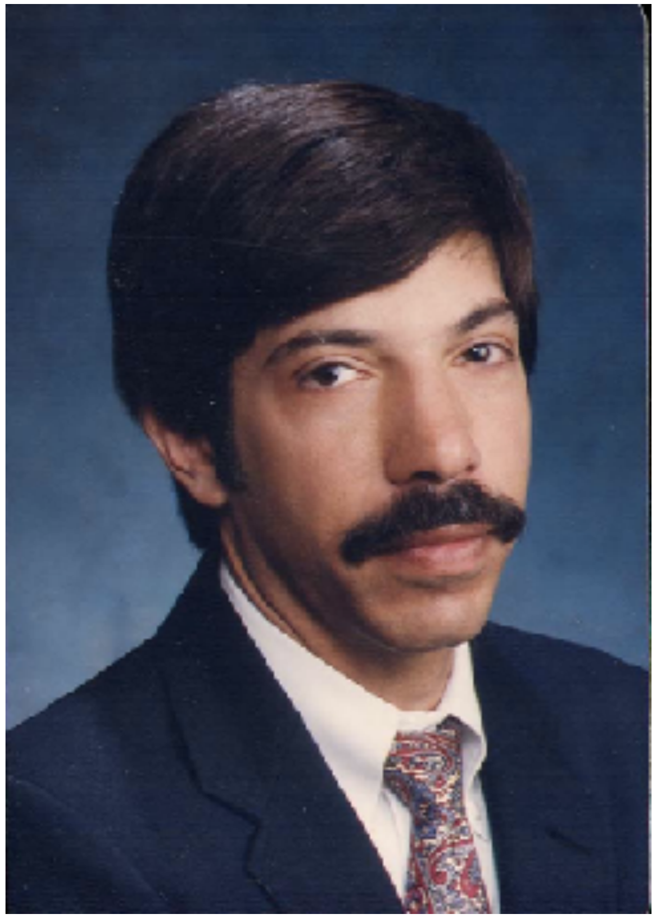
Fault-Tolerant State Machine Replication

Drew Zagieboylo

Authors

- Fred Schneider



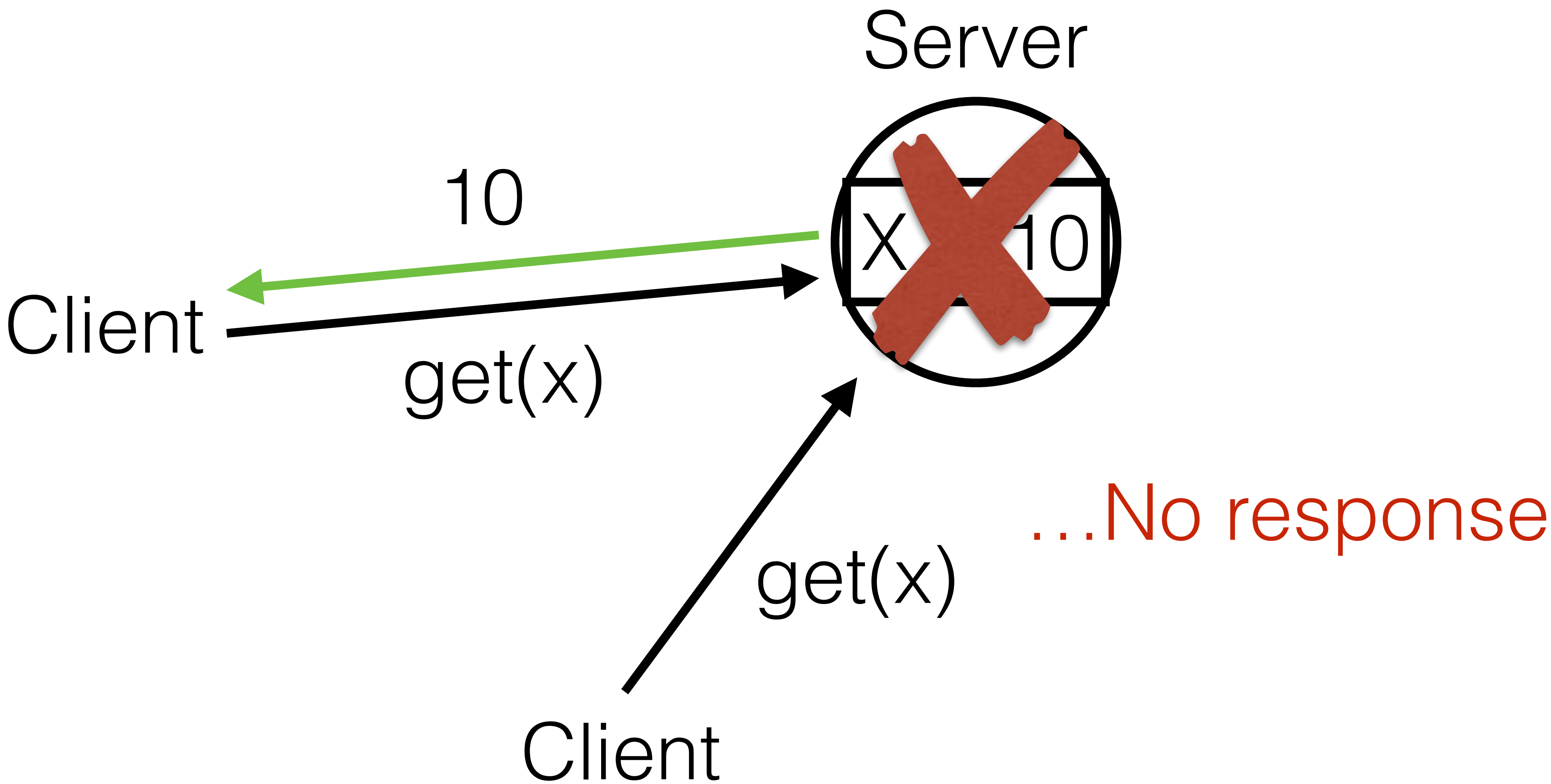
Takeaways

- 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*

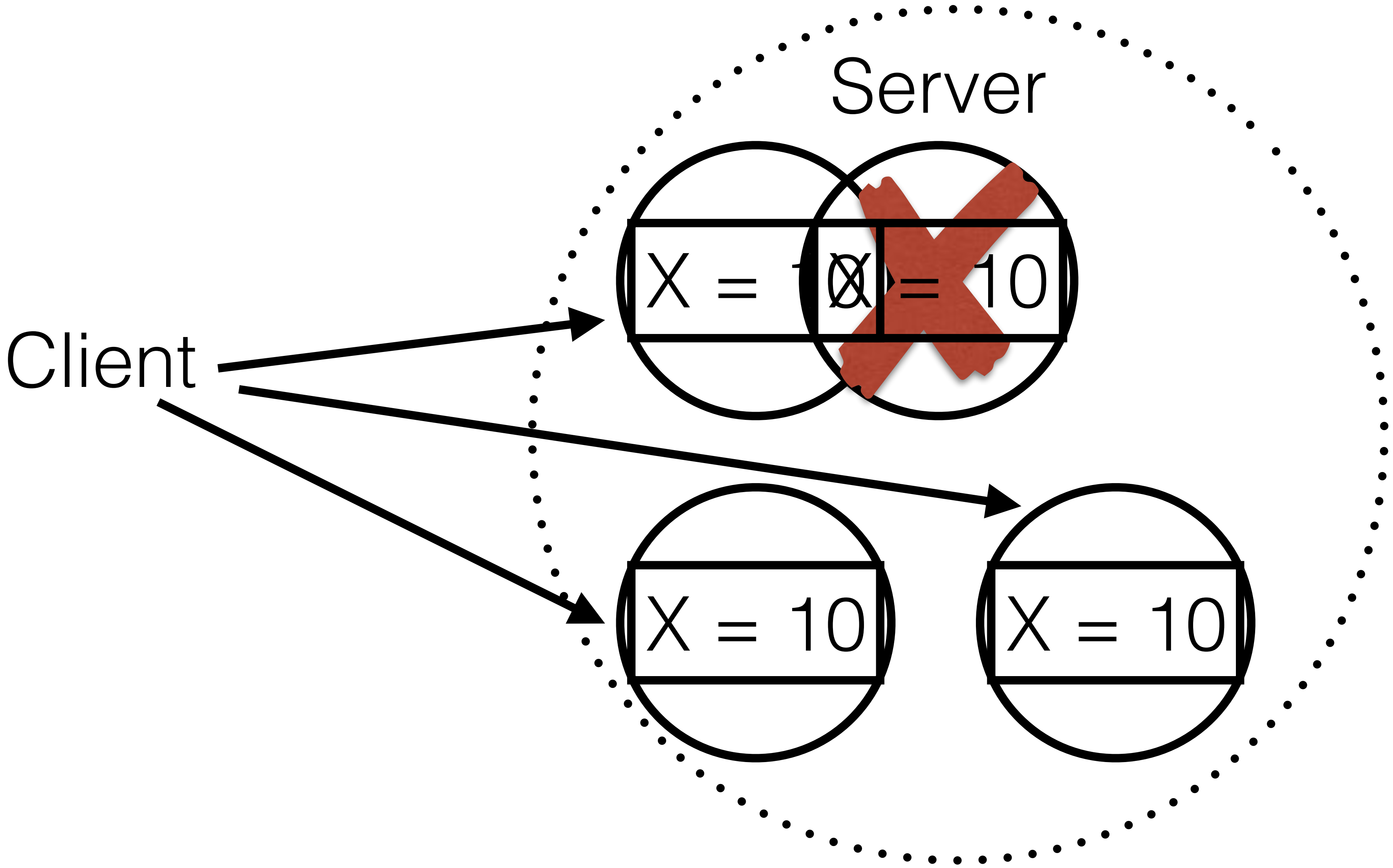
Outline

- Motivation
- State Machine Replication
- Implementation
- Fault Tolerance Requirements
- An Example - Chain Replication
- Evaluation

Motivation



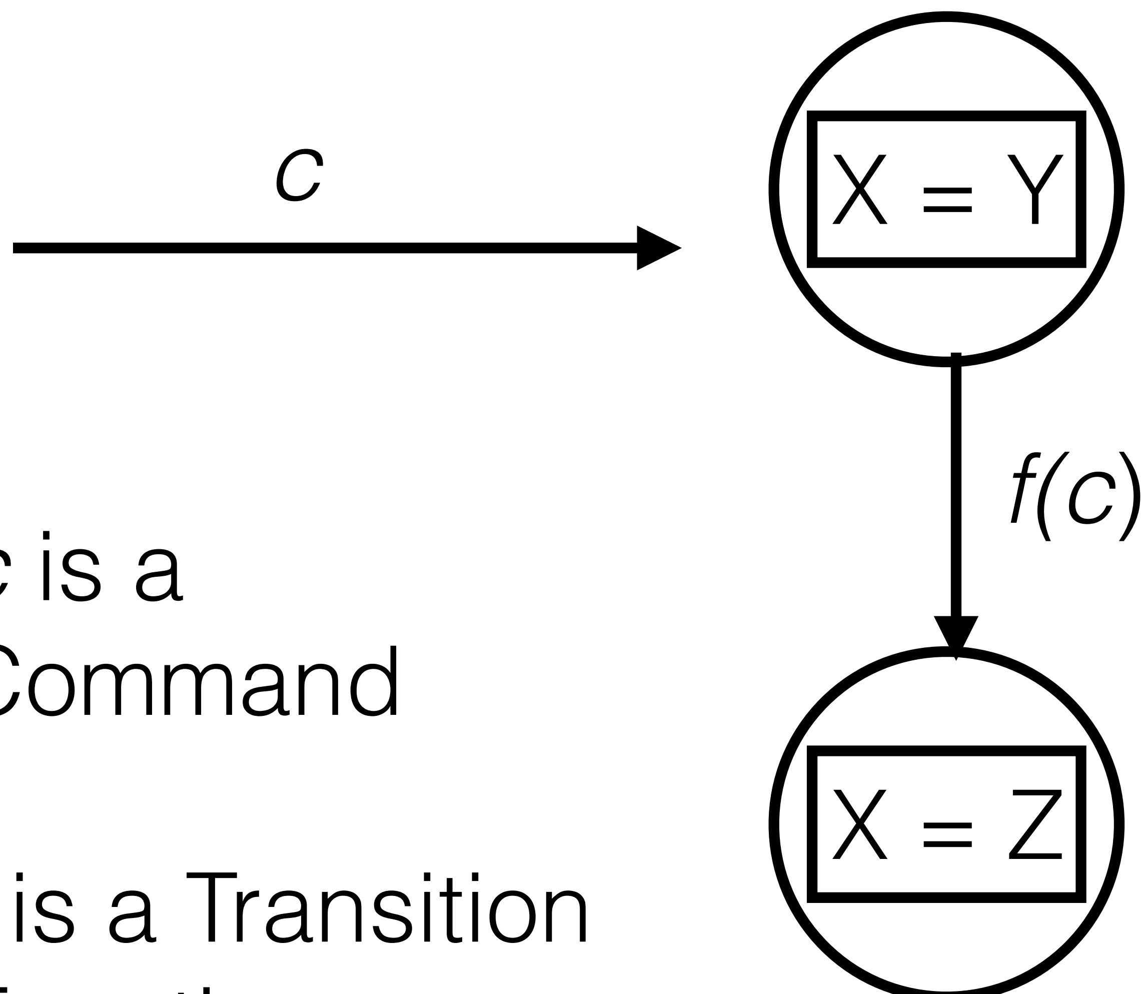
Motivation



Motivation

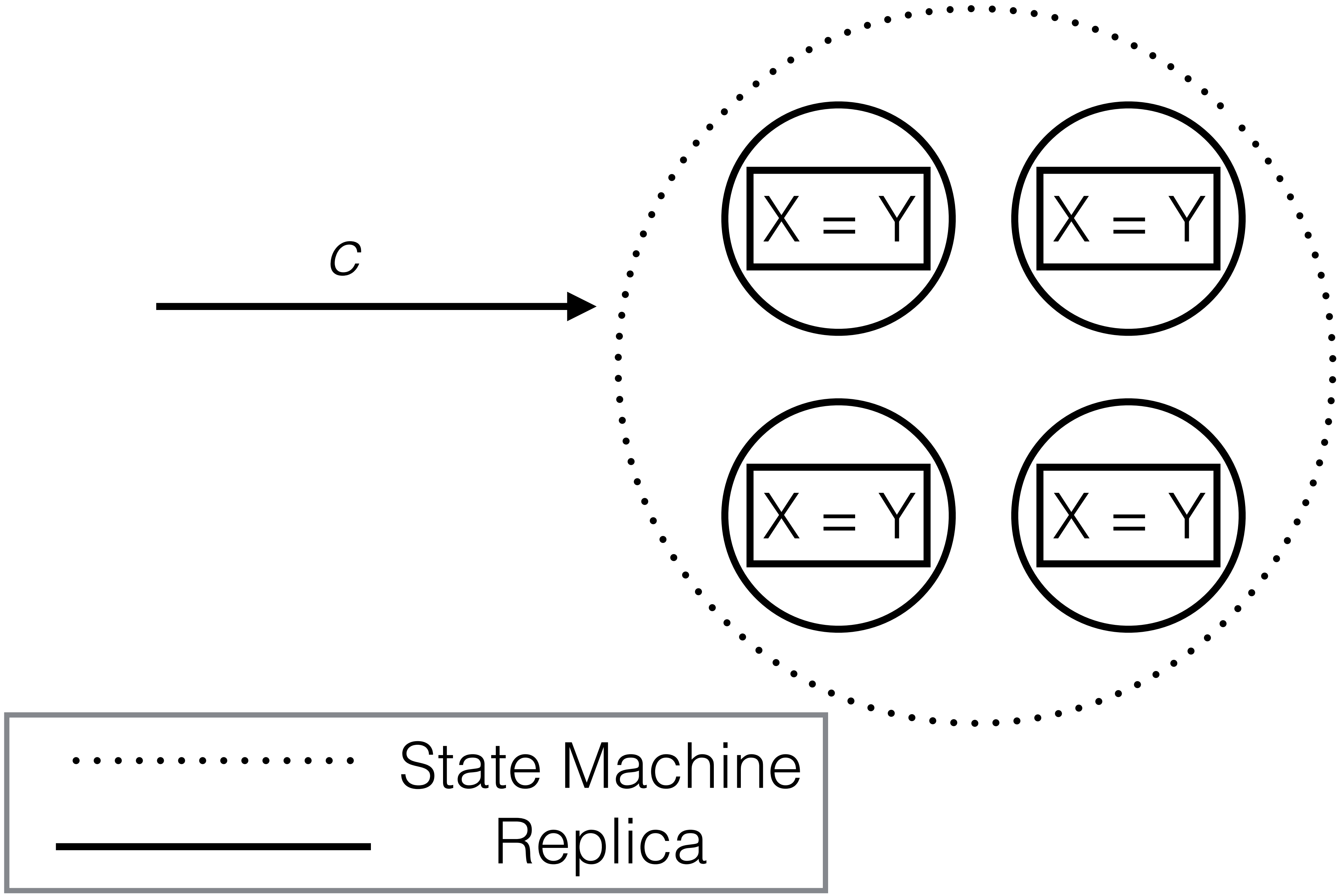
- Need replication for fault tolerance
- What happens in these scenarios without replication?
    - Storage - Disk Failure
    - Webservice - Network failure
- Be able to reason about failure tolerance
    - How badly can things go wrong and have our system continue to function?

State Machines



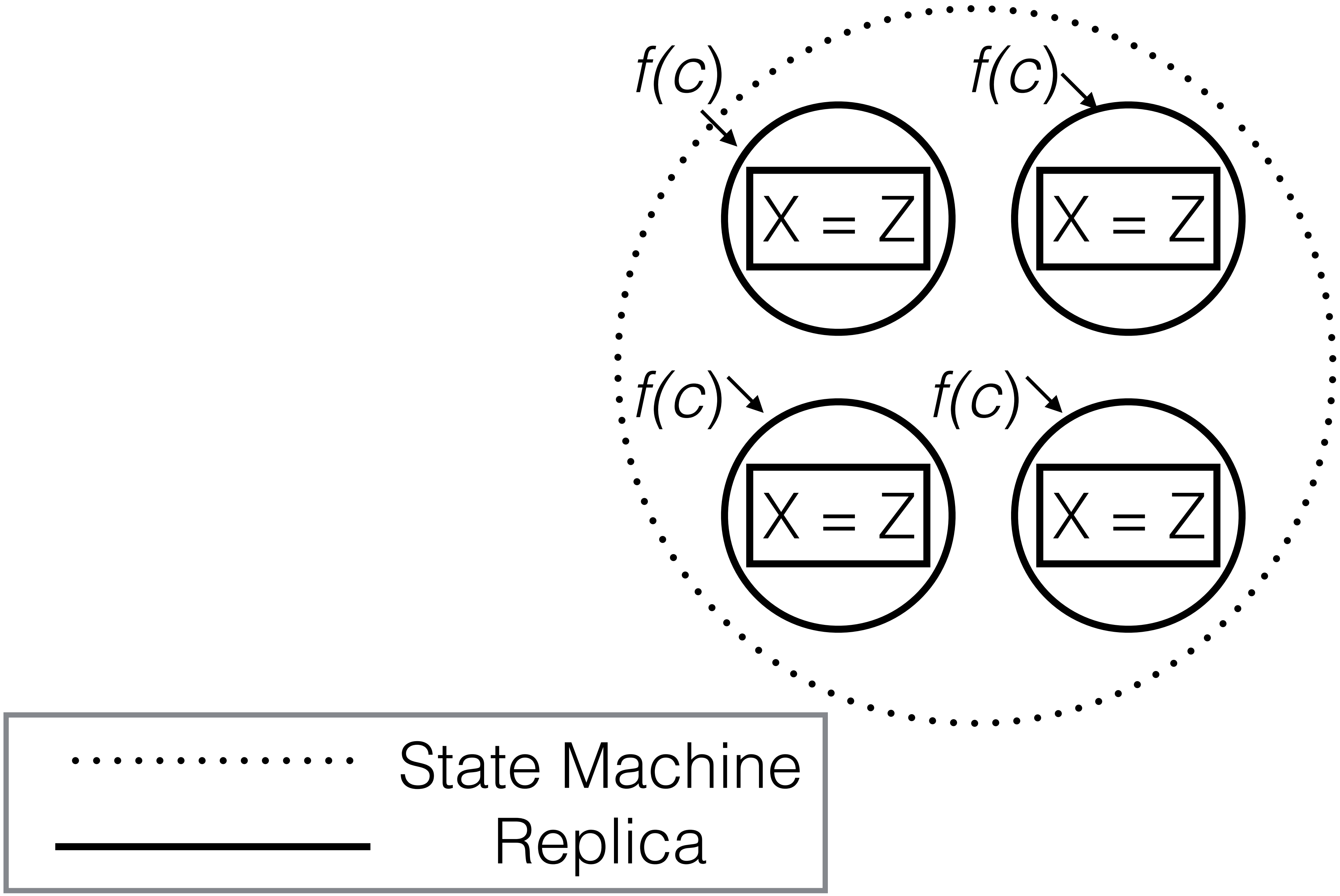
- *c* is a Command
- *f* is a Transition Function

State Machine Replication (SMR)



