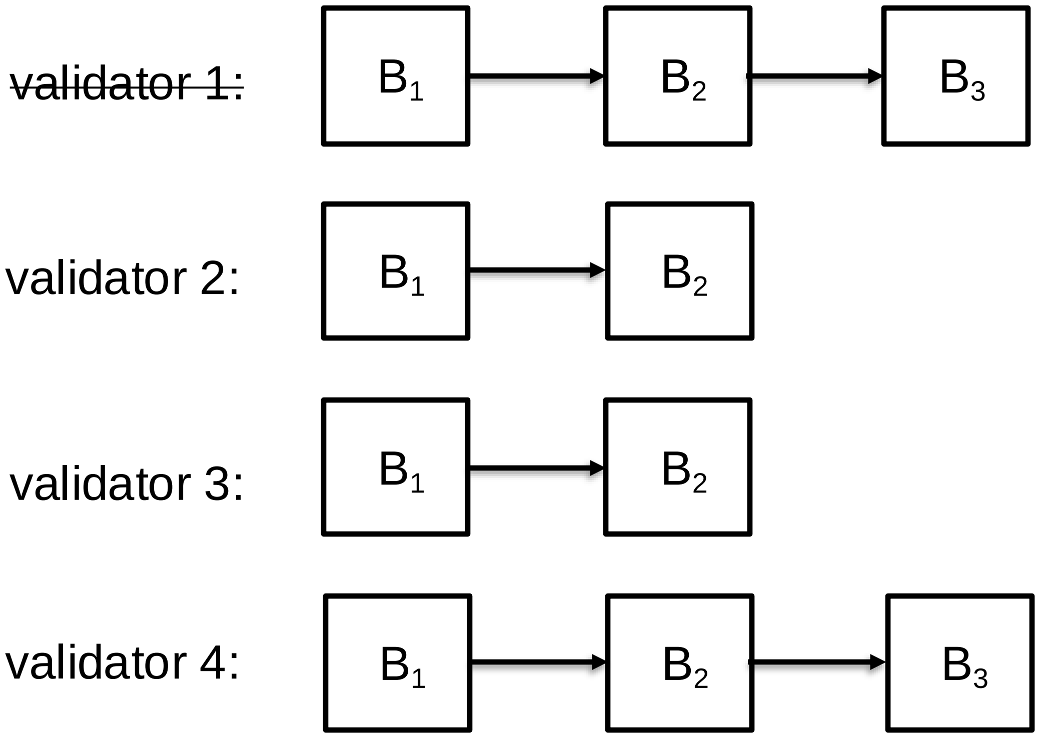


Protocol B: An Example Execution



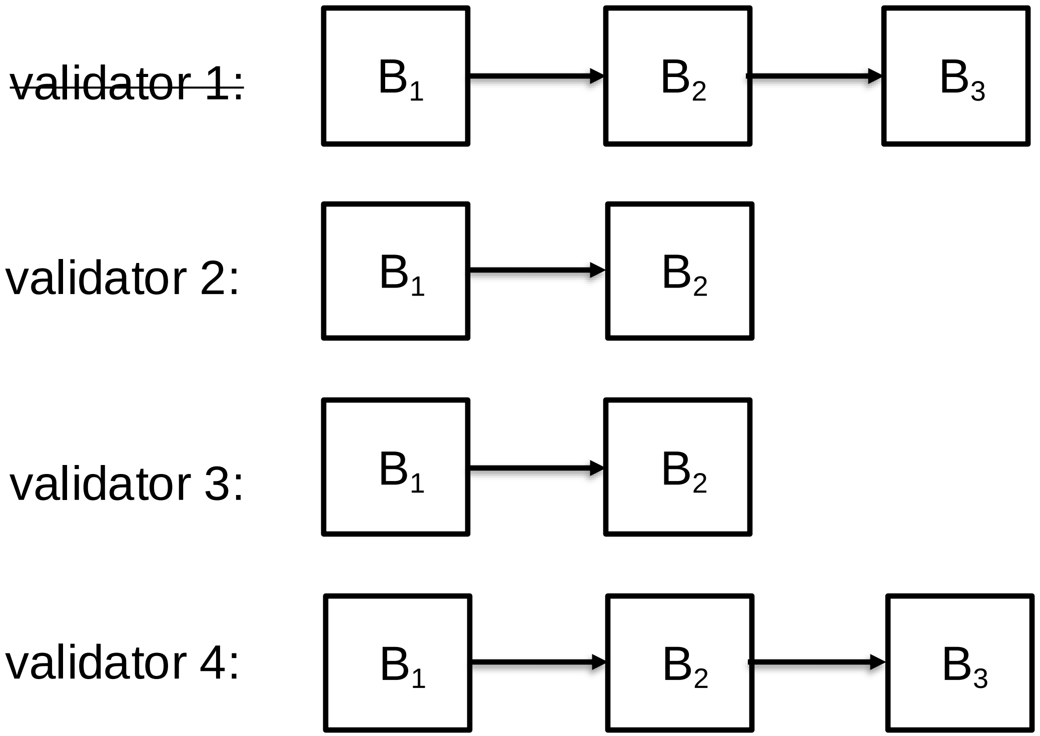
(validator 1 is next leader, prepares its proposal)

