

CS 5450

# State Machine Replication



# Key Ideas

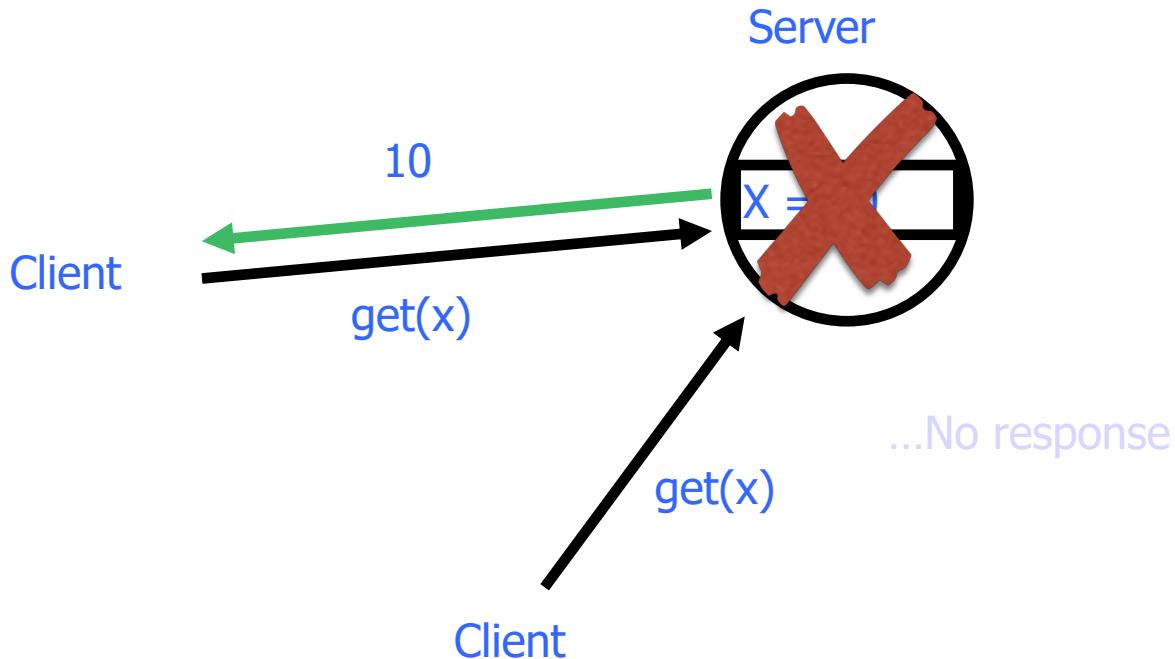
---

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
- Key examples of distributed algorithms that generically implement SMR
- Formalizes notions of fault-tolerance in SMR

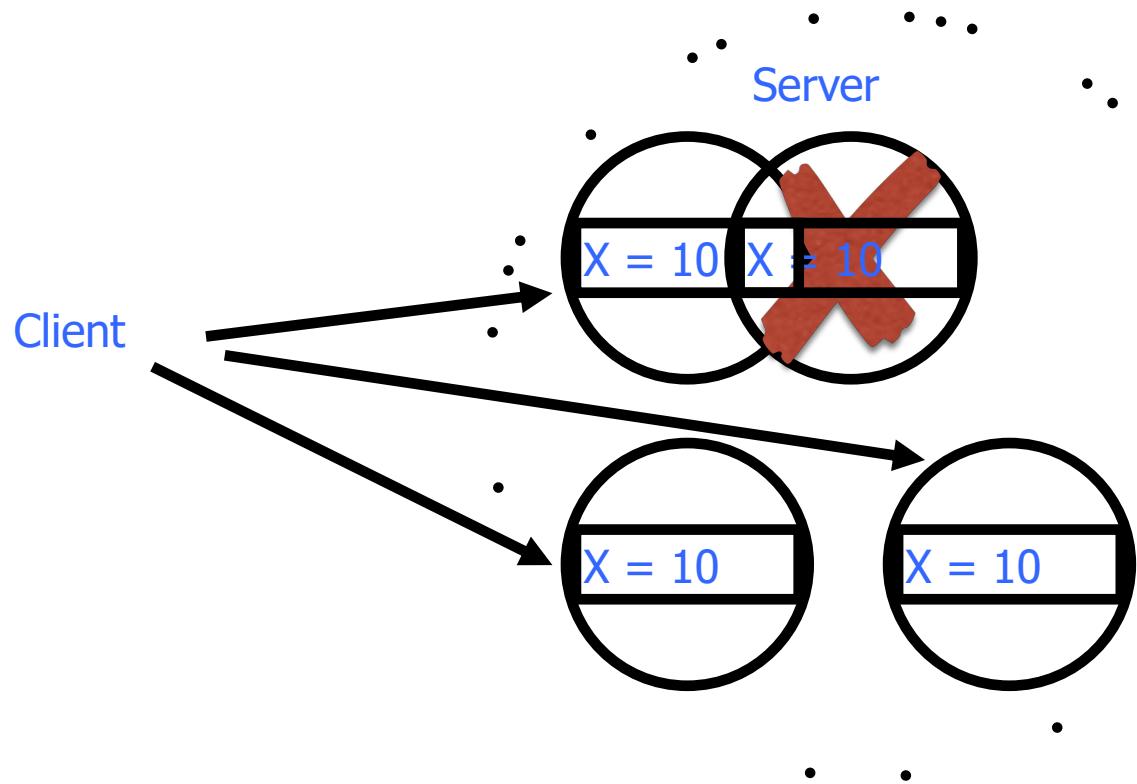
# Motivation

---



# Motivation

---



# Motivation

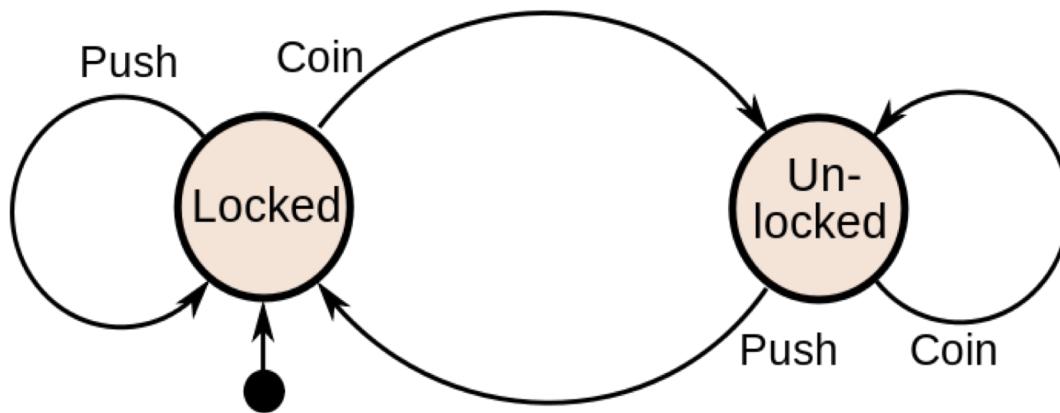
---

- ◆ Need replication for fault tolerance
- ◆ What happens in these scenarios without replication?
  - Storage - disk failure
  - Web service - network failure
- ◆ Be able to reason about failure tolerance
- ◆ How badly can things go wrong and have our system continue to function?

# State Machines

---

- ◆ State variables
- ◆ Deterministic commands



# Requests and Causality

---

Process order consistent with potential causality

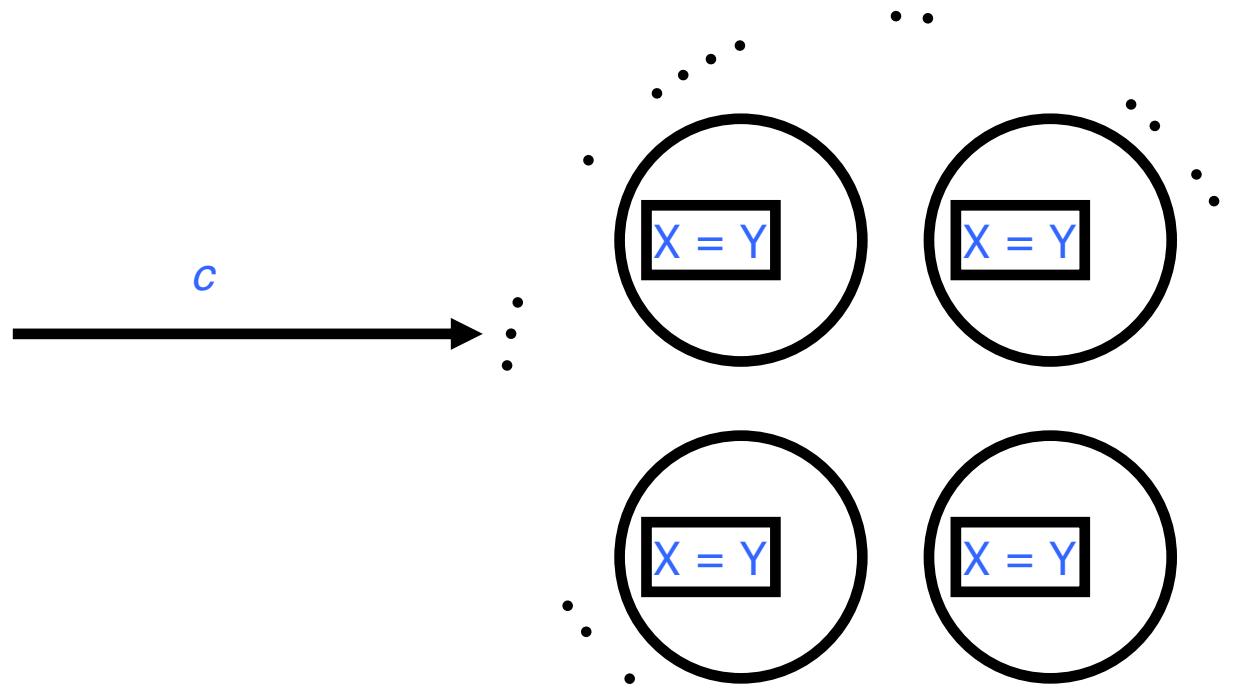
- ◆ Client A sends  $r$ , then  $r'$
- ◆  $r$  is processed before  $r'$
- ◆  $r$  causes Client B to send  $r'$
- ◆  $r$  is processed before  $r'$ .

# Coding State Machines

---

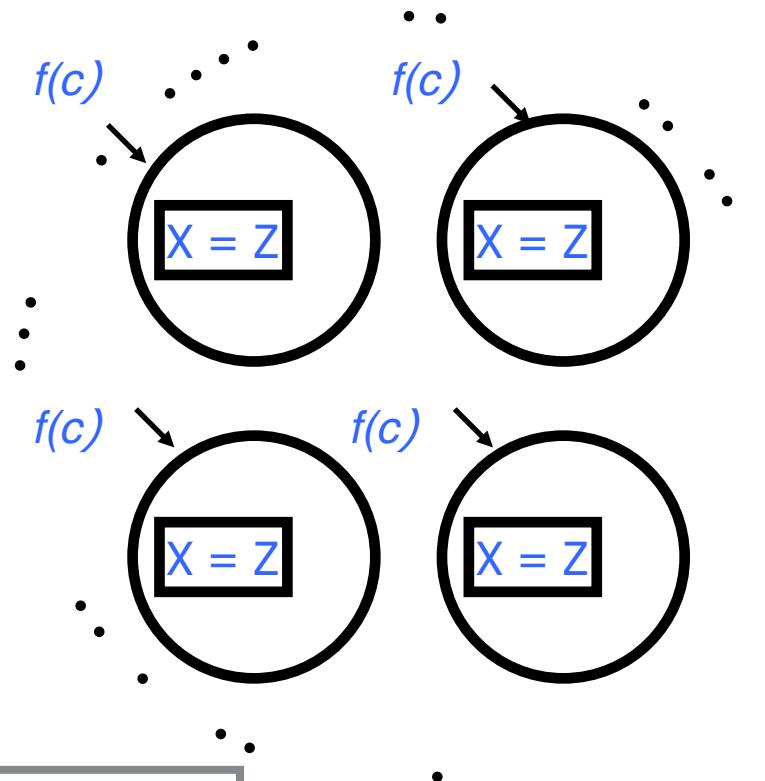
- ◆ State machines are procedures
- ◆ Client calls procedure
- ◆ Avoid loops
- ◆ More flexible structure