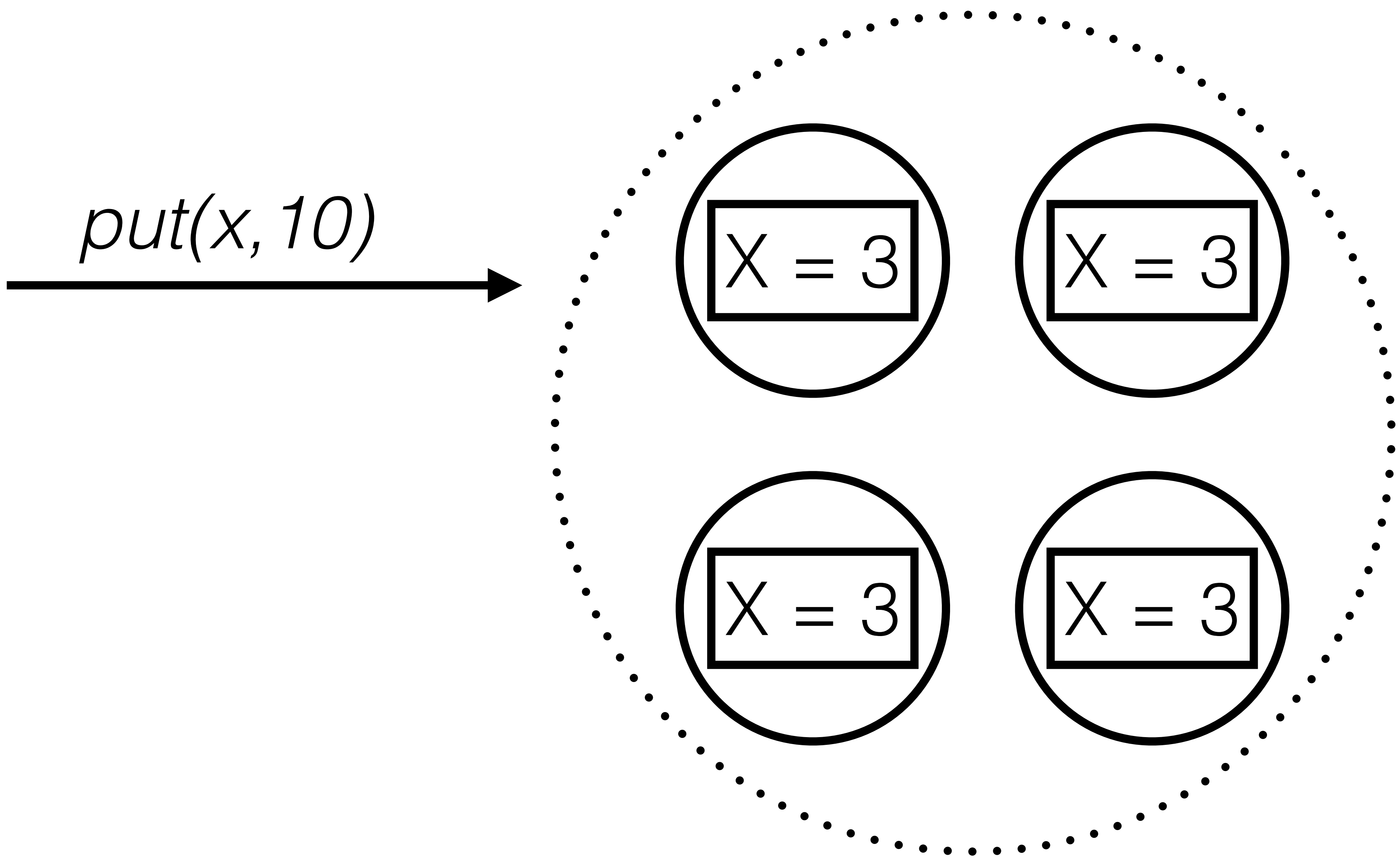
- The *State Machine Approach* to a fault tolerant distributed system
- Keep around *N* copies of the state machine

State Machine Replication (SMR)

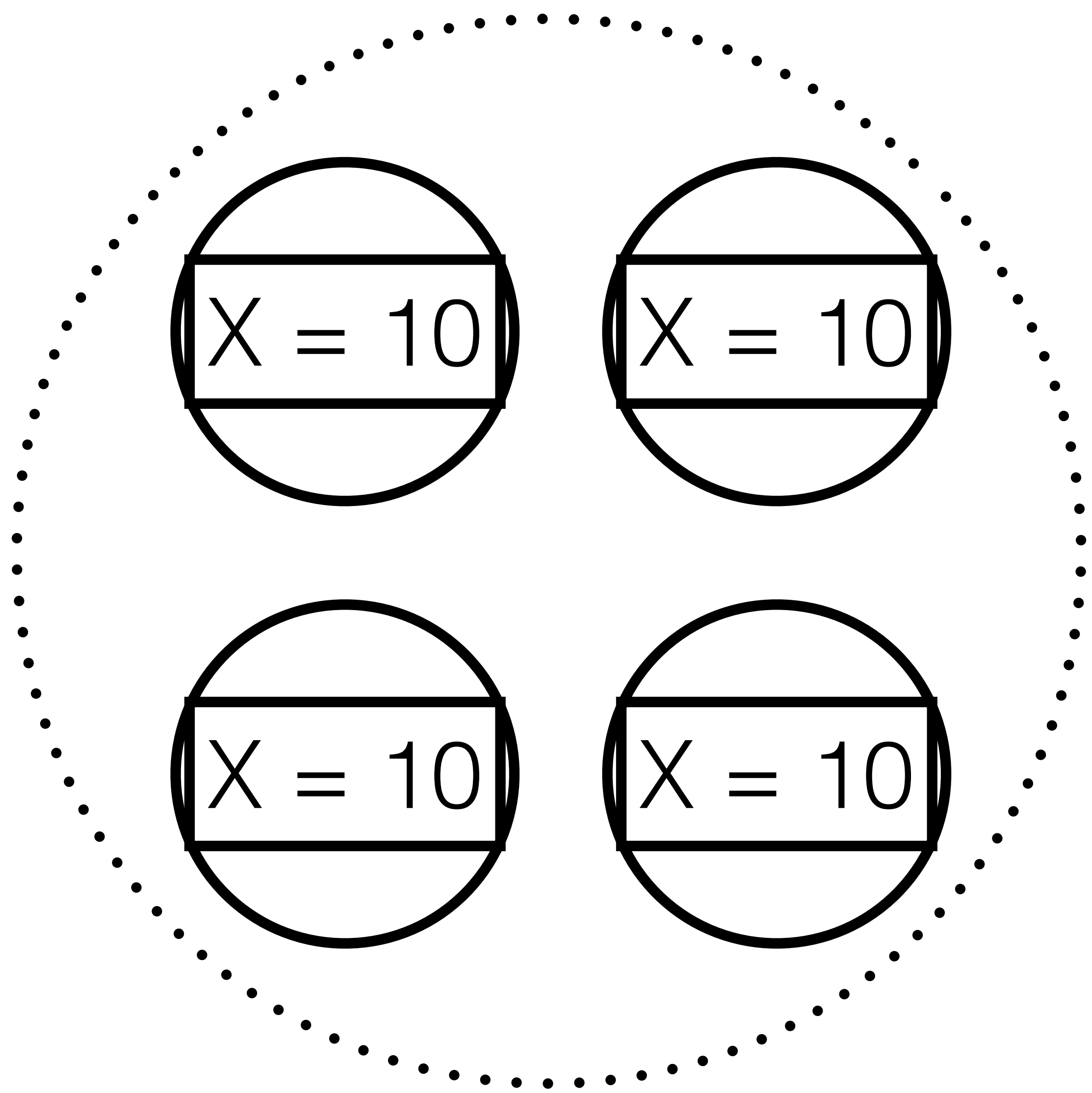


- The *State Machine Approach* to a fault tolerant distributed system
- Keep around *N* copies of the state machine

SMR Requirements

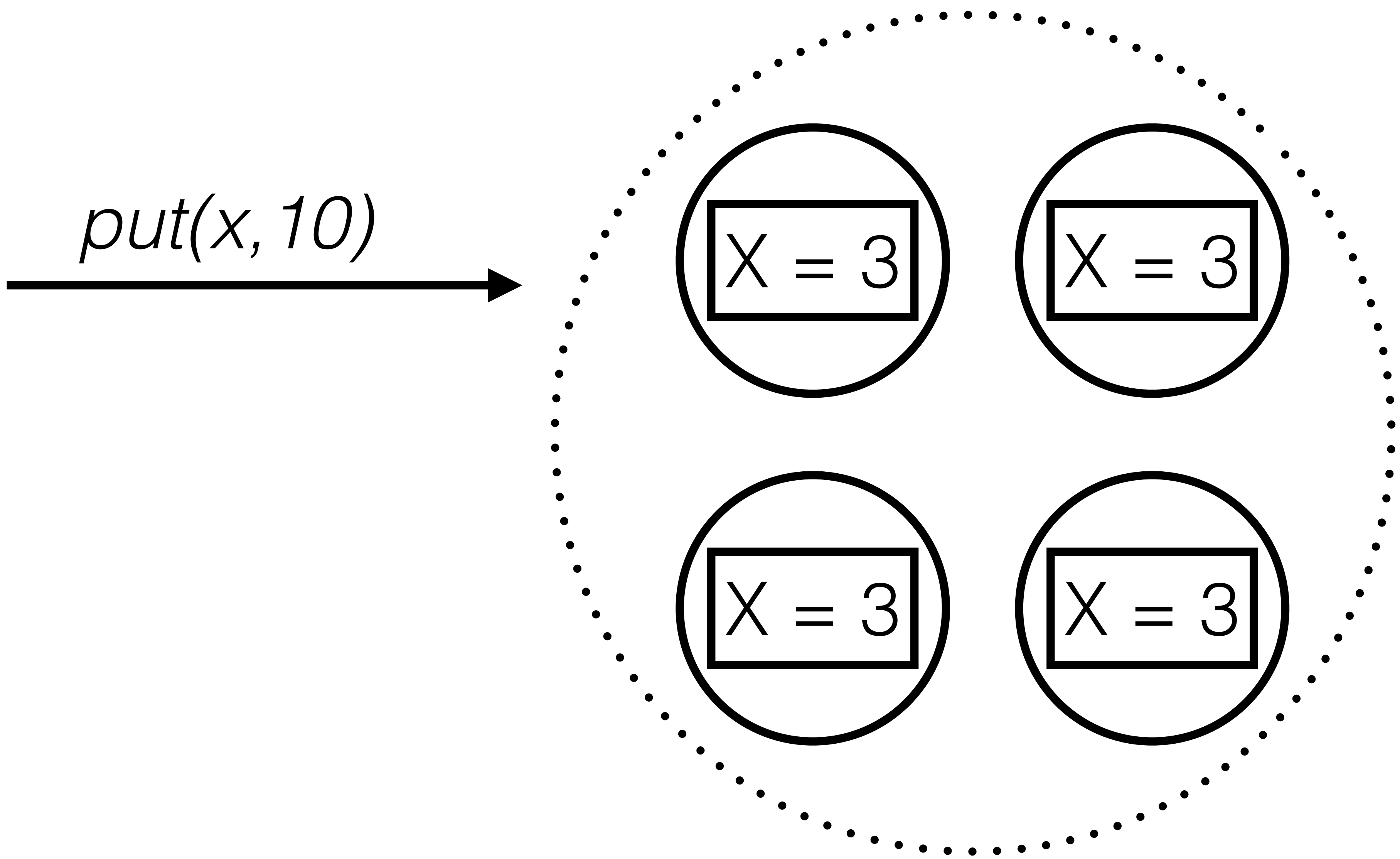


SMR Requirements

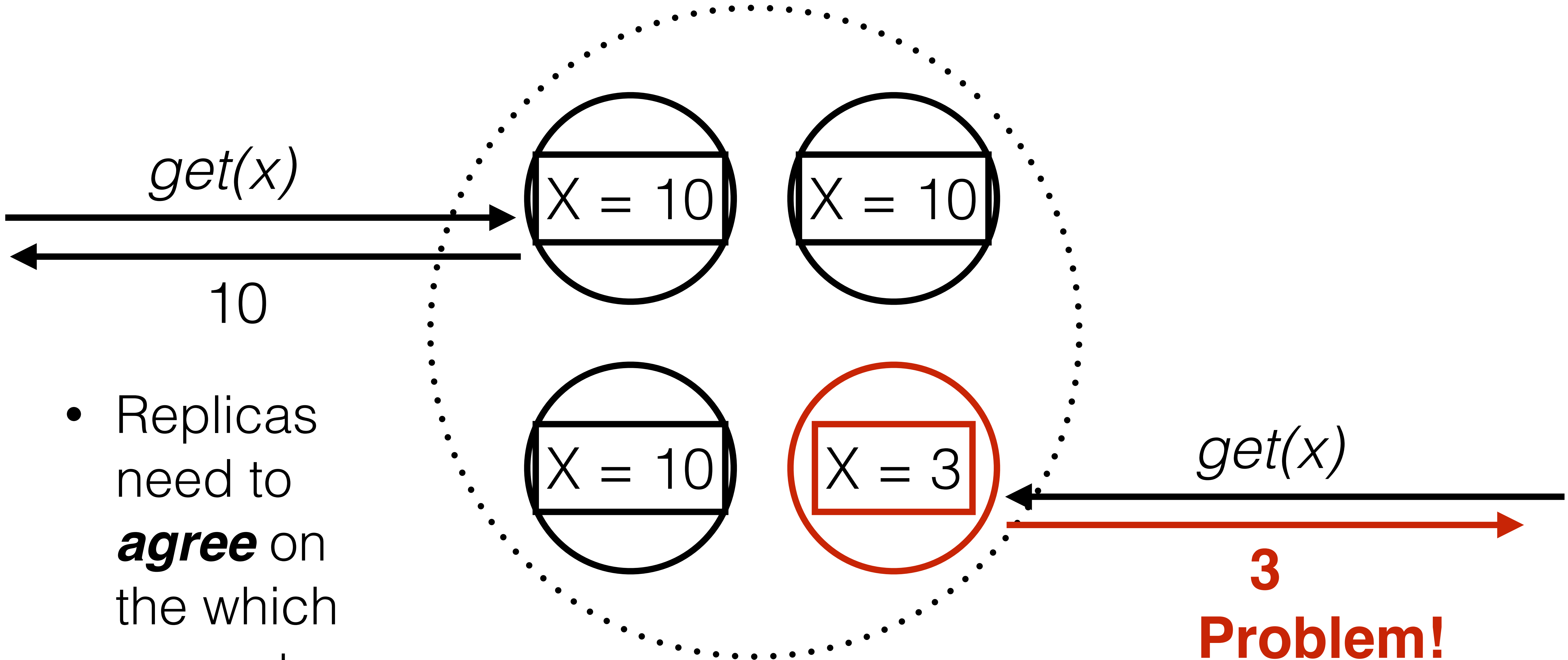


Great!

