CS 731: Blockchain Technology And Applications

Sandeep K. Shukla

IIT Kanpur

C3I Center

Acknowledgement

- Fred Schneider (Cornell Univ)
- Elli Androulaki et. al (Authors of Hyperledger Fabric paper)

Evolution of Blockchain Technology

- 1st generation: Store and transfer of value (e.g. Bitcoin, Ripple, Dash)
- 2nd generation: Programmable via smart contracts (E.g. Ethereum)
- 3rd generation: Enterprise blockchains (E.g. Hyperledger, R3 Corda & Ethereum Quorum)
- Next gen: Highly scalable with high concurrency

Source: SmartBridge: Glenn Jones, Ken Staker

Order-execute Paradigm

- Many existing smart-contract blockchains follow the blueprint of SMR and implement so-called active replication:
- a protocol for consensus or atomic broadcast first orders the transactions and propagates them to all peers
- each peer executes the transactions sequentially
- order-execute architecture
- it requires all peers to execute every transaction and all transactions to be deterministic.
- The order-execute architecture can be found in virtually all existing blockchain systems,
  - public ones such as Ethereum (with PoW-based consensus)
  - Permissioned ones (with BFT-type consensus) such as Tendermint, Chain and Quorum
- every peer executes every transaction and transactions must
- be deterministic

Other issues with other Blockchains

- Consensus is hard-coded within the platform, which contradicts
  - that there is no “one-size-fits-all” (BFT) consensus protocol
- The trust model of transaction validation is determined by the consensus protocol
  - cannot be adapted to the requirements of the smart contract
- Smart contracts must be written in a fixed, non-standard, or domain-specific language,
  - which hinders wide-spread adoption and may lead to programming errors
- The sequential execution of all transactions by all peers limits performance, and complex measures are needed to prevent denial-of-service attacks against the platform originating from untrusted contracts
  - Such as accounting for runtime with “gas” in Ethereum
- Transactions must be deterministic, which can be difficult to ensure programmatically;
- Every smart contract runs on all peers,
  - which is at odds with confidentiality, and prohibits the dissemination of contract code and state to a subset of peers.

Hyperledger Fabric

- Fabric is the first blockchain system to support the execution of distributed applications written in standard programming languages,
  - allows them to be executed consistently across many nodes, giving impression of execution on a single globally-distributed blockchain computer
- The architecture of Fabric follows a novel ***execute-order-validate*** paradigm for distributed execution of untrusted code in an untrusted environment.
- It separates the transaction flow into three steps, which may be run on different entities in the system:
  - executing a transaction and checking its correctness, thereby endorsing it (corresponding to “transaction validation” in other blockchains);
  - ordering through a consensus protocol, irrespective of transaction semantics
  - transaction validation per application specific trust assumptions, which also prevents race conditions due to concurrency.

Order-Execute Paradigm of Most Blockchains

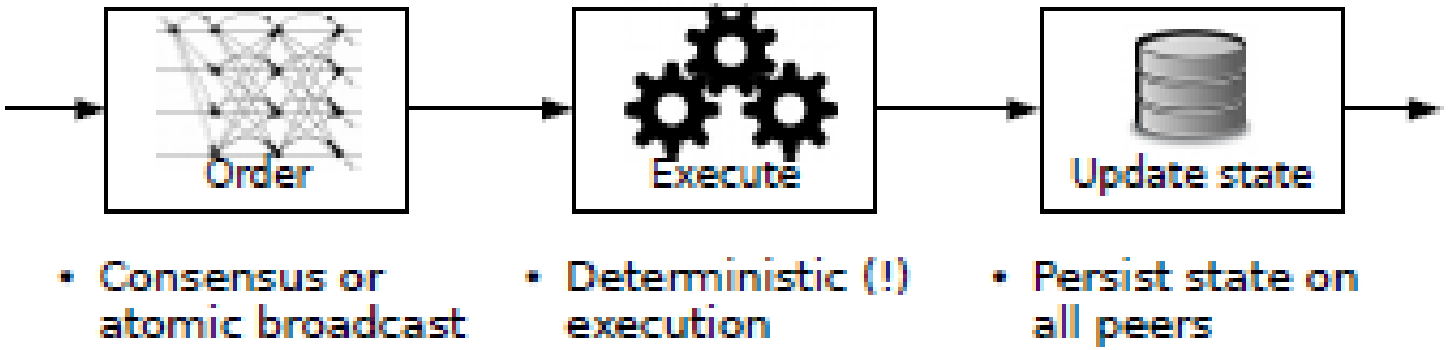


Figure 1: Order-execute architecture in replicated services.

Order-Execute in PoW blockchain

- PoW-based permissionless blockchain such as Ethereum combines consensus and execution of transactions as follows:
  - (1) every peer (i.e., a node that participates in consensus) assembles a block containing valid transactions
    - (to establish validity, this peer already pre-executes those transactions)
  - (2) the peer tries to solve a PoW puzzle
  - (3) if the peer is lucky and solves the puzzle, it disseminates the block to the network via a gossip protocol
  - (4) every peer receiving the block validates the solution to the puzzle and all transactions in the block
- Effectively, every peer repeats the execution of the lucky peer from its first step.
- All peers execute the transactions sequentially (within one block and across blocks)

Limitations of Order-Execute Paradigm

- **Sequential execution** Executing the transactions sequentially on all peers limits the effective throughput
  - In contrast to traditional SMR, the blockchain forms a universal computing engine and its payload applications might be deployed by an adversary. A denial-of-service (DoS) attack:
    - could simply introduce smart contracts that take a very long time to execute.
    - a smart contract that executes an infinite loop
  - To cope with this problem, public programmable blockchains with a cryptocurrency account for the execution cost
    - Ethereum Gas

Limitations of Order-Execute

- **Non-deterministic code**. Operations executed after consensus in SMR must be deterministic, or the distributed ledger “forks” and violates the basic premise of a blockchain, that all peers hold the same state.
  - This is usually addressed by programming blockchains in domain-specific languages
  - E.g Ethereum Solidity
  - Requires additional learning by the programmer.
  - Writing smart contracts in a general-purpose language (e.g., Go, Java, C/C++) instead accelerates the adoption of blockchain solutions

Limitations of Order-execute

- **Confidentiality of execution:** Usually they run all smart contracts on all peers.
- Many intended use cases for permissioned blockchains require confidentiality,
  - i.e., that access to smart contract logic, transaction data, or ledger state can be restricted.
- cryptographic techniques, ranging from data encryption to advanced zero-knowledge proofs and verifiable computation can help to achieve confidentiality but
  - comes with a considerable overhead
- it suffices to propagate the same state to all peers instead of running the same code everywhere.
  - Execution of a smart contract can be restricted to a subset of the peers trusted for this task, that vouch for the results of the execution.

Limitations of Order-Execute

- **Fixed trust model:** Most permissioned blockchains rely on asynchronous BFT replication protocols to establish consensus
  - Such protocols typically rely on a security assumption that among n > 3f peers, up to f are tolerated to misbehave and exhibit Byzantine faults
- The same peers often execute the applications as well, under the same security assumption
  - even though one could actually restrict BFT execution to fewer peers
- such a quantitative trust assumption, irrespective of peers' roles in the system, may not match the trust required for smart contract execution
  - trust at the application level should not be fixed to trust at the protocol level.
- A general purpose blockchain should decouple these two assumptions and permit flexible trust models for applications.

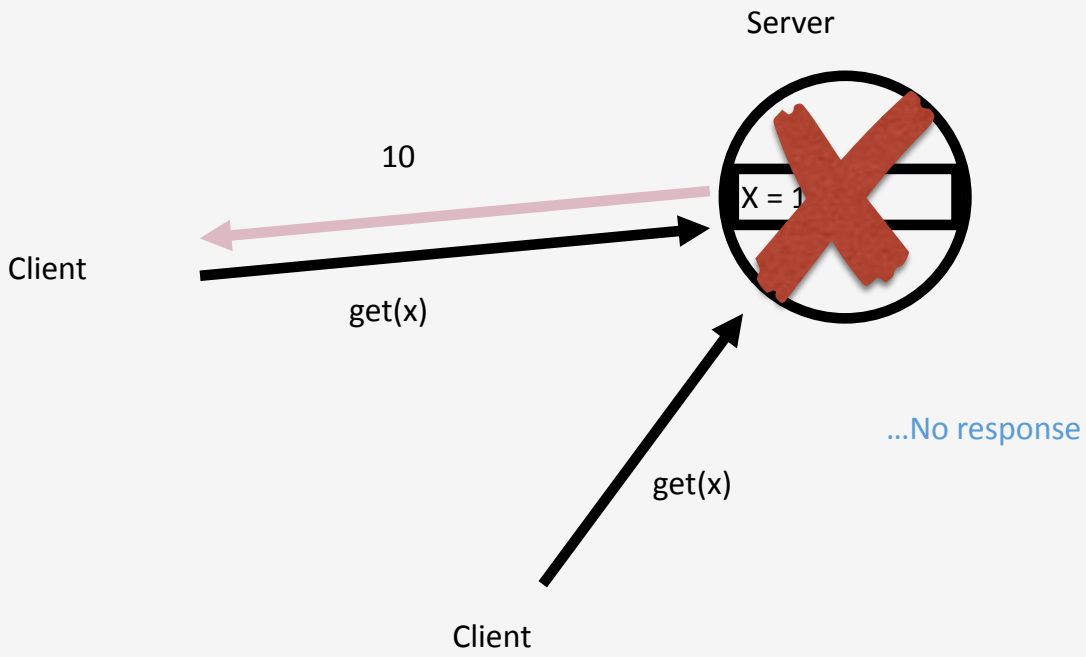
Limits of Order-execute

- **Hard-coded consensus:** all blockchain systems, permissioned or not, came with a hard-coded consensus protocol
- one may want to replace BFT consensus with a protocol based on an alternative trust such as Paxos/Raft and ZooKeeper (Kafka)

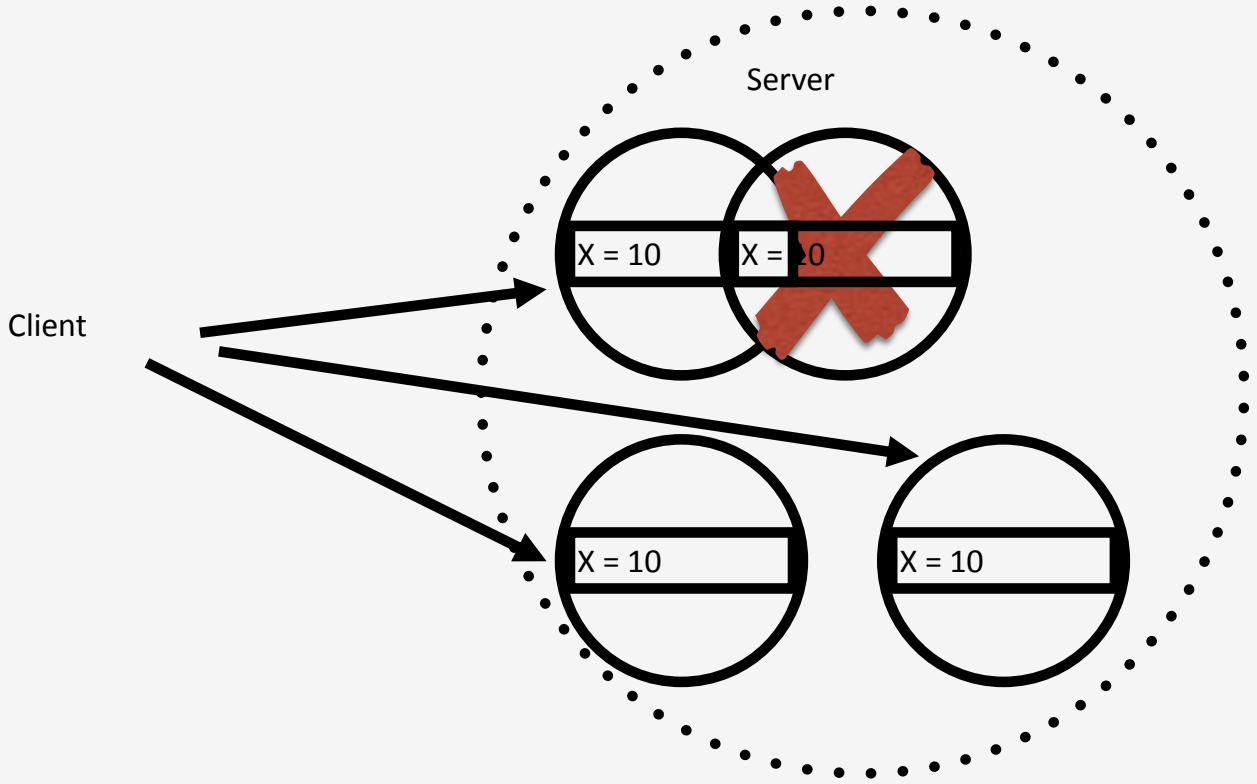
SMR (State Machine Replication Model)

- Can represent ***deterministic*** distributed system as *Replicated State Machine*
- 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



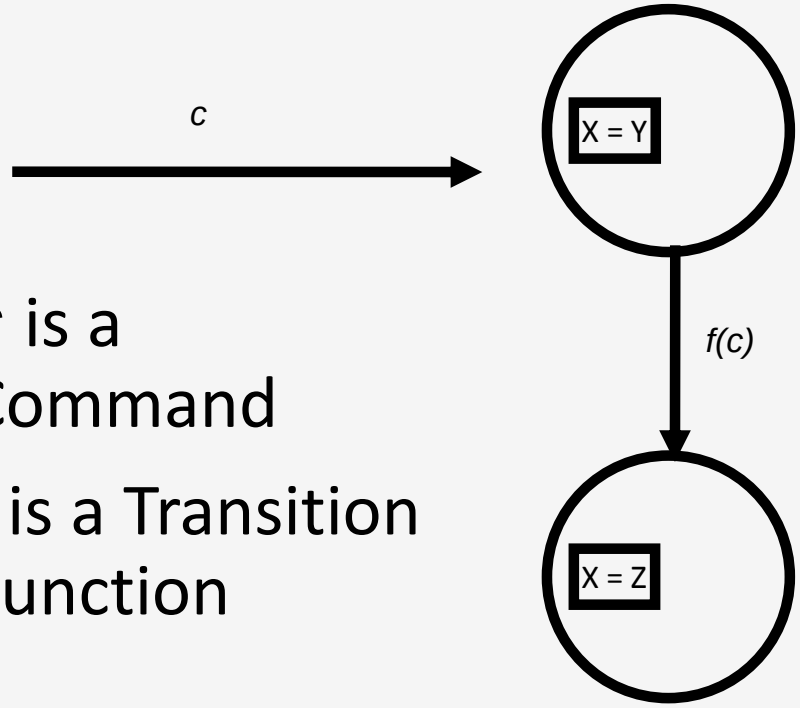
Motivation



Motivation

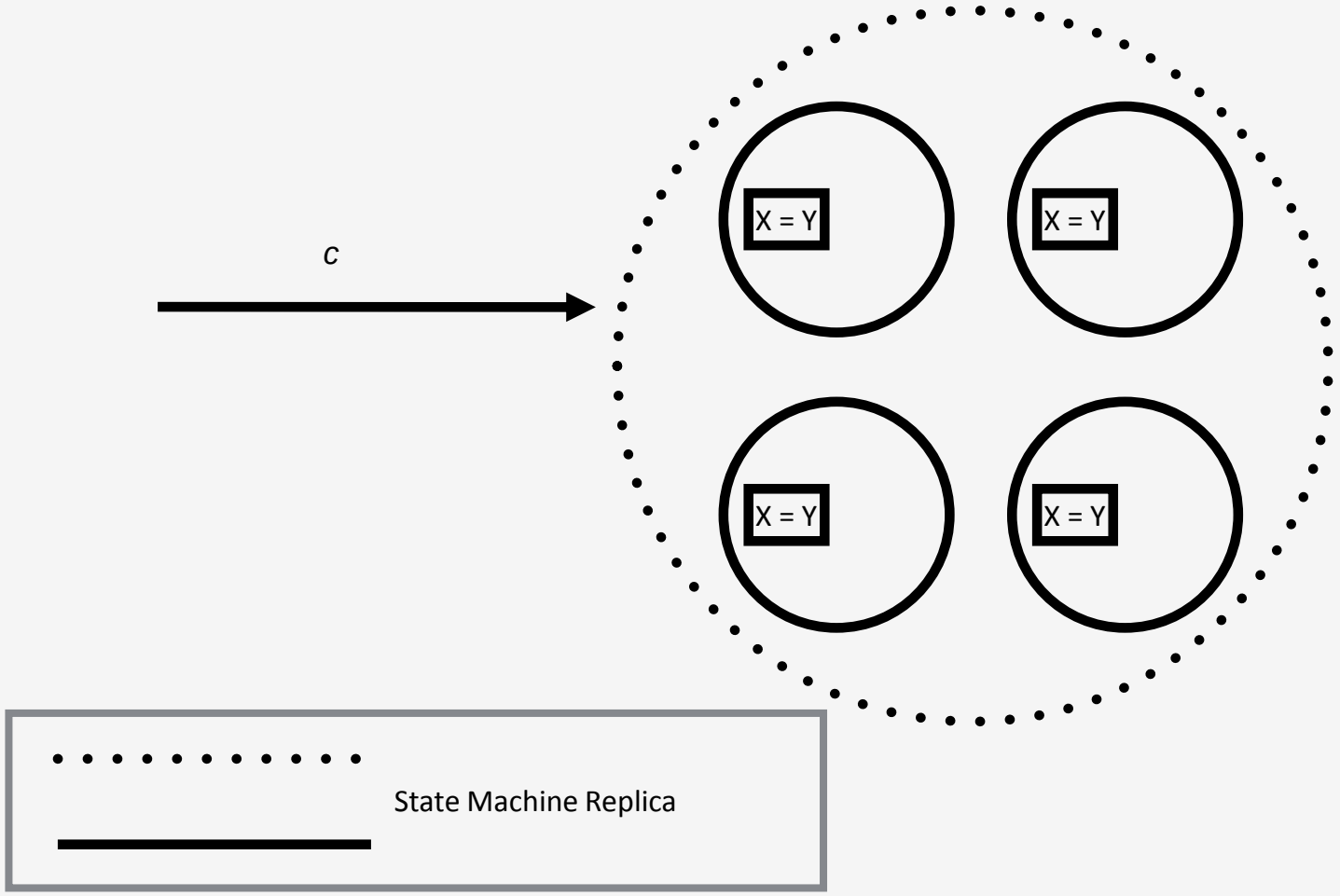
- Need replication for fault tolerance
- What happens in these scenarios without replication?
  - Storage - Disk Failure
  - Webservice - Network failure
- Be able to reason about failure tolerance
  - How badly can things go wrong and have our system continue to function?

