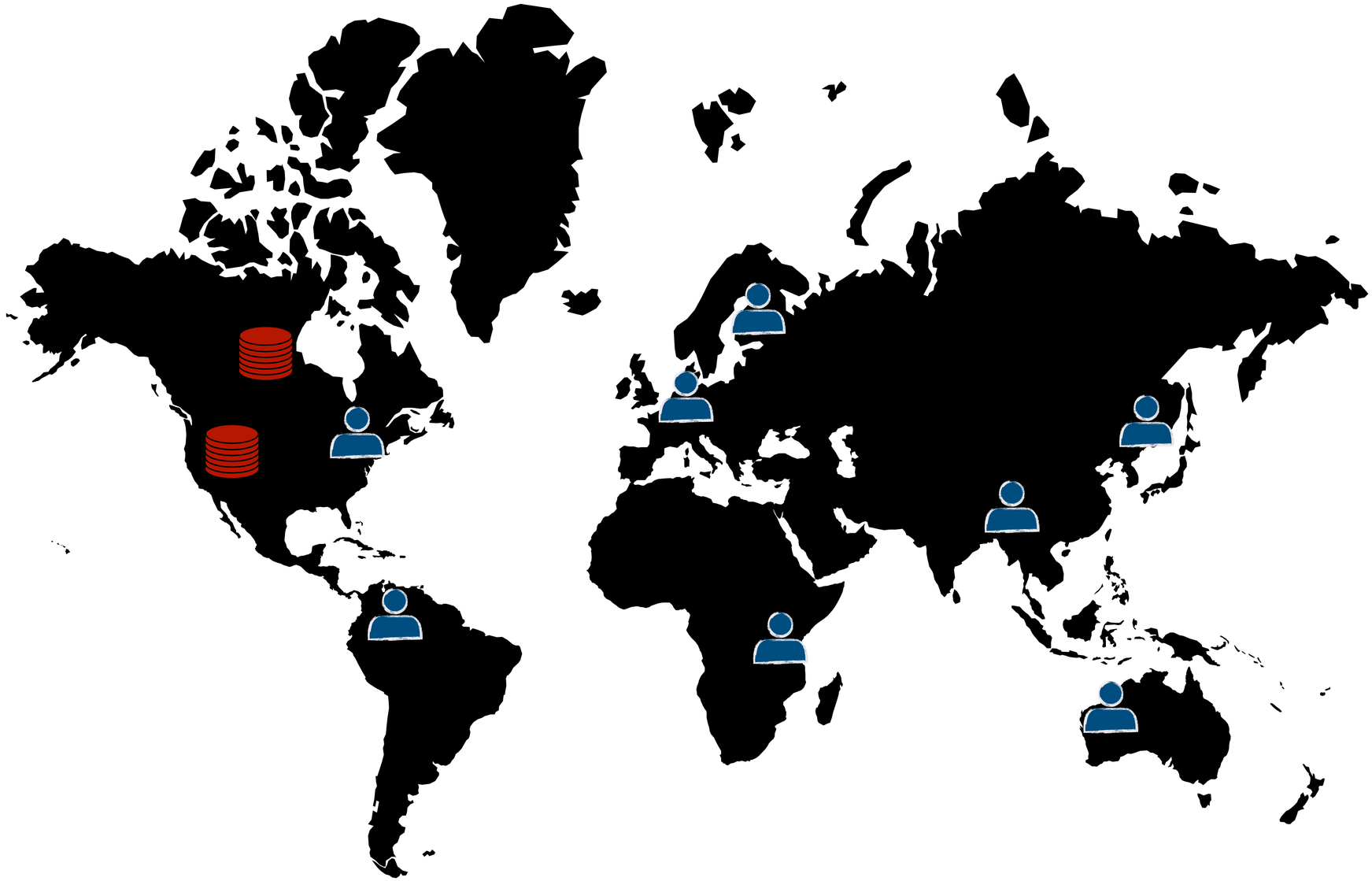
efficient replication via timestamp stability

Vitor Enes, Carlos Baquero, Alexey Gotsman, Pierre Sutra

27 Apr. 2021 @ EuroSys'21

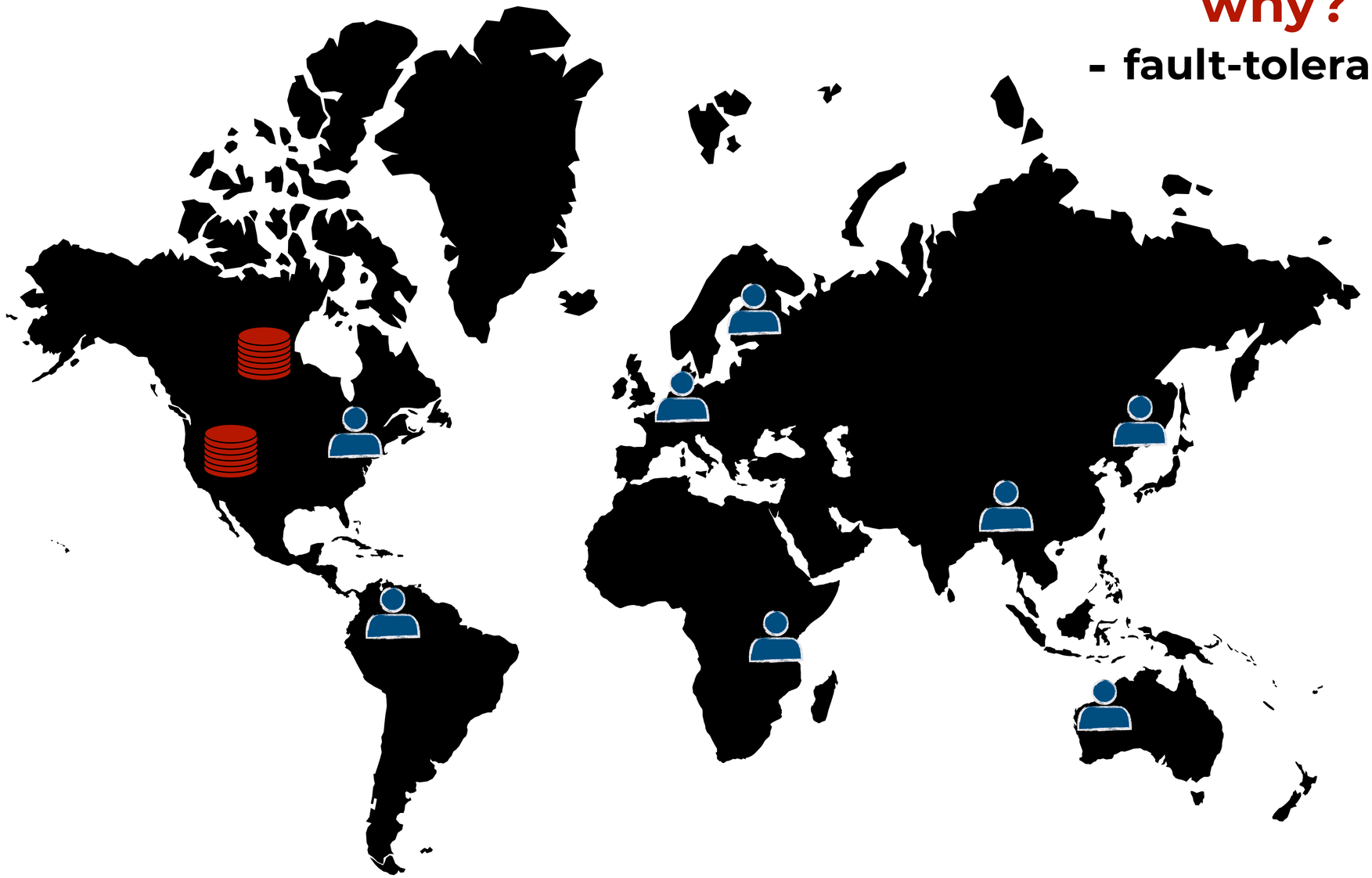
planet-scale replicated systems



planet-scale replicated systems

why?

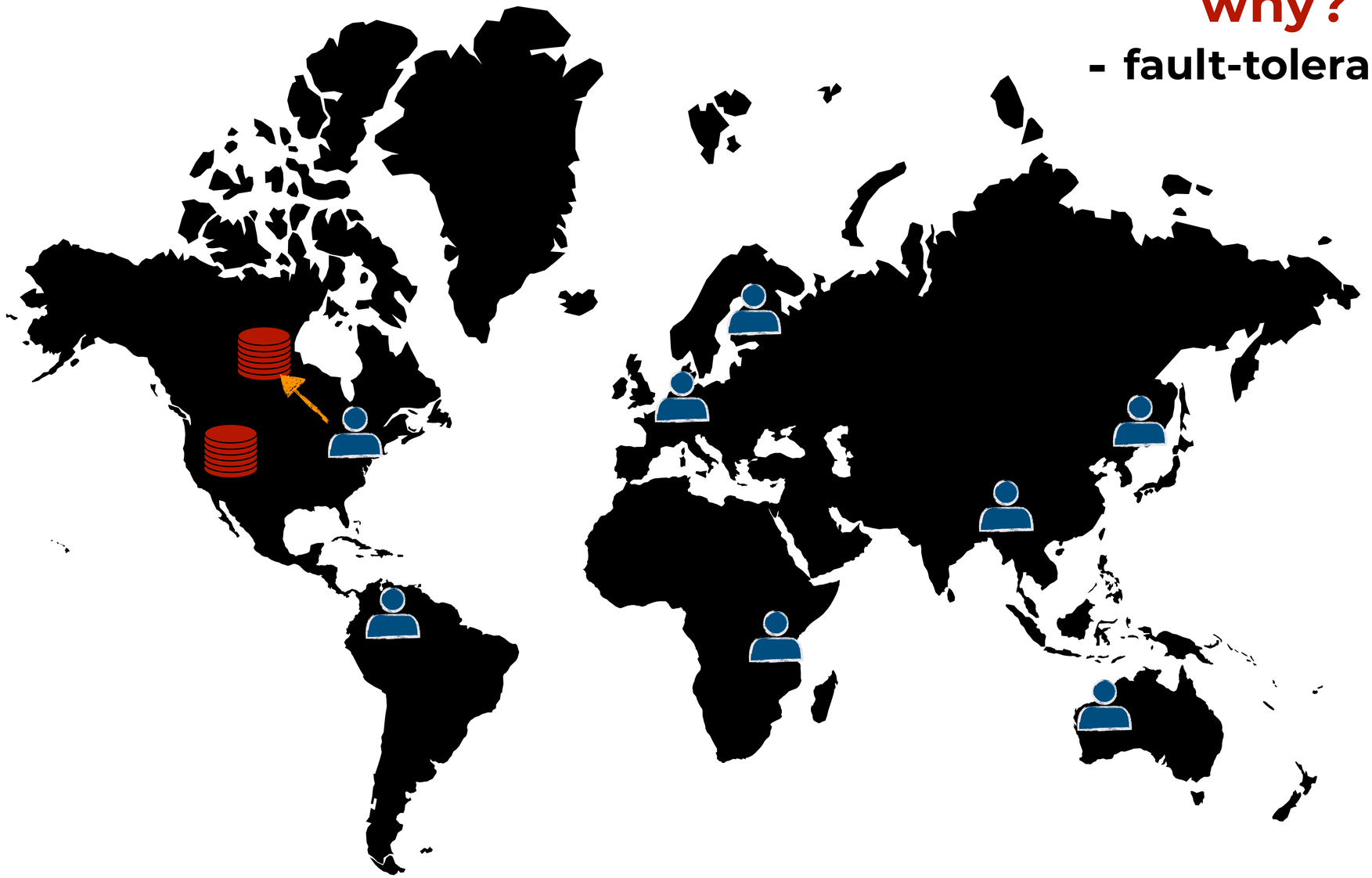
- **fault-tolerance**



planet-scale replicated systems

why?

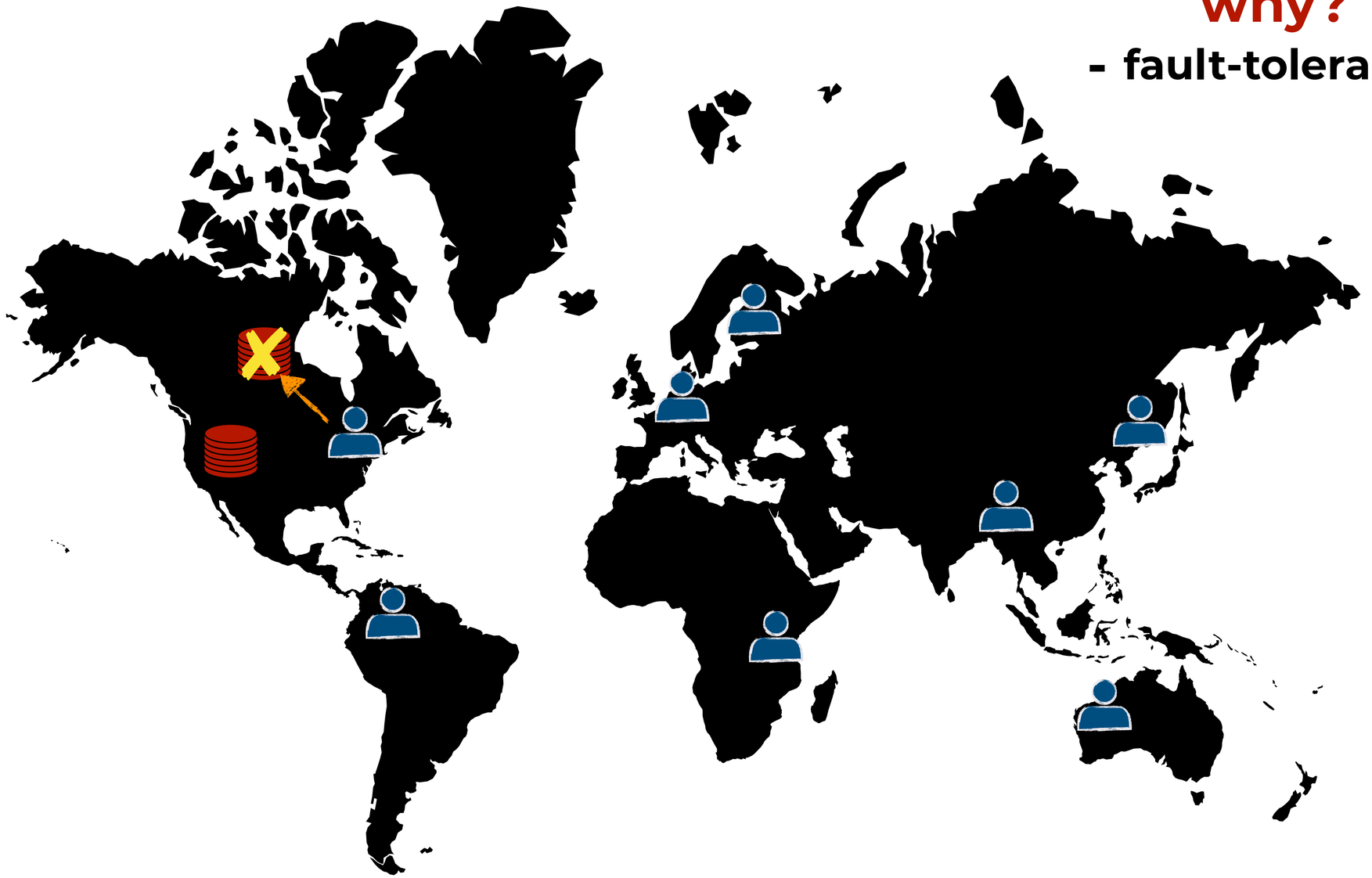
- **fault-tolerance**



planet-scale replicated systems

why?

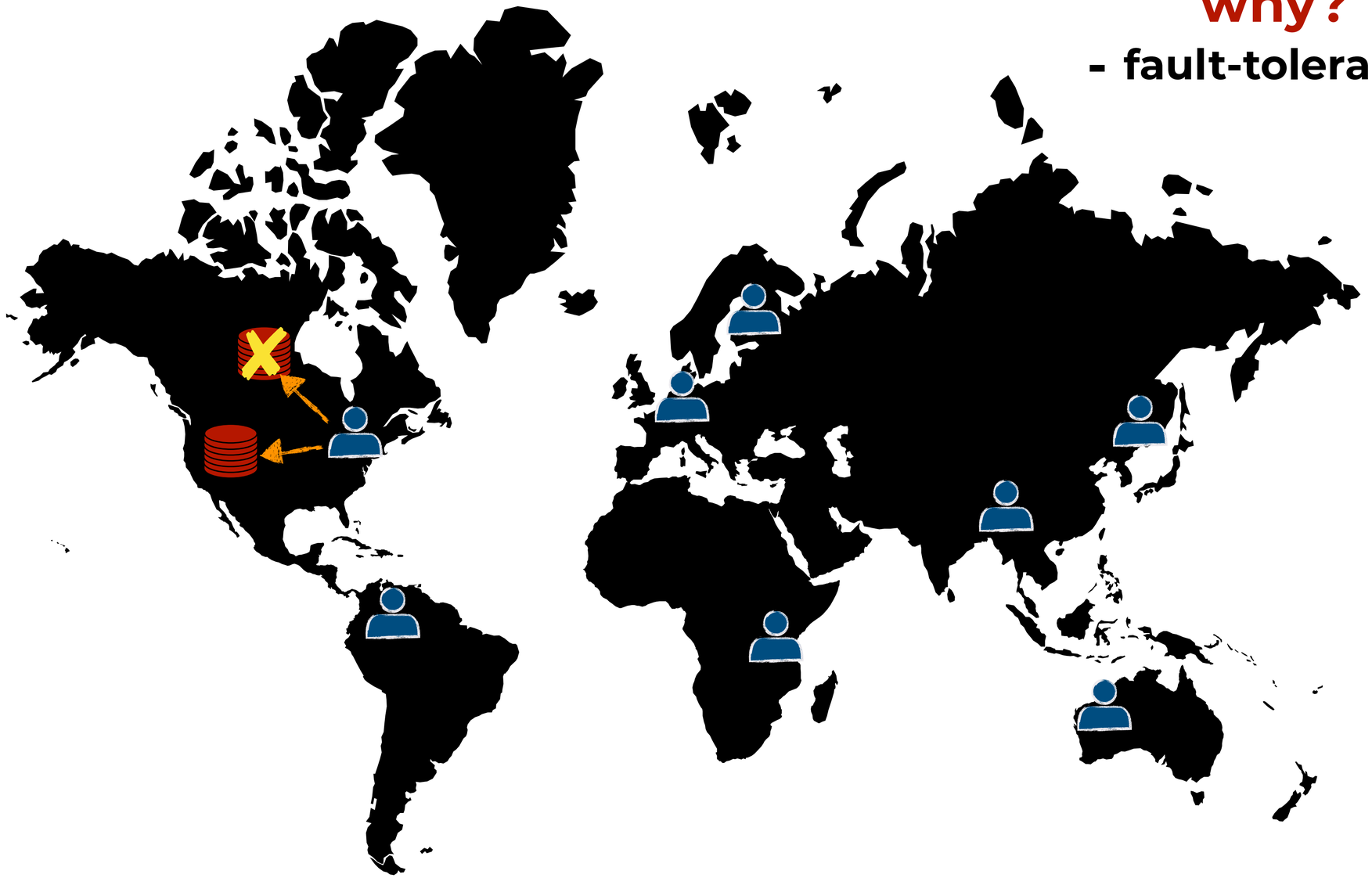
- **fault-tolerance**



planet-scale replicated systems

why?

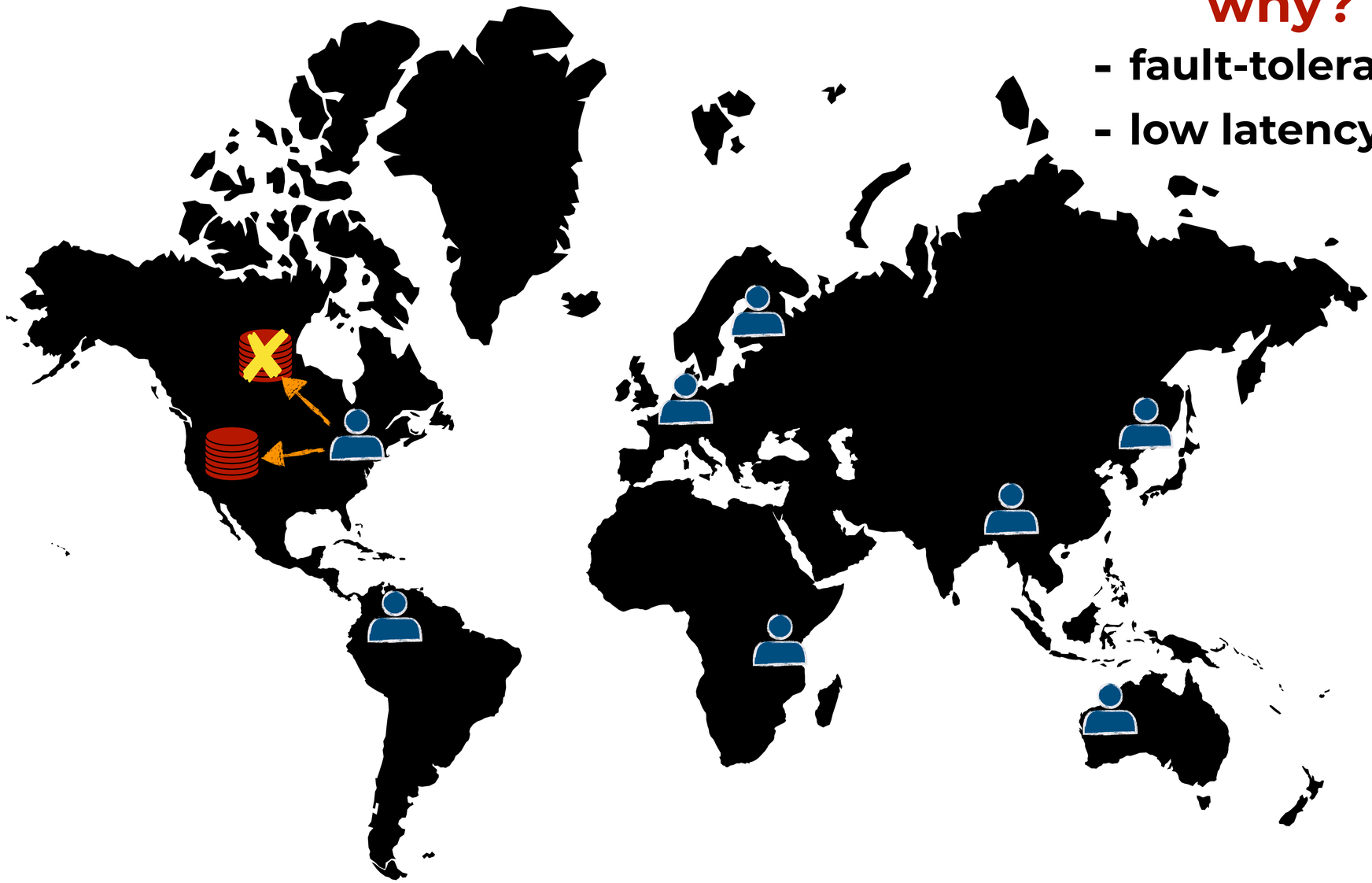
- **fault-tolerance**



planet-scale replicated systems

why?

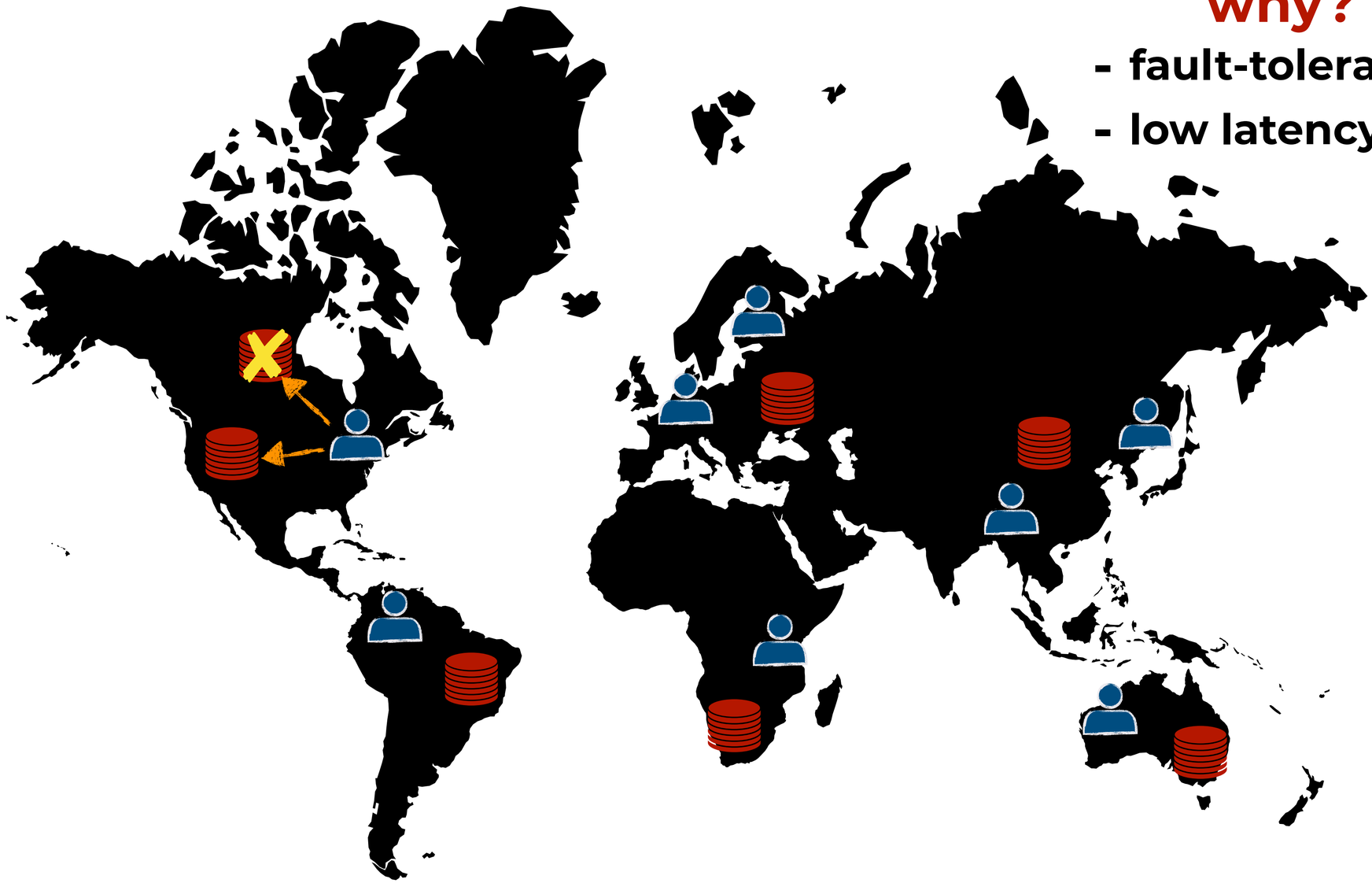
- **fault-tolerance**
- **low latency**



planet-scale replicated systems

why?