SMR Requirements

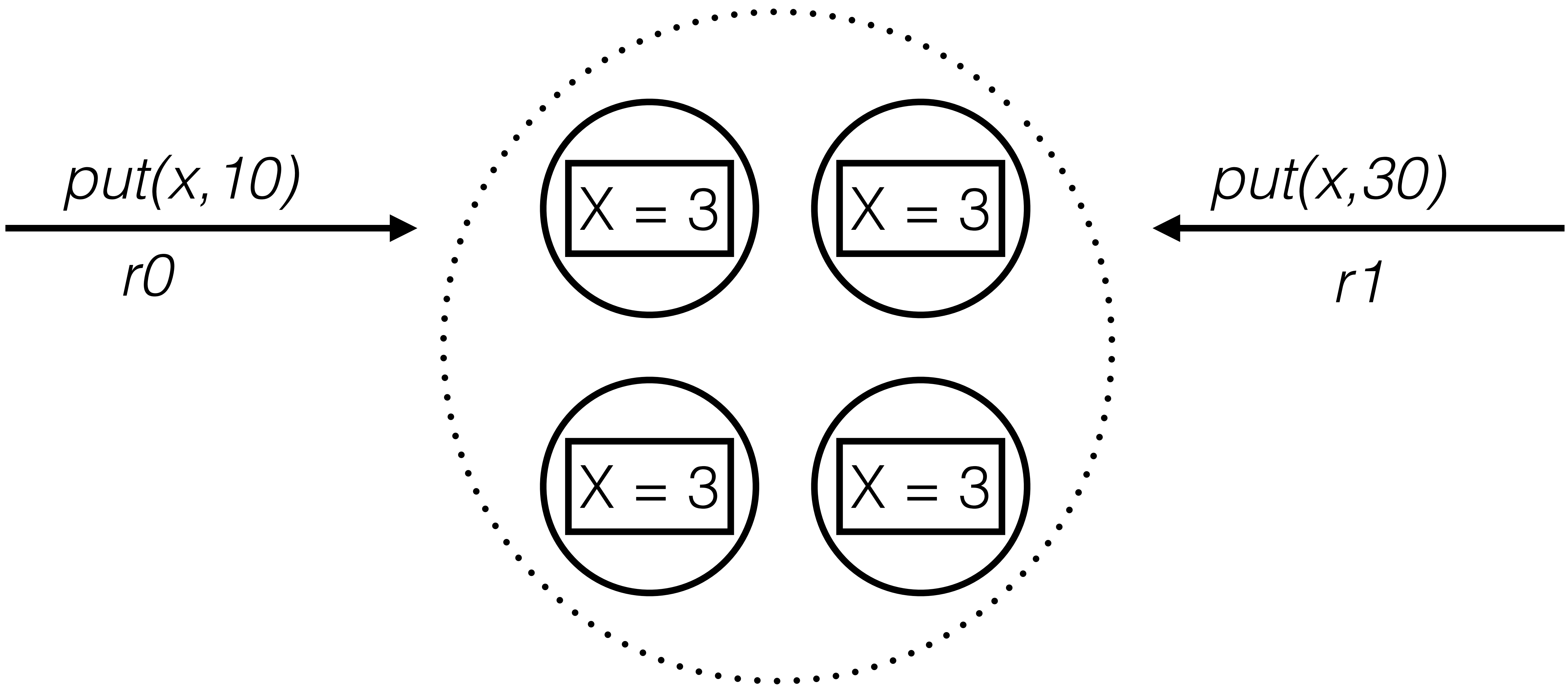


SMR Requirements

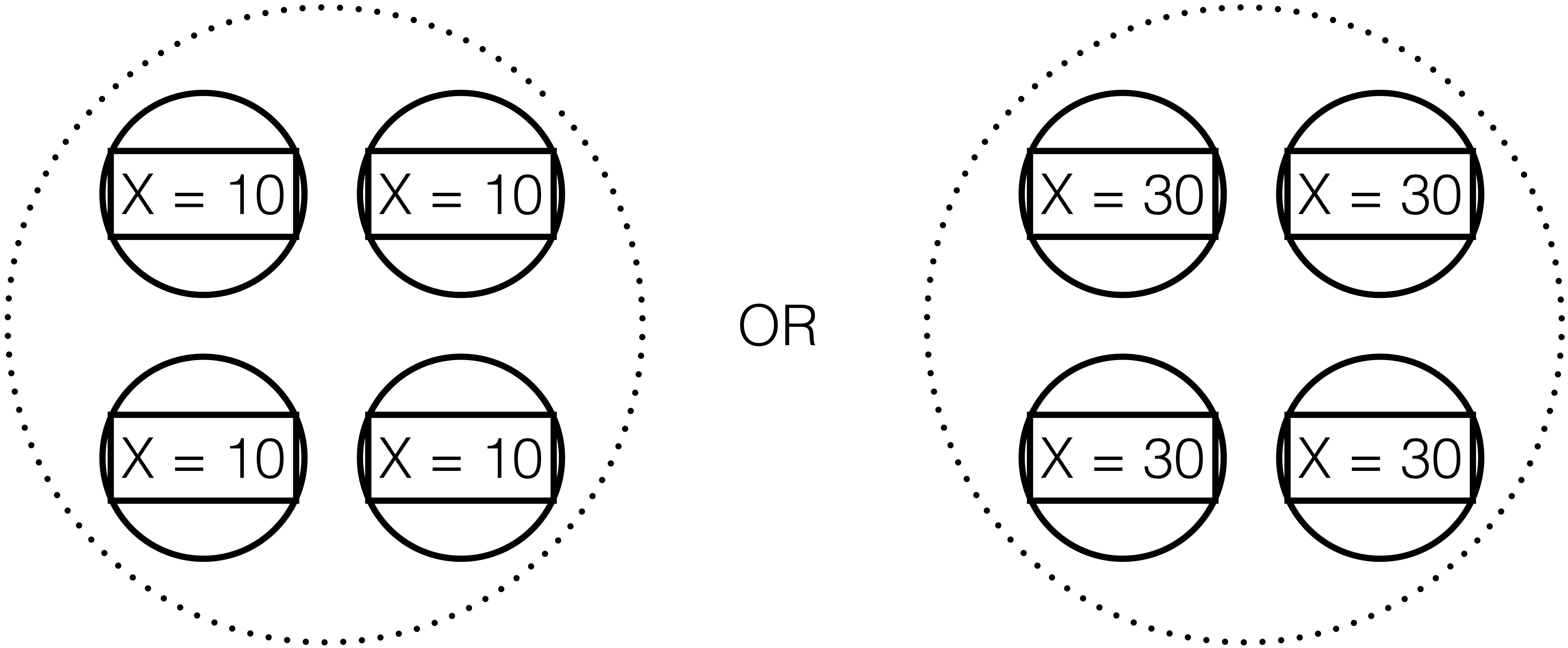


- Replicas need to ***agree*** on the which requests have been handled

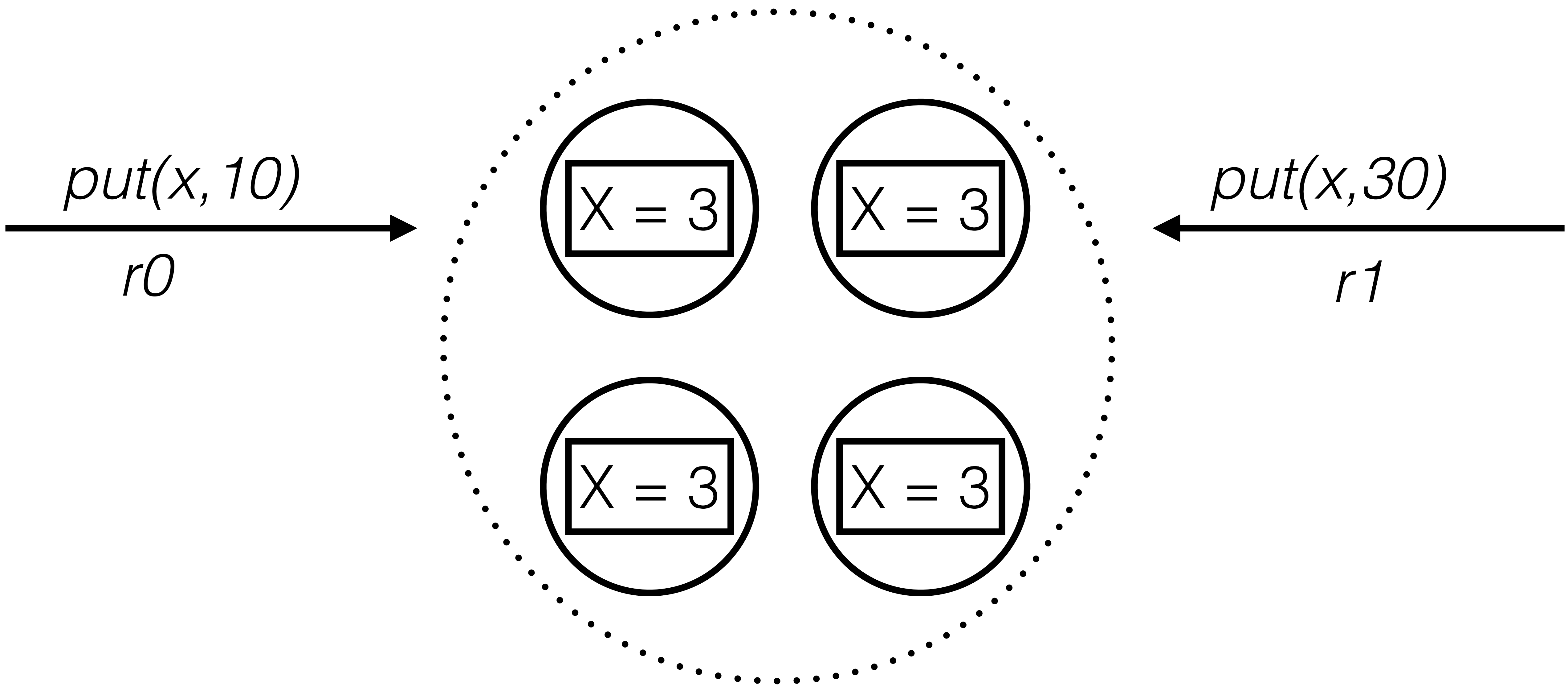
SMR Requirements



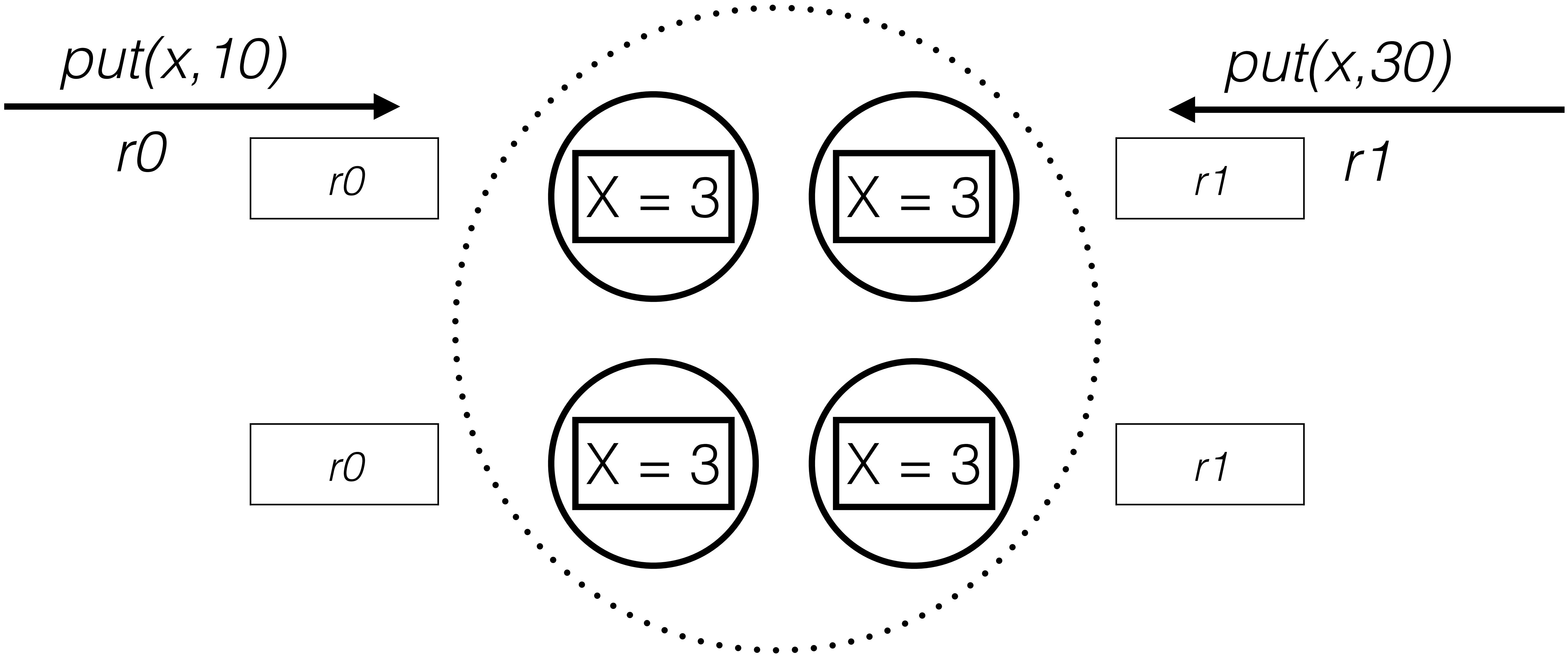
SMR
Requirements



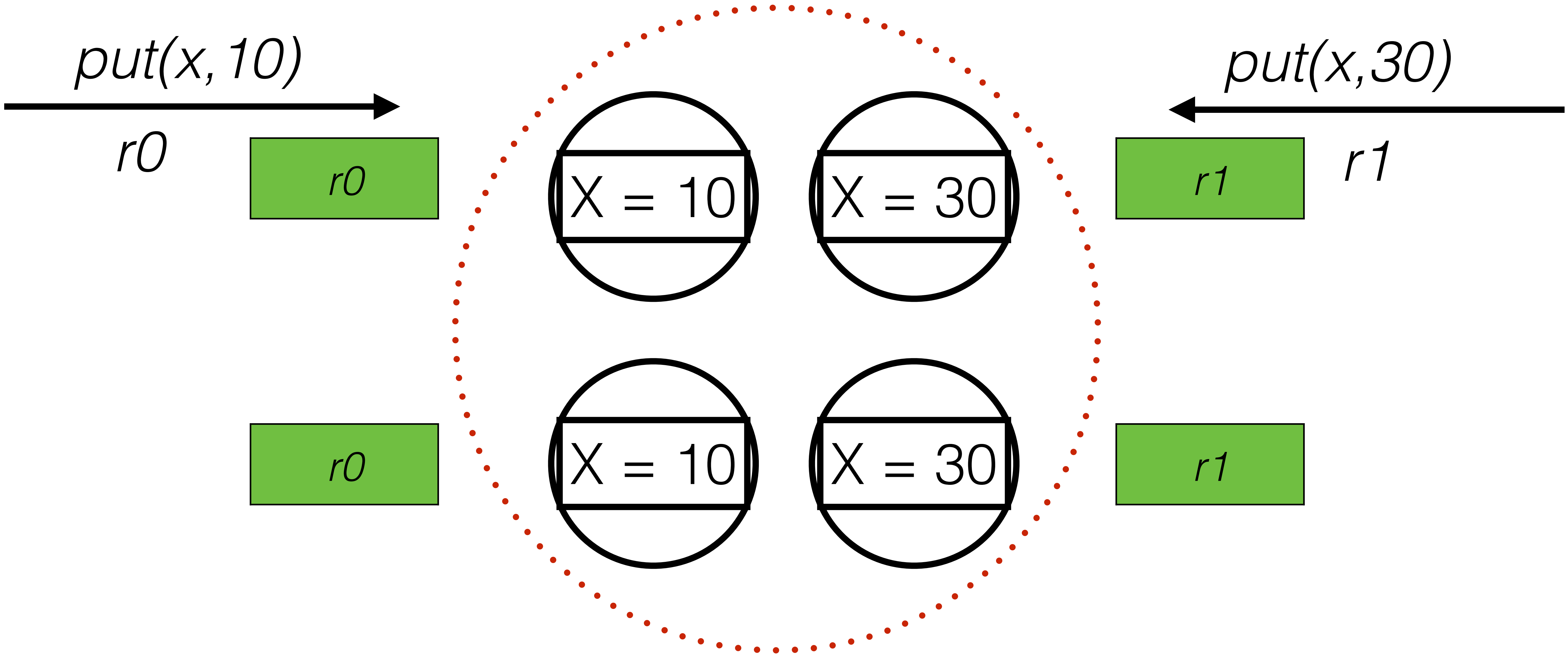
SMR Requirements



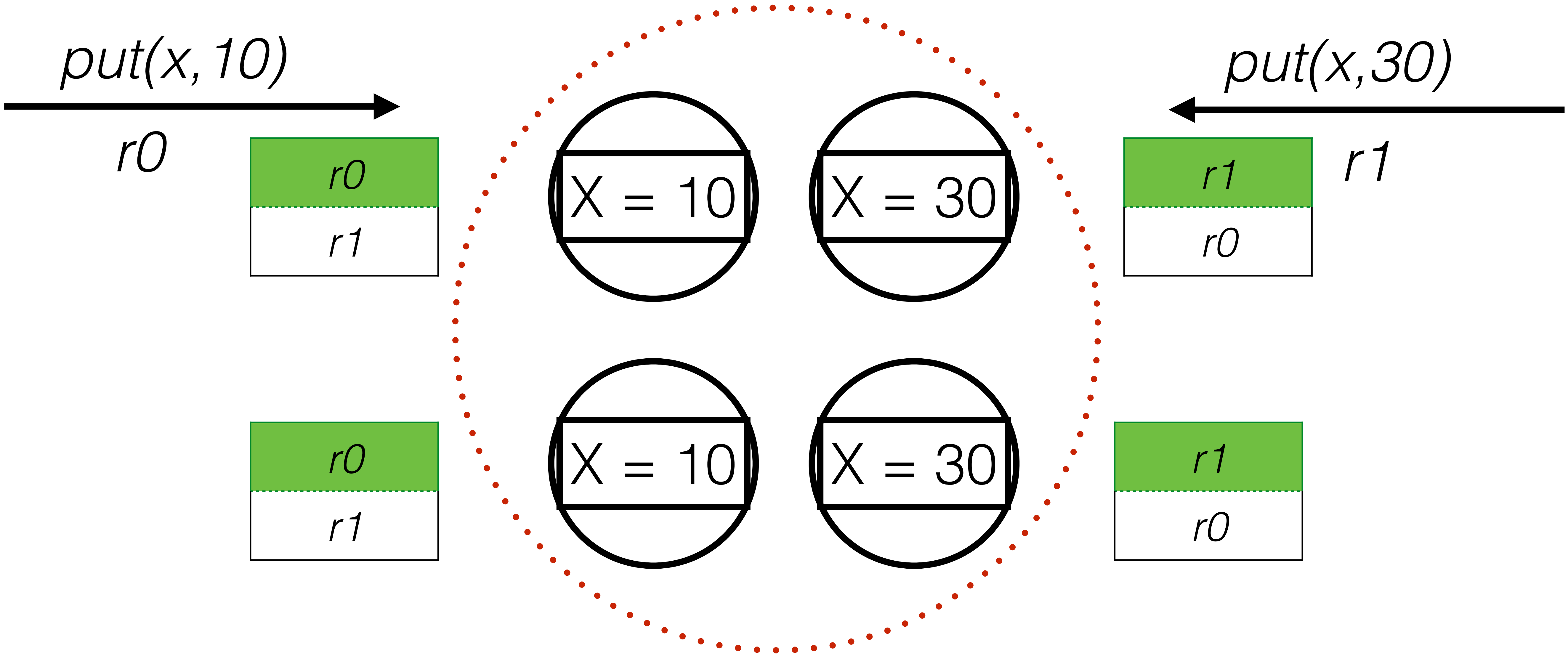
SMR Requirements



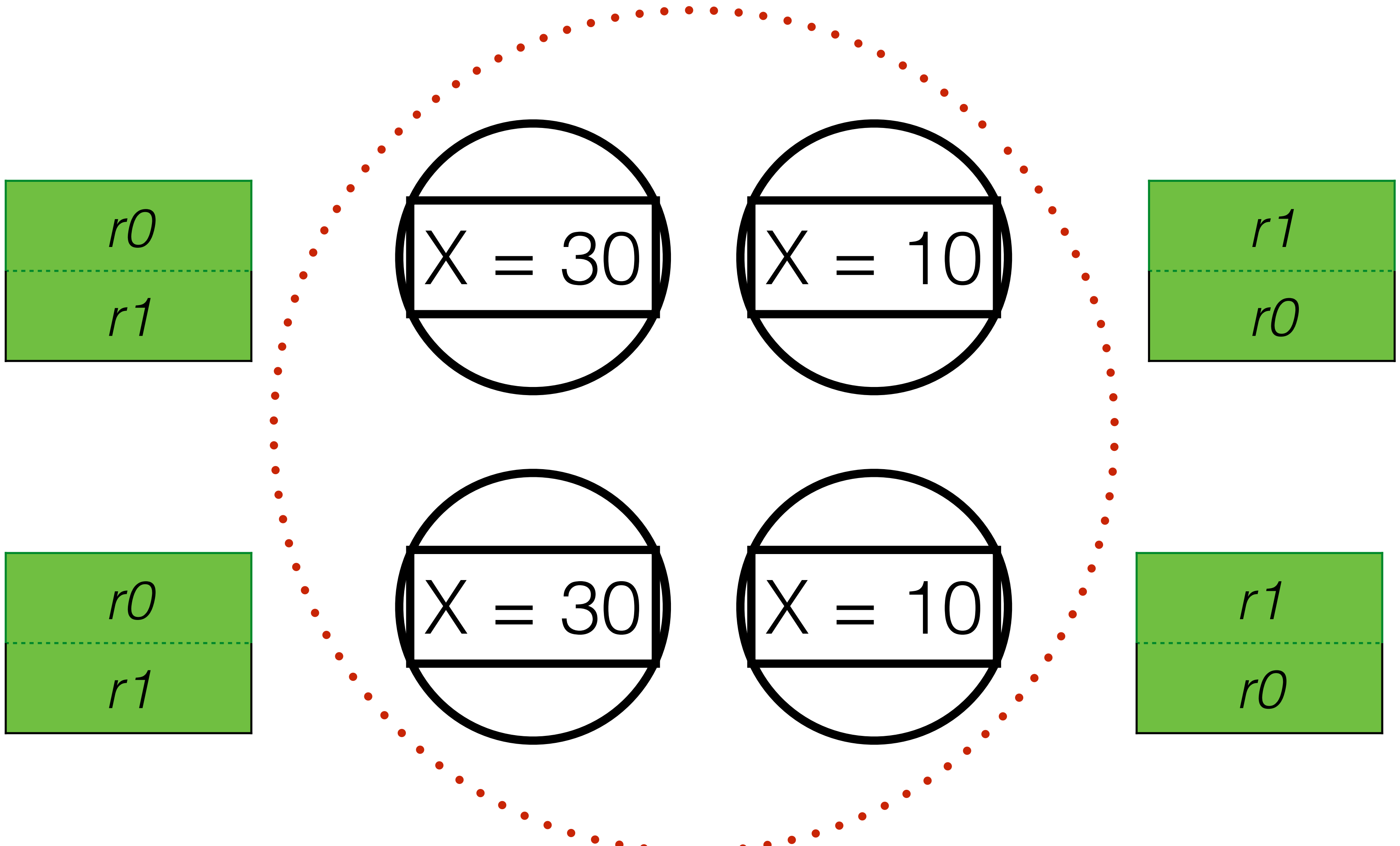
SMR Requirements



SMR Requirements



SMR Requirements



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

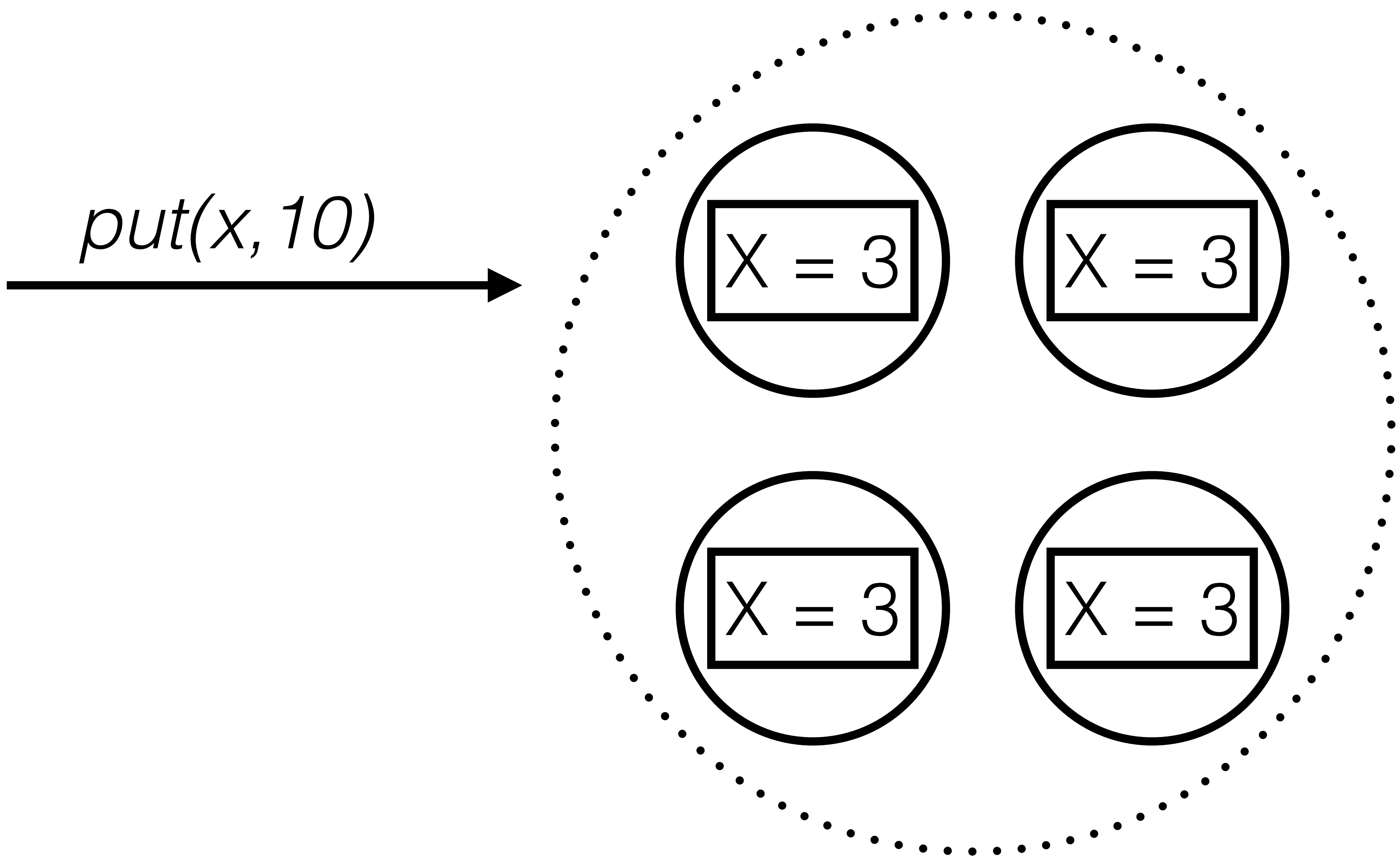
SMR

- 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

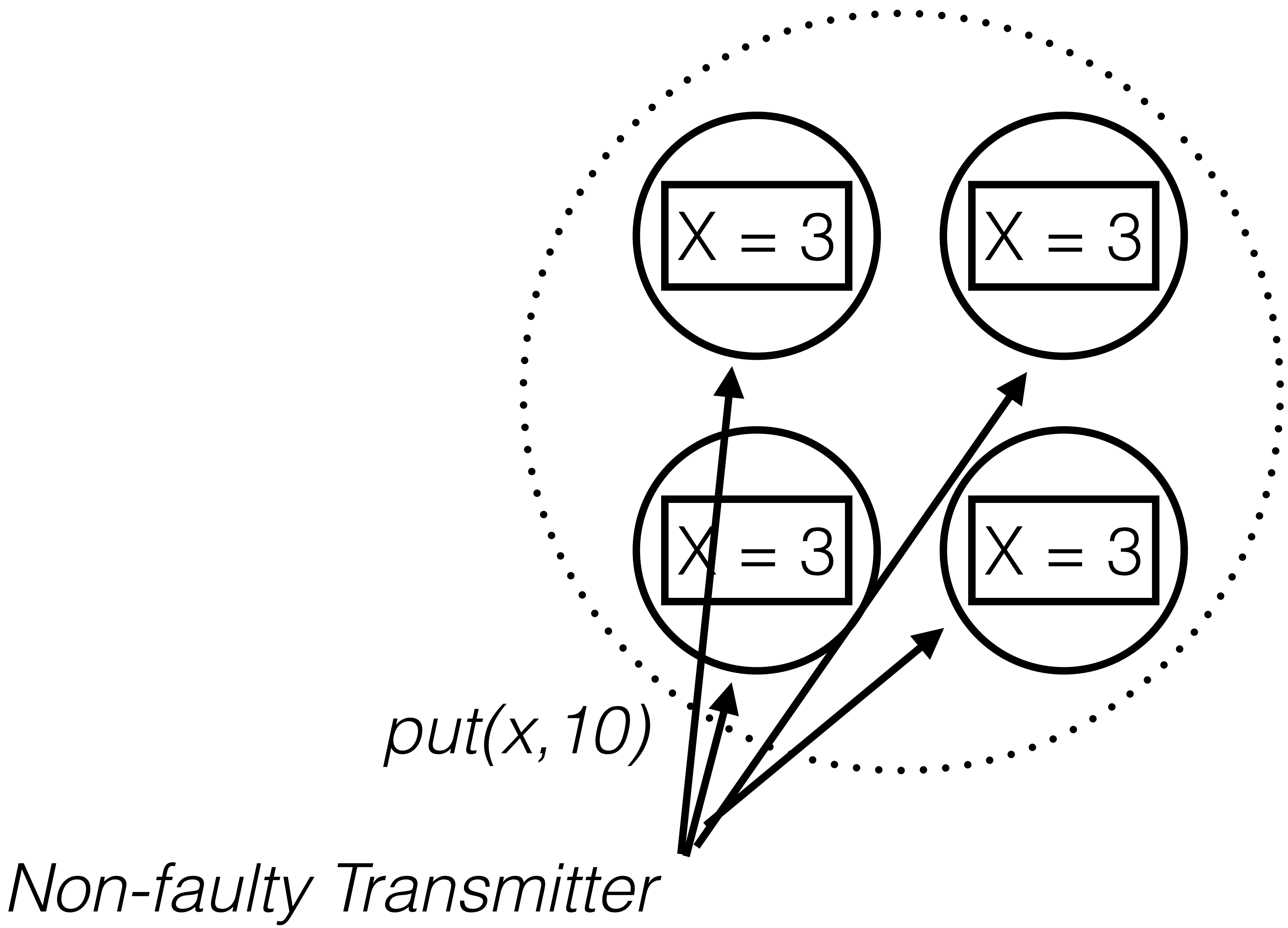
Implementation