# State Machine Replication



State Machine Replica

# State Machine Replication

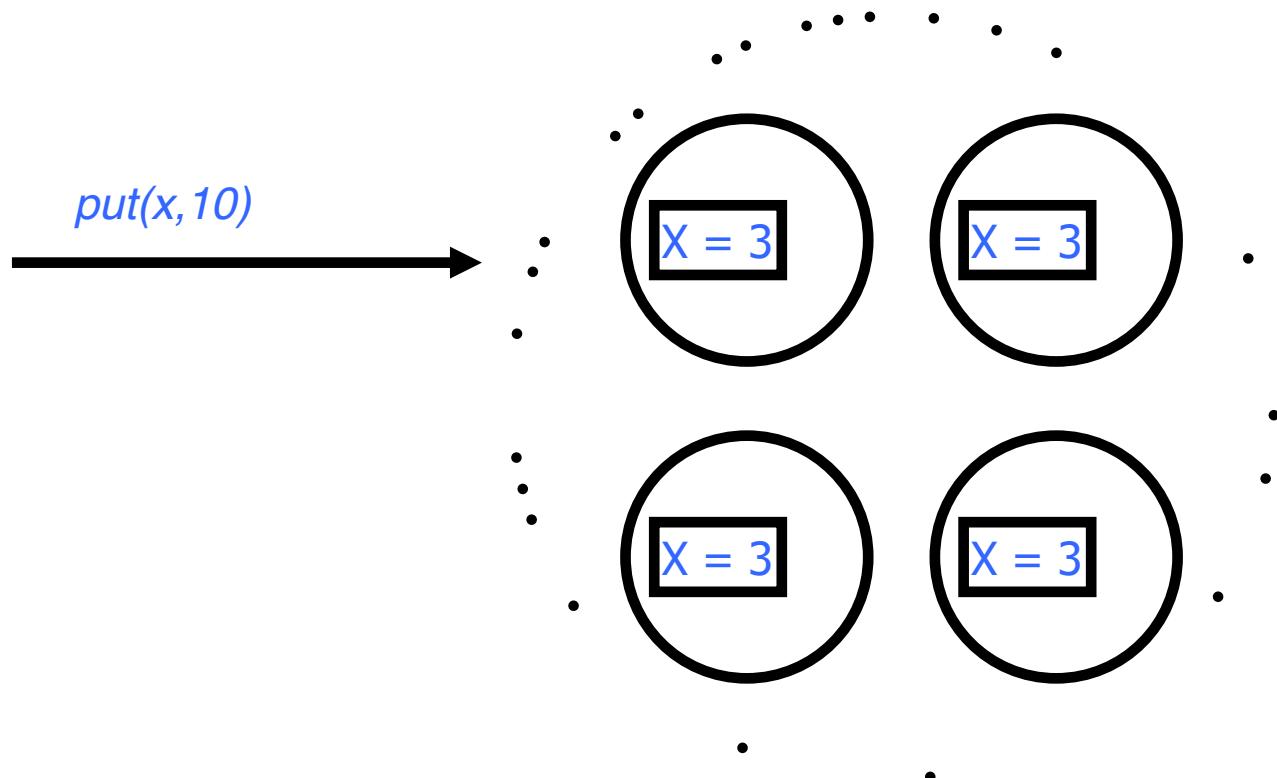


State Machine Replica

# Write

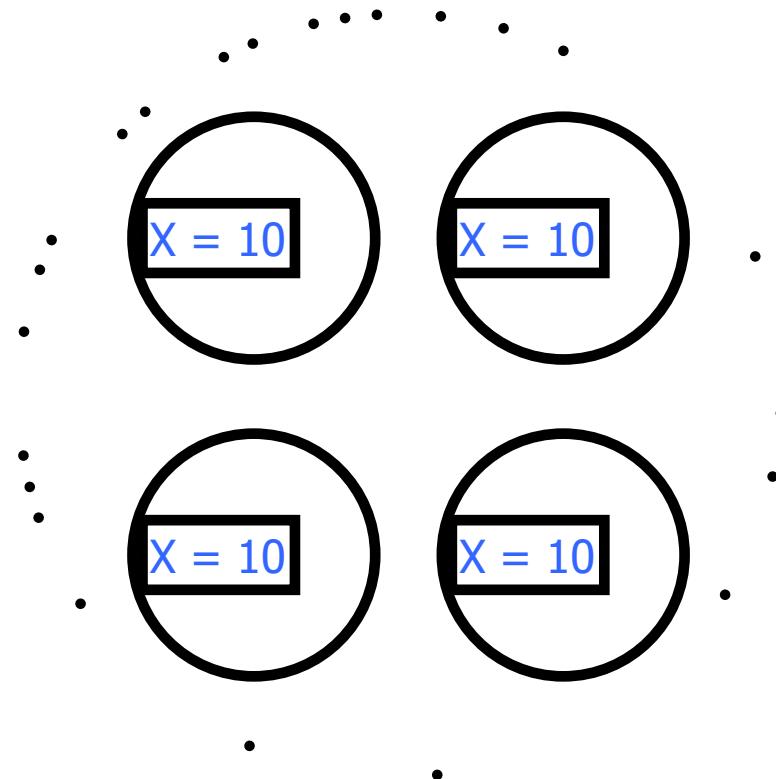
---

$put(x, 10)$



# After the Write

---

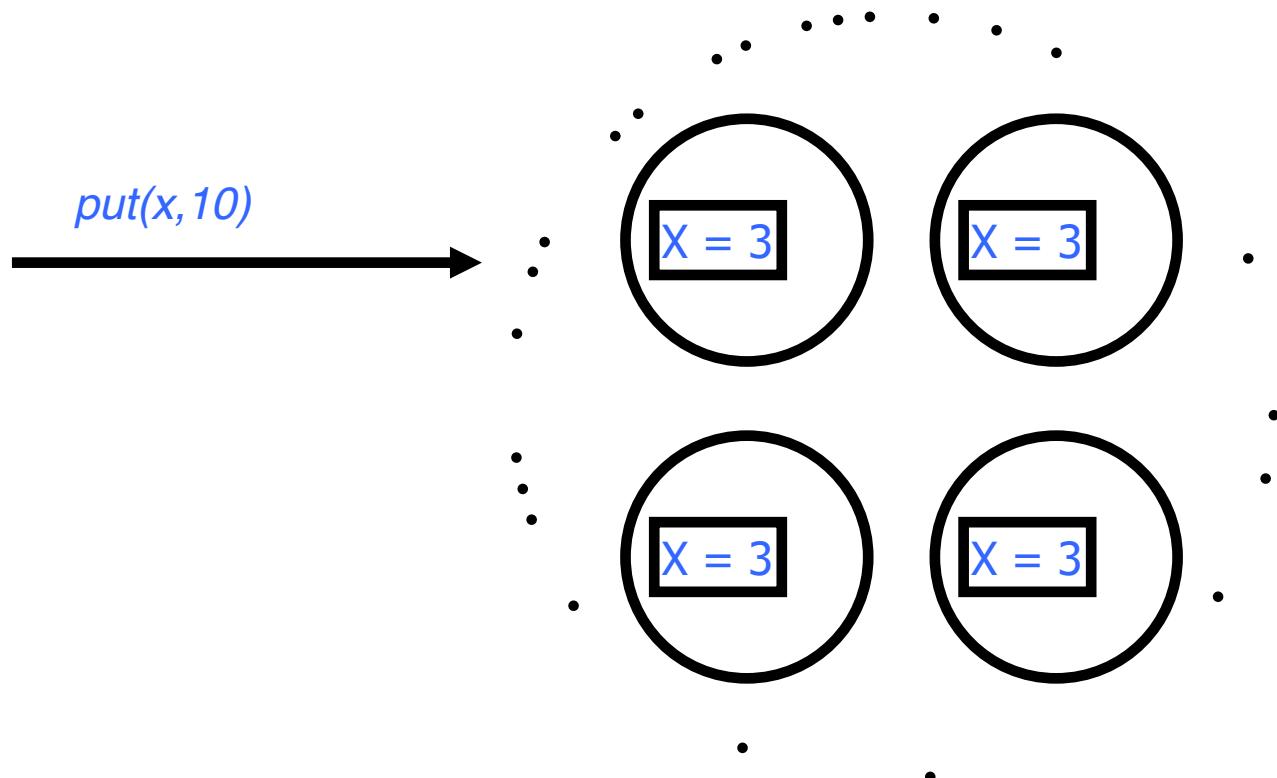


**Great!**

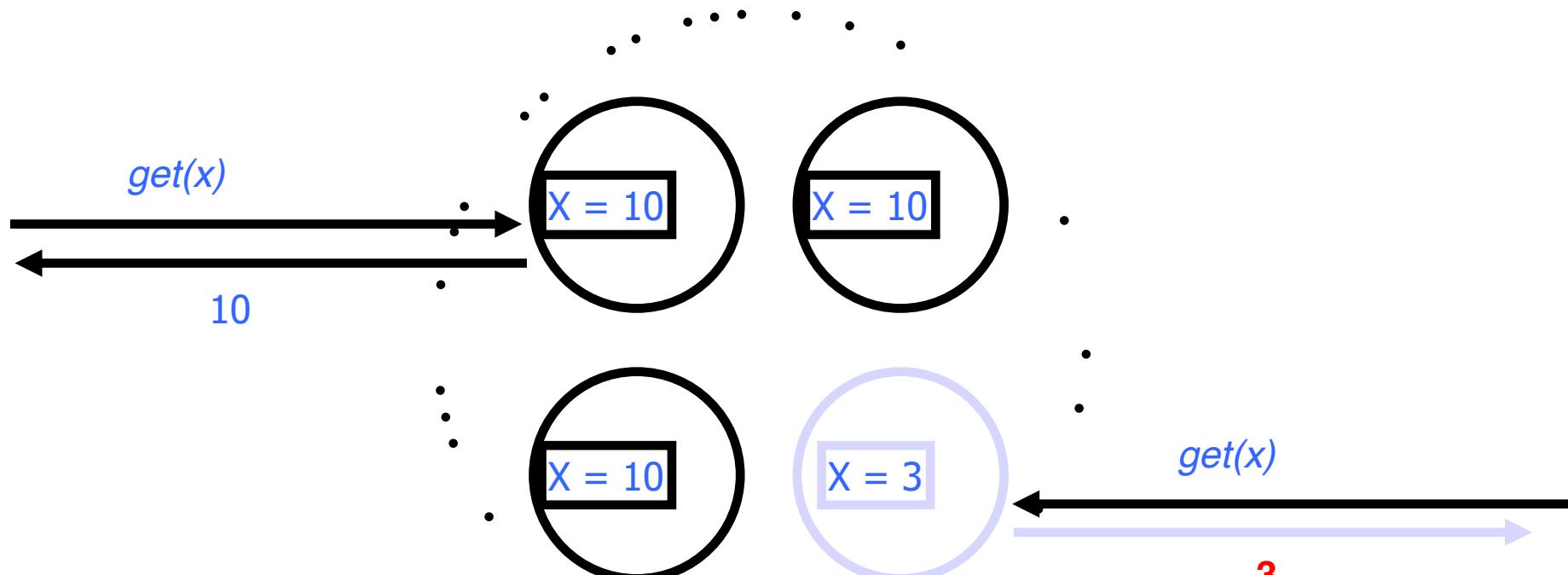
# Write

---