Protocol B: An Example Execution



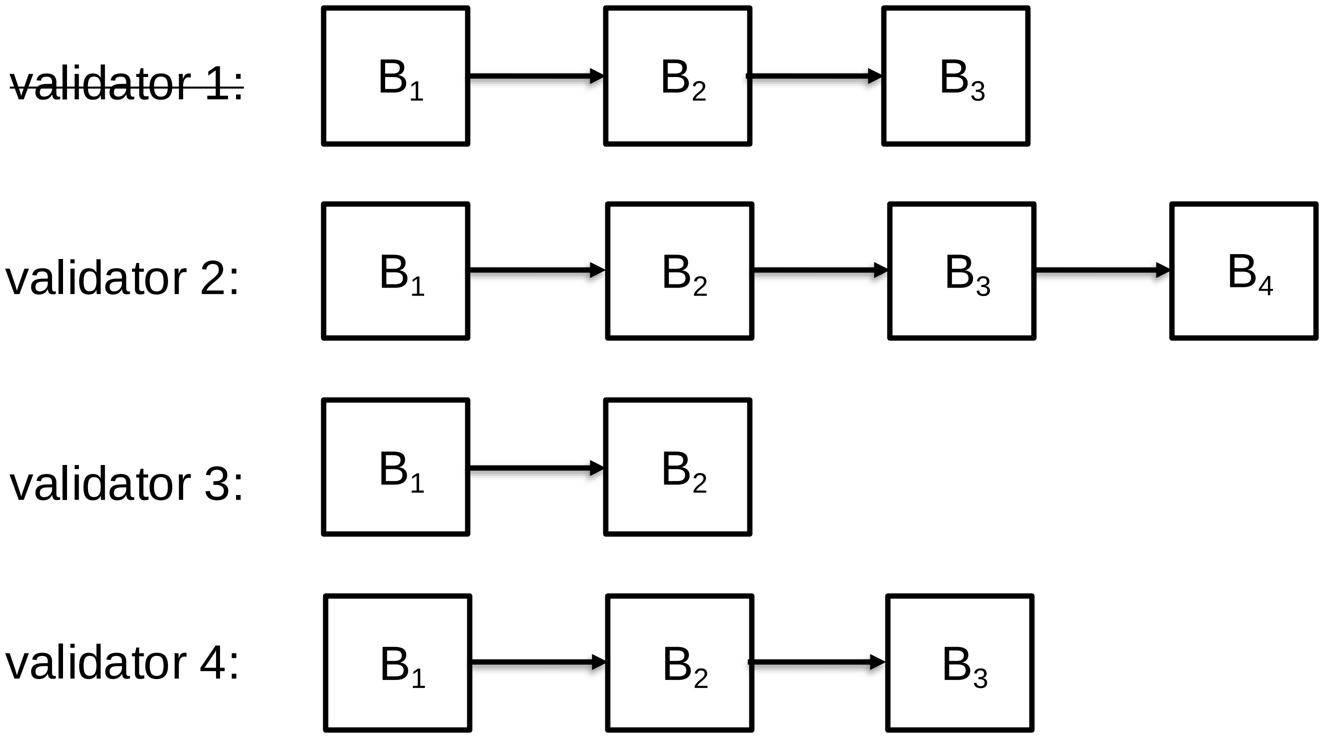
(validator 1 crashes after sending its proposal only to validator 4)