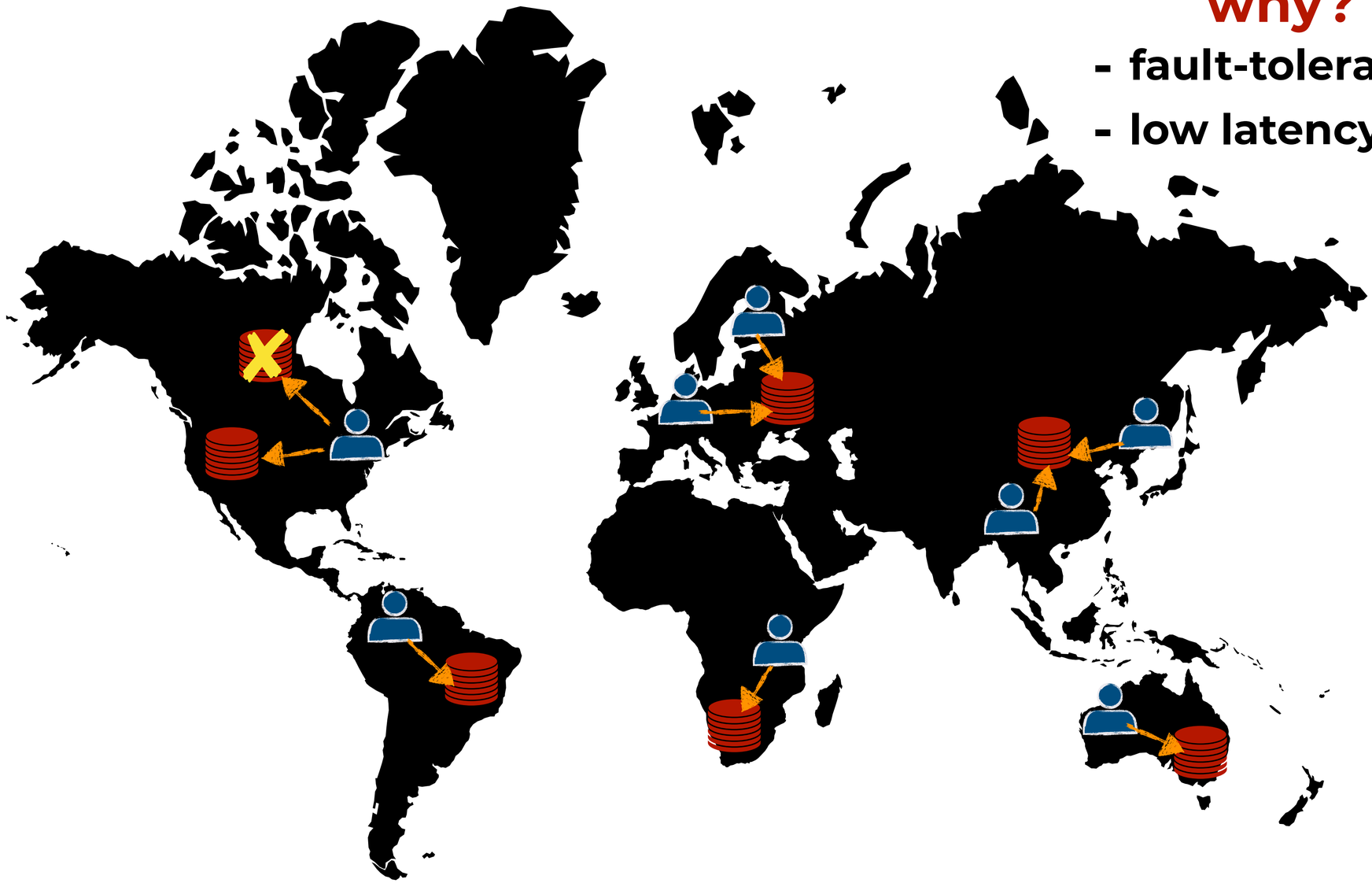
- **fault-tolerance**
- **low latency**



planet-scale replicated systems

why?

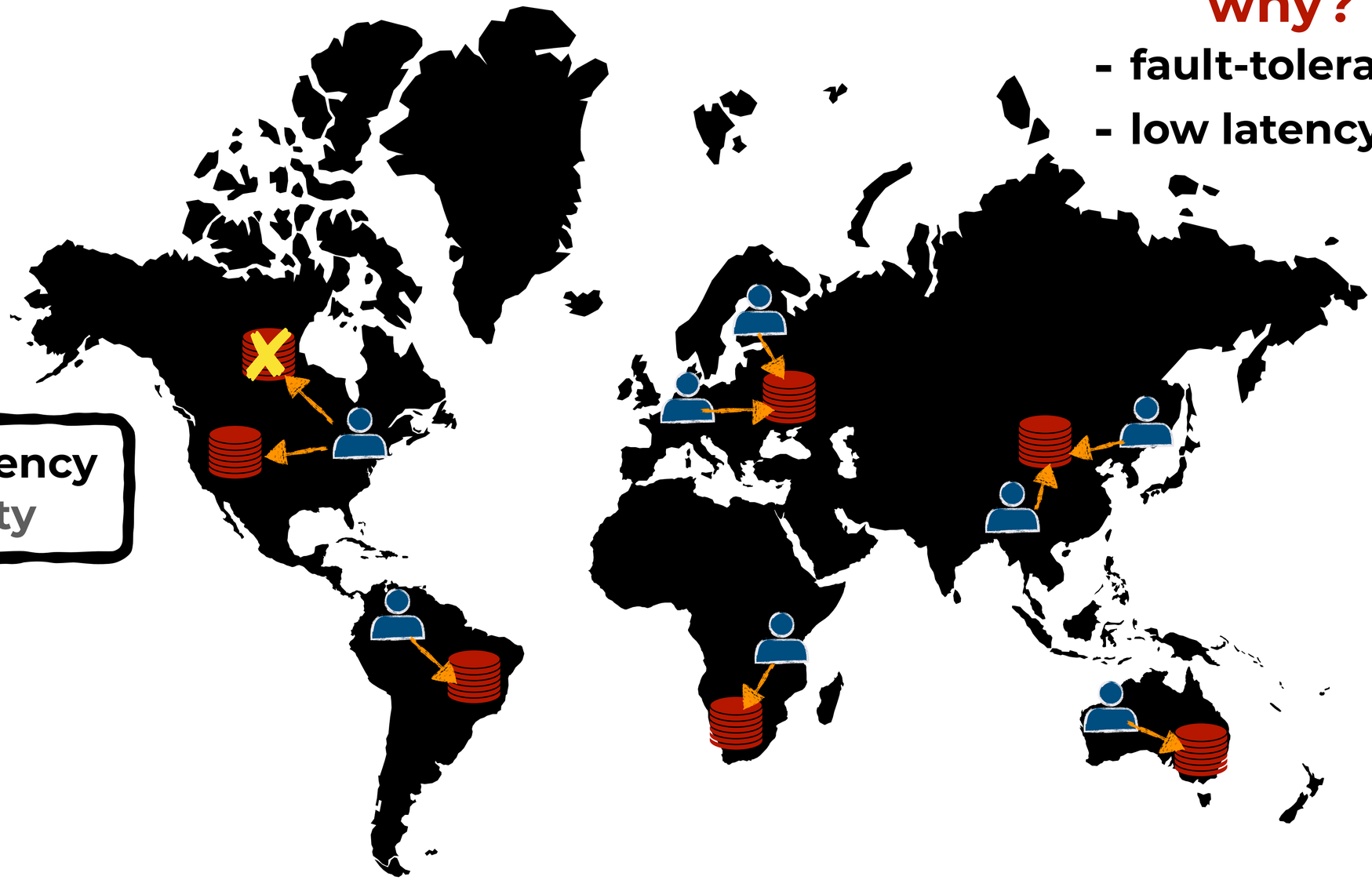
- **fault-tolerance**
- **low latency**



planet-scale replicated systems

why?

- **fault-tolerance**
- **low latency**



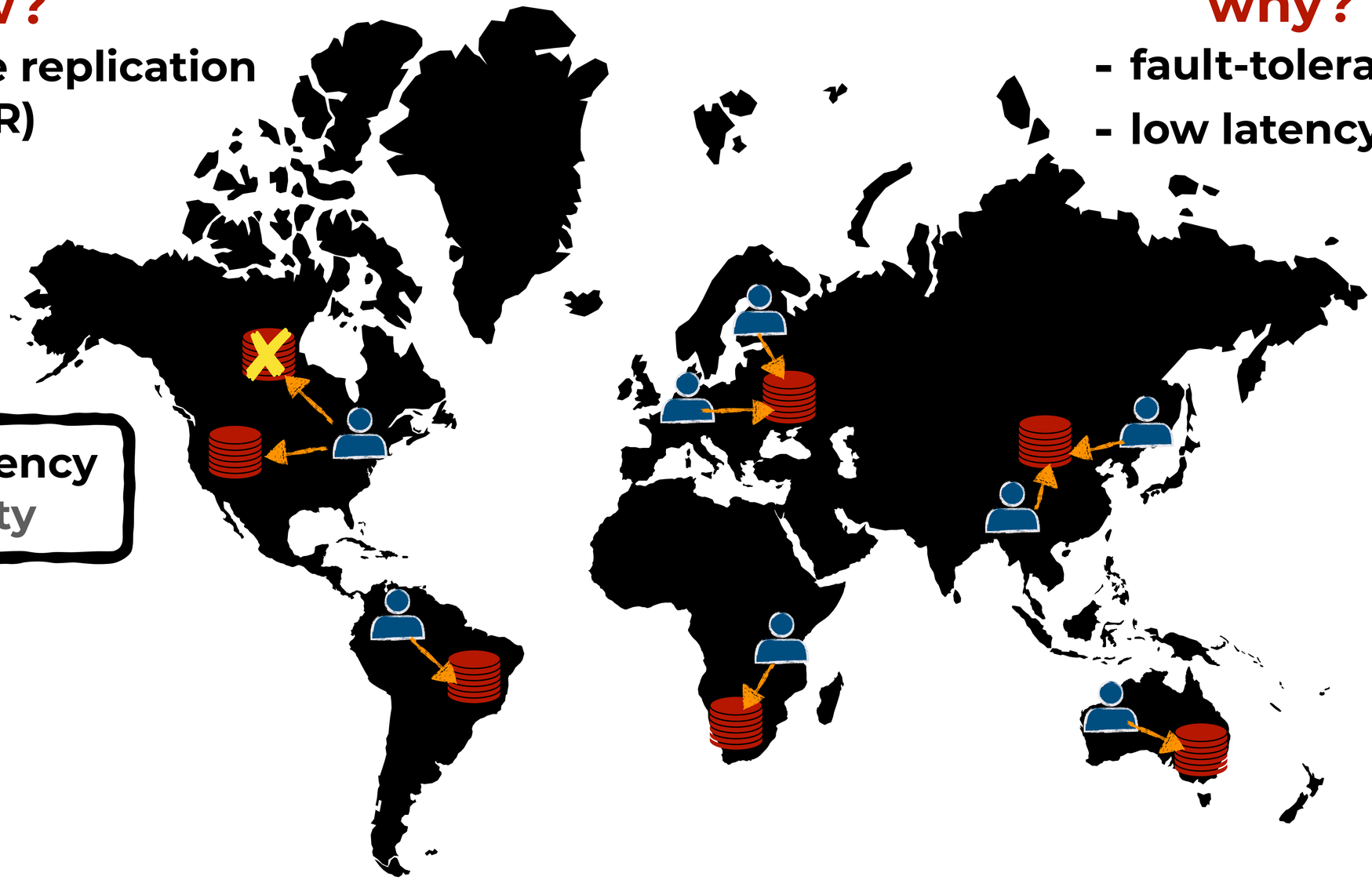
strong consistency
linearizability

planet-scale replicated systems

how?

- state-machine replication (SMR)

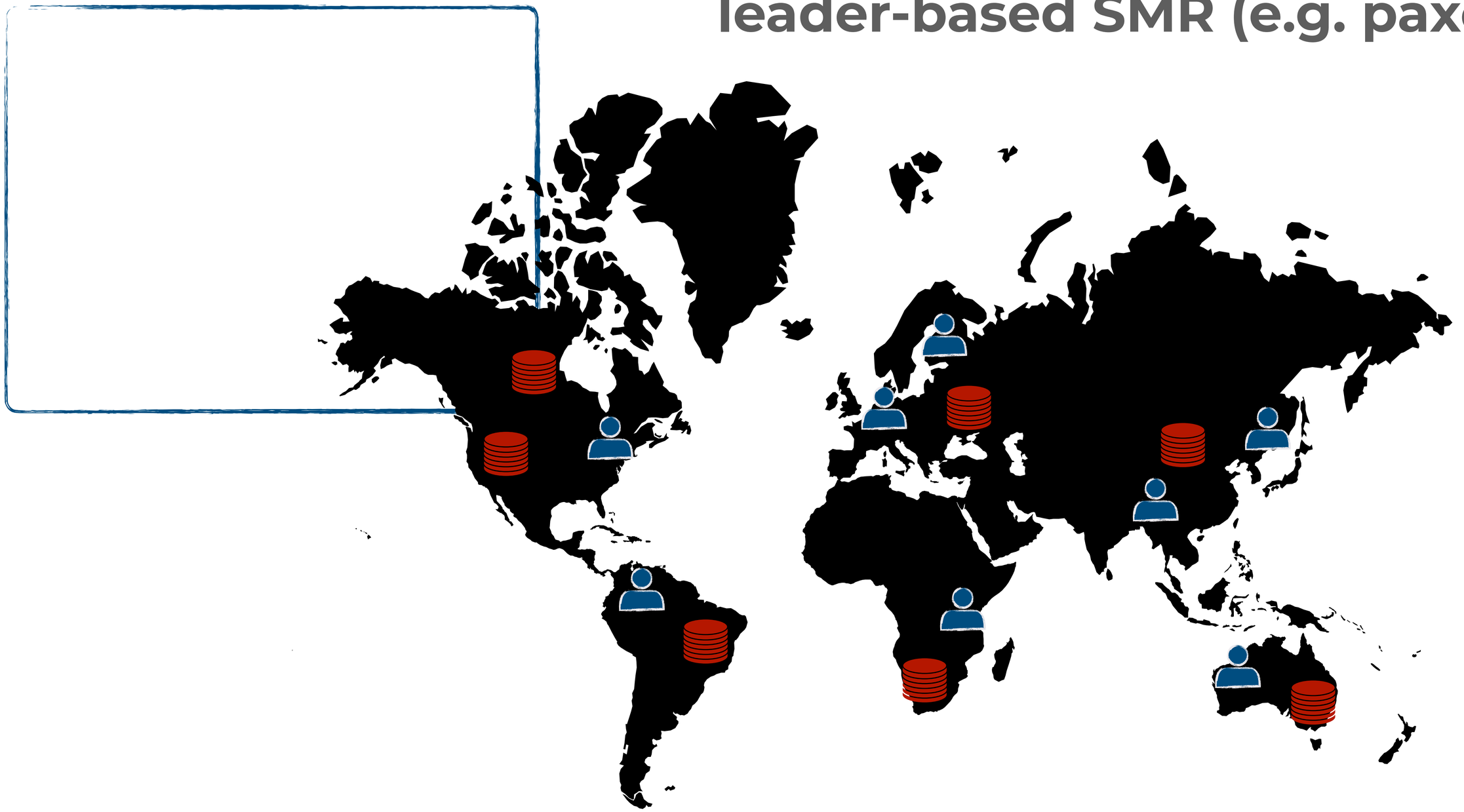
strong consistency
linearizability



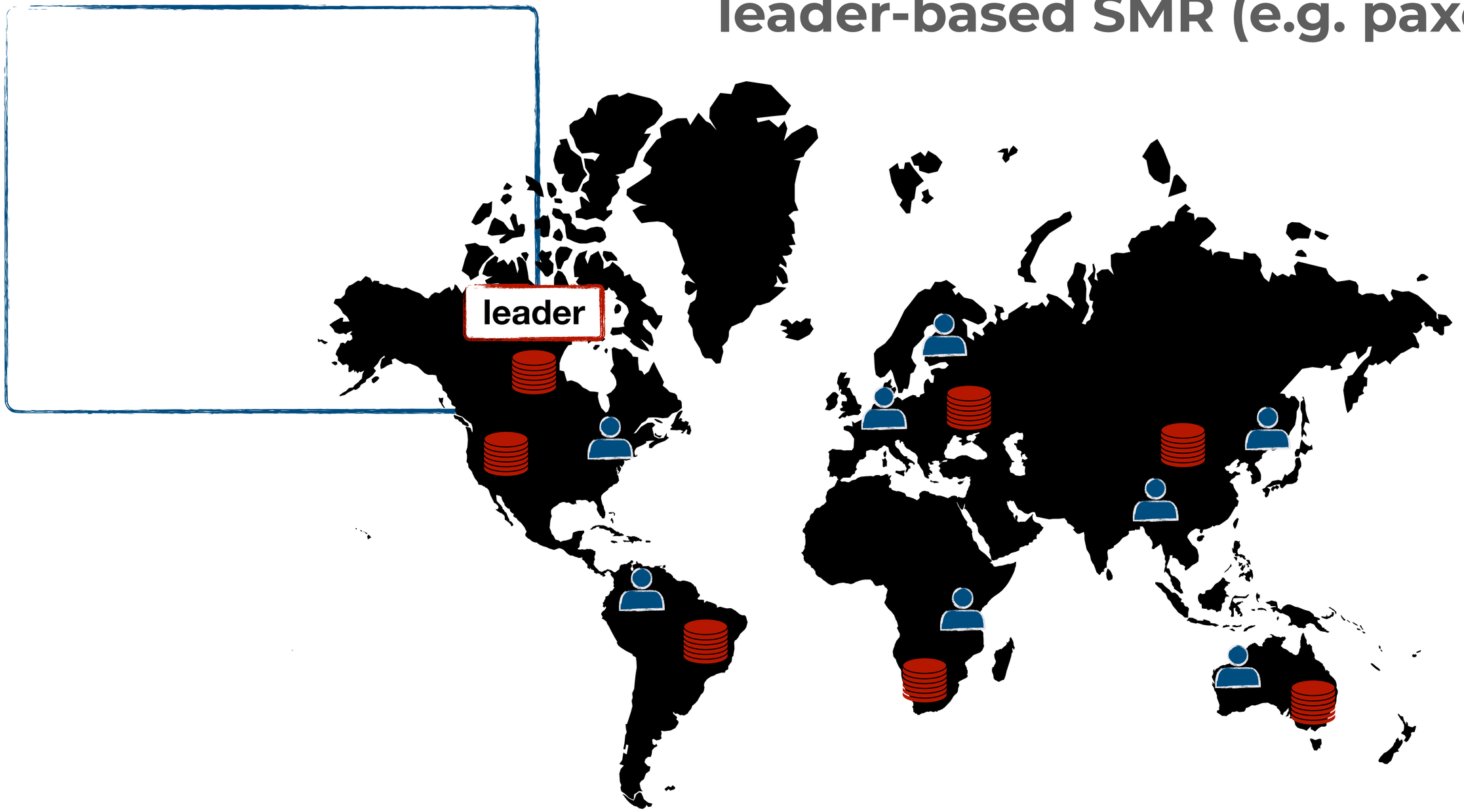
why?

- fault-tolerance
- low latency

leader-based SMR (e.g. paxos)

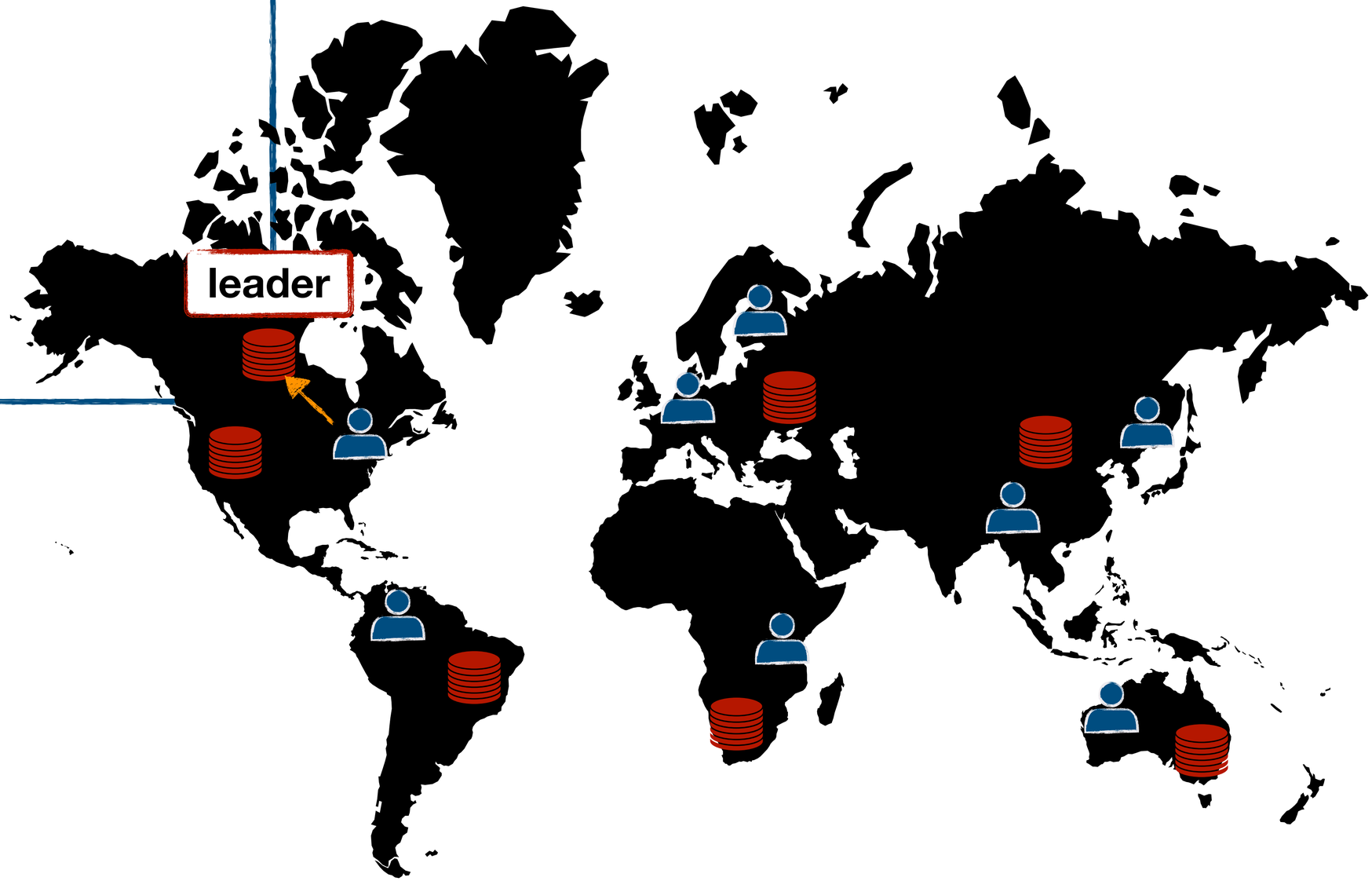


leader-based SMR (e.g. paxos)



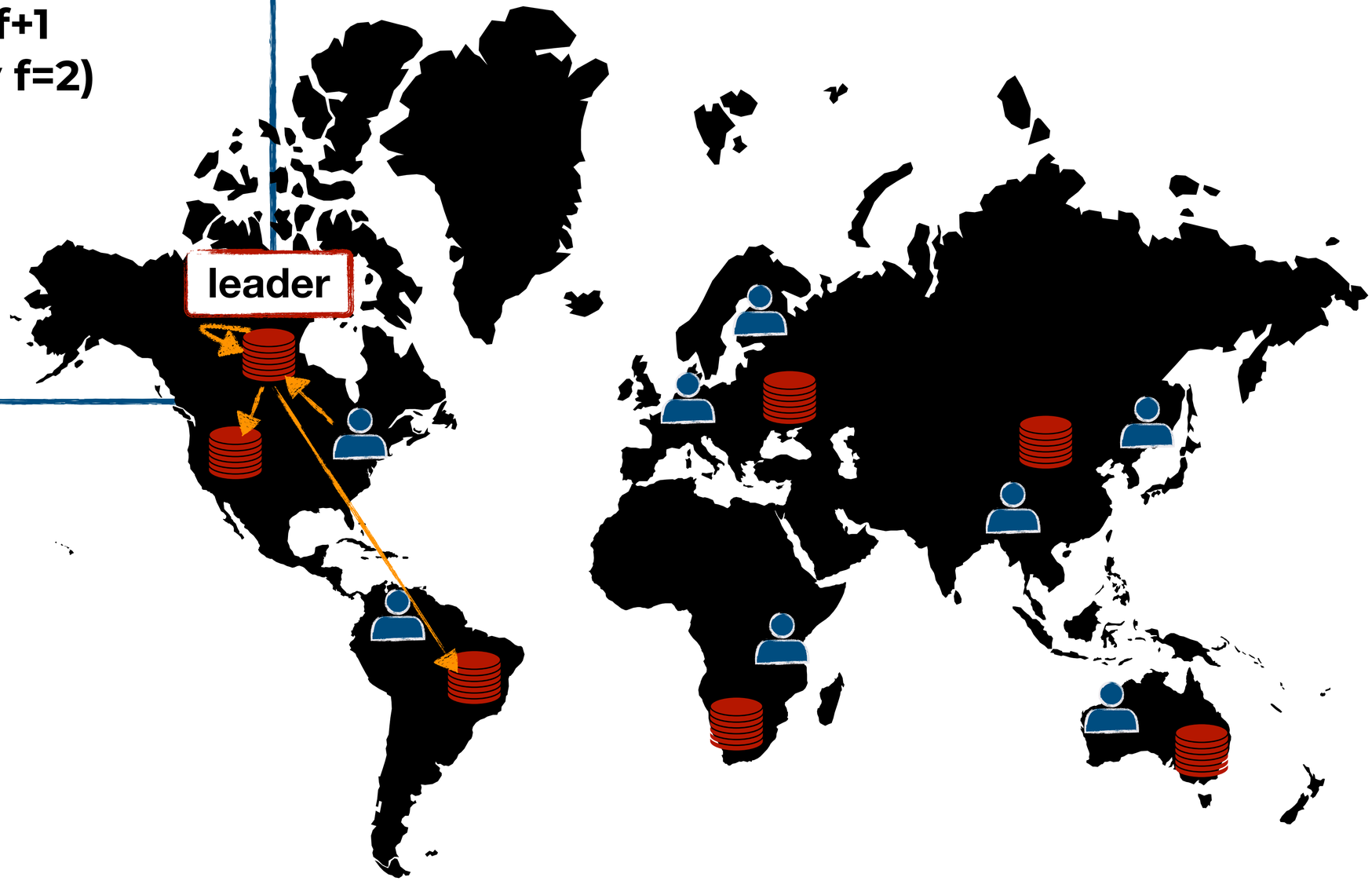
leader-based SMR (e.g. paxos)

- leader receives command



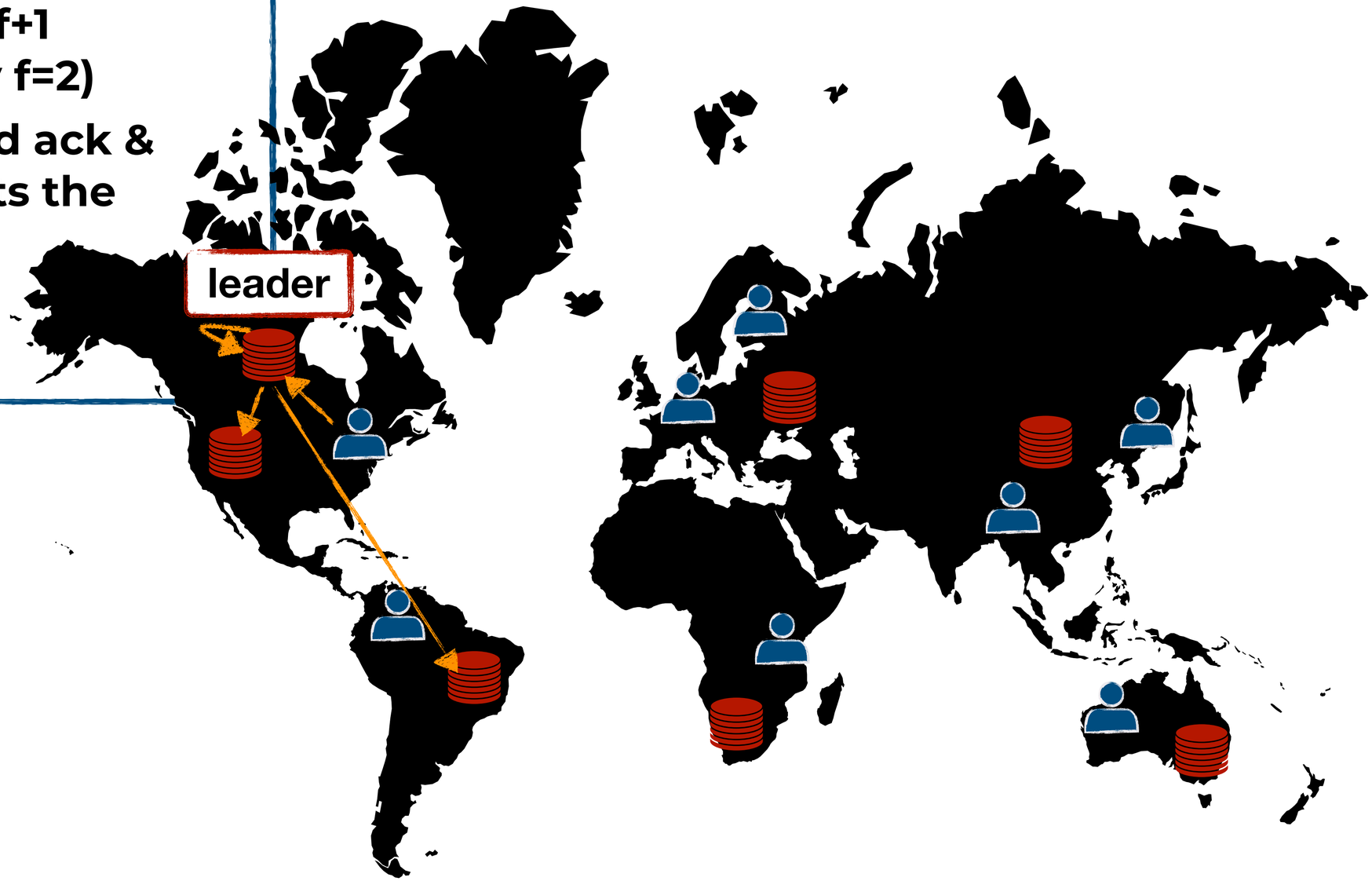
leader-based SMR (e.g. paxos)