$put(x, 10)$



# Need Agreement

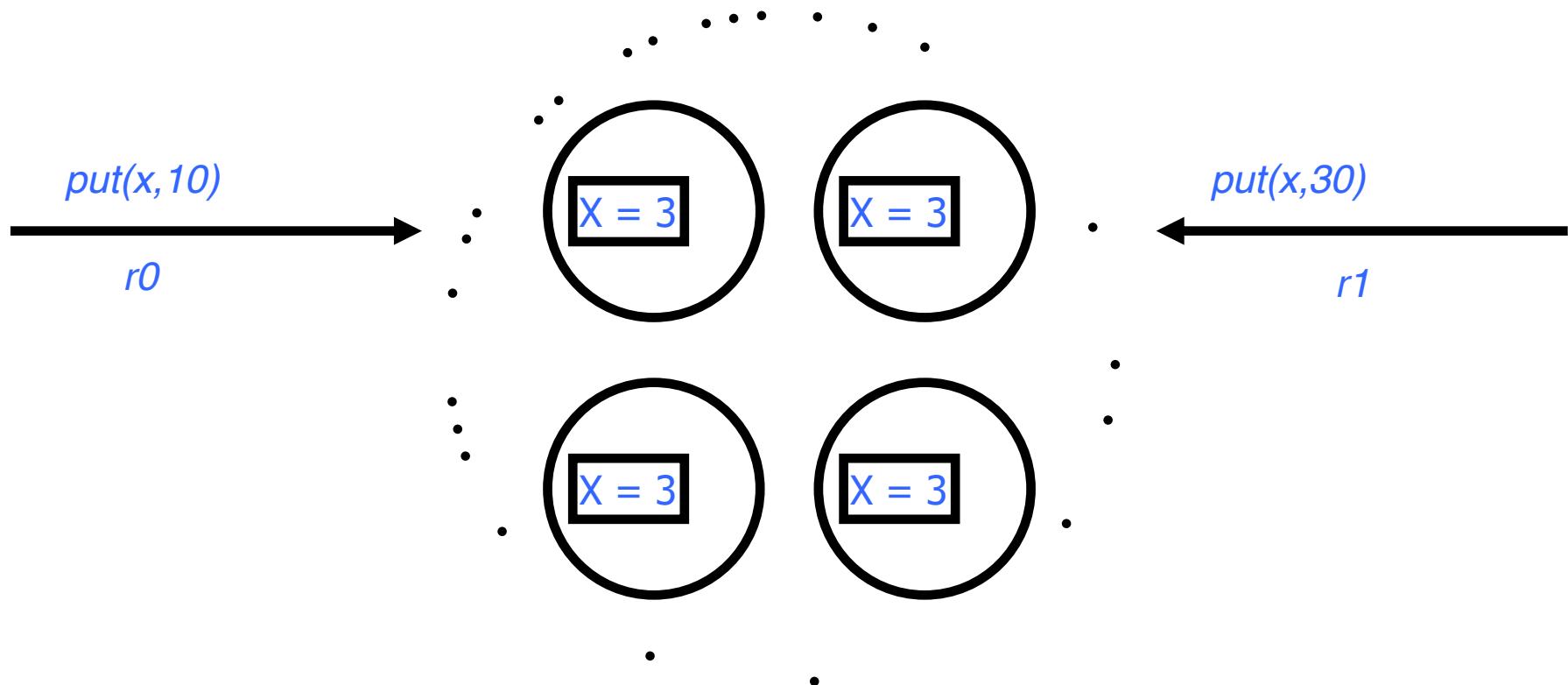


**Problem!**

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

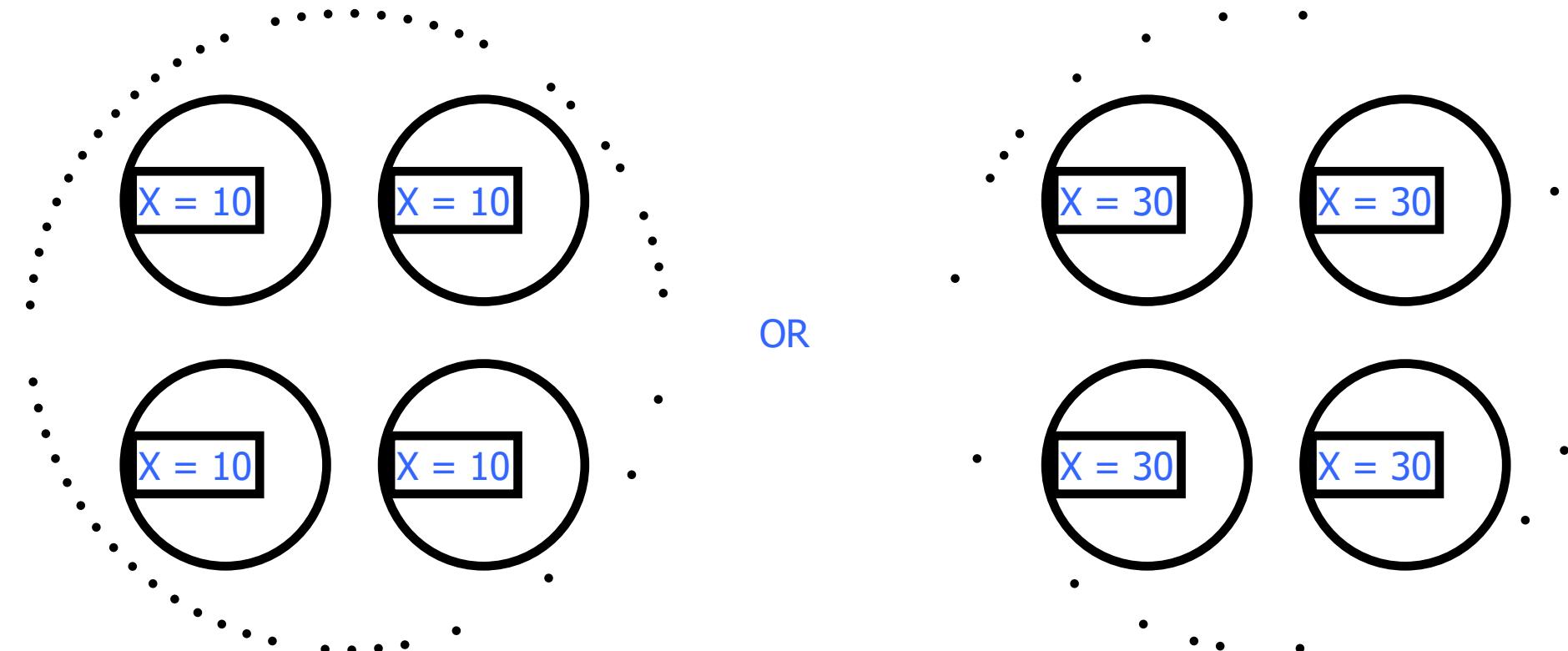
# Two Writes

---



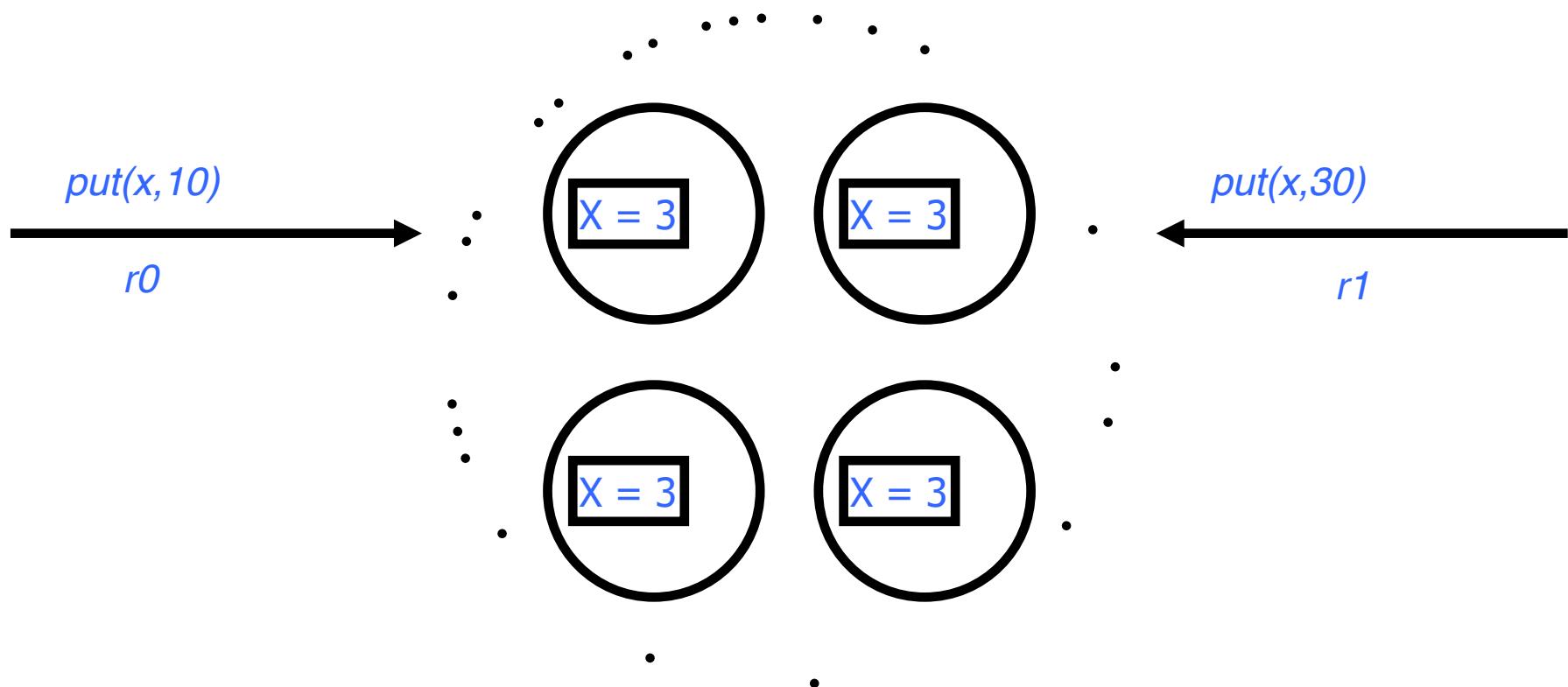
# Either Outcome is Fine

---

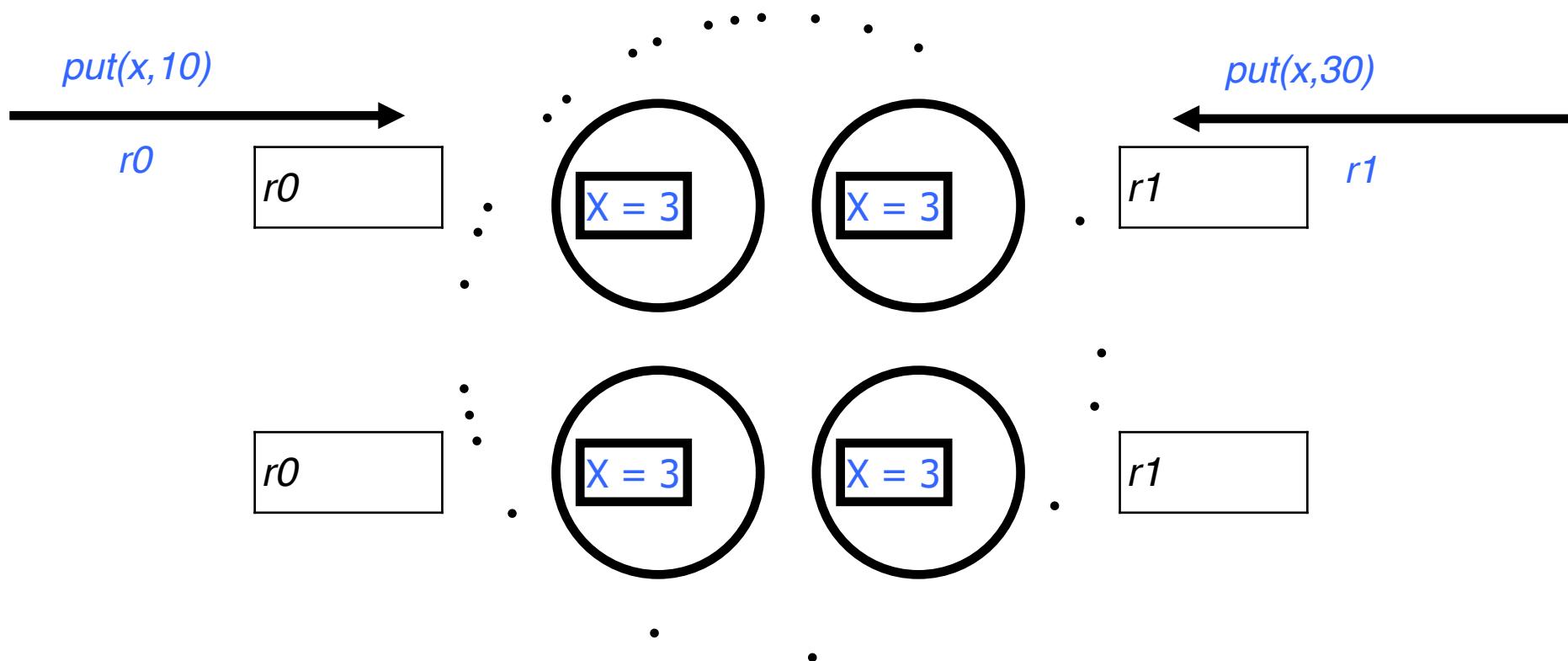


# Order Matters

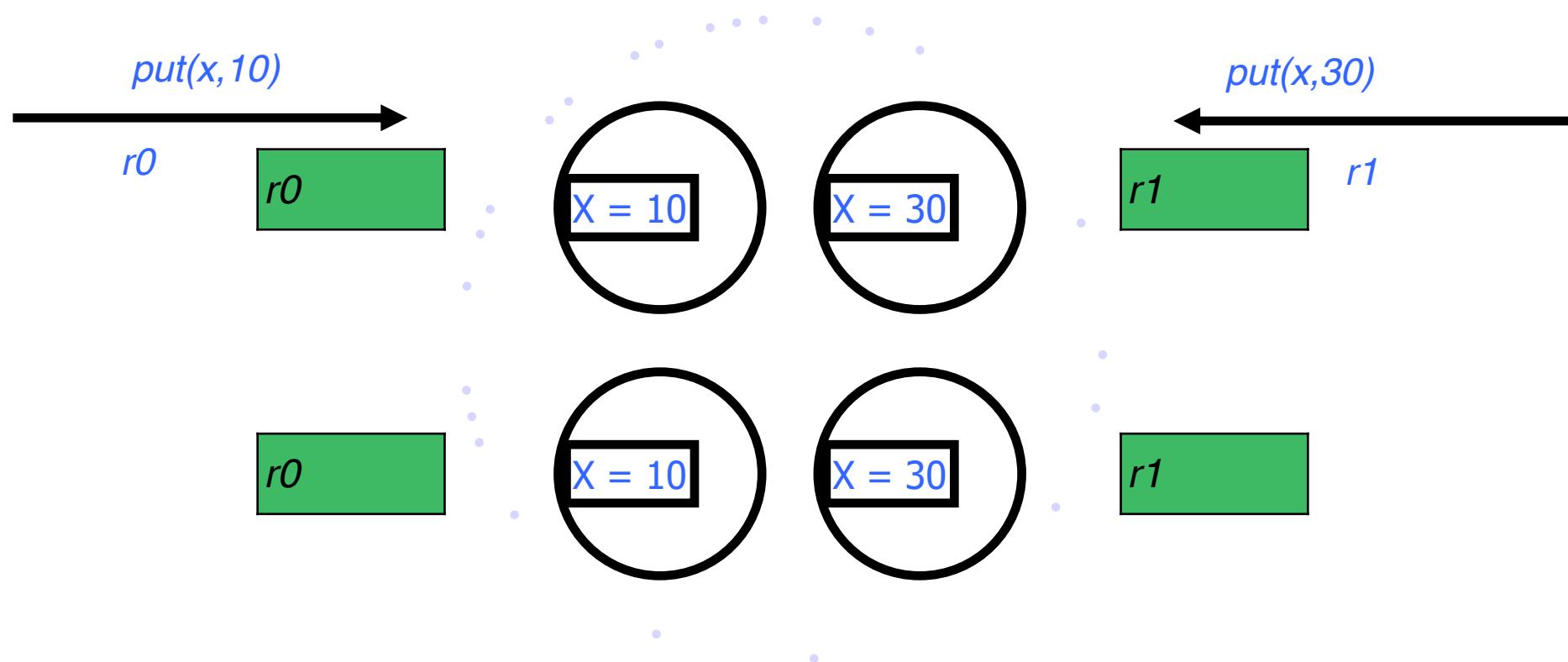