State Machines



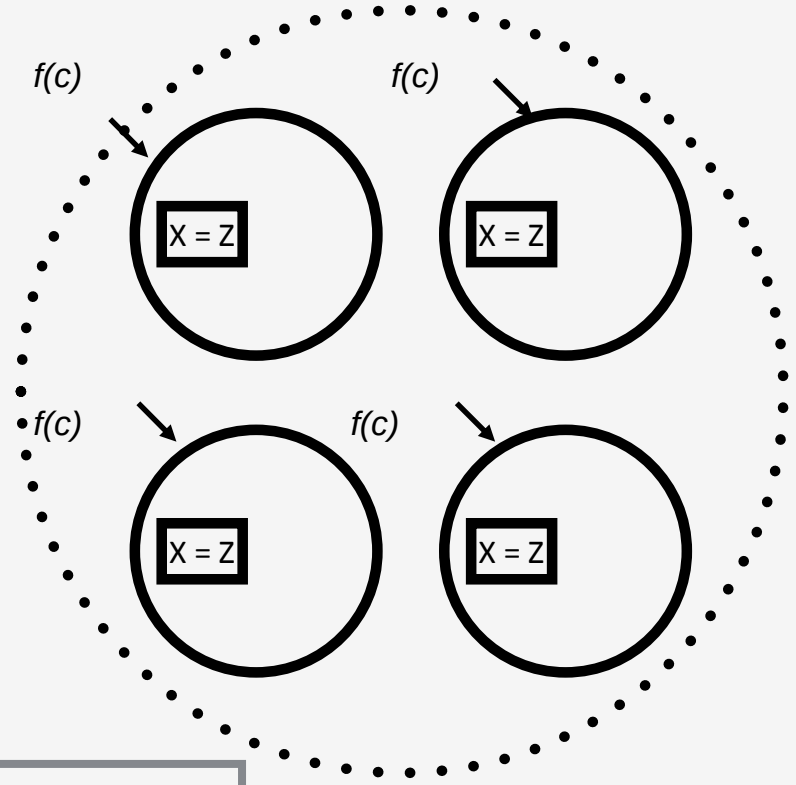
- *c* is a Command
- *f* is a Transition Function

State Machine Replication (SMR)



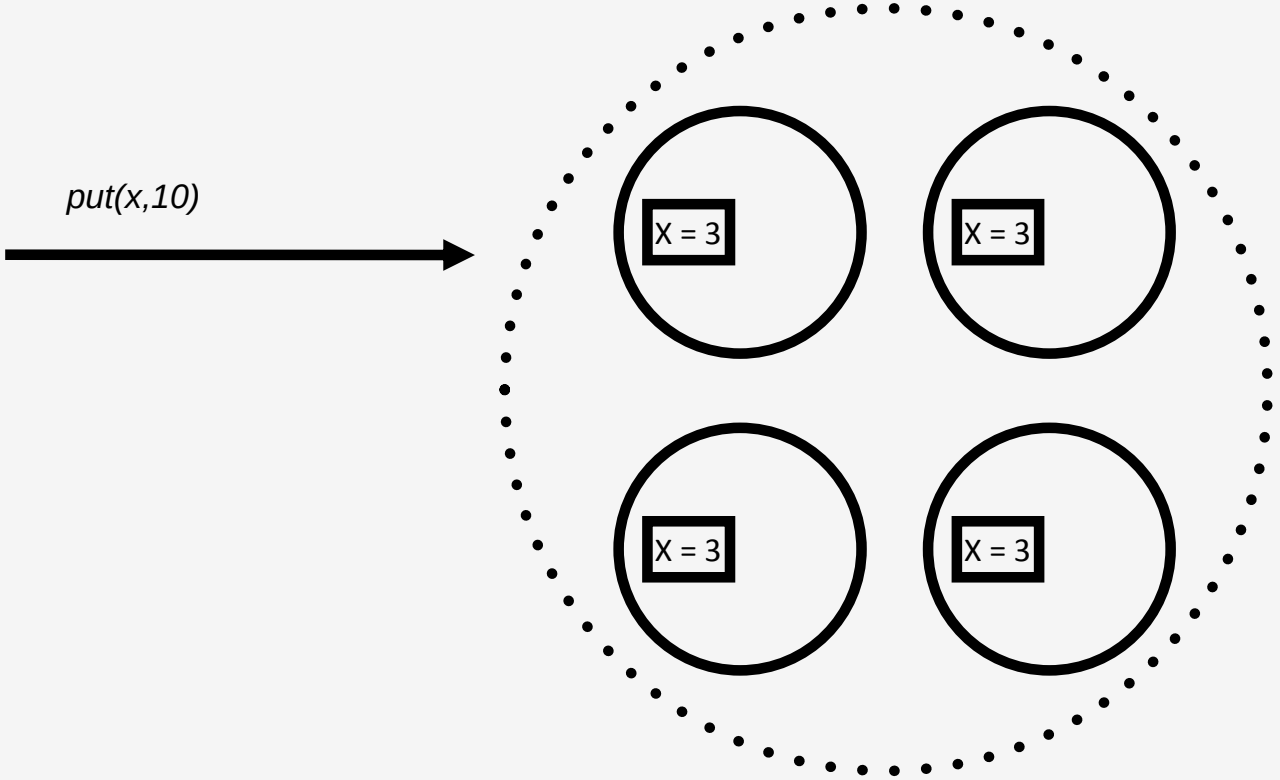
- The *State Machine Approach* to a fault tolerant distributed system
- Keep around *N* copies of the state machine

State Machine Replication (SMR)

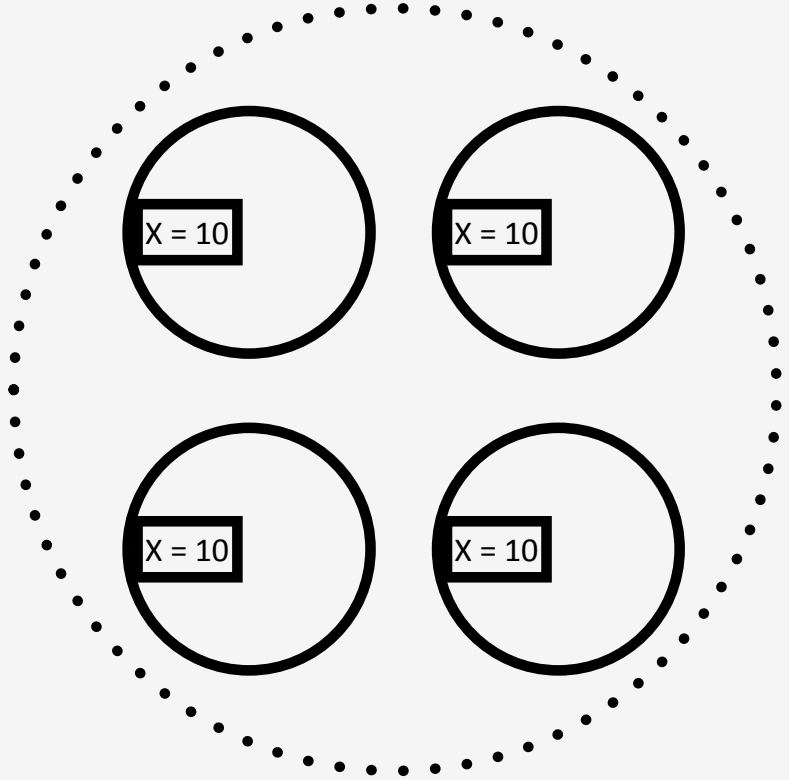


- The *State Machine Approach* to a fault tolerant distributed system
- Keep around *N* copies of the state machine

SMR Requirements

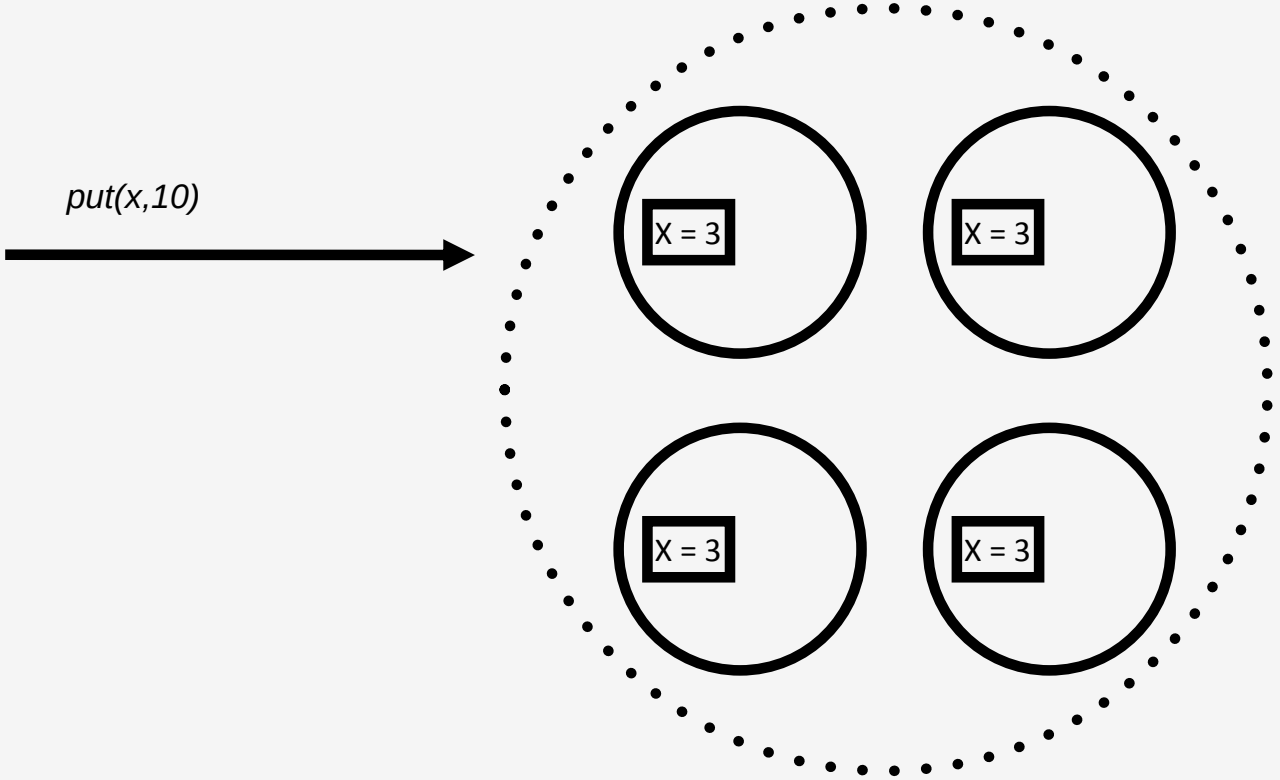


SMR Requirements

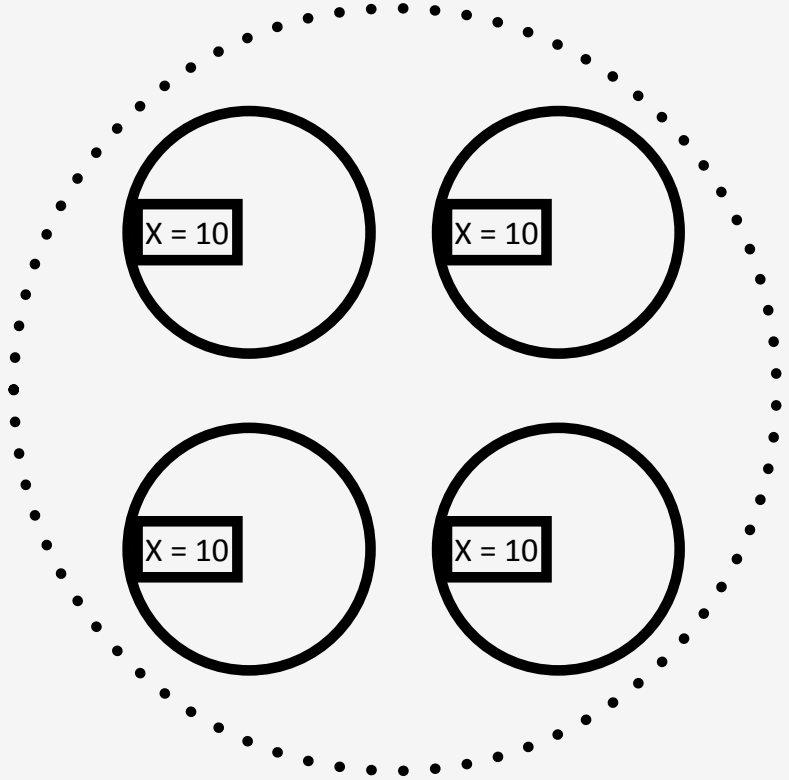


Great!

SMR Requirements

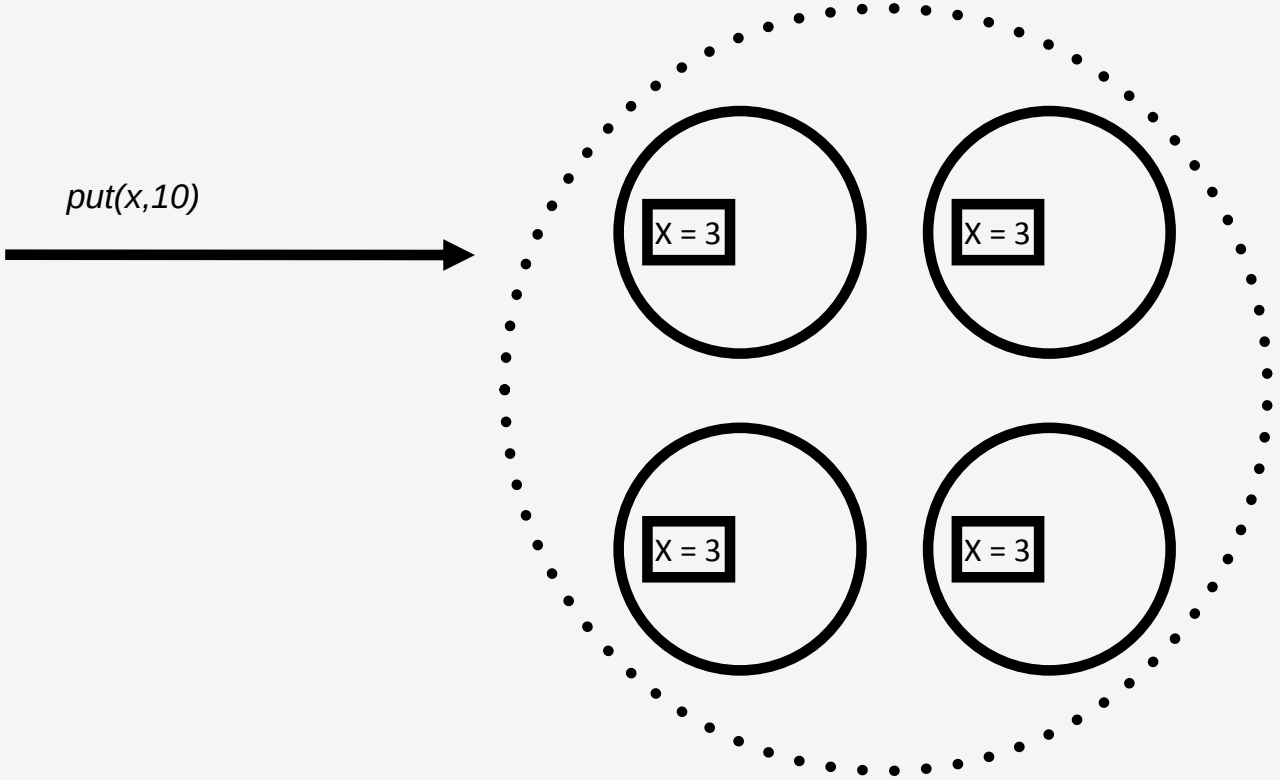


SMR Requirements

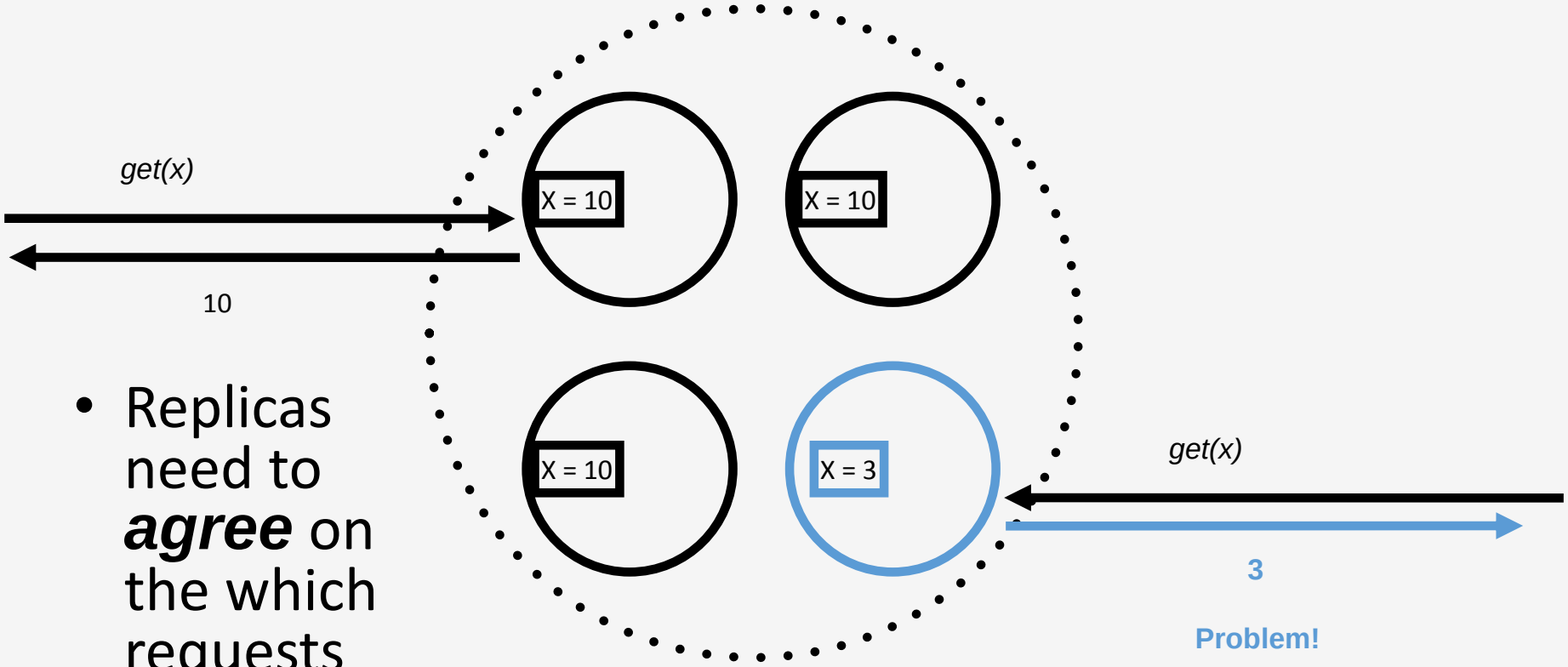


Great!

