CS 5450

State Machine Replication

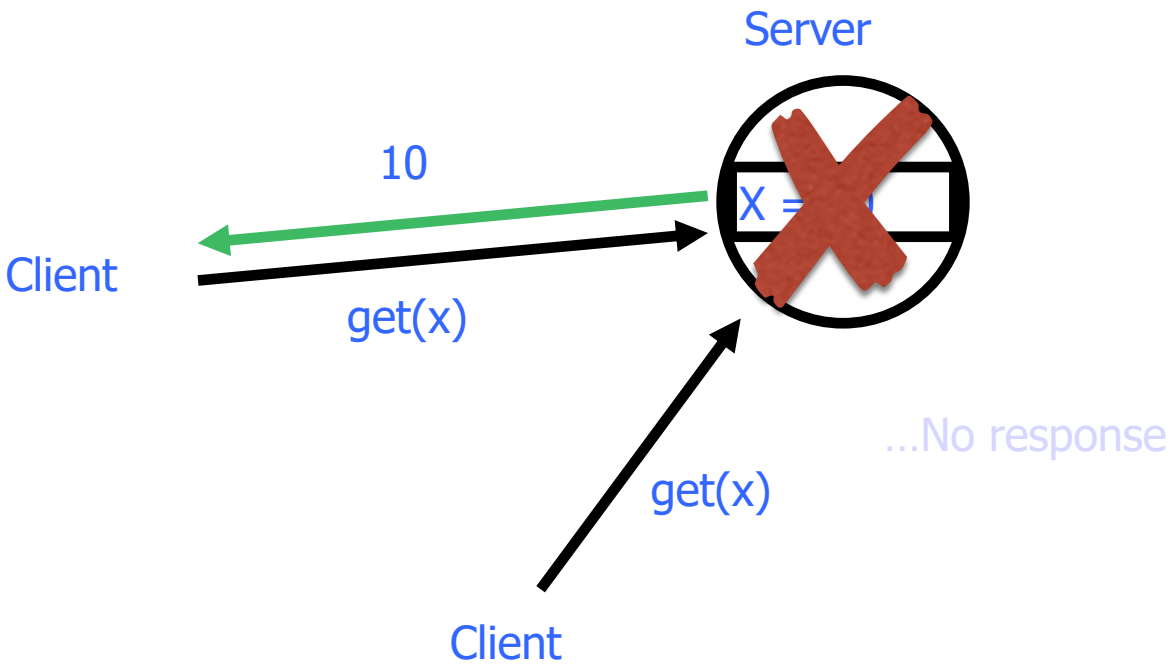
Key Ideas

To tolerate faults…

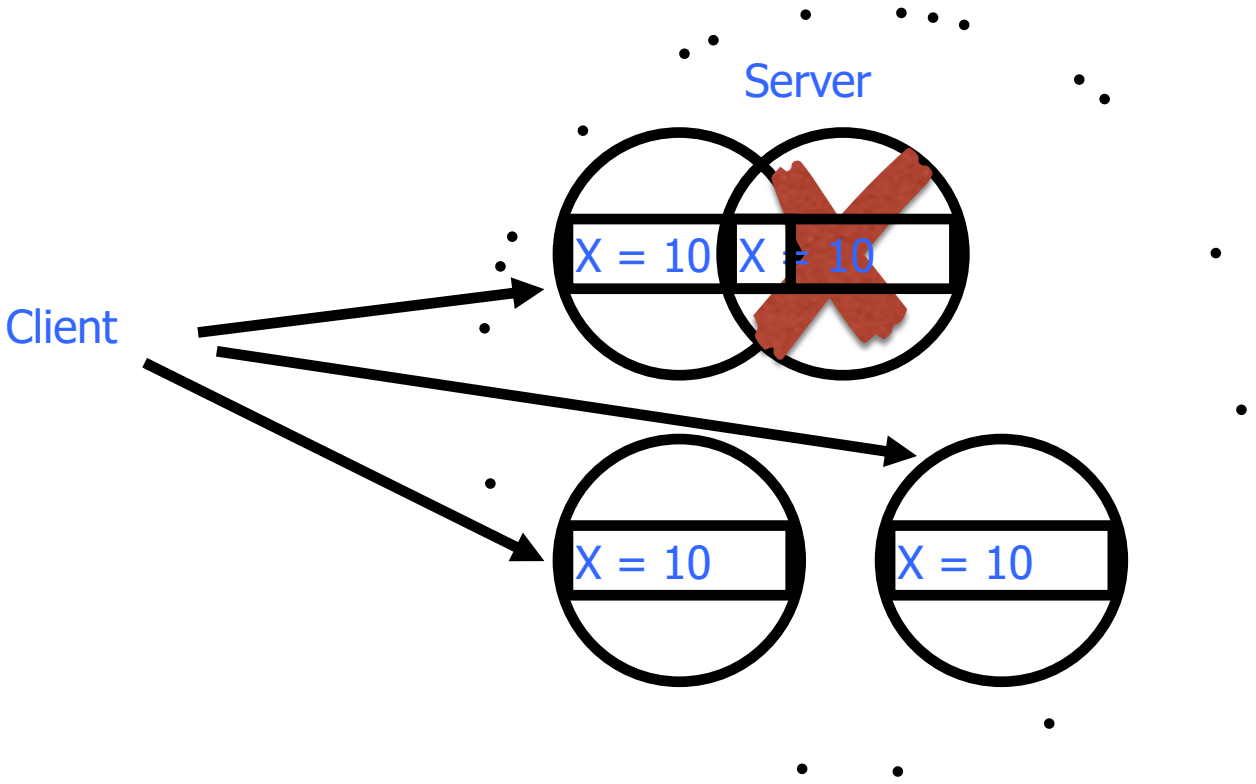
… replicate functionality!

- Can represent deterministic distributed system as replicated state machine (SMR)
- Each replica reaches the same conclusion about the system independently
- Examples of distributed algorithms that generically implement SMR
- Formal notion of fault-tolerance in SMR

Motivation



Motivation

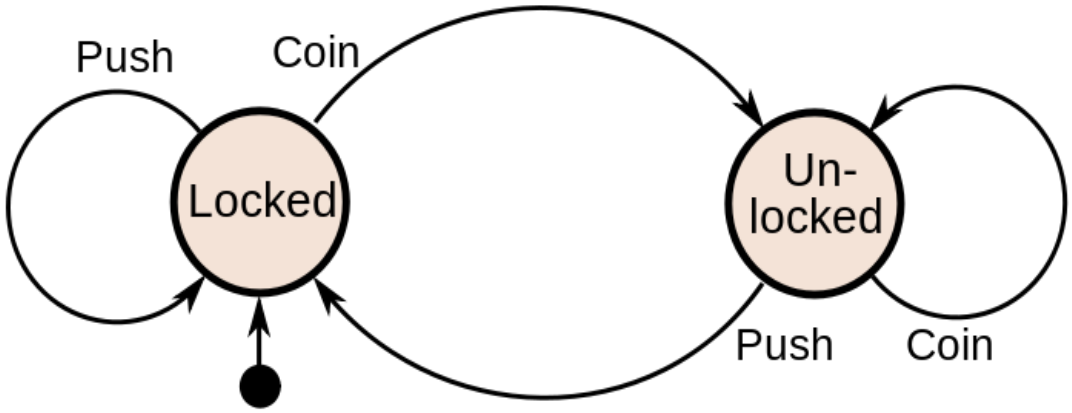


Motivation

- Need replication for fault tolerance
  - Without replication, what happens to storage if disk fails? To a web service if network fals?
- Reason about failure tolerance
  - How badly can things go wrong and the system would continue to function?

State Machines

- State variables
- Deterministic commands



Requests and Causality

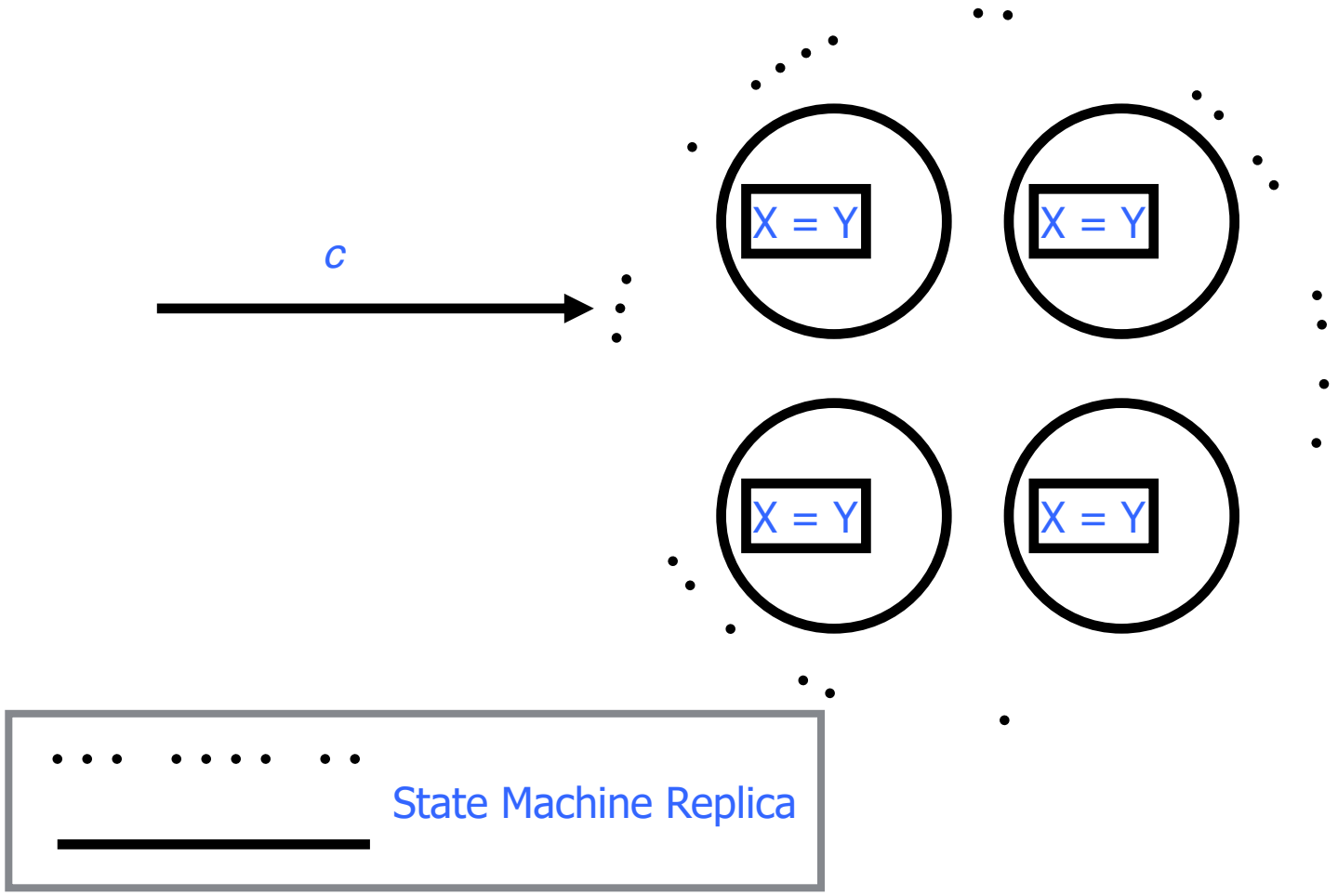
Process order consistent with potential causality

- Client A sends r1, then sends r2
  - r1 is processed before r2
- r1 causes Client B to send r3
  - r1 is processed before r3

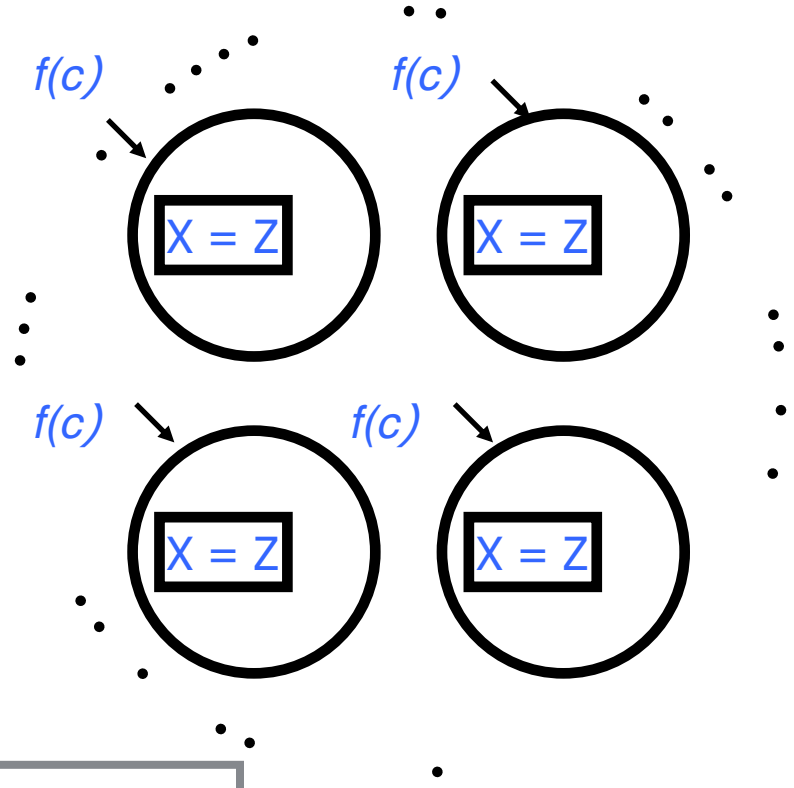
Coding State Machines

- State machines are procedures
  - Client calls procedure
- Avoid loops
- More flexible structure