---



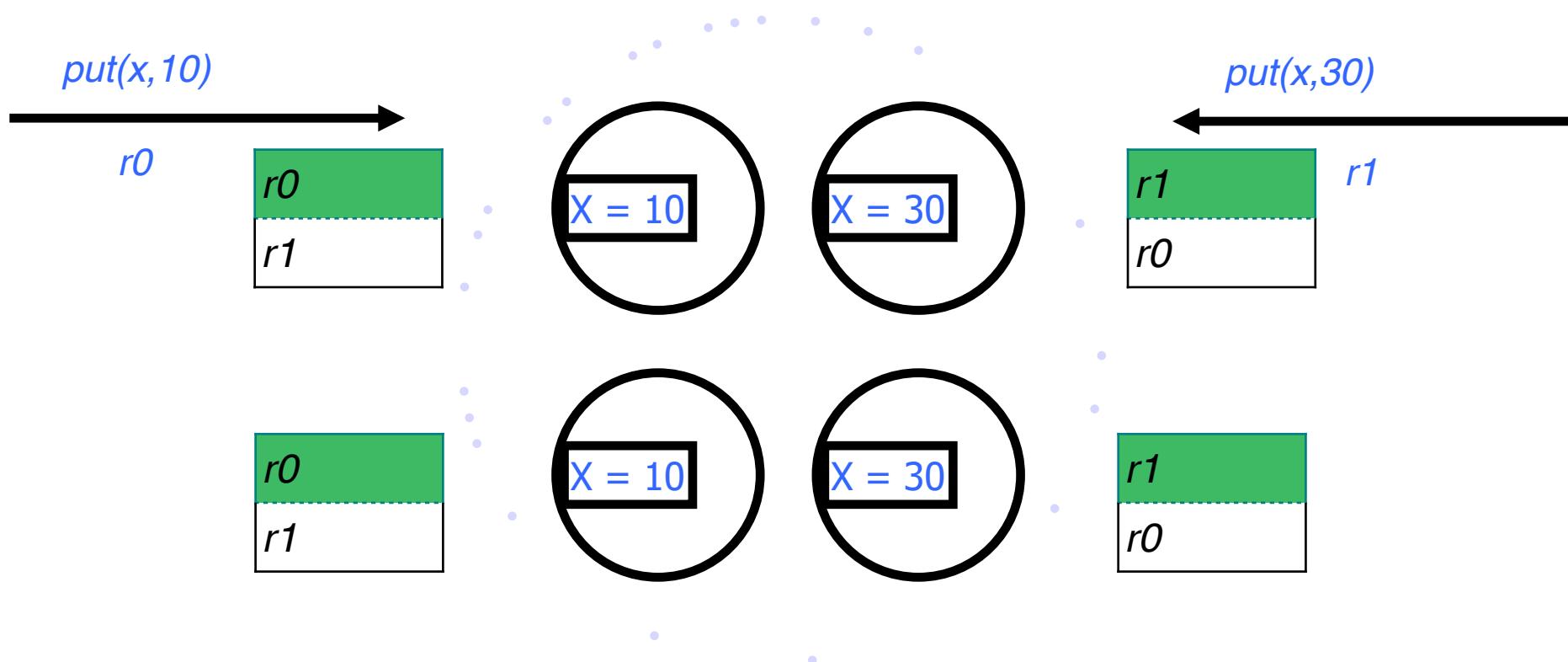
# Order Matters



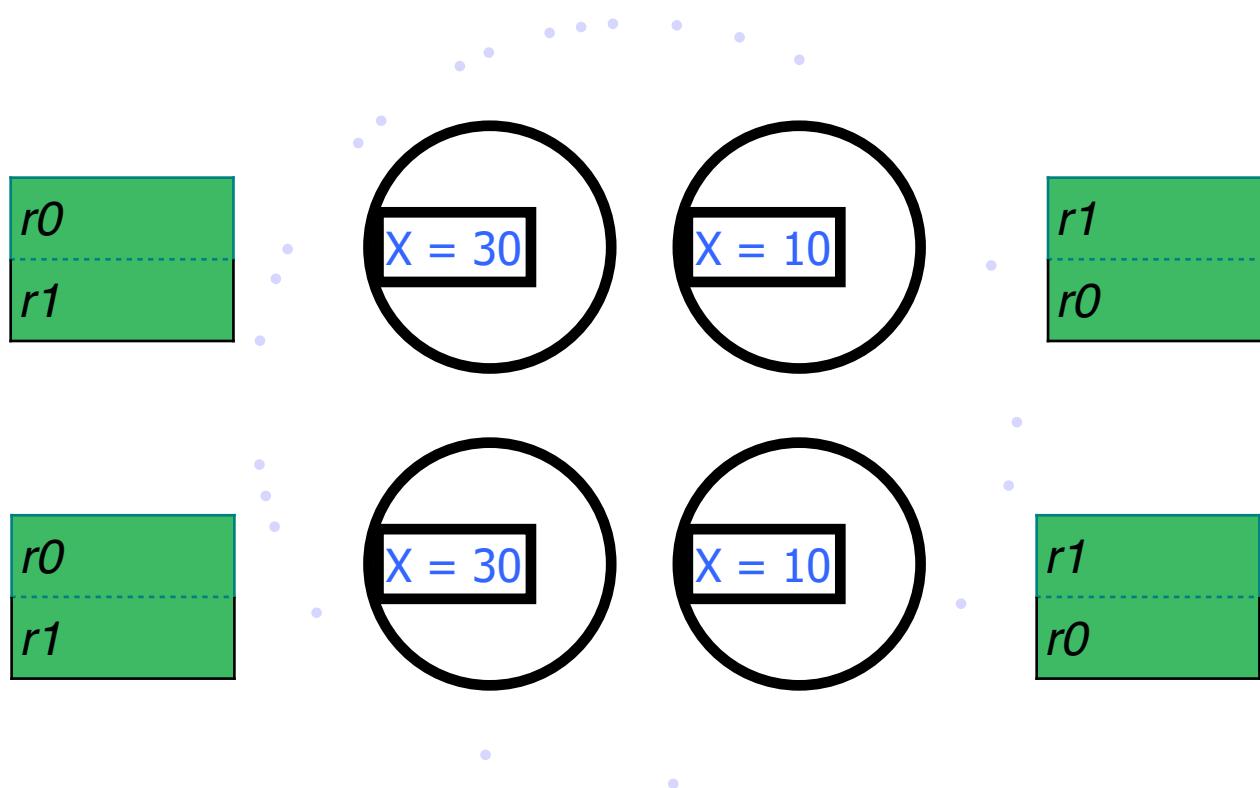
# Order Matters



# 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

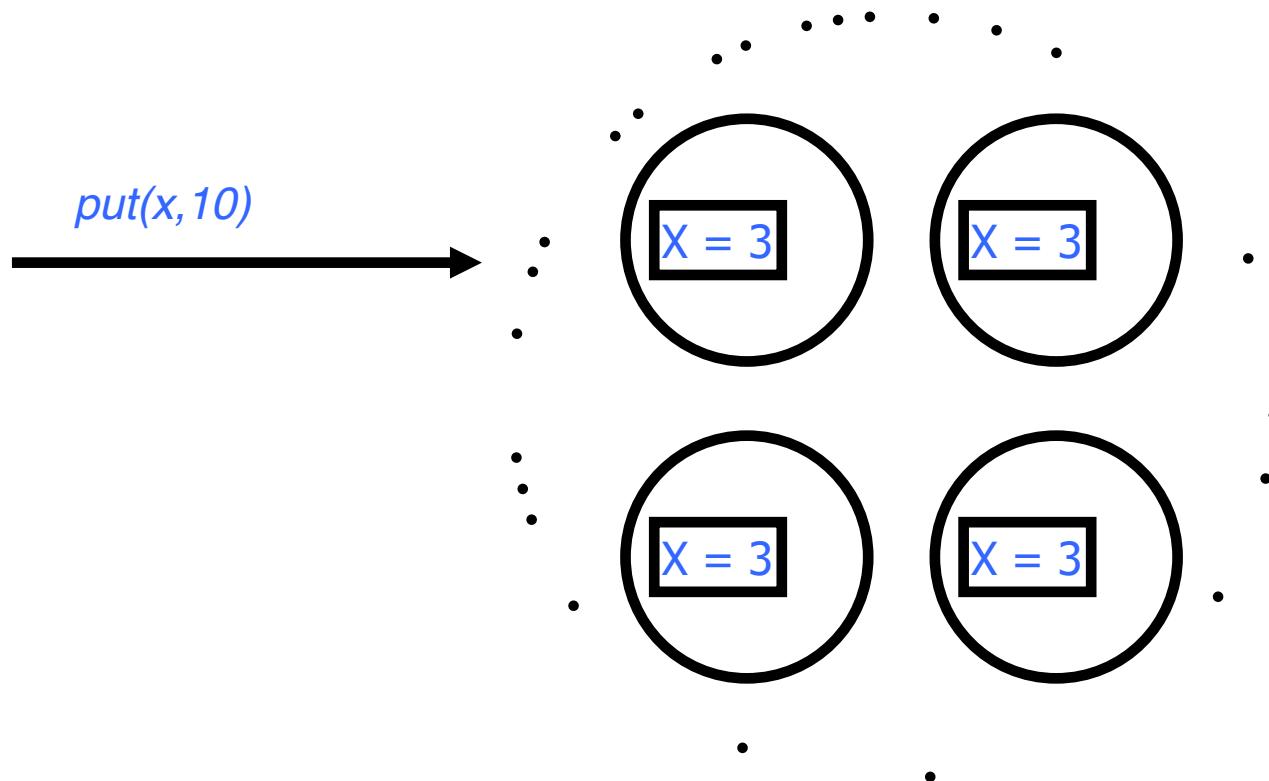
# Idea for Agreement

---

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

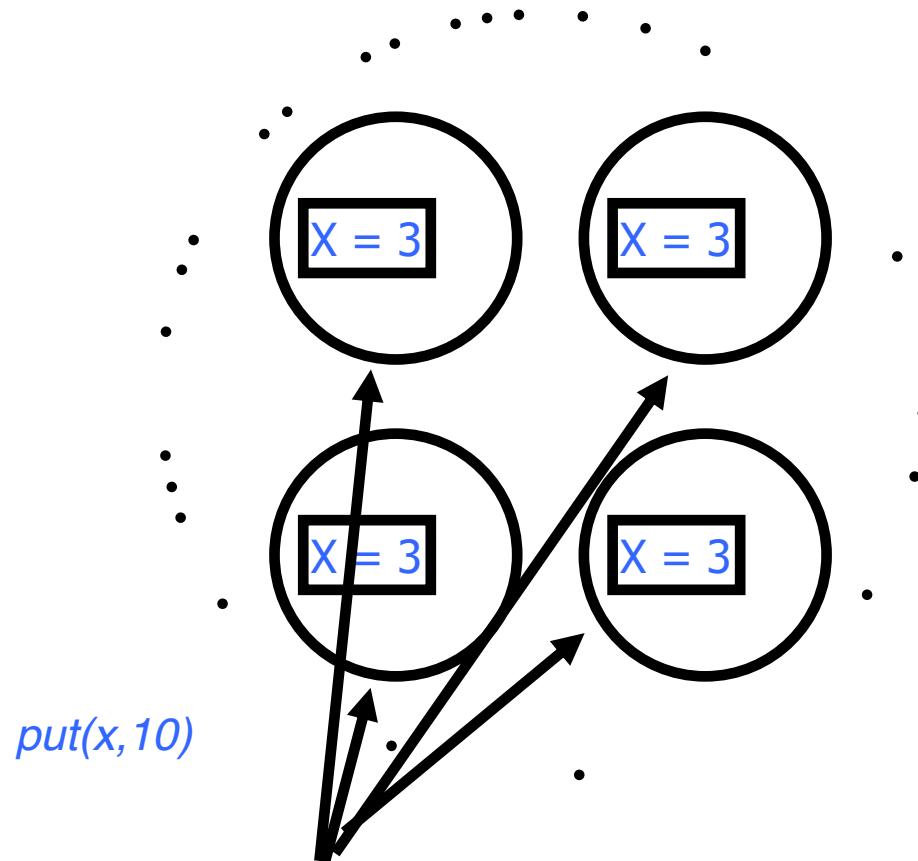