SMR Requirements

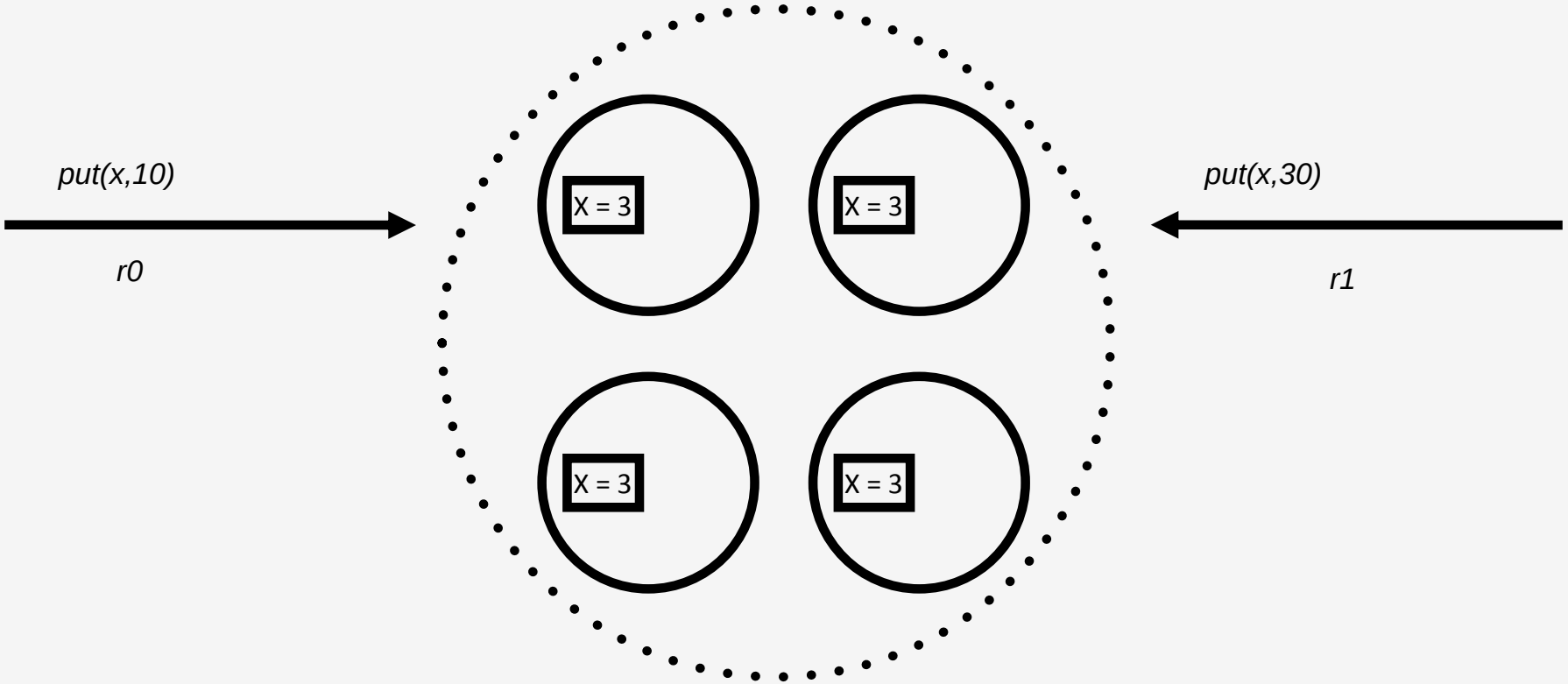


SMR Requirements

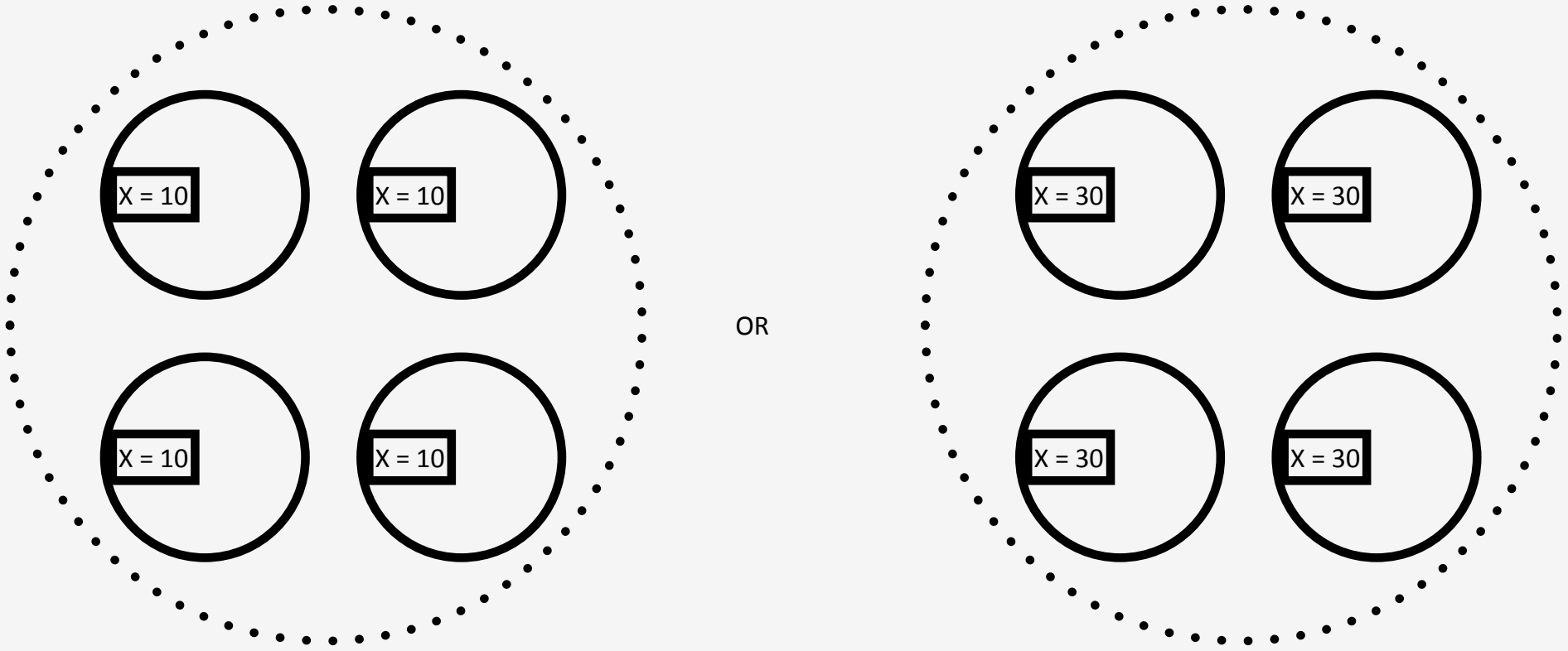


- Replicas need to ***agree*** on the which requests have been handled

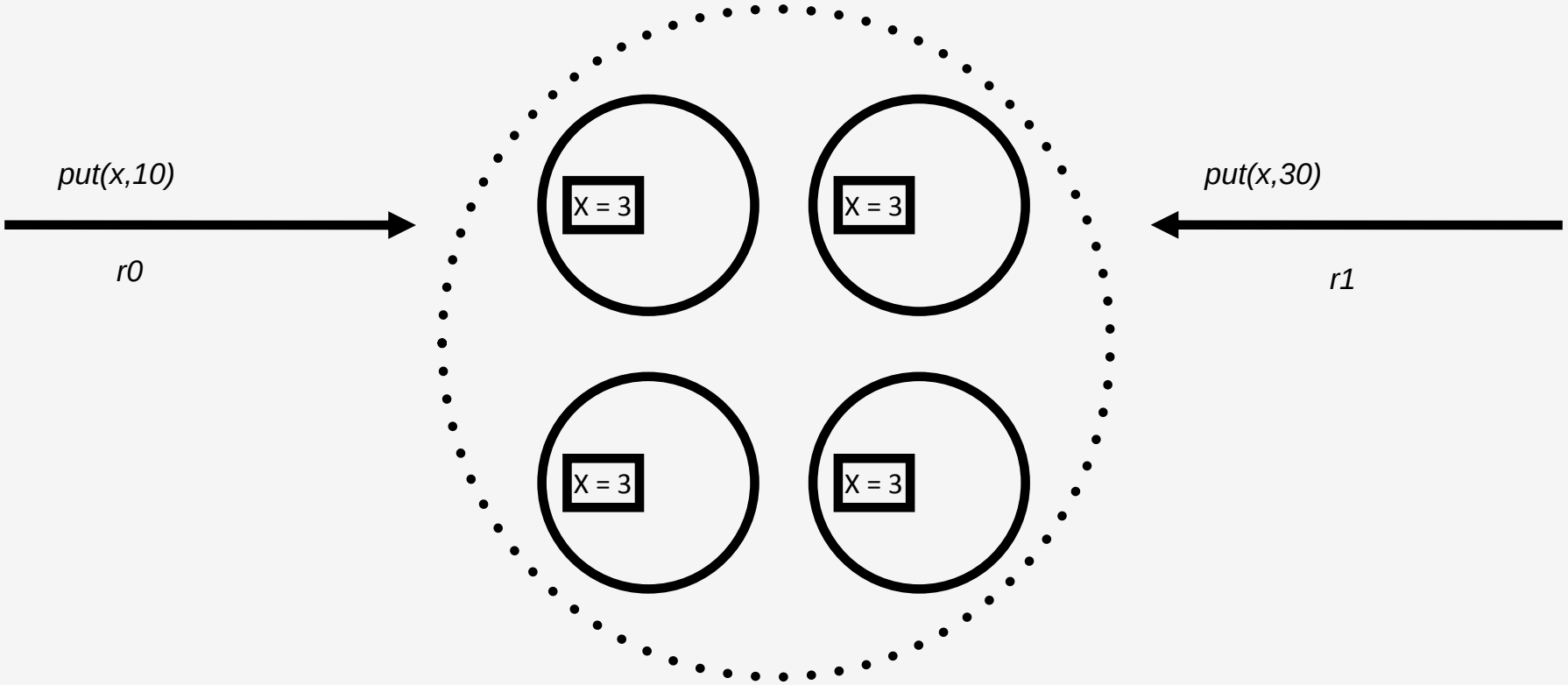
SMR Requirements



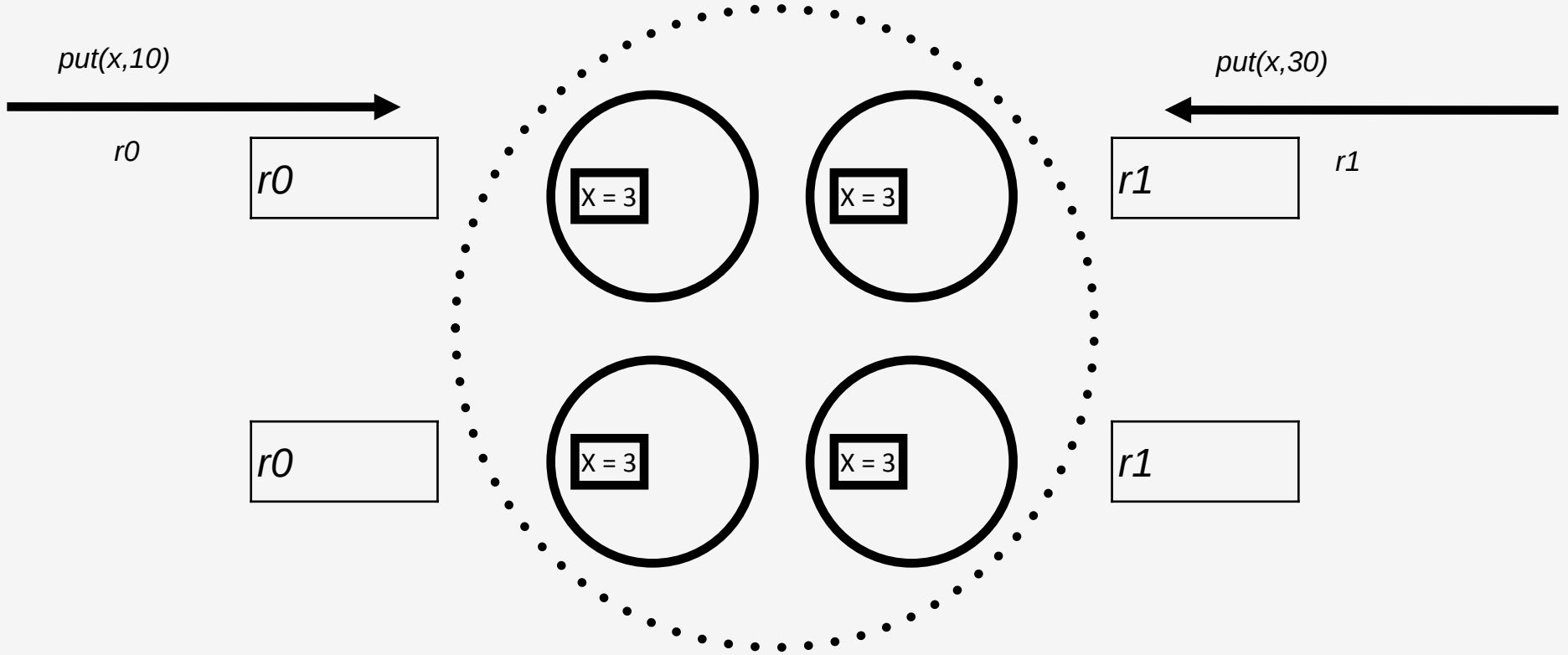
SMR Requirements



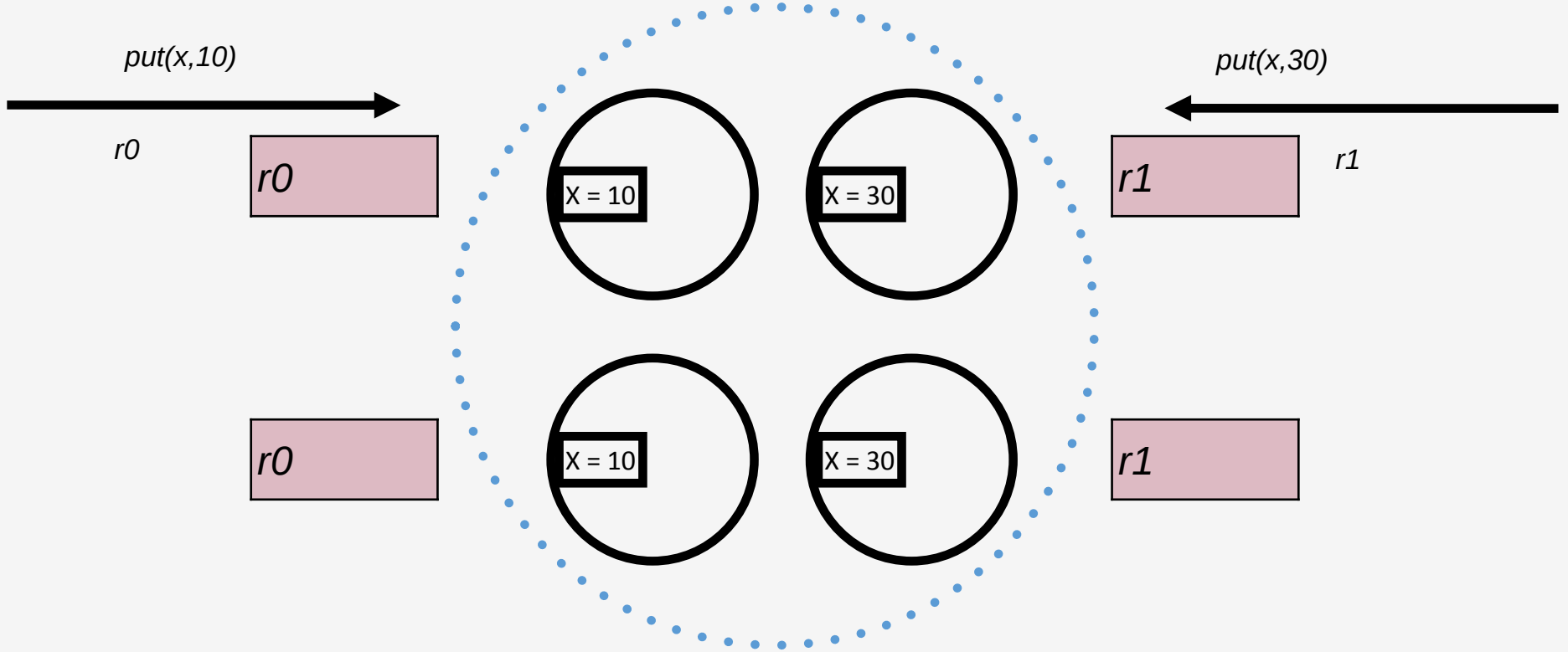
SMR Requirements



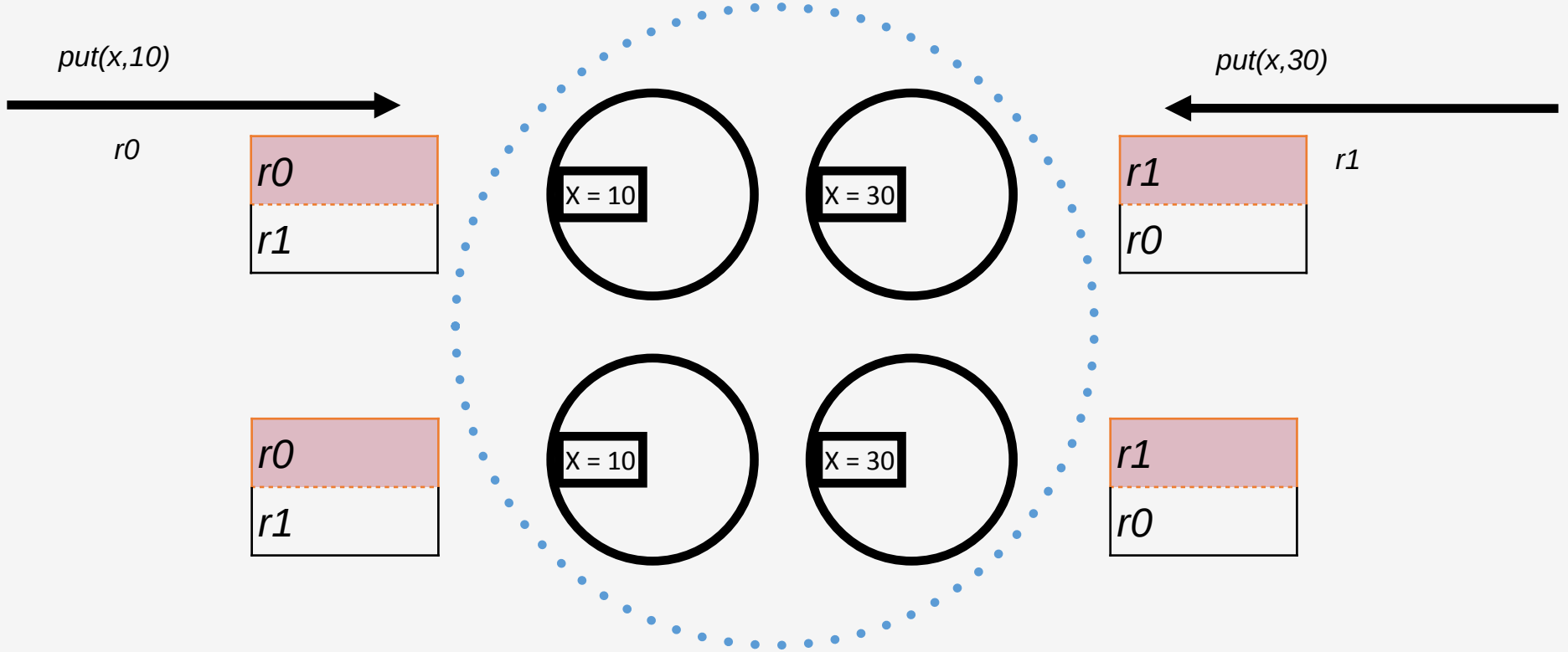
SMR Requirements



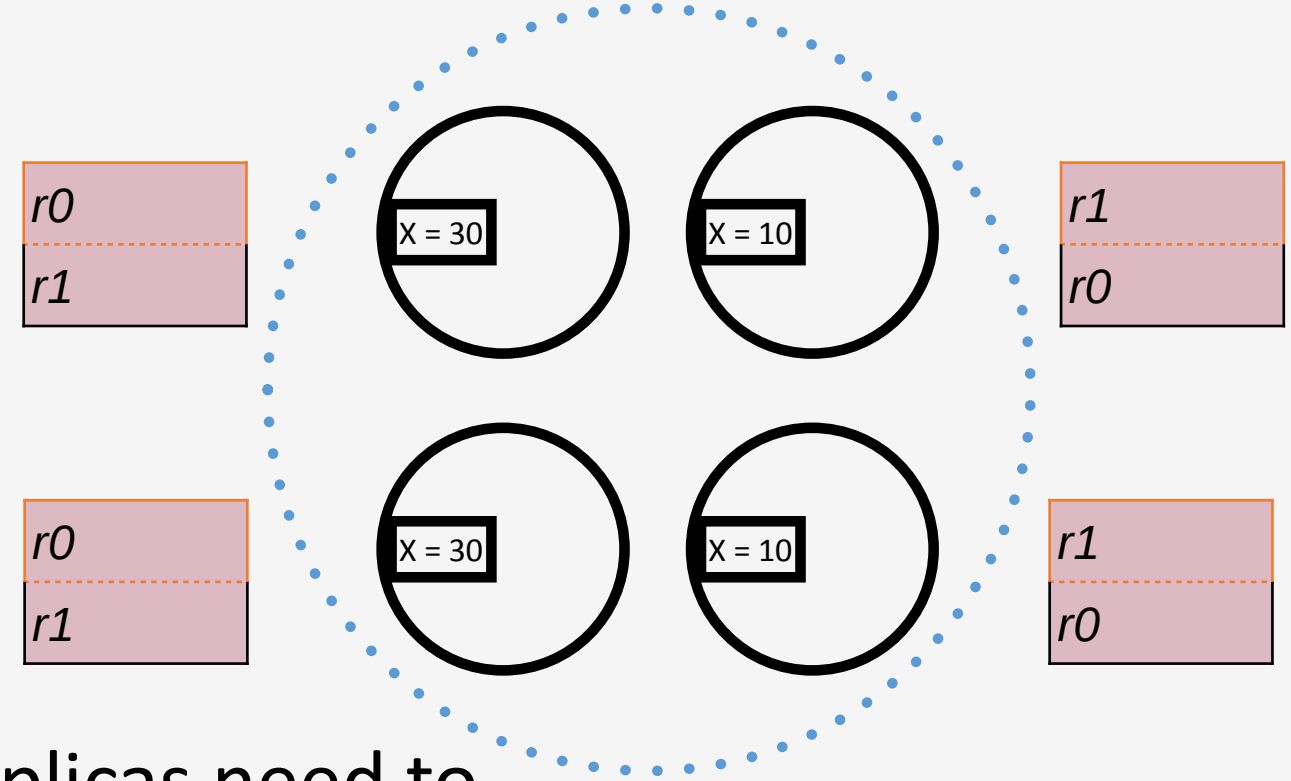
SMR Requirements



SMR Requirements



SMR Requirements



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

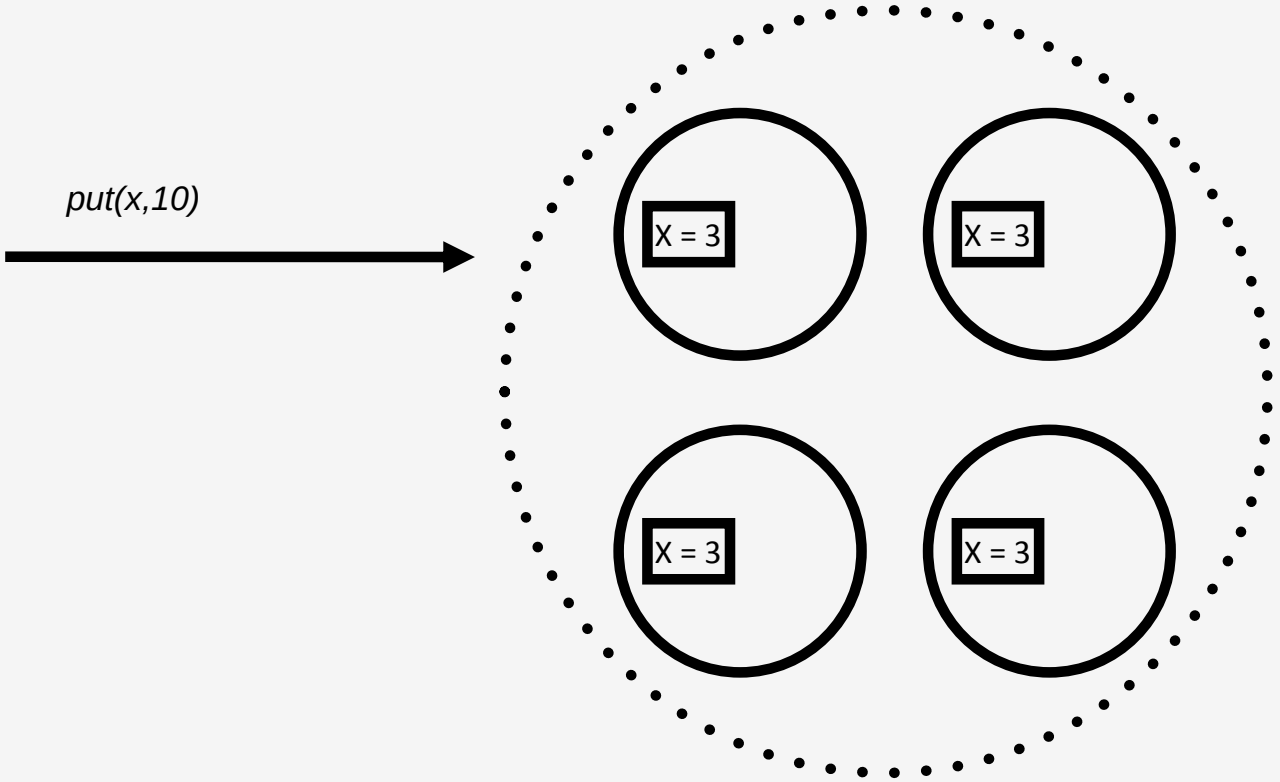
SMR

- 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

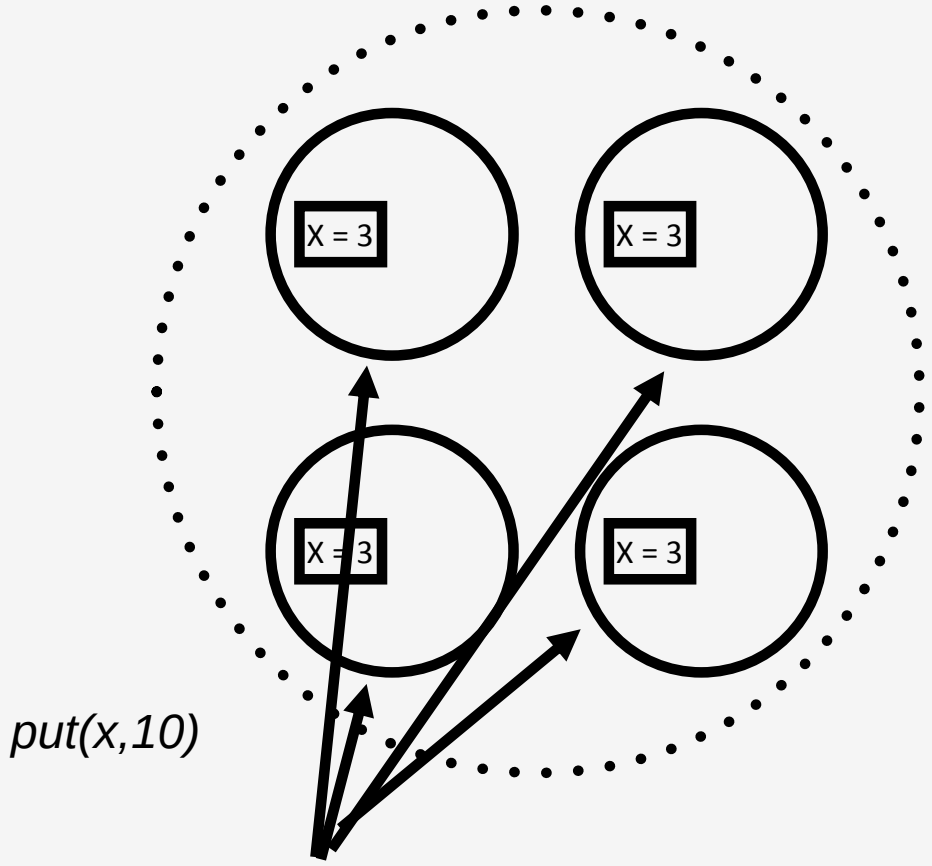
Implementation

- Agreement
  - Someone proposes a request; if that person is nonfaulty all servers will accept that request
  - Strong and Dolev [1983] and Schneider [1984] for implementations
  - Client or Server can propose the request