Protocol B: An Example Execution



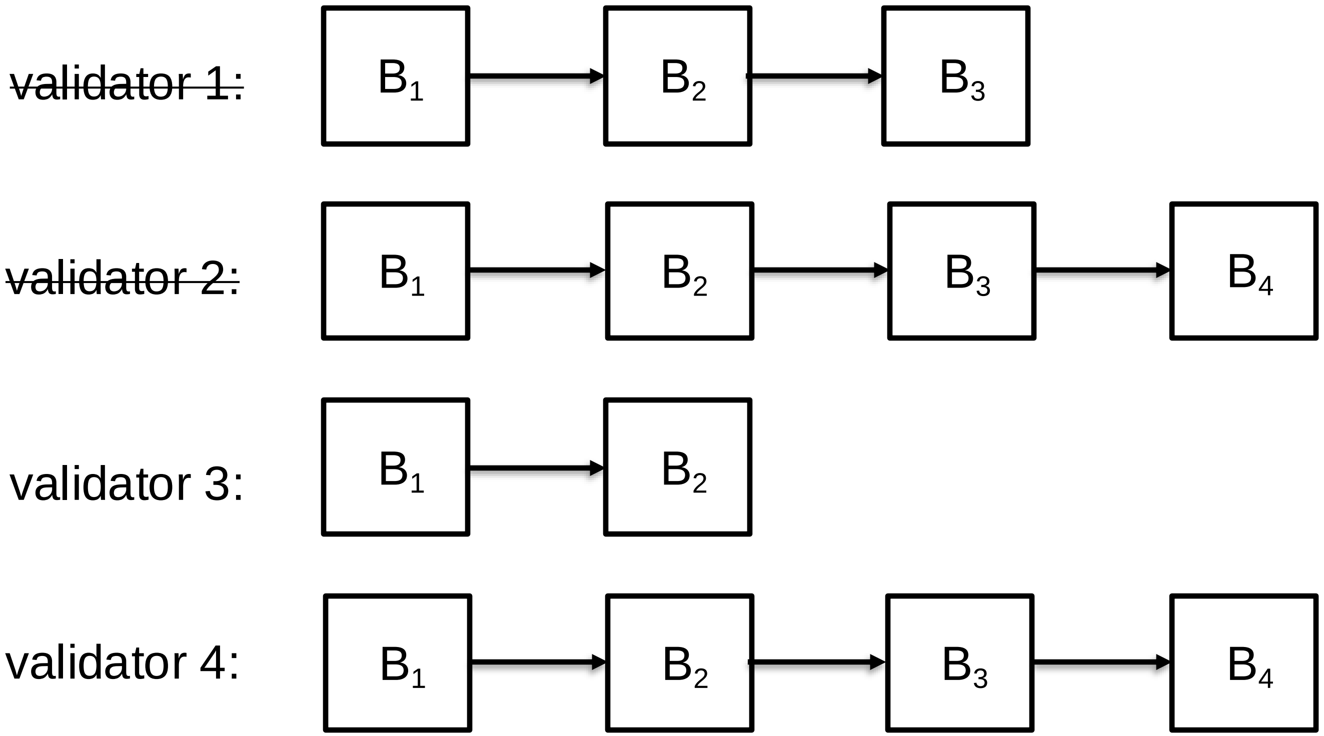
(validator 4 informs next leader about its current chain)

Protocol B: An Example Execution



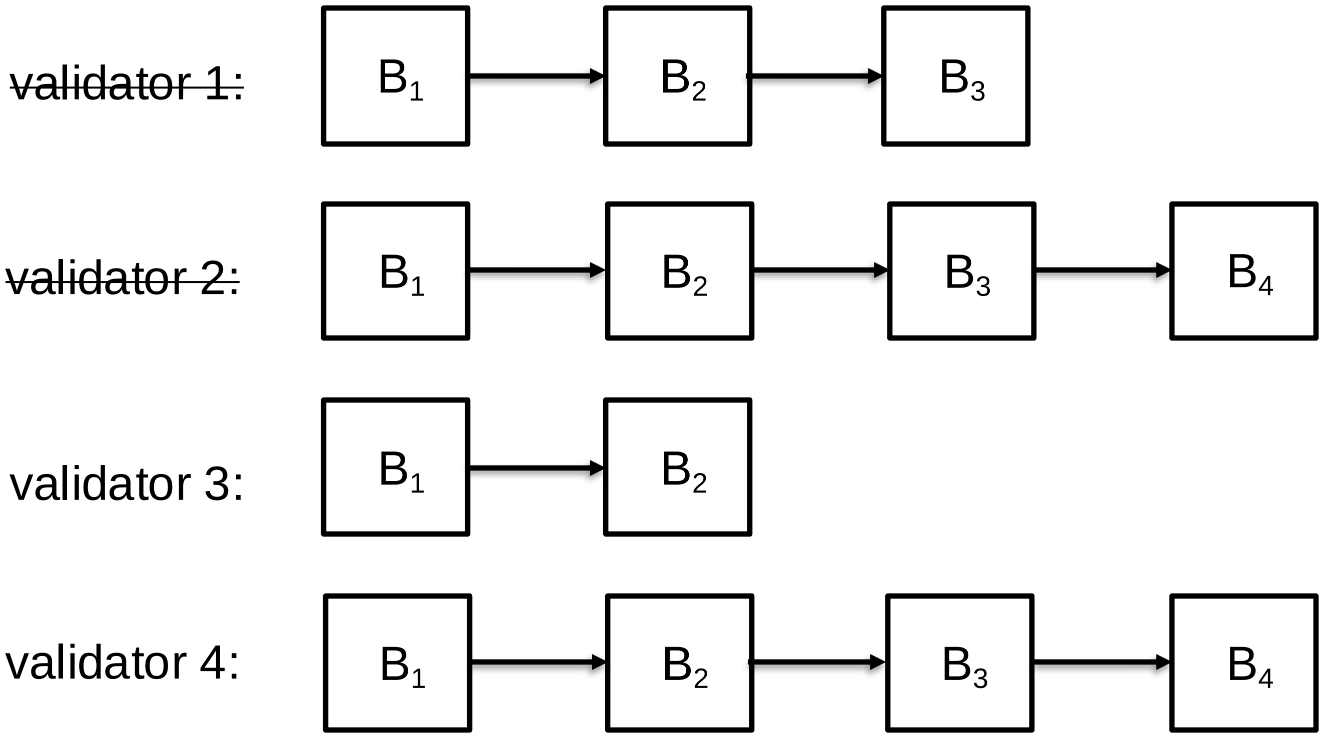
(validator 2 is next leader, prepares its proposal)