# Agreement

---



# Agreement

---



*Non-faulty Transmitter*

# Idea for Order

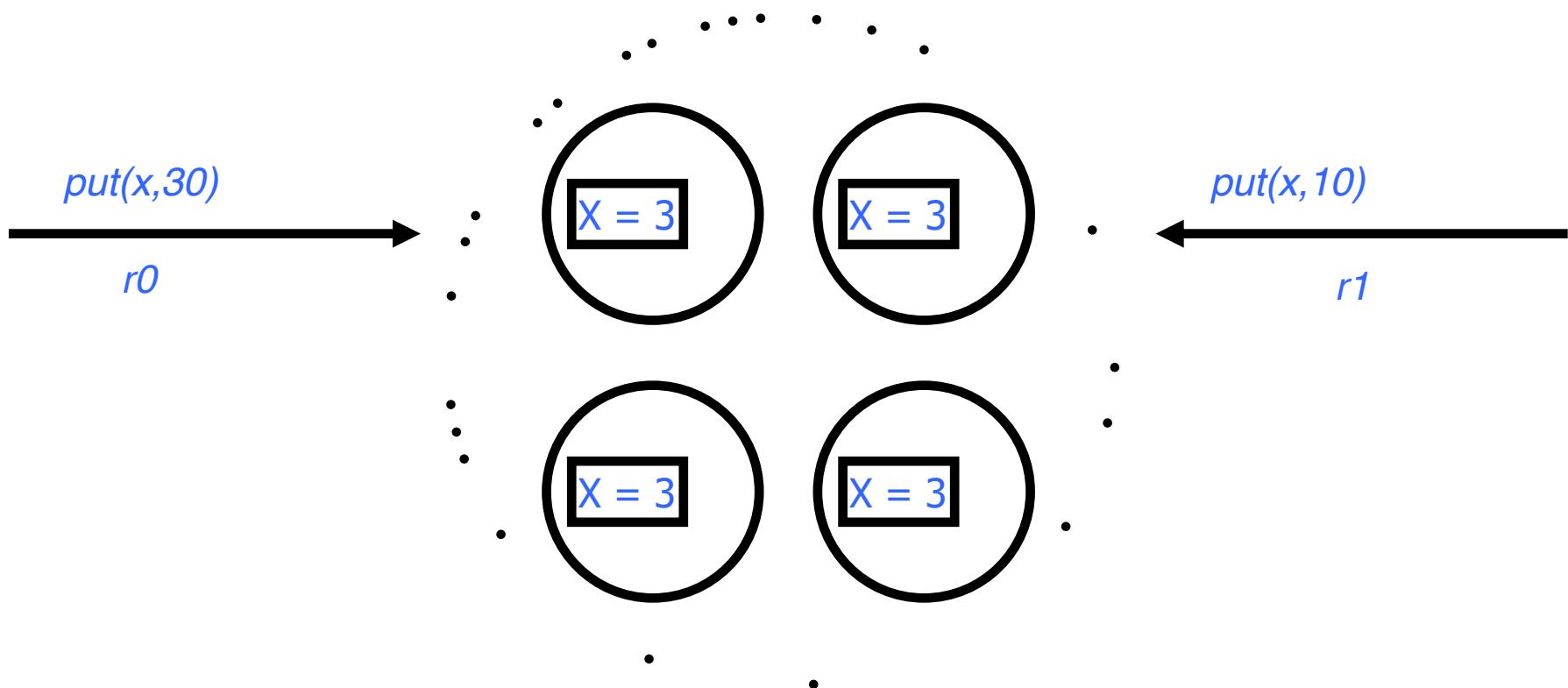
---

Assign unique ids to requests, process them in ascending order

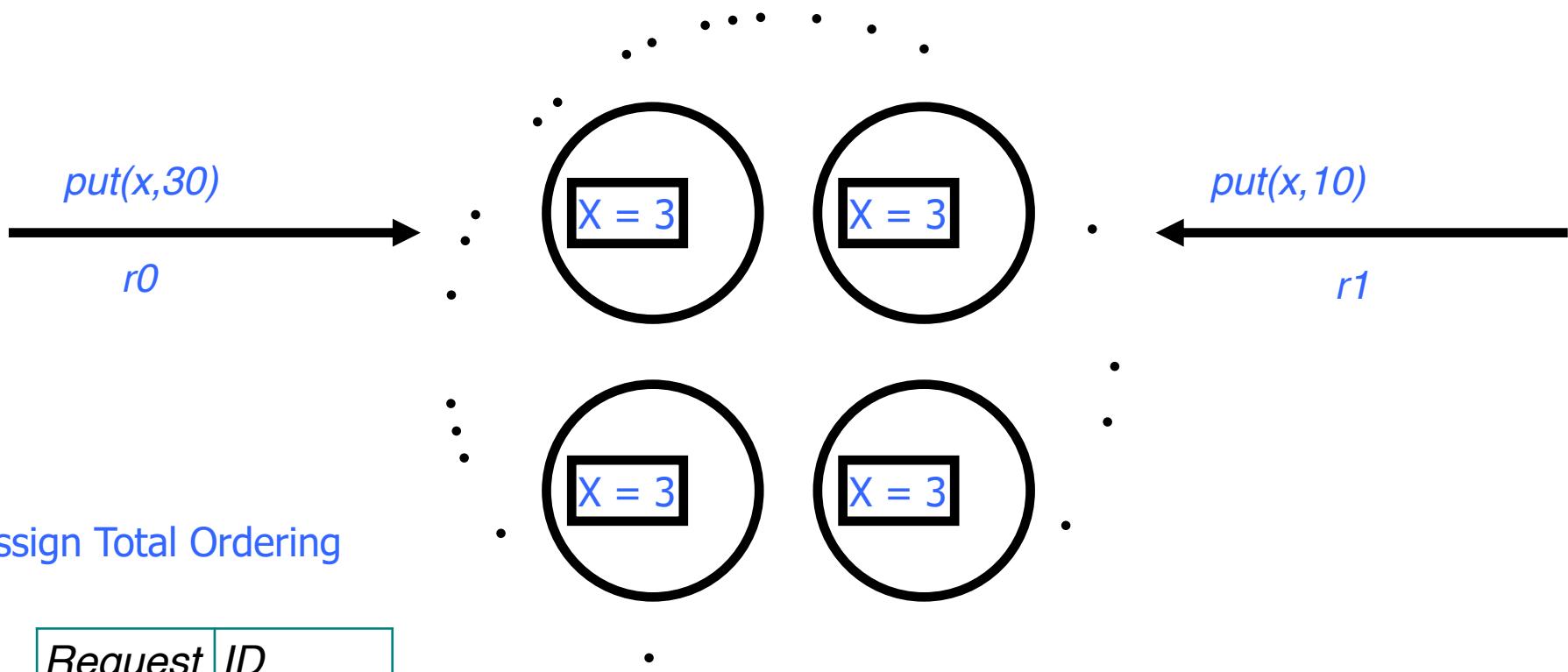
- ◆ How do we assign unique ids in a distributed system?
- ◆ How do we know when every replica has processed a given request?