State Machine Replication

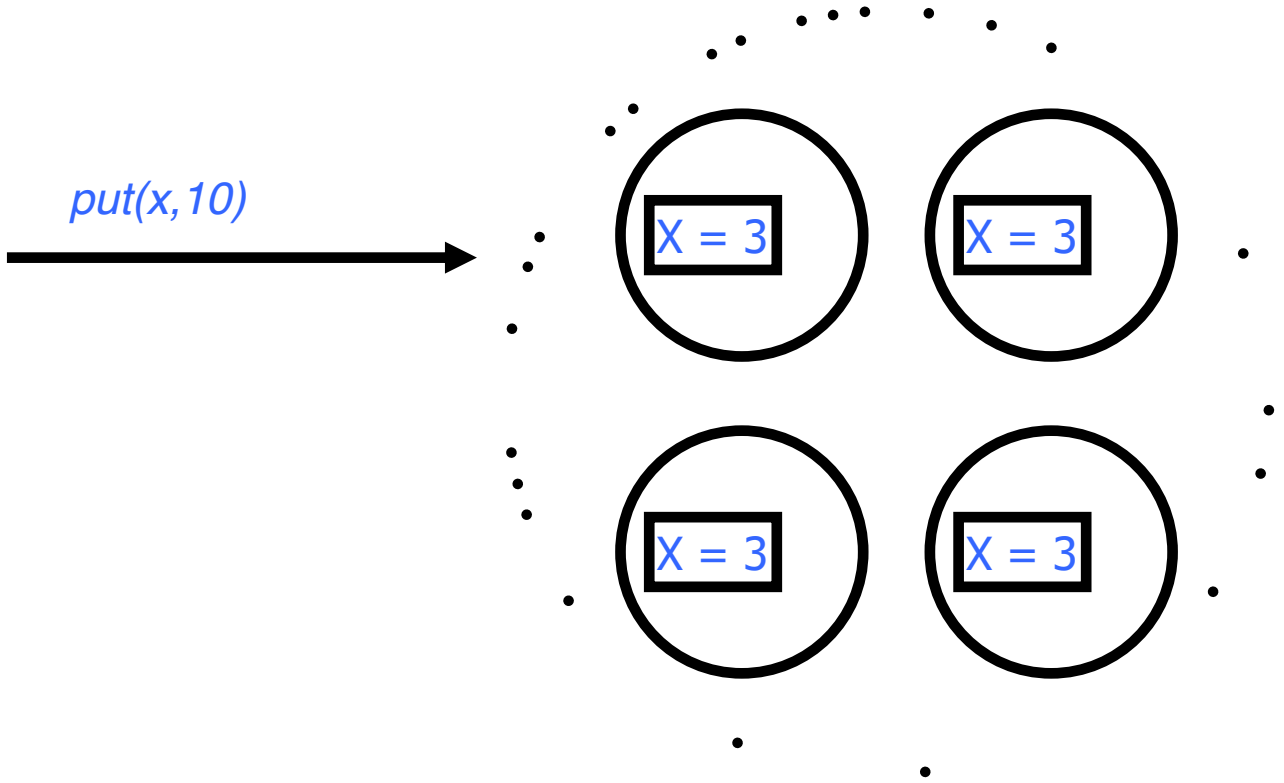


State Machine Replication

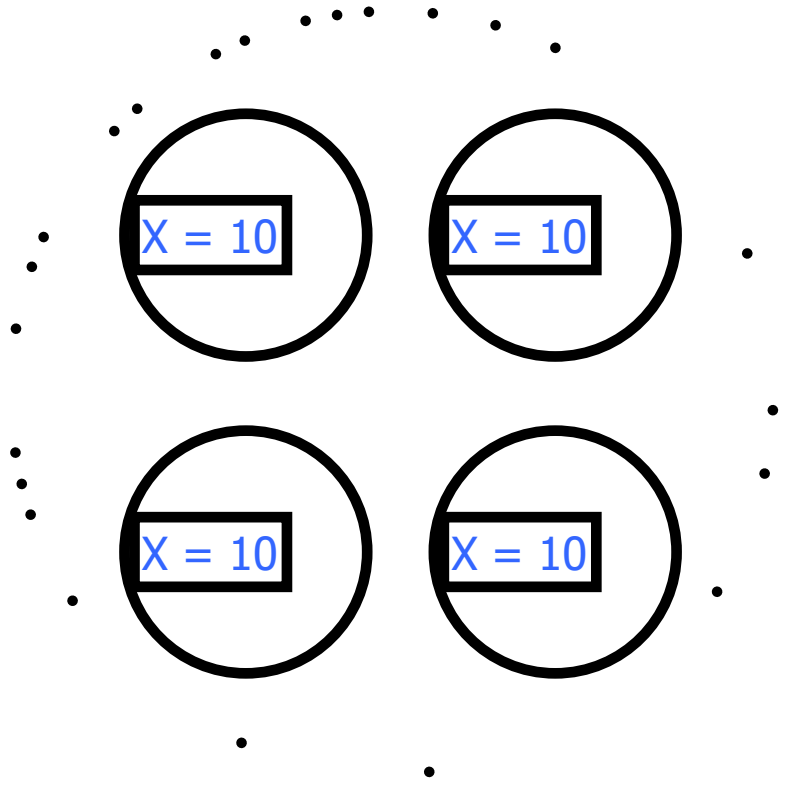


State Machine Replica

Write

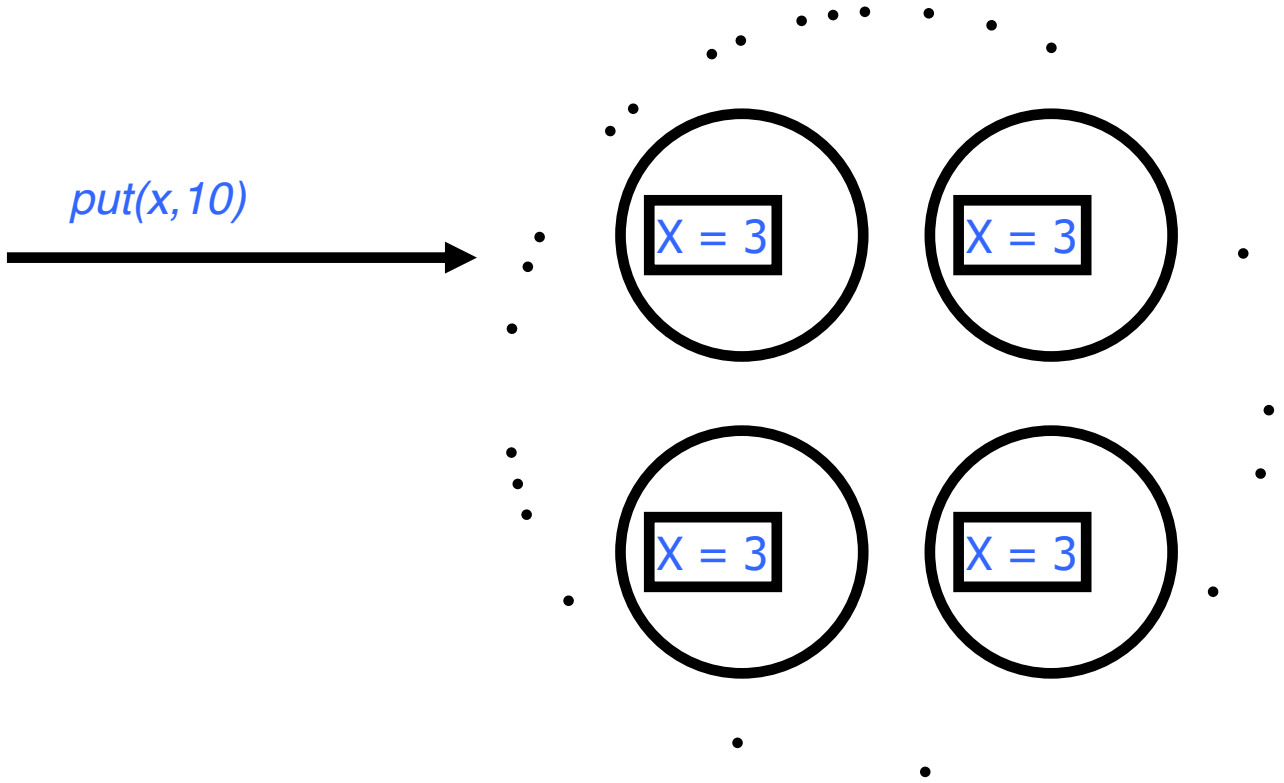


After the Write

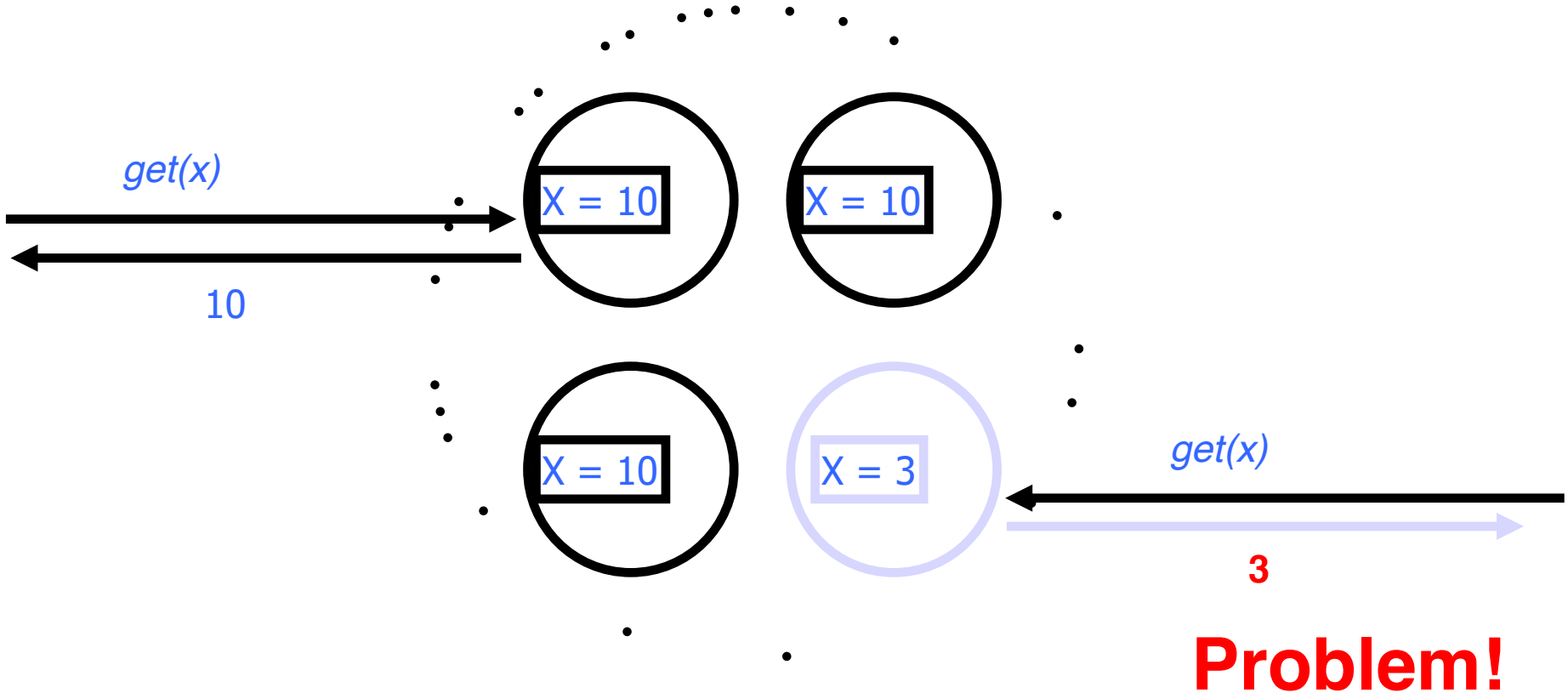


Great!