SMR Implementation



SMR Implementation

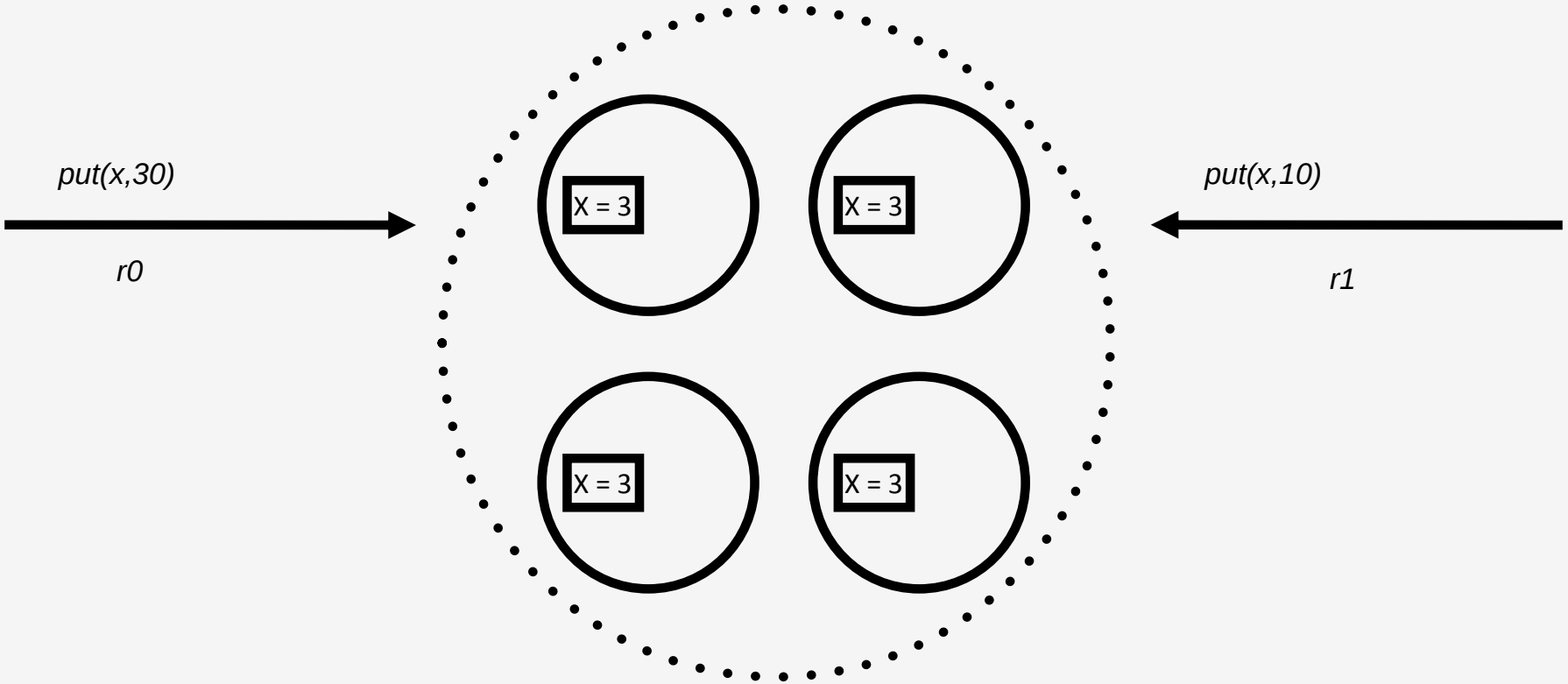


Non-faulty Transmitter

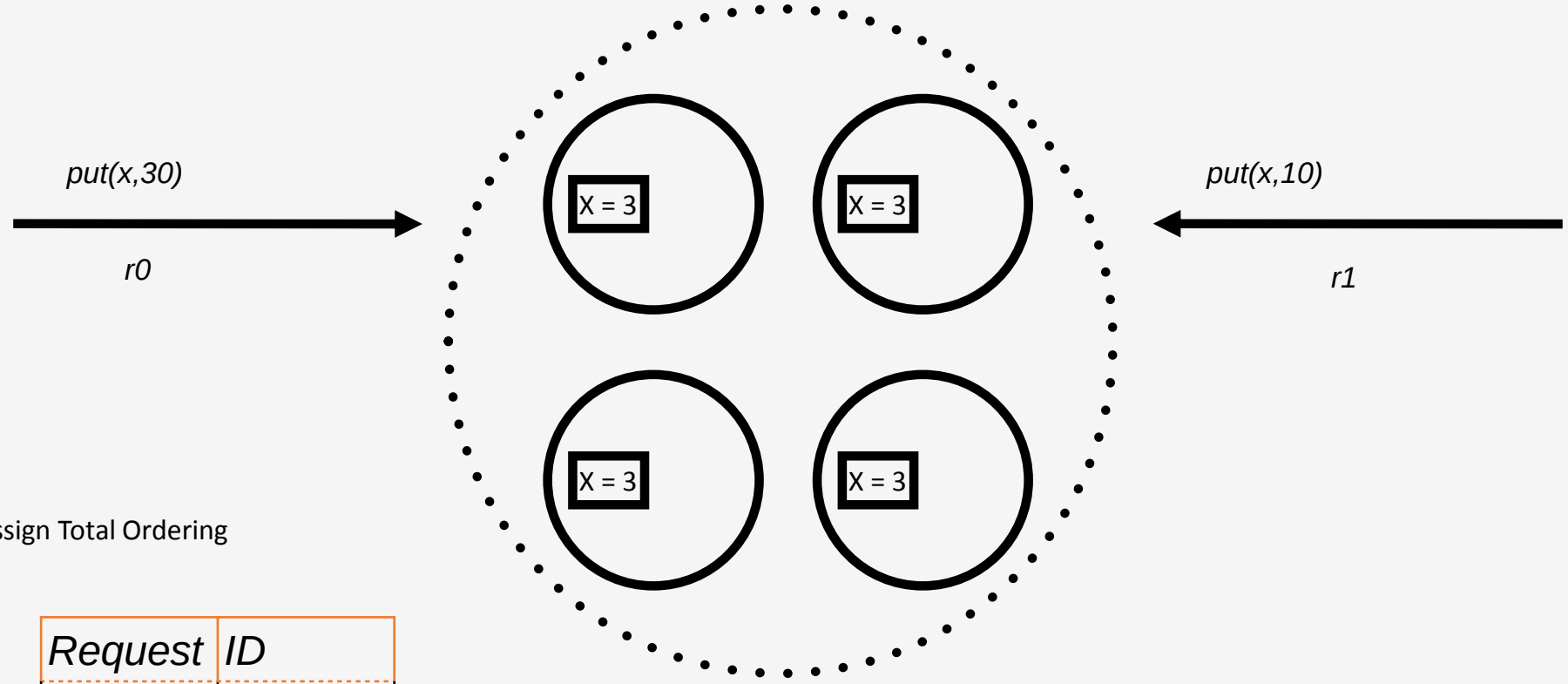
Implementation

- 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?

SMR Requirements



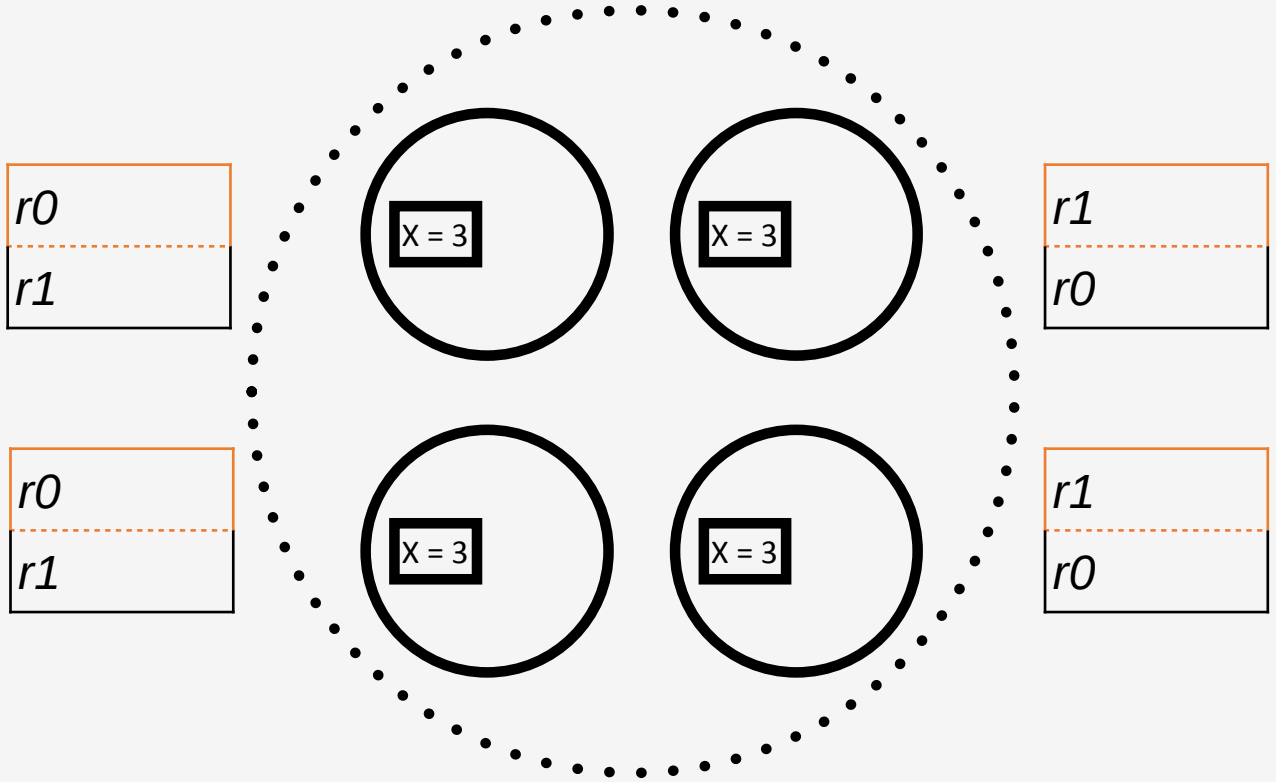
SMR Requirements



Assign Total Ordering

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

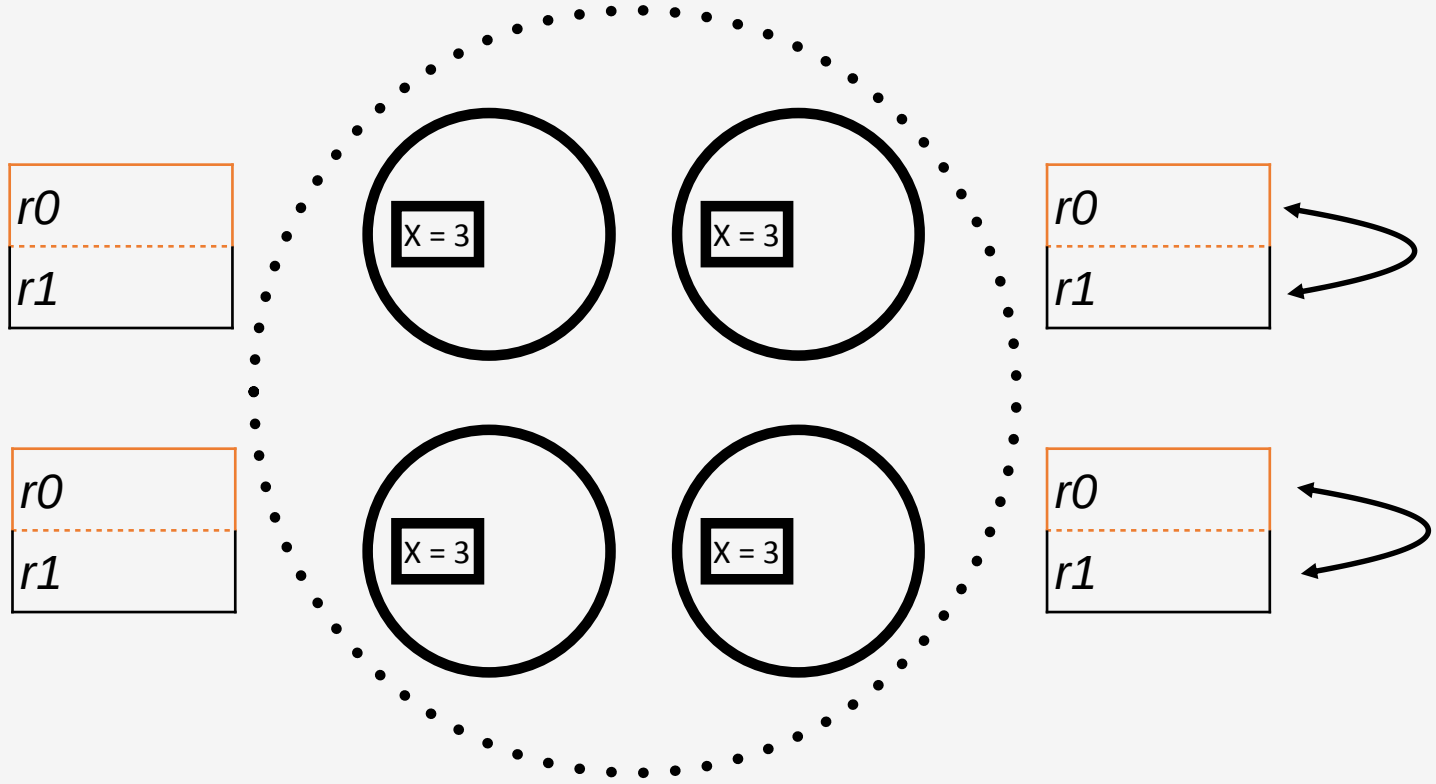
SMR Requirements



Assign Total Ordering

Request	ID
r0	1
r1	2

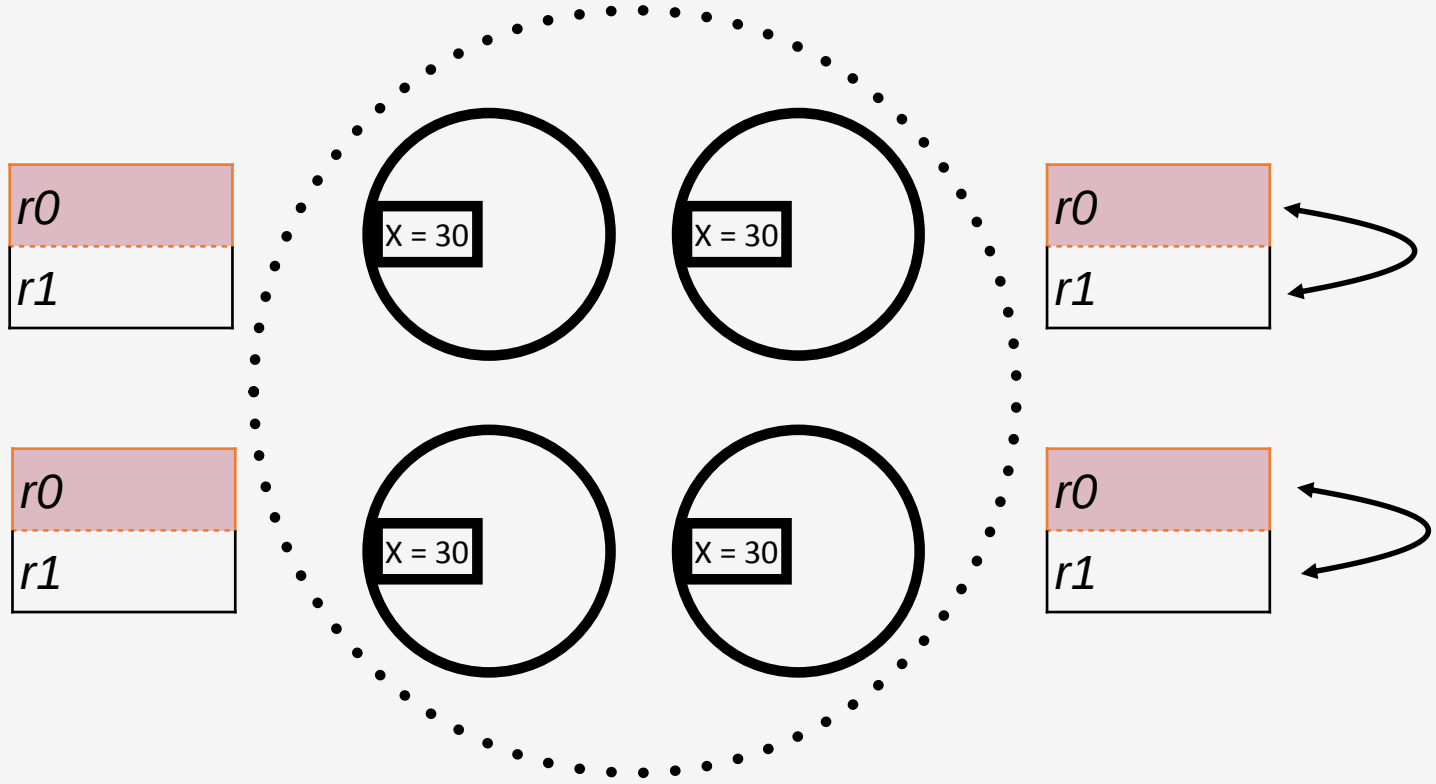
SMR Requirements



Assign Total Ordering

Request	ID
r0	1
r1	2

SMR Requirements

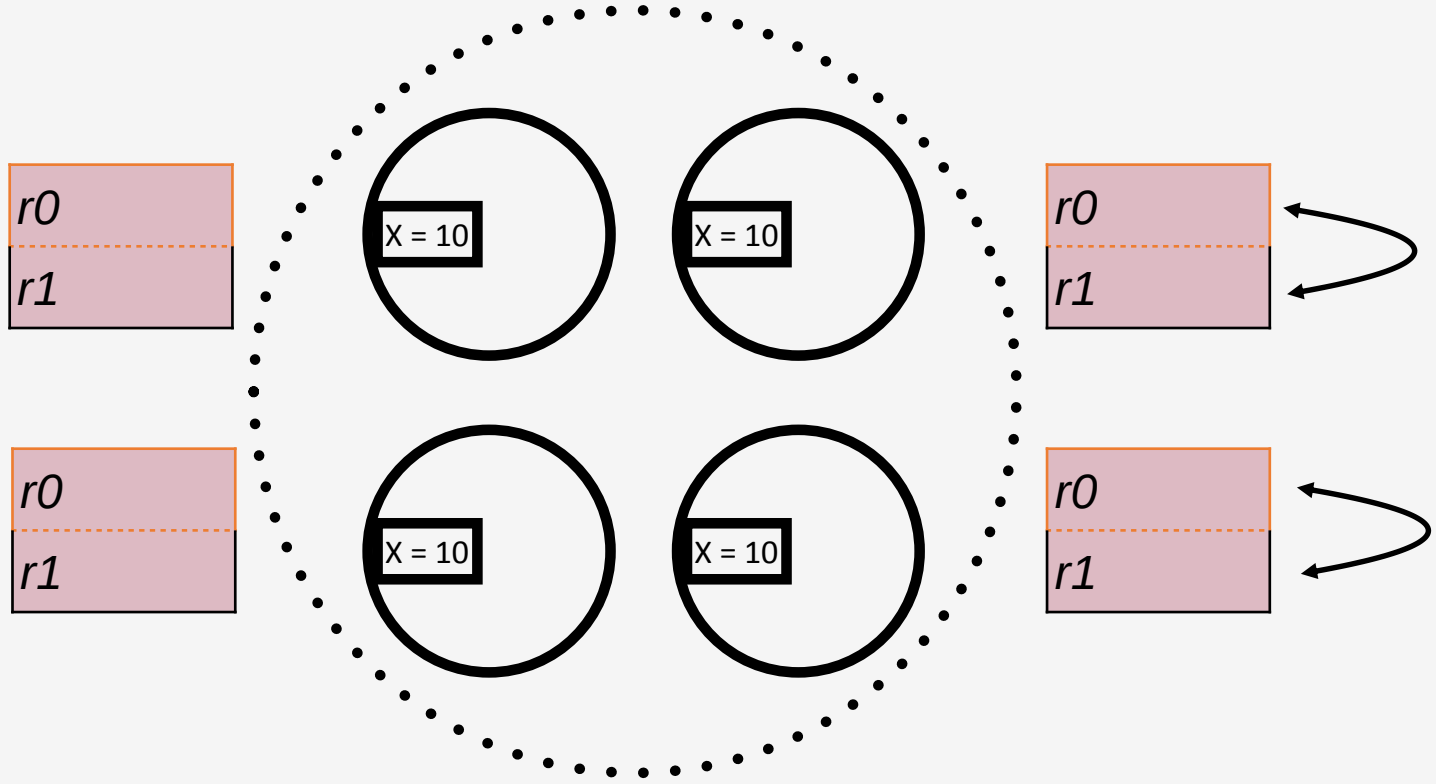


Assign Total Ordering

Request	ID
r0	1
r1	2

r0 is now stable!