# Order

---

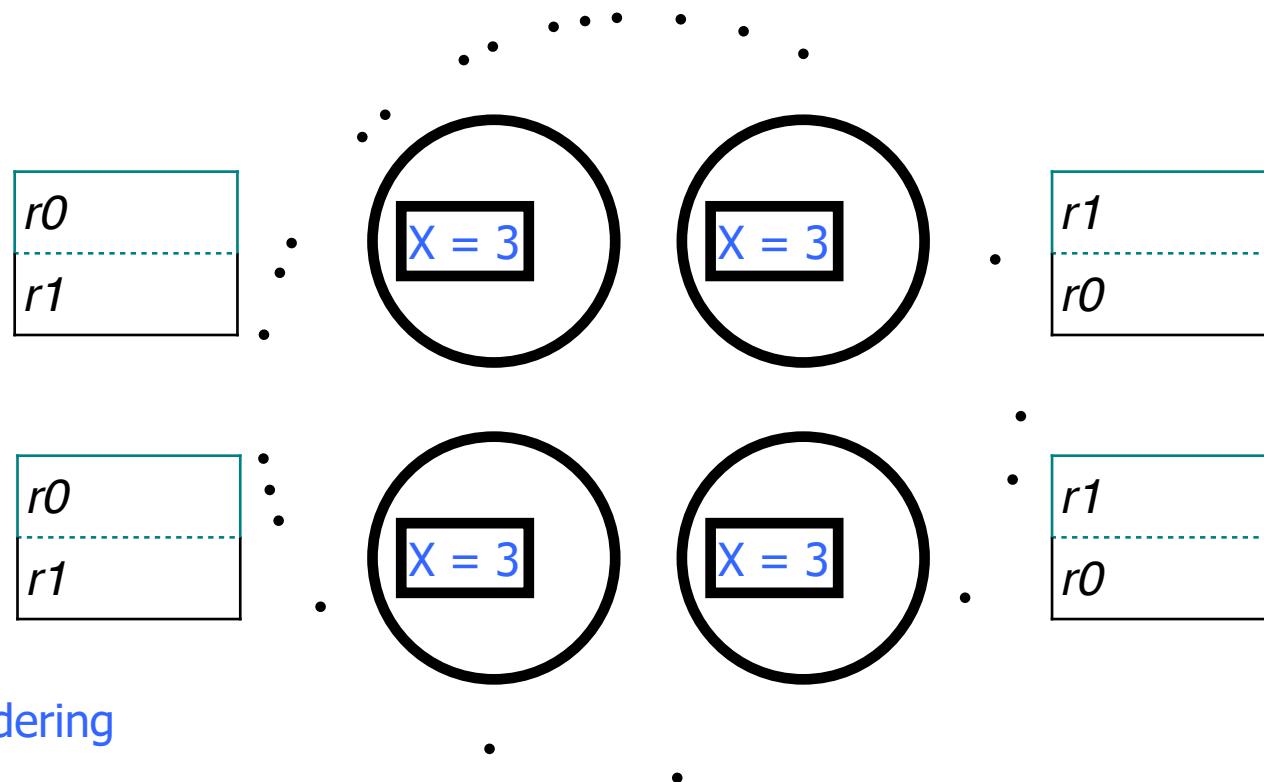


# Order



Request	ID
$r0$	1
$r1$	2

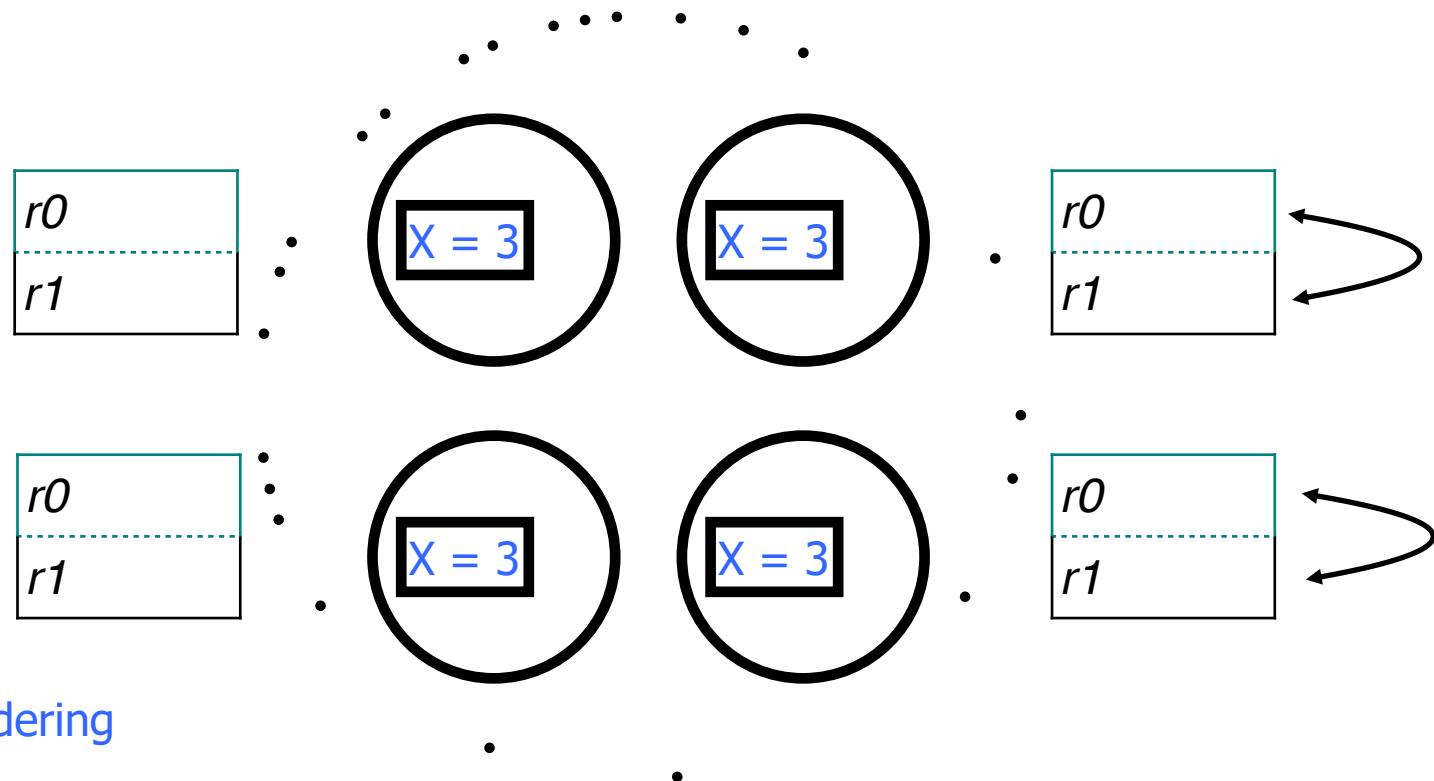
# Order



Assign Total Ordering

Request	ID
$r0$	1
$r1$	2

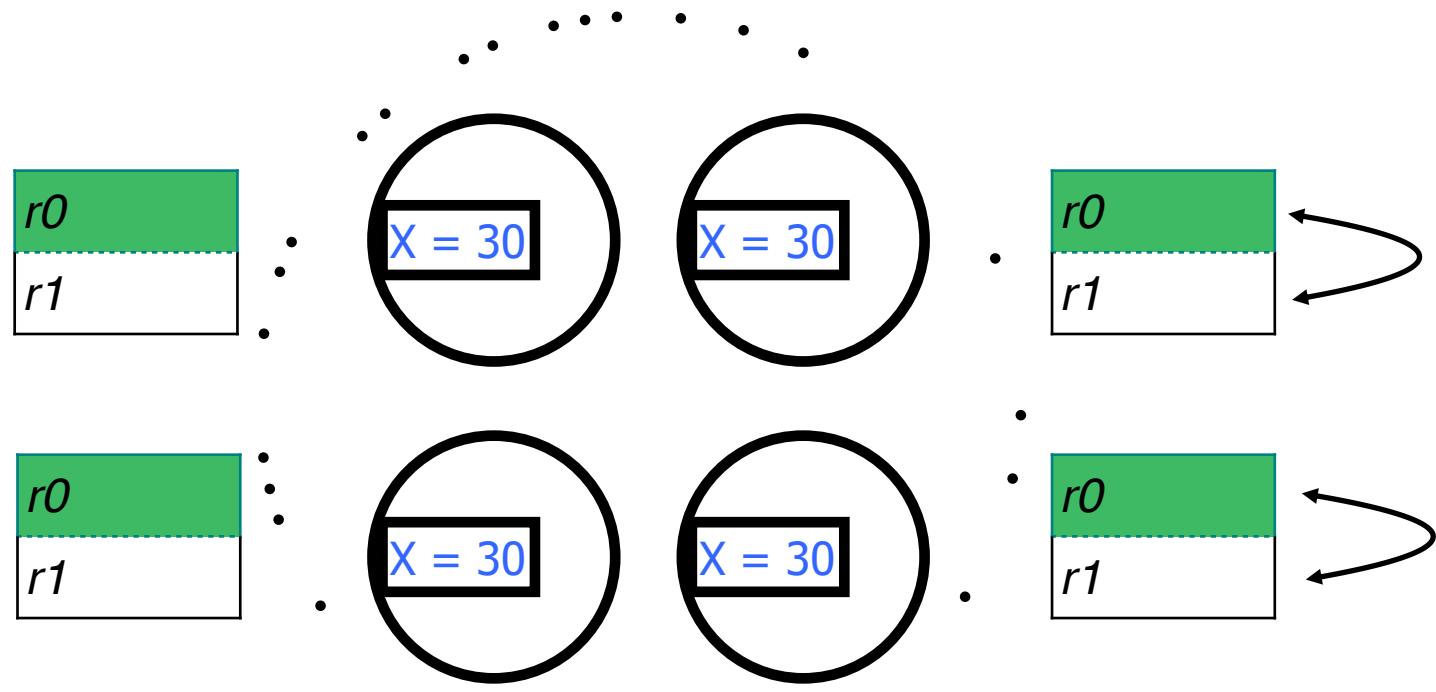
# Order



Assign Total Ordering

Request	ID
$r0$	1
$r1$	2

# Order

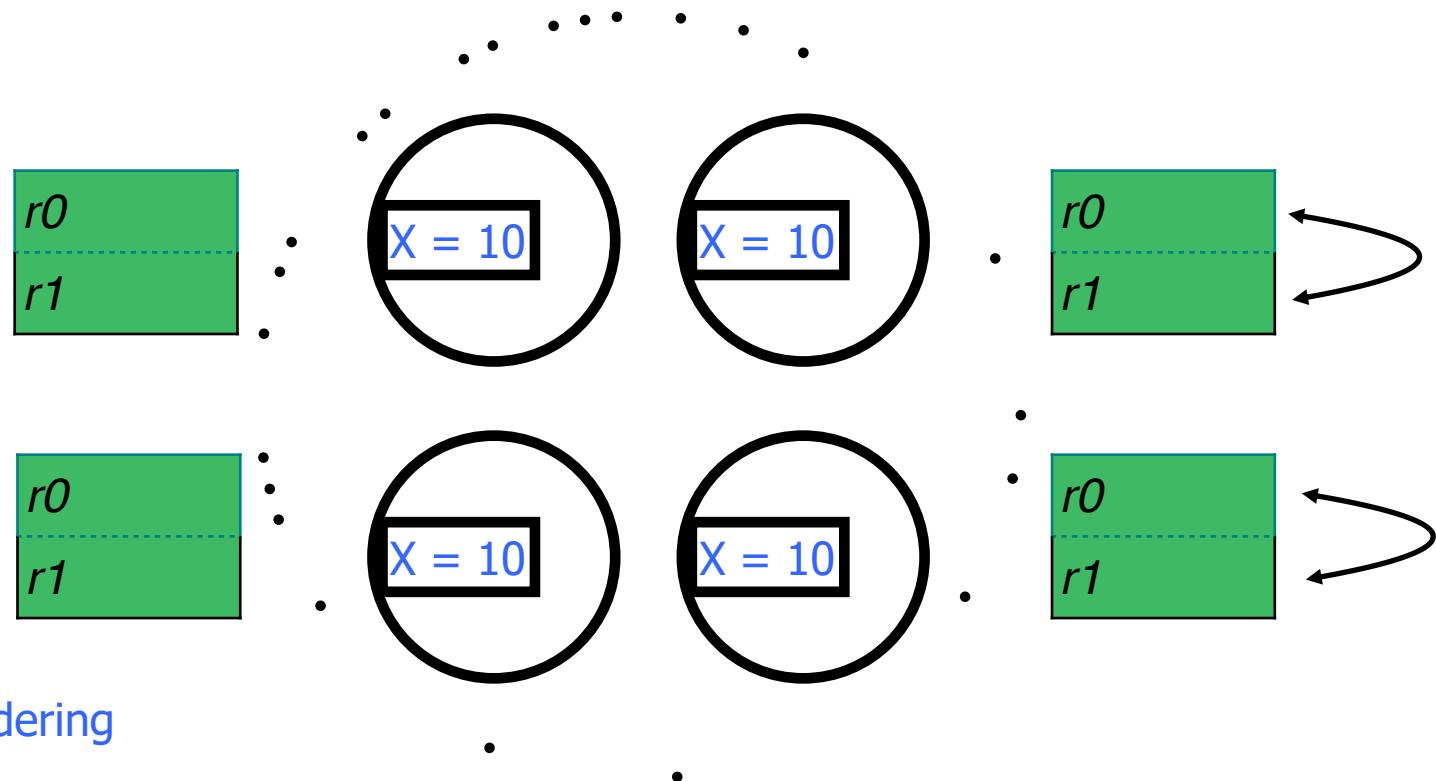


Assign Total Ordering

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

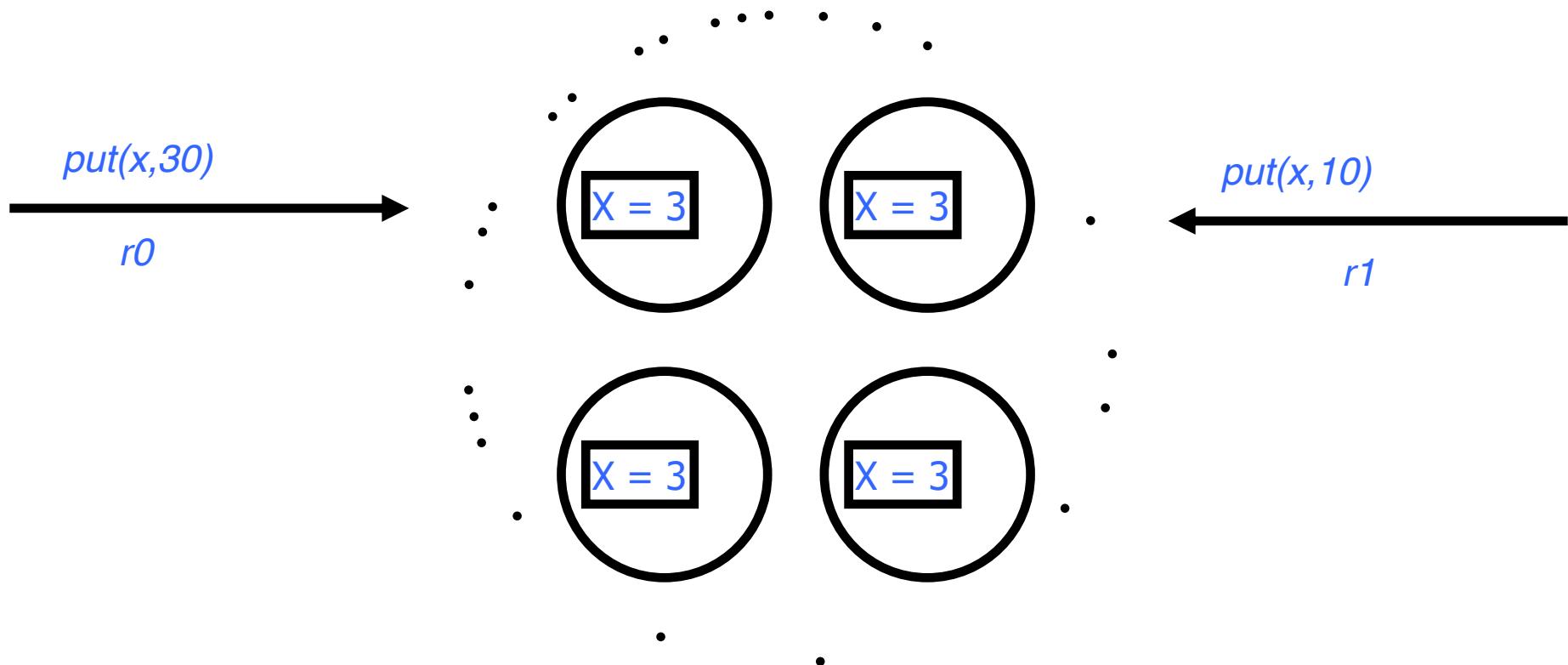
*$r0$  is now stable!*  
 *$r1$  is now stable!*