- leader receives command
- forwards it to f+1 acceptors (say f=2)



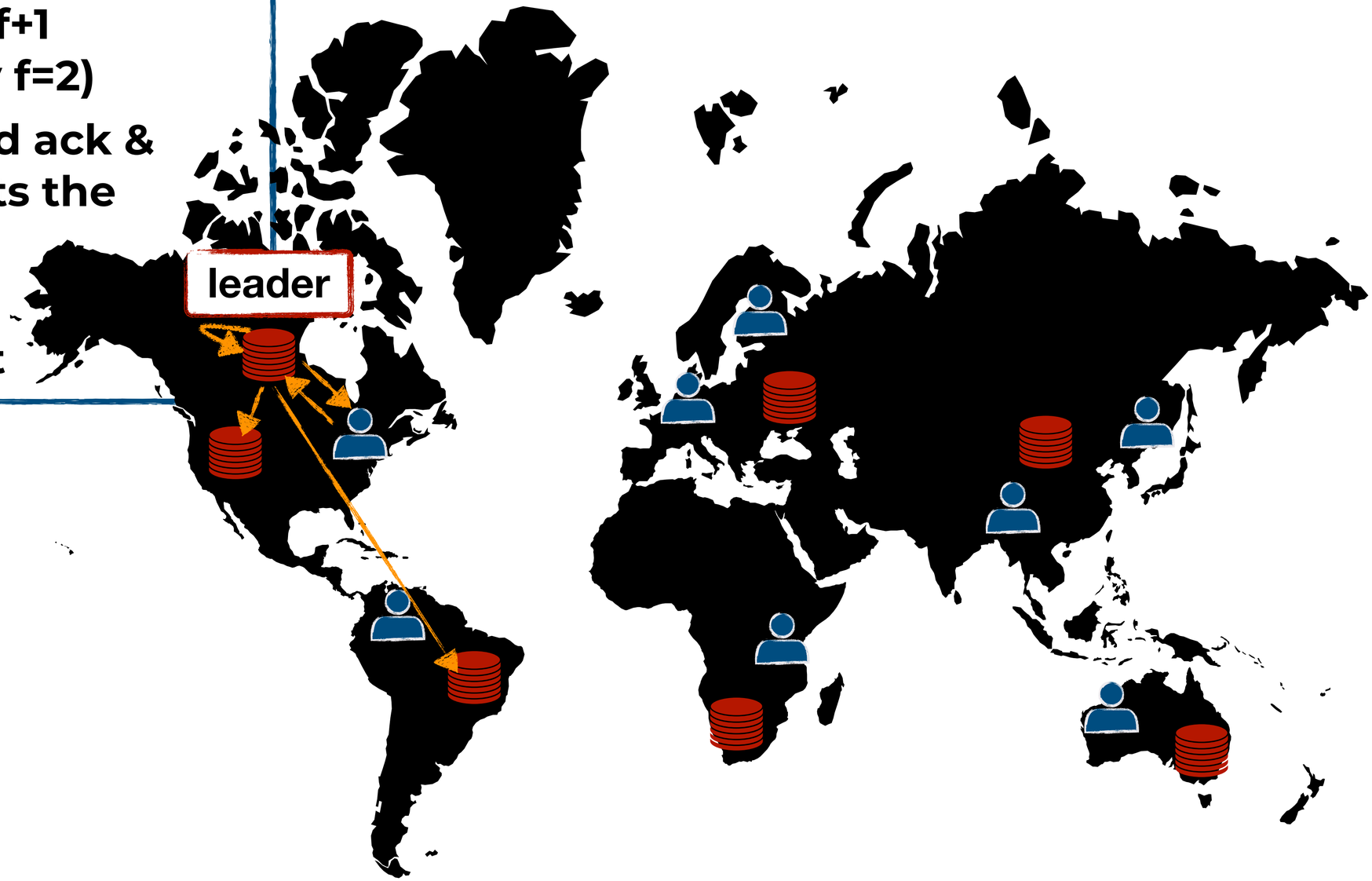
leader-based SMR (e.g. paxos)

- leader receives command
- forwards it to f+1 acceptors (say f=2)
- acceptors send ack & leader commits the command



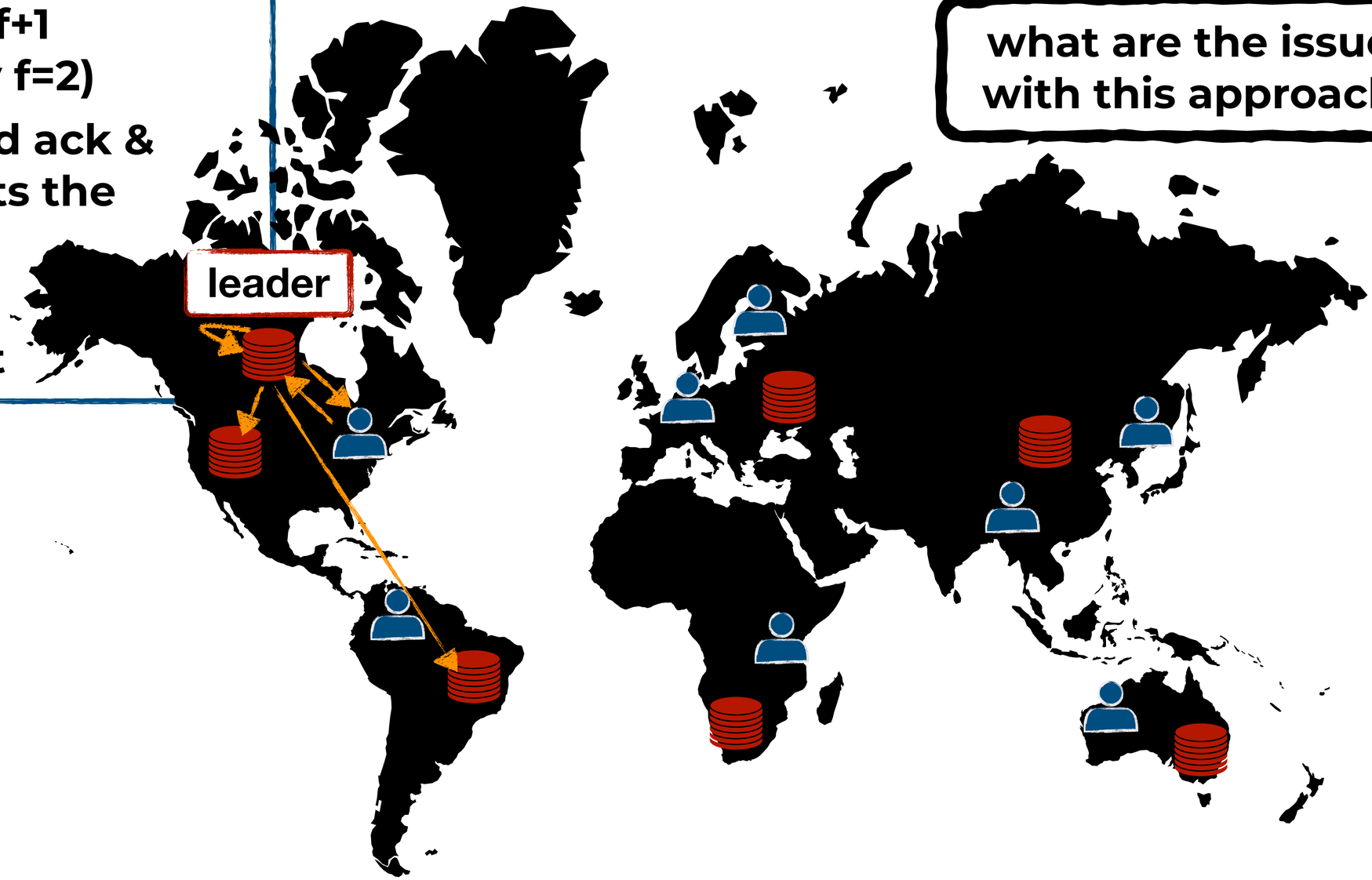
leader-based SMR (e.g. paxos)

- leader receives command
- forwards it to f+1 acceptors (say f=2)
- acceptors send ack & leader commits the command
- leader sends result to client



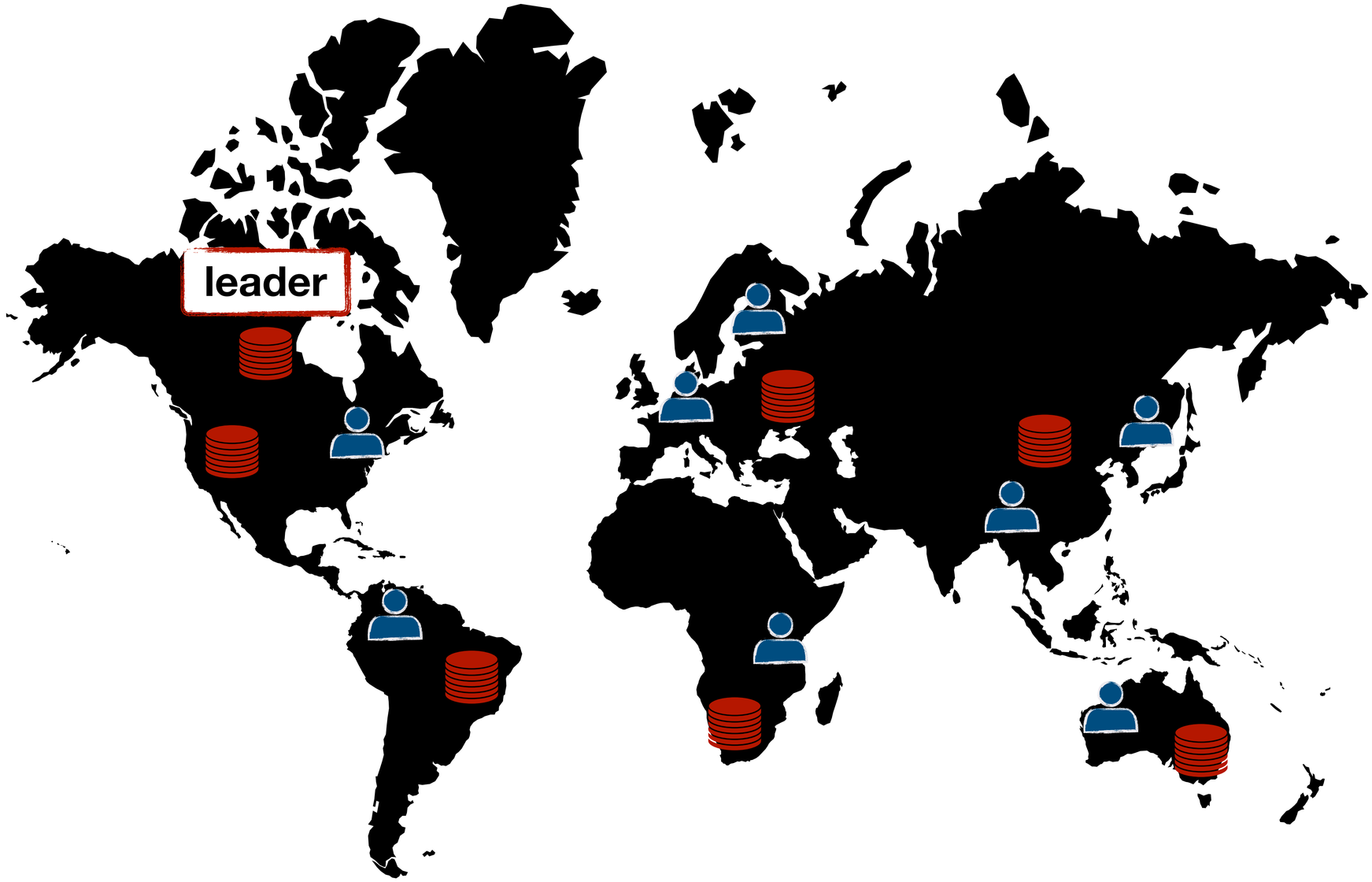
leader-based SMR (e.g. paxos)

- leader receives command
- forwards it to f+1 acceptors (say f=2)
- acceptors send ack & leader commits the command
- leader sends result to client

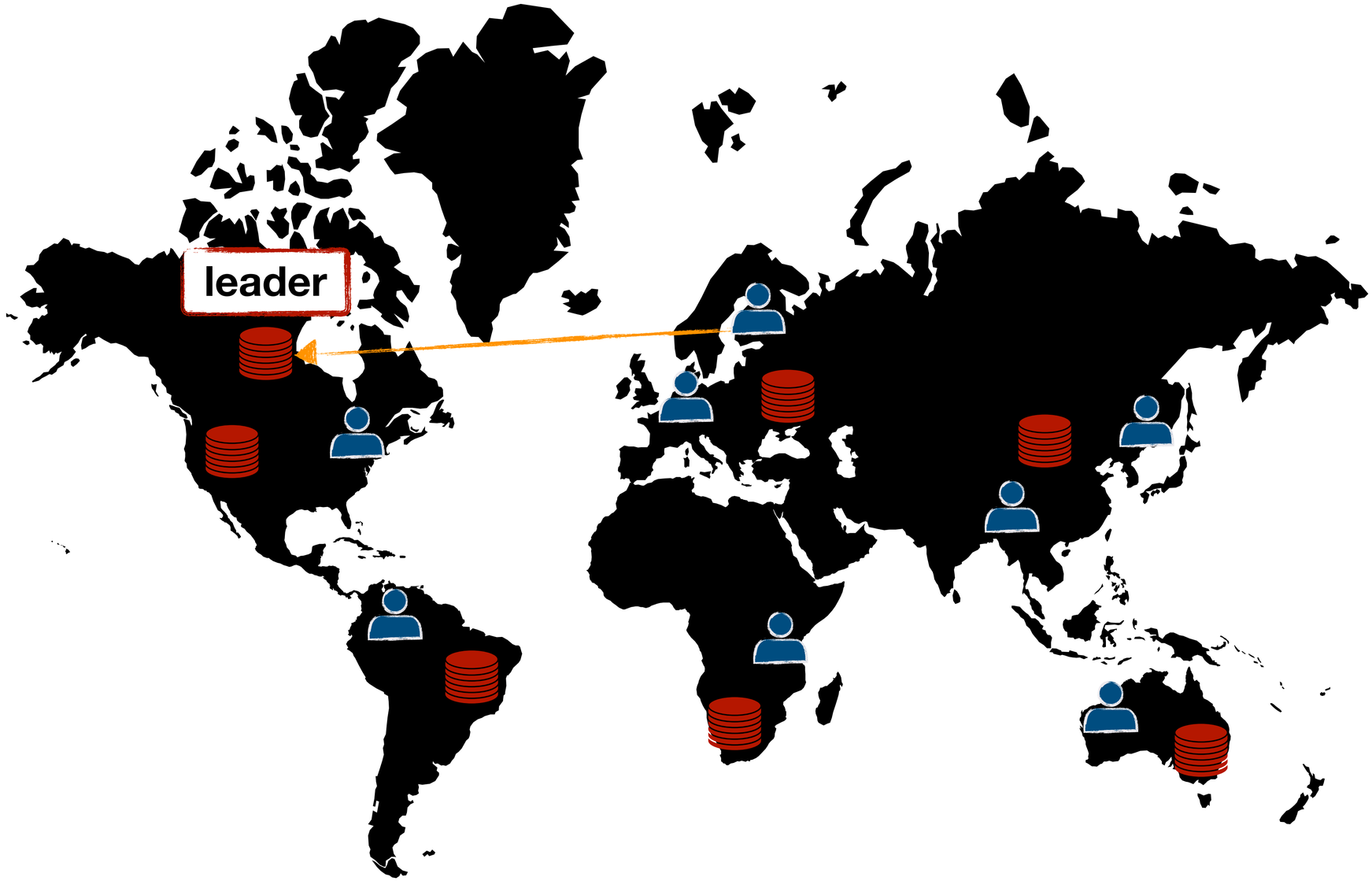


what are the issues with this approach?