Write

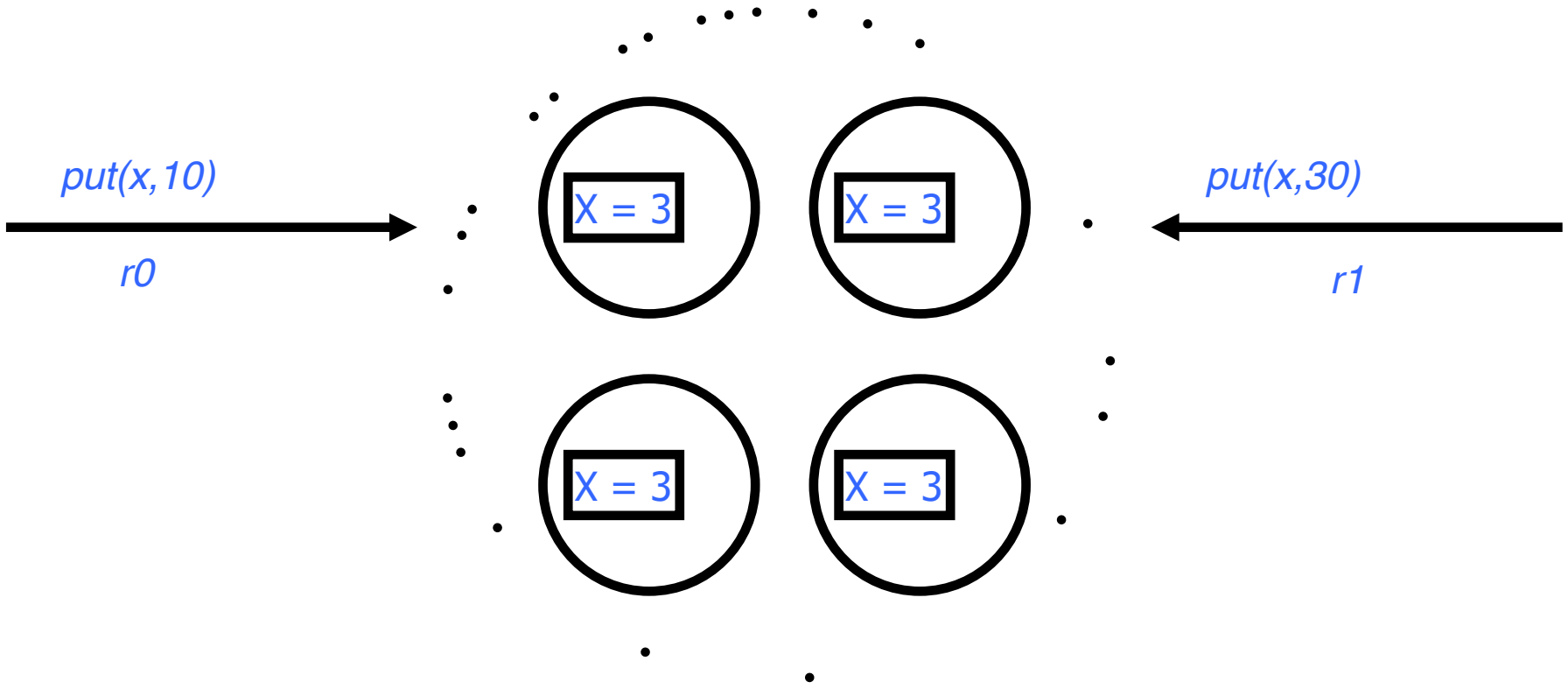


Need Agreement

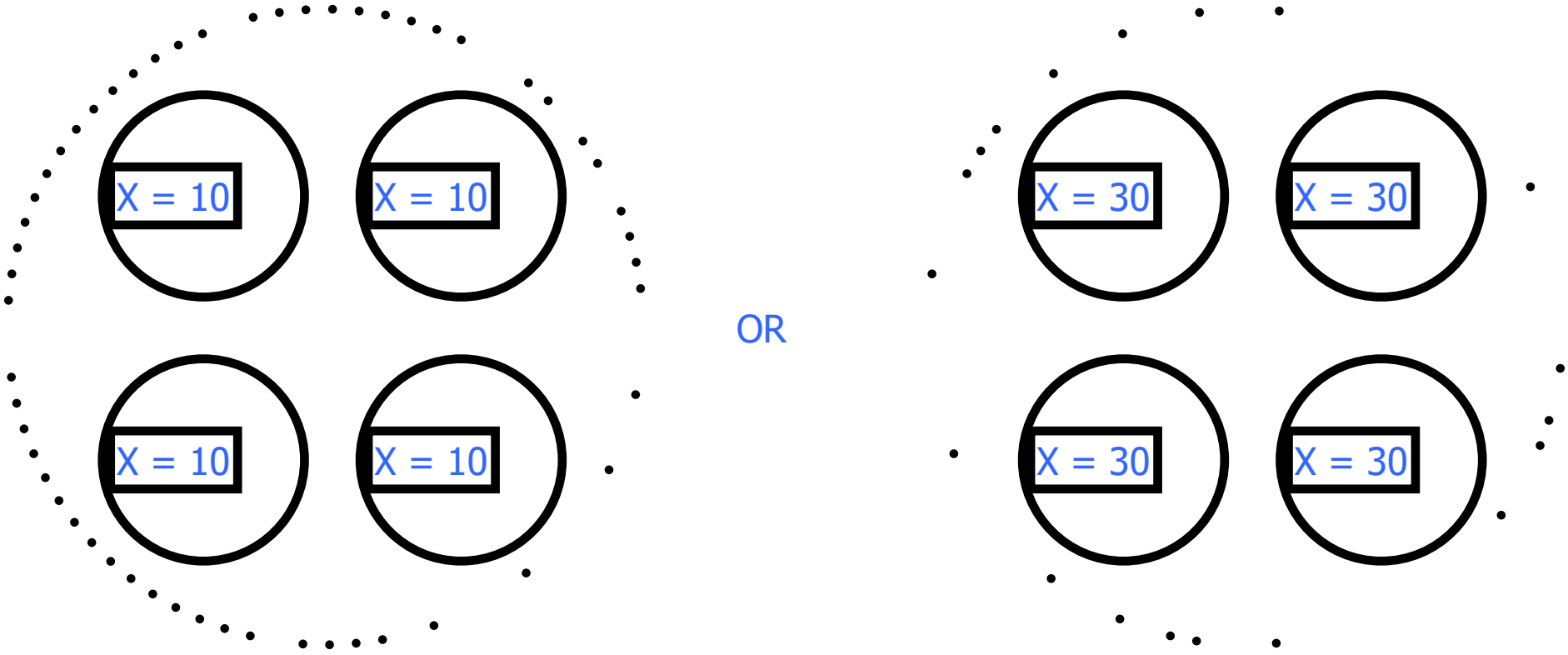


Replicas need to **agree** which requests have been handled

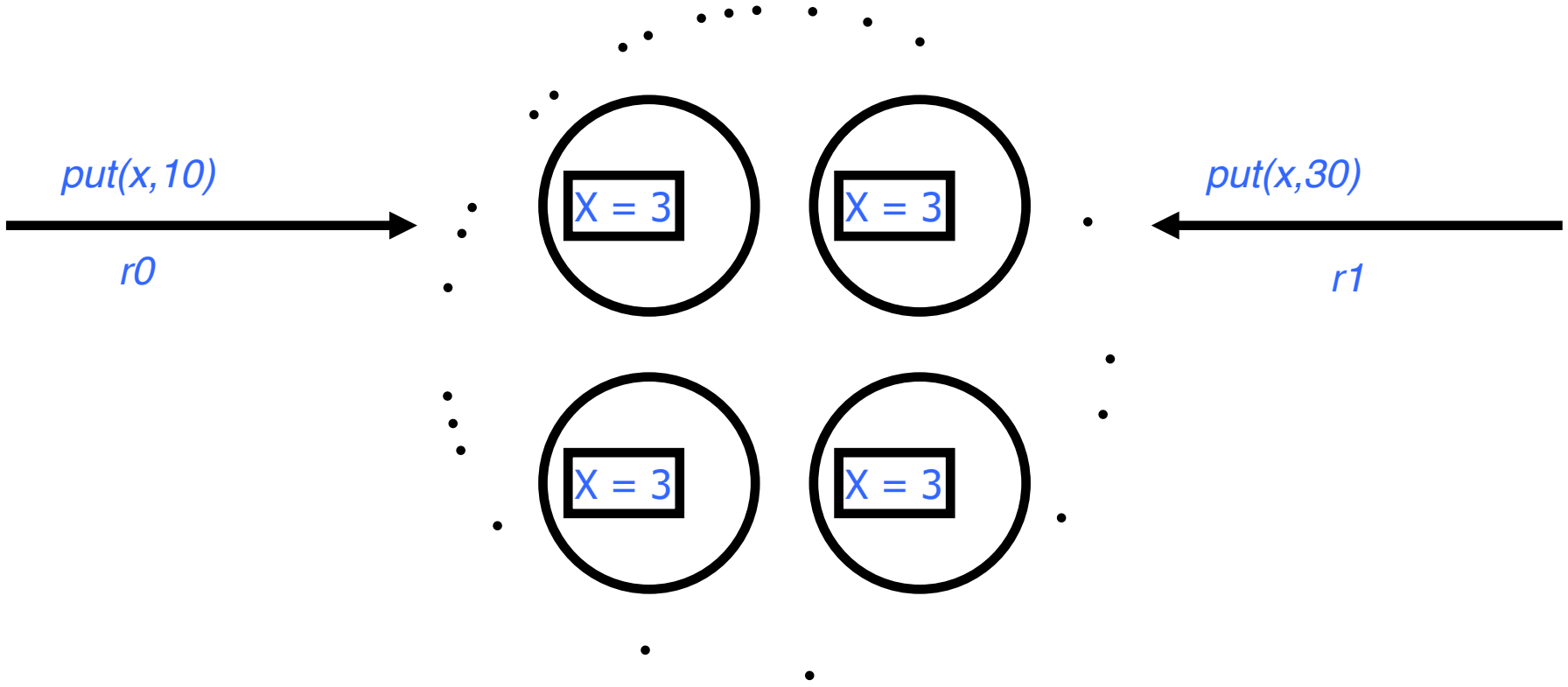
Two Writes



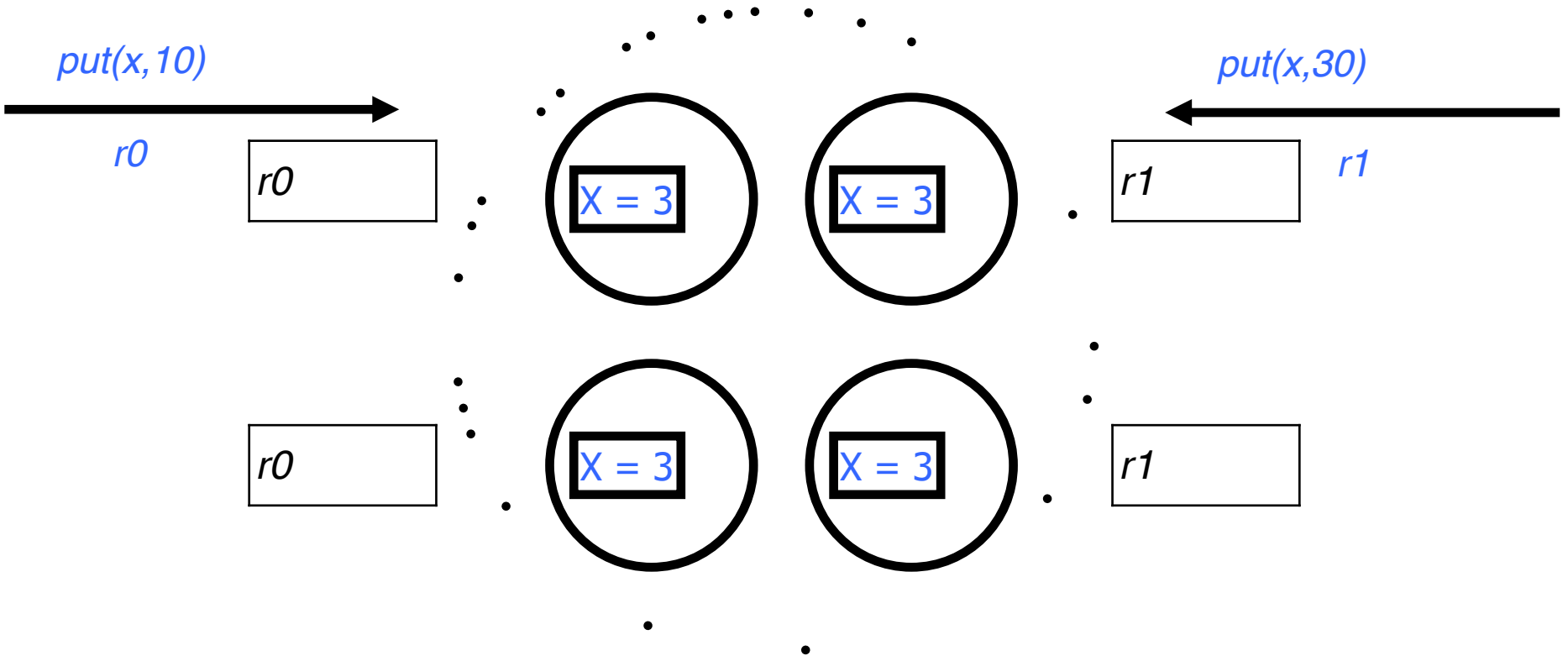
Either Outcome is Fine



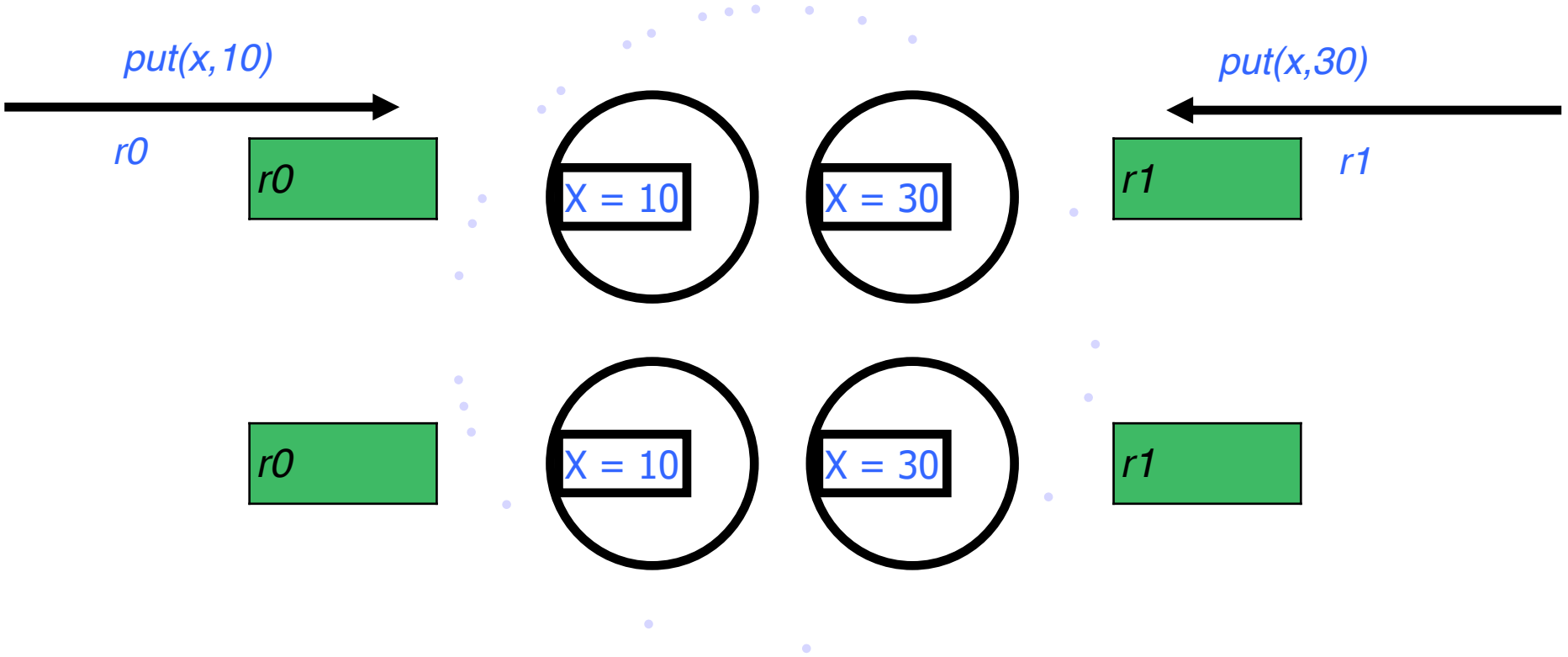
Order Matters



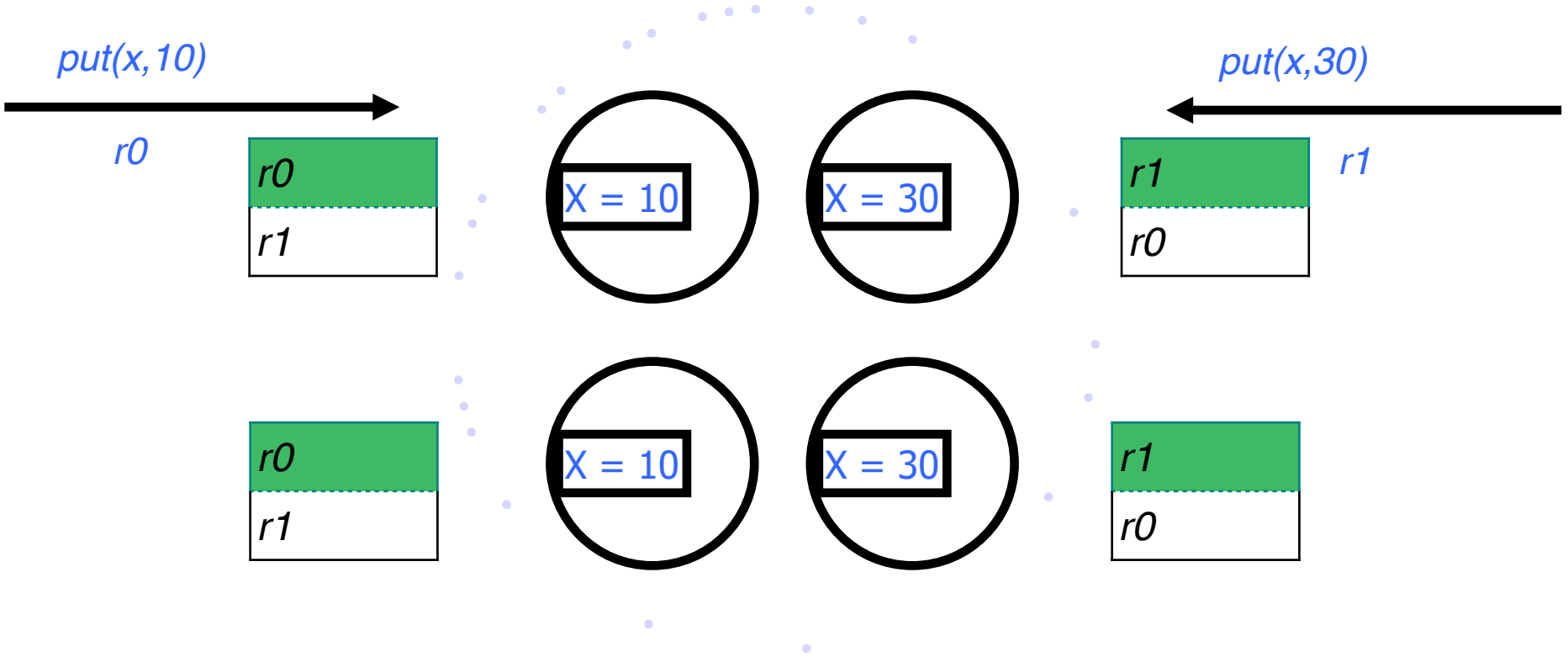
Order Matters



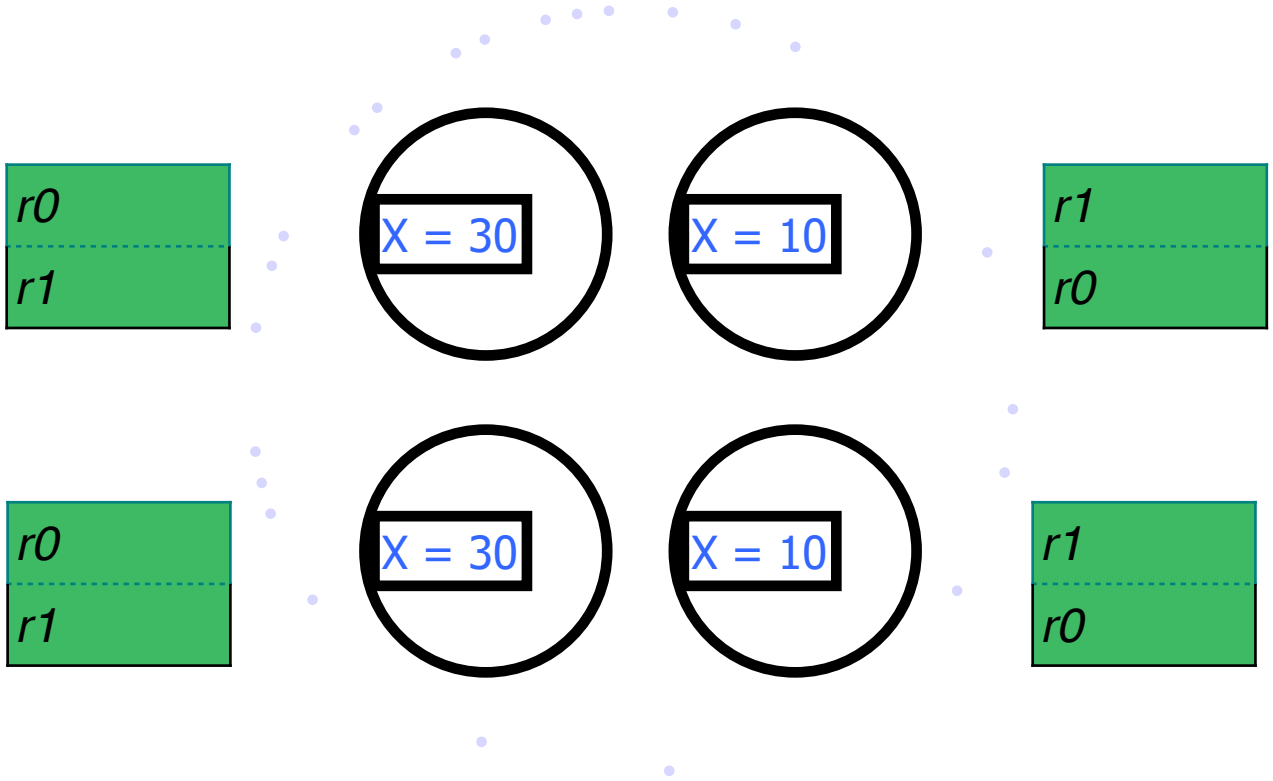
Order Matters