SMR Requirements



Assign Total Ordering

Request	ID
r0	1
r1	2

r0 is now stable!

r1 is now stable!

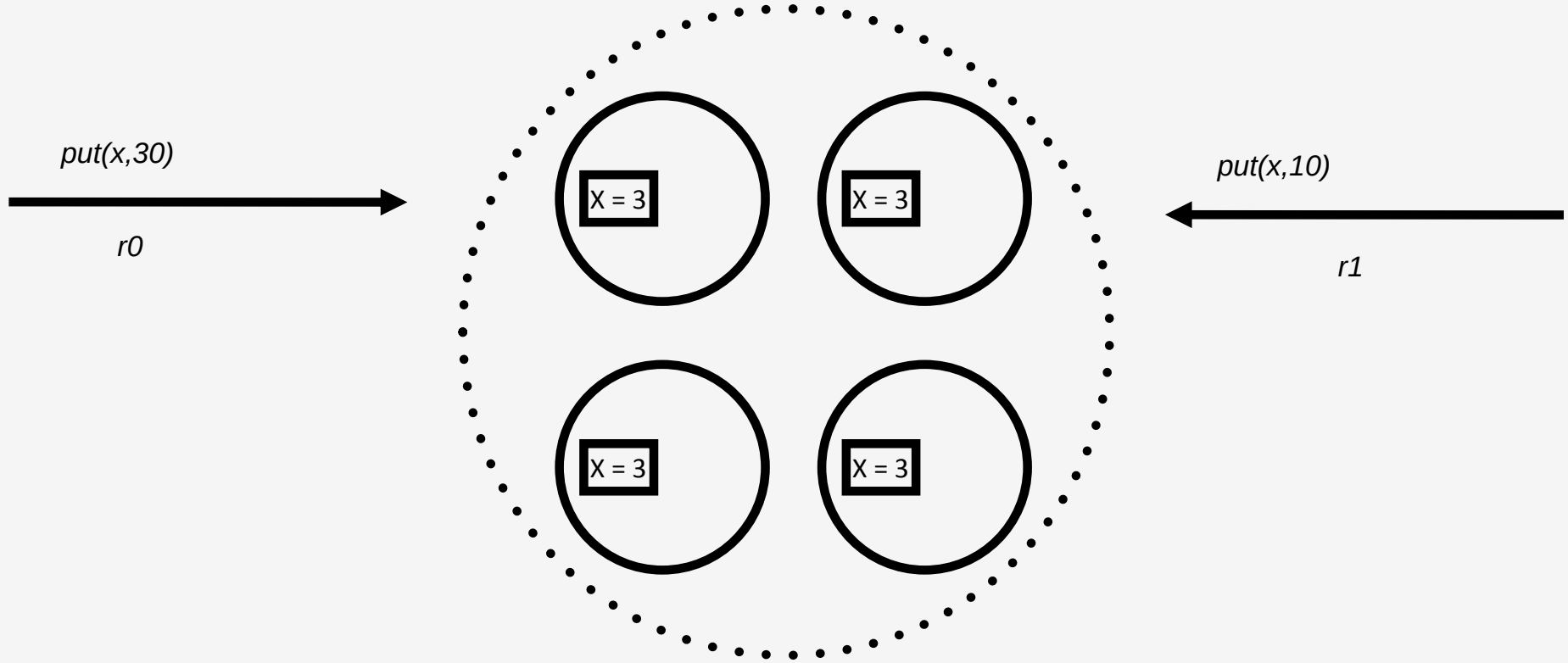
Implementation Client Generated IDs

- Order via Clocks (Client timestamps represent IDs)
  - Logical Clocks
  - Synchronized Clocks
- Ideas from [Lamport 1978]

Implementation Replica Generated IDs

- 2 Phase ID generation
  - Every Replica proposes a *candidate*
  - One candidate is chosen and agreed upon by all replicas

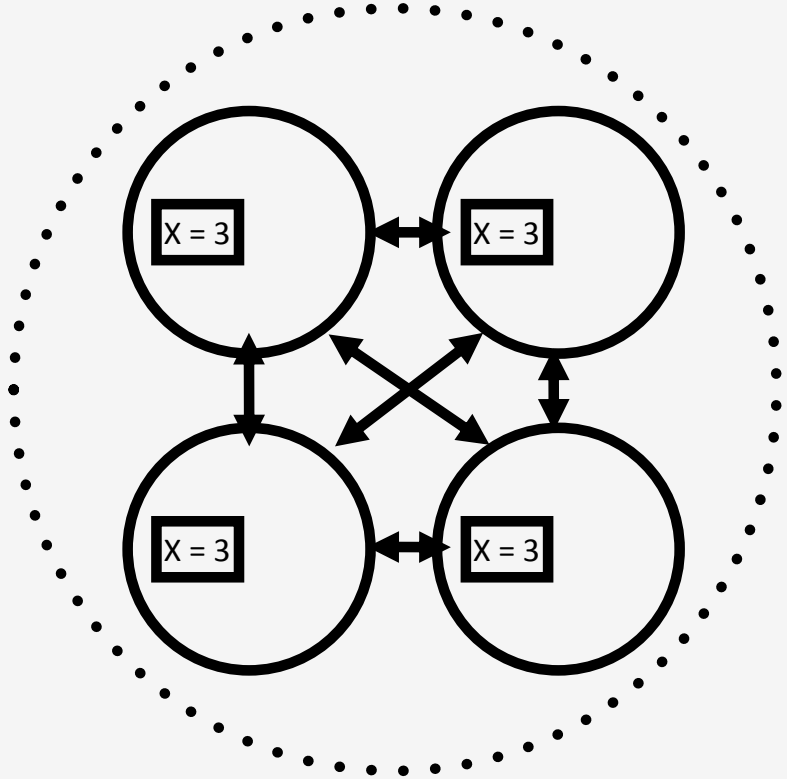
Replica ID Generation



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	

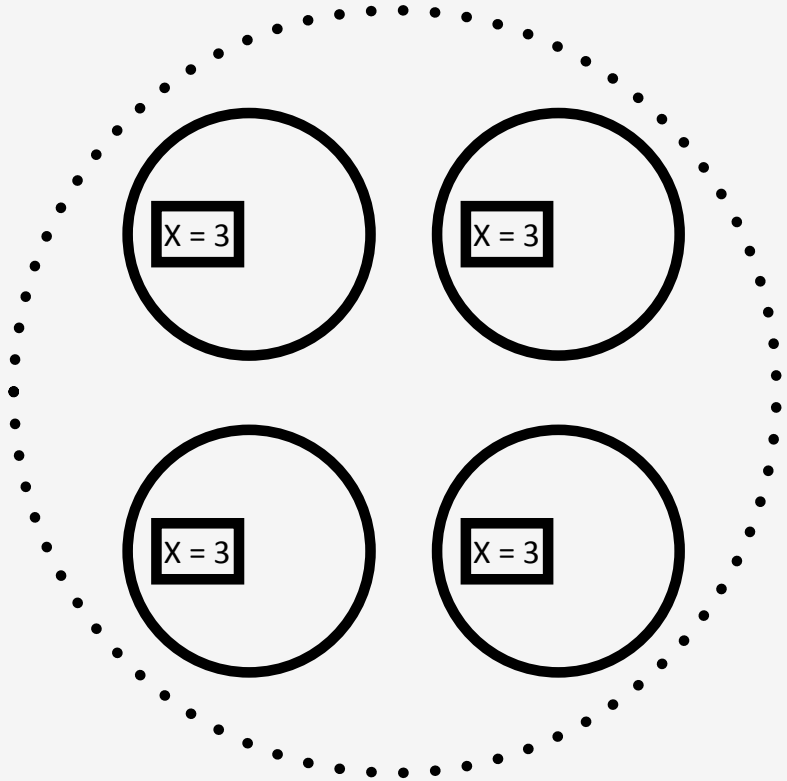
Req.	CUID	UID
r1	1.4	
r0	2.4	

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*

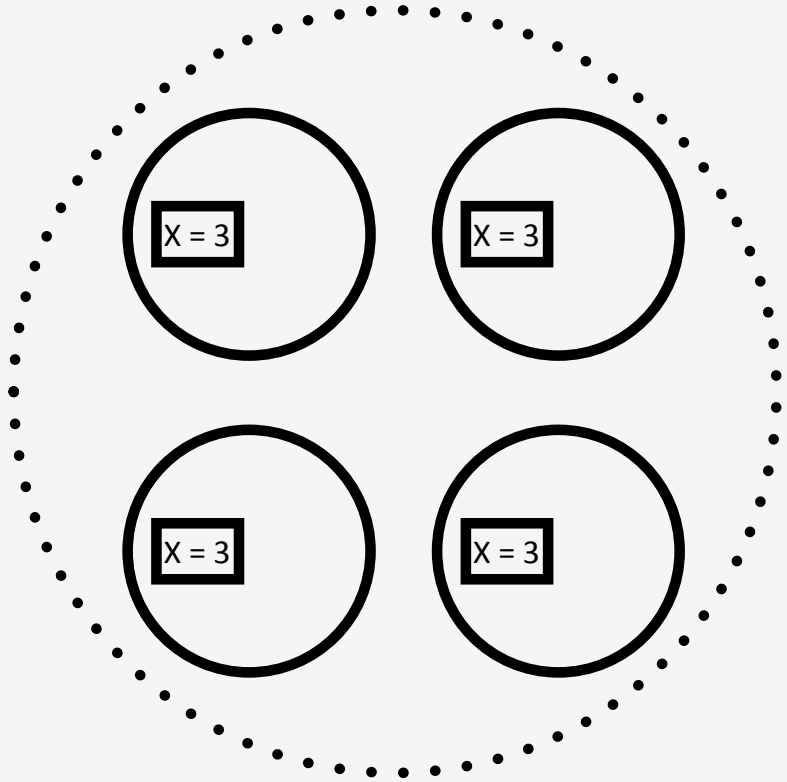
Replica ID Generation

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

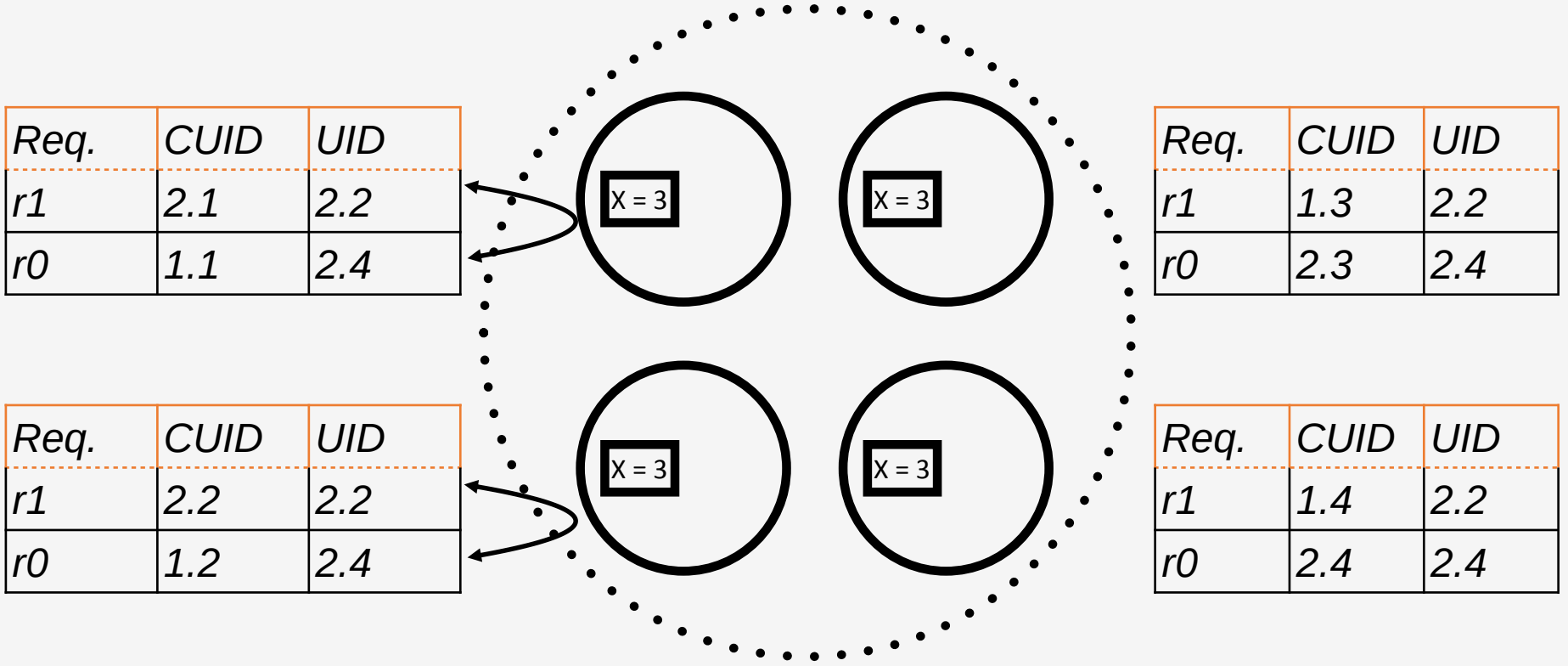
Req.	CUID	UID
r0	1.2	2.4
r1	2.2	2.2

Req.	CUID	UID
r1	1.4	2.2
r0	2.4	2.4



3) Accept *r1*

Replica ID Generation



r1 is now stable

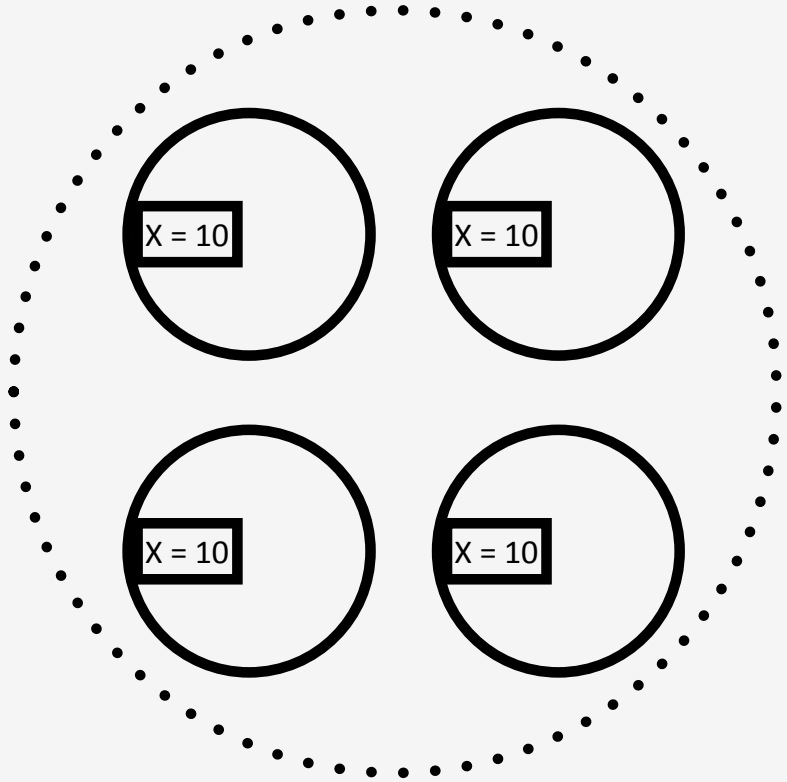
Replica ID Generation

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*

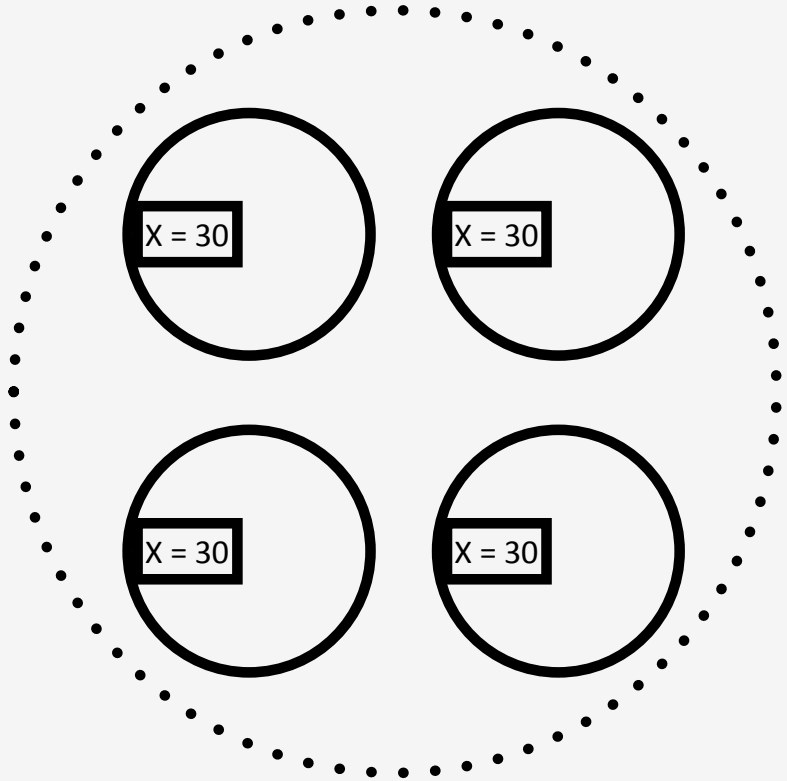


4) Apply *r1*

Replica ID Generation

Req.	CUID	UID
r1	2.1	2.2
r0	1.1	2.4

Req.	CUID	UID
r1	2.2	2.2
r0	1.2	2.4



Req.	CUID	UID
r1	1.3	2.2
r0	2.3	2.4

Req.	CUID	UID
r1	1.4	2.2
r0	2.4	2.4

5) Apply *r0*

Implementation Replica Generated IDs

- 2 Rules for Candidate Generation/Selection
  - Any new candidate ID must be > the id of any *accepted* request.
  - The ID selected from the candidate list must be >= each candidate
- In the paper these are written as:
  - If a request *r'* is seen by a replica sm_i after *r* has been accepted by sm_i then $uid(r) < cuid(sm_i, r')$
  - $cuid(sm_i, r) <= uid(r)$