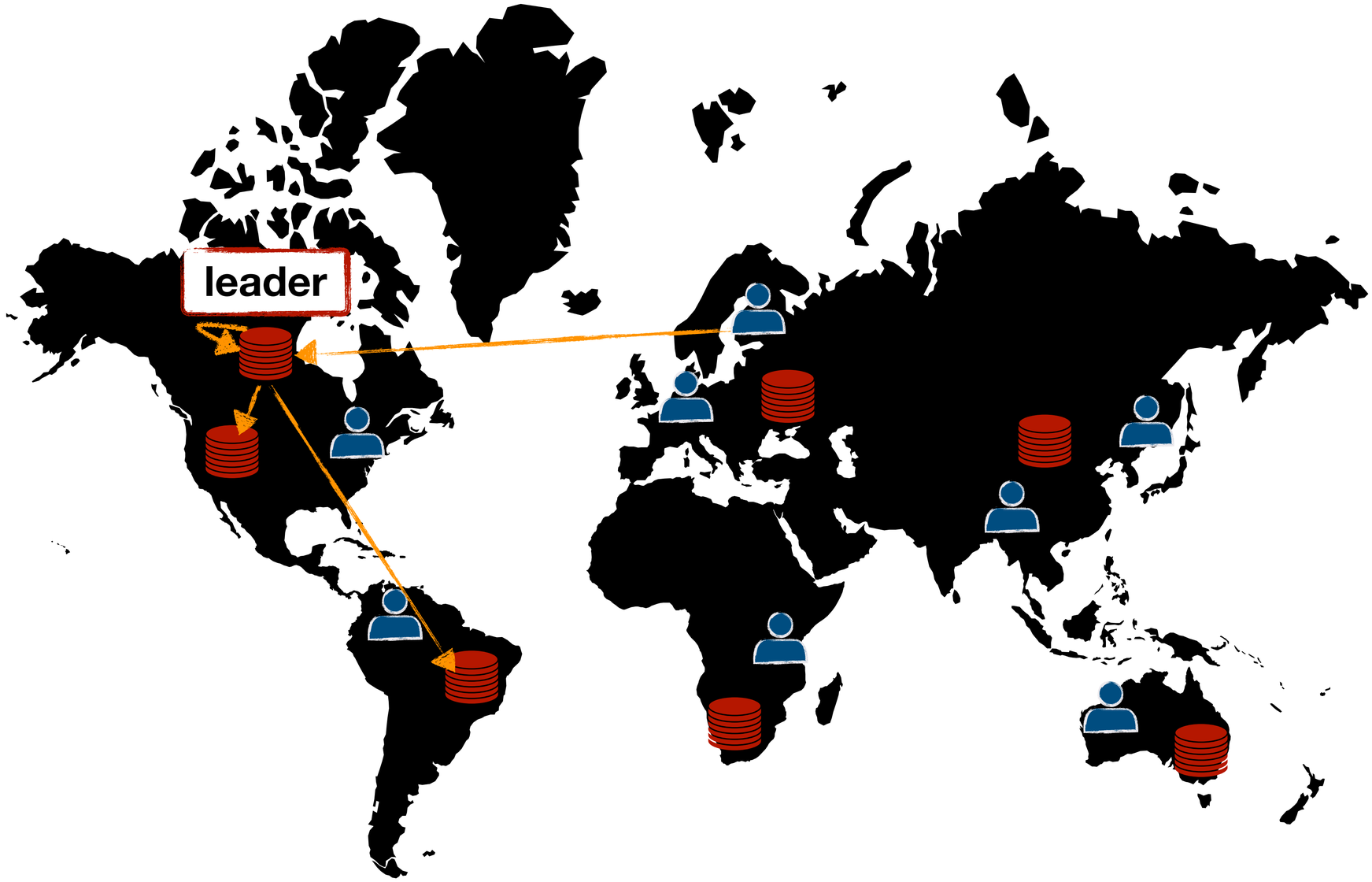
leader-based SMR pitfalls



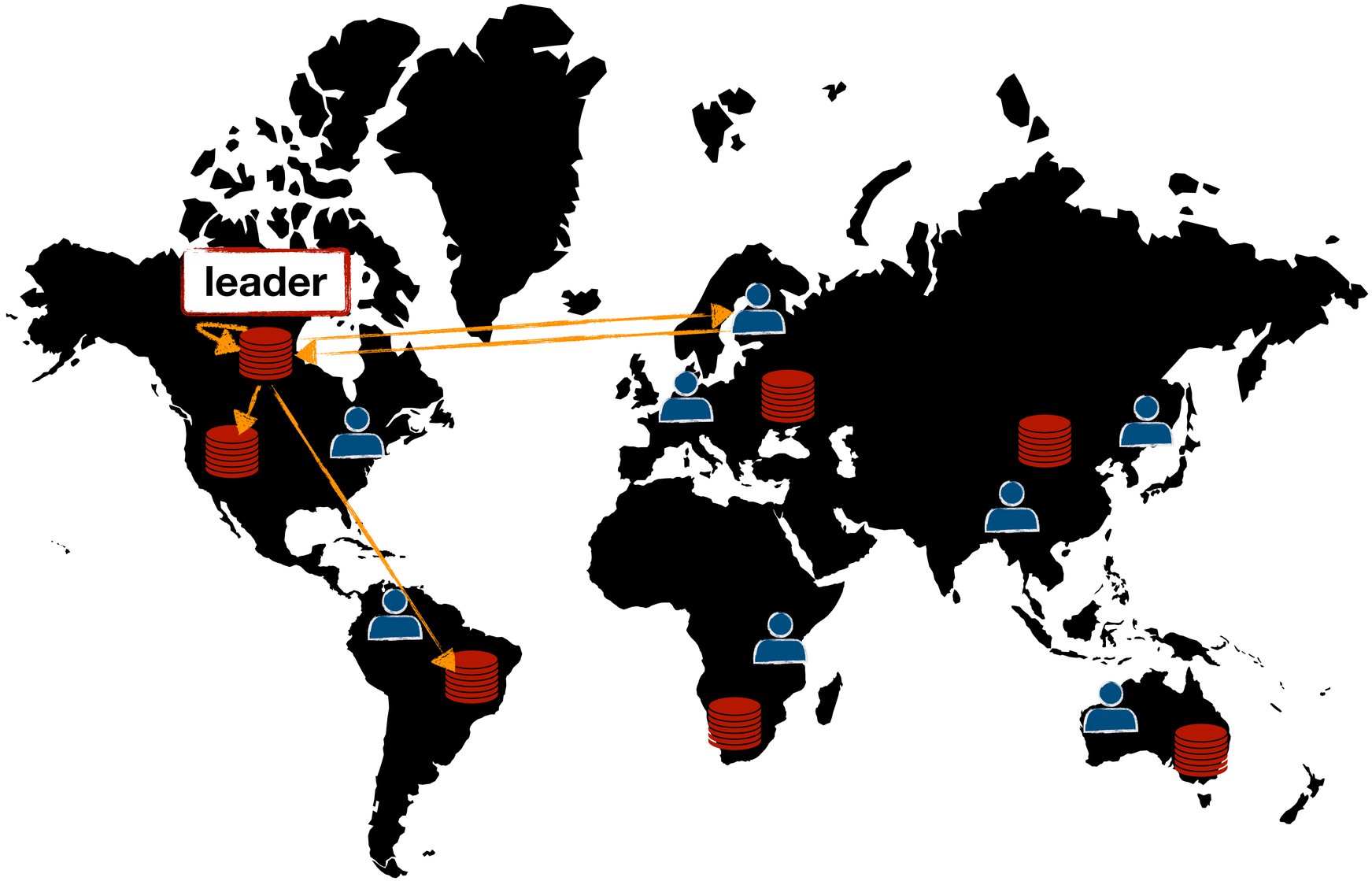
leader-based SMR pitfalls



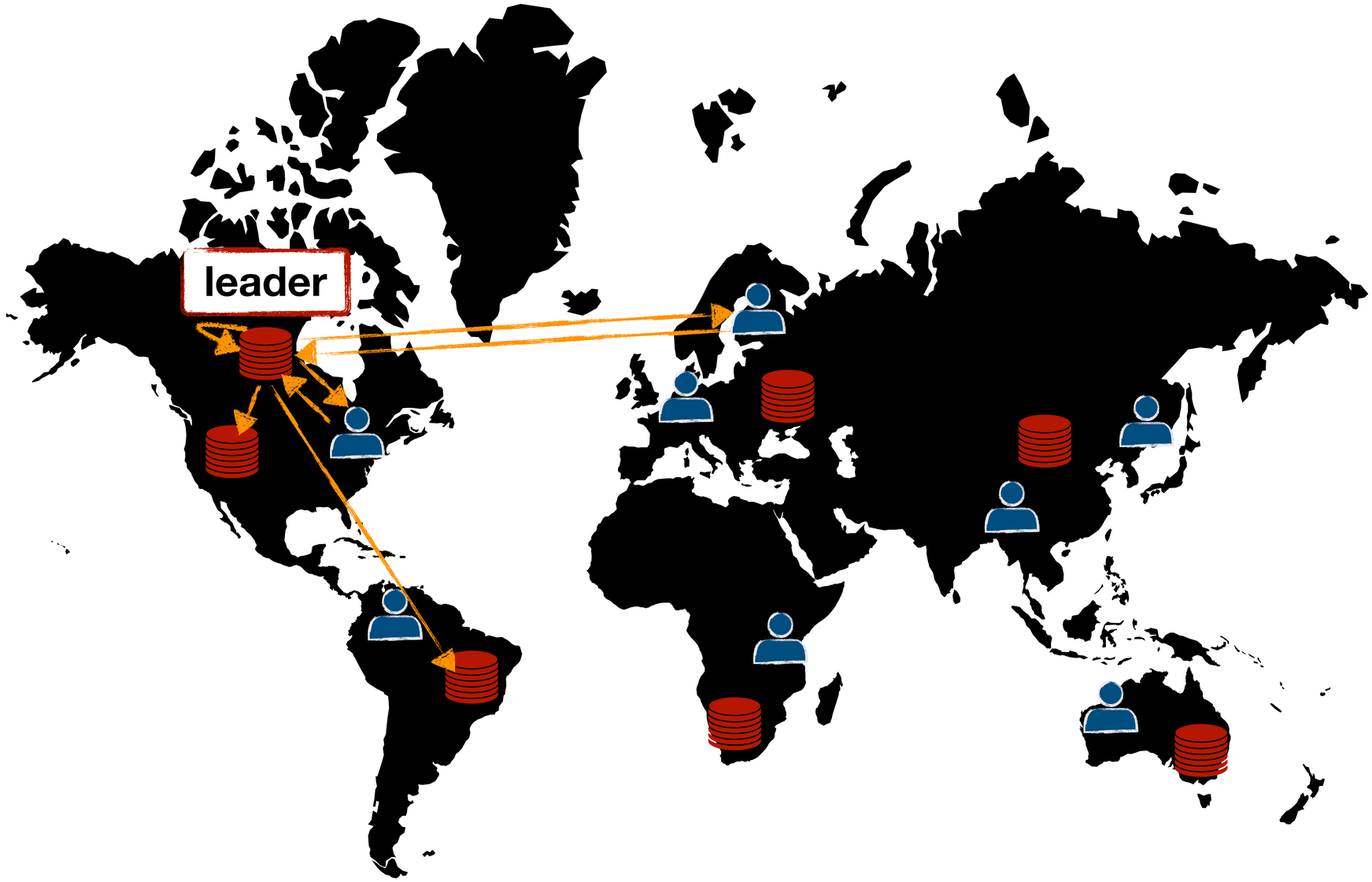
leader-based SMR pitfalls



leader-based SMR pitfalls

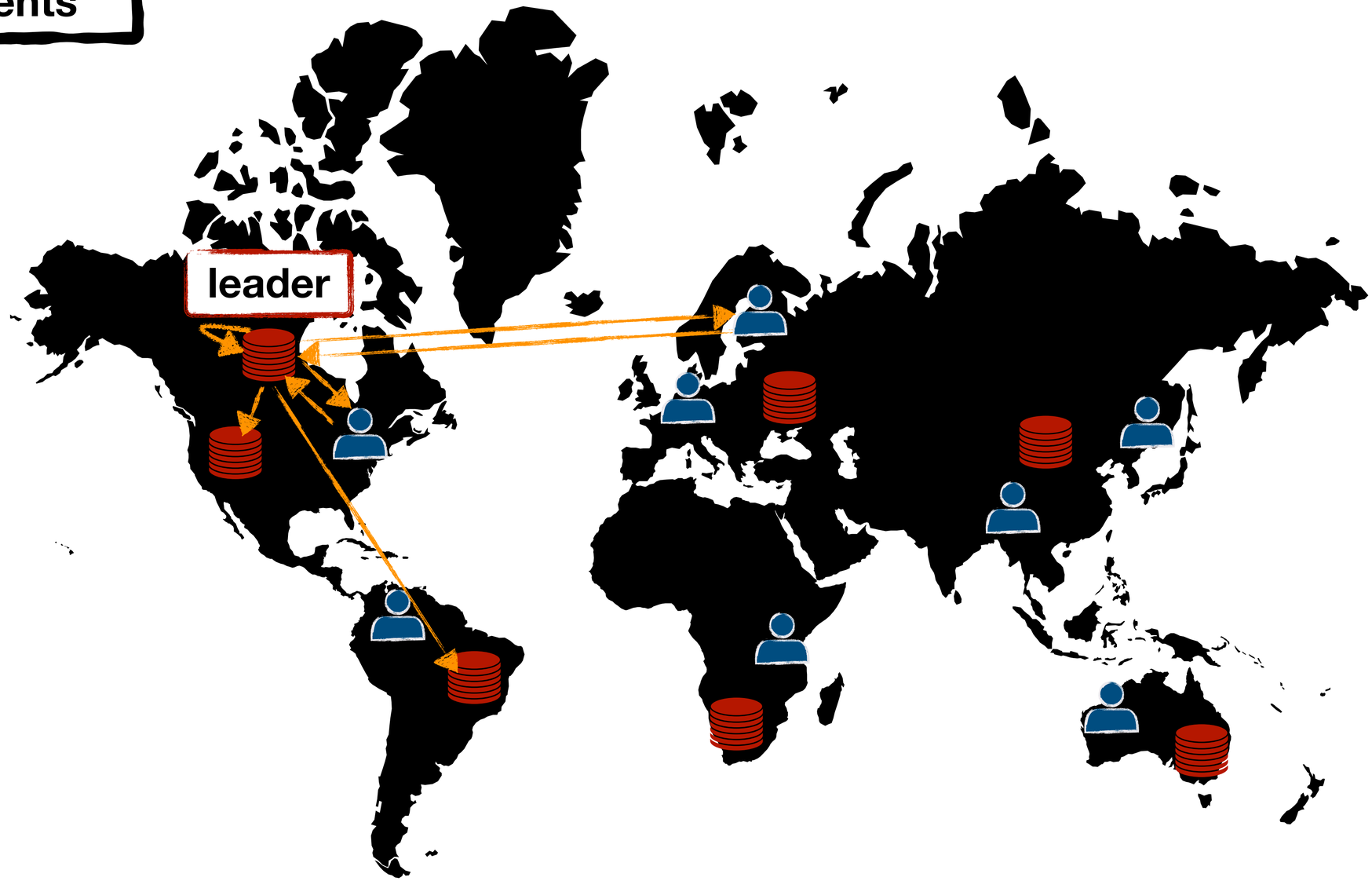


leader-based SMR pitfalls



leader-based SMR pitfalls

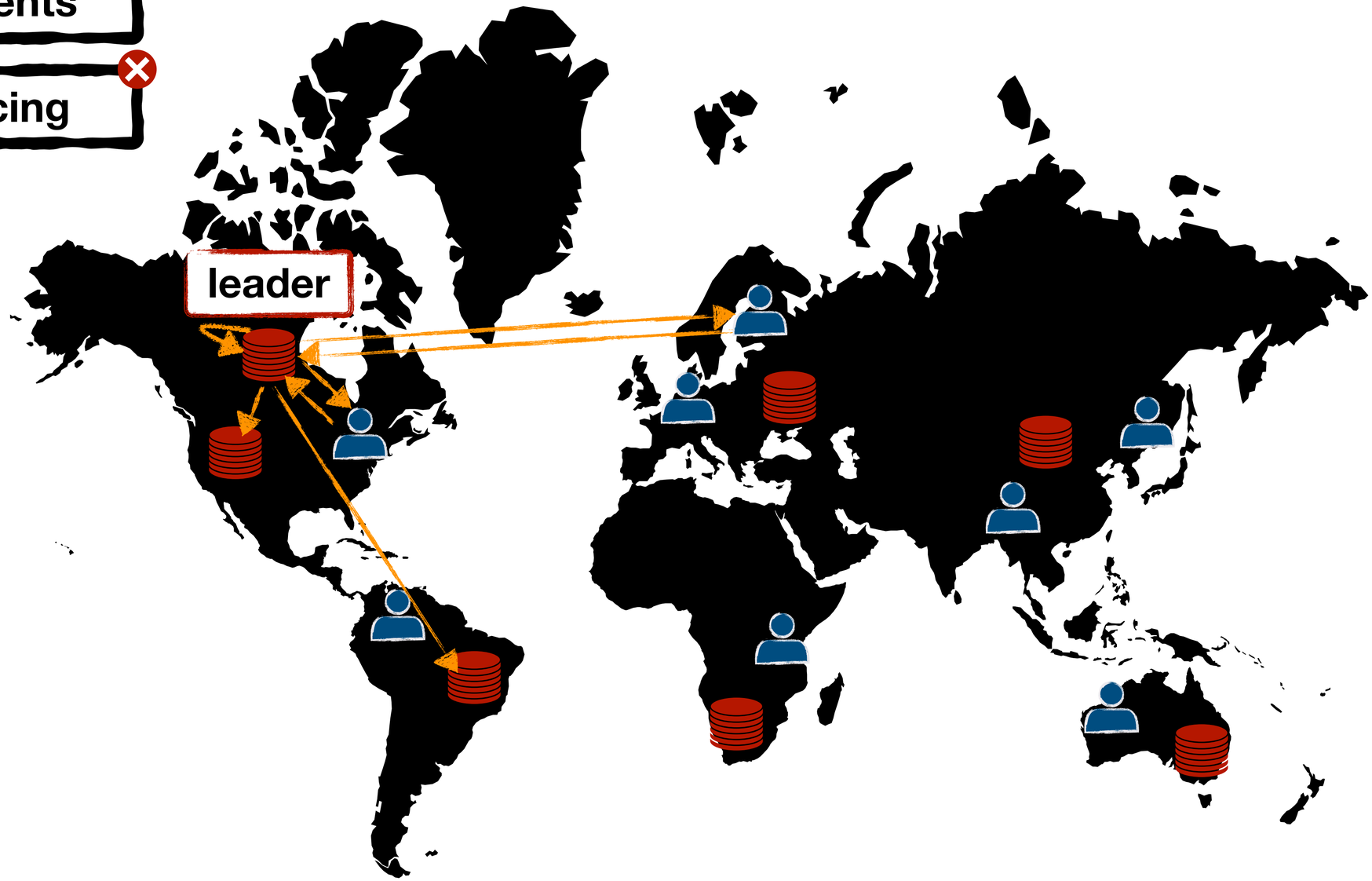
unfairness/high latency for faraway clients



leader-based SMR pitfalls

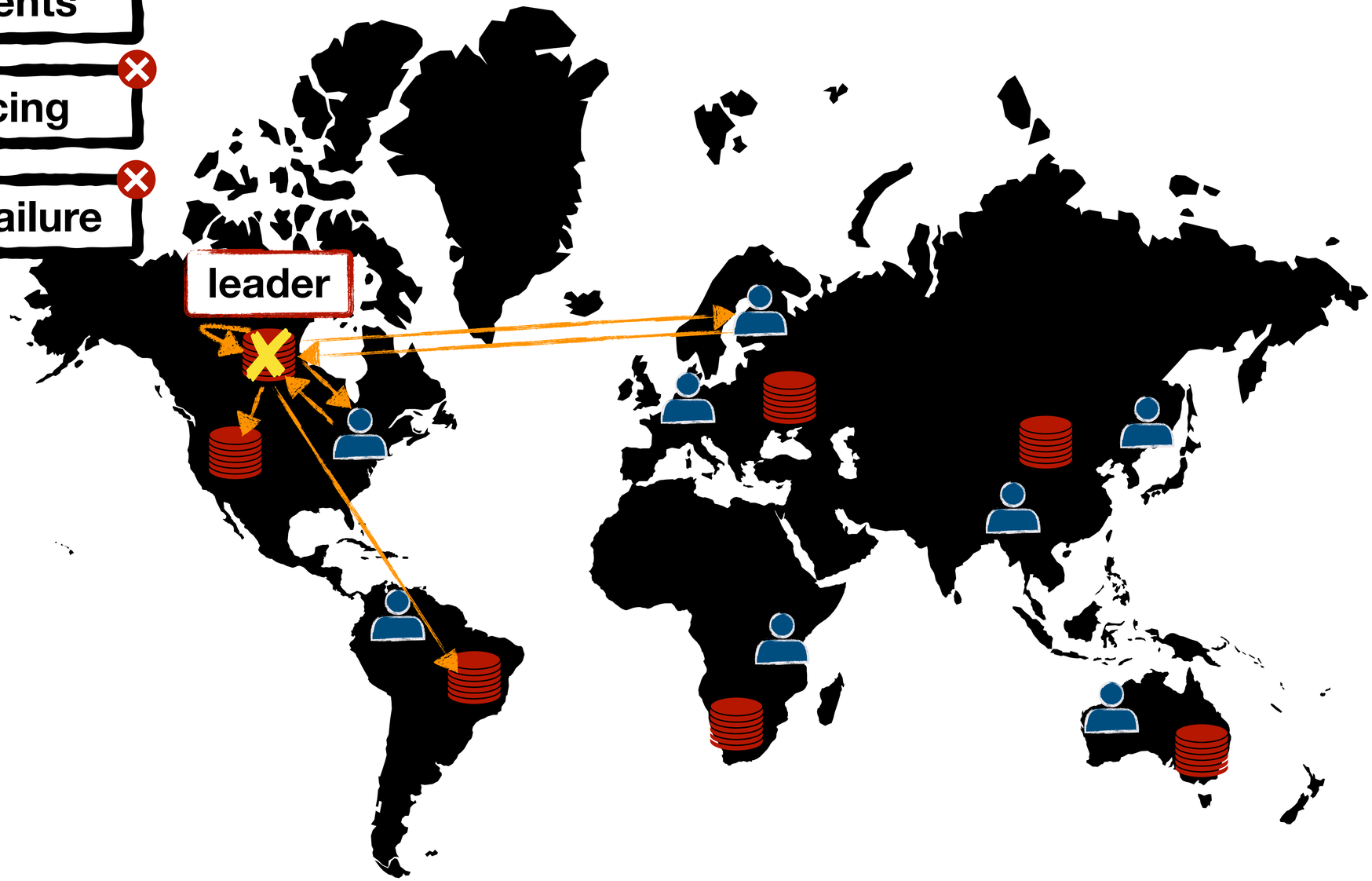
unfairness/high latency for faraway clients

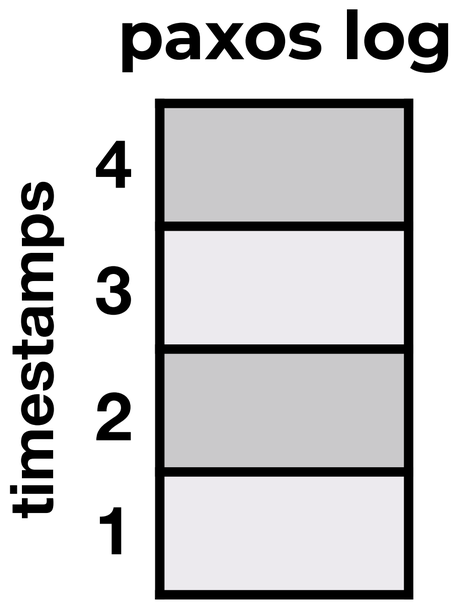
no load balancing

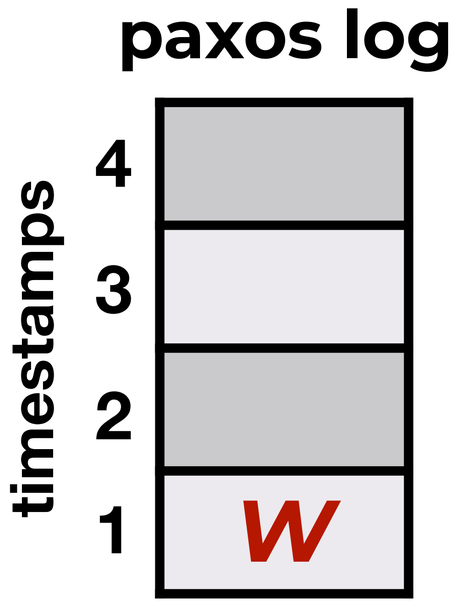


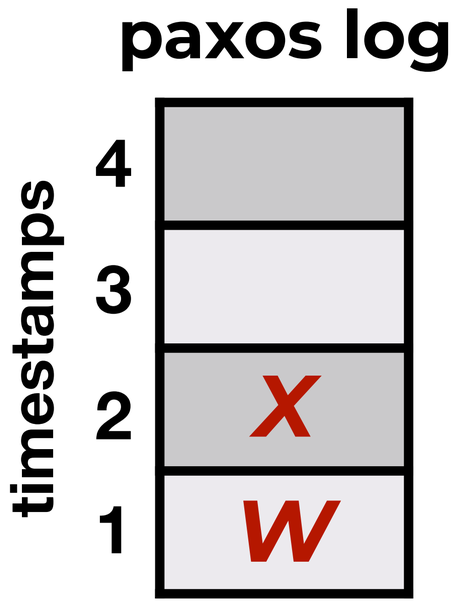
leader-based SMR pitfalls

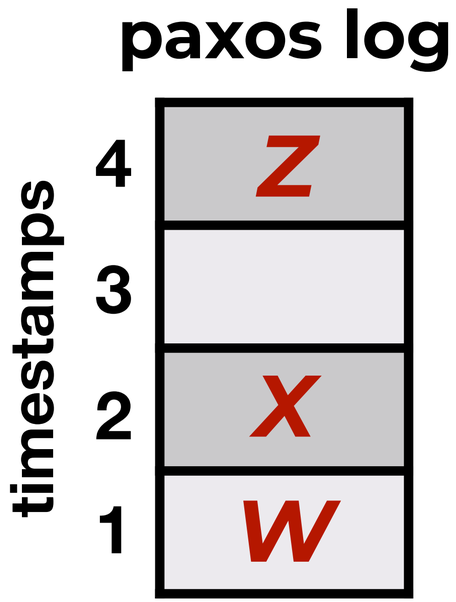
- unfairness/high latency for faraway clients
- no load balancing
- single point of failure

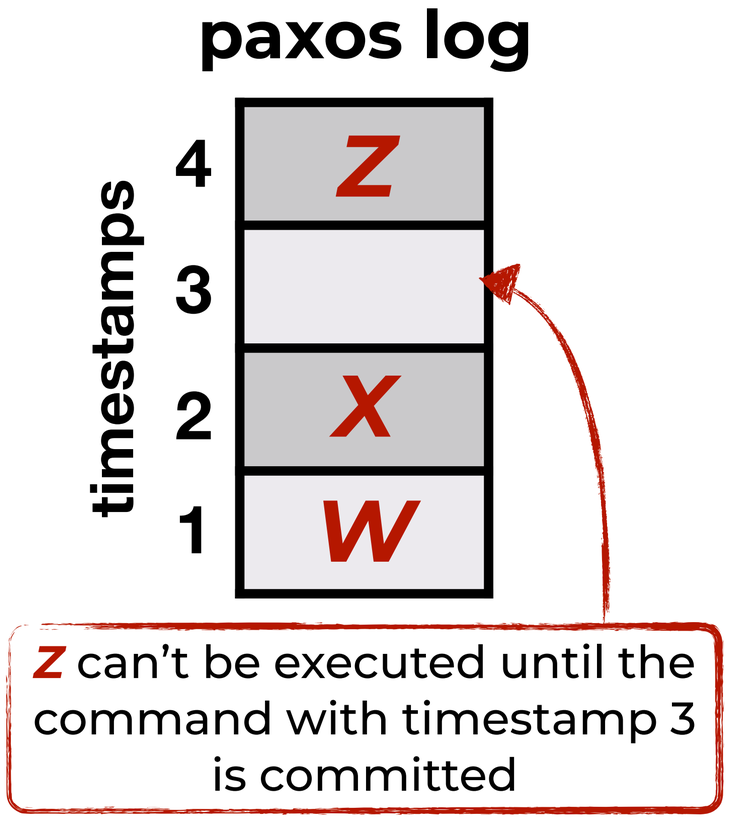



paxos log
timestamps
4
3
2
1

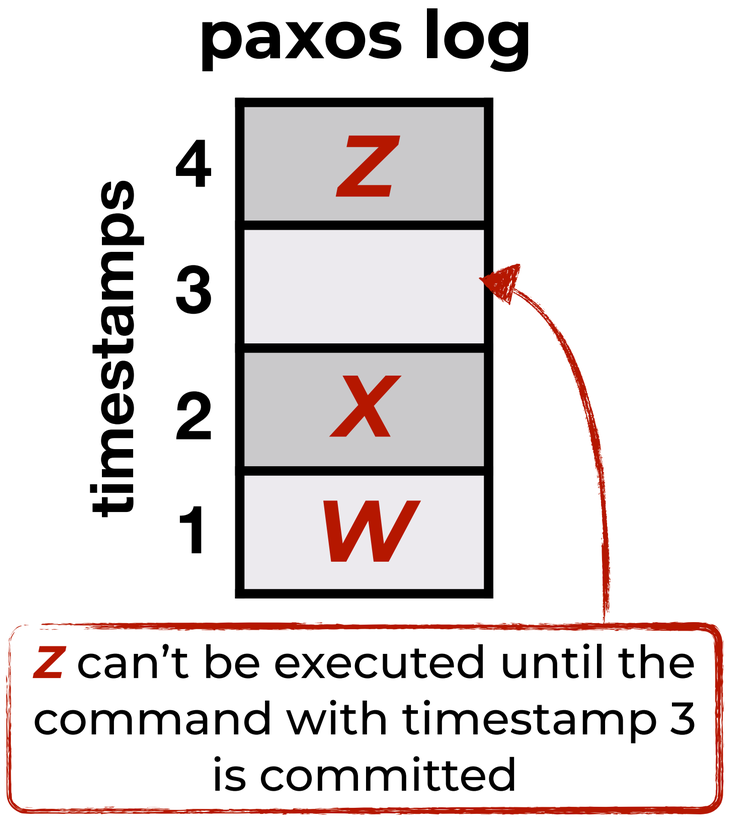

paxos log
timestamps
4
3
2
1
W

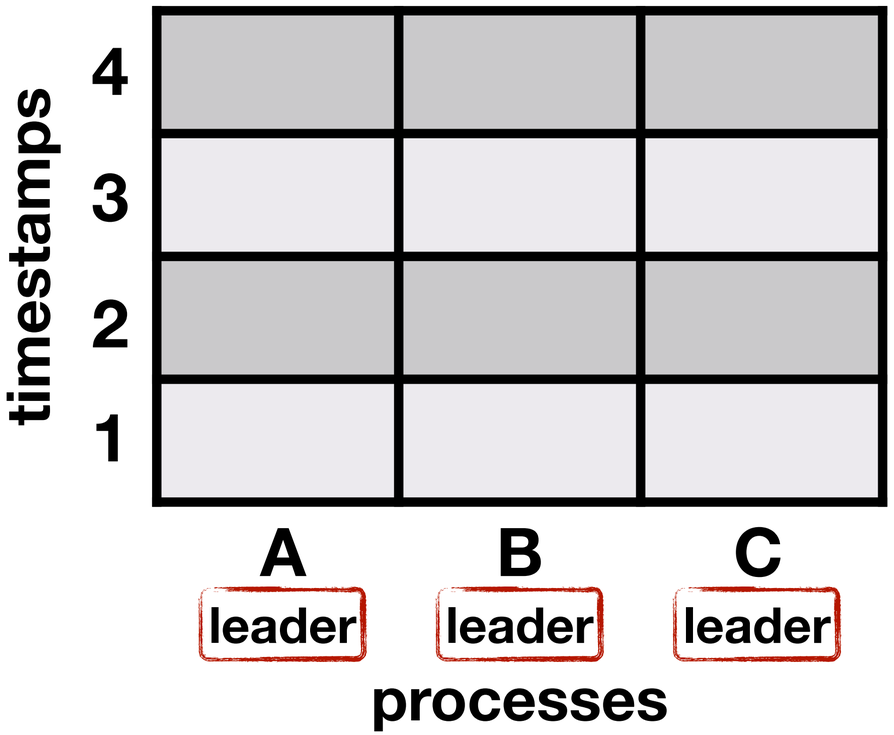

paxos log
timestamps
4
3
2
1
X
W


paxos log
timestamps
4
3
2
1
Z
X
W

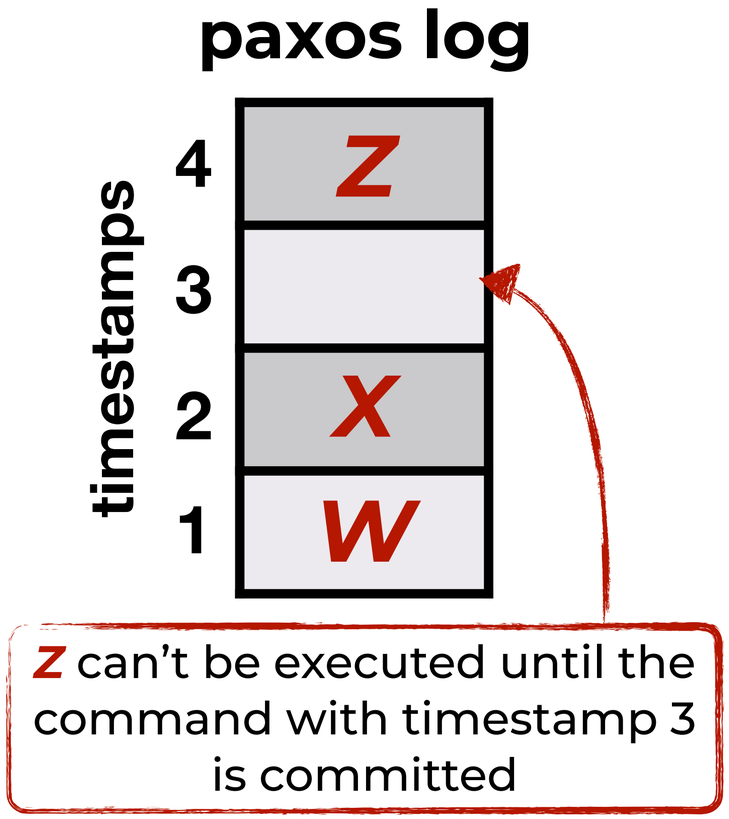

paxos log
timestamps
4
3
2
1
Z
X
W
Z can’t be executed until the command with timestamp 3 is committed

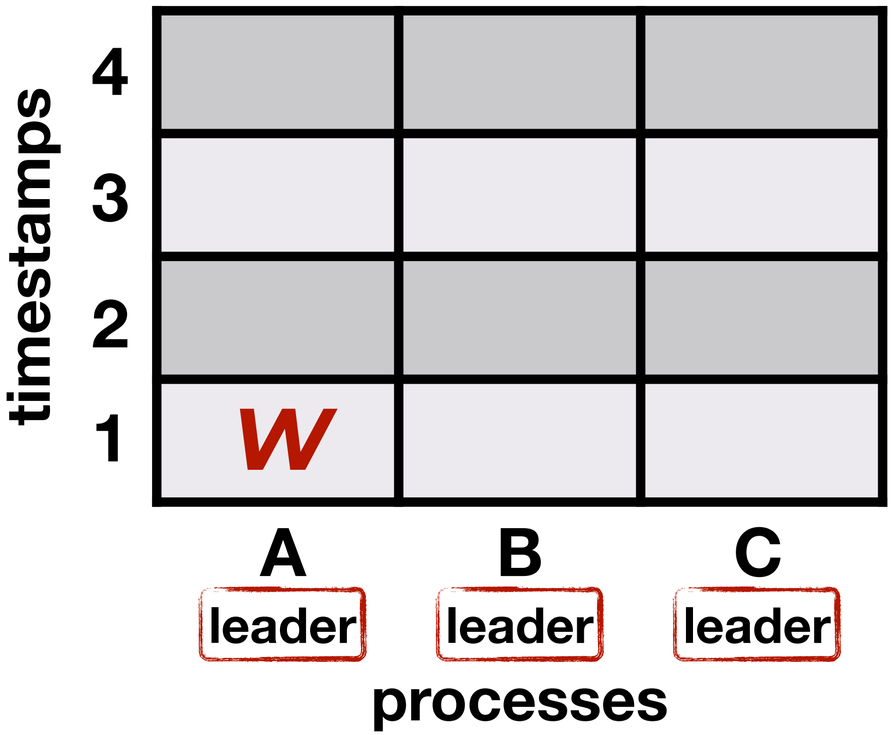
multi-leader SMR (e.g. mencius)



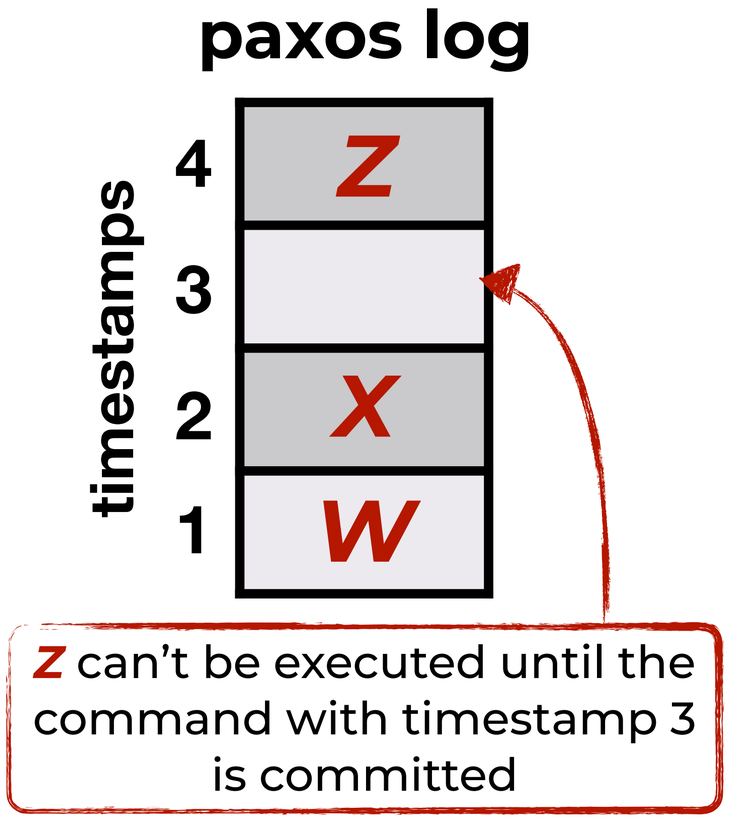


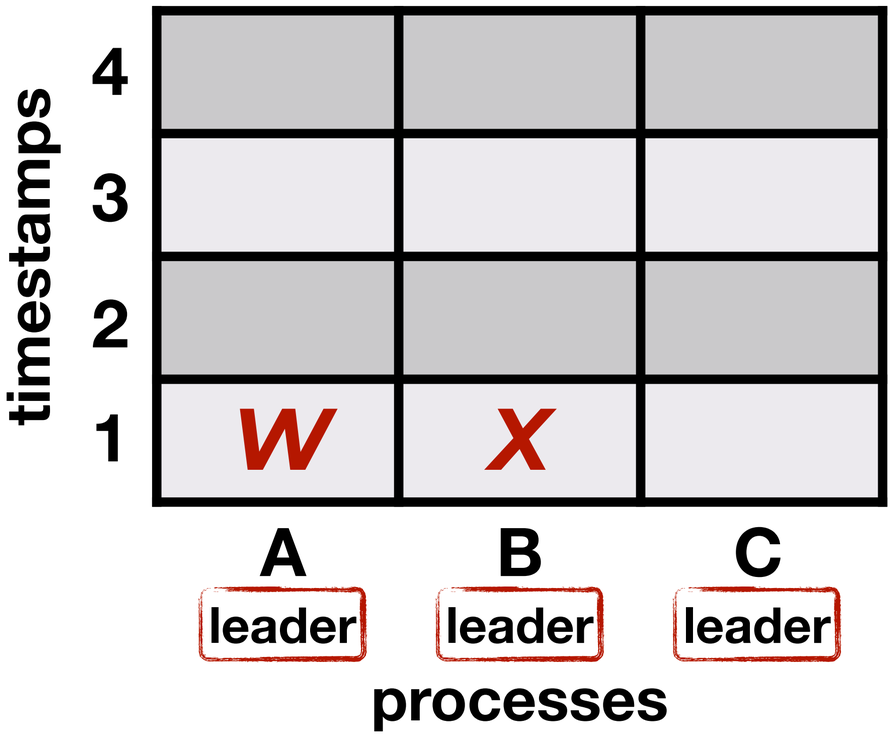
multi-leader SMR (e.g. mencius)