Protocol B: An Example Execution



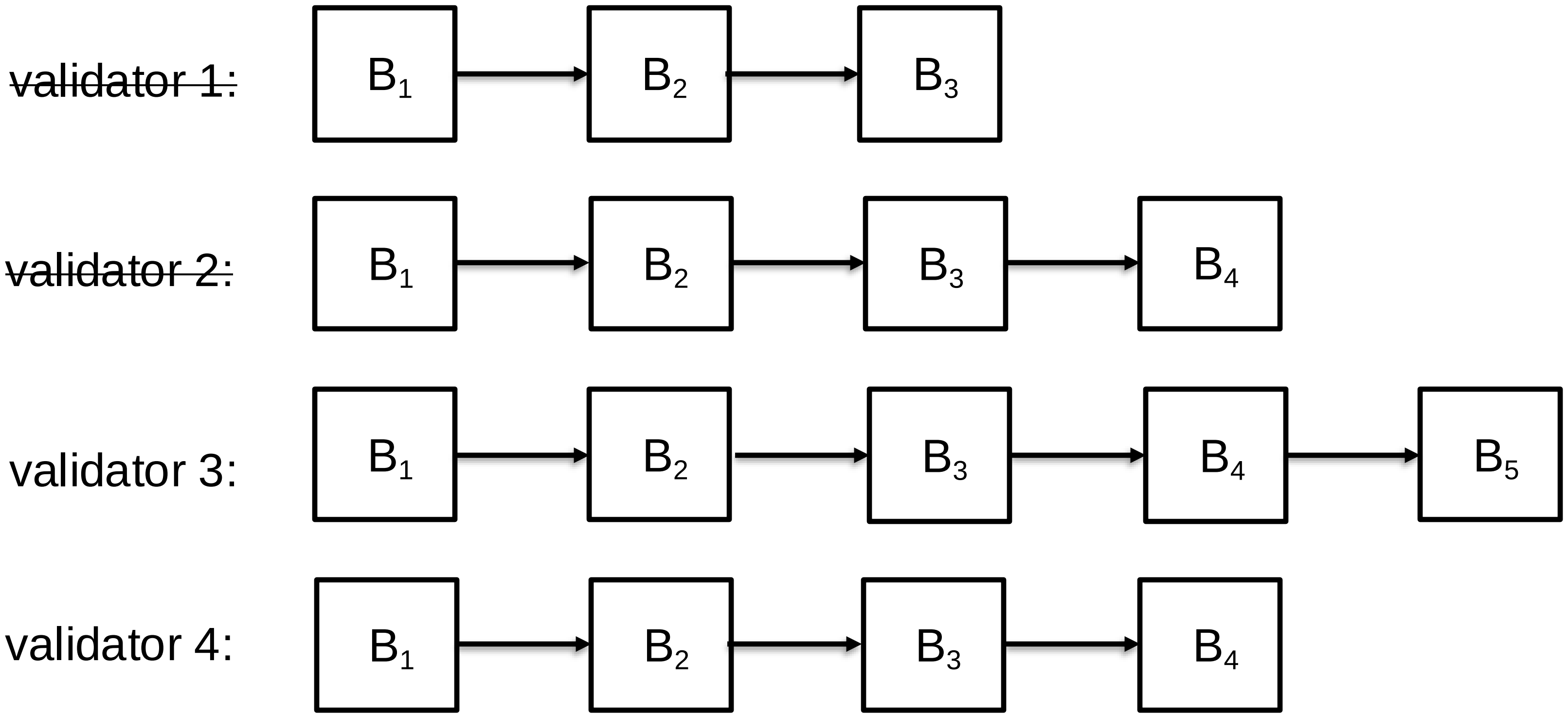
(validator 2 crashes after sending its proposal only to validator 4)