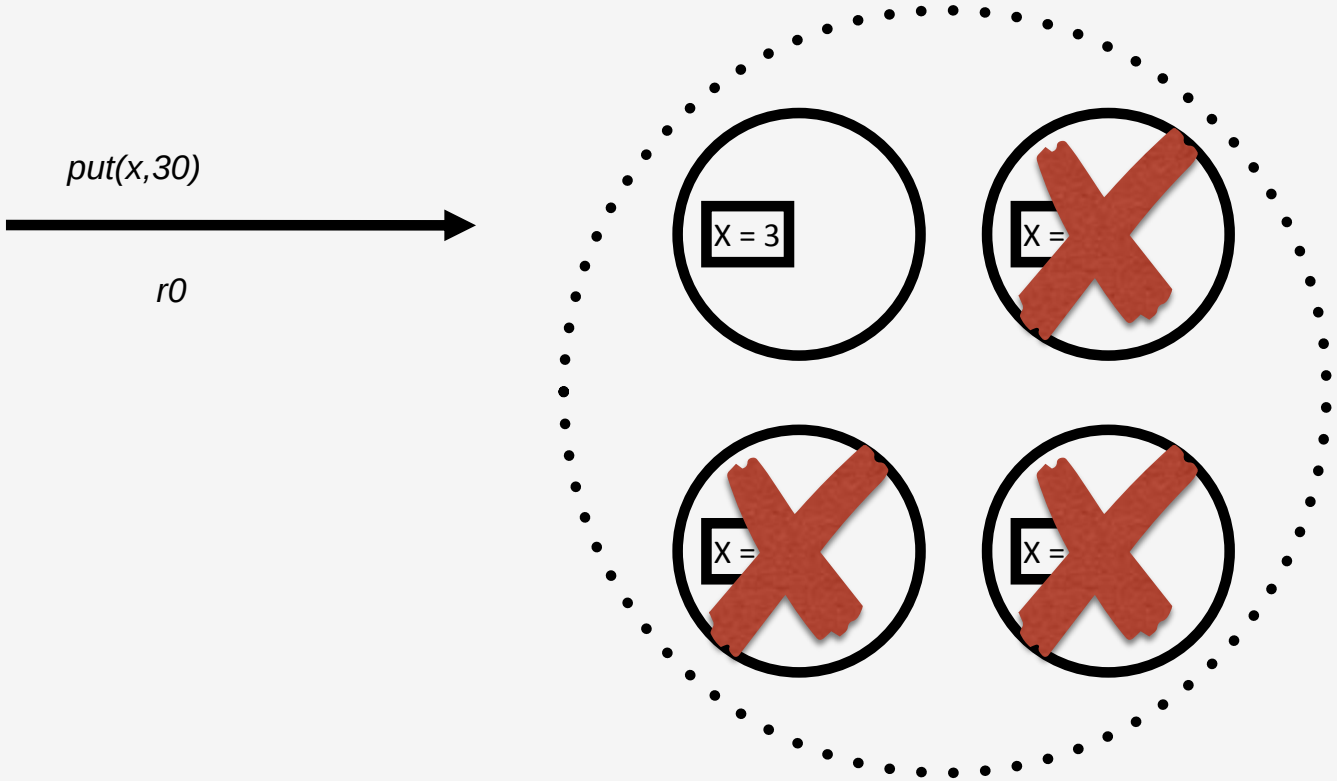
Fault Tolerance

- Fail-Stop
  - A faulty server can be detected as faulty
- Byzantine
  - Faulty servers can do arbitrary, perhaps malicious things
- Crash Failures
  - Server can stop responding without notification (subset of Byzantine)

Fault Tolerance

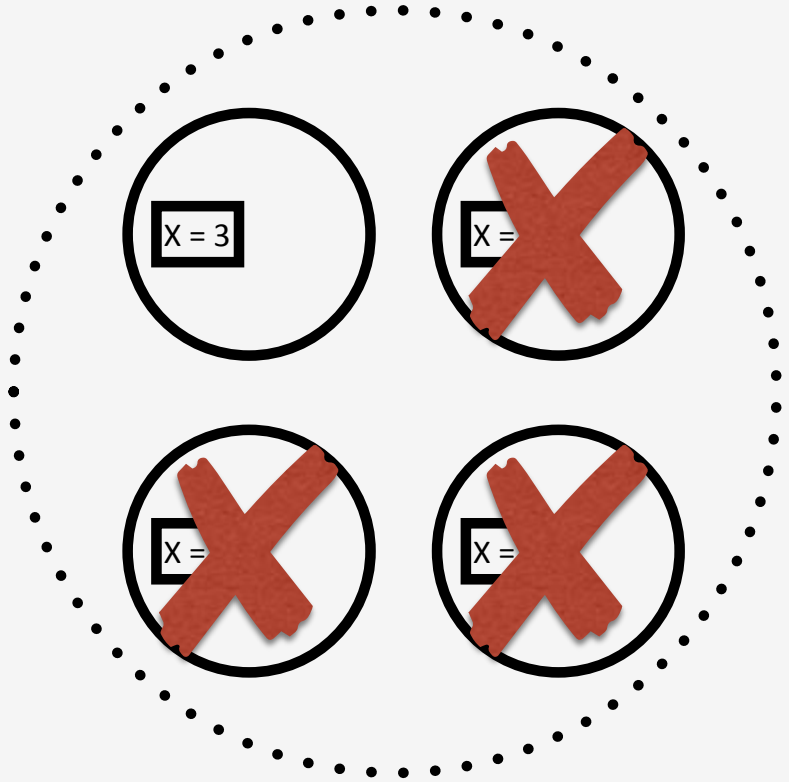
- Fail-Stop
  - A faulty server can be detected as faulty
- Byzantine
  - Faulty servers can do arbitrary, perhaps malicious things
- Crash Failures
  - Server can stop responding without notification (subset of Byzantine)

Fail-Stop Tolerance



Fail-Stop Tolerance

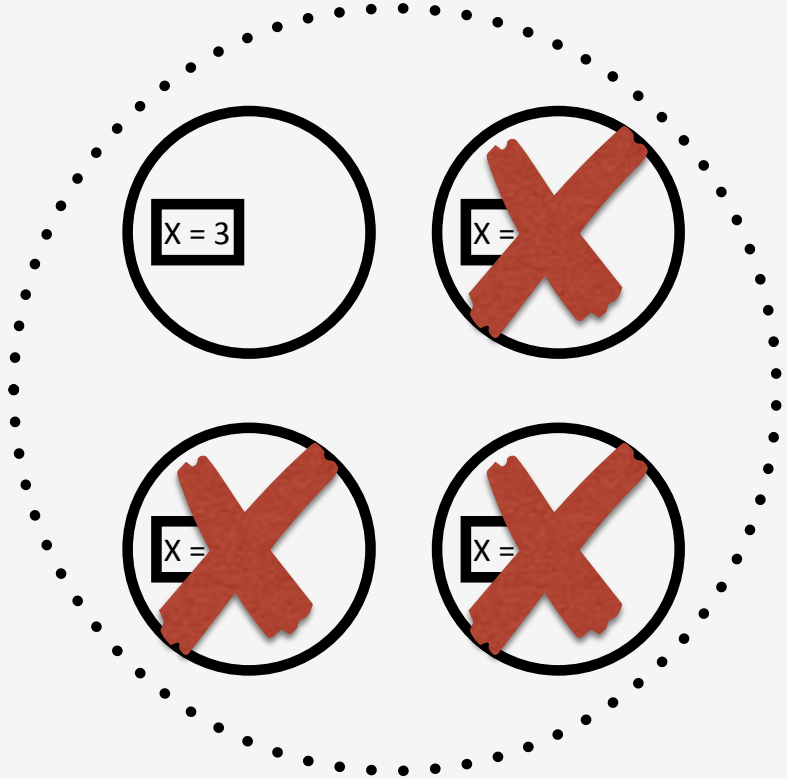
Req.	*CUID*	*UID*
r0	*1.1*	



1) Propose Candidates….

Fail-Stop Tolerance

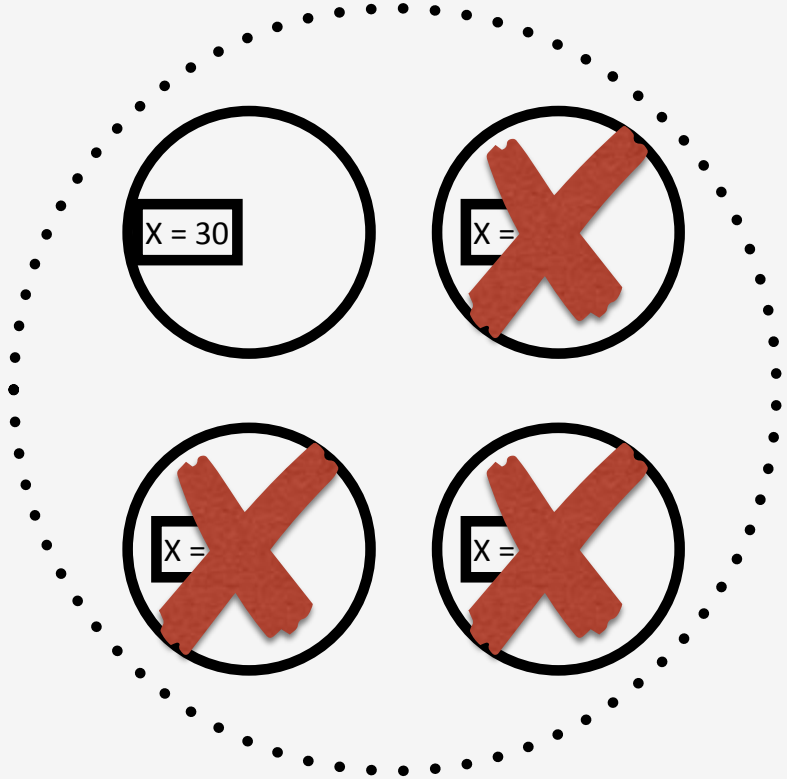
Req.	*CUID*	*UID*
r0	*1.1*	*1.1*



2) Accept *r0*

Fail-Stop Tolerance

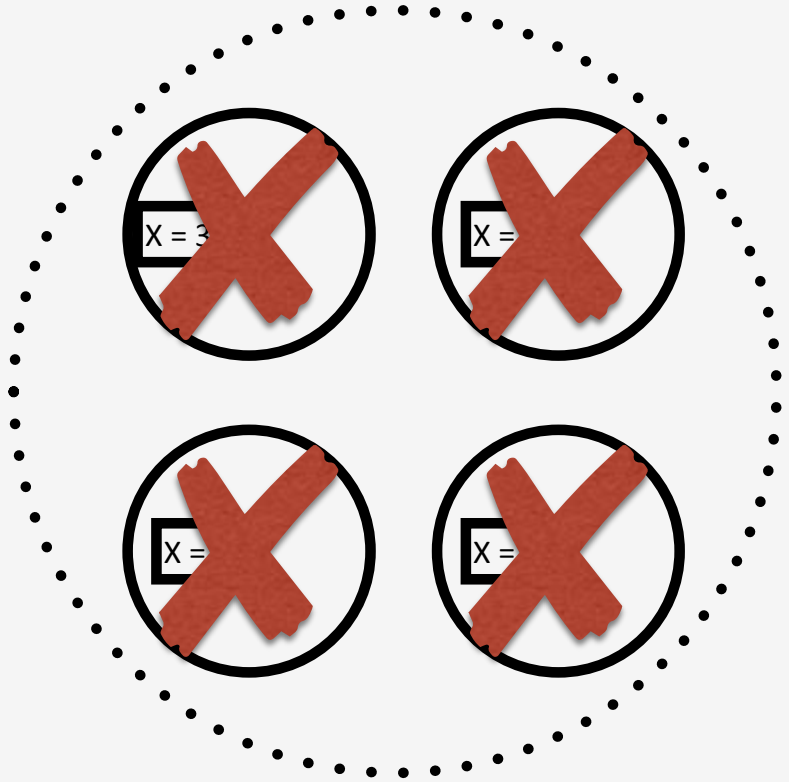
Req.	*CUID*	*UID*
r0	*1.1*	*1.1*



2) Apply *r0*

Fail-Stop Tolerance

GAME OVER!!!