Order Matters



Order Matters



Replicas need to handle requests in the same **order**

Requirements

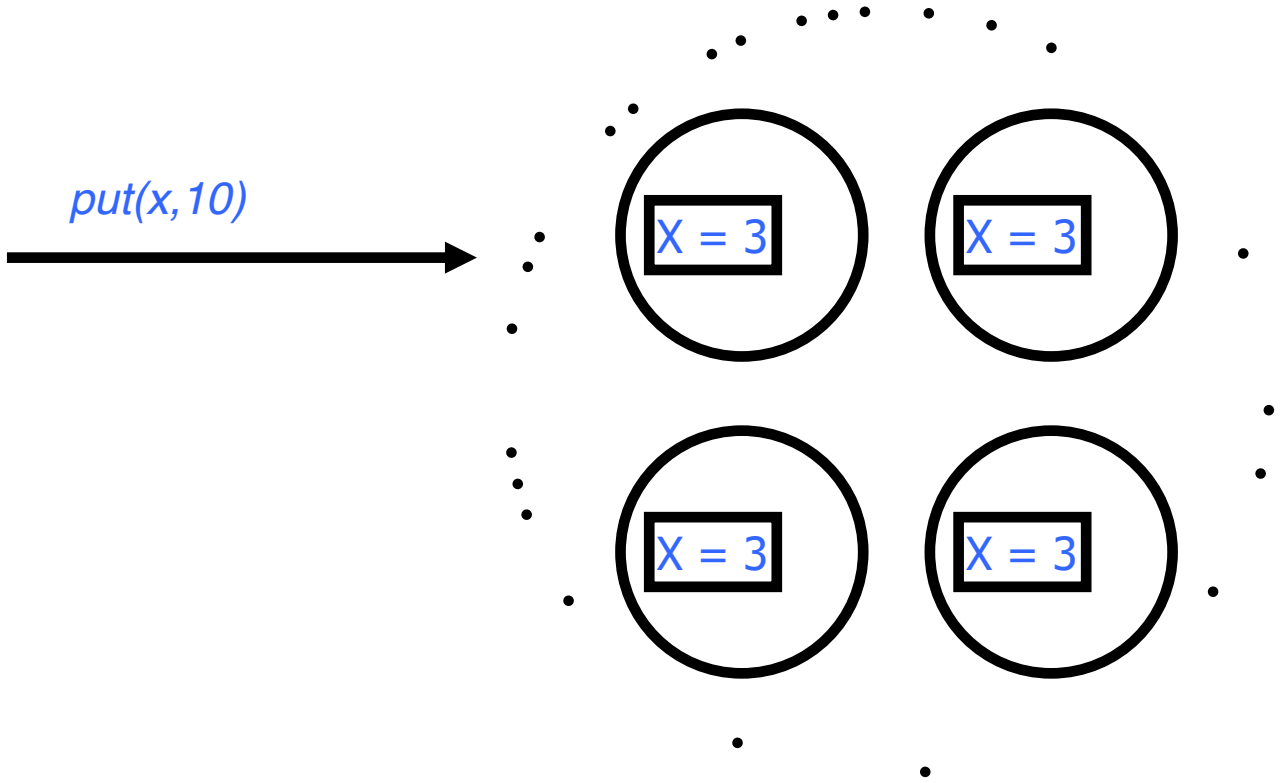
All non-faulty servers need…

- Agreement
  - Every replica needs to accept the same set of requests
- Order
  - All replicas process requests in the same relative order

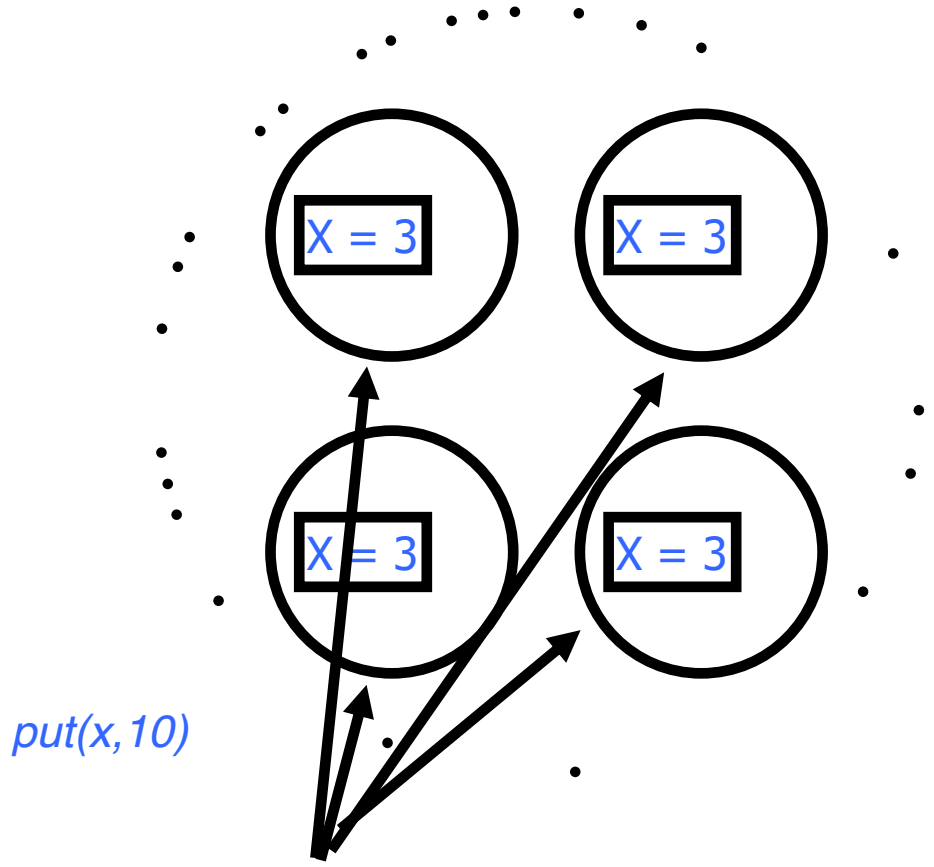
Idea for Agreement

- Someone proposes a request
- If the proposer is non-faulty, all servers will accept that request