Protocol B: An Example Execution



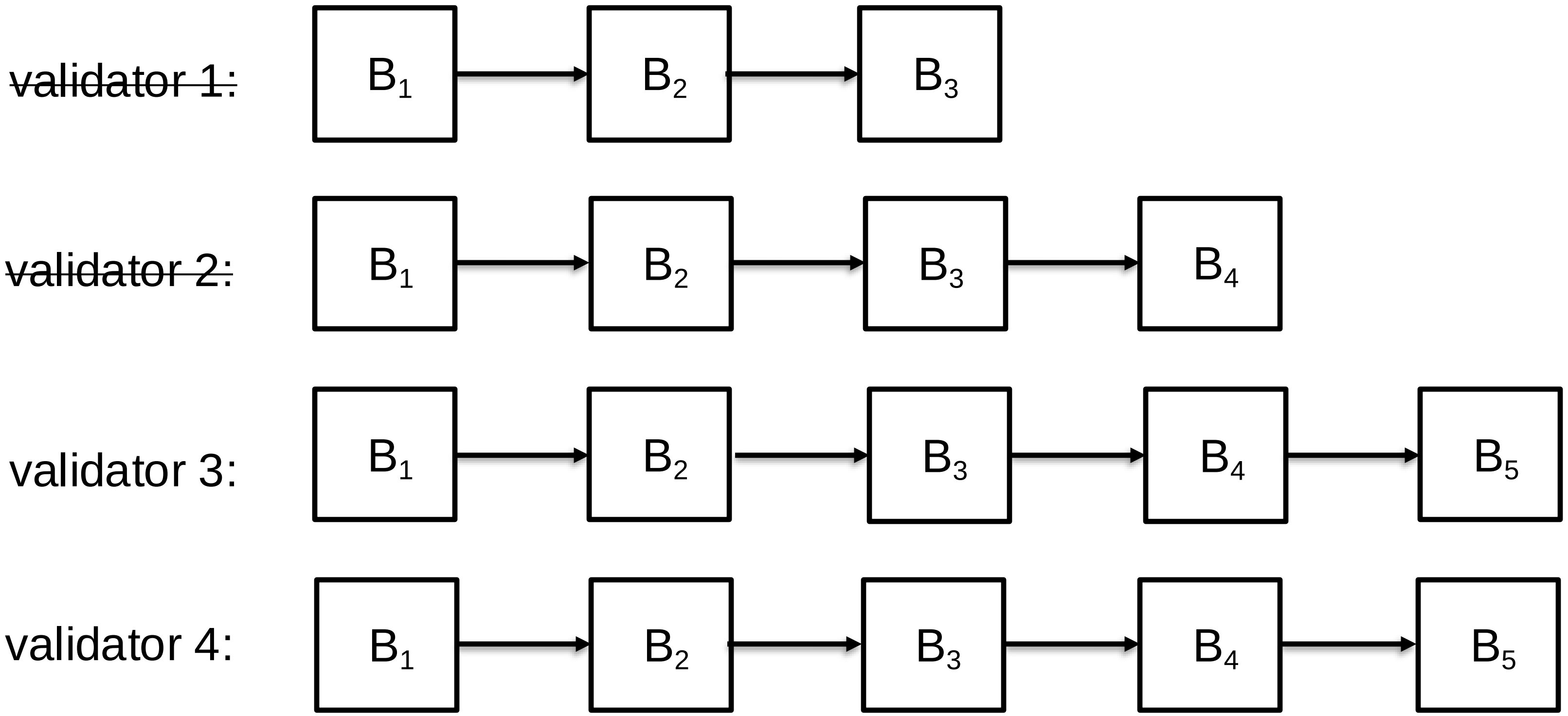
(validator 4 informs next leader about its current chain)

Protocol B: An Example Execution



(validator 3 is next leader, prepares its proposal)

Protocol B: An Example Execution



(if leader doesn't crash, all uncrashed validators adopt its proposal)