- 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



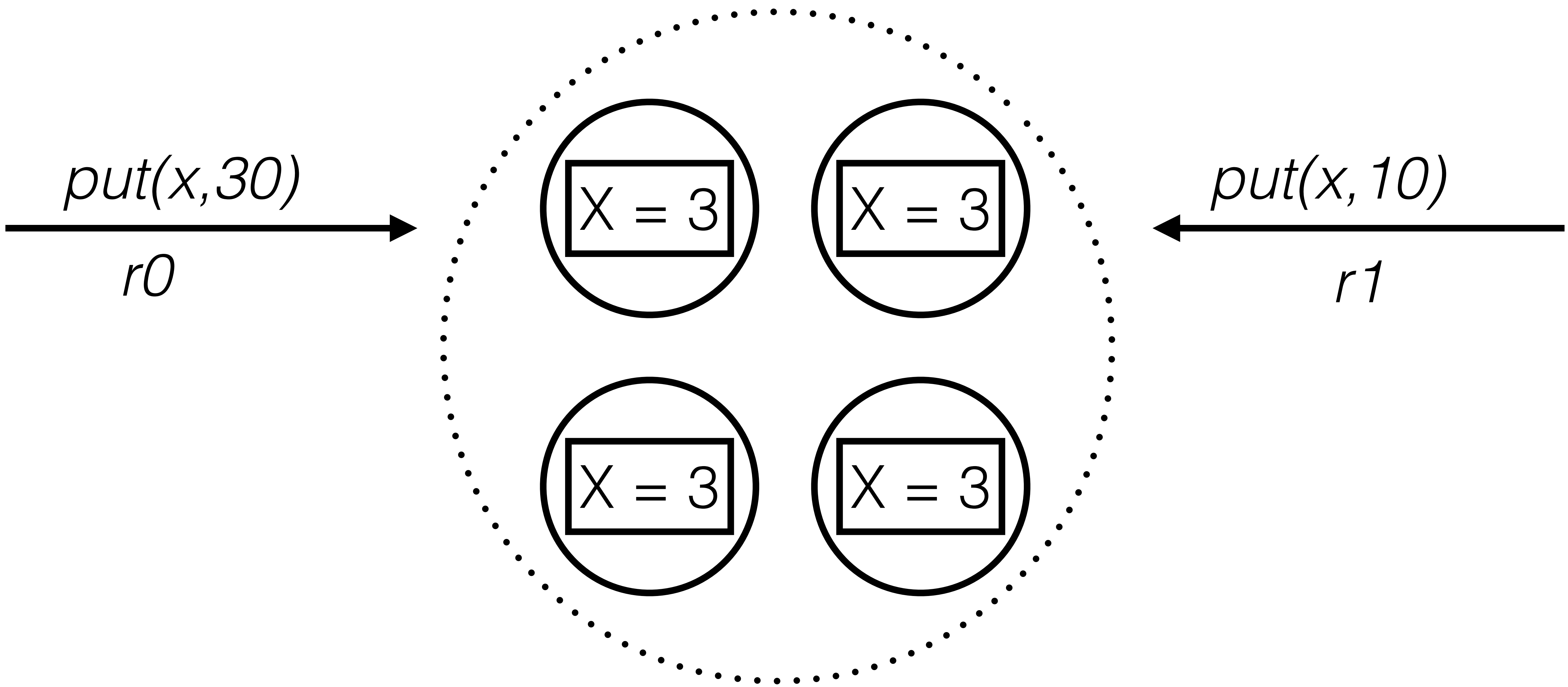
SMR Implementation



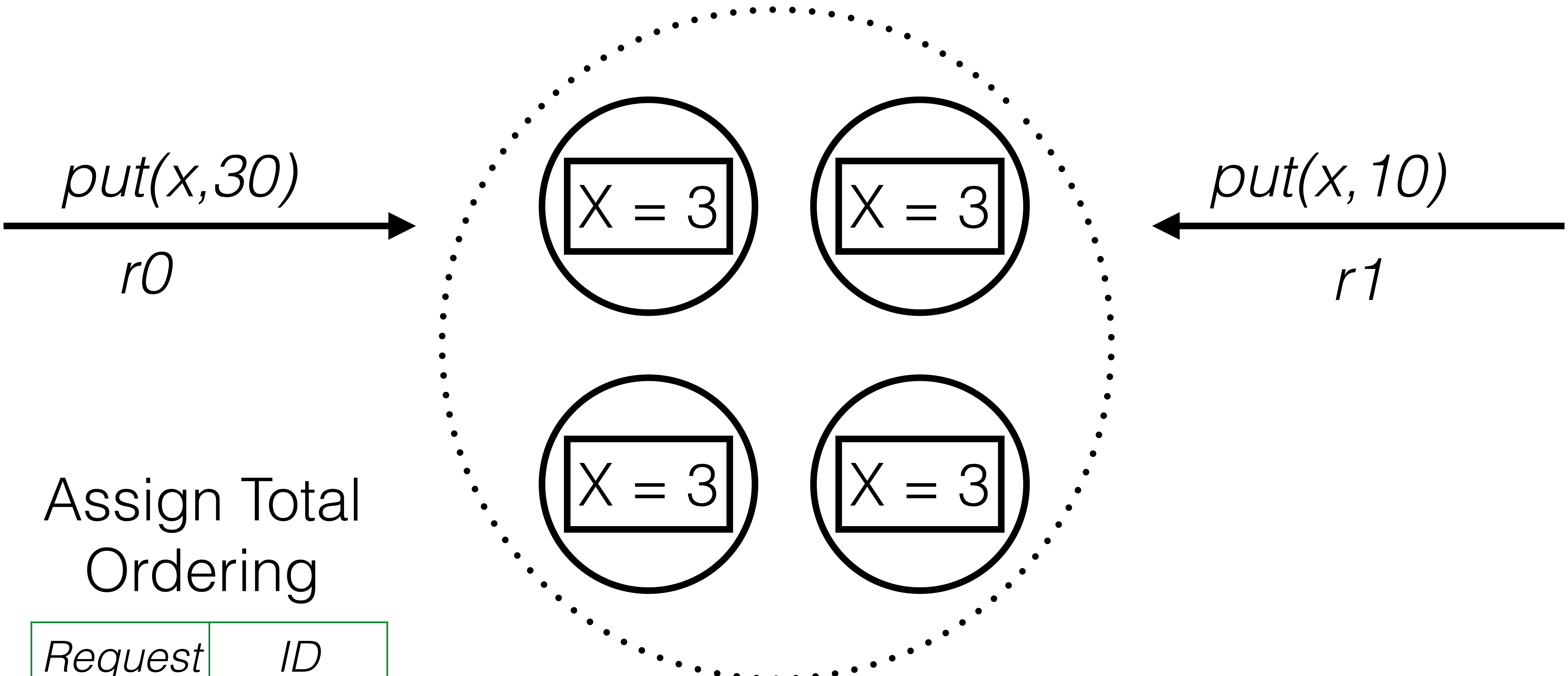
Implementation

- 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



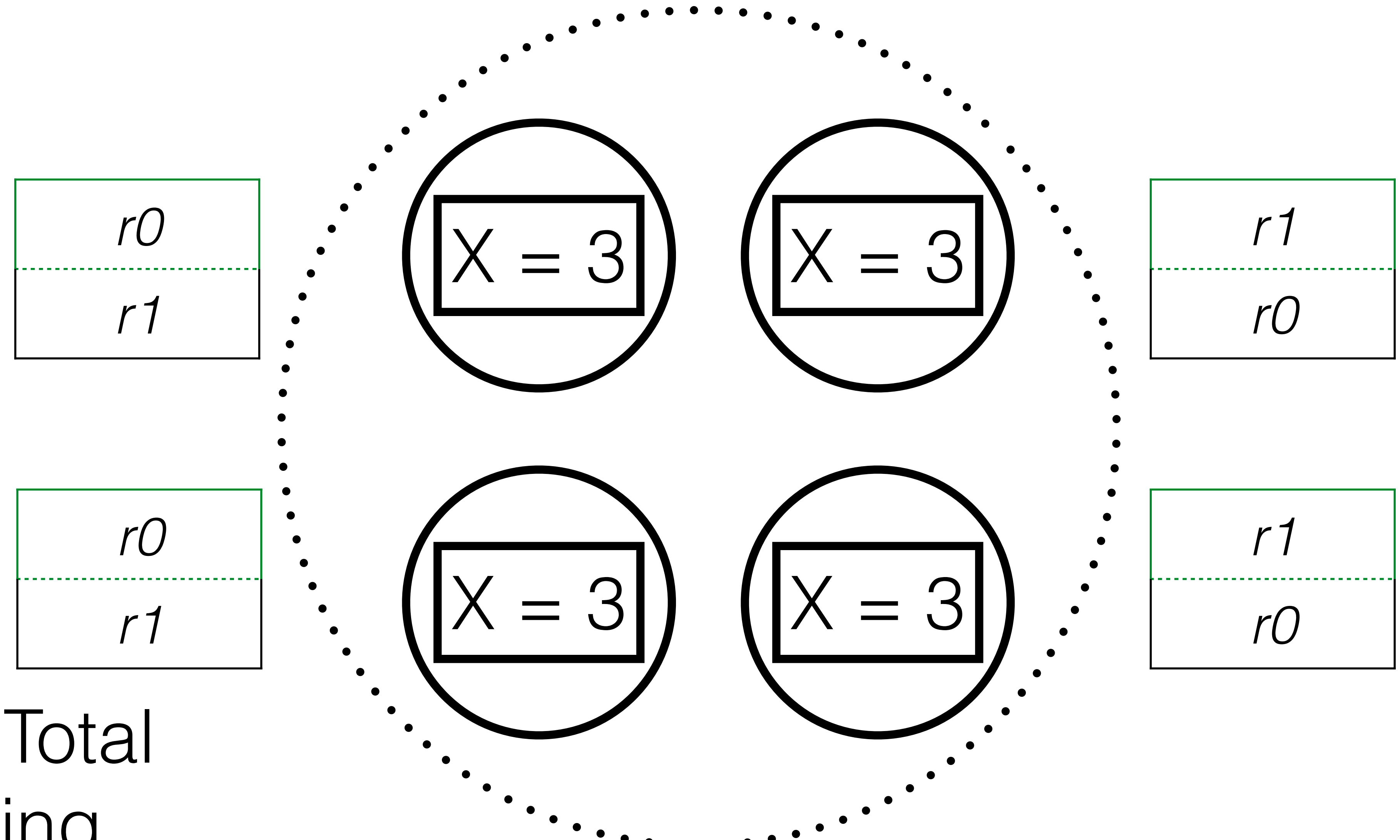
SMR Requirements



Assign Total Ordering

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

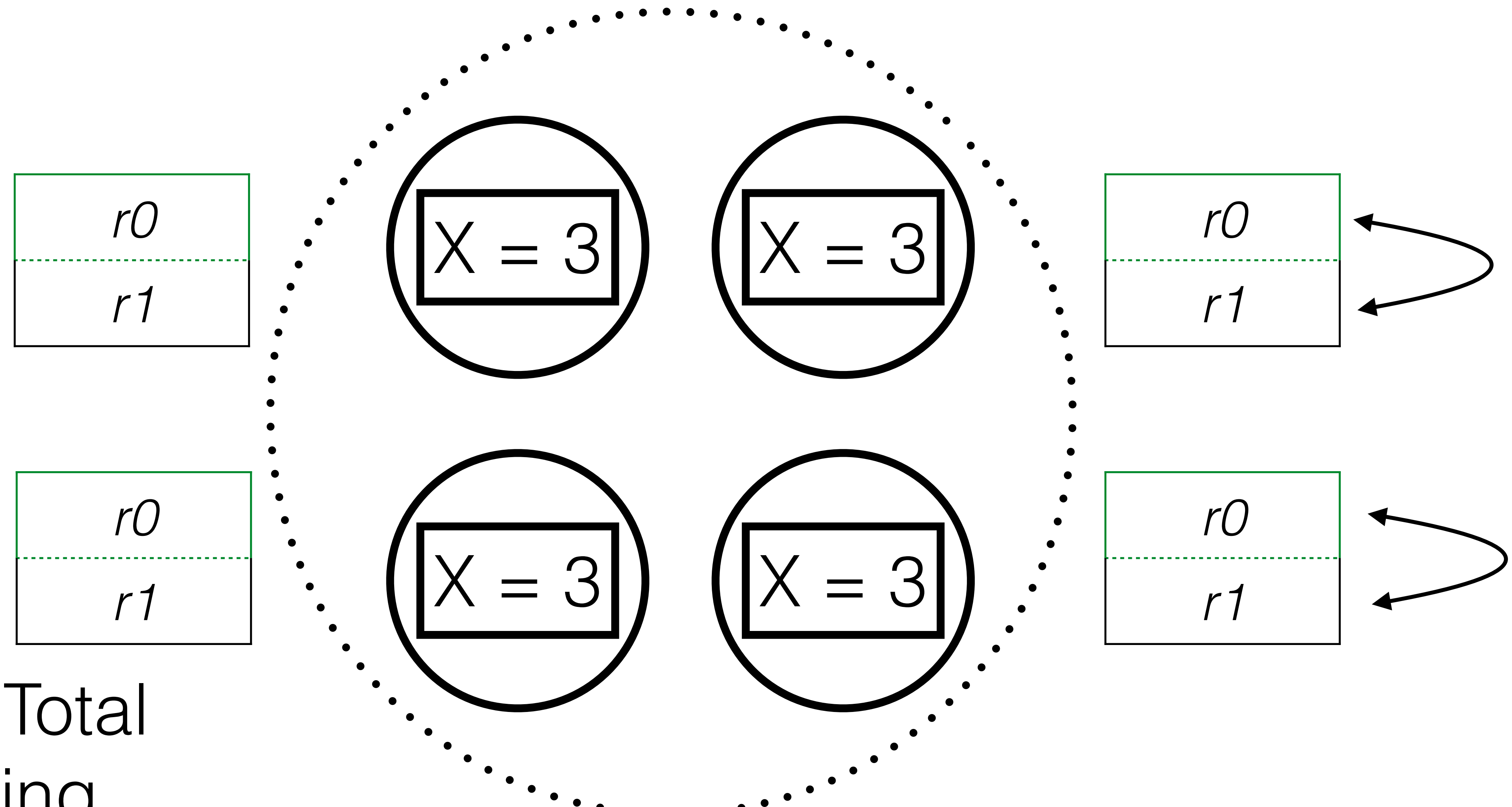
SMR Requirements



Assign Total Ordering

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

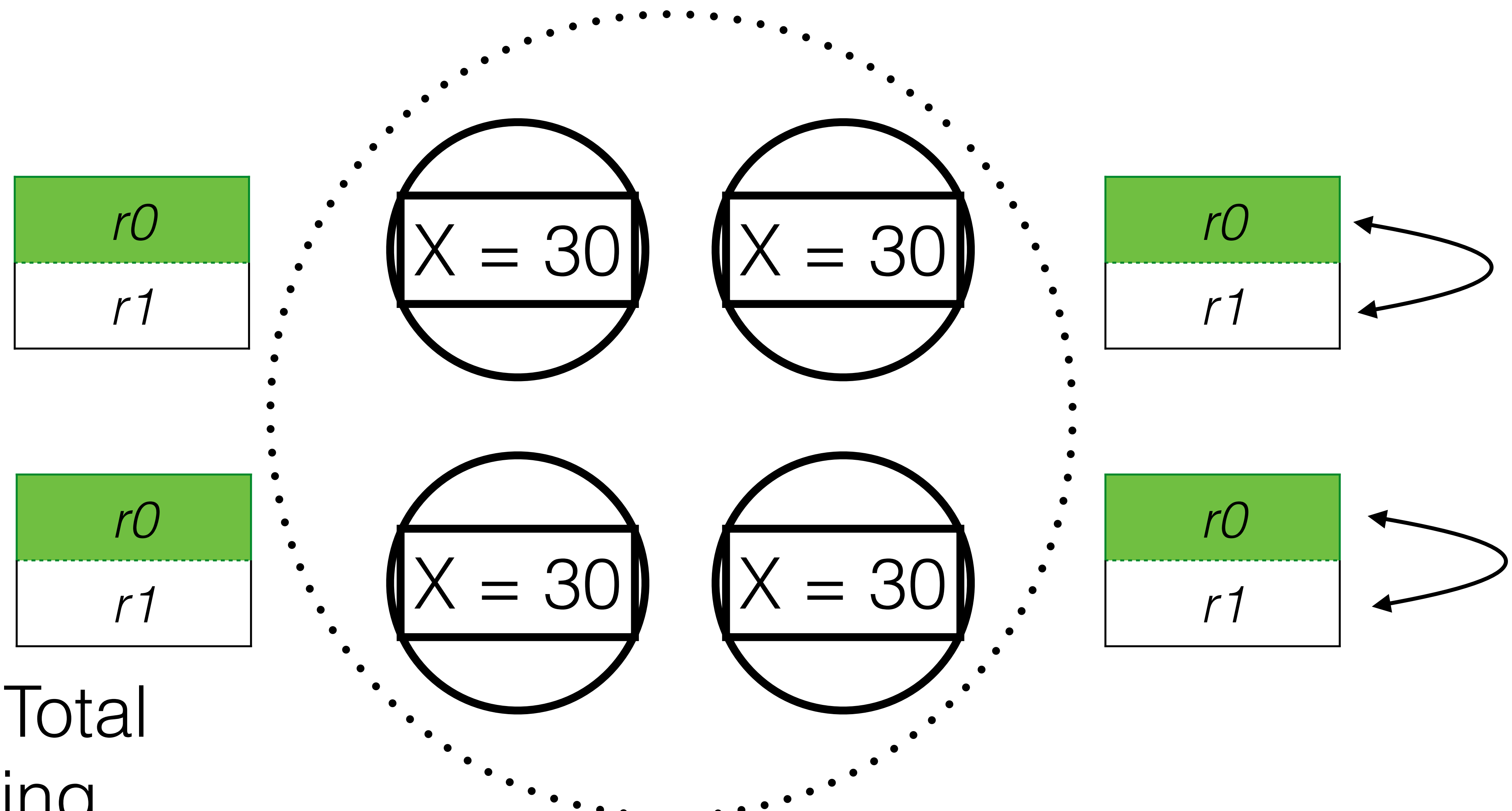
SMR Requirements



Assign Total Ordering

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

SMR Requirements

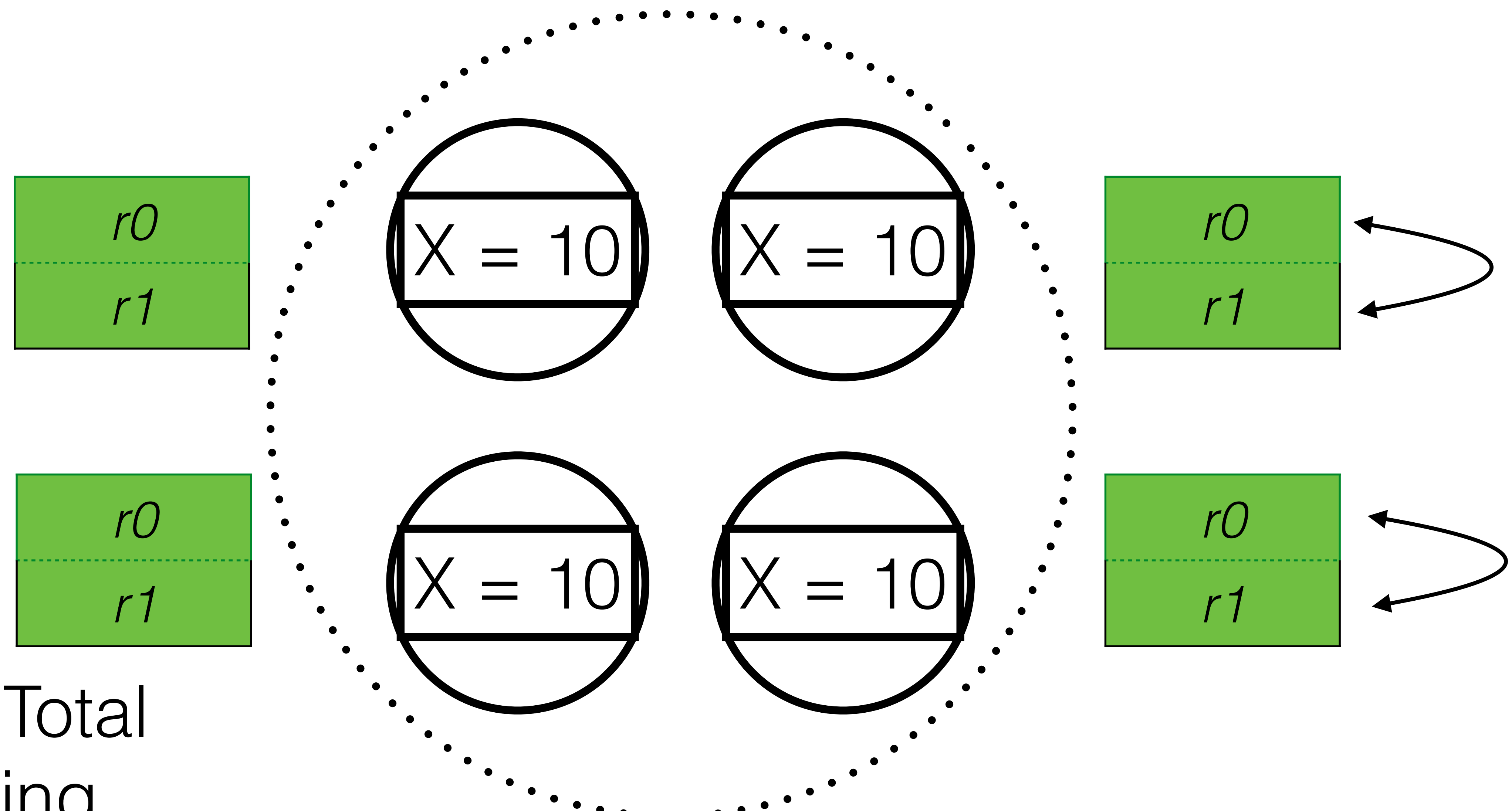


Assign Total Ordering

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!