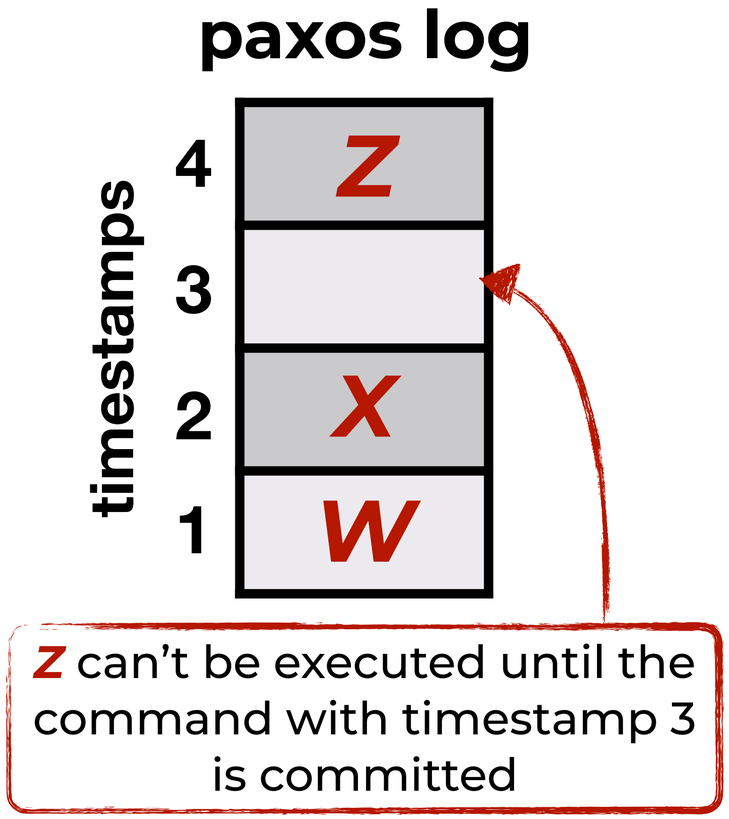


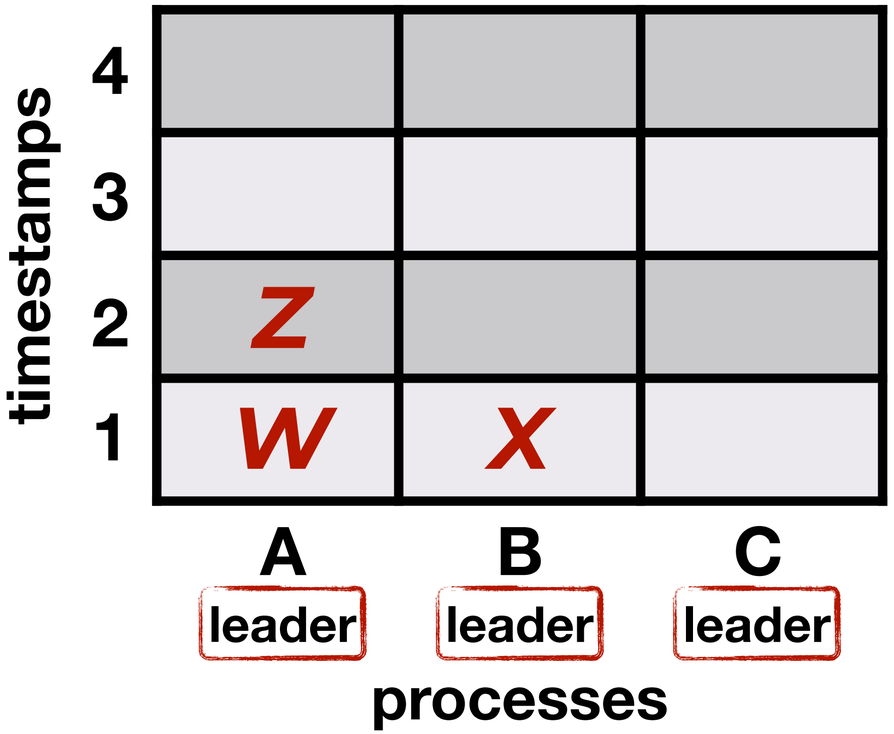
multi-leader SMR (e.g. mencius)



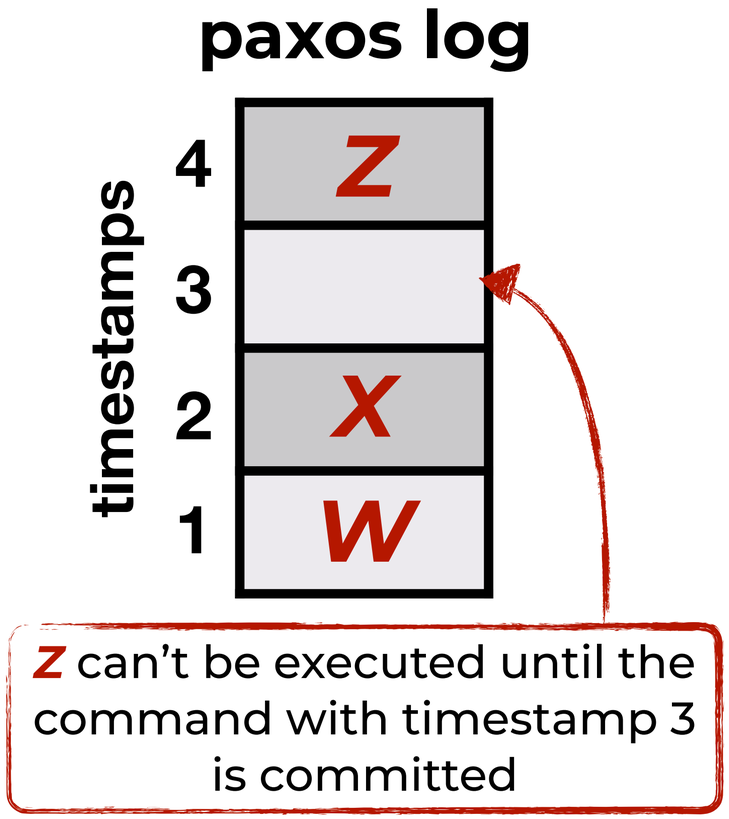


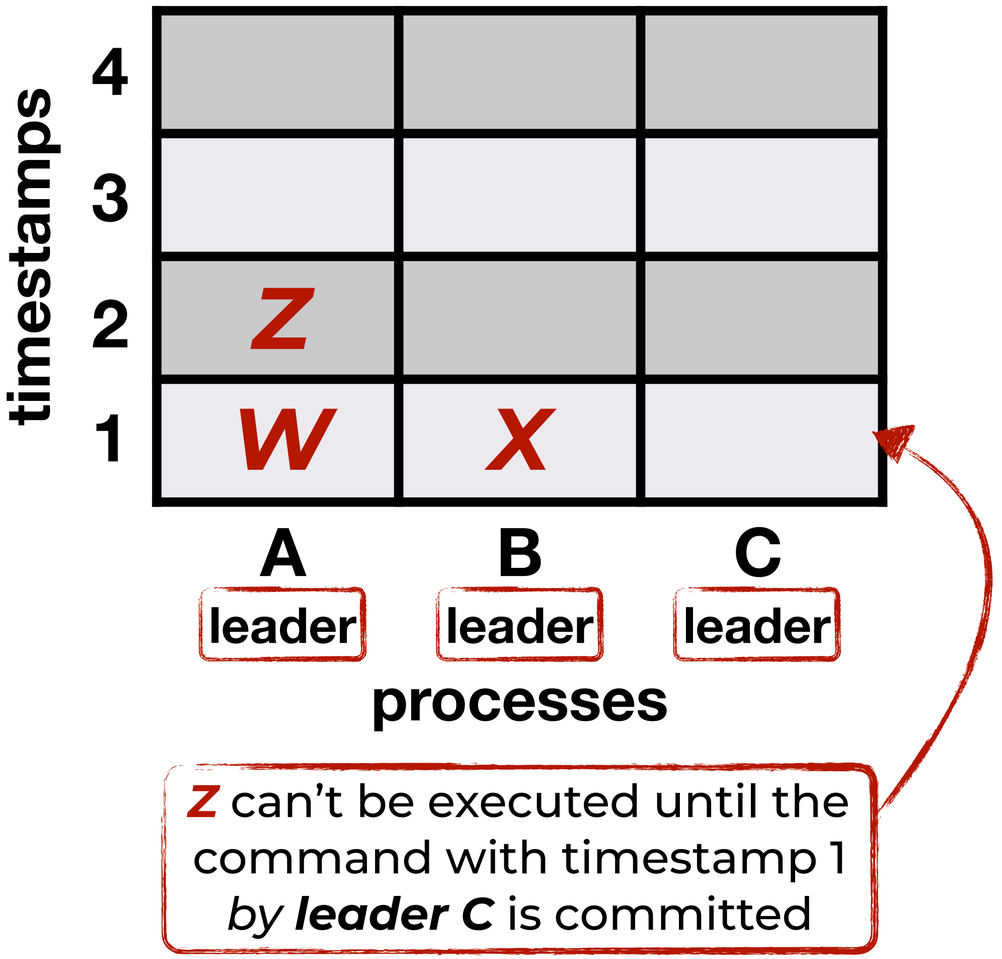
multi-leader SMR (e.g. mencius)



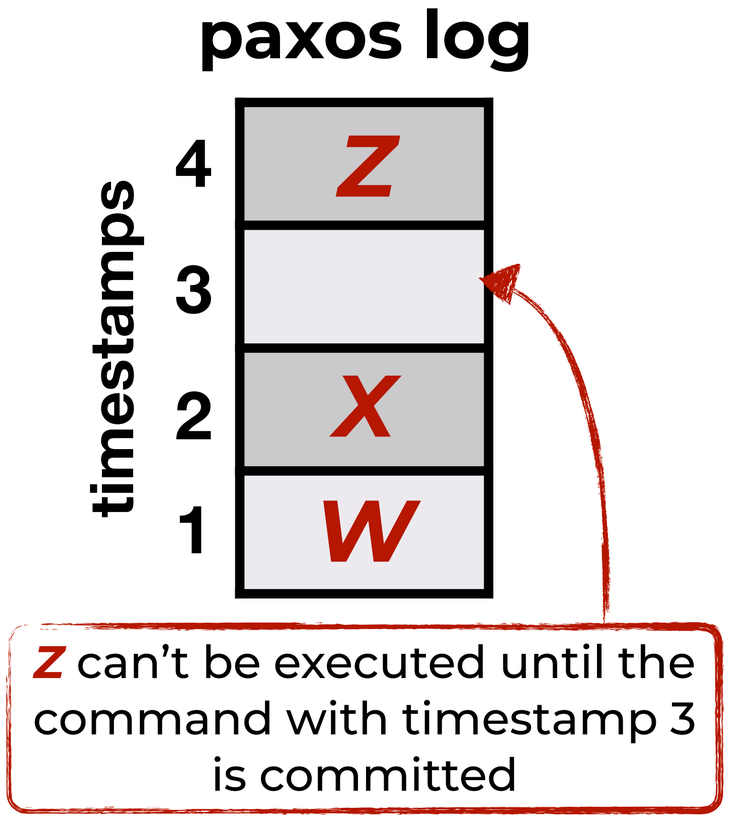


multi-leader SMR (e.g. mencius)

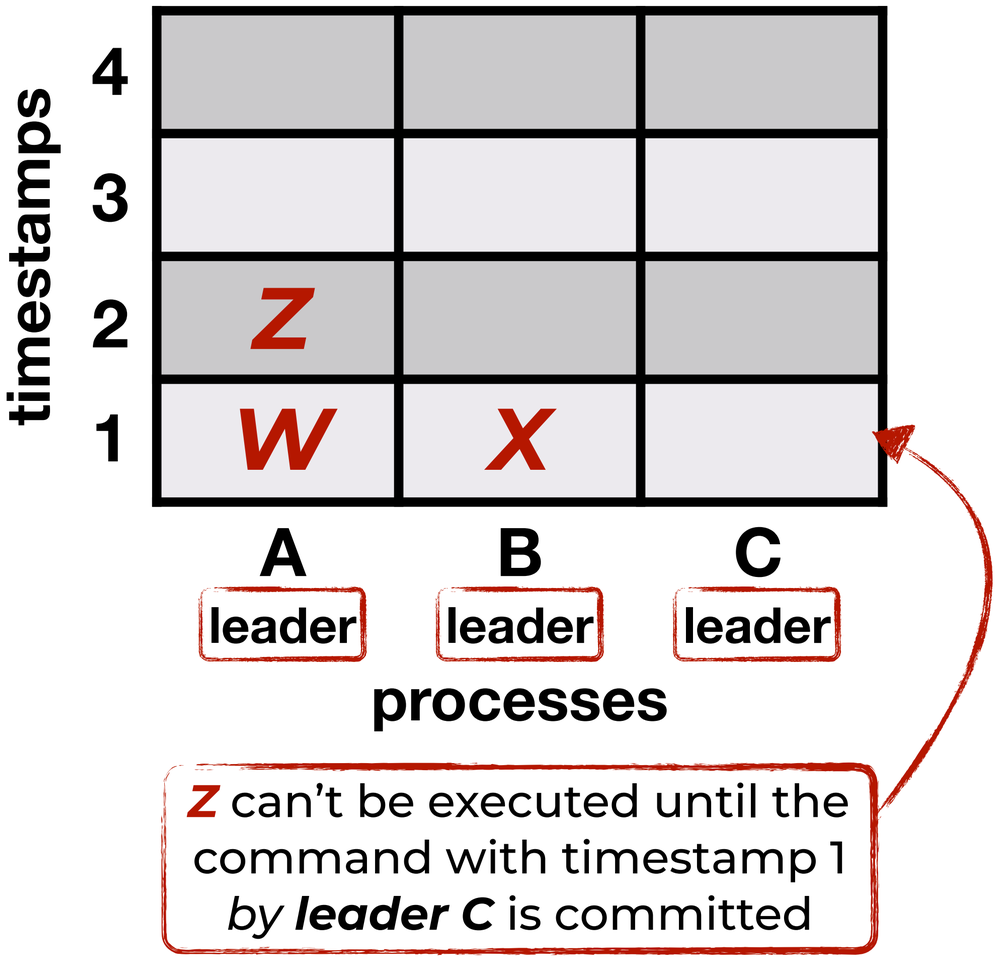




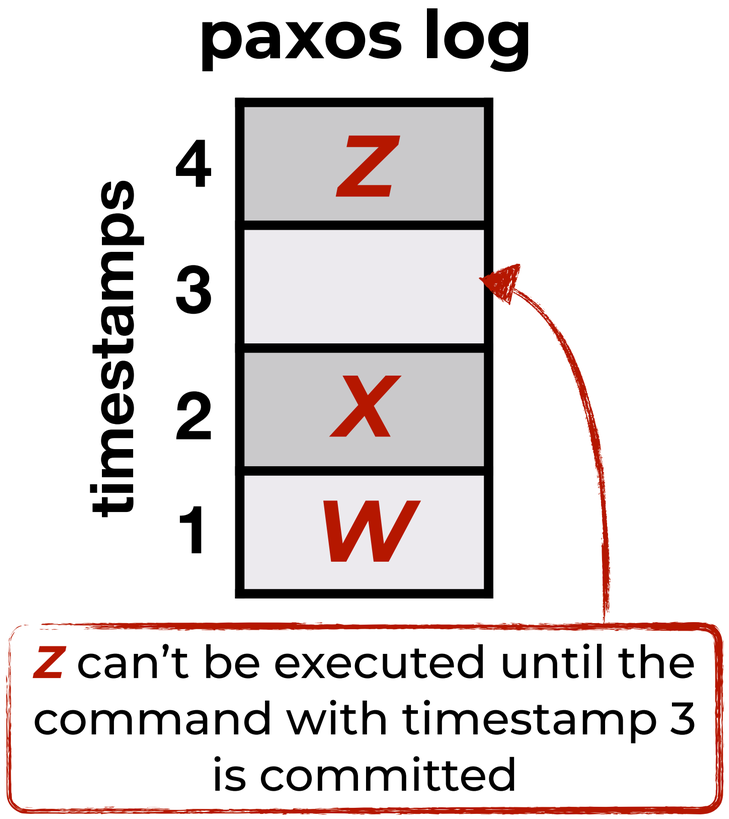
multi-leader SMR (e.g. mencius)



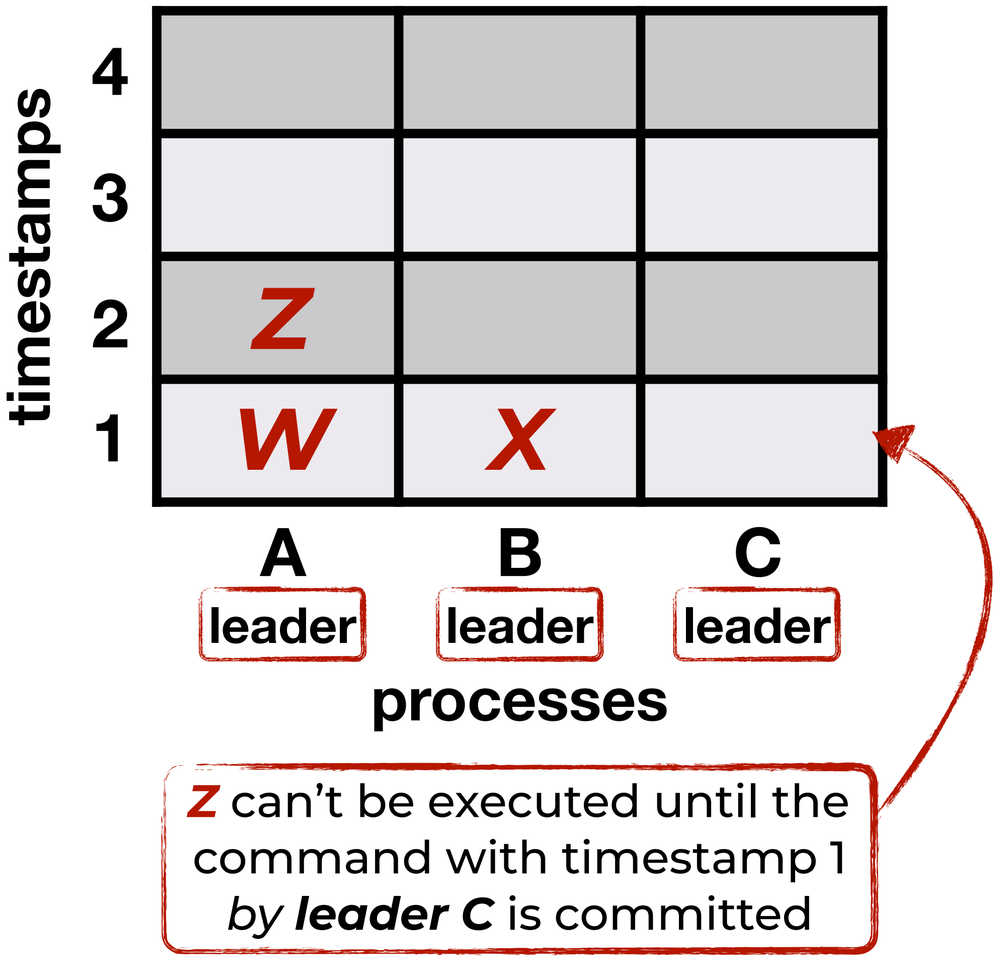
- fairness
- load balancing



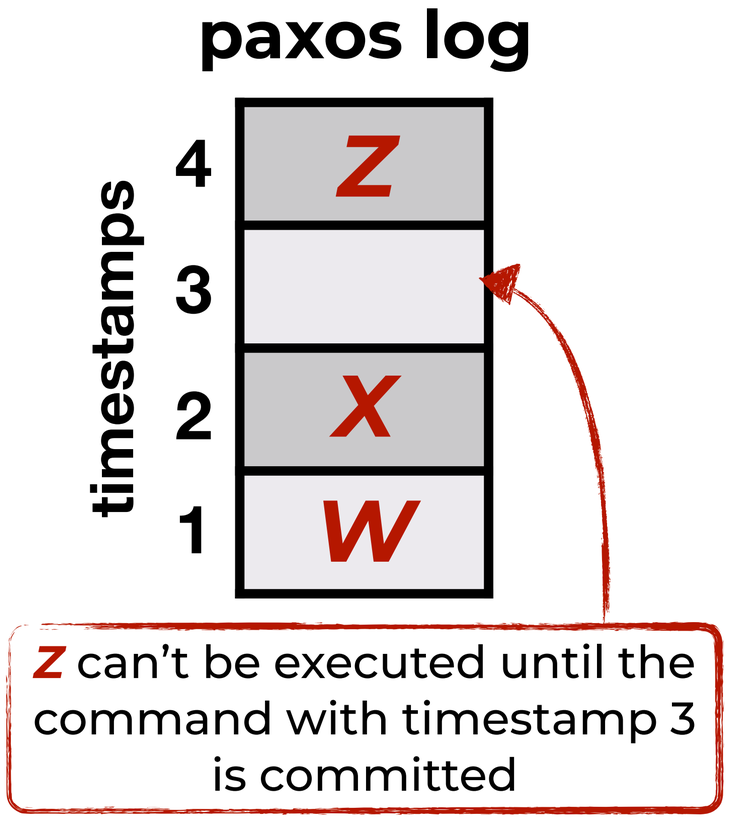
multi-leader SMR (e.g. mencius)



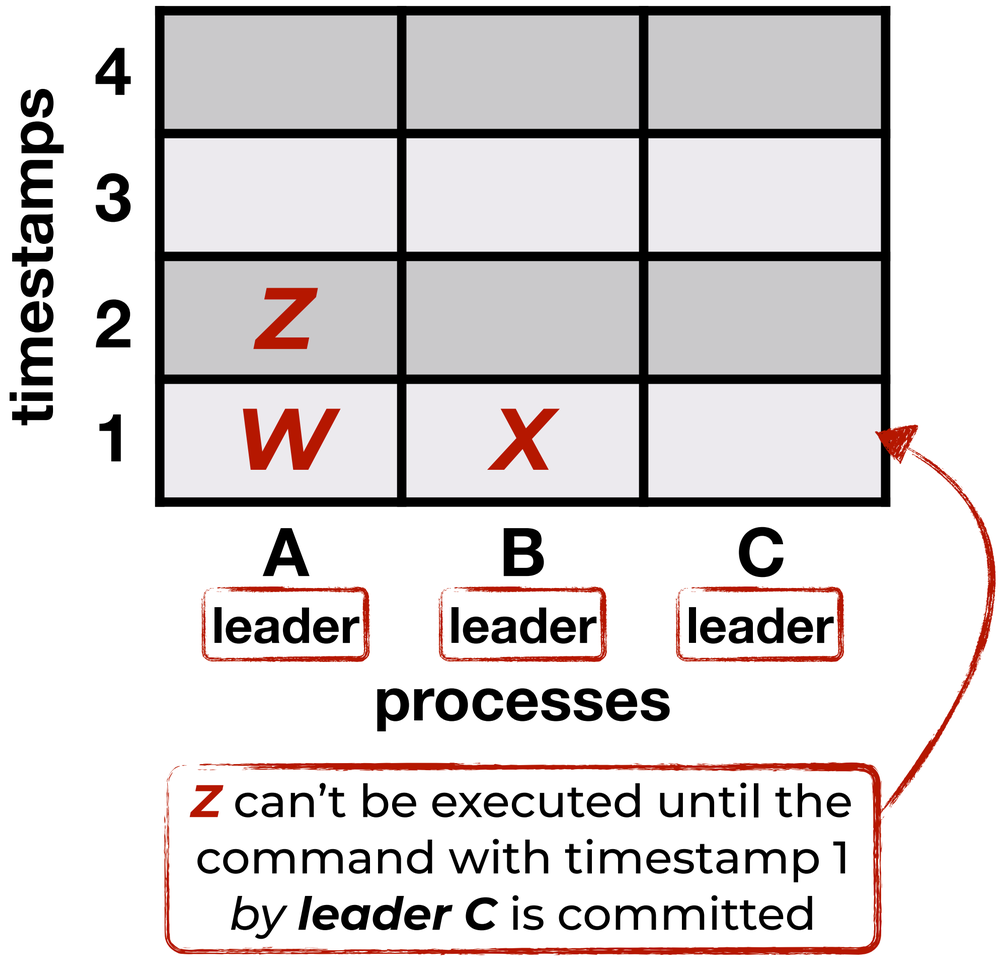
- fairness
- load balancing
- as fast as the slowest



multi-leader SMR (e.g. mencius)



- fairness ✓
- load balancing ✓
- as fast as the slowest ✗



root problem: a command's timestamp is computed by a single process

leaderless SMR with tempo

leaderless SMR with tempo

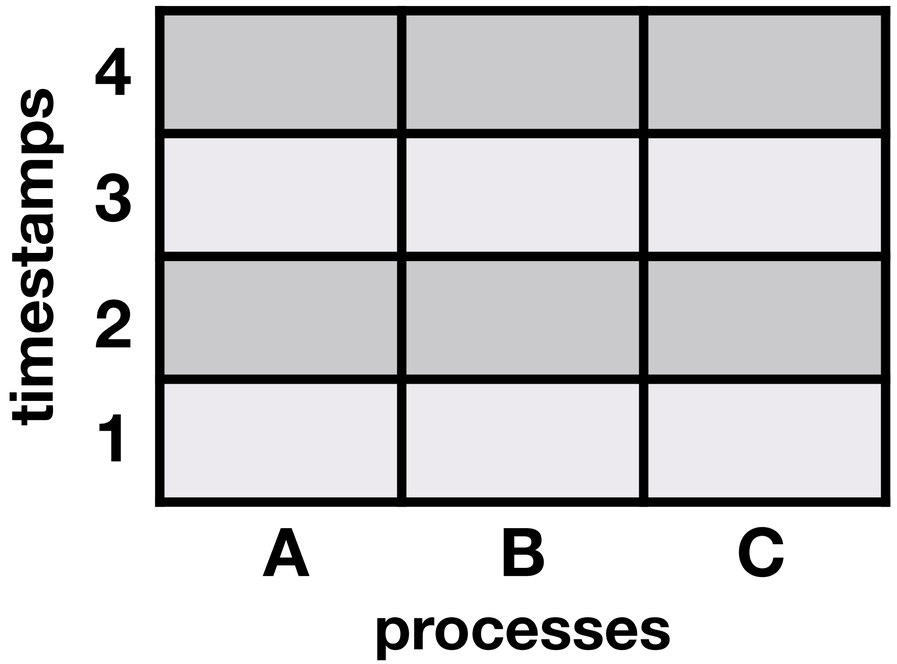
- a majority of processes proposes a timestamp for the command

leaderless SMR with tempo

- a majority of processes proposes a timestamp for the command
- the command's timestamp is the highest proposal

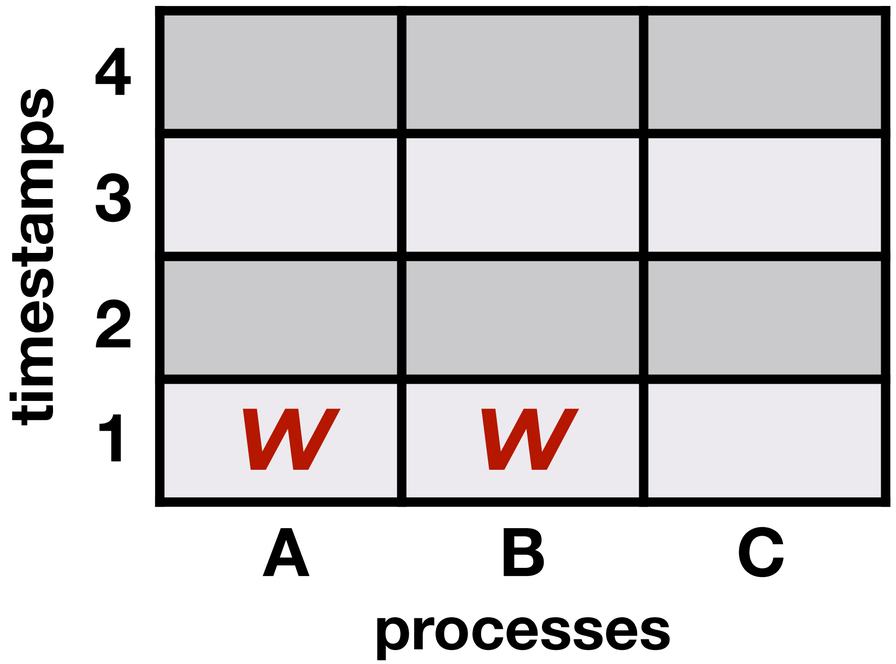
leaderless SMR with tempo

- a majority of processes proposes a timestamp for the command
- the command's timestamp is the highest proposal