Agreement



Agreement



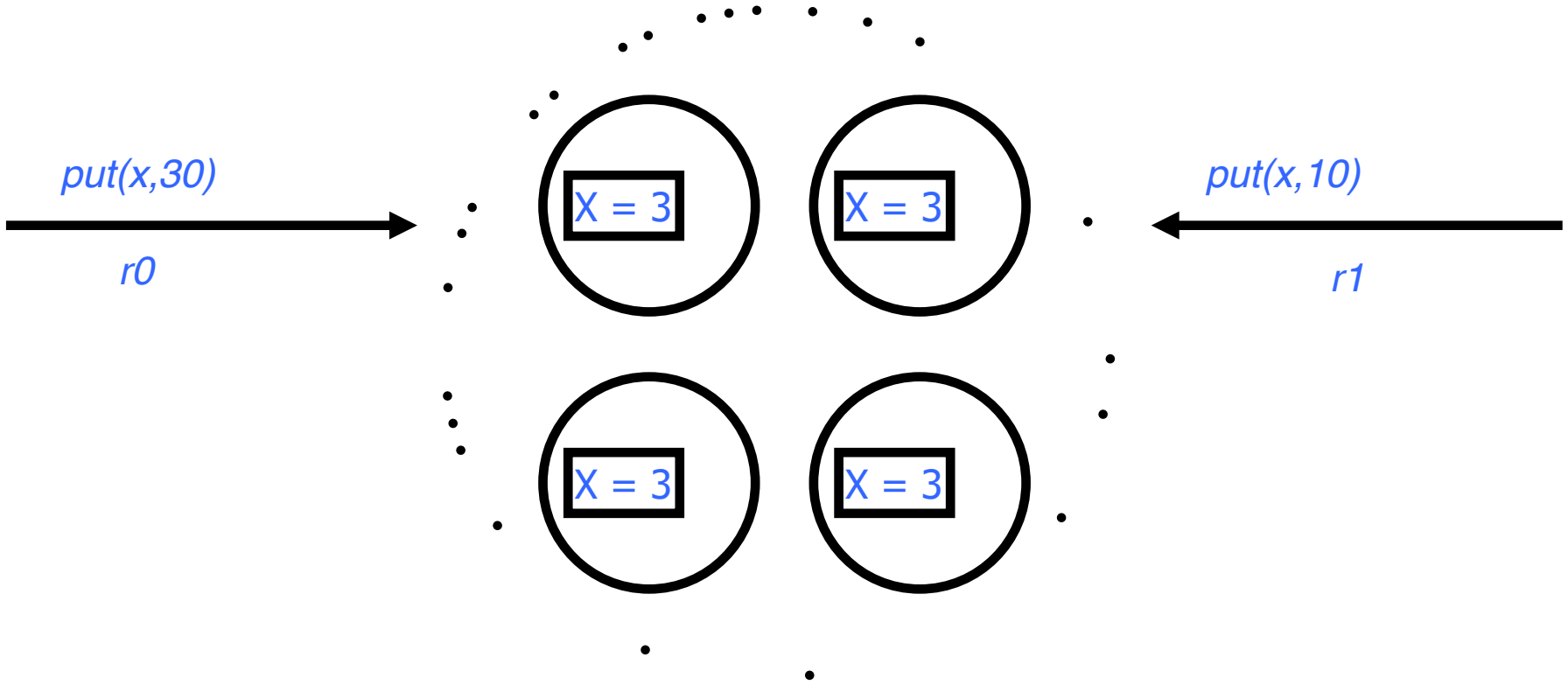
Non-faulty Transmitter

Idea for Order

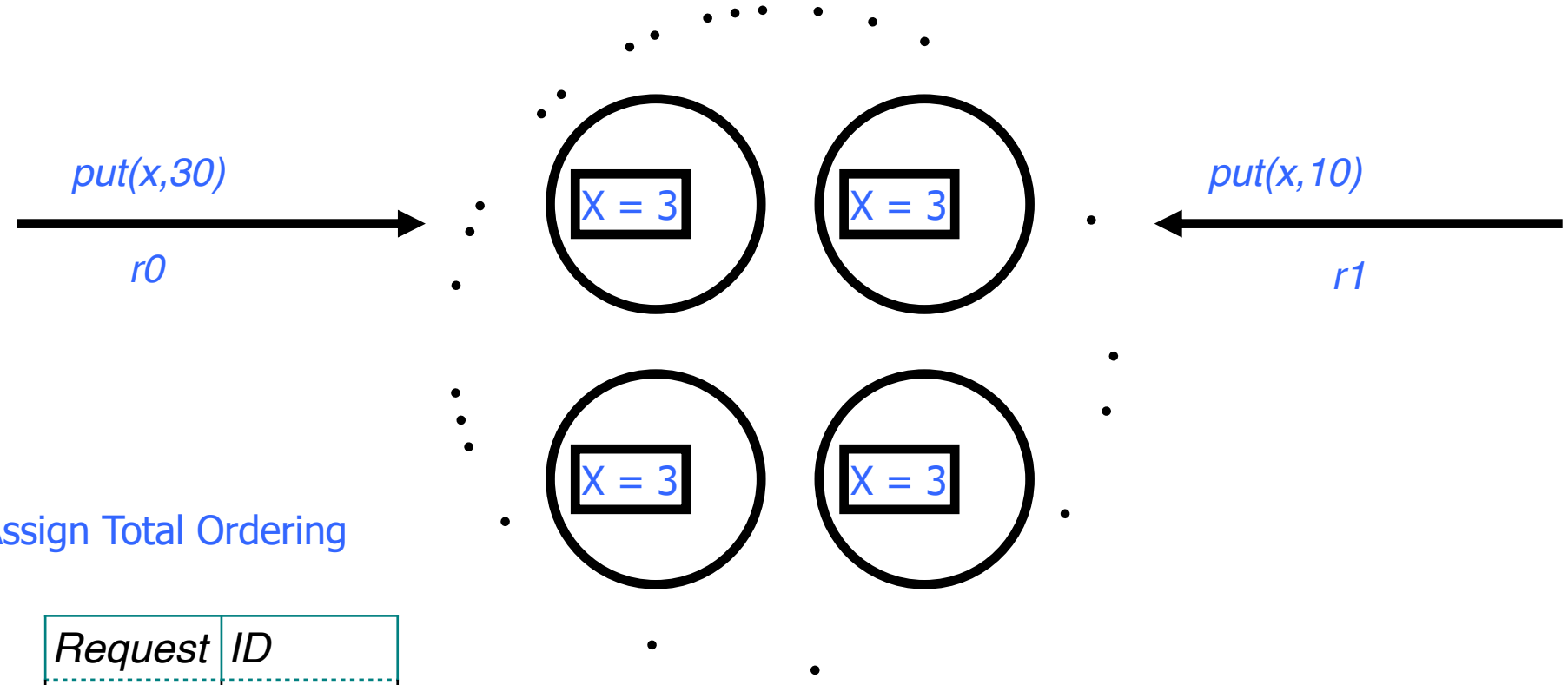
Assign unique ids to requests, process them in ascending order

- How do we assign unique ids in a distributed system?
- How do we know when every replica has processed a given request?