# Generating IDs

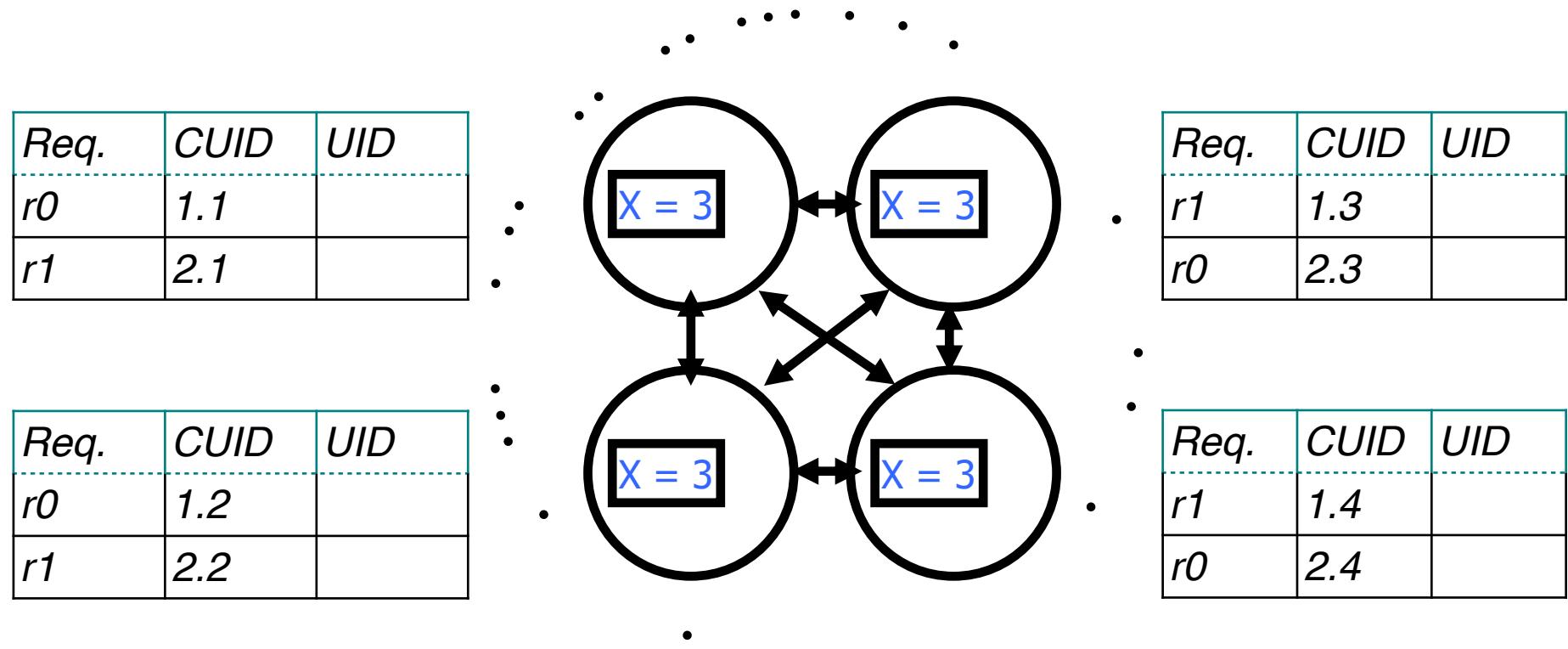
---

- ◆ 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

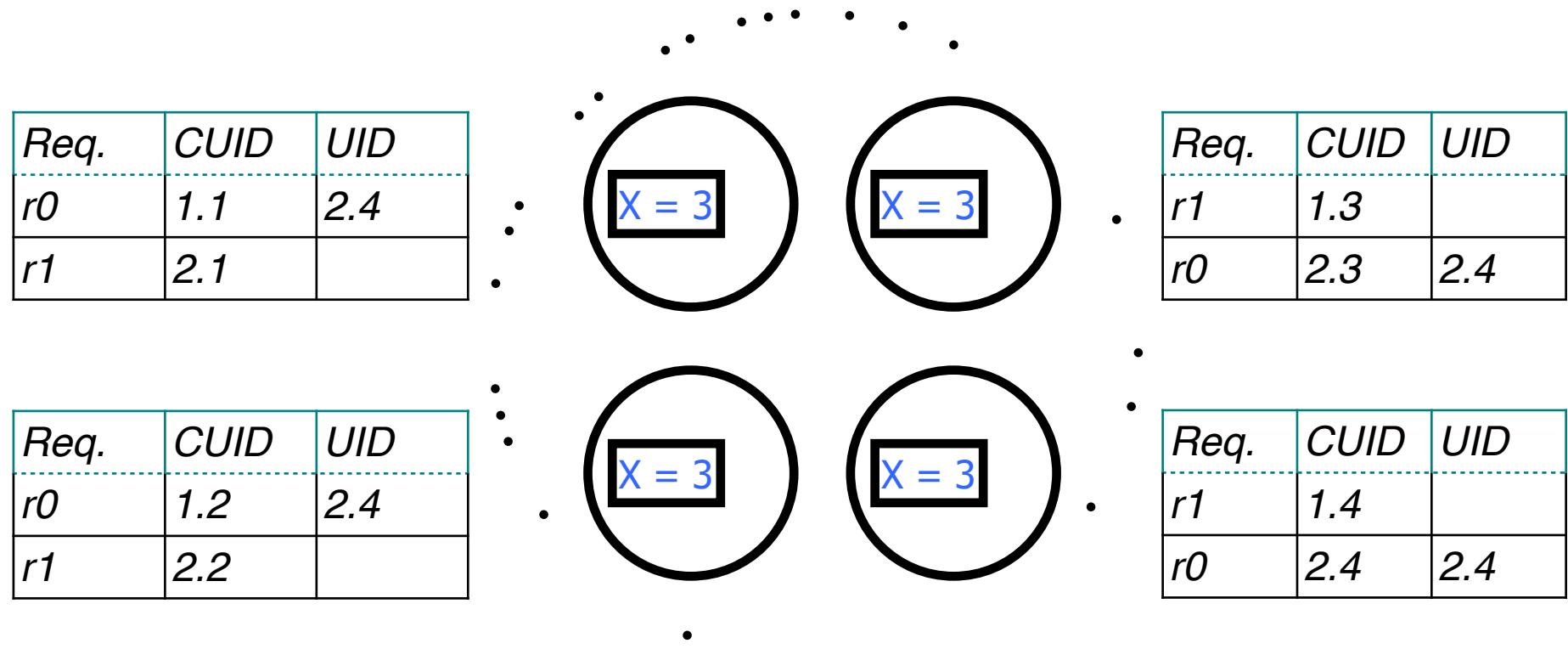


# Replica ID Generation



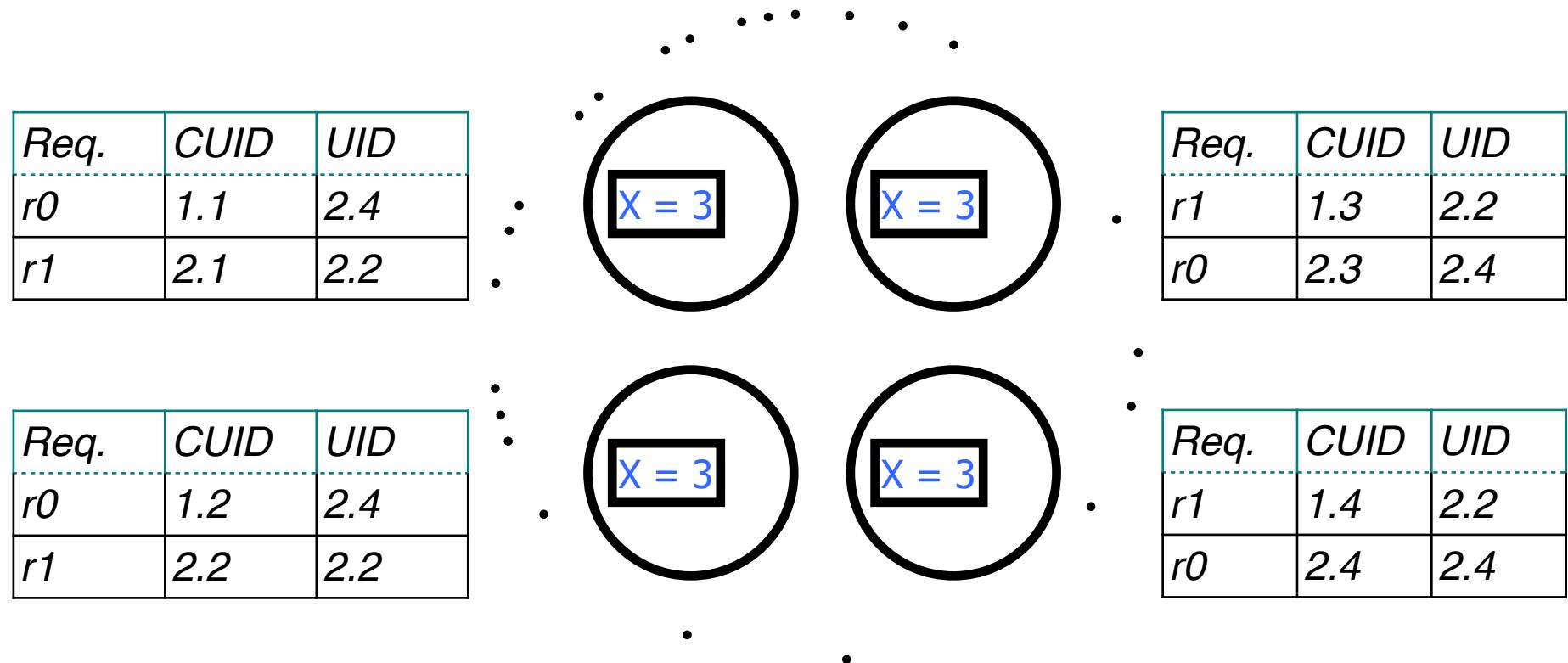
1) Propose candidates

# Replica ID Generation



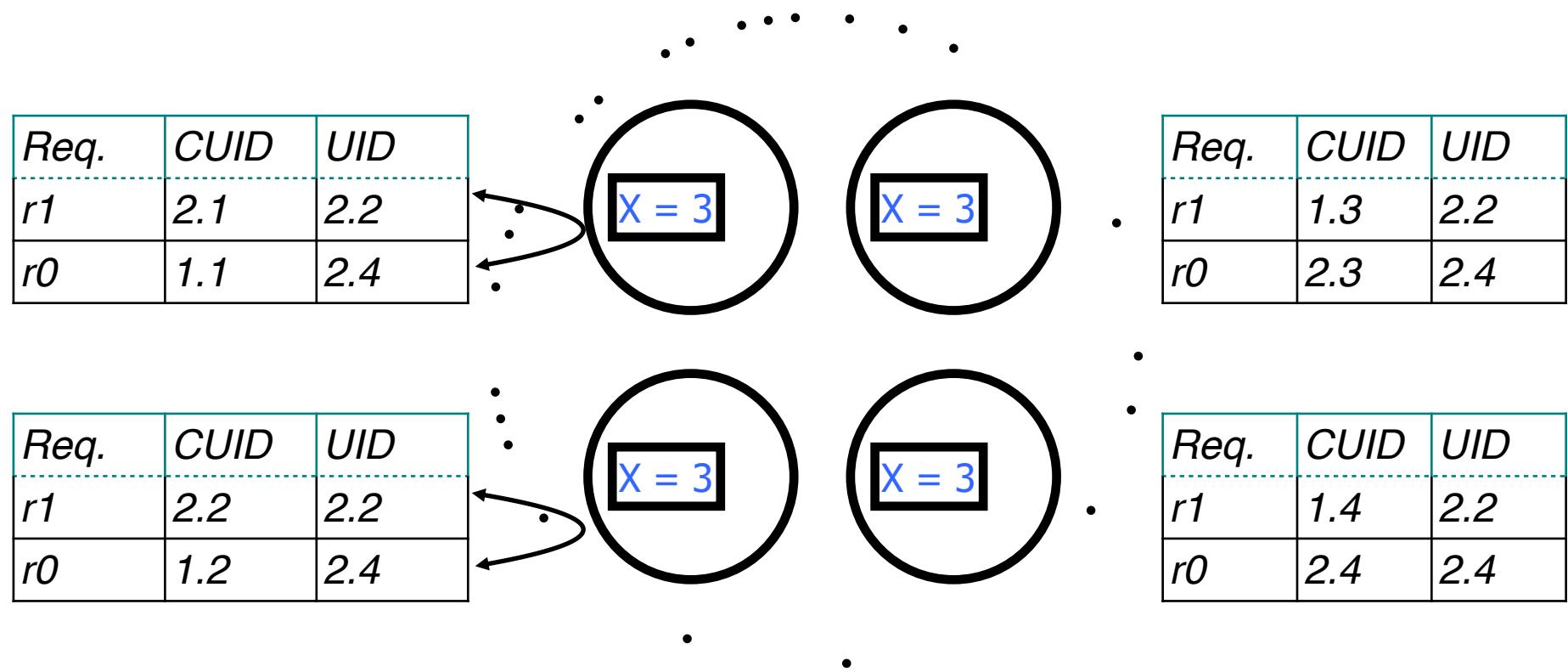
2) Accept r0

# Replica ID Generation



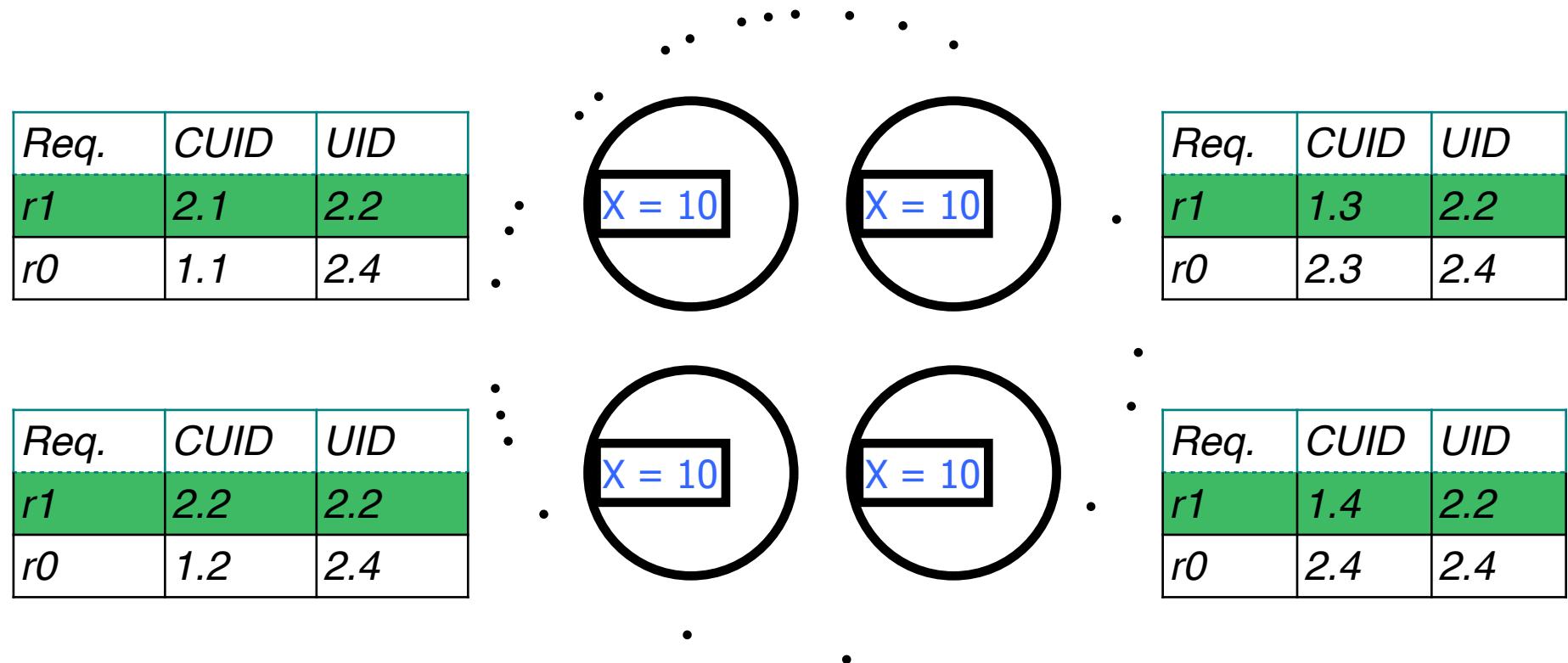
3) Accept r1

# Replica ID Generation



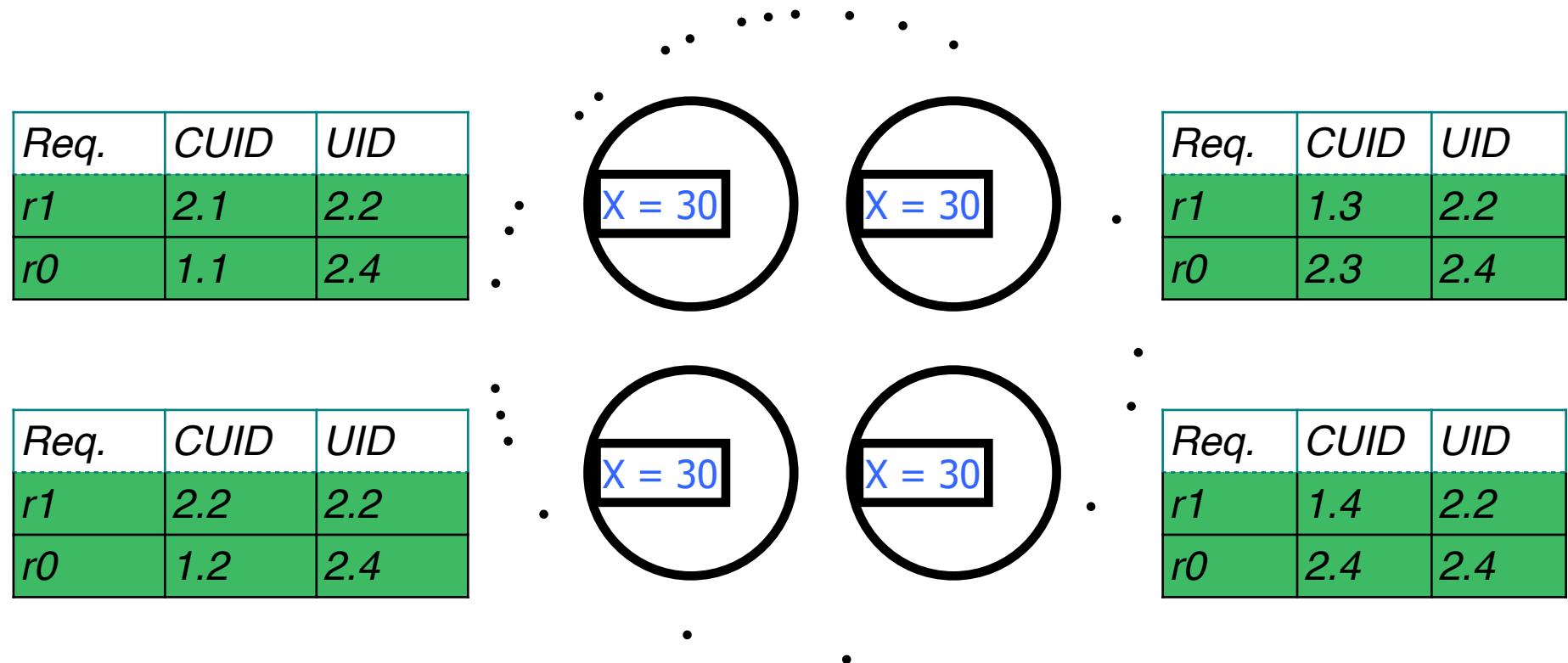
*r1 is now stable*

# Replica ID Generation



4) Apply  $r1$

# Replica ID Generation



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  $\geq$  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

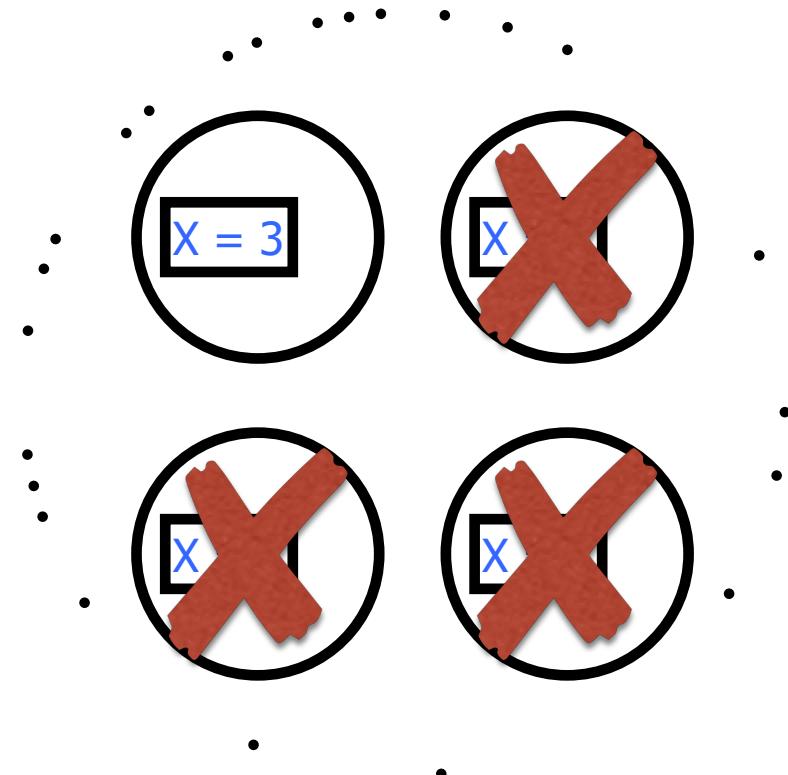
## ◆ Byzantine

- Faulty servers can do arbitrary, perhaps malicious things
- This includes crash failures (server can stop responding without notification)