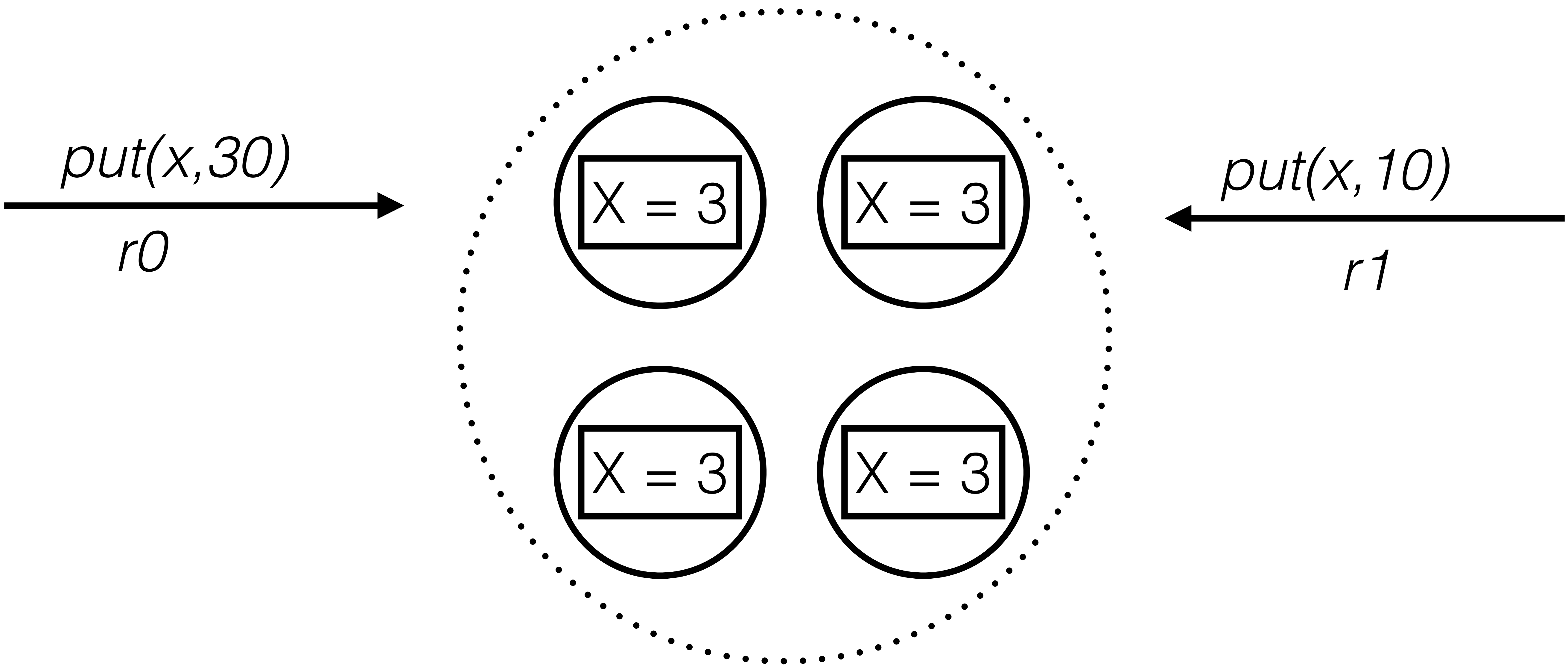
Implementation
Client Generated IDs

- Order via Clocks (Client timestamps represent IDs)
  - Logical Clocks
  - Synchronized Clocks
- Ideas from last week! [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

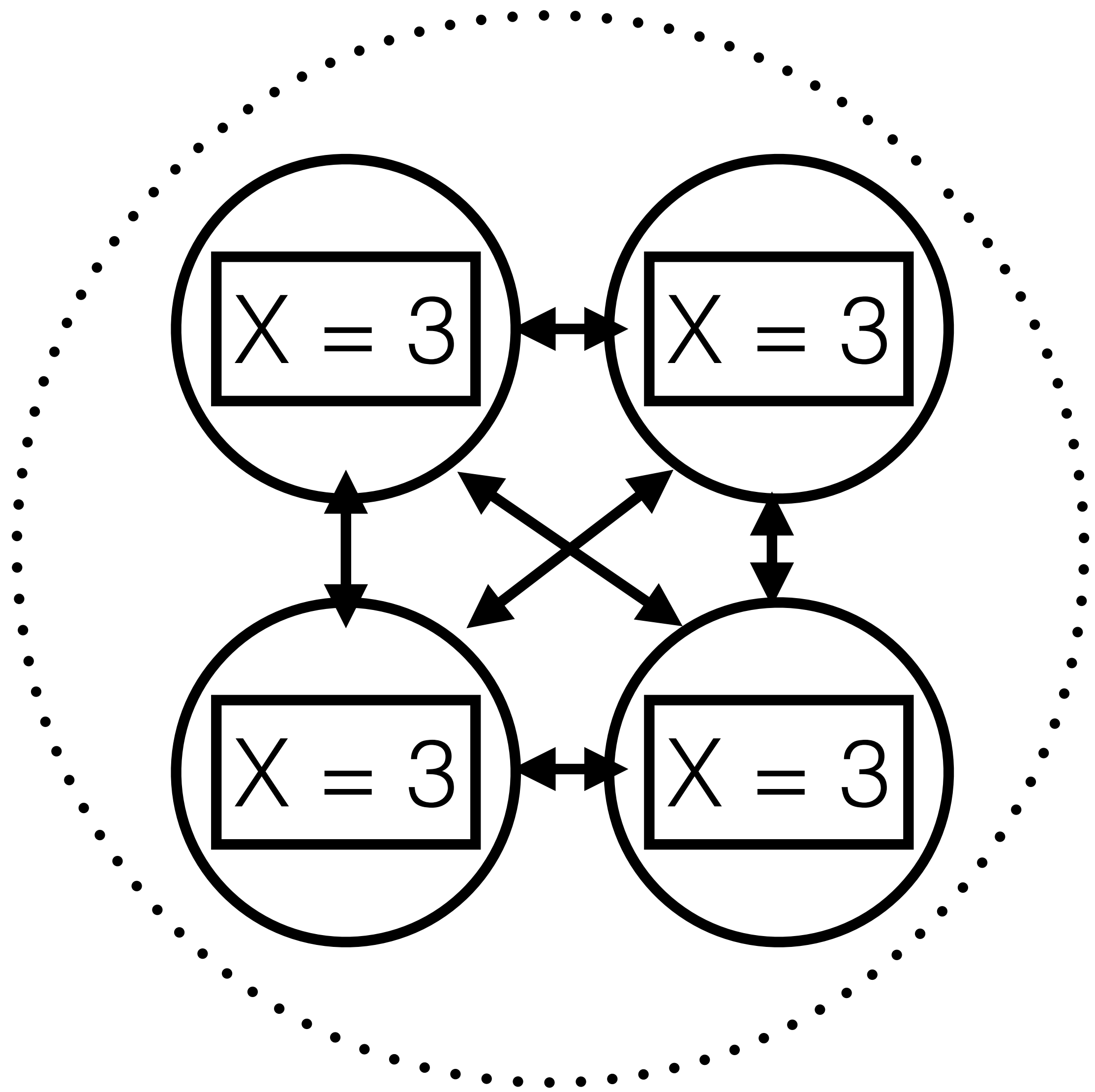
Replica ID Generation



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	

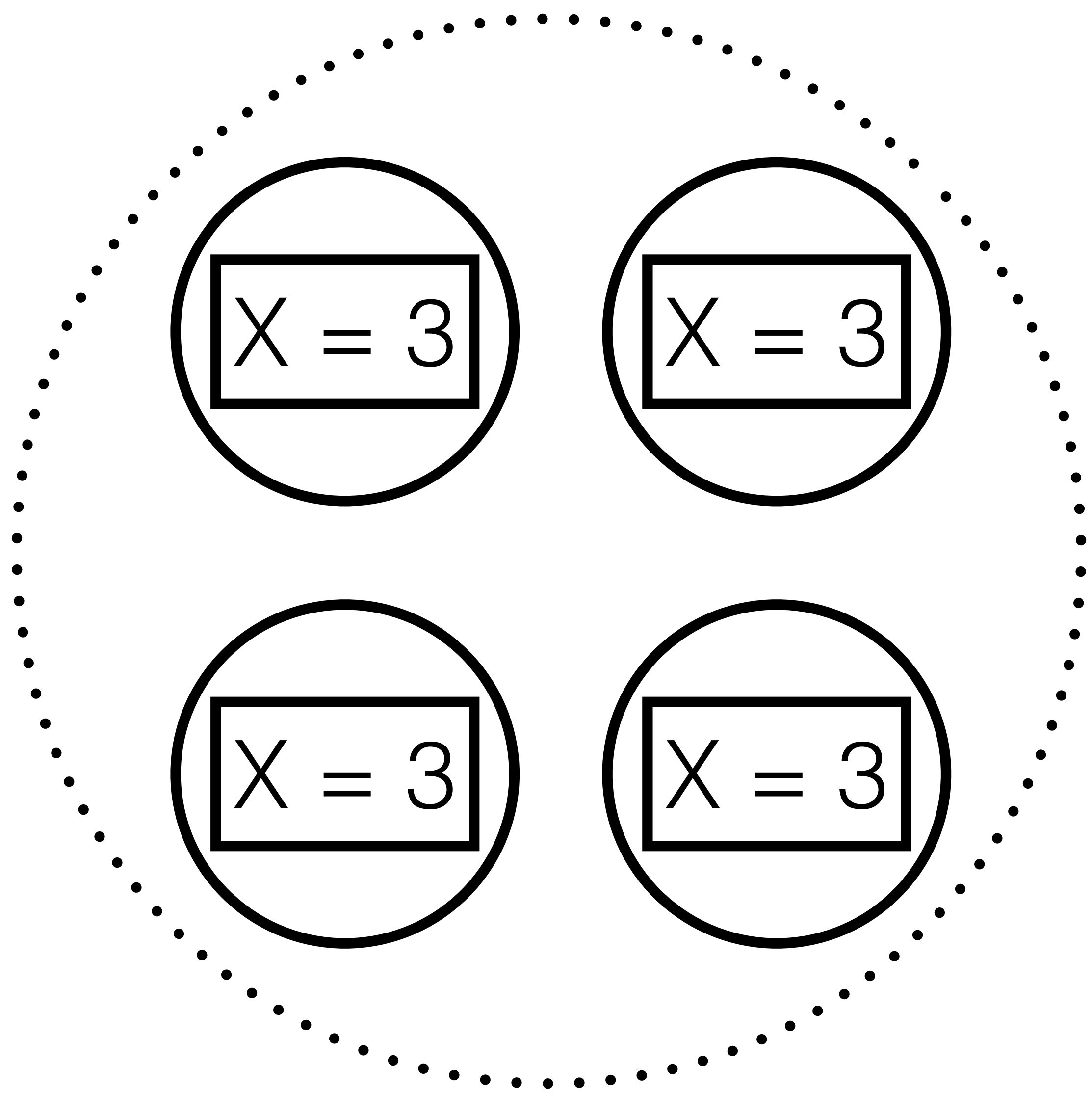
Req.	CUID	UID
r1	1.4	
r0	2.4	

1) Propose Candidates

Replica ID Generation

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

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



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

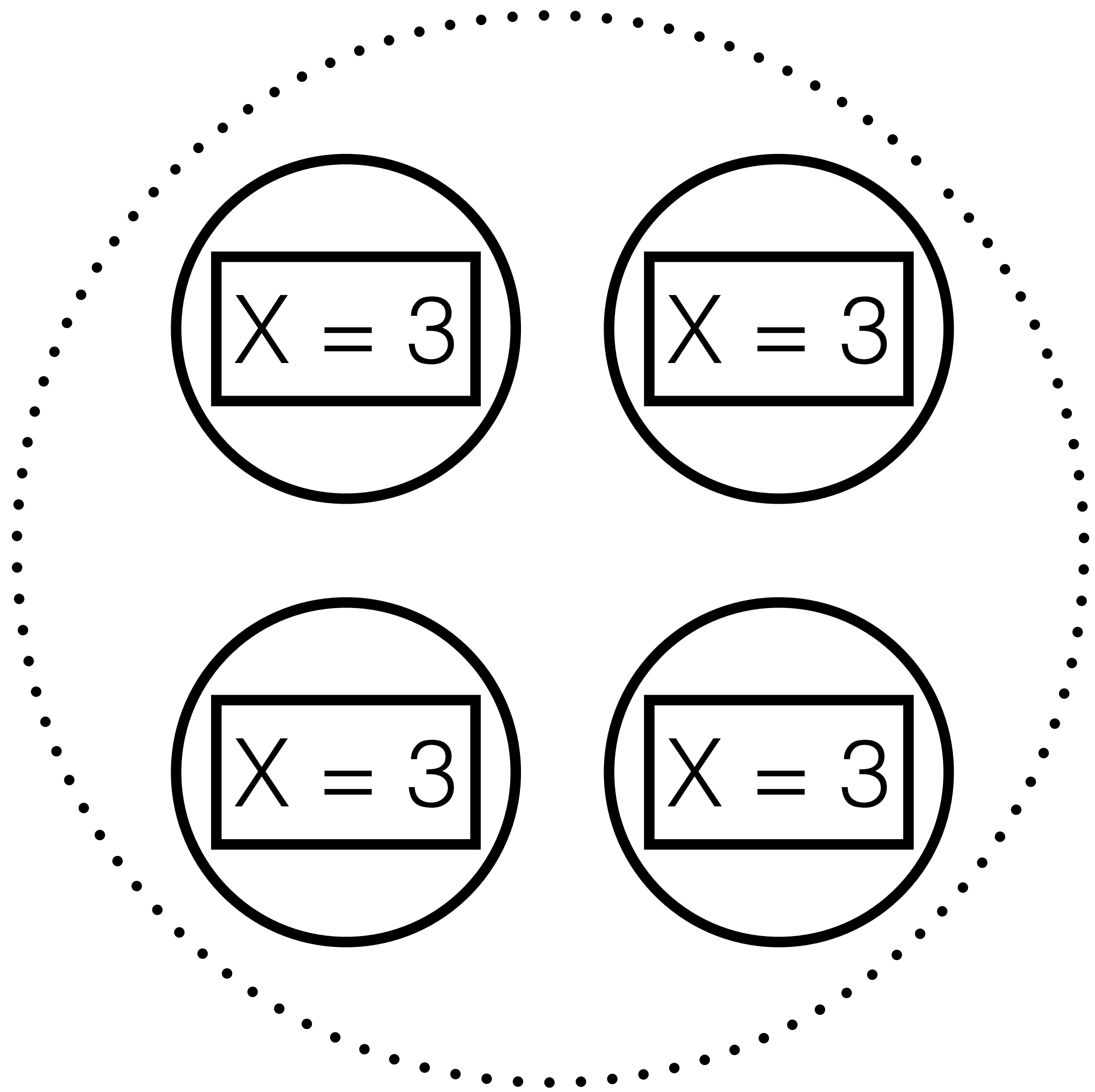
Req.	*CUID*	*UID*
r1	*1.4*	
r0	*2.4*	*2.4*

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*

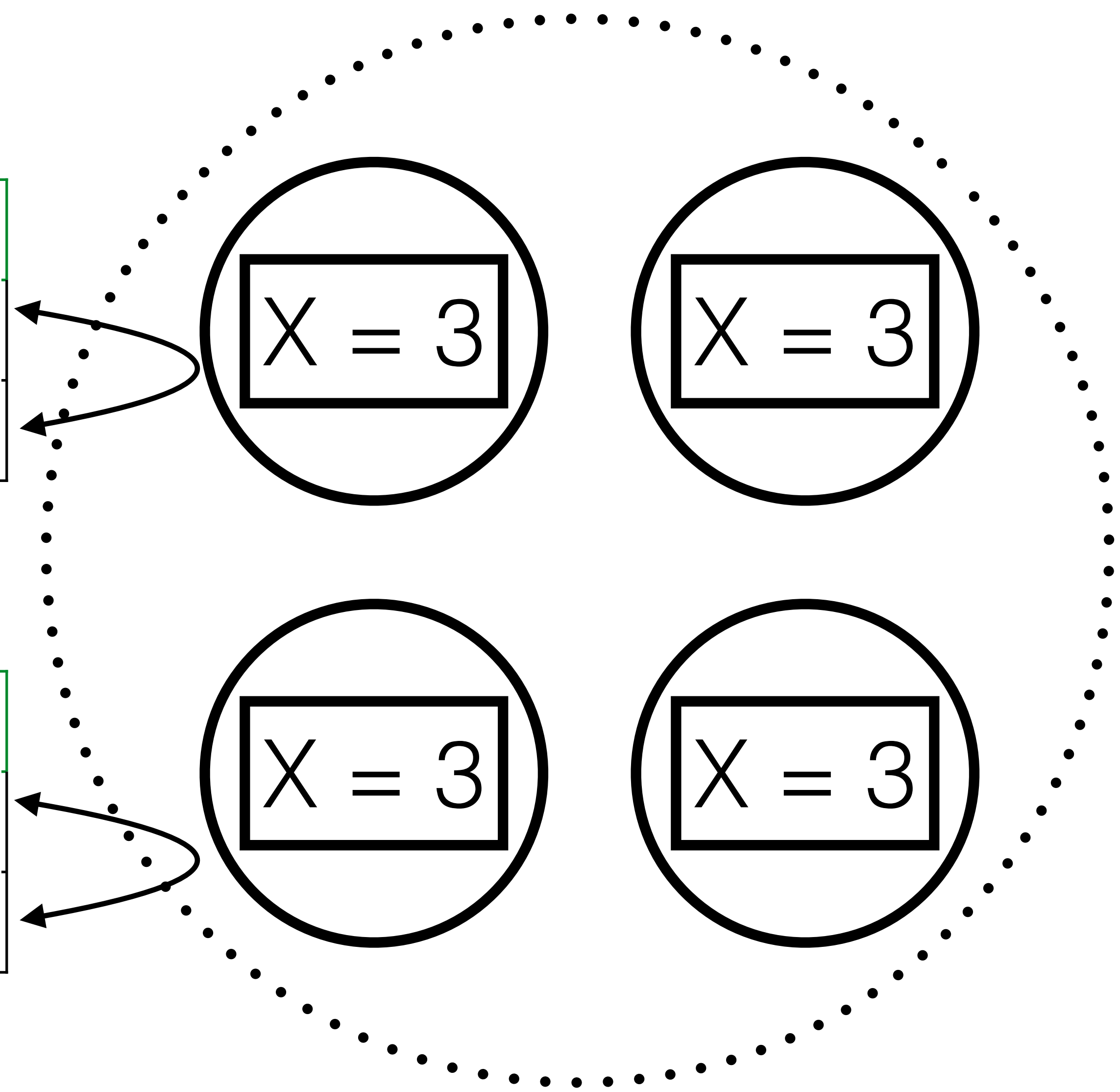
Req.	*CUID*	*UID*
r1	*1.4*	*2.2*
r0	*2.4*	*2.4*