Protocol B: Proof of Consistency

Comments:

Protocol B: Proof of Consistency

Comments:

- not a theory class, not trying to train you to do your own proofs
  - though I *am* trying to train you to recognize broken protocols

Protocol B: Proof of Consistency

Comments:

- not a theory class, not trying to train you to do your own proofs
  - though I *am* trying to train you to recognize broken protocols
- but consensus protocol design driven by correctness proofs
  - will help you understand why famous consensus protocols like Paxos/Raft or Tendermint work the way they do

Protocol B: Proof of Consistency

Comments:

- not a theory class, not trying to train you to do your own proofs
  - though I *am* trying to train you to recognize broken protocols
- but consensus protocol design driven by correctness proofs
  - will help you understand why famous consensus protocols like Paxos/Raft or Tendermint work the way they do
- a protocol without a proof is probably buggy
  - embarrassing number of bugs in early drafts of these lectures!

Protocol B: Proof of Consistency

Comments:

- not a theory class, not trying to train you to do your own proofs
  - though I *am* trying to train you to recognize broken protocols
- but consensus protocol design driven by correctness proofs
  - will help you understand why famous consensus protocols like Paxos/Raft or Tendermint work the way they do
- a protocol without a proof is probably buggy
  - embarrassing number of bugs in early drafts of these lectures!
- and bugs in a global consensus protocol likely to be exposed
  - run for multiple years under widely varying workloads/conditions

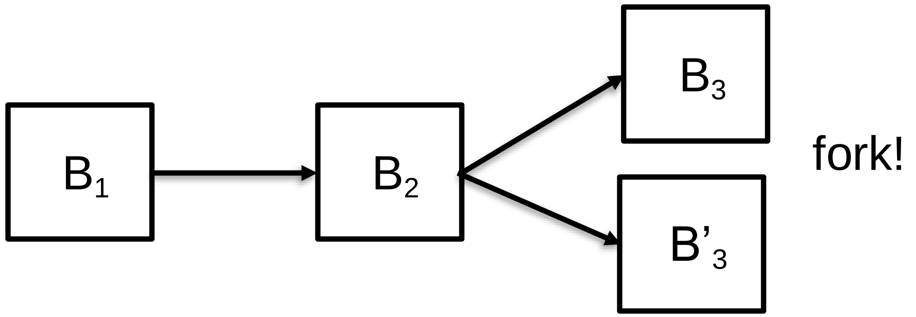
Protocol B: Proof of Consistency

Tricky point: could be multiple versions of e.g. block #3 over lifetime of protocol (with earlier version forgotten with crashes).

Protocol B: Proof of Consistency

Tricky point: could be multiple versions of e.g. block #3 over lifetime of protocol (with earlier version forgotten with crashes).

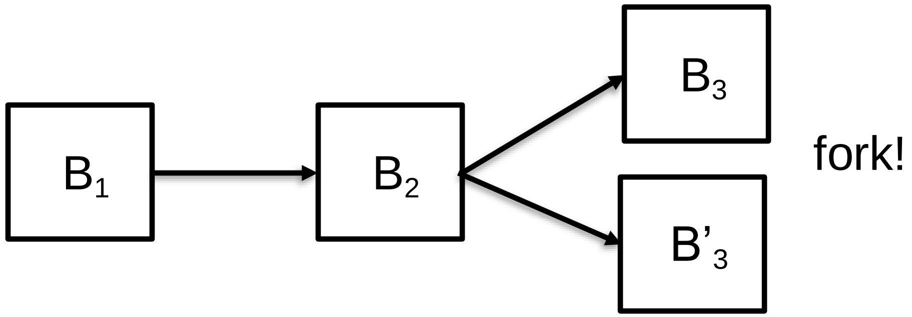
Recall: validators' local chains are consistent ⇔ all prefixes of a common chain (i.e., no forks).



Protocol B: Proof of Consistency

Tricky point: could be multiple versions of e.g. block #3 over lifetime of protocol (with earlier version forgotten with crashes).

Recall: validators' local chains are consistent ⬄ all prefixes of a common chain (i.e., no forks).



Claim: at each time step, the chains of the not-yet-crashed validators are consistent.

Protocol B: Proof of Consistency

Claim: at each time step, the chains of the not-yet-crashed validators are consistent.

Protocol B: Proof of Consistency

Claim: at each time step, the chains of the not-yet-crashed validators are consistent.

- proceed by induction on the number of timesteps (true initially)

Protocol B: Proof of Consistency

Claim: at each time step, the chains of the not-yet-crashed validators are consistent.