Order



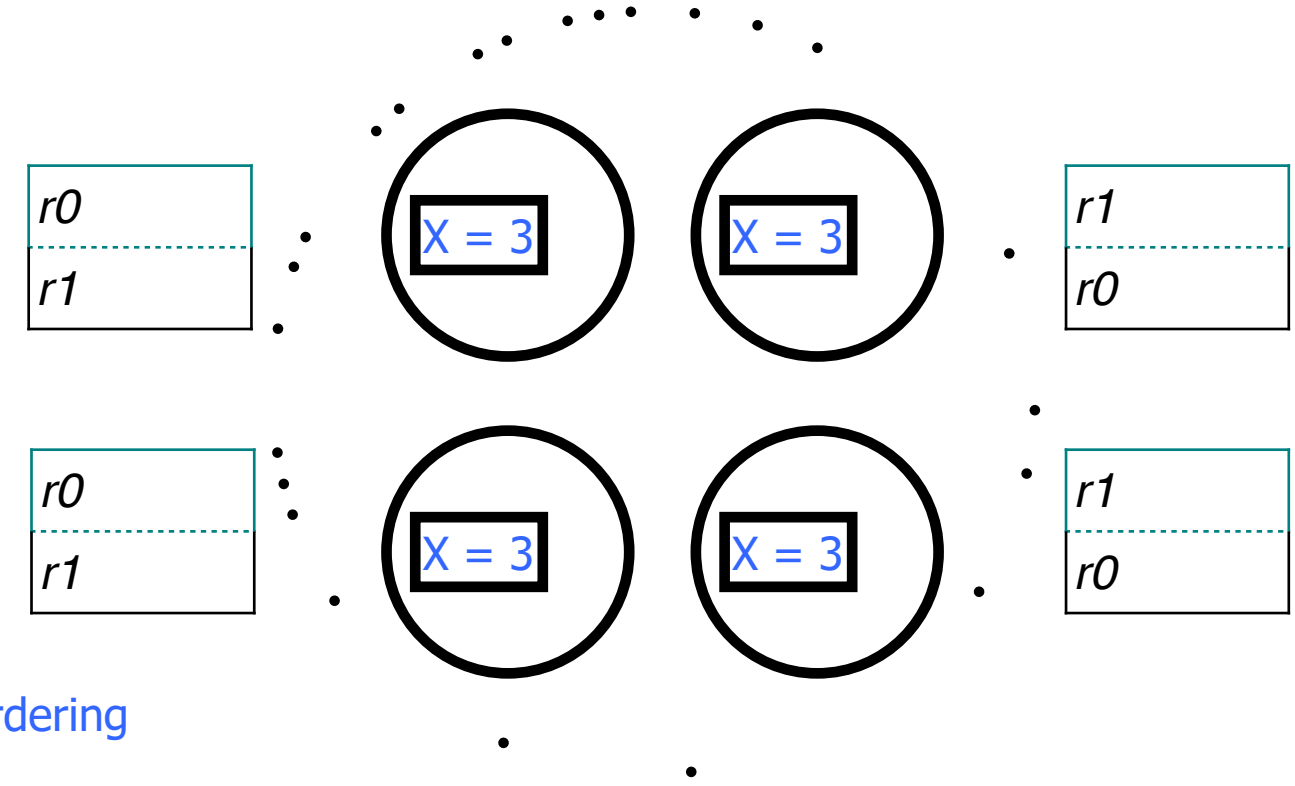
Order



Assign Total Ordering

Request	*ID*
r0	*1*
r1	*2*

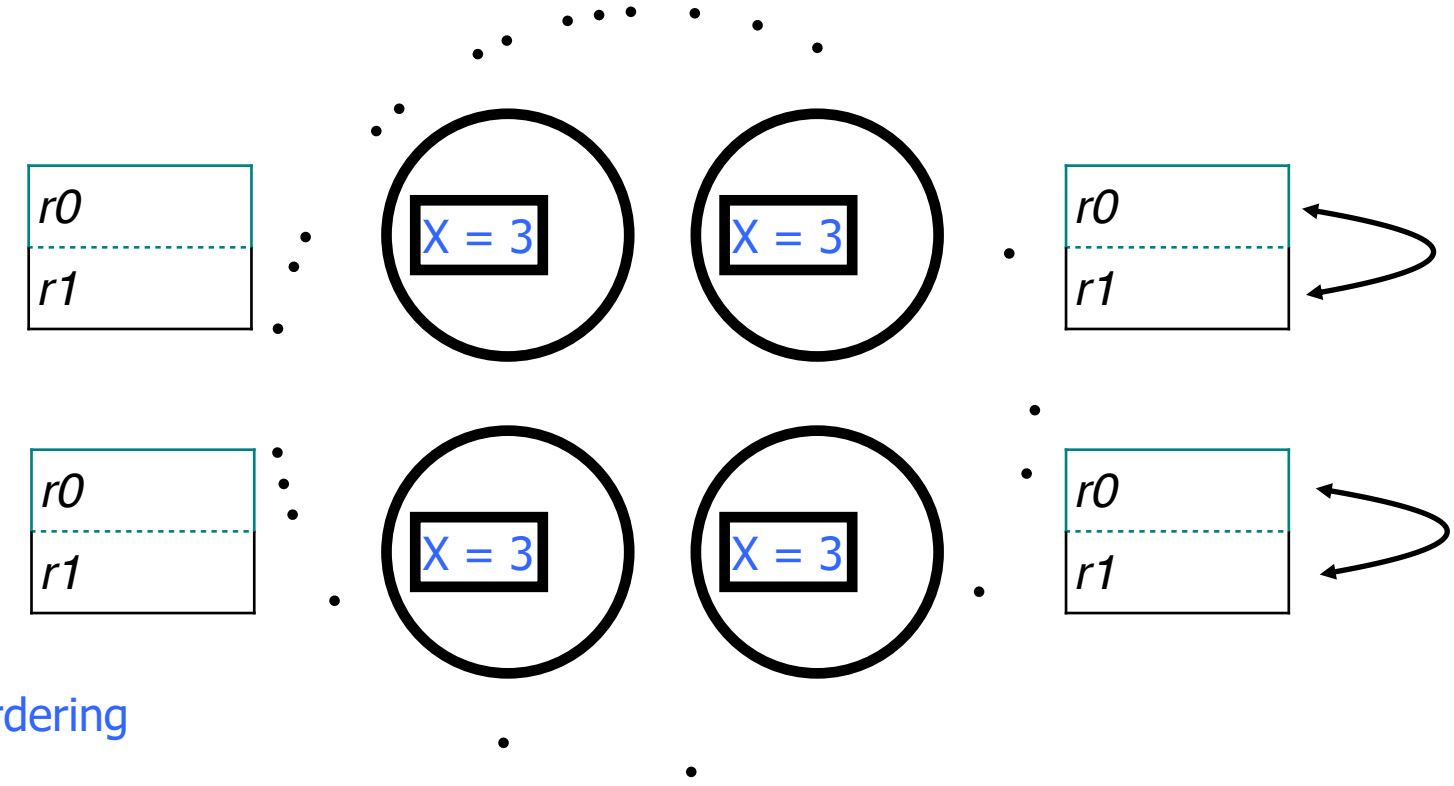
Order



Assign Total Ordering

Request	ID
r0	1
r1	2

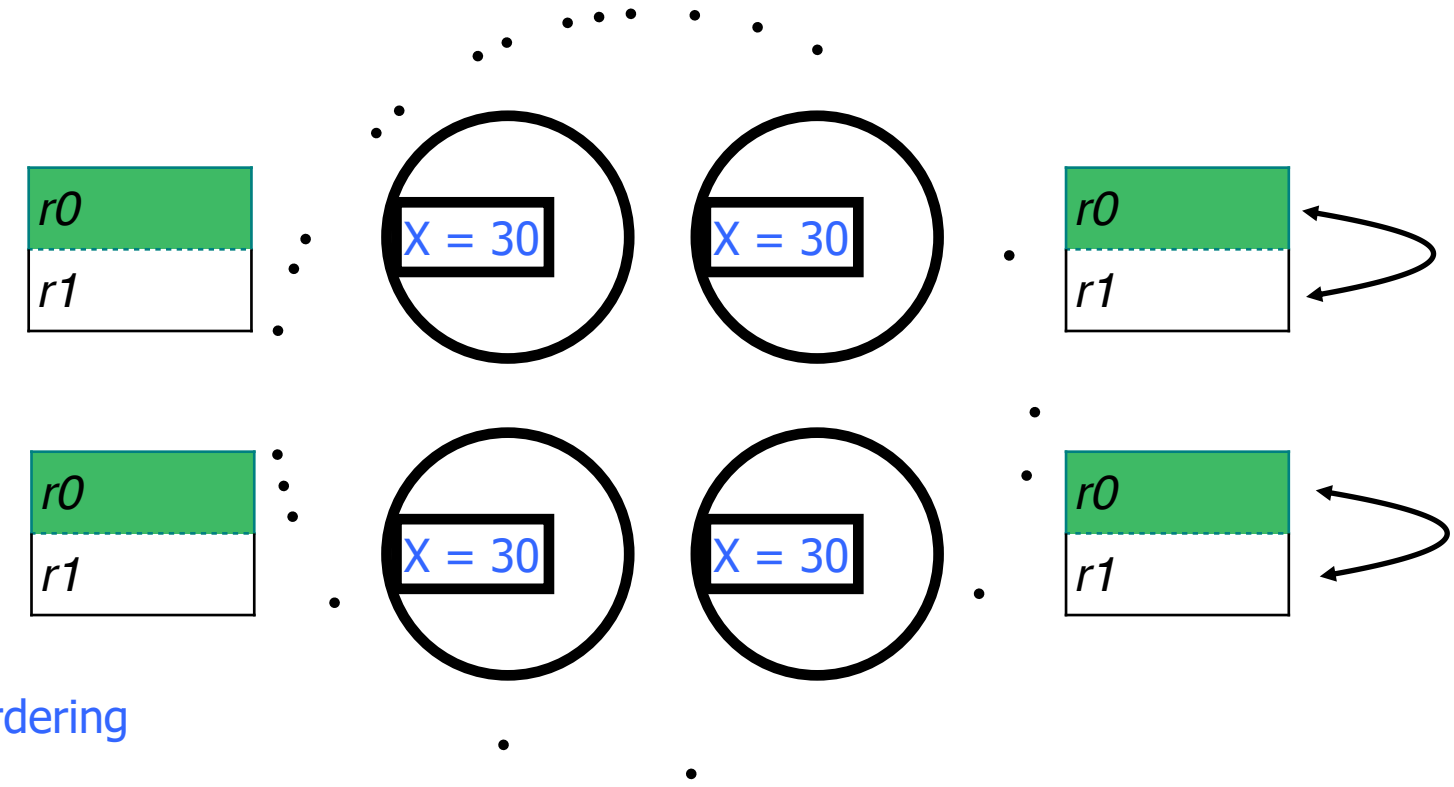
Order



Assign Total Ordering

Request	ID
r0	1
r1	2

Order



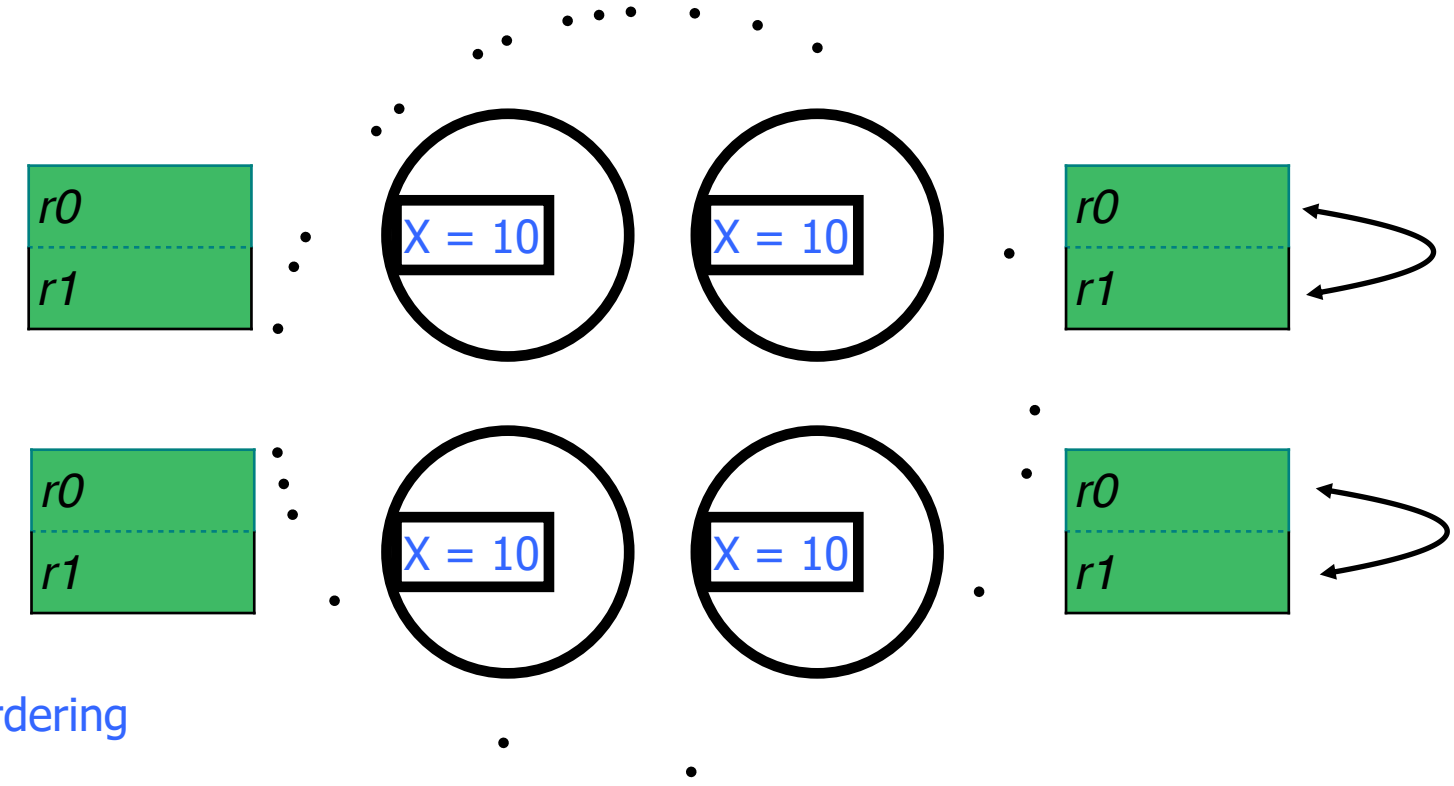
Assign Total Ordering

Request	ID
r0	1
r1	2

Cannot receive request with smaller ID

r0 is now stable!

Order



Assign Total Ordering

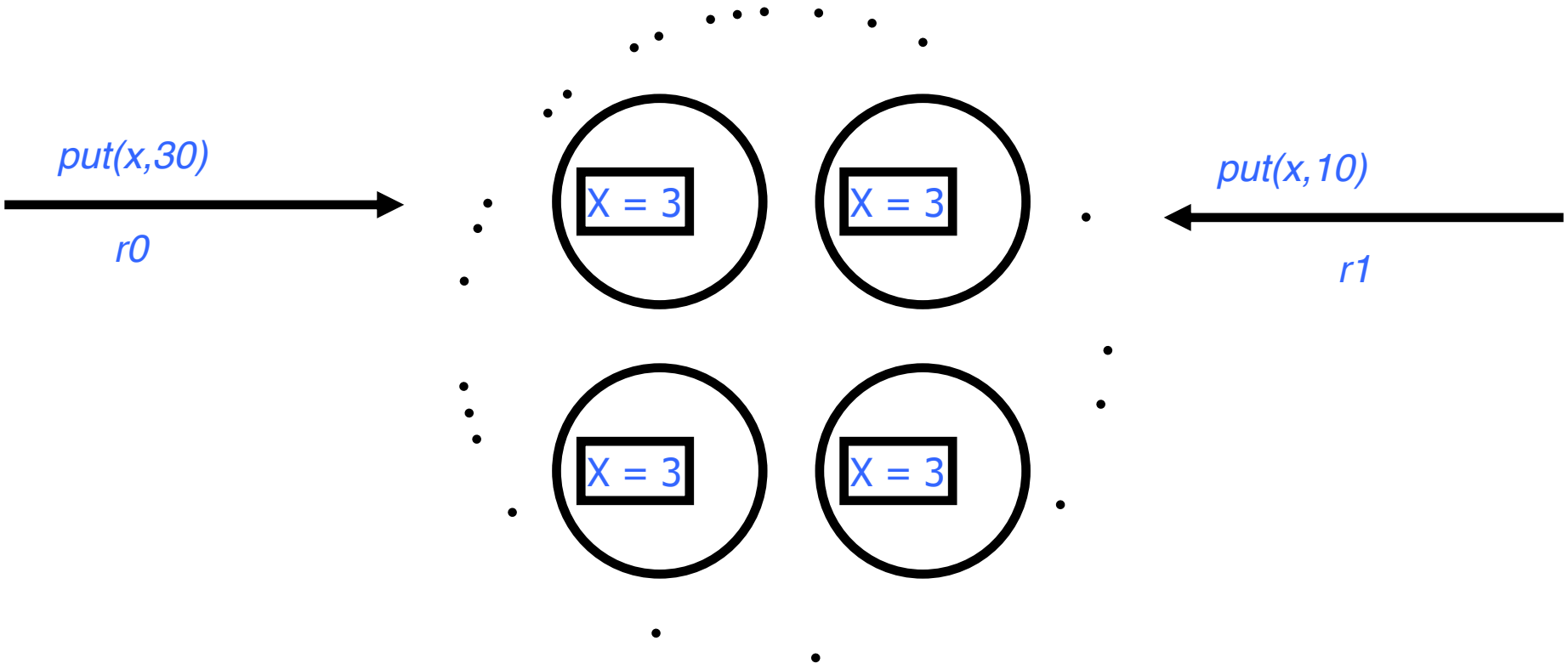
Request	ID
r0	1
r1	2

r0 is now stable!
r1 is now stable!

Generating IDs

- Order via clocks (client timestamp = id)
  - Logical clocks
  - Synchronized clocks
- Two-phase ID generation
  - Every replica proposes a candidate
  - One candidate is chosen and agreed upon by all replicas