3) Accept *r1*

Replica ID Generation

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



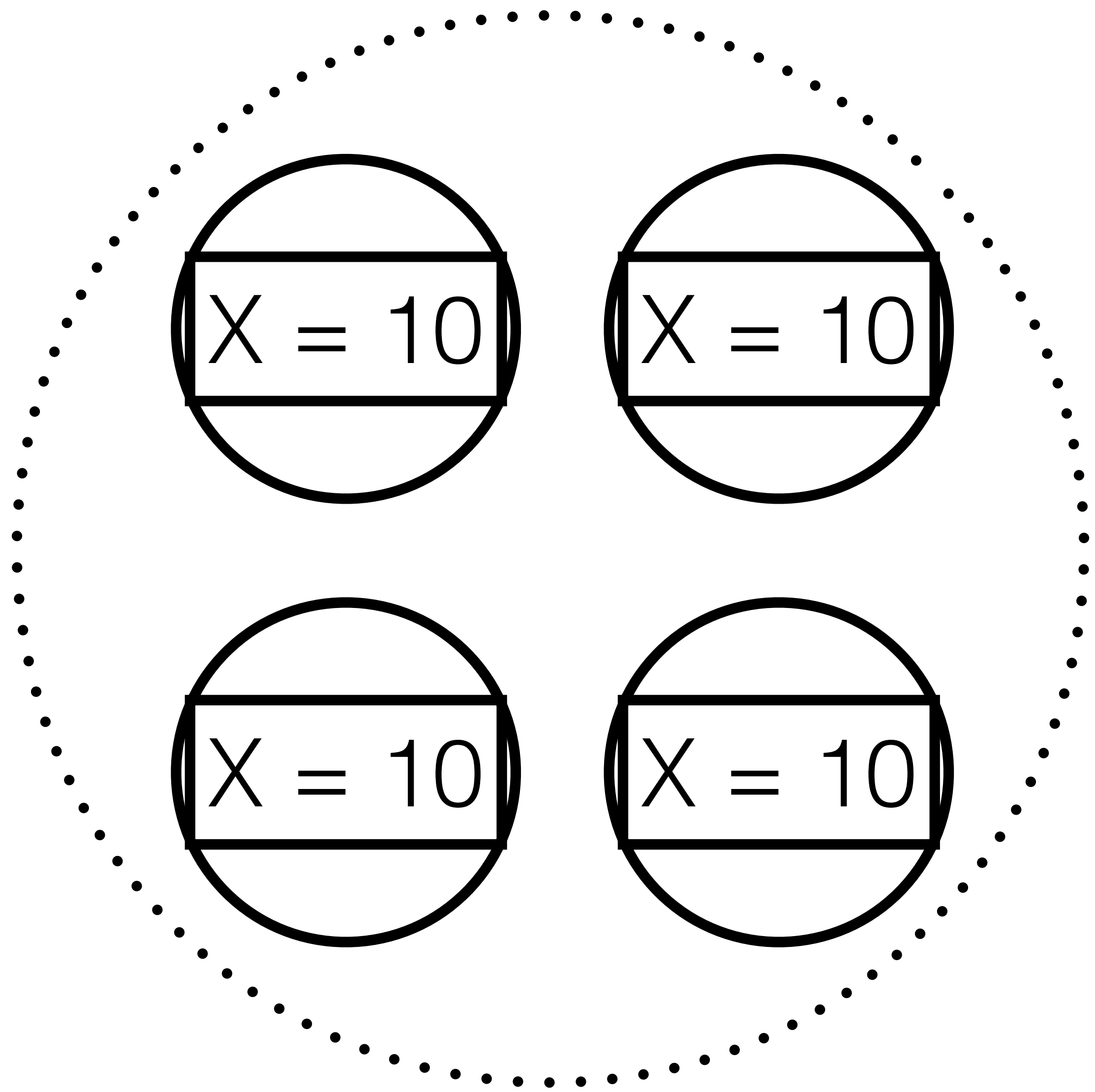
Req.	CUID	UID
r1	1.3	2.2
r0	2.3	2.4

Req.	CUID	UID
r1	1.4	2.2
r0	2.4	2.4

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*

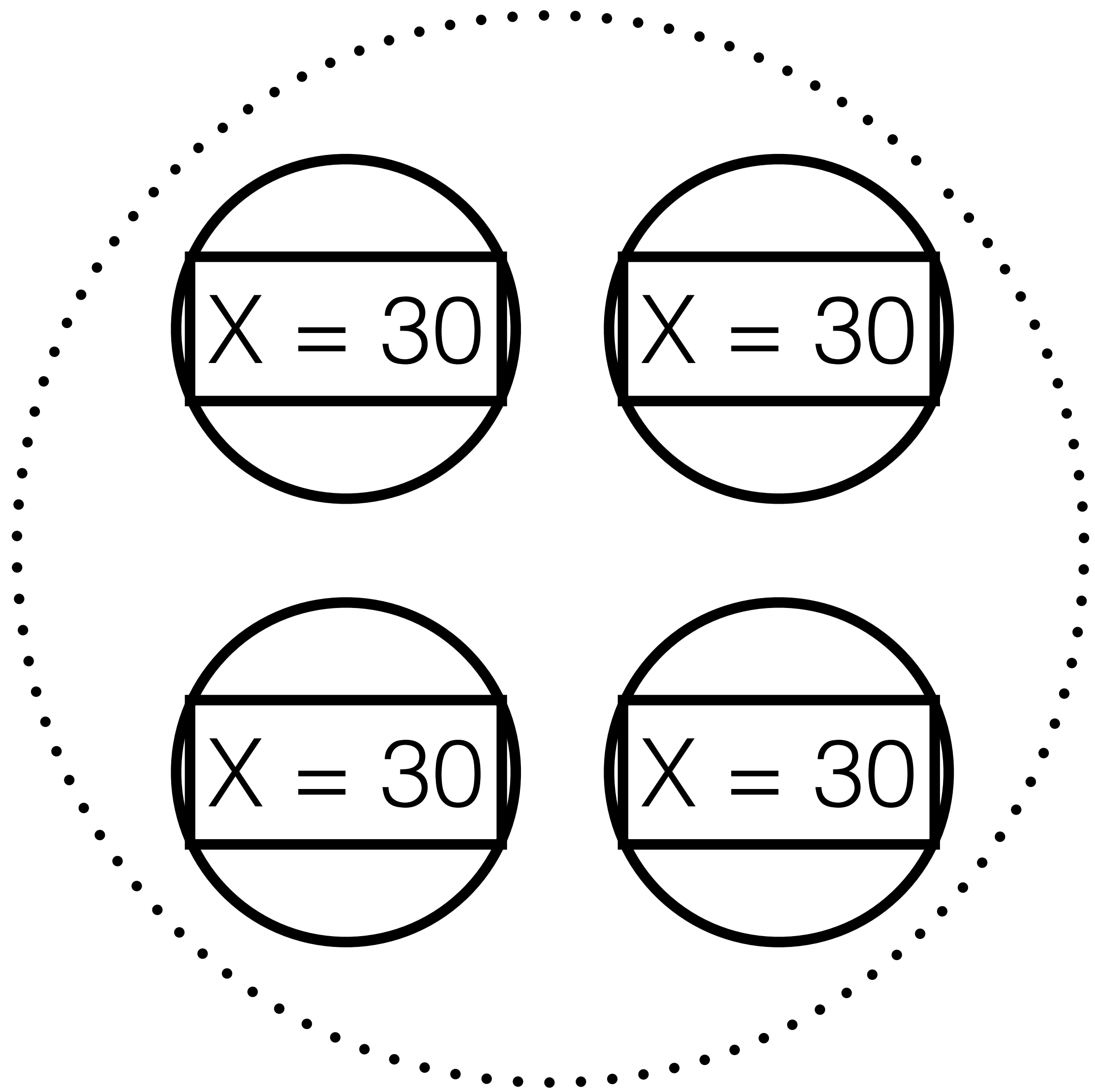
Req.	*CUID*	*UID*
r1	*1.4*	*2.2*
r0	*2.4*	*2.4*

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 $uid(r) < cuid(sm_i, r')$
  - $cuid(sm_i, r) <= uid(r)$

Implementation
Replica Generated IDs

- When do we know a candidate is *stable?*
  - A candidate is *accepted*
  - No other pending requests with smaller candidate ids

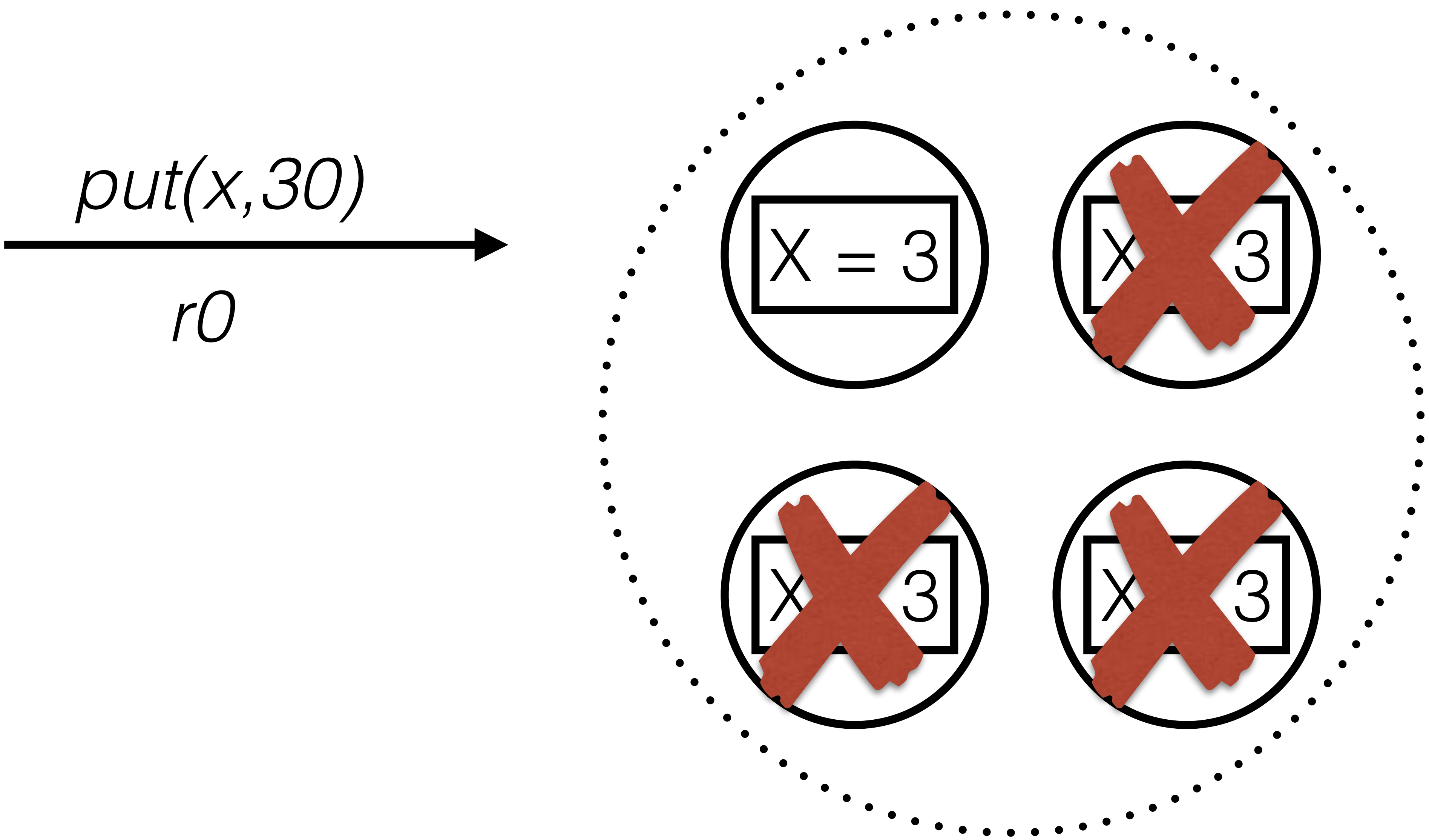
Fault Tolerance

- 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

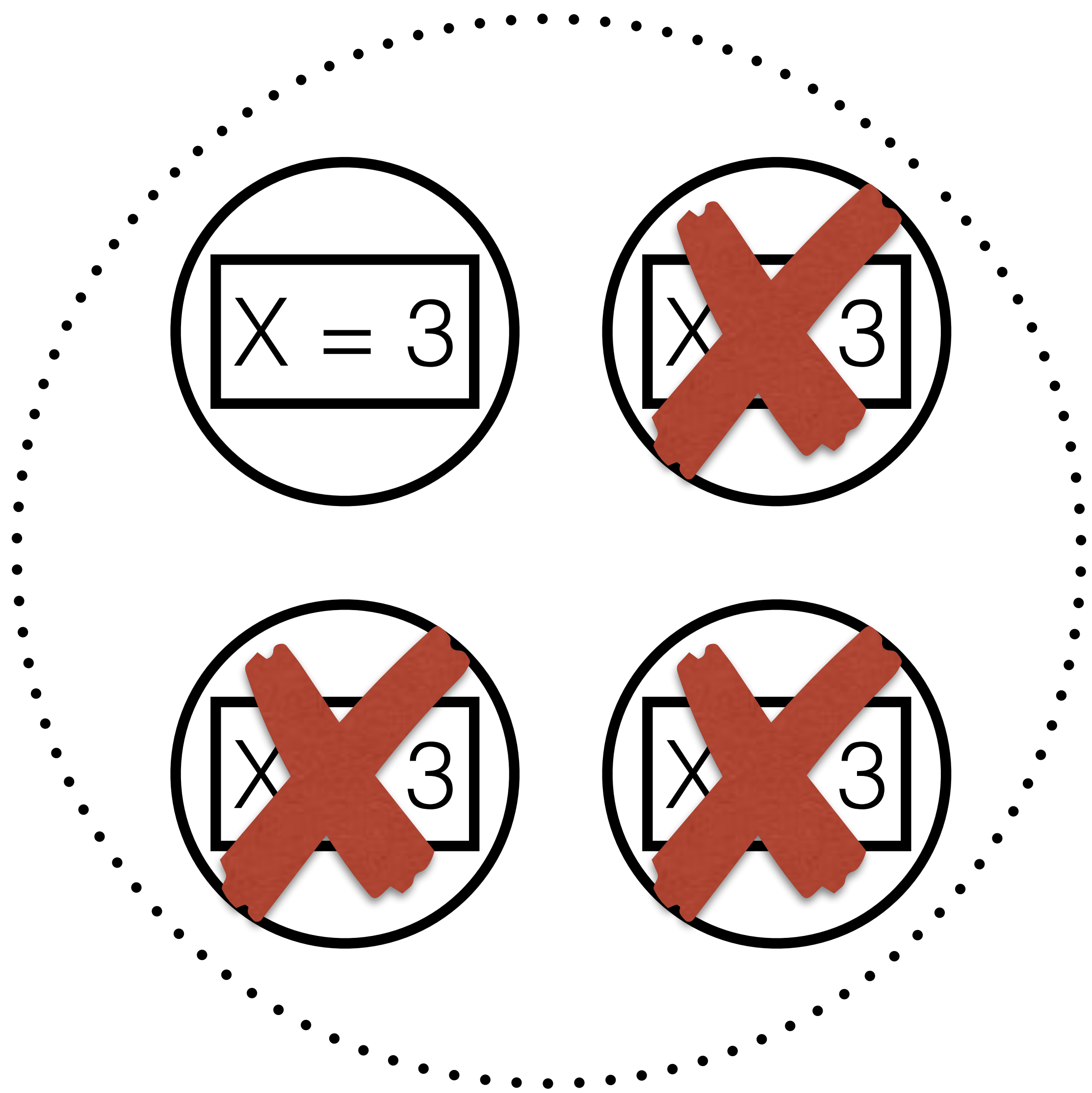
- Fail-Stop
  - A faulty server can be detected as faulty
- Byzantine
  - Faulty servers can do arbitrary, perhaps malicious things
- Crash Failures - NOT covered in paper
  - Server can stop responding without notification (subset of Byzantine)

Fail-Stop Tolerance



Fail-Stop Tolerance

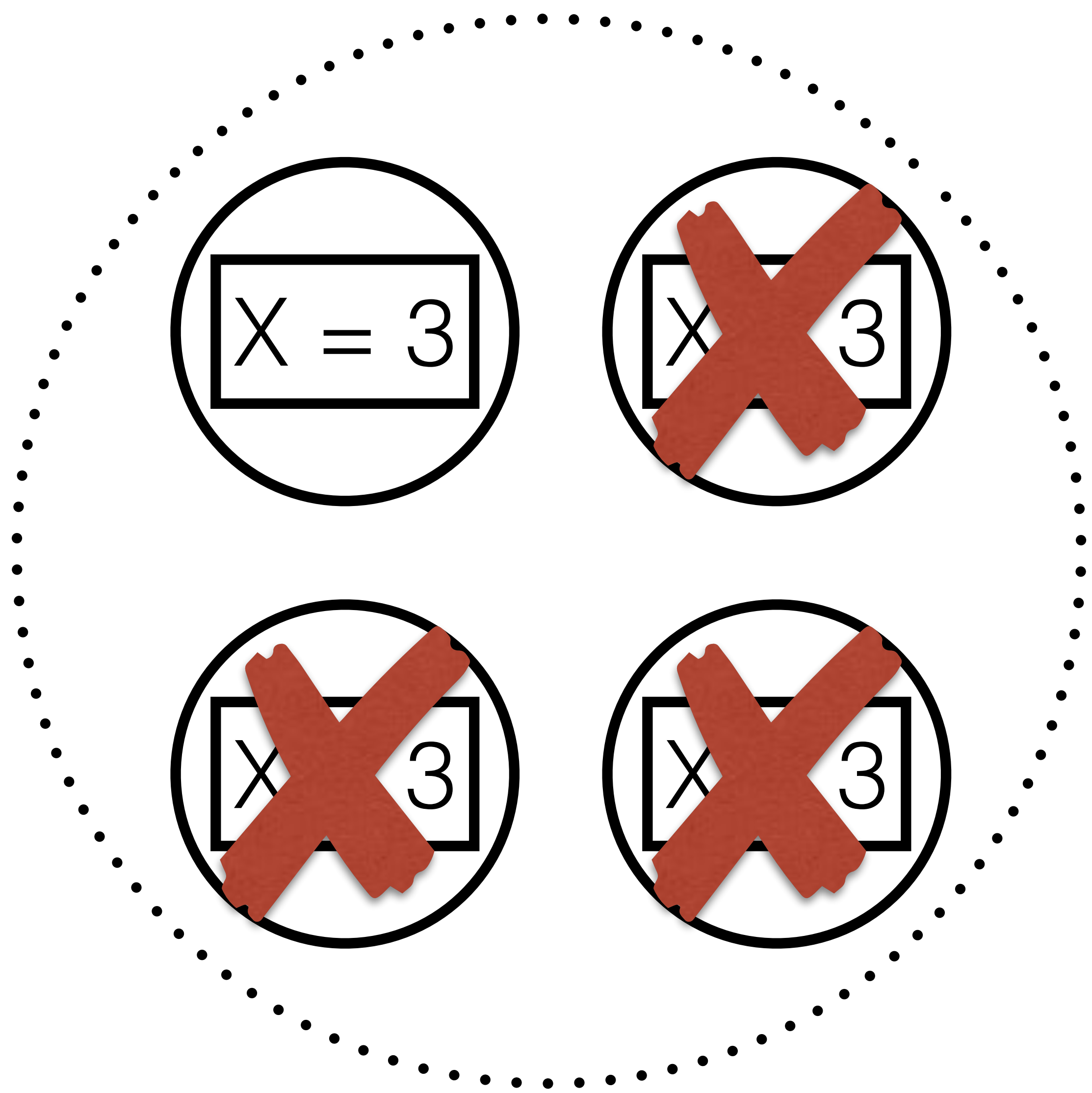
Req.	*CUID*	*UID*
r0	*1.1*	



1) Propose Candidates….