- proceed by induction on the number of timesteps (true initially)
- in view v, by the inductive hypothesis, all the C_i's received by the leader are consistent (i.e., prefixes of a common chain)
  - these were the local chains of all not-yet-crashed validators at time $\Delta \cdot v$
  - leader receives all such C_i's by time $\Delta \cdot v + \Delta$ (due to synchrony)

Protocol B: Proof of Consistency

Claim: at each time step, the chains of the not-yet-crashed validators are consistent.

- proceed by induction on the number of timesteps (true initially)
- in view v, by the inductive hypothesis, all the C_i's received by the leader are consistent (i.e., prefixes of a common chain)
  - these were the local chains of all not-yet-crashed validators at time $\Delta \cdot v$
  - leader receives all such C_i's by time $\Delta \cdot v + \Delta$ (due to synchrony)
- C will extend all these C_i's (will be the longest of them)

Protocol B: Proof of Consistency

Claim: at each time step, the chains of the not-yet-crashed validators are consistent.

- proceed by induction on the number of timesteps (true initially)
- in view v, by the inductive hypothesis, all the C_i's received by the leader are consistent (i.e., prefixes of a common chain)
  - these were the local chains of all not-yet-crashed validators at time $\Delta \cdot v$
  - leader receives all such C_i's by time $\Delta \cdot v + \Delta$ (due to synchrony)
- C will extend all these C_i's (will be the longest of them)
- C^* extends all these C_i's

Protocol B: Proof of Consistency

Claim: at each time step, the chains of the not-yet-crashed validators are consistent.

- proceed by induction on the number of timesteps (true initially)
- in view v, by the inductive hypothesis, all the C_i's received by the leader are consistent (i.e., prefixes of a common chain)
- C will extend all these C_i's (will be the longest of them)
- C^* extends all these C_i's
- no matter which validators update their C_i's in this view, will stay consistent

Protocol B: Proof of Liveness

Suppose tx z known to some non-faulty validator i at time step t.

Protocol B: Proof of Liveness

Suppose tx z known to some non-faulty validator i at time step t.

- let v be the next view for which i is the leader (must exist)

Protocol B: Proof of Liveness

Suppose tx z known to some non-faulty validator i at time step t.

- let v be the next view for which i is the leader (must exist)
- i’s proposal C* := (C,B) in view v will include the tx z
  – if not already in C, will put it in the new block B

Protocol B: Proof of Liveness

Suppose tx z known to some non-faulty validator i at time step t.

- let v be the next view for which i is the leader (must exist)
- i’s proposal C* := (C,B) in view v will include the tx z
  - if not already in C, will put it in the new block B
- since i is non-faulty, sends proposal C* to all validators

Protocol B: Proof of Liveness

Suppose tx z known to some non-faulty validator i at time step t.

- let v be the next view for which i is the leader (must exist)
- i's proposal C* := (C,B) in view v will include the tx z
  - if not already in C, will put it in the new block B
- since i is non-faulty, sends proposal C* to all validators
- C* adopted by all (uncrashed) validators

Takeaways/Design Patterns

Takeaways/Design Patterns

1. views = repeated attempts to finalize new transactions.

Takeaways/Design Patterns

1. views = repeated attempts to finalize new transactions.
2. leaders = coordinate the transactions proposed in each view.
   - chosen e.g. round-robin (variation: chosen randomly)

Takeaways/Design Patterns

1. views = repeated attempts to finalize new transactions.
2. leaders = coordinate the transactions proposed in each view.
   - chosen e.g. round-robin (variation: chosen randomly)
3. view may end with non-faulty validators in different states.
   - leader may need to “clean up the mess” left by previous view

Takeaways/Design Patterns

1. views = repeated attempts to finalize new transactions.
2. leaders = coordinate the transactions proposed in each view.
   - chosen e.g. round-robin (variation: chosen randomly)
3. view may end with non-faulty validators in different states.
   - leader may need to “clean up the mess” left by previous view
4. leader should be as up-to-date as all non-faulty validators.
   - otherwise, leader’s out-of-date proposal might conflict with the local chains of more up-to-date non-faulty validators
   - reason for the “catch-up” messages in first half of view in Protocol B

Takeaways/Design Patterns

1. views = repeated attempts to finalize new transactions.
2. leaders = coordinate the transactions proposed in each view.
   – chosen e.g. round-robin (variation: chosen randomly)
3. view may end with non-faulty validators in different states.
   – leader may need to “clean up the mess” left by previous view
4. leader should be as up-to-date as all non-faulty validators.
   – otherwise, leader’s out-of-date proposal might conflict with the local chains of more up-to-date non-faulty validators
   – reason for the “catch-up” messages in first half of view in Protocol B
5. distributed computing is hard! [no proof ➔ probably buggy!]