# Fail-Stop Tolerance

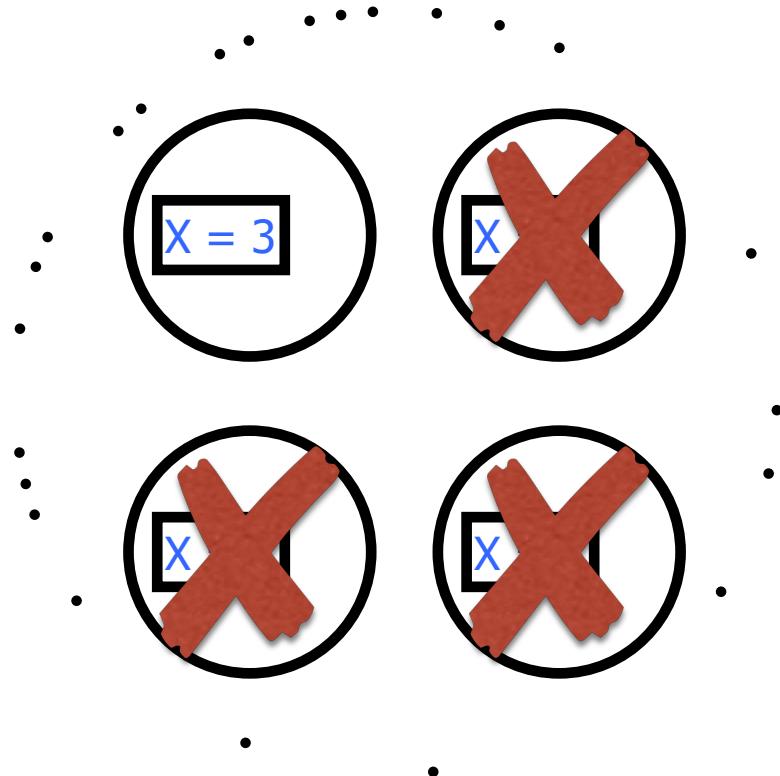
$put(x, 30)$

$r0$



# Fail-Stop Tolerance

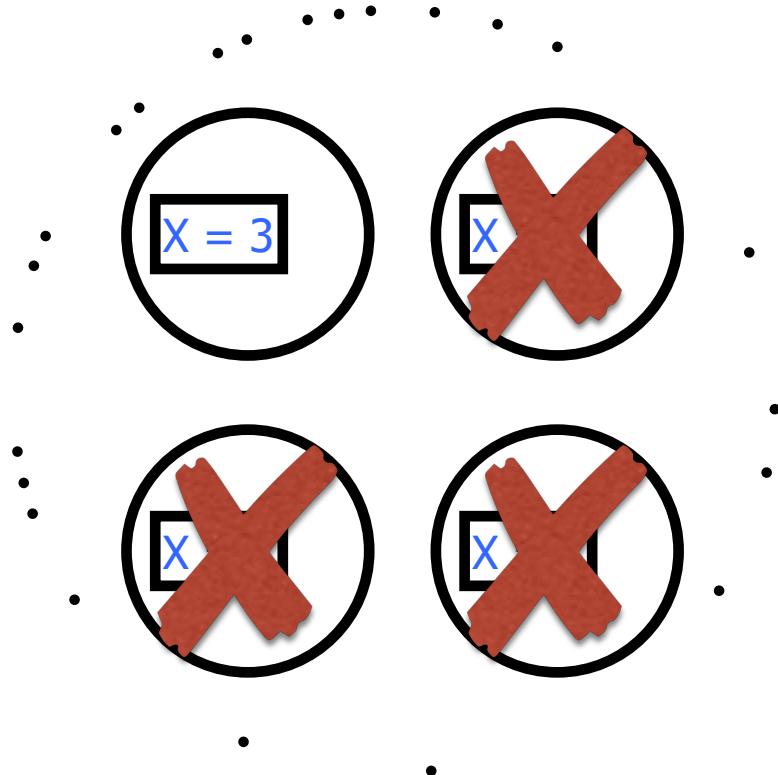
Req.	CUID	UID
r0	1.1	



1) Propose Candidates....

# Fail-Stop Tolerance

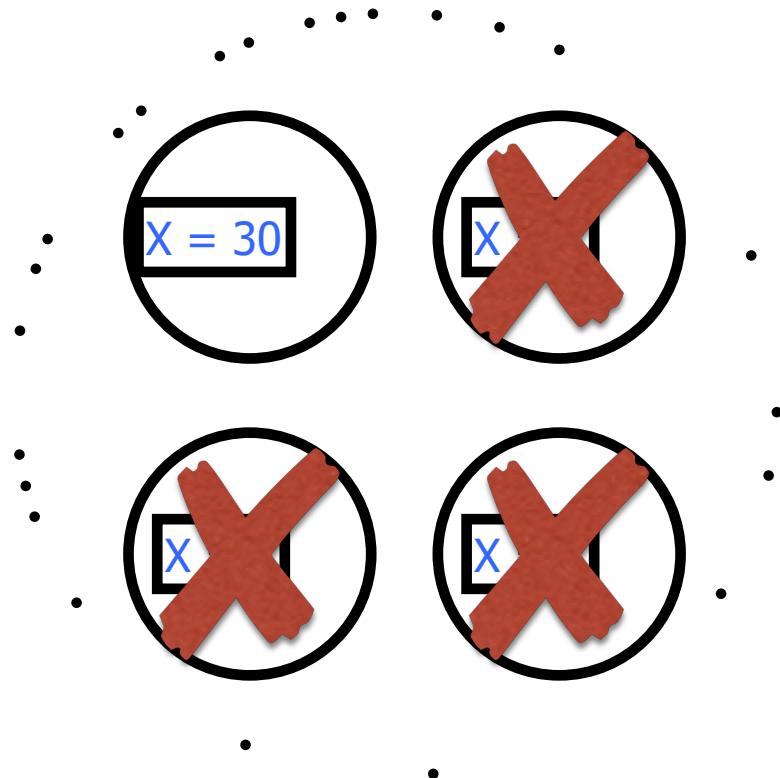
<i>Req.</i>	<i>CUID</i>	<i>UID</i>
<i>r0</i>	1.1	1.1



2) Accept *r0*

# Fail-Stop Tolerance

<i>Req.</i>	<i>CUID</i>	<i>UID</i>
<i>r0</i>	1.1	1.1

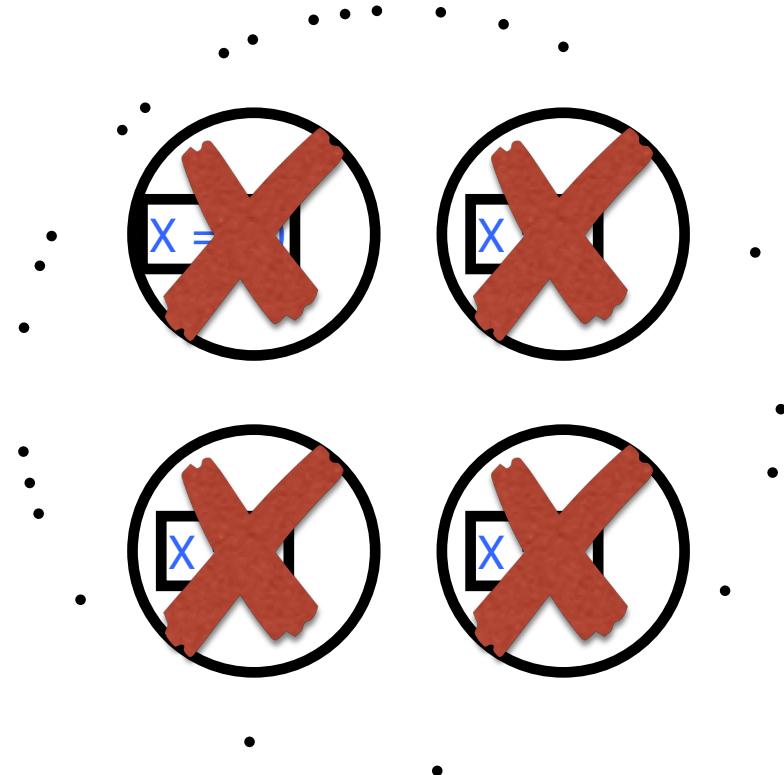


2) Apply *r0*

# Fail-Stop Tolerance

---

GAME OVER!!!



2) Apply  $rO$

# Fail-Stop Fault Tolerance

---

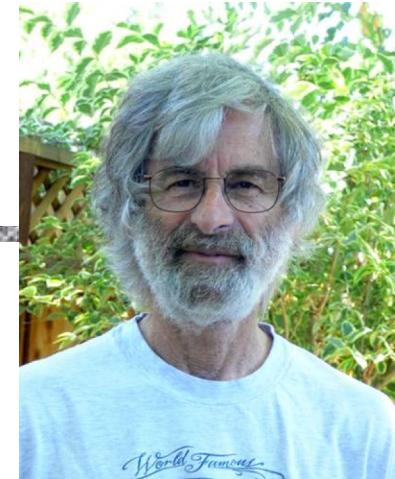
- ◆ To tolerate  $t$  failures, need  $t+1$  servers.
- ◆ As long as 1 server remains, we're OK
- ◆ Only need to participate in protocols with other live servers

# 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