Fail-Stop Tolerance

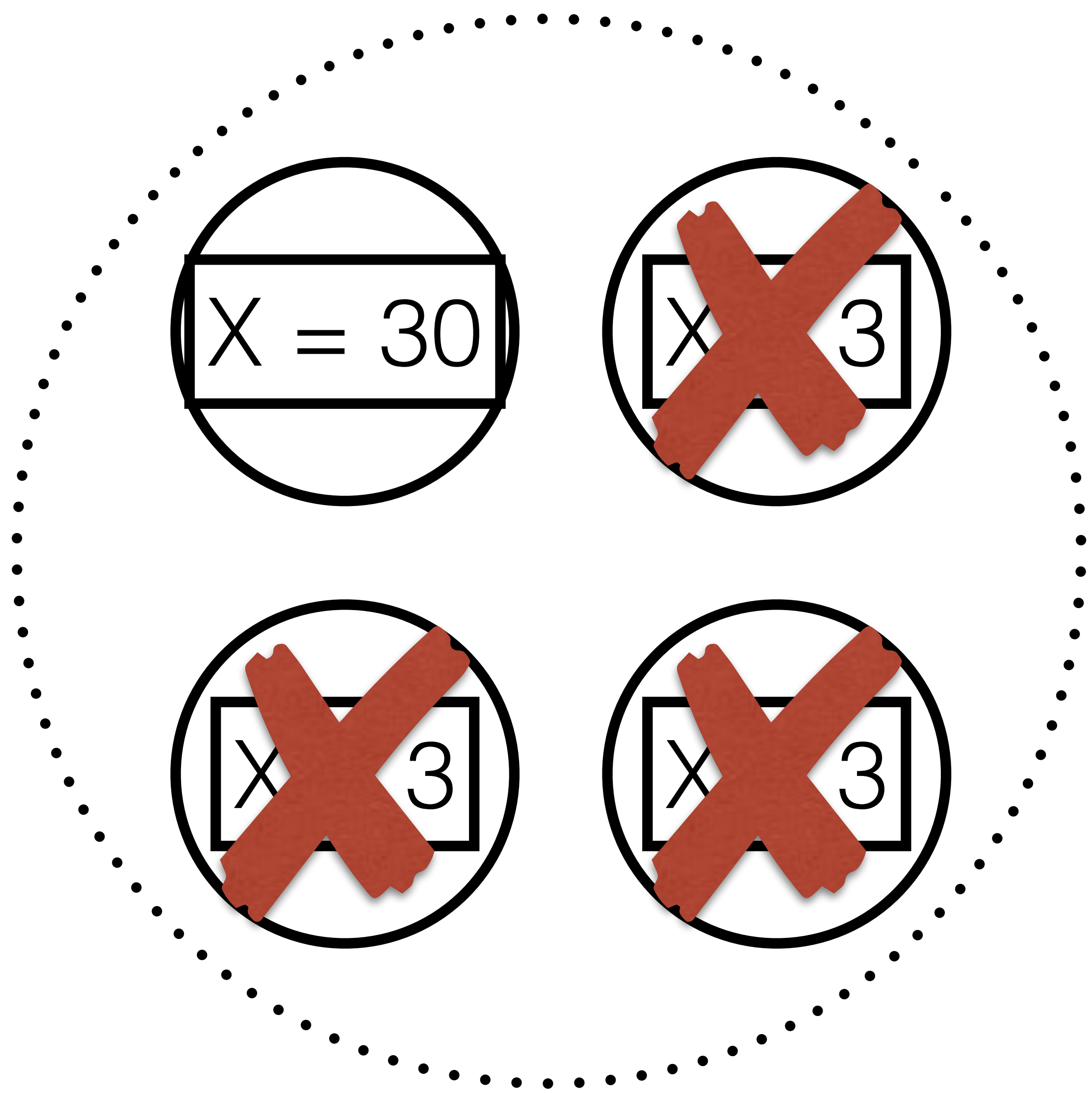
Req.	*CUID*	*UID*
r0	*1.1*	*1.1*



2) Accept *r0*

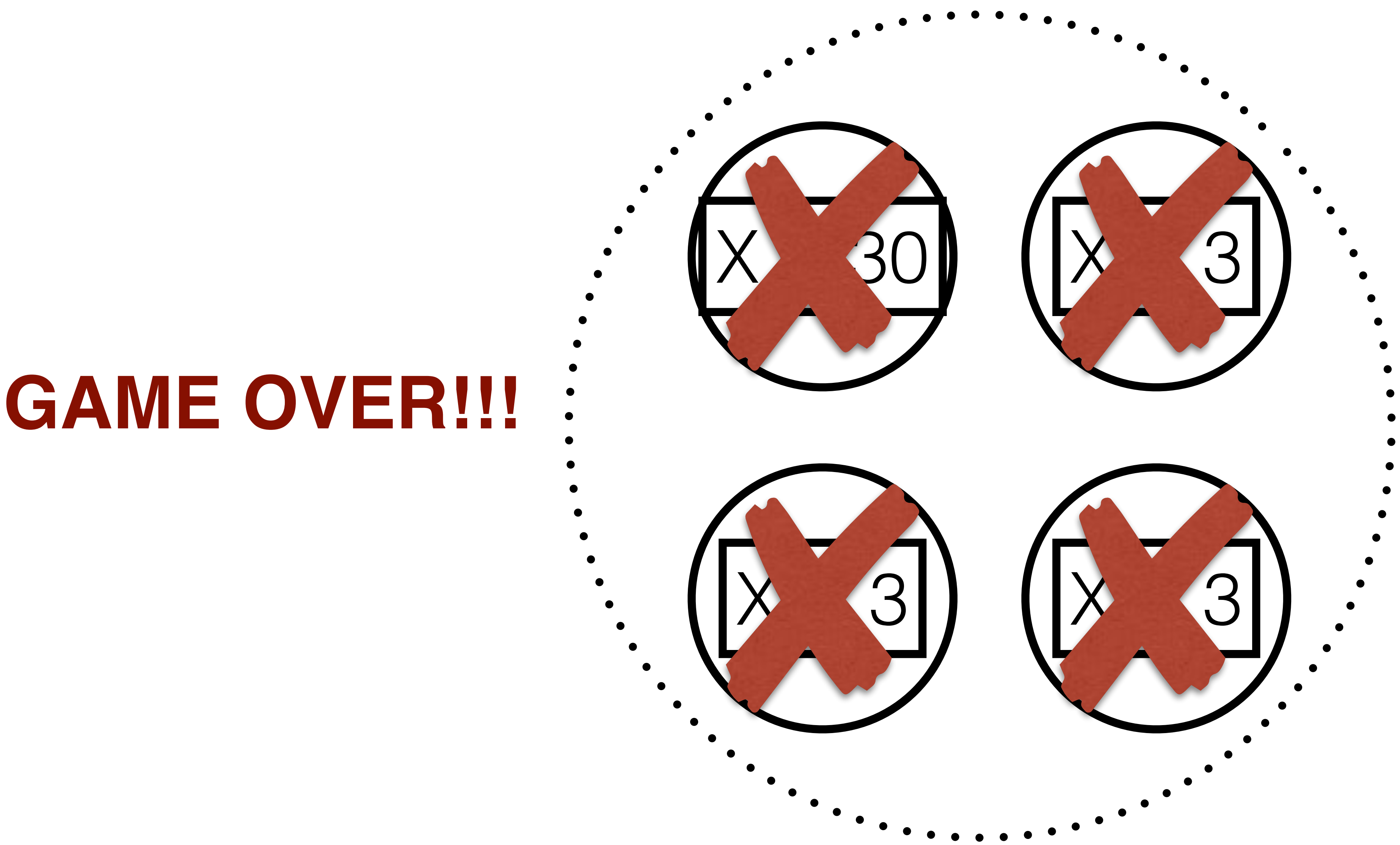
Fail-Stop Tolerance

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



2) Apply *r0*

Fail-Stop Tolerance

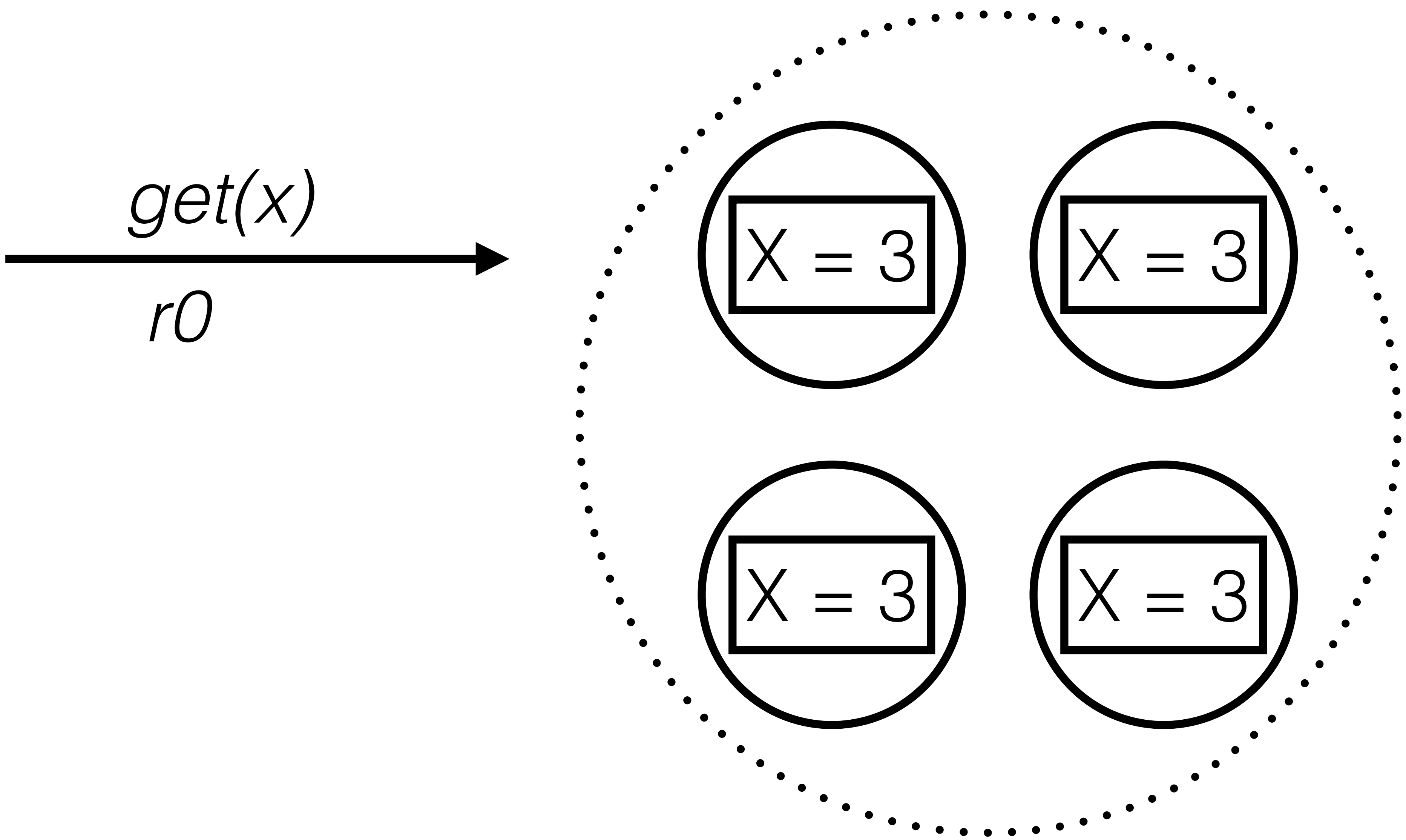


2) Apply *r0*

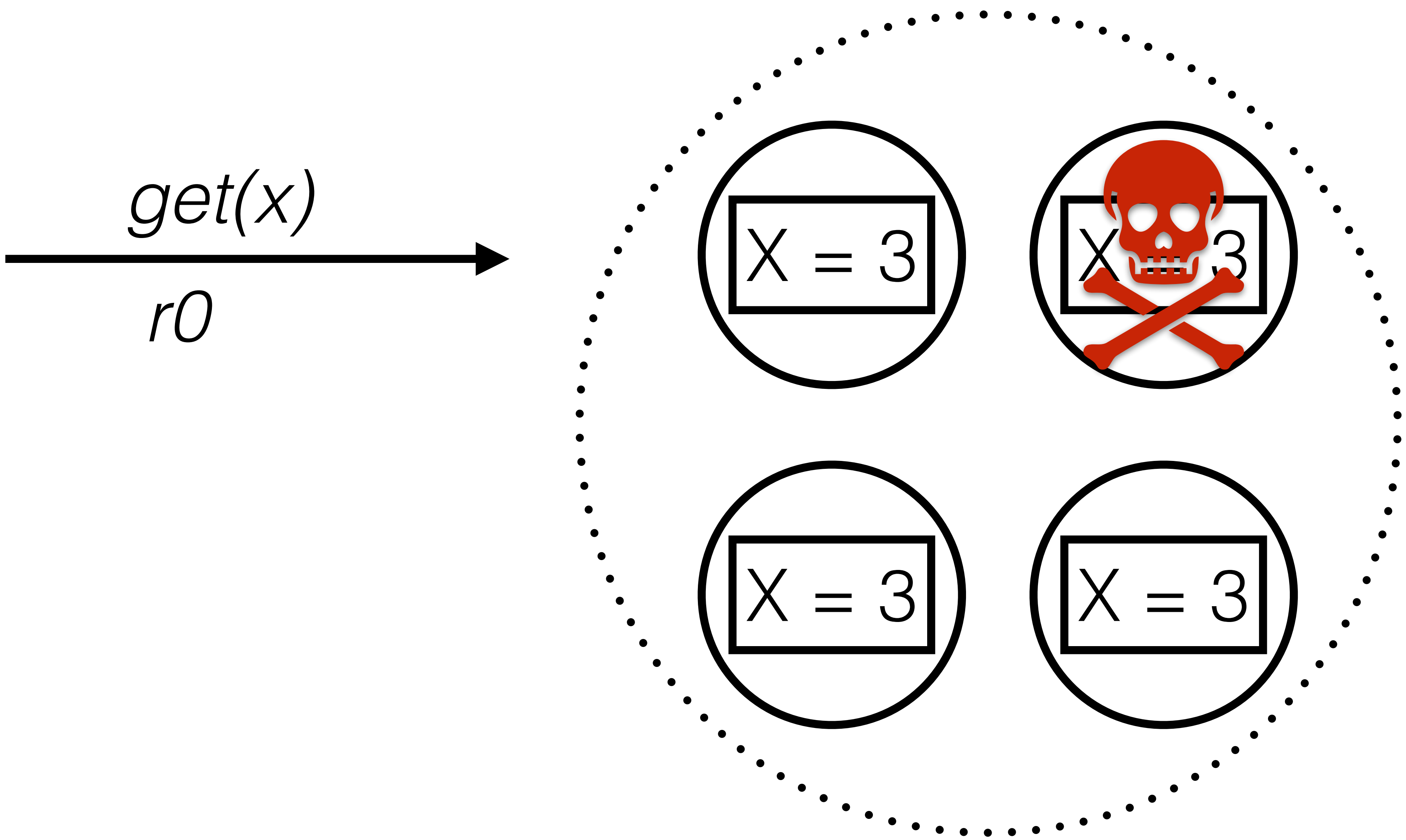
Fail-Stop Tolerance

- 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

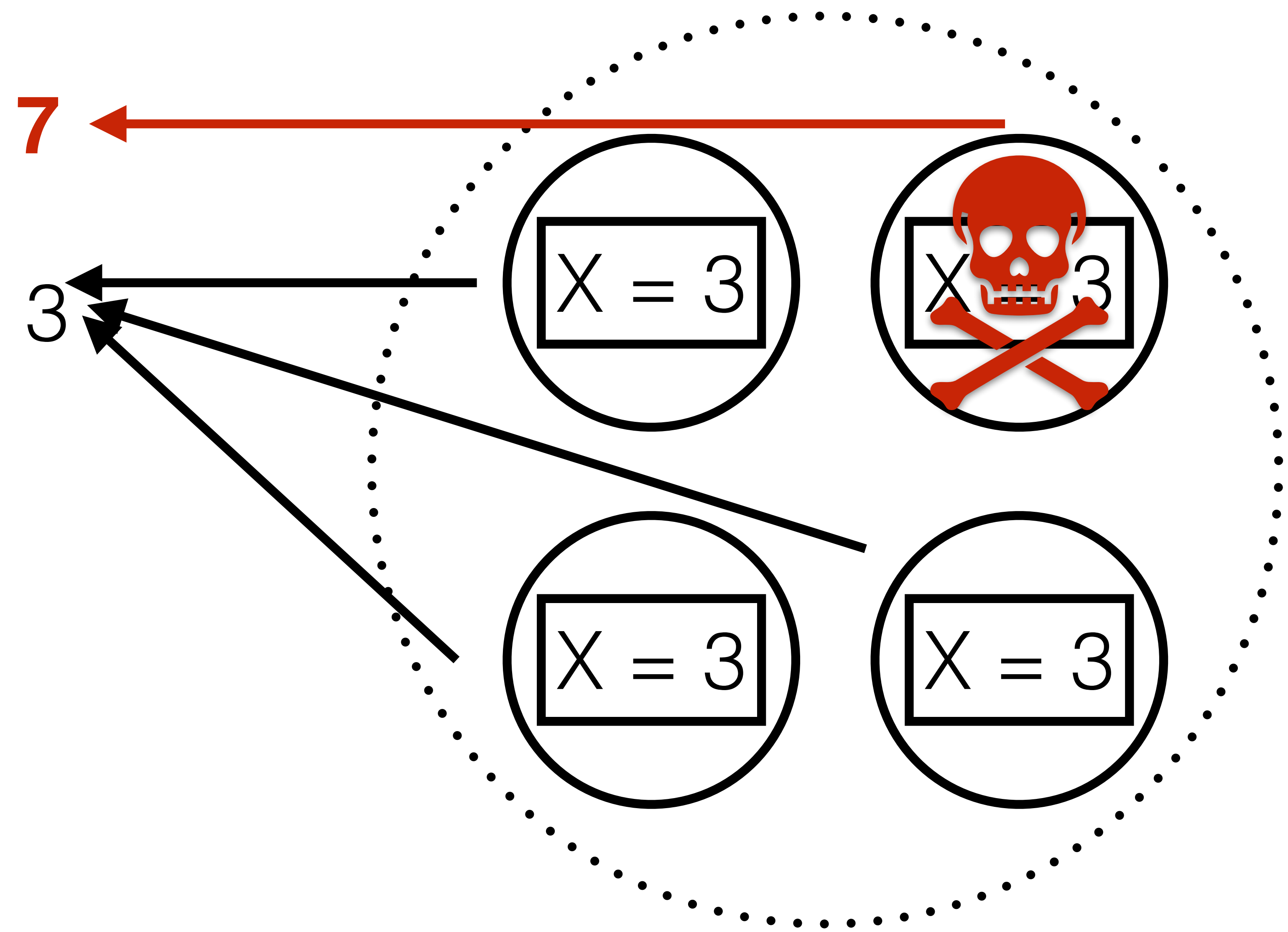
Byzantine Tolerance



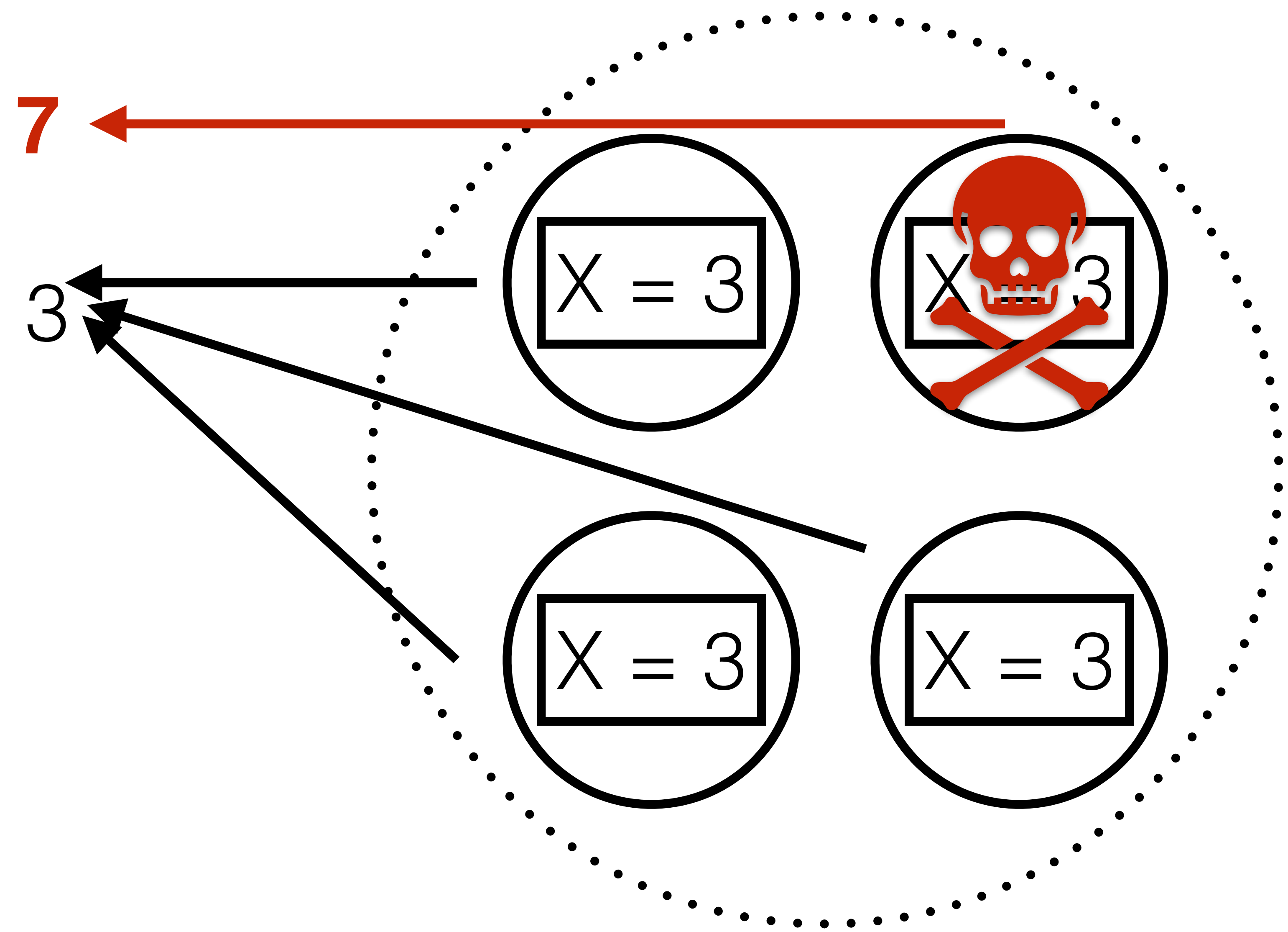
Byzantine Tolerance



Byzantine Tolerance

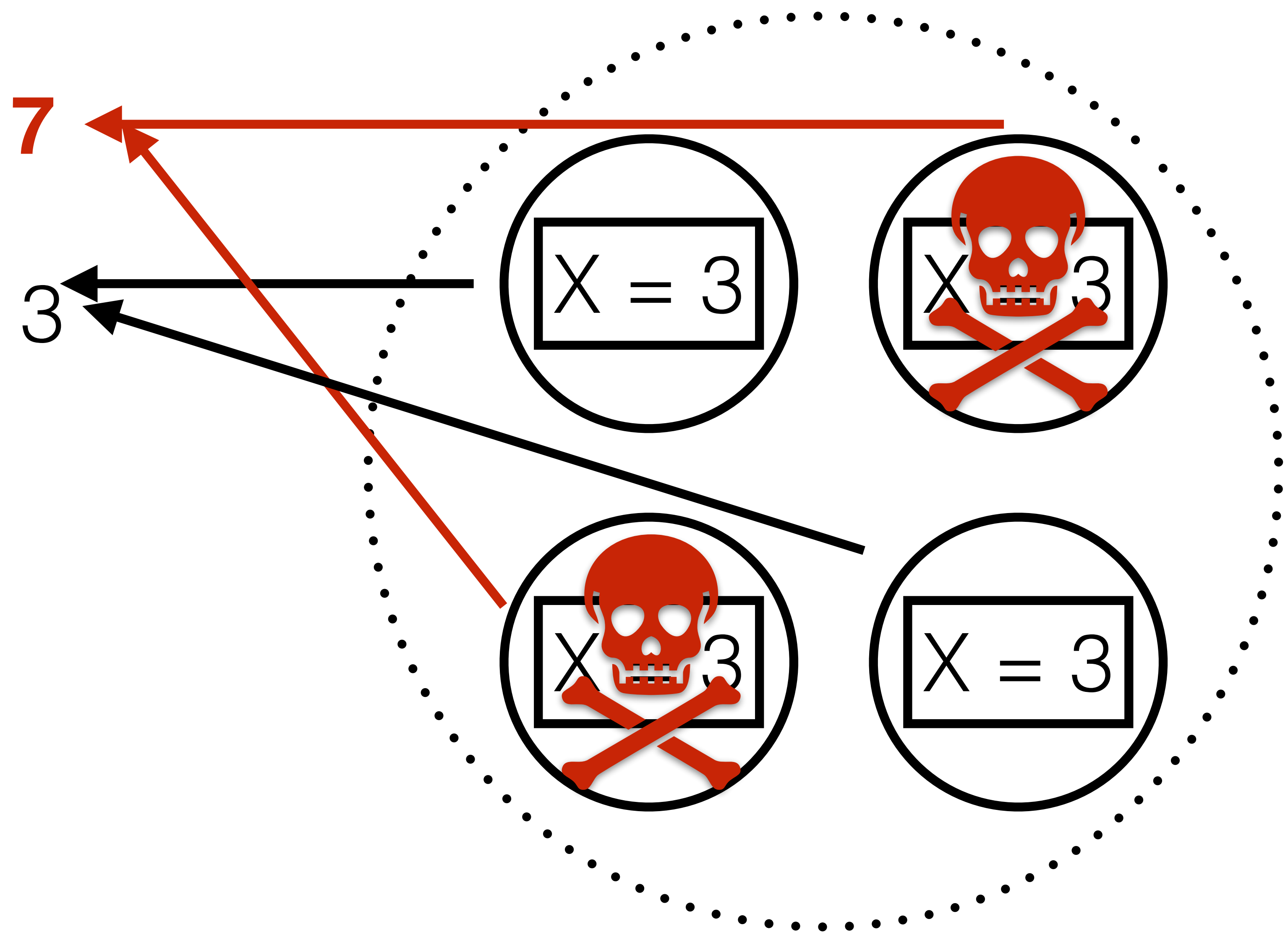


Byzantine Tolerance



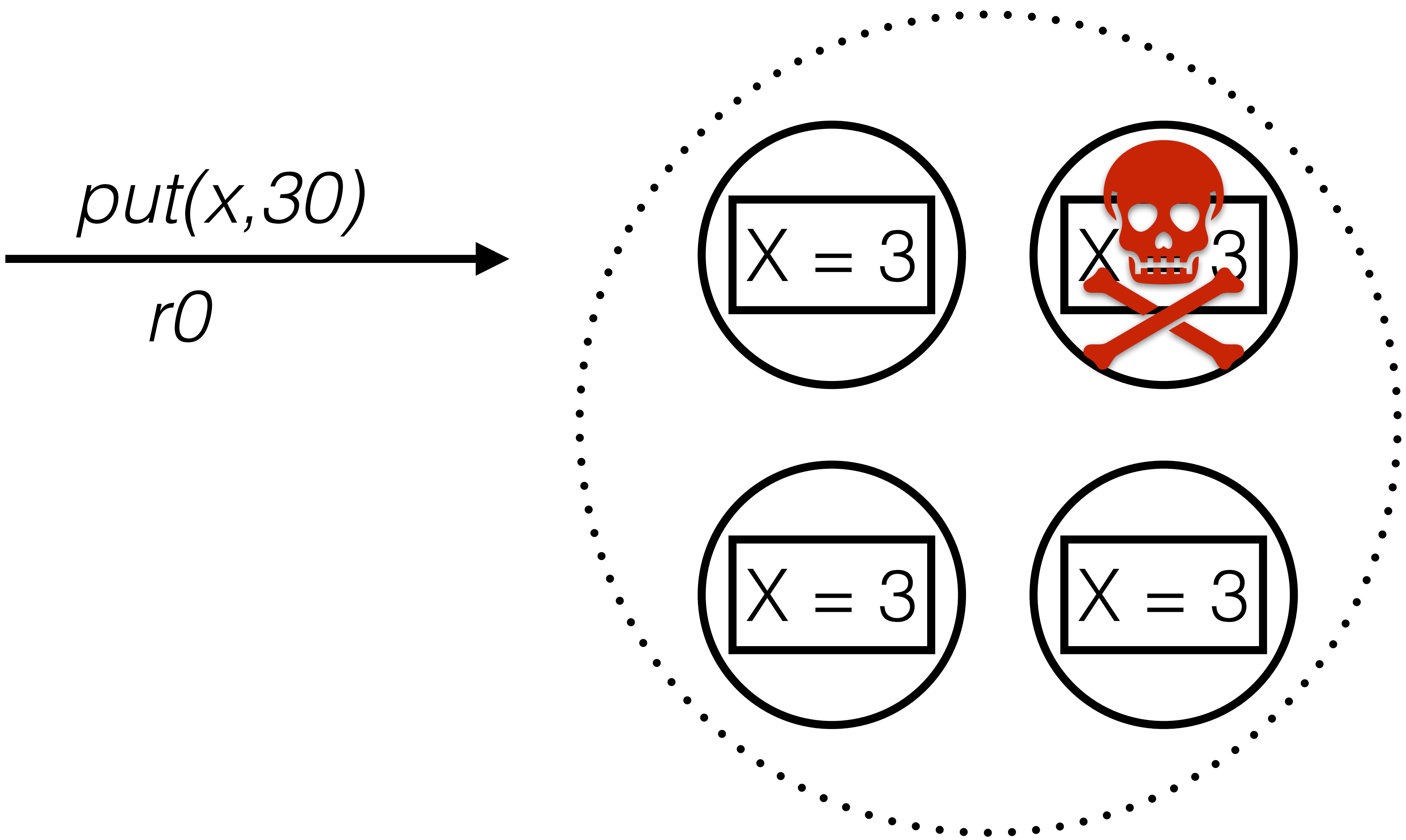
Client trusts the majority =>
Need majority to participate in replication

Byzantine Tolerance