A Road Map to Practical SMR Protocols

crash faults +
synchronous network

crash faults +
asynchronous network

Byzantine faults +
asynchronous network

easier → harder

The Challenges of Asynchrony

Question: Is Protocol B still consistent w/unbounded msg delays?

The Challenges of Asynchrony

Question: Is Protocol B still consistent w/unbounded msg delays?

Answer: No!

The Challenges of Asynchrony

Question: Is Protocol B still consistent w/unbounded msg delays?

Answer: No! Reason: leader may not hear about all C_i's of non-faulty validators by the time it makes a proposal.

- if $C_i = B_1 \rightarrow B_2 \rightarrow B_3$ but leader only hears about $B_1 \rightarrow B_2$, might propose $B_1 \rightarrow B_2 \rightarrow B'_3$, potentially leading to consistency violation

The Challenges of Asynchrony

Question: Is Protocol B still consistent w/unbounded msg delays?

Answer: No! Reason: leader may not hear about all C_i's of non-faulty validators by the time it makes a proposal.

- if $C_i = B_1 \rightarrow B_2 \rightarrow B_3$ but leader only hears about $B_1 \rightarrow B_2$, might propose $B_1 \rightarrow B_2 \rightarrow B'_3$, potentially leading to consistency violation

Key challenge: how to ensure leader knows about the C_i's of all non-faulty validators by the time it makes a proposal (despite unpredictable message delays)?

The Challenges of Asynchrony

Question: Is Protocol B still consistent w/unbounded msg delays?

Answer: No! Reason: leader may not hear about all C_i's of non-faulty validators by the time it makes a proposal.

- if $C_i = B_1 \rightarrow B_2 \rightarrow B_3$ but leader only hears about $B_1 \rightarrow B_2$, might propose $B_1 \rightarrow B_2 \rightarrow B'_3$, potentially leading to consistency violation

Key challenge: how to ensure leader knows about the C_i's of all non-faulty validators by the time it makes a proposal (despite unpredictable message delays)?

- – will resolve next lecture (add friction to proposing and to finalizing new transactions, also assume strict majority of non-faulty validators)

Modeling Asynchrony

Question: how to model an “unreliable network”?

Modeling Asynchrony

Question: how to model an "unreliable network"?

Bad answer: Make Δ really big.

- avoids the issue, leads to completely impractical protocols

Modeling Asynchrony

Question: how to model an "unreliable network"?

Bad answer: Make Δ really big.

- avoids the issue, leads to completely impractical protocols

Ambitious answer: no assumptions on message delays at all (in effect, controlled by a worst-case adversary).

- subject to every message eventually getting delivered
- called the *asynchronous model*

Modeling Asynchrony

Question: how to model an "unreliable network"?

Bad answer: Make Δ really big.

– avoids the issue, leads to completely impractical protocols

Ambitious answer: no assumptions on message delays at all.

– subject to every message eventually getting delivered
– called the *asynchronous model*

FLP Theorem ('85): even with the threat of a single crash fault, can't solve SMR in the asynchronous model.

– see Friday bonus lecture for discussion and proof

Getting Around the FLP Theorem

Perspective: impossibility results like the FLP Theorem give guidance on how to compromise to make progress.

Getting Around the FLP Theorem

Perspective: impossibility results like the FLP Theorem give guidance on how to compromise to make progress.

Possible compromises:

1. Pull back from asynchrony to “partial synchrony” (next lecture).
   – “sweet spot” hybrid of the synchronous, asynchronous models

Getting Around the FLP Theorem

Perspective: impossibility results like the FLP Theorem give guidance on how to compromise to make progress.

Possible compromises:

1. Pull back from asynchrony to “partial synchrony” (next lecture).
   - “sweet spot” hybrid of the synchronous, asynchronous models
2. Solve a problem easier than SMR (e.g., with relaxed consistency requirements).
   - agreement on total ordering of txs is overkill in some applications

Getting Around the FLP Theorem

Perspective: impossibility results like the FLP Theorem give guidance on how to compromise to make progress.

Possible compromises:

1. Pull back from asynchrony to "partial synchrony" (next lecture).
   – "sweet spot" hybrid of the synchronous, asynchronous models
2. Solve a problem easier than SMR (e.g., with relaxed consistency requirements).
   – agreement on total ordering of txs is overkill in some applications
3. Use randomized protocols, solve SMR with high probability.
   – rich academic literature on this topic