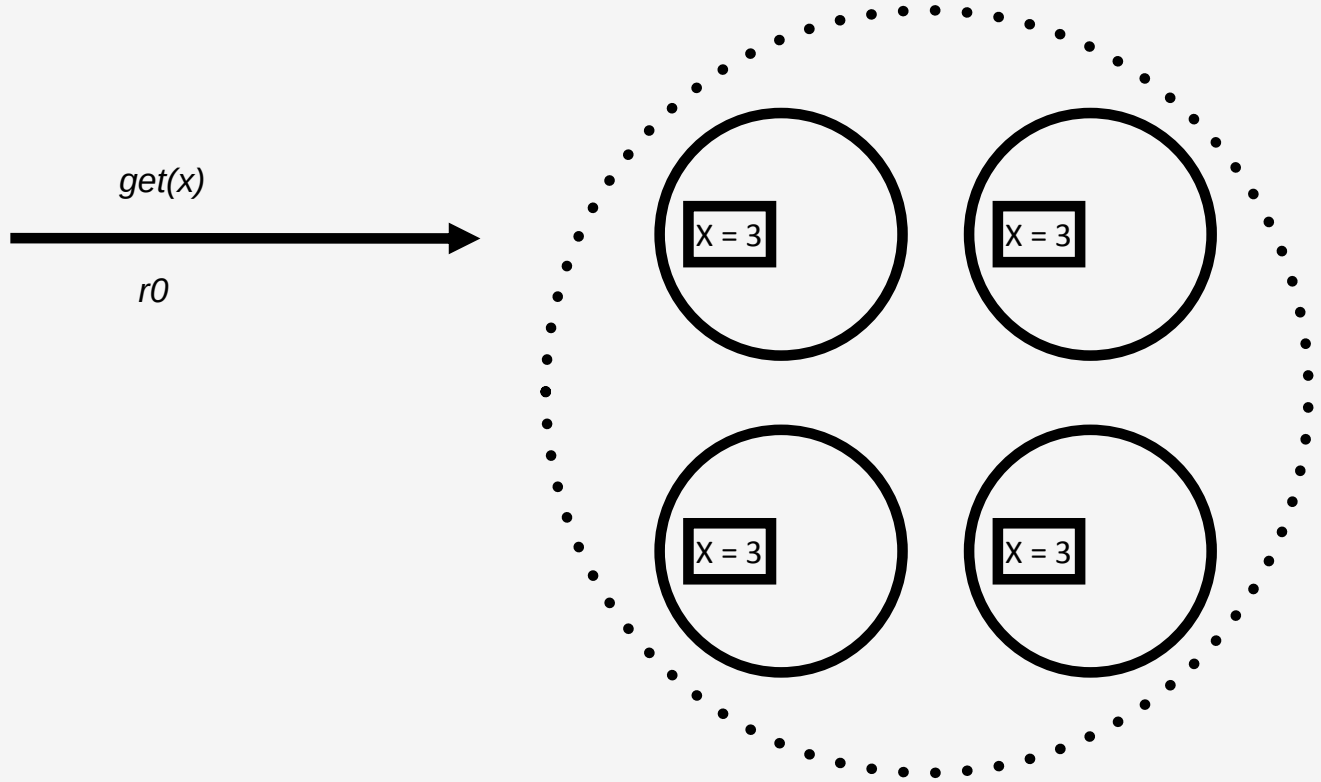


2) Apply *r0*

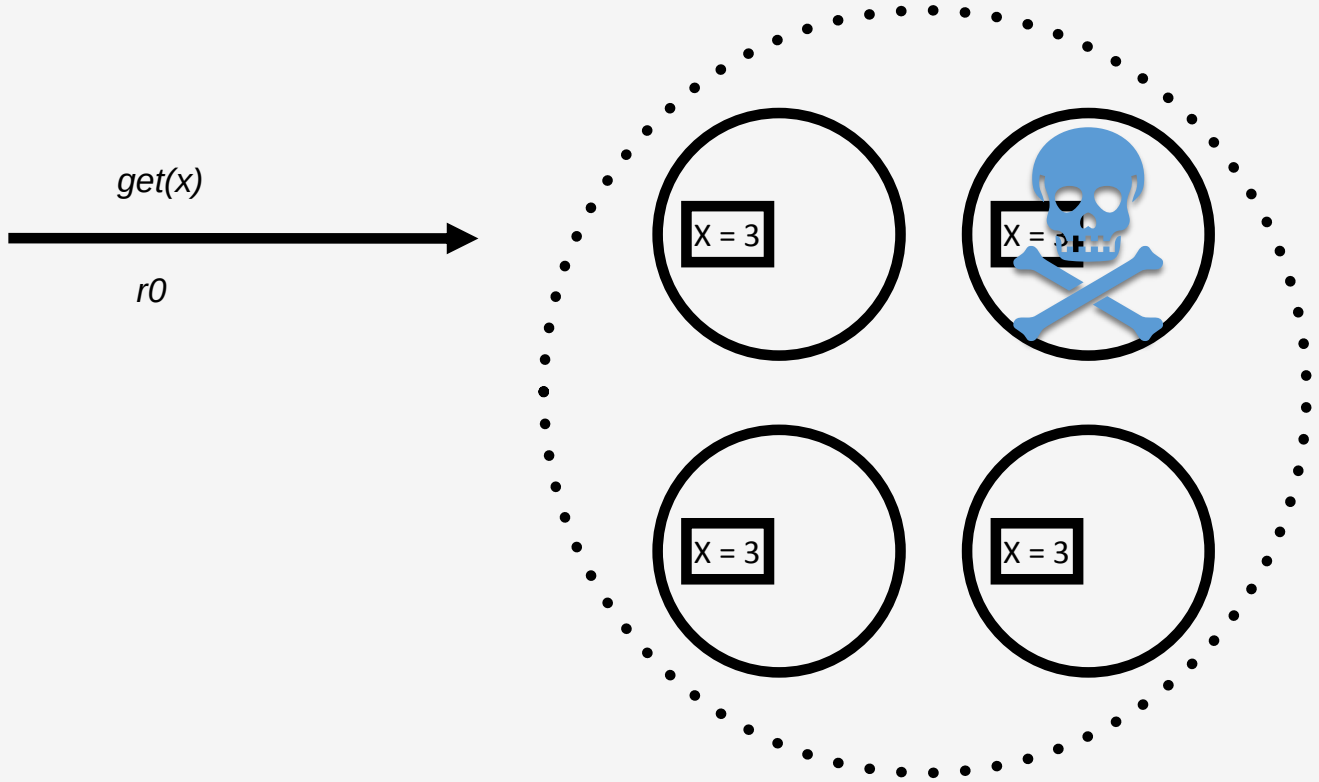
Fail-Stop Tolerance

- 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

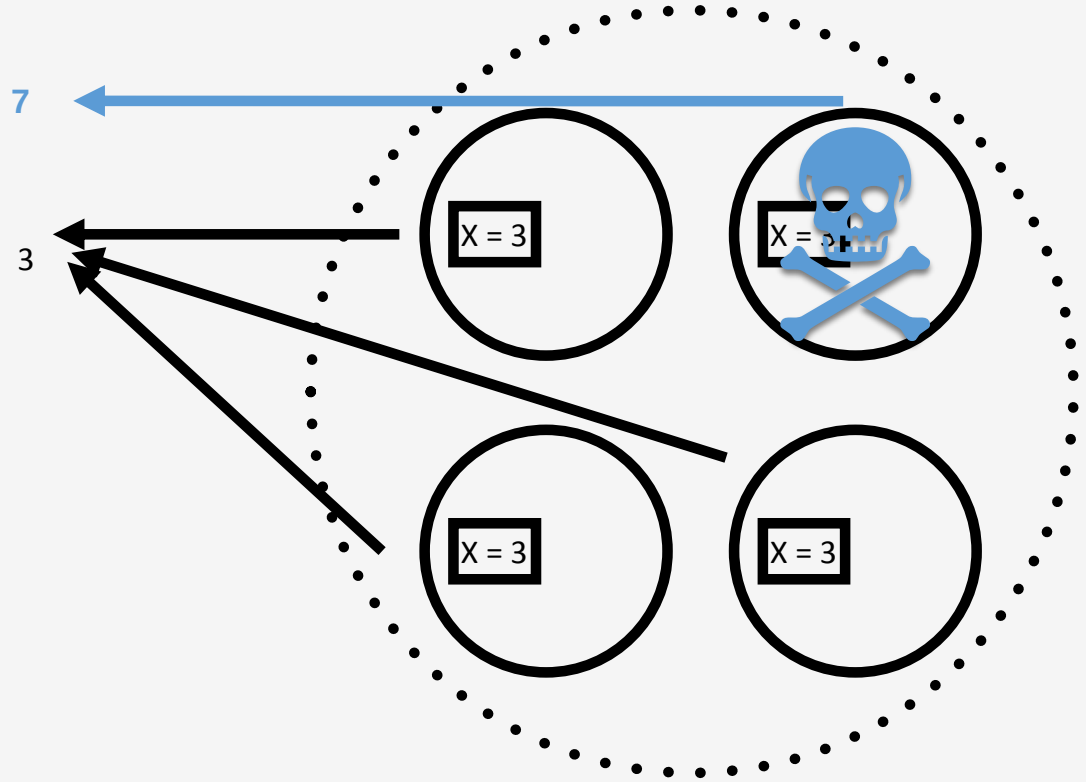
Byzantine Tolerance



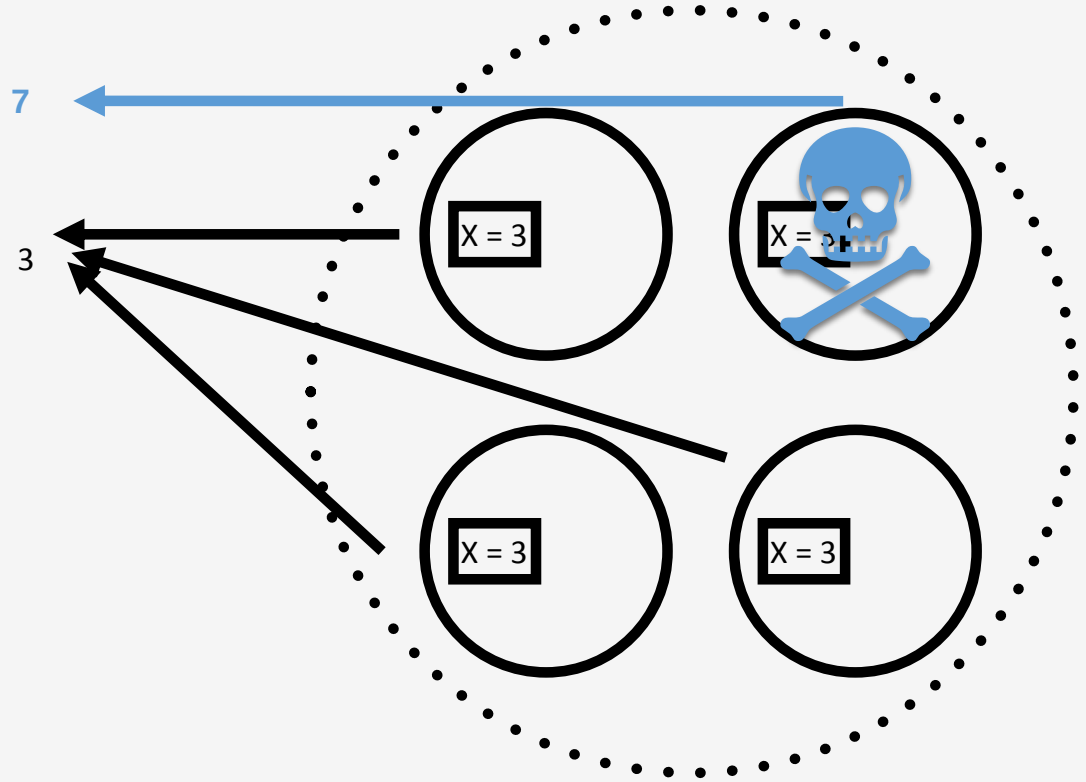
Byzantine Tolerance



Byzantine Tolerance

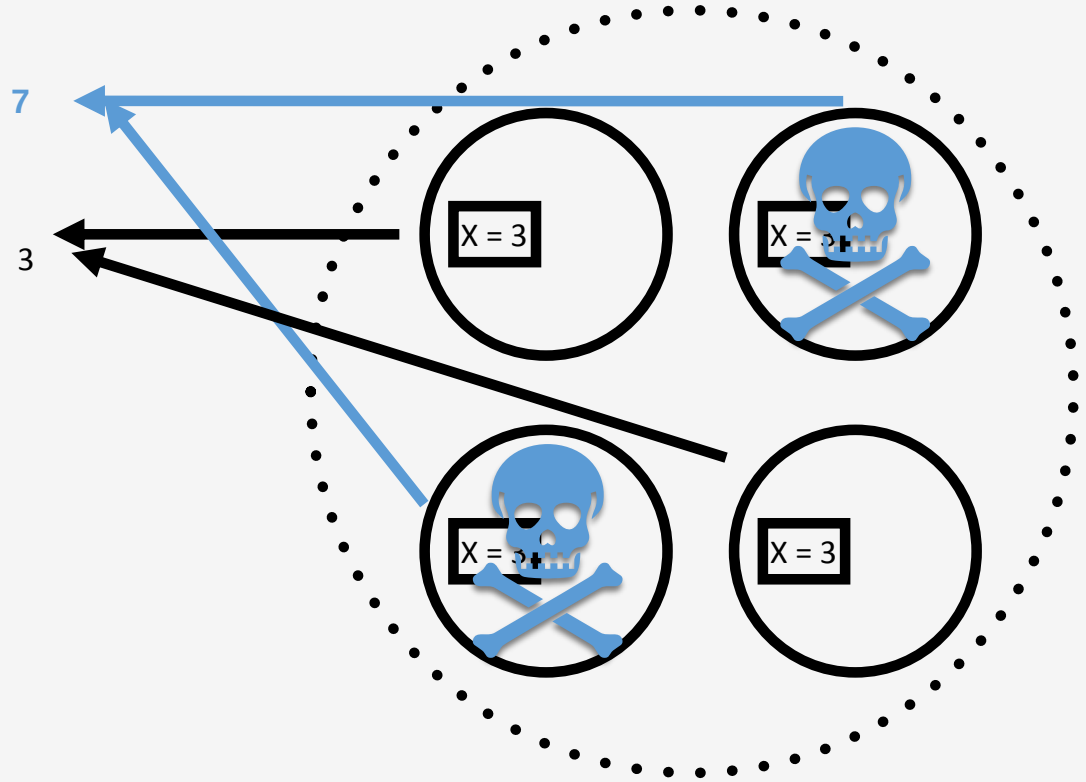


Byzantine Tolerance



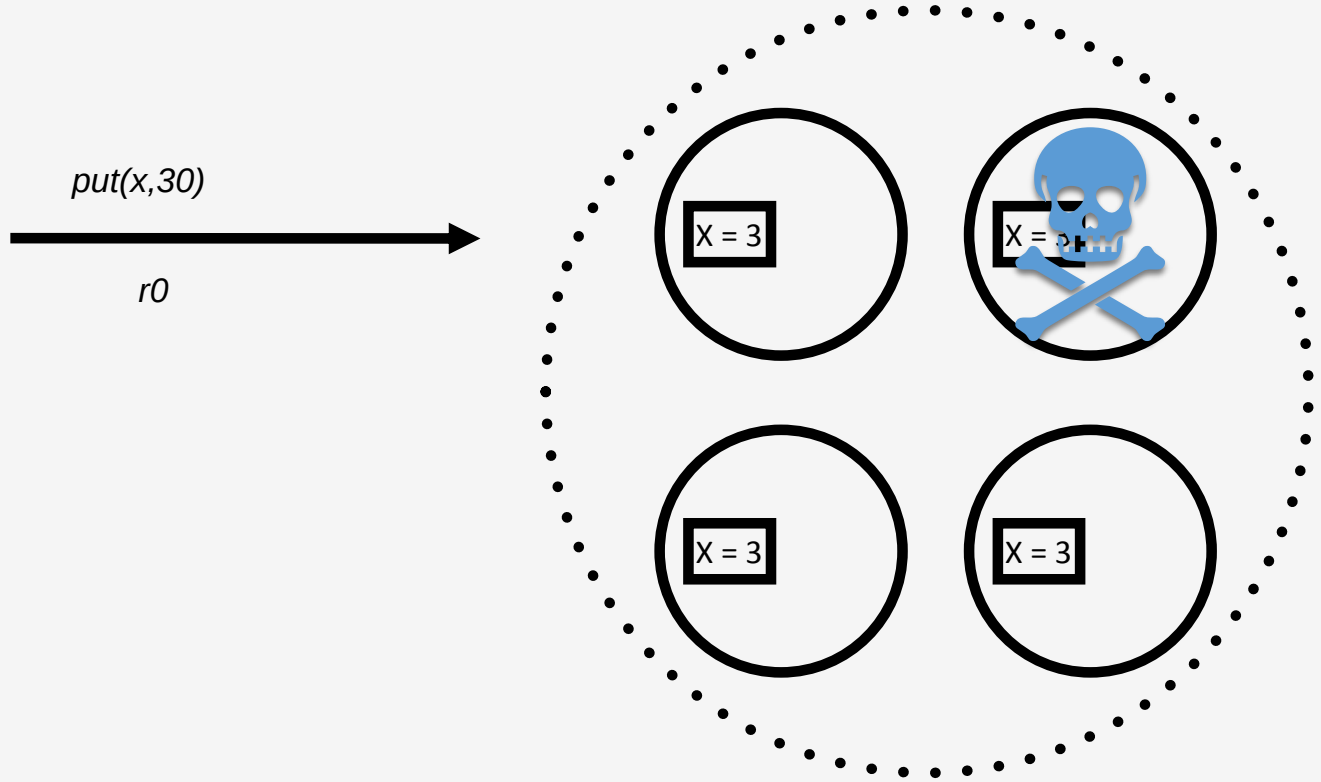
Client trusts the majority =>
Need majority to participate in replication

Byzantine Tolerance

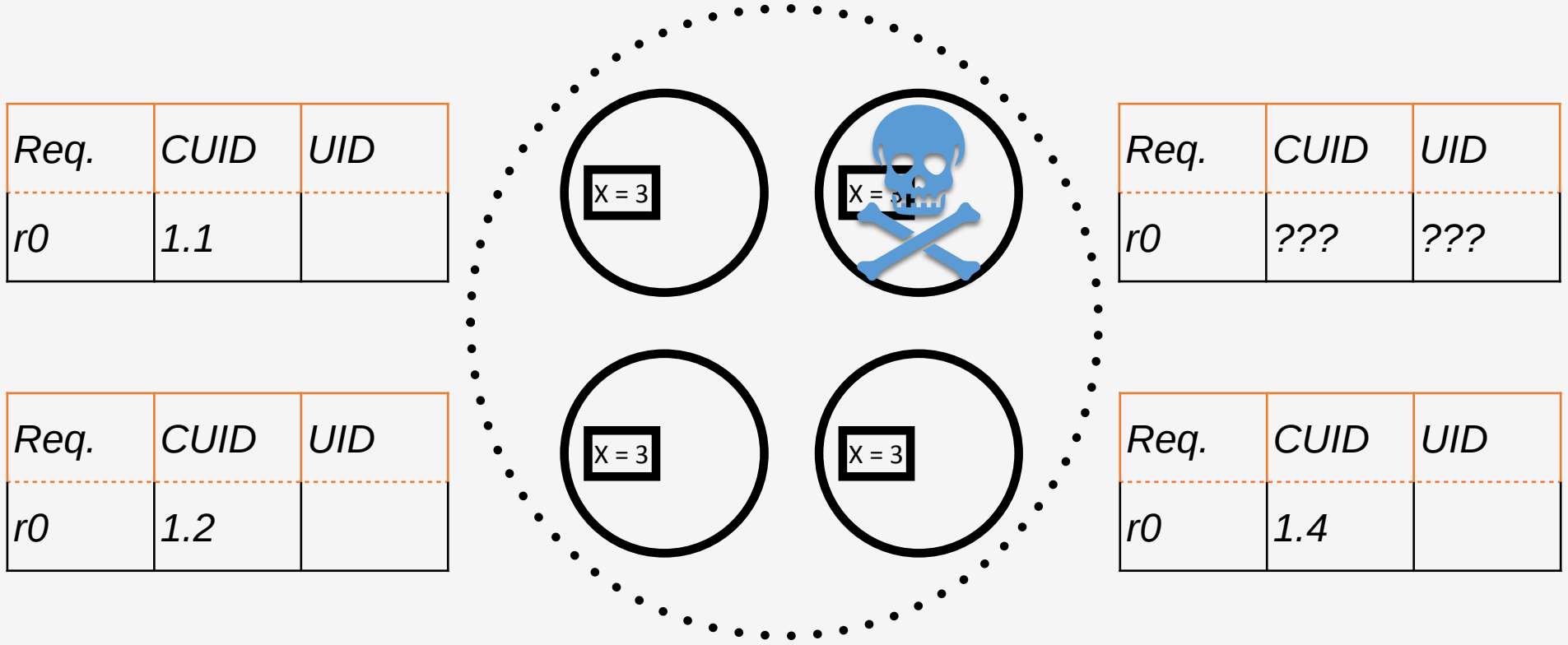


Who to trust?? 3 or 7?

Byzantine Tolerance



Byzantine Tolerance

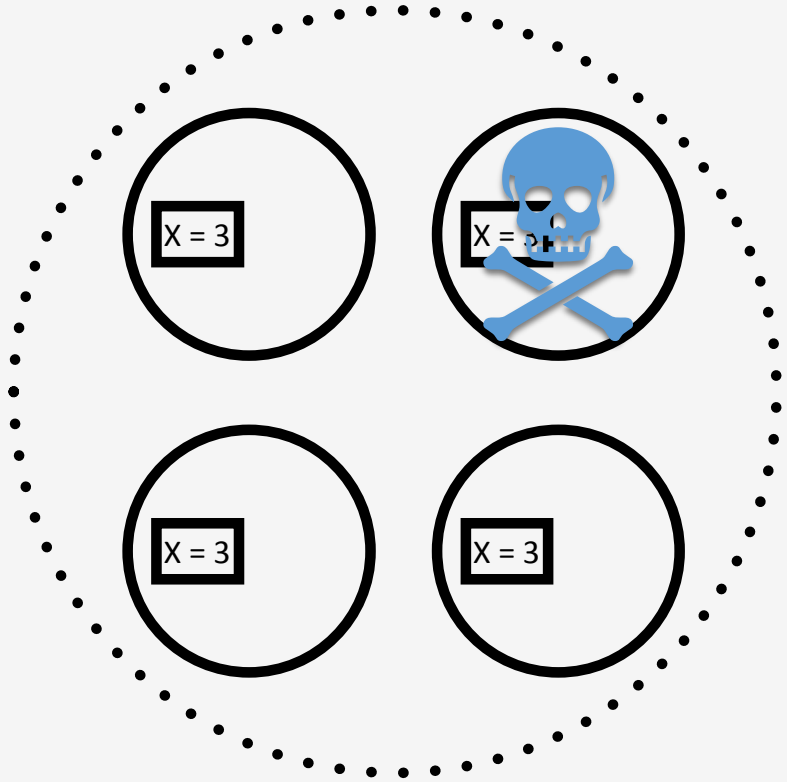


1) Propose Candidates

Byzantine Tolerance : a) No response

Req.	CUID	UID
r0	1.1	

Req.	CUID	UID
r0	1.2	



Req.	CUID	UID
r0	???	???

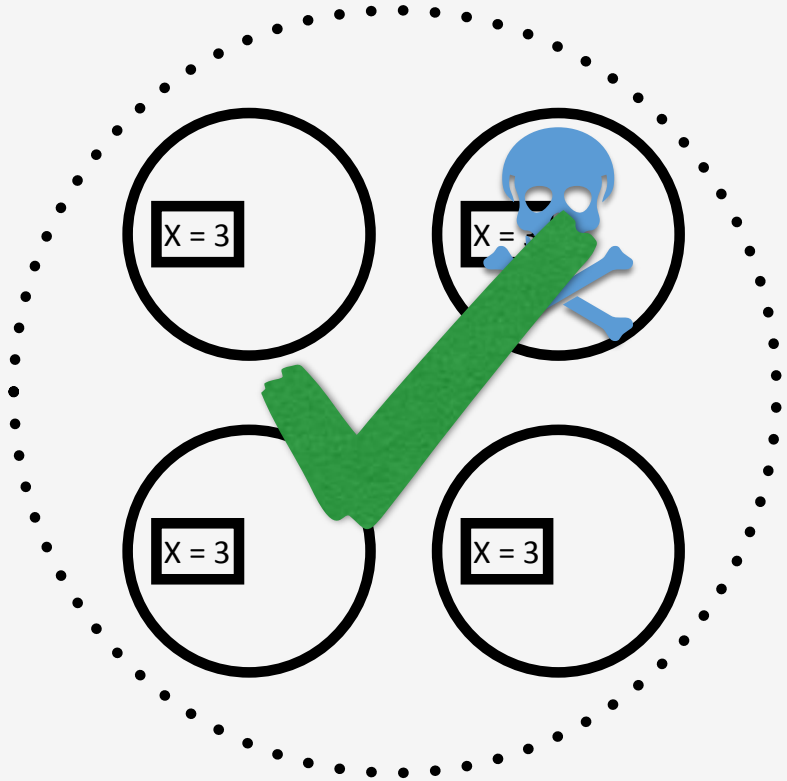
Req.	CUID	UID
r0	1.4	

a) Wait for majority candidates
Timeout long requests & notify others

Byzantine Tolerance
a) No response

Req.	CUID	UID
r0	1.1	1.4

Req.	CUID	UID
r0	1.2	1.4

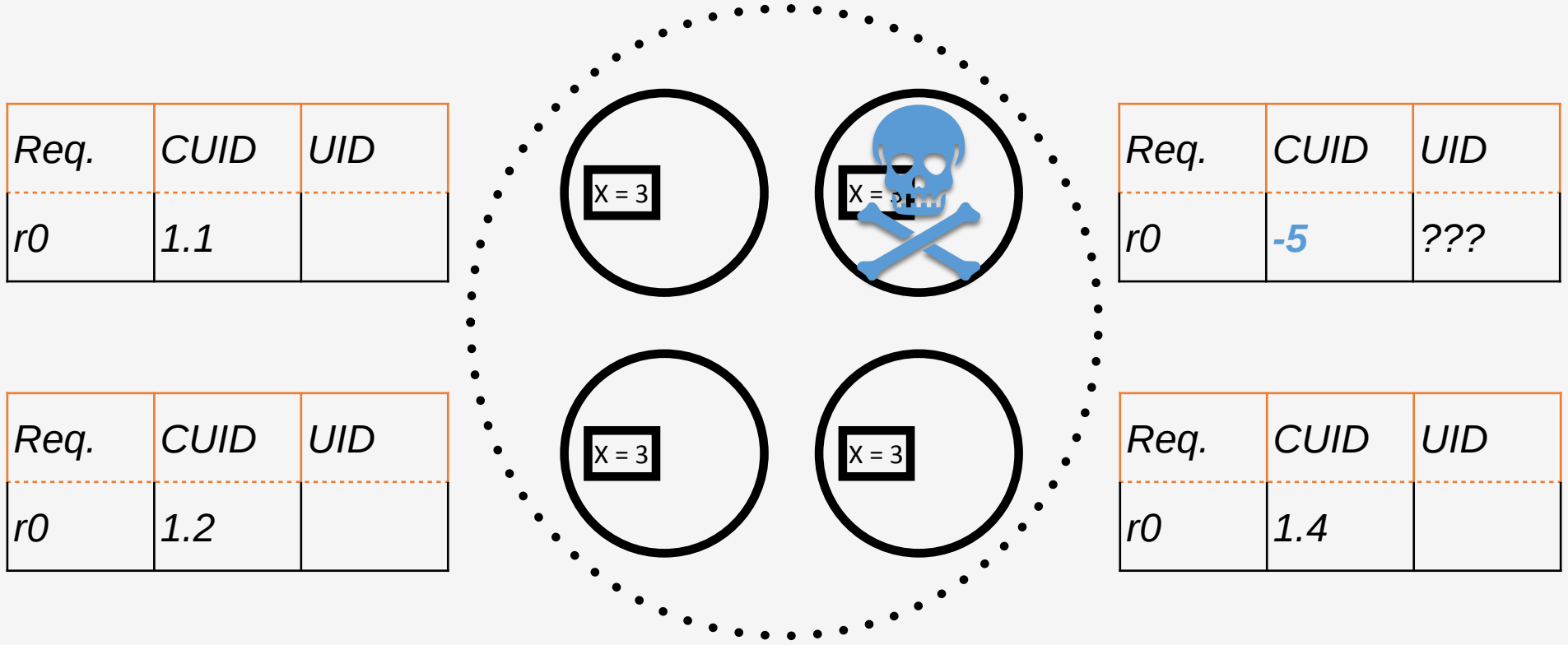


Req.	CUID	UID
r0	???	???