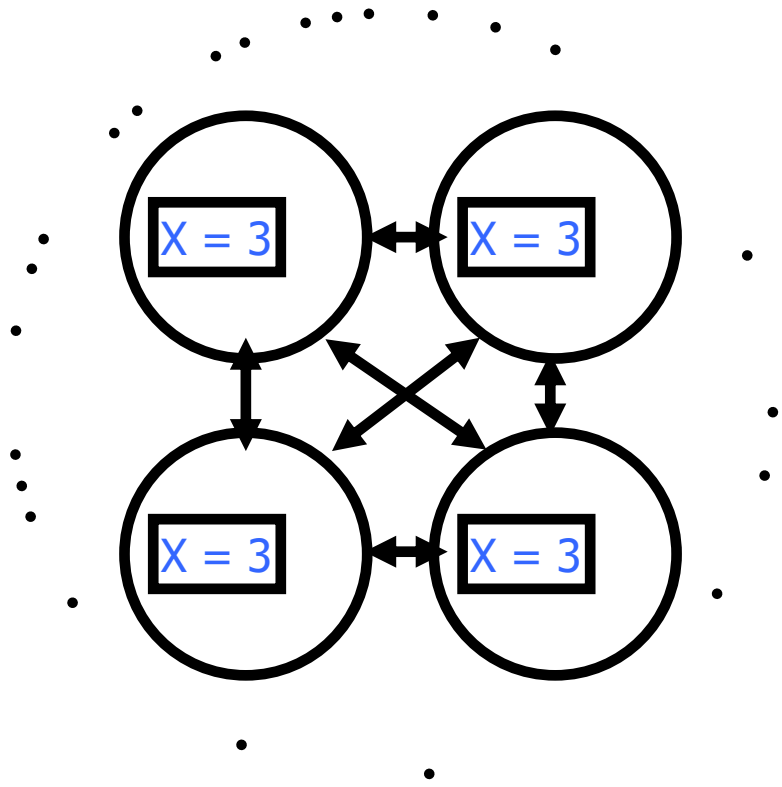
Replica ID Generation



Replica ID Generation

Req.	CUID	UID
r0	1.1	
r1	2.1	

Req.	CUID	UID
r0	1.2	
r1	2.2	



Req.	CUID	UID
r1	1.3	
r0	2.3	

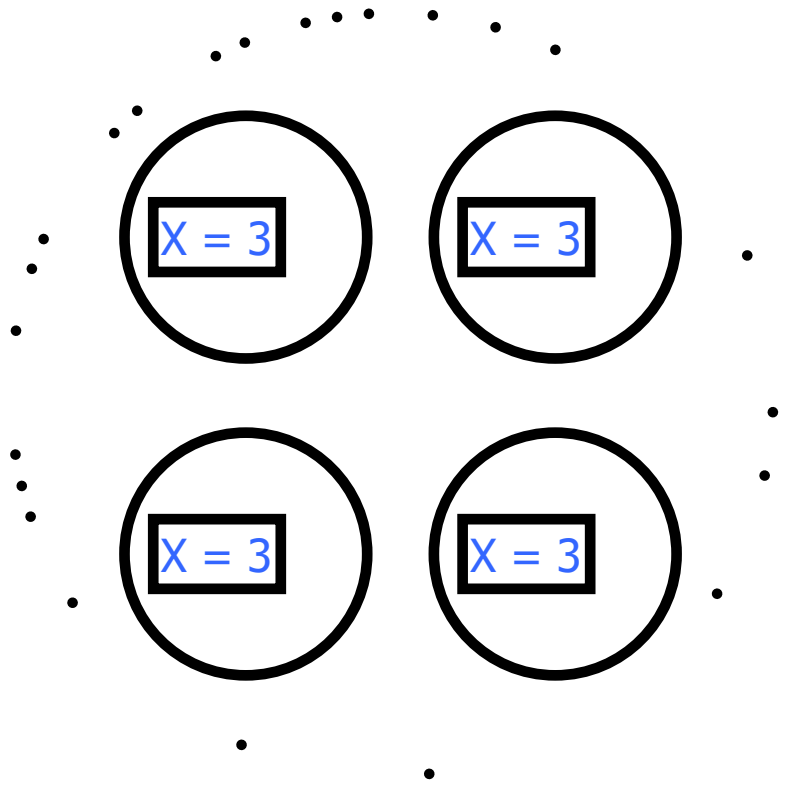
Req.	CUID	UID
r1	1.4	
r0	2.4	

1) Propose candidates

Replica ID Generation

Req.	CUID	UID
r0	1.1	2.4
r1	2.1	

Req.	CUID	UID
r1	1.3	
r0	2.3	2.4



Req.	CUID	UID
r0	1.2	2.4
r1	2.2	

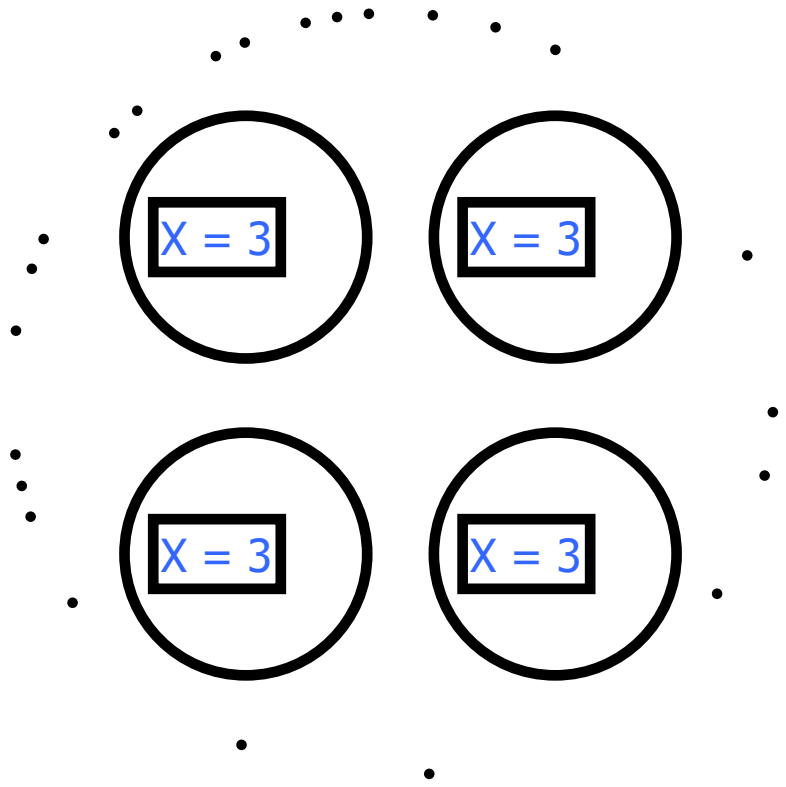
Req.	CUID	UID
r1	1.4	
r0	2.4	2.4

2) Accept *r0*

Replica ID Generation

Req.	*CUID*	*UID*
r0	*1.1*	*2.4*
r1	*2.1*	*2.2*

Req.	*CUID*	*UID*
r1	*1.3*	*2.2*
r0	*2.3*	*2.4*

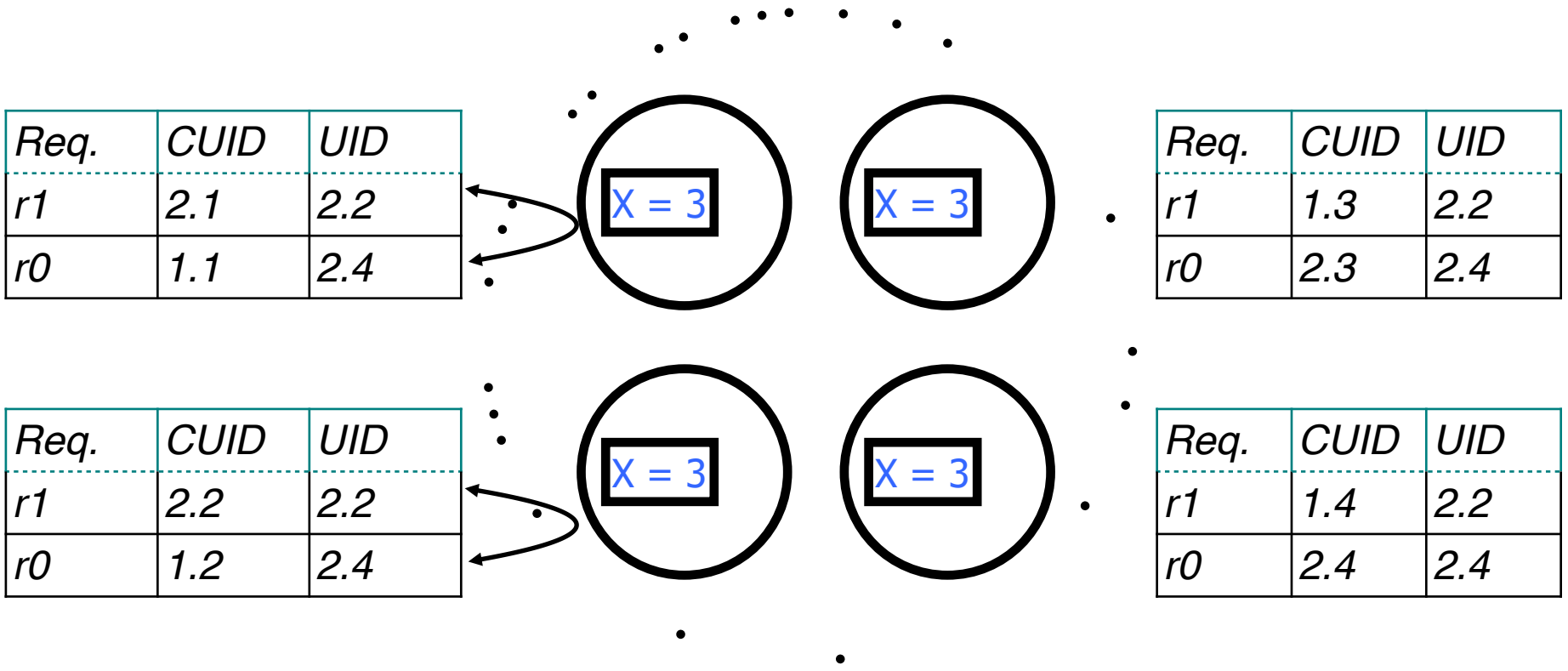


Req.	*CUID*	*UID*
r0	*1.2*	*2.4*
r1	*2.2*	*2.2*

Req.	*CUID*	*UID*
r1	*1.4*	*2.2*
r0	*2.4*	*2.4*

3) Accept *r1*

Replica ID Generation

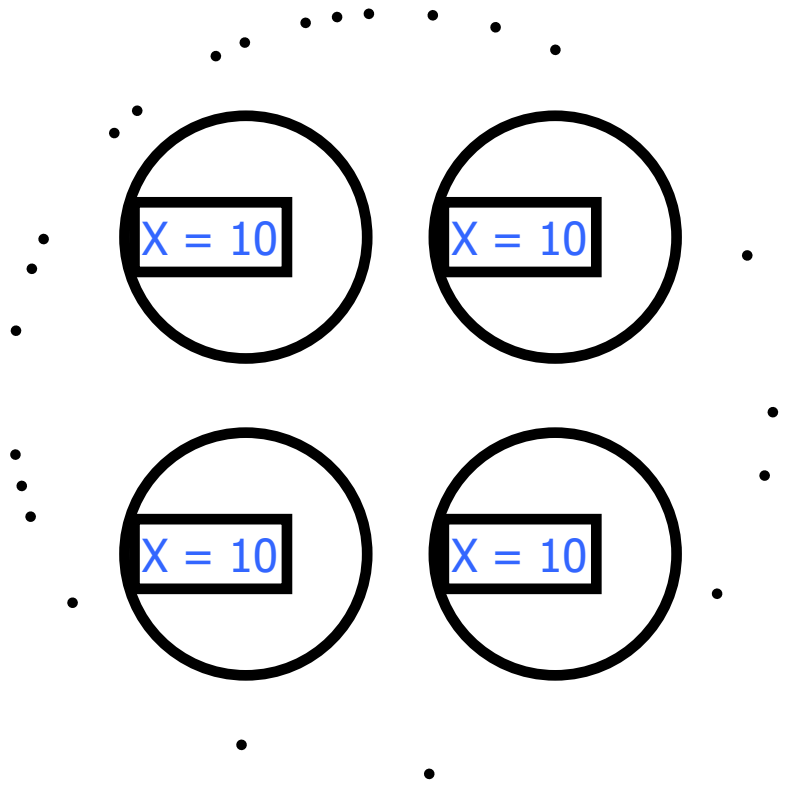


r1 is now stable

Replica ID Generation

Req.	*CUID*	*UID*
r1	*2.1*	*2.2*
r0	*1.1*	*2.4*

Req.	*CUID*	*UID*
r1	*1.3*	*2.2*
r0	*2.3*	*2.4*



Req.	*CUID*	*UID*
r1	*2.2*	*2.2*
r0	*1.2*	*2.4*

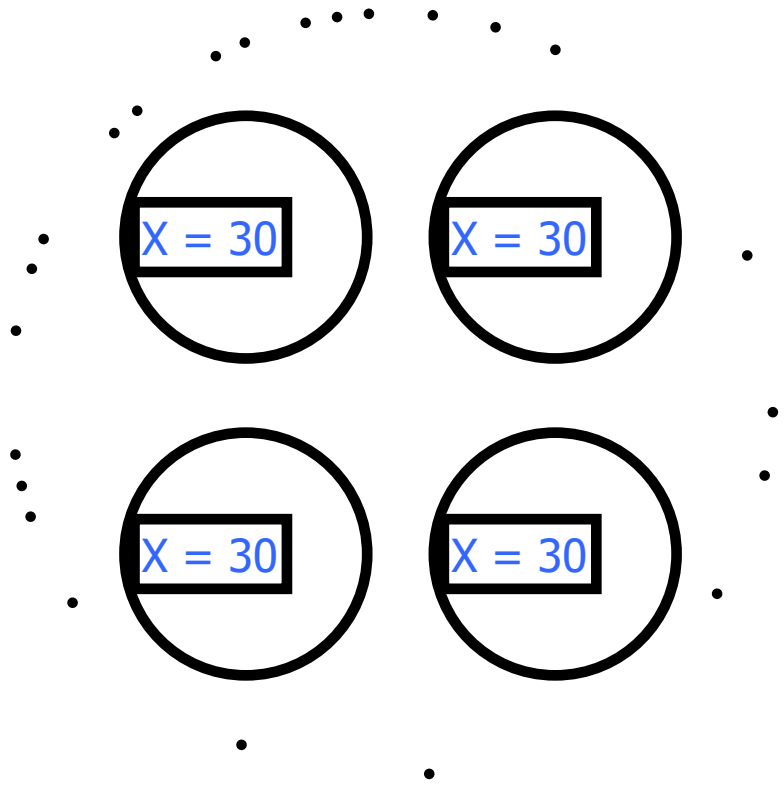
Req.	*CUID*	*UID*
r1	*1.4*	*2.2*
r0	*2.4*	*2.4*