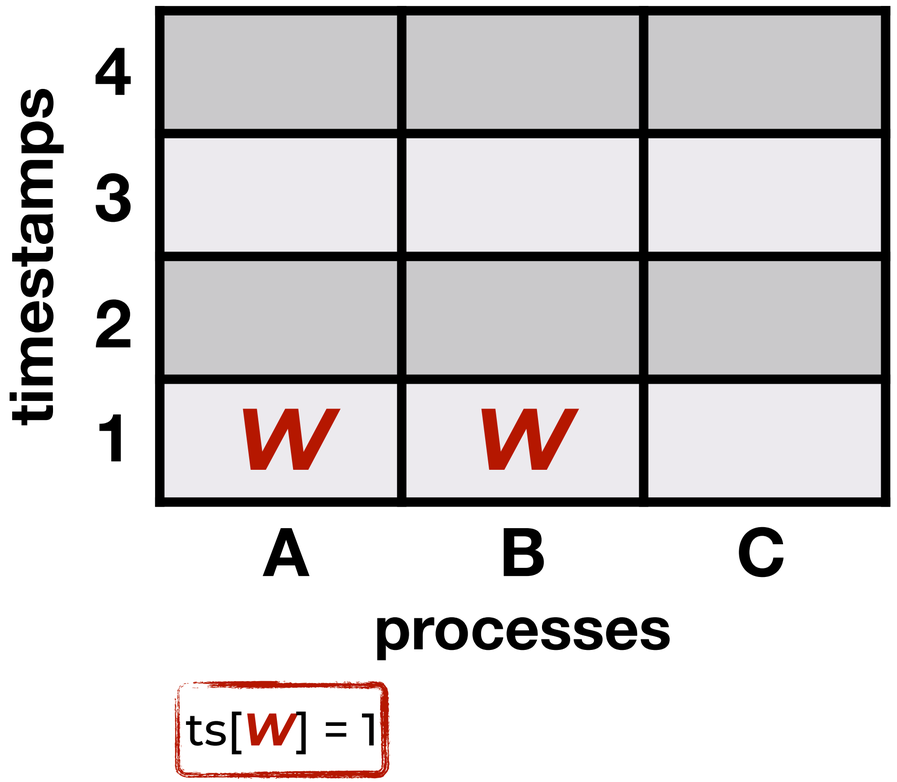
leaderless SMR with tempo

- a majority of processes proposes a timestamp for the command
- the command's timestamp is the highest proposal



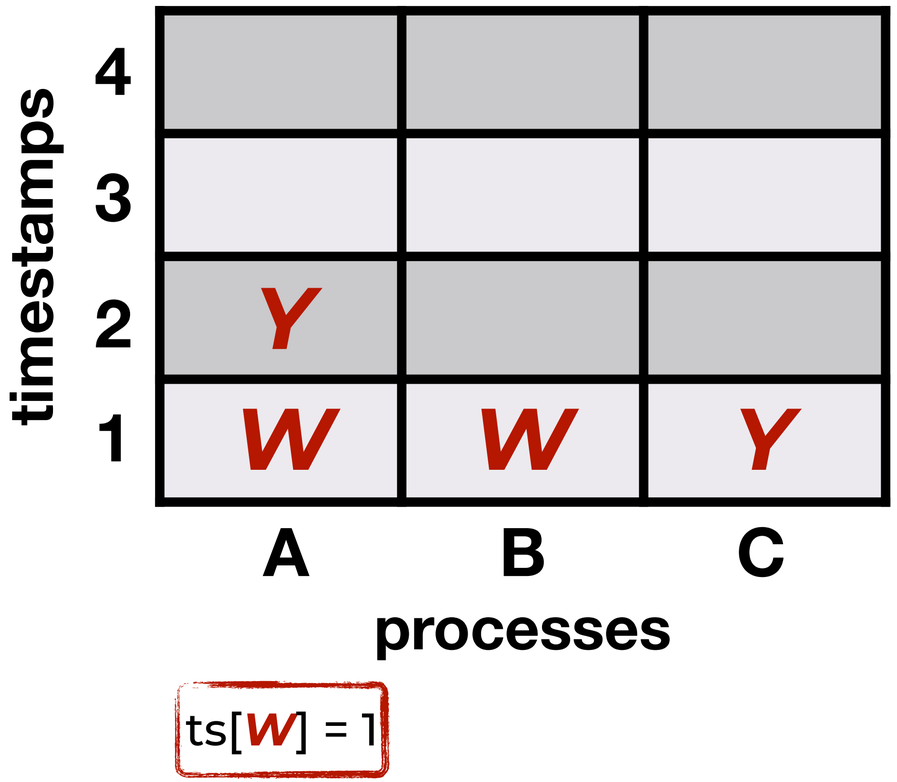
leaderless SMR with tempo

- a majority of processes proposes a timestamp for the command
- the command's timestamp is the highest proposal



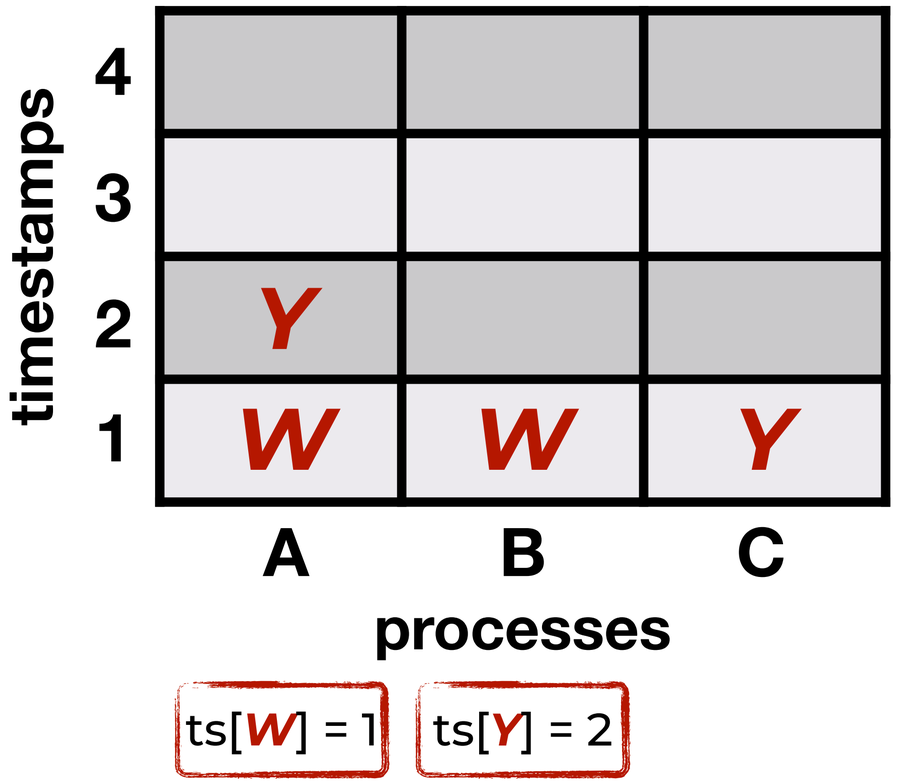
leaderless SMR with tempo

- a majority of processes proposes a timestamp for the command
- the command's timestamp is the highest proposal



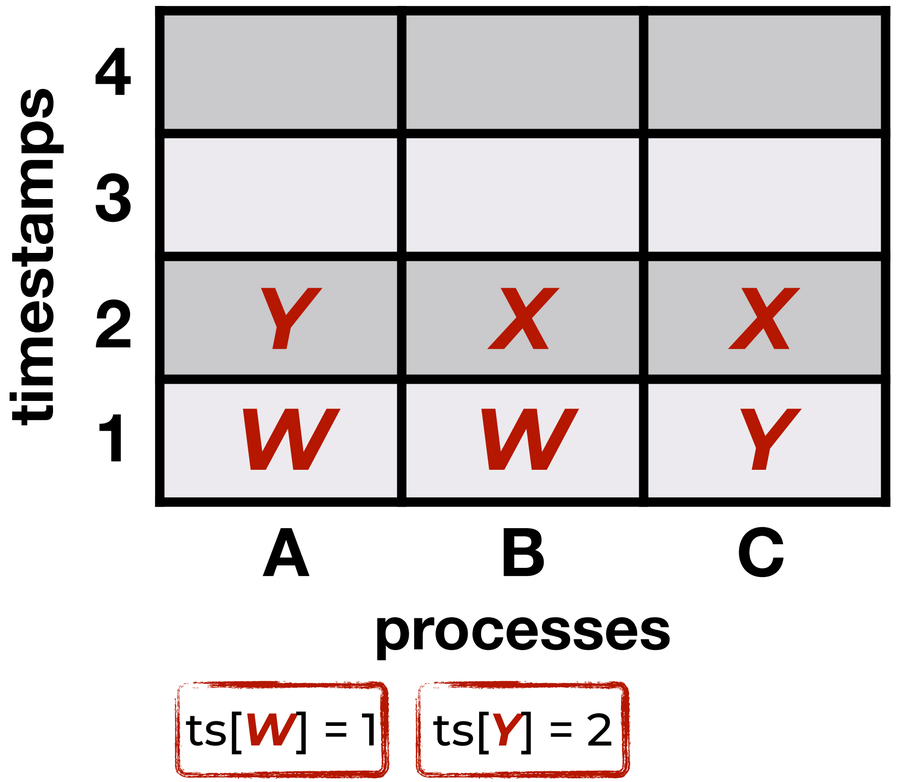
leaderless SMR with tempo

- a majority of processes proposes a timestamp for the command
- the command's timestamp is the highest proposal



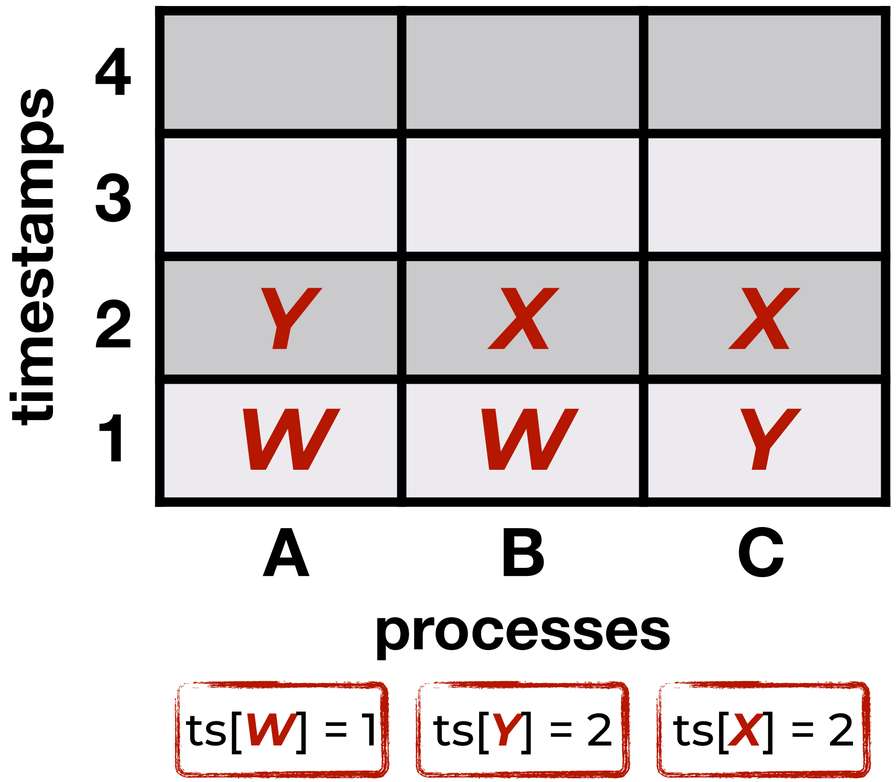
leaderless SMR with tempo

- a majority of processes proposes a timestamp for the command
- the command's timestamp is the highest proposal



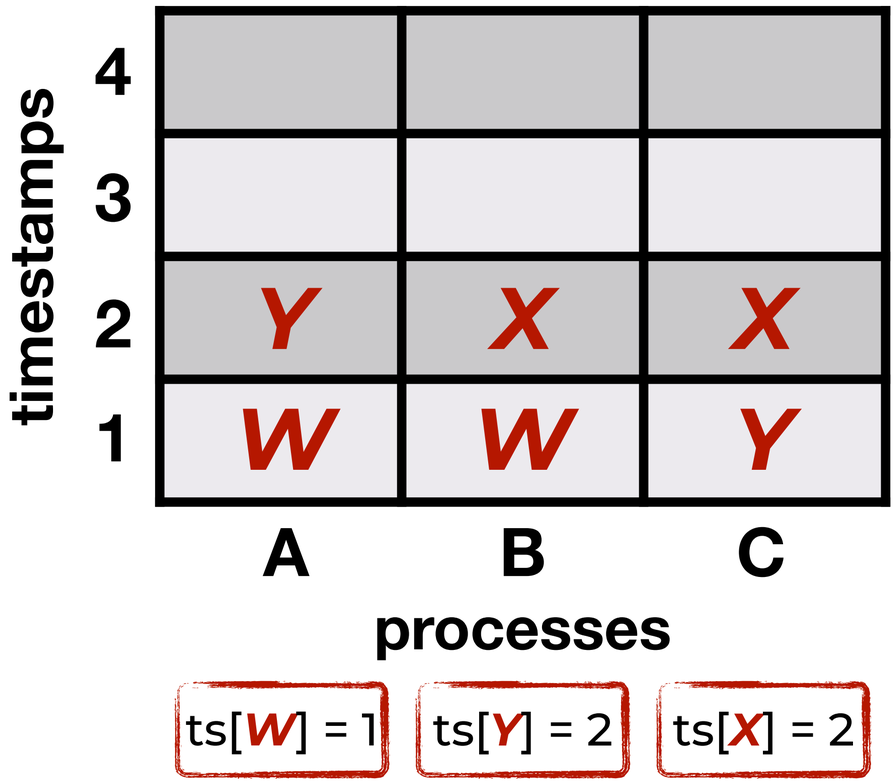
leaderless SMR with tempo

- a majority of processes proposes a timestamp for the command
- the command's timestamp is the highest proposal



leaderless SMR with tempo

- a majority of processes proposes a timestamp for the command
- the command's timestamp is the highest proposal



question: *when is it safe to execute a committed command?*

timestamp stability

process i can only execute command c after its timestamp t is stable, i.e., every command with a timestamp equal or lower to t is also committed at i

timestamp stability

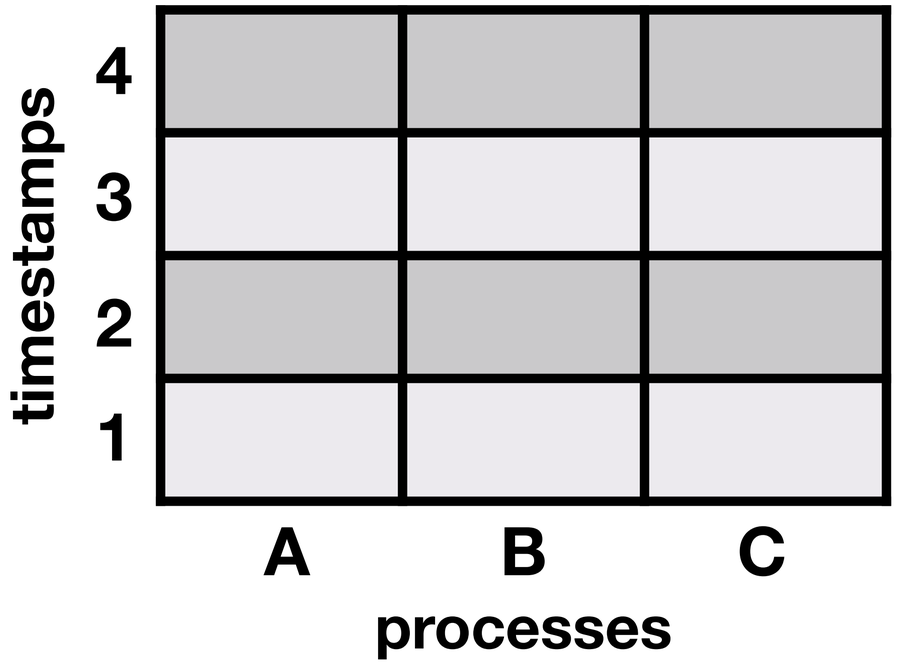
process i can only execute command c after its timestamp t is stable, i.e., every command with a timestamp equal or lower to t is also committed at i

theorem: *timestamp **t** is stable at process i once it knows all the proposals up to **t** by any majority*

timestamp stability

process i can only execute command c after its timestamp t is stable, i.e., every command with a timestamp equal or lower to t is also committed at i

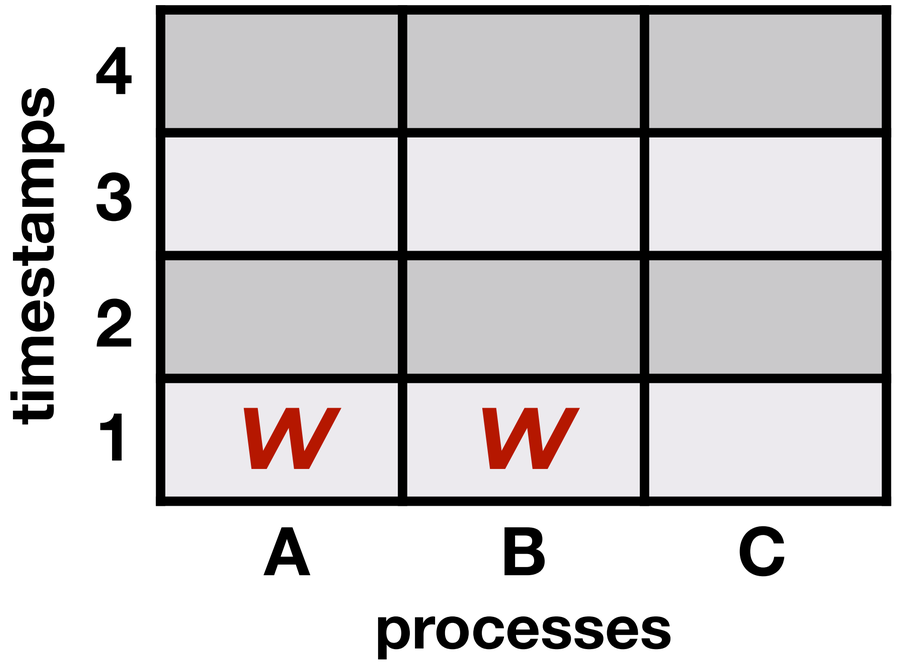
theorem: *timestamp **t** is stable at process i once it knows all the proposals up to **t** by any majority*



timestamp stability

process i can only execute command c after its timestamp t is stable, i.e., every command with a timestamp equal or lower to t is also committed at i

theorem: *timestamp **t** is stable at process i once it knows all the proposals up to **t** by any majority*

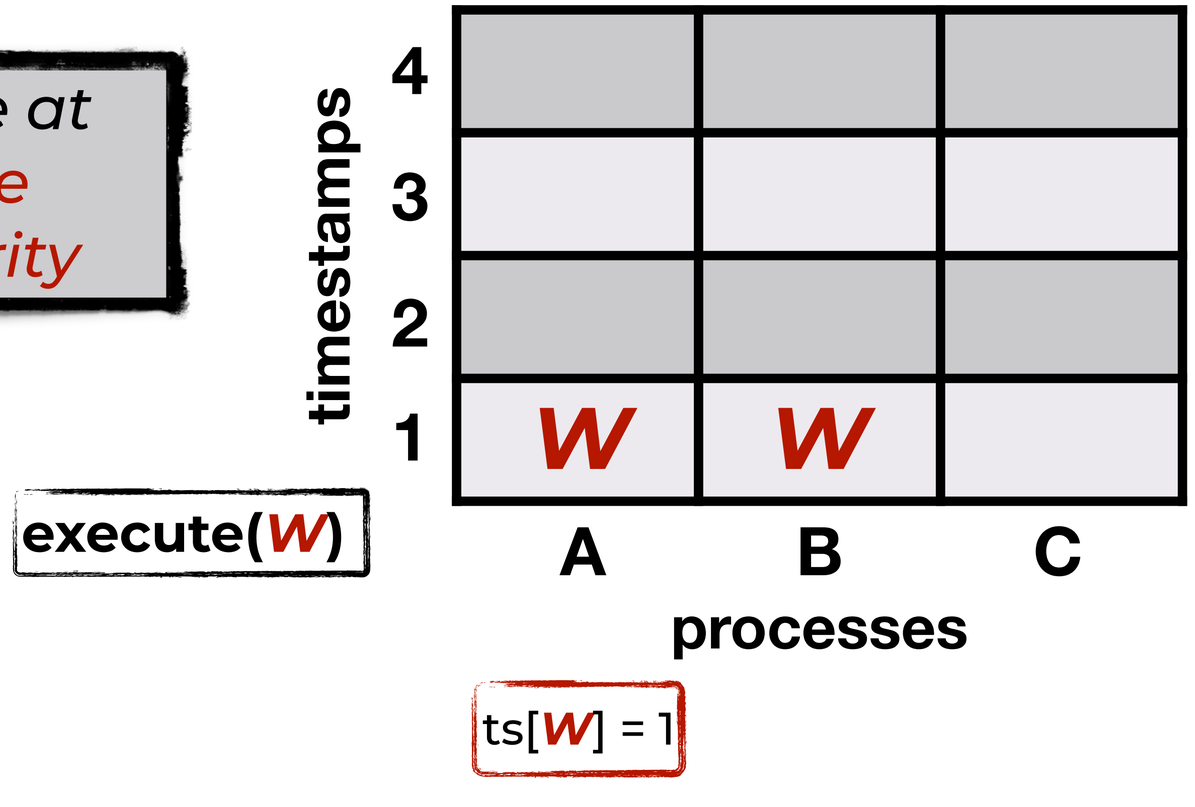


ts[*w*] = 1

timestamp stability

process i can only execute command c after its timestamp t is stable, i.e., every command with a timestamp equal or lower to t is also committed at i

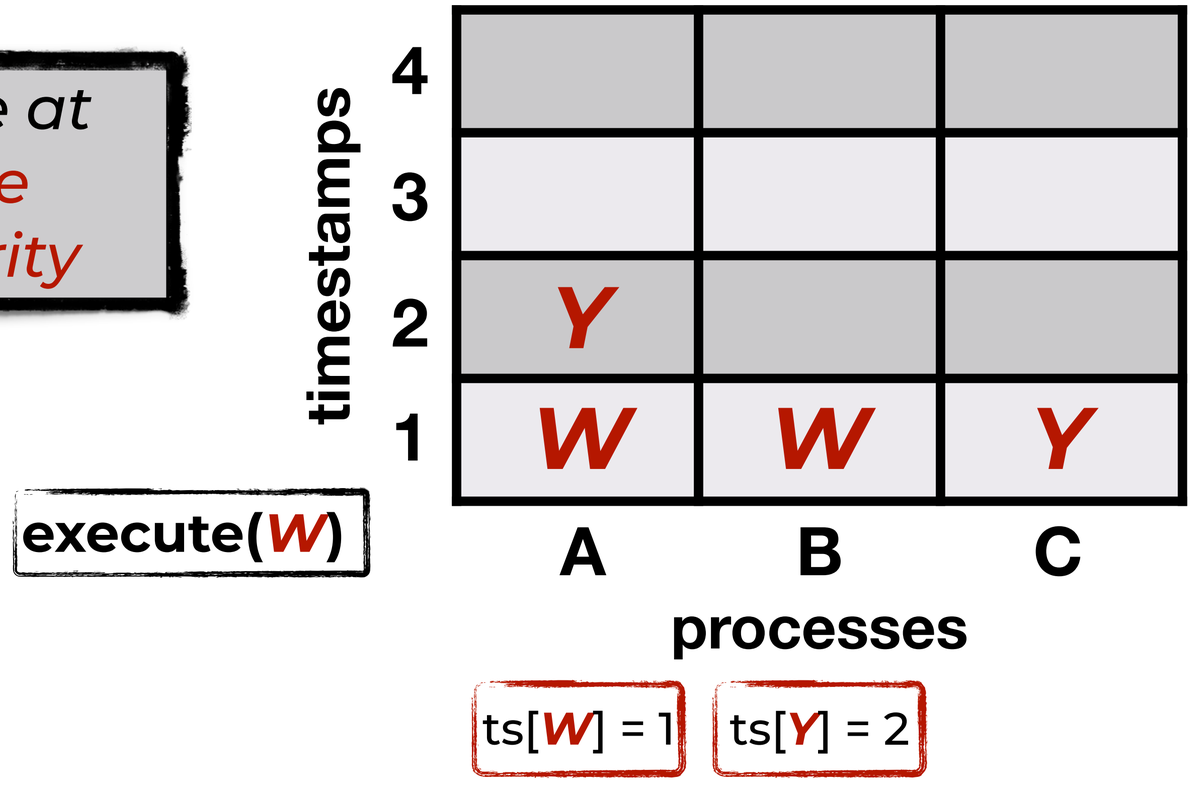
theorem: *timestamp **t** is stable at process i once it knows all the proposals up to **t** by any majority*



timestamp stability

process i can only execute command c after its timestamp t is stable, i.e., every command with a timestamp equal or lower to t is also committed at i