Req.	CUID	UID
r0	1.4	1.4

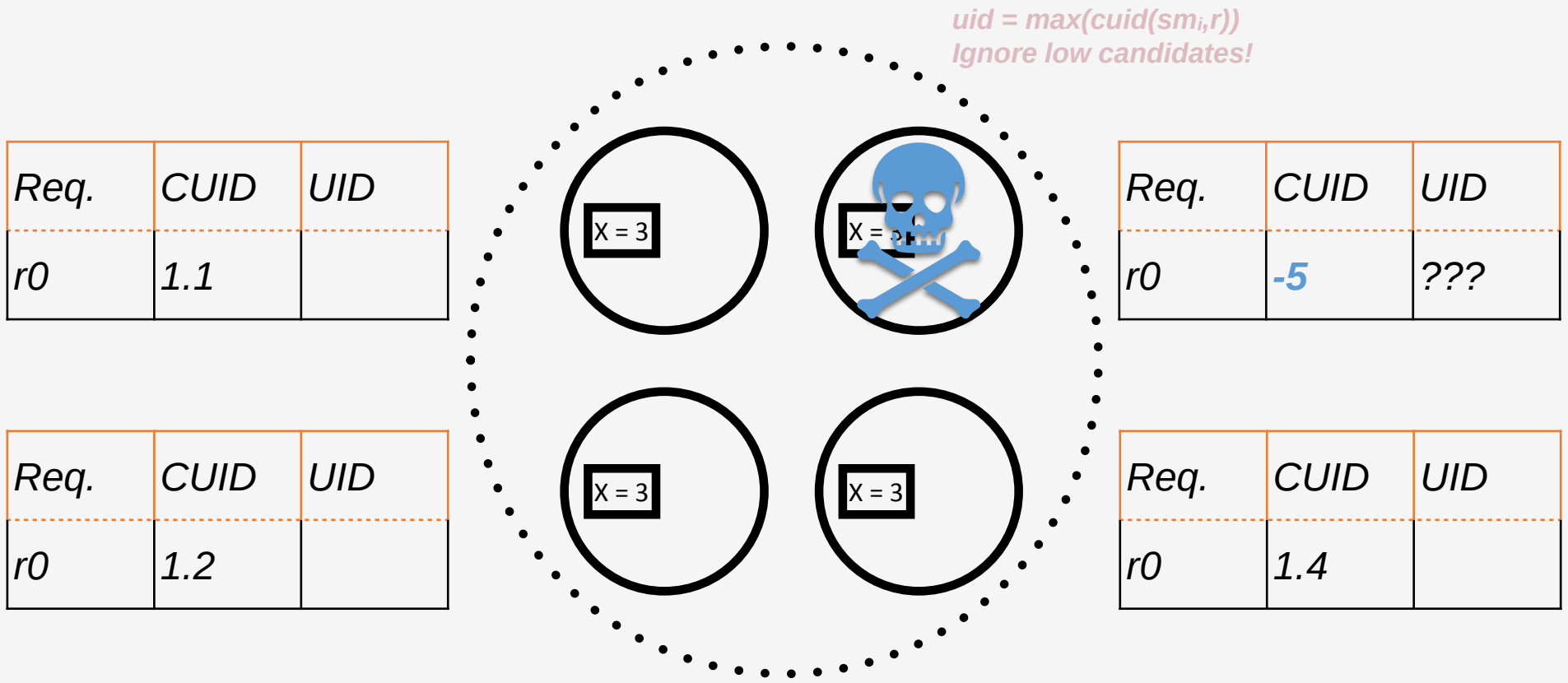
a) *Accept r0*

Byzantine Tolerance: Small ID



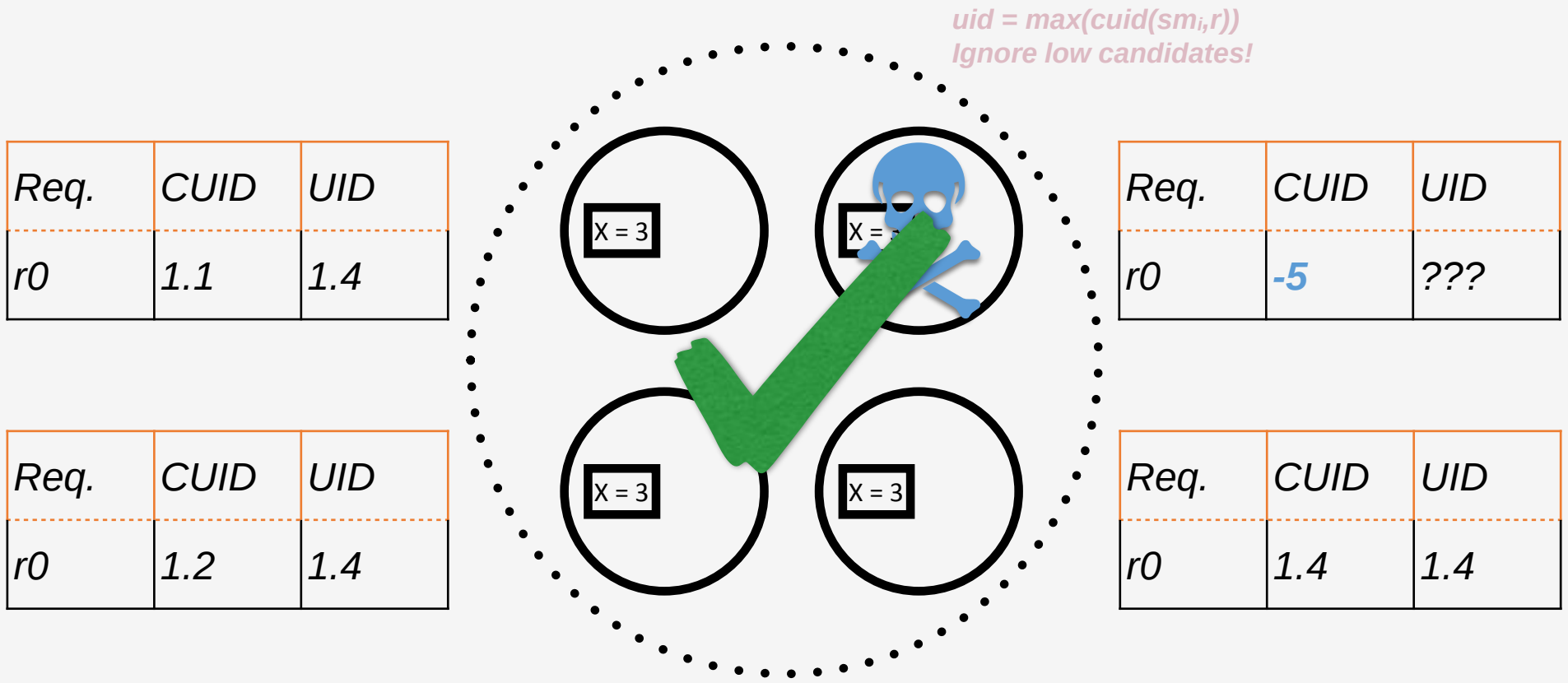
1) Propose Candidates

Byzantine Tolerance: Small ID



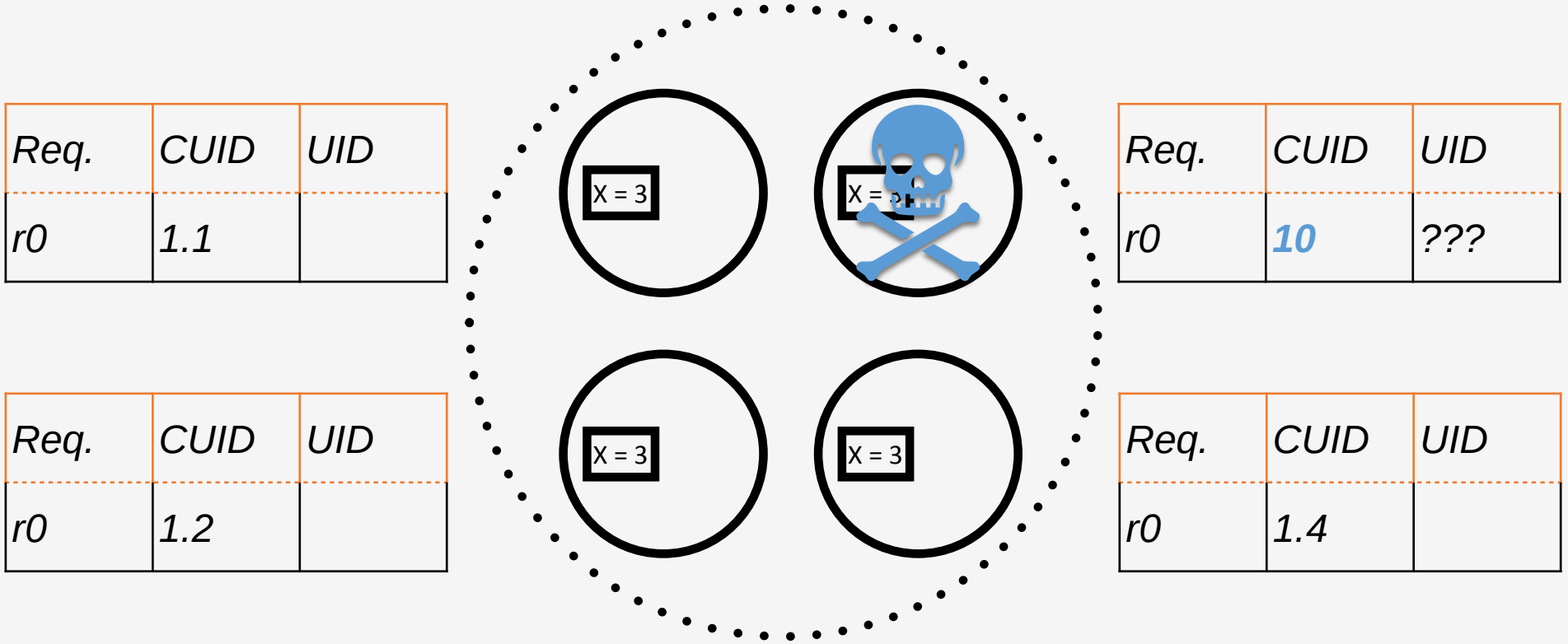
2) *Accept r0*

Byzantine Tolerance: Small ID



2) *Accept r0*

Byzantine Tolerance: Large ID



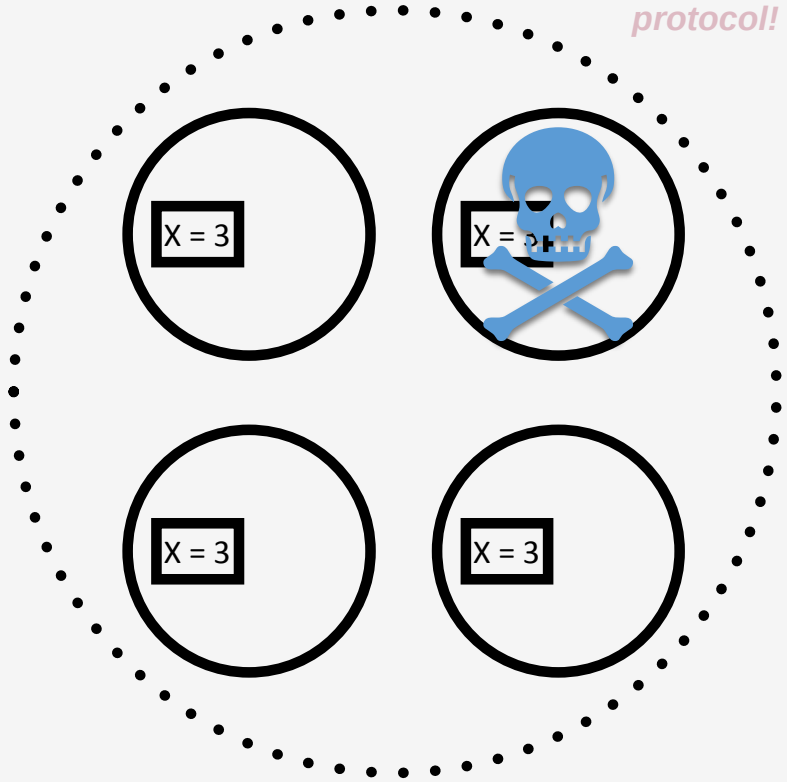
1) Propose Candidates

Byzantine Tolerance: Large ID

Large numbers follow protocol!

Req.	CUID	UID
r0	1.1	

Req.	CUID	UID
r0	1.2	



Req.	CUID	UID
r0	**10**	???

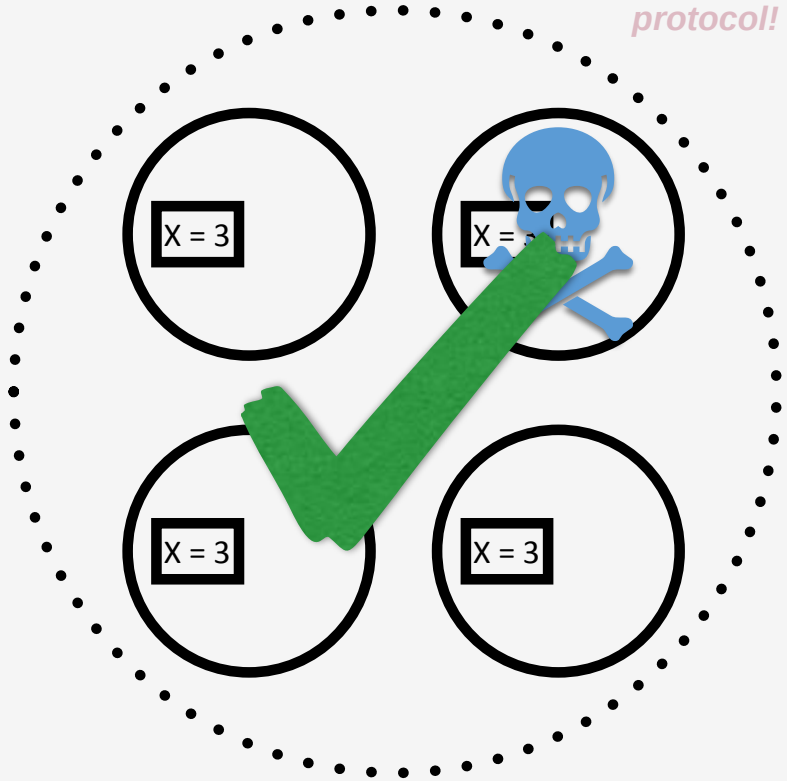
Req.	CUID	UID
r0	1.4	

Byzantine Tolerance: Large ID

Large numbers follow protocol!

Req.	CUID	UID
r0	1.1	10

Req.	CUID	UID
r0	1.2	10



Req.	CUID	UID
r0	**10**	???

Req.	CUID	UID
r0	1.4	10

2) *Accept r0*

Fault Tolerance

- Byzantine Failures
  - 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

How Blockchains differ from SMRs

- not only one, but many distributed applications run concurrently
- applications may be deployed dynamically and by anyone
- the application code is untrusted, potentially even malicious

How Fabric's Execute-order-validate works

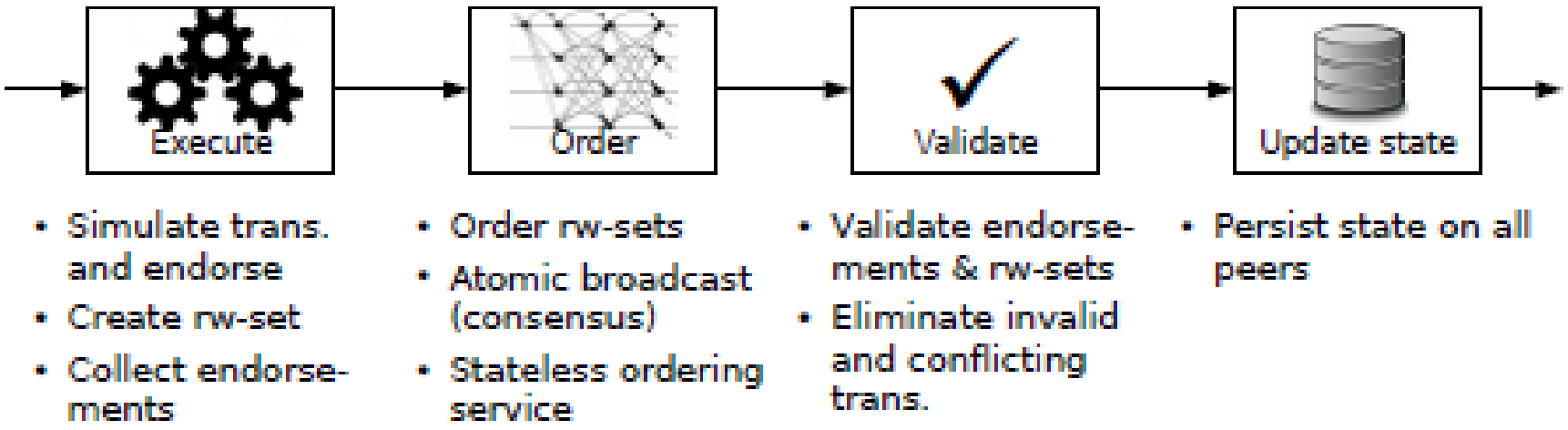


Figure 2: Execute-order-validate architecture of Fabric (*rw-set* means a readset and writeset as explained in Sec. 3.2).

Fabric Overview

- Fabric is a **distributed operating system** for permissioned blockchains that
  - executes distributed applications written in general purpose programming languages (e.g., Go, Java, Node.js)
  - securely tracks its execution history in an append-only replicated ledger data structure and has no cryptocurrency built in

Smart Contract in Fabric

- A smart contract, called **chaincode**, which is program code that implements the application logic and runs during the execution phase.
- The chaincode is the central part of a distributed application in Fabric and may be written by an untrusted developer.
- Special chaincodes exist for managing the blockchain system and maintaining parameters, collectively called **system chaincodes**

Endorsement Policy

- An **endorsement policy** that is evaluated in the validation phase.
- Endorsement policies cannot be chosen or modified by untrusted application developers
- An endorsement policy acts as a static library for transaction validation in Fabric, which can merely be parameterized by the chaincode.
- Only designated administrators may have a permission to modify endorsement policies through system management functions.
- A typical endorsement policy lets the chaincode specify the endorsers for a transaction in the form of a set of peers that are necessary for endorsement
- it uses a logical expression on sets, such as “three out of five” or “(A ∧ B) ∨ C.” Custom endorsement policies may implement arbitrary logic

Fabric Transaction flow

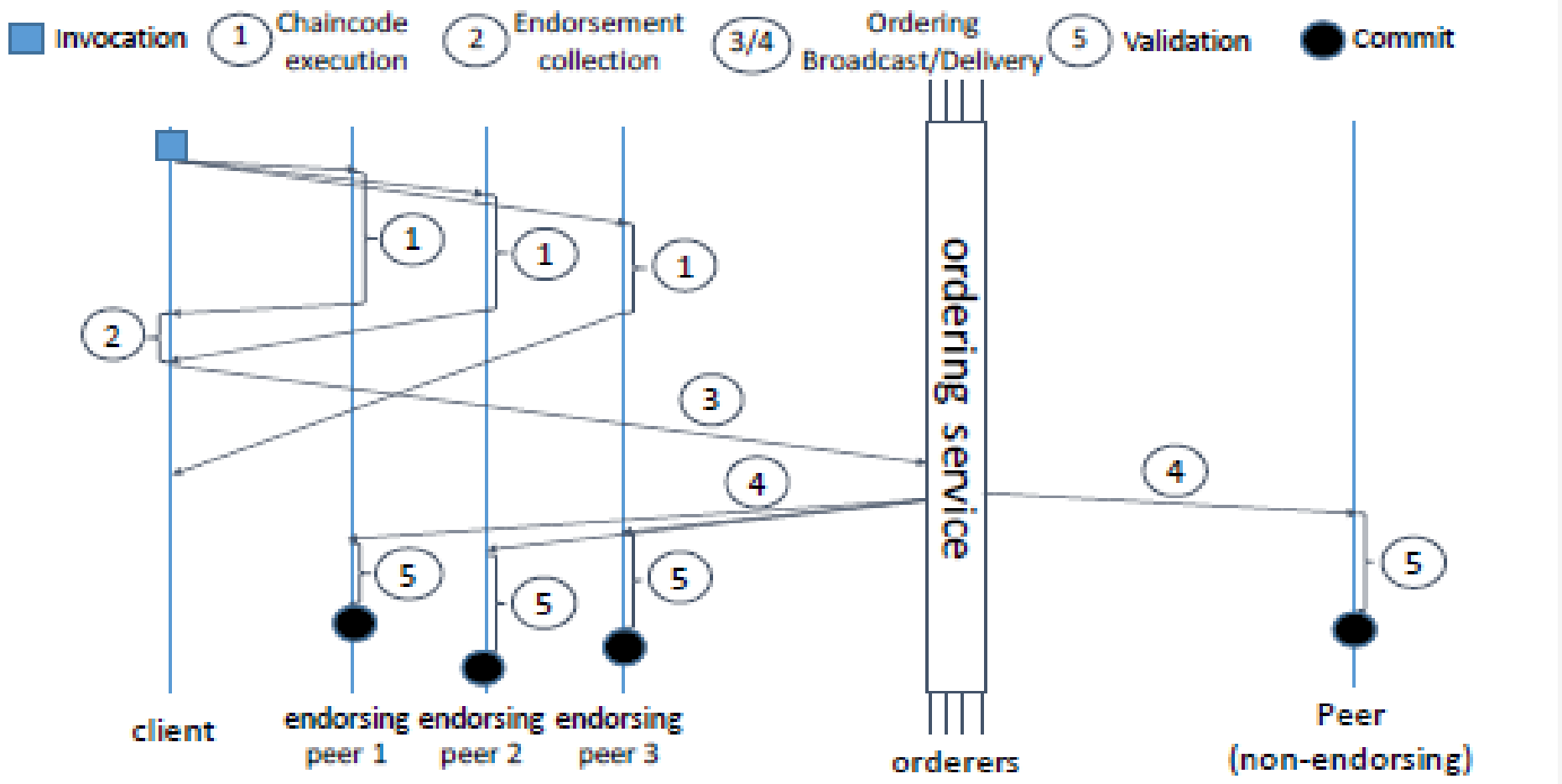


Figure 4: Fabric high level transaction flow.

Summary of Transaction Flow

- A client sends transactions to the peers specified by the endorsement policy
- Each transaction is then executed by specific peers and its output is recorded; this step is also called endorsement (no state change)
- After endorsement, transactions enter the ordering phase,
  - consensus protocol to produce a totally ordered sequence of endorsed transactions grouped in blocks.
- These are broadcast to all peers, with the (optional) help of gossip
- Each peer then validates the state changes from endorsed transactions with respect to the endorsement policy and the consistency of the execution in the validation phase (states updated)
- All peers validate the transactions in the same order and validation is deterministic.
- Fabric introduces a novel hybrid replication paradigm in the Byzantine model,
  - which combines passive replication (the pre-consensus computation of state updates) and
  - active replication (the post-consensus validation of execution results and state changes).