4) Apply *r1*

Replica ID Generation

Req.	*CUID*	*UID*
r1	*2.1*	*2.2*
r0	*1.1*	*2.4*

Req.	*CUID*	*UID*
r1	*1.3*	*2.2*
r0	*2.3*	*2.4*



Req.	*CUID*	*UID*
r1	*2.2*	*2.2*
r0	*1.2*	*2.4*

Req.	*CUID*	*UID*
r1	*1.4*	*2.2*
r0	*2.4*	*2.4*

5) Apply *r0*

Rules for Replica-Generated IDs

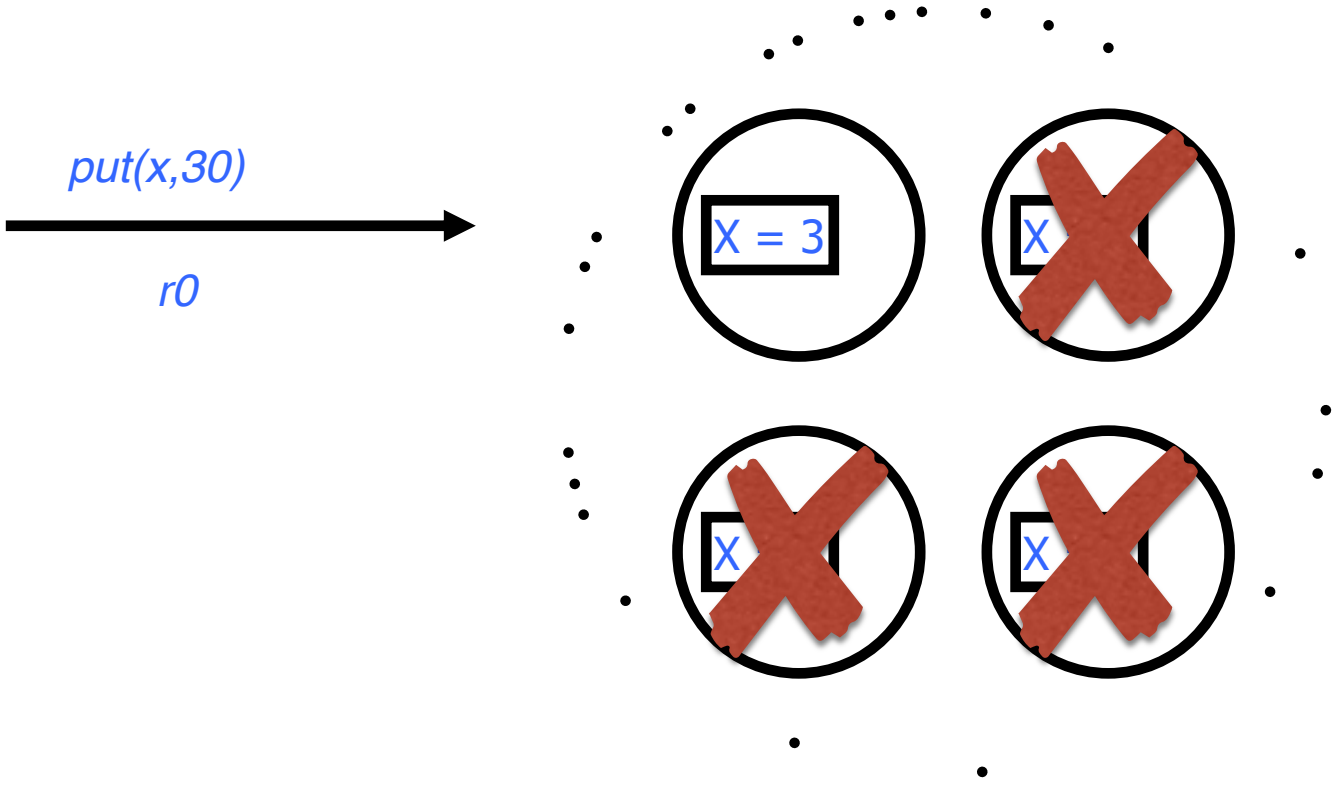
- Any new candidate ID must be > ID of any accepted request
- The ID selected from the candidate list must be >= each candidate

- When is a candidate stable?
  - It has been accepted
  - No other pending request with a smaller candidate ID

Faults

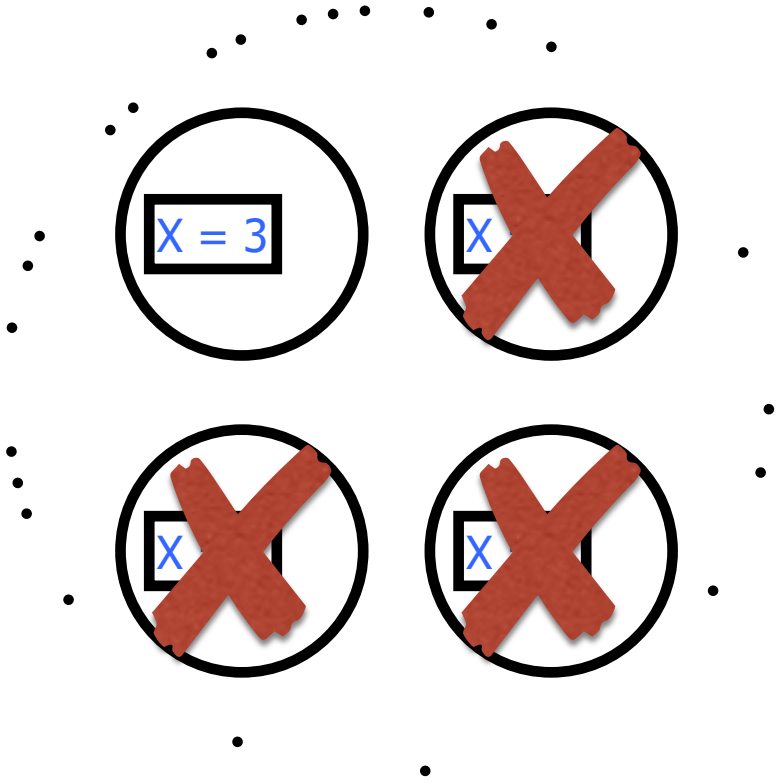
- Fail-Stop
  - A faulty server can be detected as faulty
- Byzantine
  - Faulty servers can do arbitrary, perhaps malicious things
  - This includes crash failures (server can stop responding without notification)

Fail-Stop Tolerance



Fail-Stop Tolerance

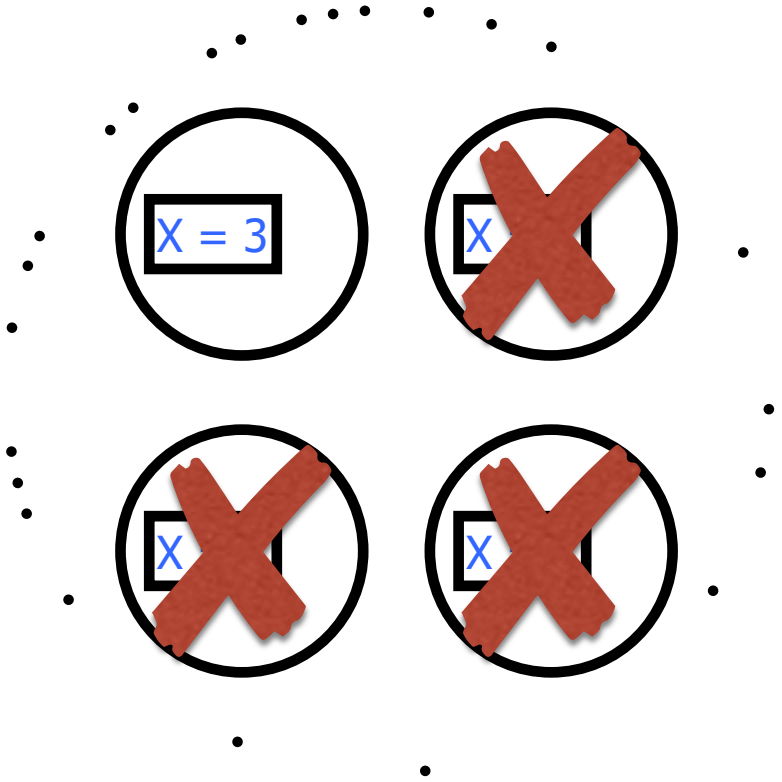
Req.	*CUID*	*UID*
r0	*1.1*	



1) Propose Candidates….

Fail-Stop Tolerance

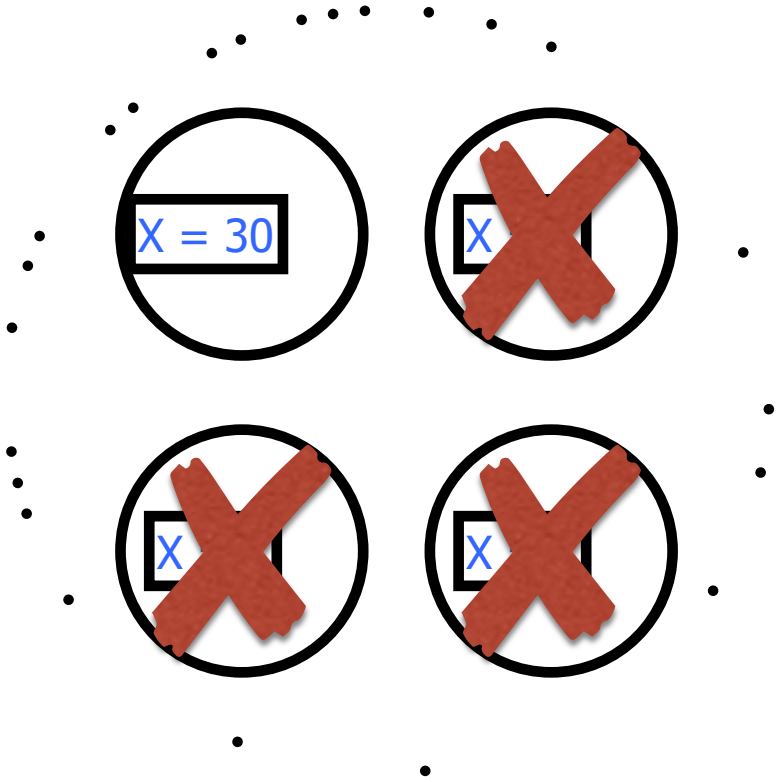
Req.	*CUID*	*UID*
r0	*1.1*	*1.1*



2) Accept *r0*

Fail-Stop Tolerance

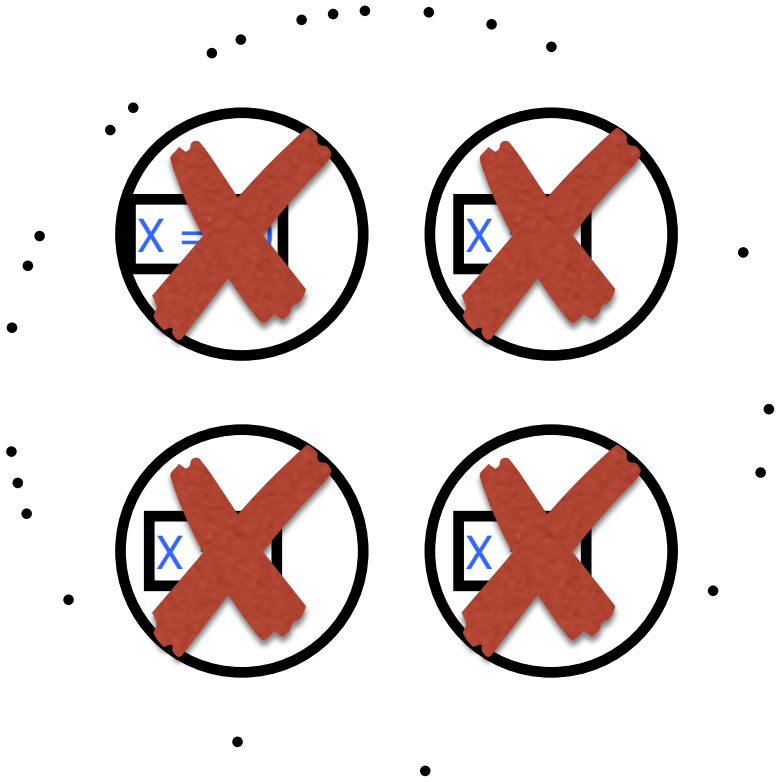
Req.	*CUID*	*UID*
r0	*1.1*	*1.1*



2) Apply *r0*

Fail-Stop Tolerance

GAME OVER!!!



2) Apply *r0*

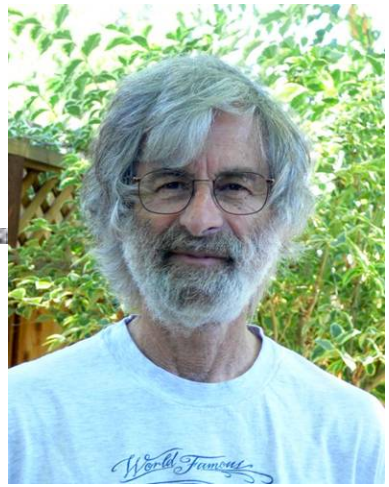
Fail-Stop Fault Tolerance

- To tolerate t failures, need t+1 servers
- As long as 1 server remains, we're OK
- Only need to participate in protocols with other live servers

Byzantine Fault Tolerance

- To tolerate t failures, need 2t + 1 servers
- Protocols now involve votes
  - Can only trust server response if the majority of servers say the same thing
- t + 1 servers need to participate in replication protocols

Lamport (1978)



This is a distributed algorithm. Each process independently follows these rules, and there is no central synchronizing process or central storage. This approach can be generalized to implement any desired synchronization for such a distributed multiprocess system. The synchronization is specified in terms of a *State Machine*,

Fault-Tolerant State Machines

- Implement the state machine on multiple processors
- State machine replication
  - Each starts in the same initial state
  - Executes the same requests
  - Requires consensus to execute in same order
  - Deterministic, each will do the exact same thing
  - Produce the same output

Consensus

- Termination
- Validity
- Integrity
- Agreement

Ensures procedures are called in same order across all machines