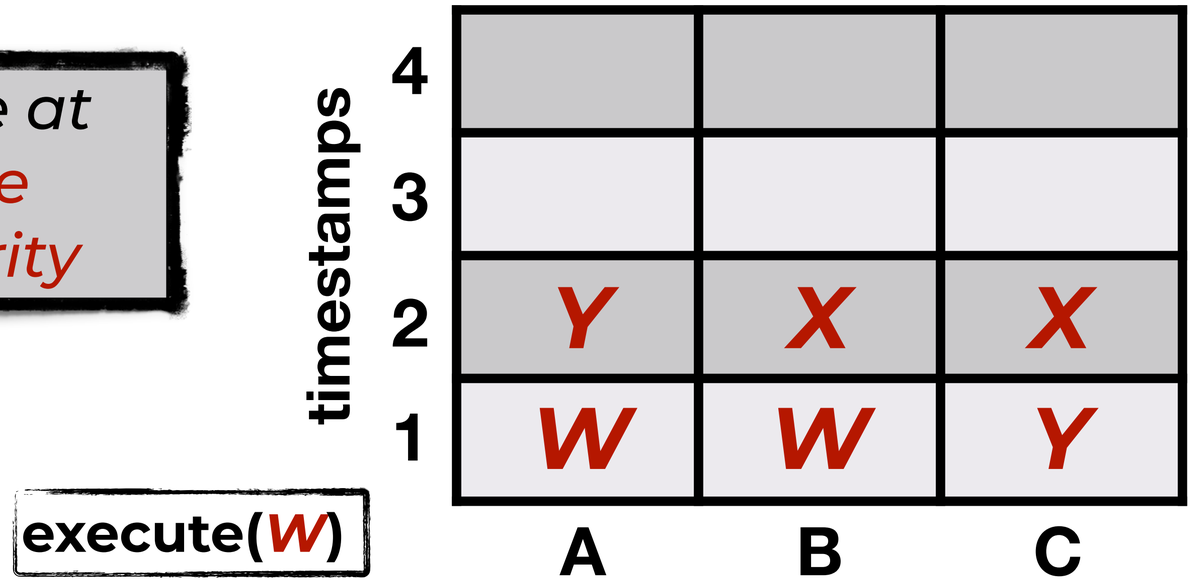
theorem: *timestamp **t** is stable at process i once it knows all the proposals up to **t** by any majority*



timestamp stability

process i can only execute command c after its timestamp t is stable, i.e., every command with a timestamp equal or lower to t is also committed at i

theorem: *timestamp **t** is stable at process i once it knows all the proposals up to **t** by any majority*



execute(*W*)

ts[*W*] = 1

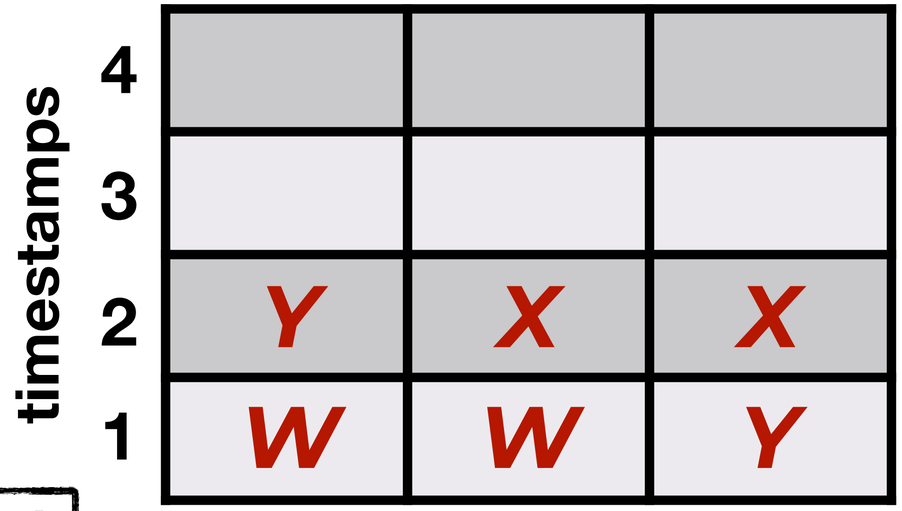
ts[*Y*] = 2

ts[*X*] = 2

timestamp stability

process i can only execute command c after its timestamp t is stable, i.e., every command with a timestamp equal or lower to t is also committed at i

theorem: *timestamp **t** is stable at process i once it knows all the proposals up to **t** by any majority*



execute(*W*)

execute(*X*) ; execute(*Y*)
since *X* < *Y*

ts[*W*] = 1

ts[*Y*] = 2

ts[*X*] = 2

timestamp stability vs explicit dependencies

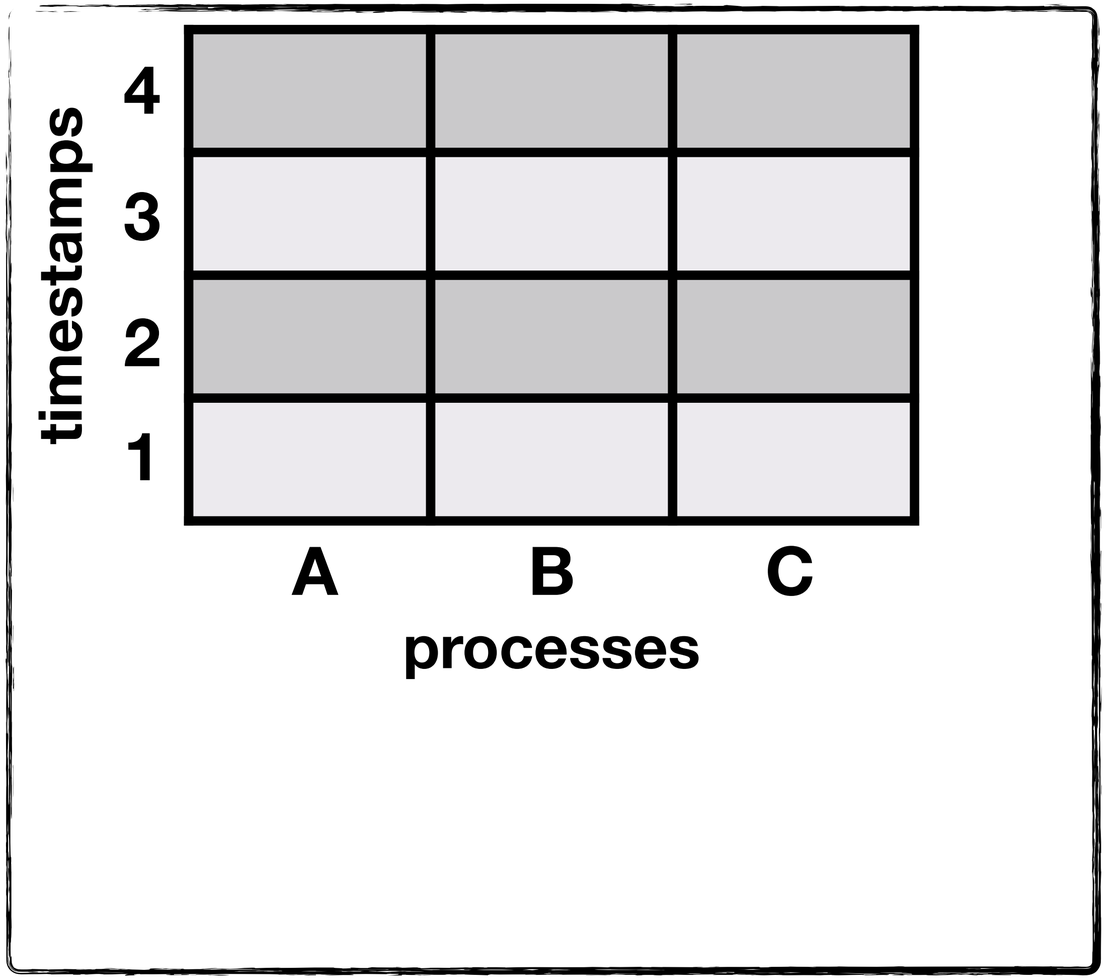
command arrival order:

- ***W X Z*** at **A**
- ***Y W*** at **B**
- ***Z Y*** at **C**

timestamp stability vs explicit dependencies

command arrival order:

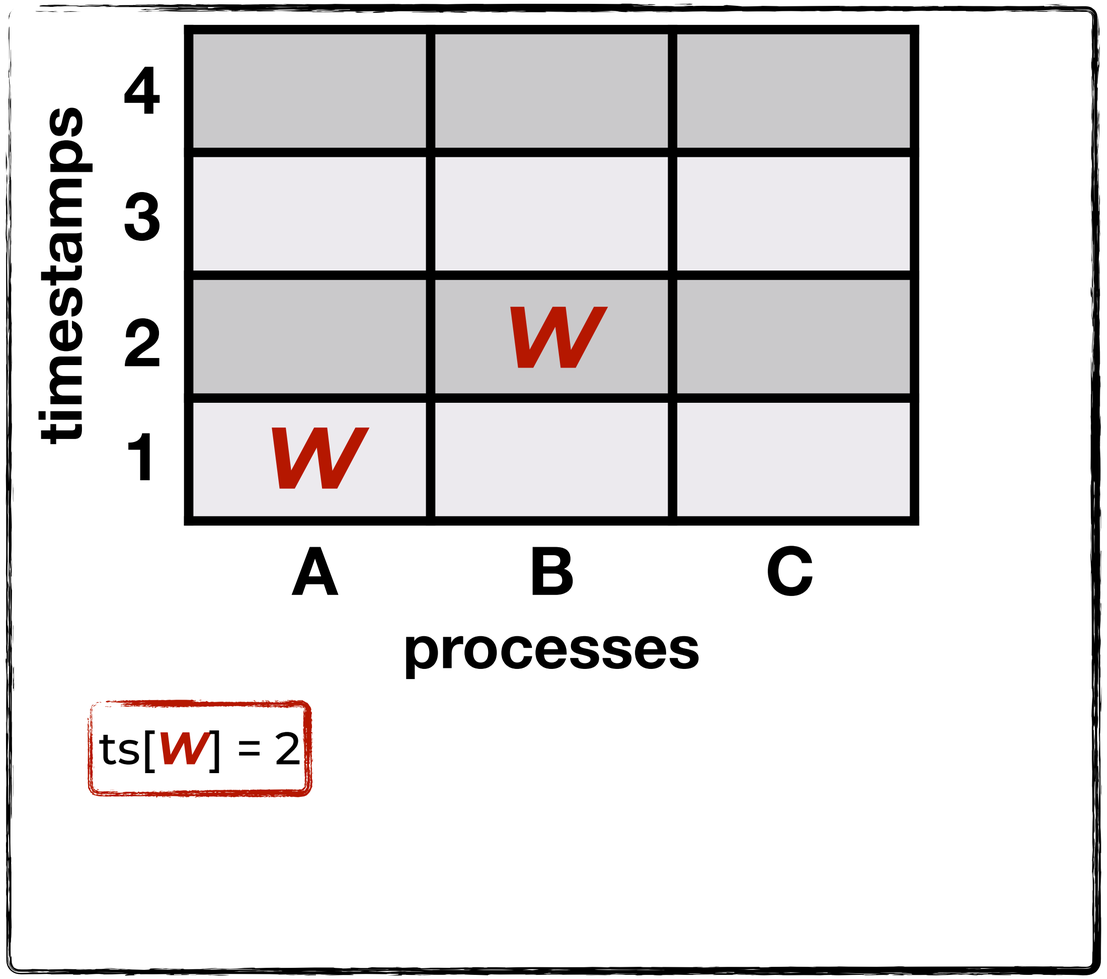
- ***W X Z*** at **A**
- ***Y W*** at **B**
- ***Z Y*** at **C**



timestamp stability vs explicit dependencies

command arrival order:

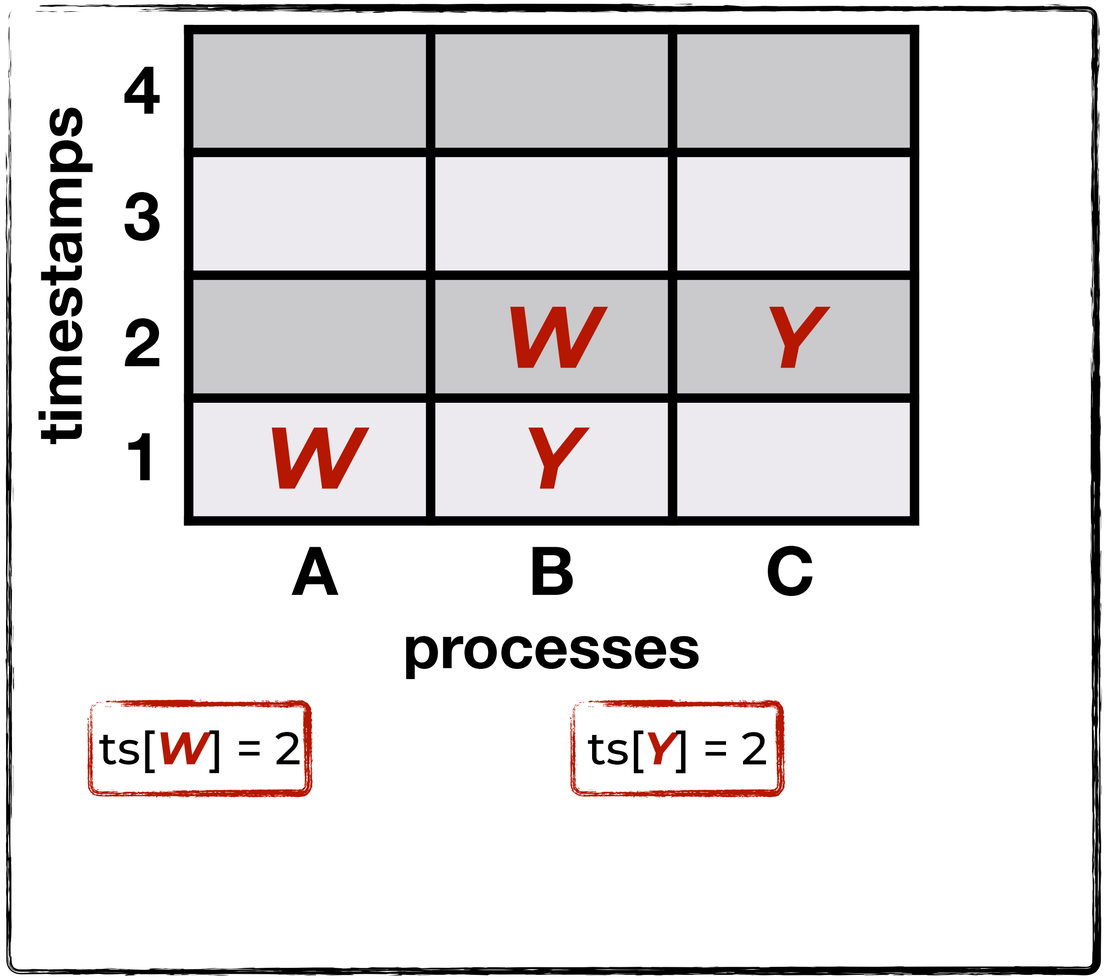
- ***W X Z*** at **A**
- ***Y W*** at **B**
- ***Z Y*** at **C**



timestamp stability vs explicit dependencies

command arrival order:

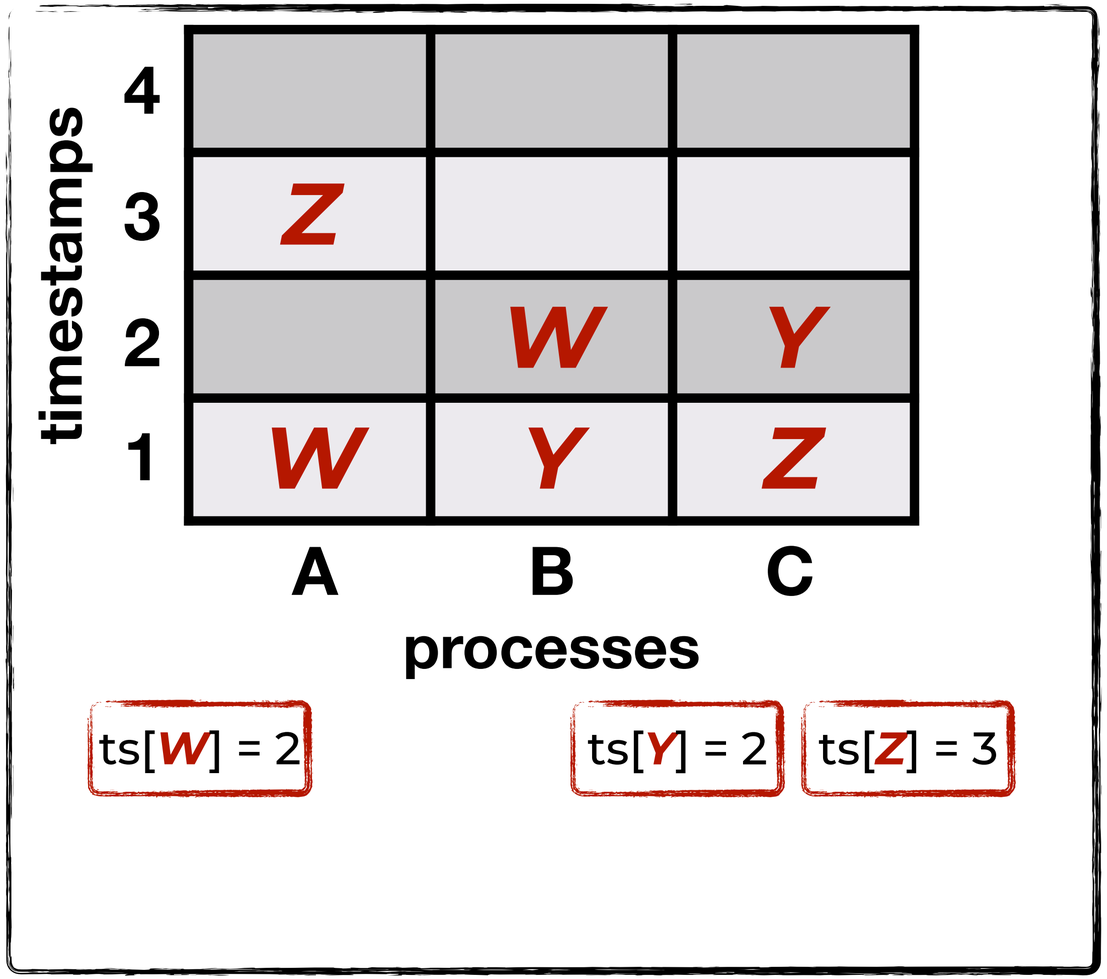
- ***W X Z*** at **A**
- ***Y W*** at **B**
- ***Z Y*** at **C**



timestamp stability vs explicit dependencies

command arrival order:

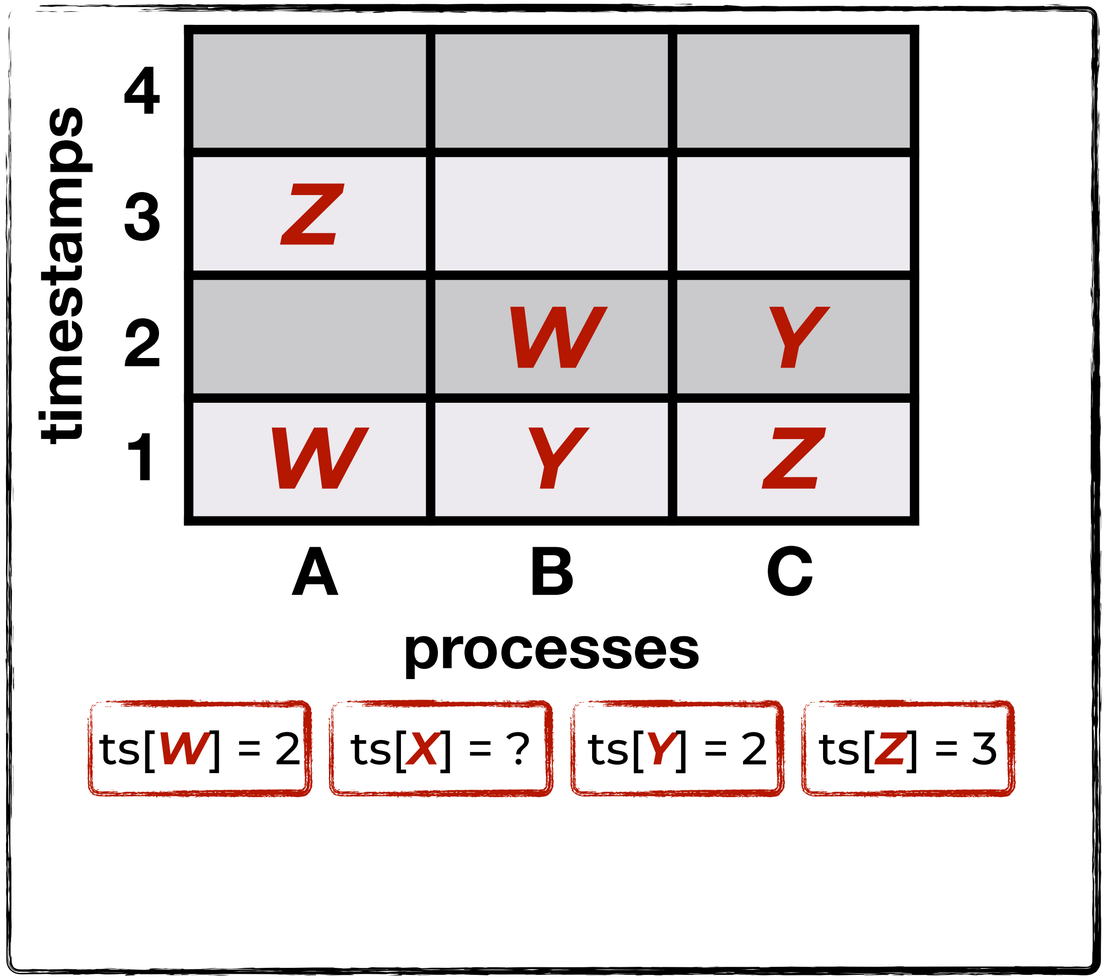
- ***W X Z*** at **A**
- ***Y W*** at **B**
- ***Z Y*** at **C**



timestamp stability vs explicit dependencies

command arrival order:

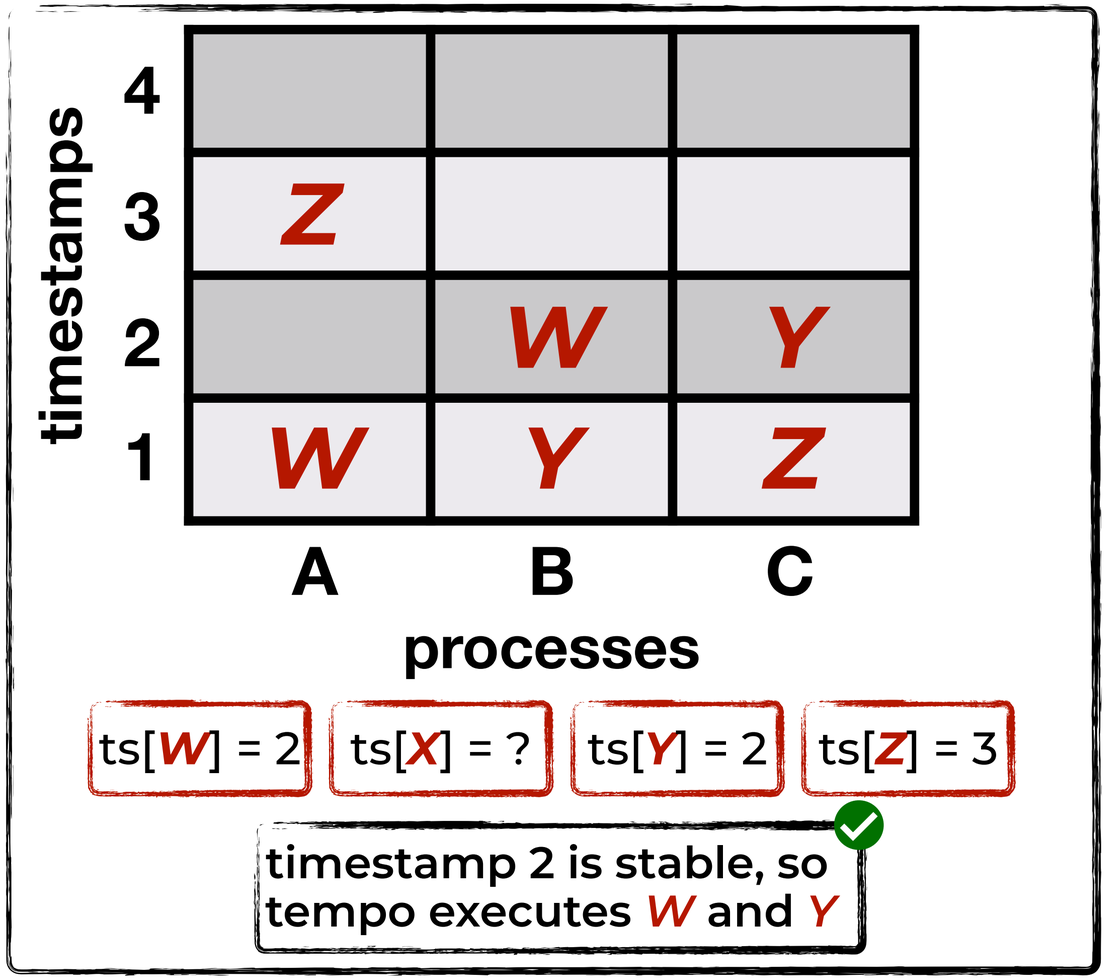
- ***W X Z*** at **A**
- ***Y W*** at **B**
- ***Z Y*** at **C**



timestamp stability vs explicit dependencies

command arrival order:

- ***W X Z*** at **A**
- ***Y W*** at **B**
- ***Z Y*** at **C**

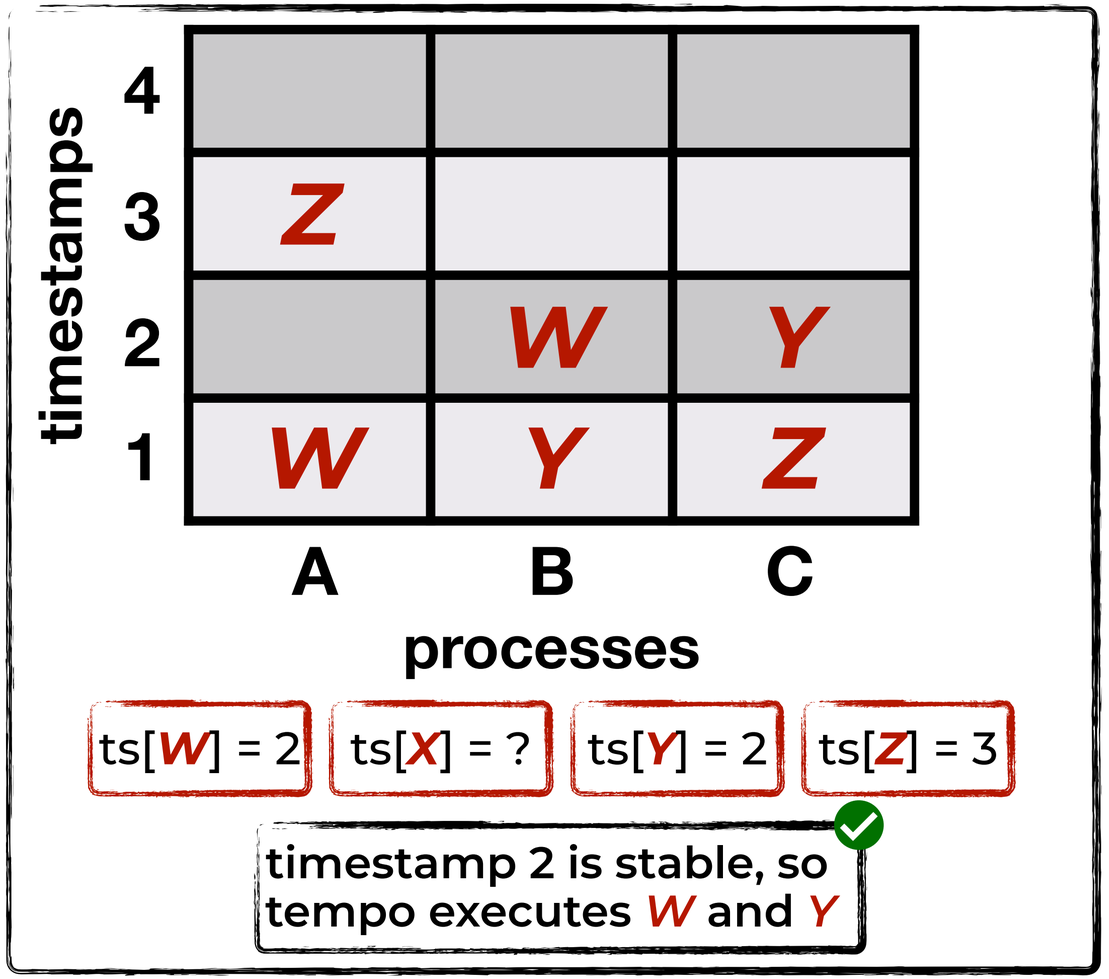


timestamp stability vs explicit dependencies

epaxos / atlas:

command arrival order:

- *W X Z* at **A**
- *Y W* at **B**
- *Z Y* at **C**



timestamp stability vs explicit dependencies

epaxos / atlas:

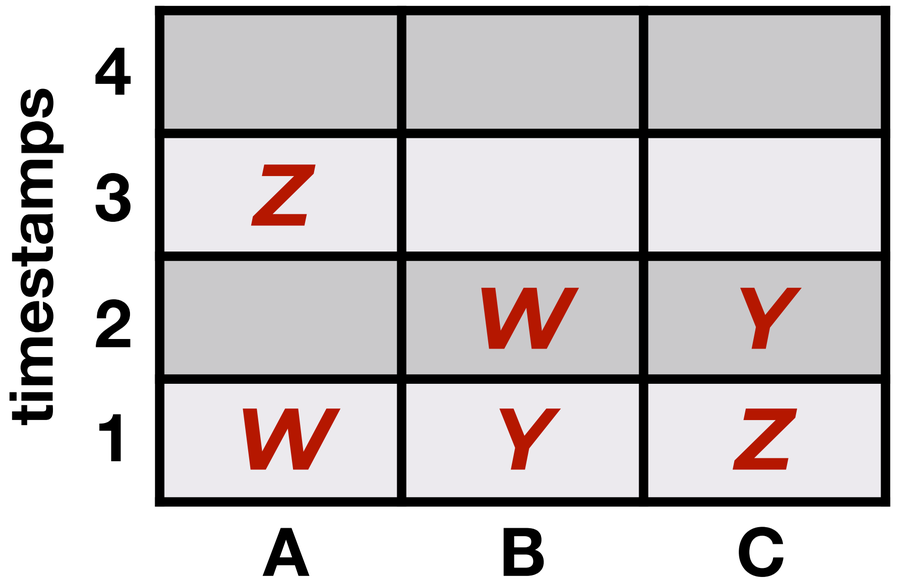
W → *Y*

→ *"depends on"*

dep[*W*] = {*Y*}

command arrival order:

- *W X Z* at **A**
- *Y W* at **B**
- *Z Y* at **C**



ts[*W*] = 2 | ts[*X*] = ? | ts[*Y*] = 2 | ts[*Z*] = 3

timestamp 2 is stable, so tempo executes *W* and *Y*

timestamp stability vs explicit dependencies

epaxos / atlas:

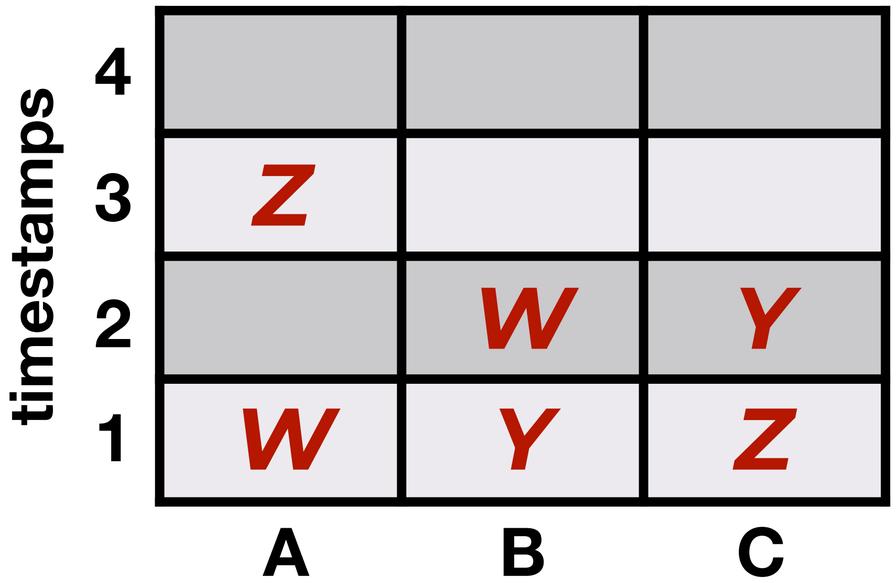
W → *Y* → *Z*

→ *"depends on"*

dep[*W*] = {*Y*} dep[*Y*] = {*Z*}

command arrival order:

- *W X Z* at **A**
- *Y W* at **B**
- *Z Y* at **C**



ts[*W*] = 2 ts[*X*] = ? ts[*Y*] = 2 ts[*Z*] = 3

timestamp 2 is stable, so tempo executes *W* and *Y*

timestamp stability vs explicit dependencies

epaxos / atlas:

→ *"depends on"*

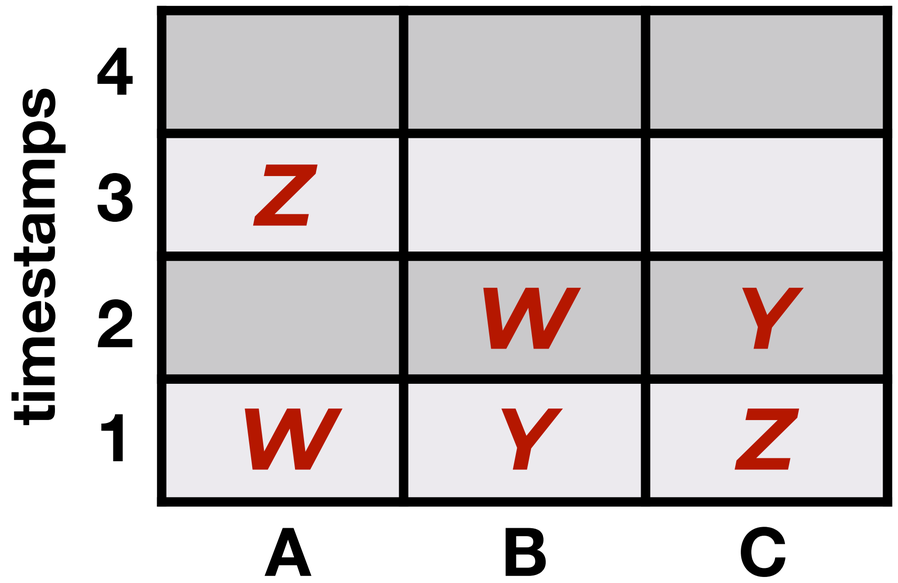
dep[*W*] = {*Y*}

dep[*Y*] = {*Z*}

dep[*Z*] = {*W*, *X*}

command arrival order:

- *W X Z* at **A**
- *Y W* at **B**
- *Z Y* at **C**



ts[*W*] = 2

ts[*X*] = ?

ts[*Y*] = 2

ts[*Z*] = 3

timestamp 2 is stable, so tempo executes *W* and *Y*

timestamp stability vs explicit dependencies

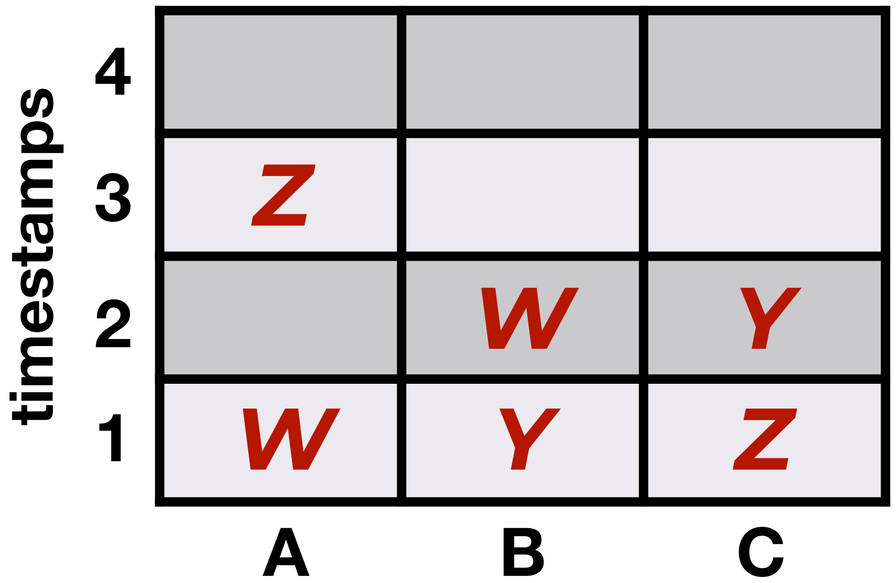
epaxos / atlas:

dep[*W*] = {*Y*} | dep[*Y*] = {*Z*}

dep[*X*] = ? | dep[*Z*] = {*W*, *X*}

command arrival order:

- *W X Z* at **A**
- *Y W* at **B**
- *Z Y* at **C**



ts[*W*] = 2 | ts[*X*] = ? | ts[*Y*] = 2 | ts[*Z*] = 3

timestamp 2 is stable, so tempo executes *W* and *Y*

timestamp stability vs explicit dependencies

epaxos / atlas:

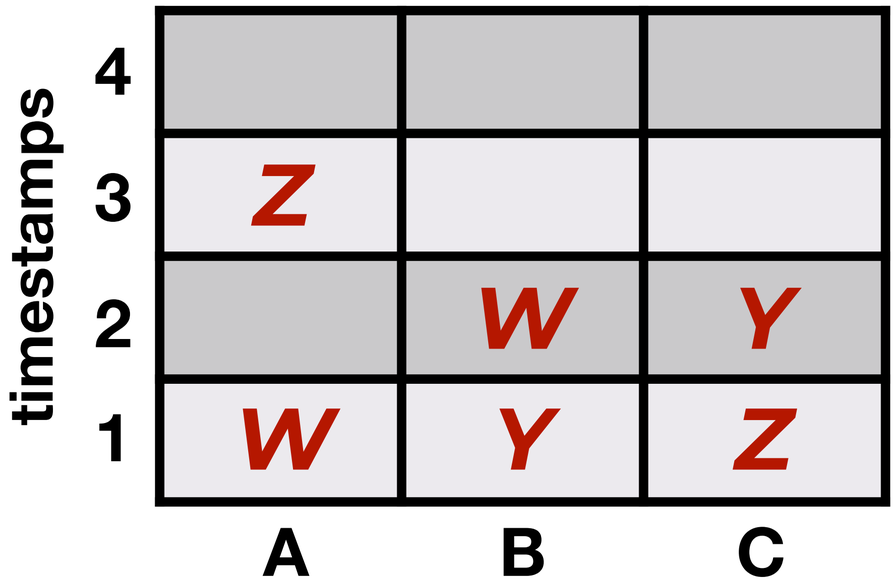
dep[*W*] = {*Y*} | dep[*Y*] = {*Z*}

dep[*X*] = ? | dep[*Z*] = {*W*, *X*}

no command is executed!

command arrival order:

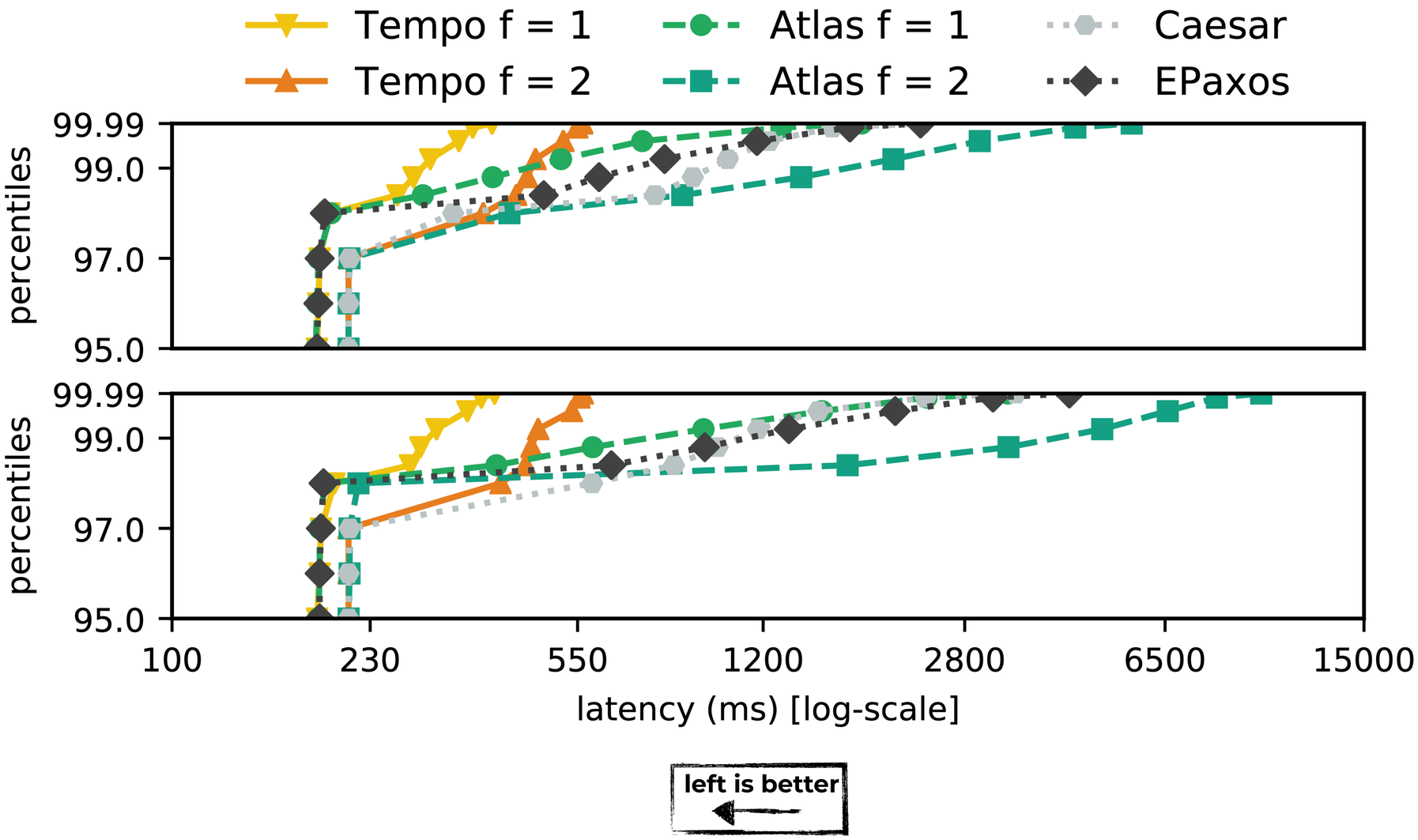
- *W X Z* at **A**
- *Y W* at **B**
- *Z Y* at **C**



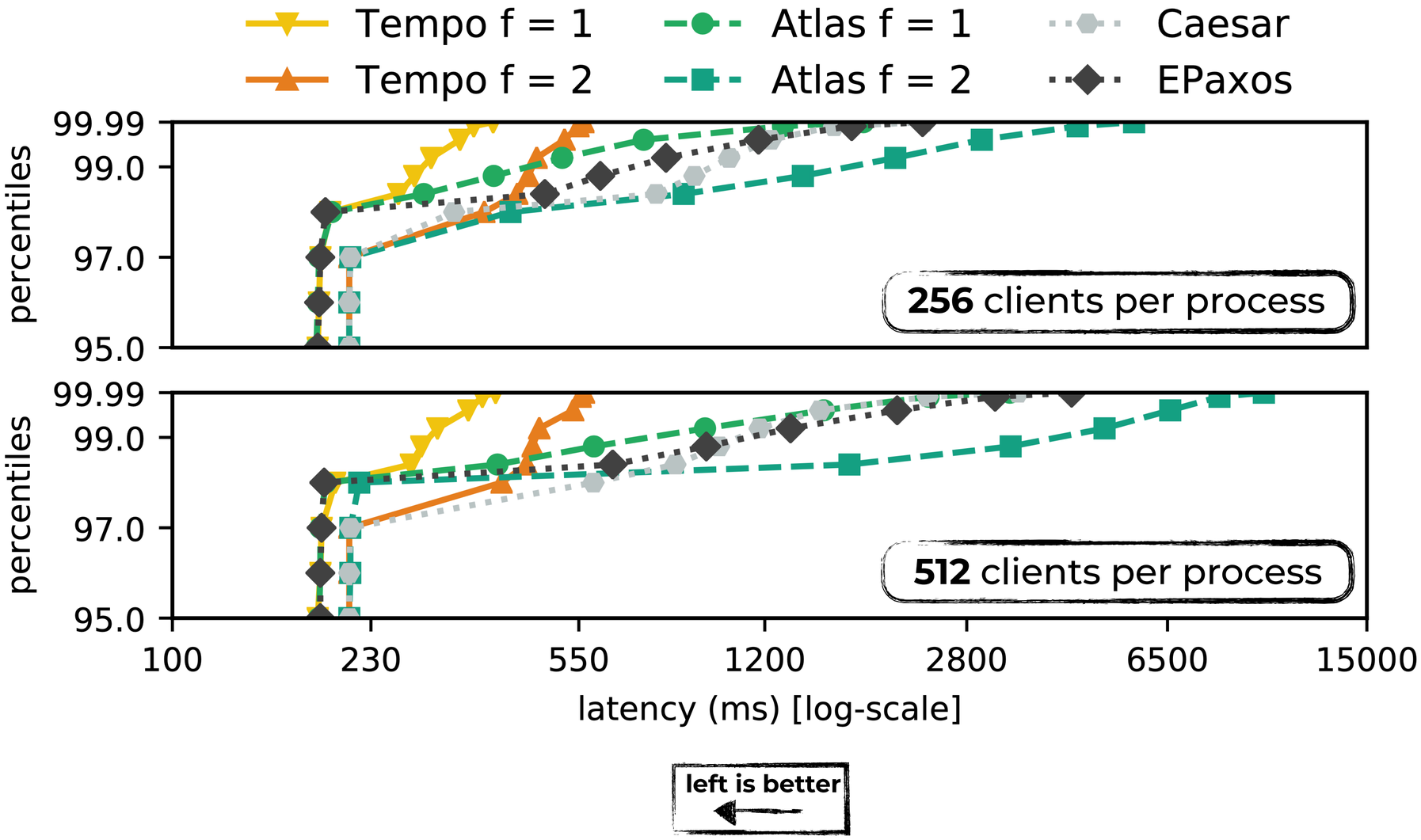
ts[*W*] = 2 | ts[*X*] = ? | ts[*Y*] = 2 | ts[*Z*] = 3

timestamp 2 is stable, so tempo executes *W* and *Y*

tempo provides predictable latency

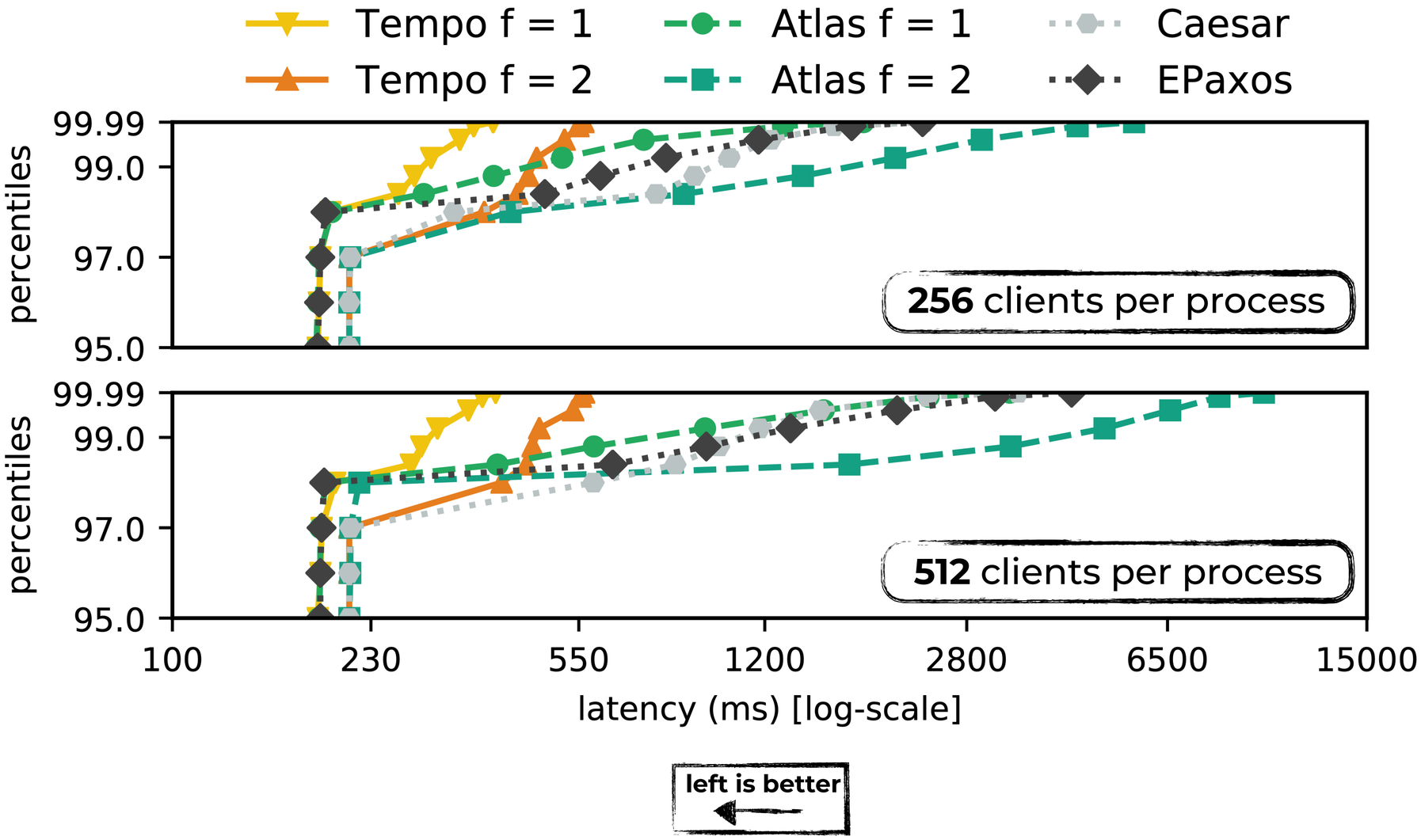


tempo provides predictable latency



tempo provides predictable latency

	99.9th	
	256	512
atlas f=1	1.3s	2.4s
epaxos	1.7s	3.1s
caesar	1.6s	2.4s
tempo f=1	354ms	367ms

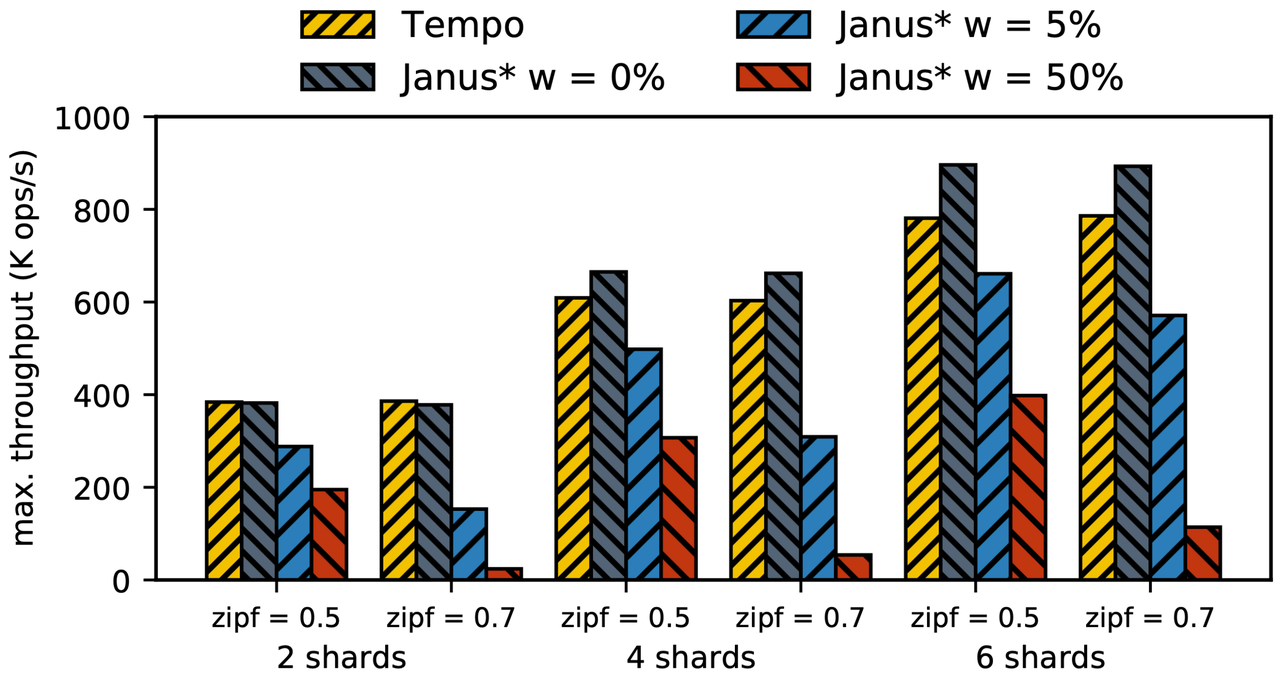


more in the paper

- **simple** generalization to **partial replication**

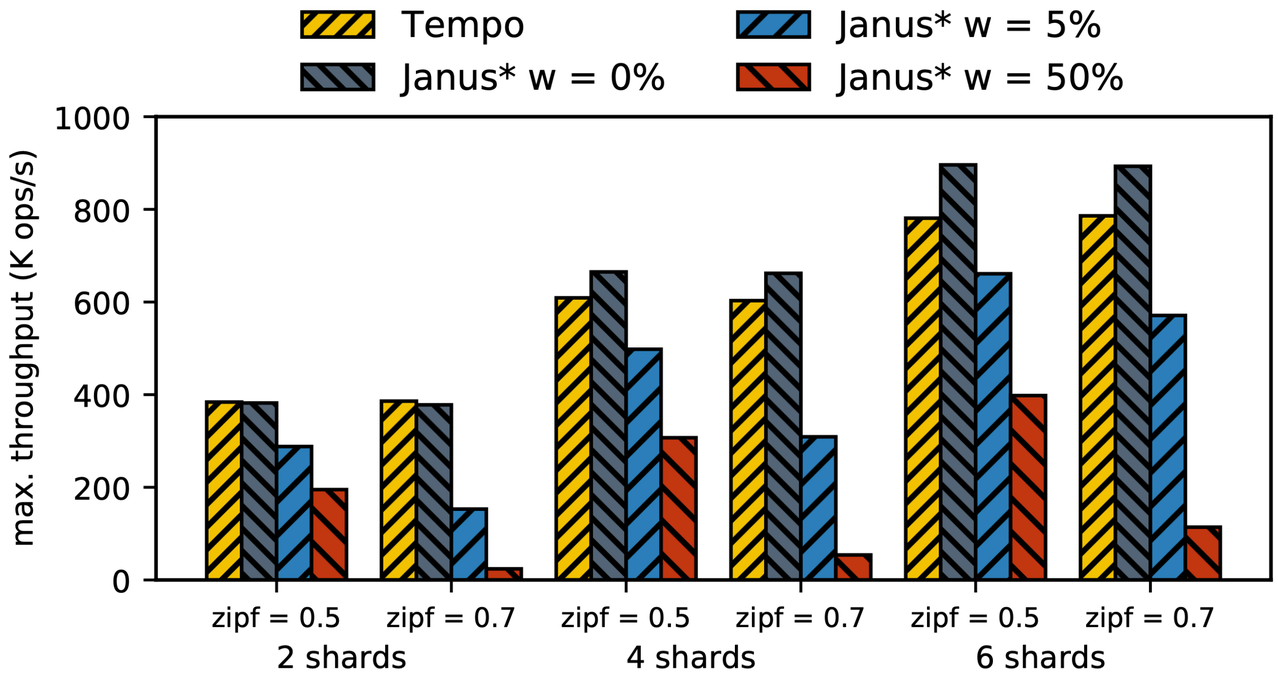
more in the paper

- **simple** generalization to **partial replication**



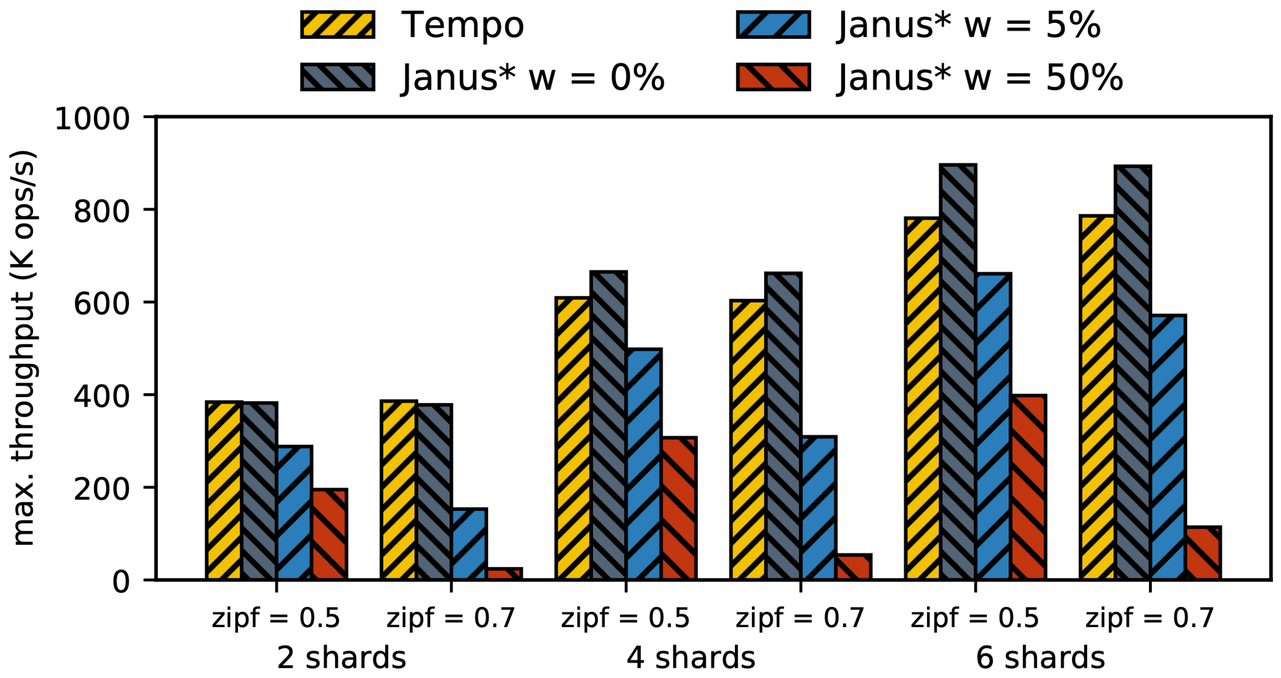
more in the paper

- **simple** generalization to **partial replication**
- **permissive fast-path** condition
- **simple recovery** mechanism



more in the paper

- **simple** generalization to **partial replication**
- **permissive fast-path** condition
- **simple recovery** mechanism



evaluation framework
github.com/vitorenesduarte/fantoch

summary

- tempo guarantees **progress under a synchronous network** without the need to contact all replicas

summary

- tempo guarantees **progress under a synchronous network** without the need to contact all replicas

- tempo provides **predictable performance** even in contended workloads

summary

- tempo guarantees **progress under a synchronous network** without the need to contact all replicas
- tempo provides **predictable performance** even in contended workloads
- tempo handles both **full** and **partial replication** scenarios
  - timestamping and stability detection are **fully decentralized**

efficient replication via timestamp stability

Vitor Enes, Carlos Baquero, Alexey Gotsman, Pierre Sutra

27 Apr. 2021 @ EuroSys'21

vitorenes.org

@vitorenesduarte