Who to trust?? 3 or 7?

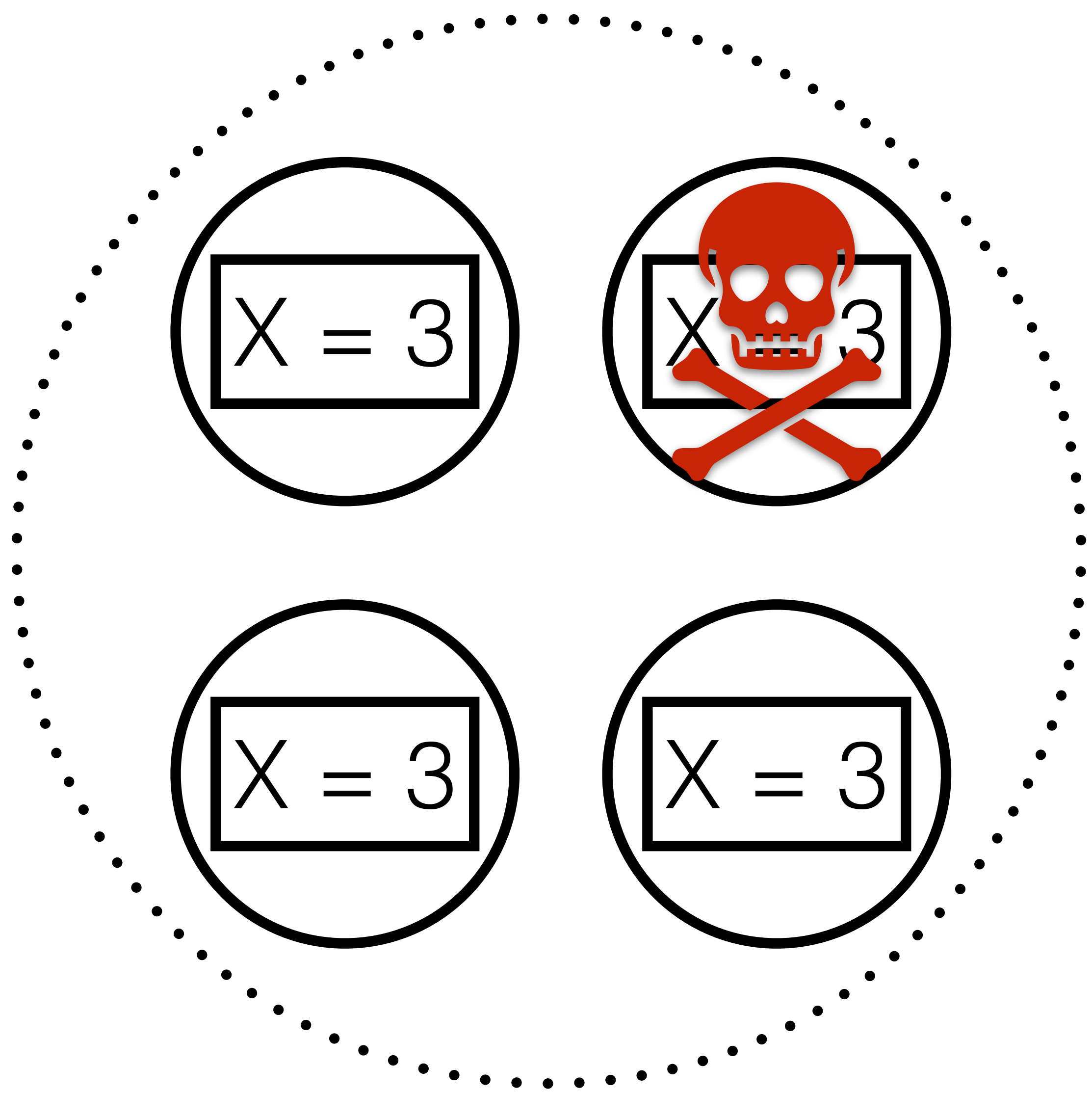
Byzantine Tolerance



Byzantine Tolerance

Req.	CUID	UID
r0	1.1	

Req.	CUID	UID
r0	1.2	



Req.	CUID	UID
r0	???	???

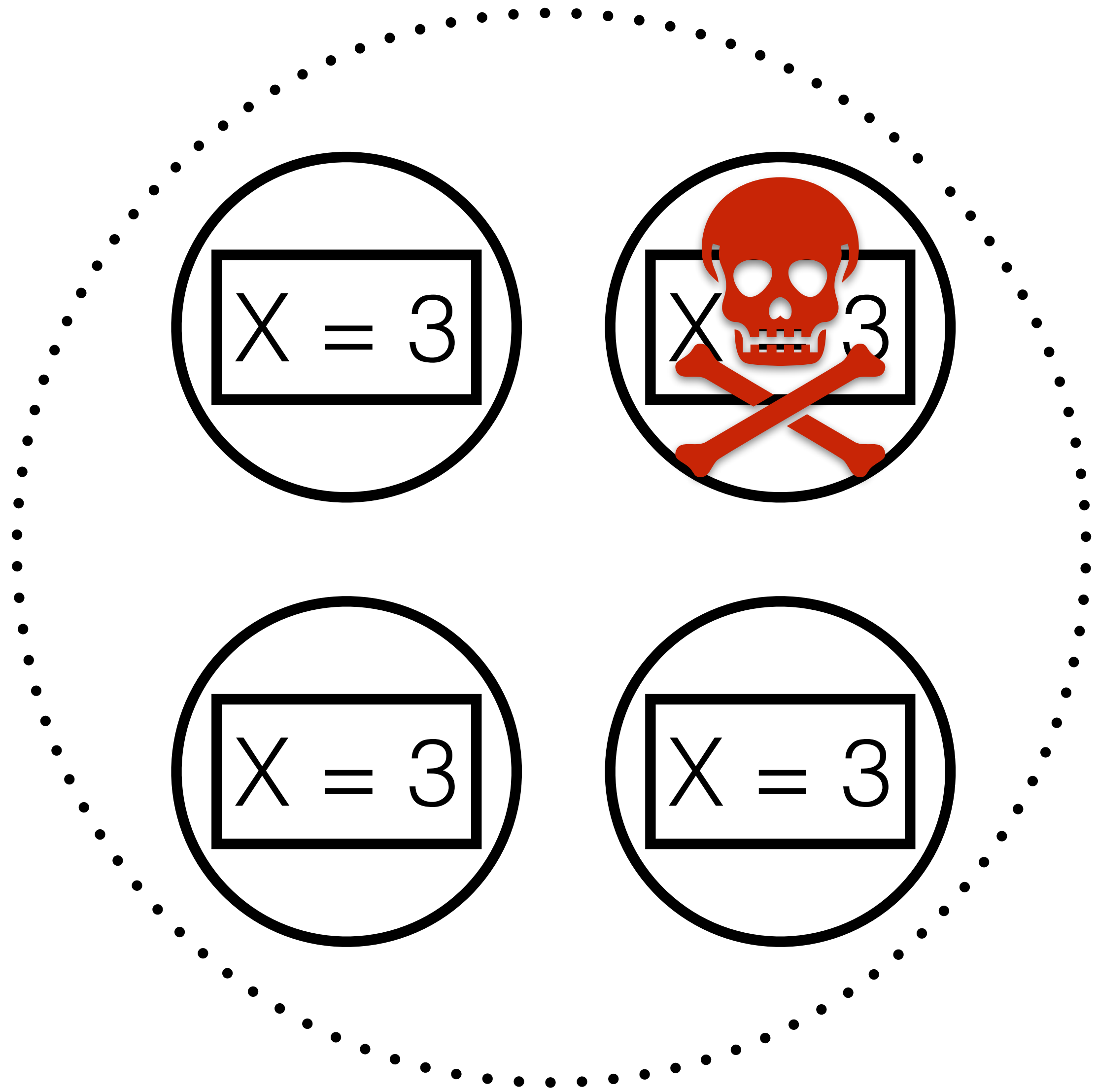
Req.	CUID	UID
r0	1.4	

1) Propose Candidates

Byzantine Tolerance
a) No response

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

Req.	*CUID*	*UID*
r0	*1.2*	



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

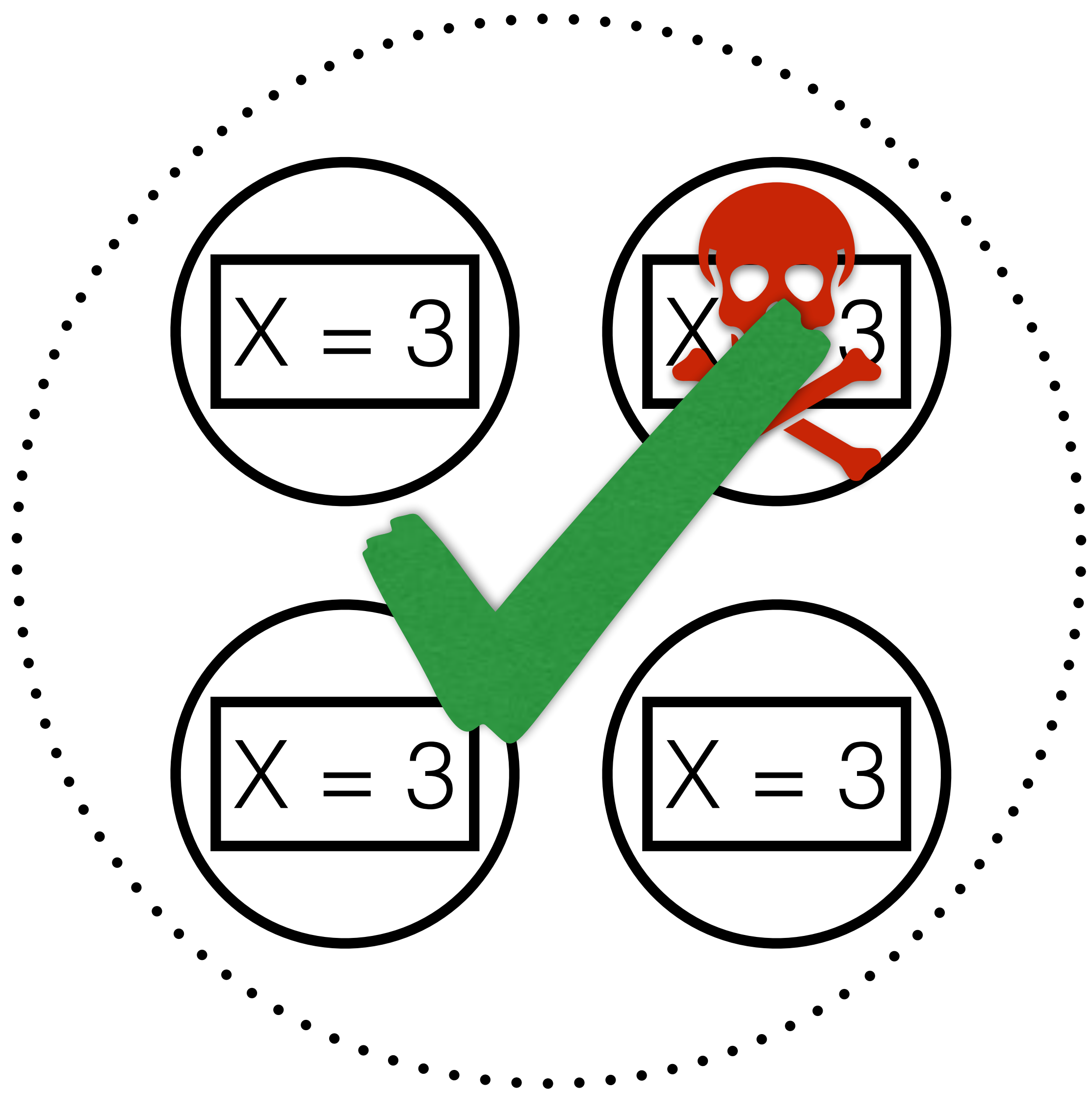
Req.	*CUID*	*UID*
r0	*1.4*	

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	???	???

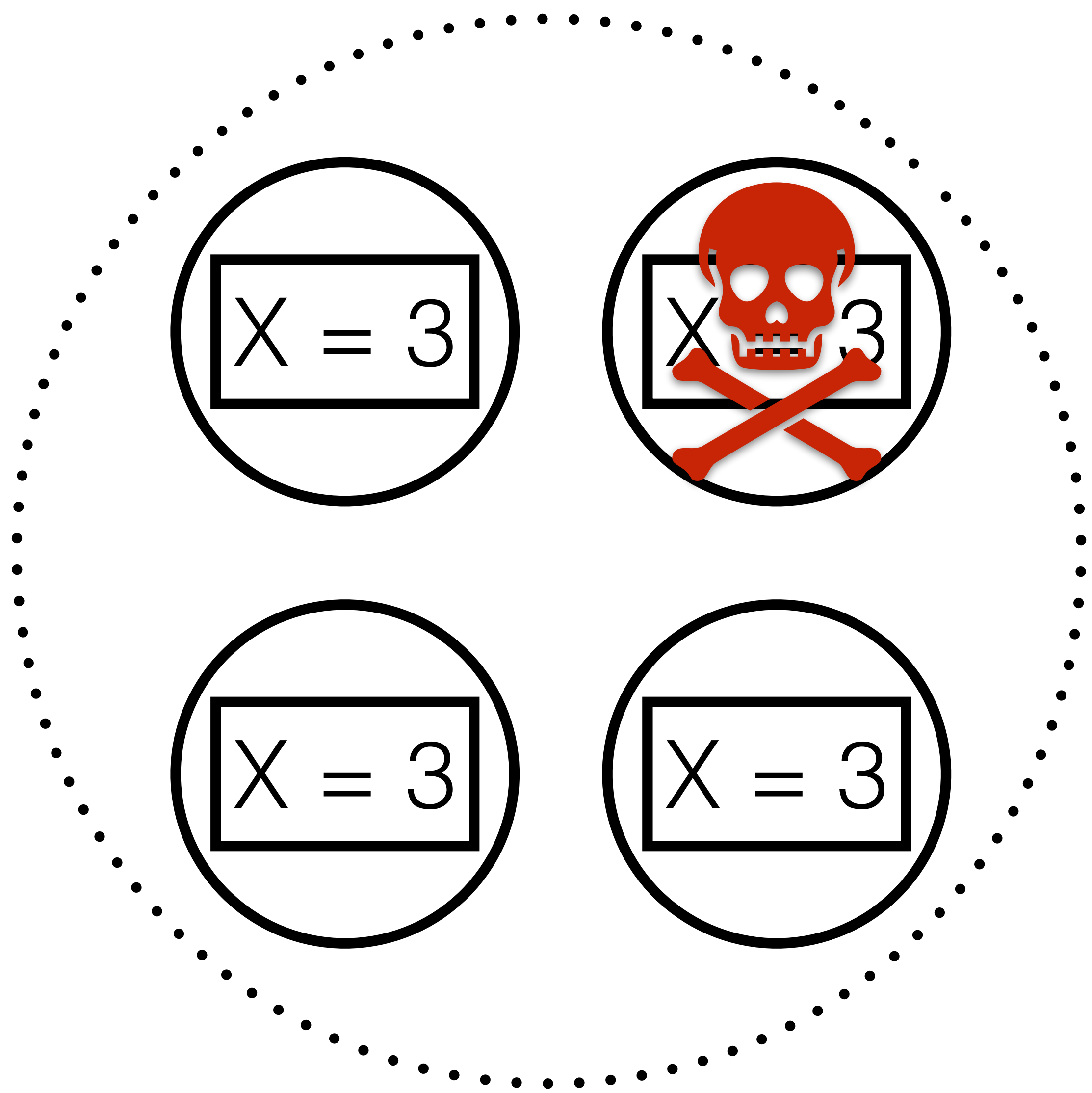
Req.	CUID	UID
r0	1.4	1.4

a) *Accept r0*

Byzantine Tolerance
Small ID

Req.	CUID	UID
r0	1.1	

Req.	CUID	UID
r0	1.2	



Req.	CUID	UID
r0	**-5**	???

Req.	CUID	UID
r0	1.4	

1) Propose Candidates

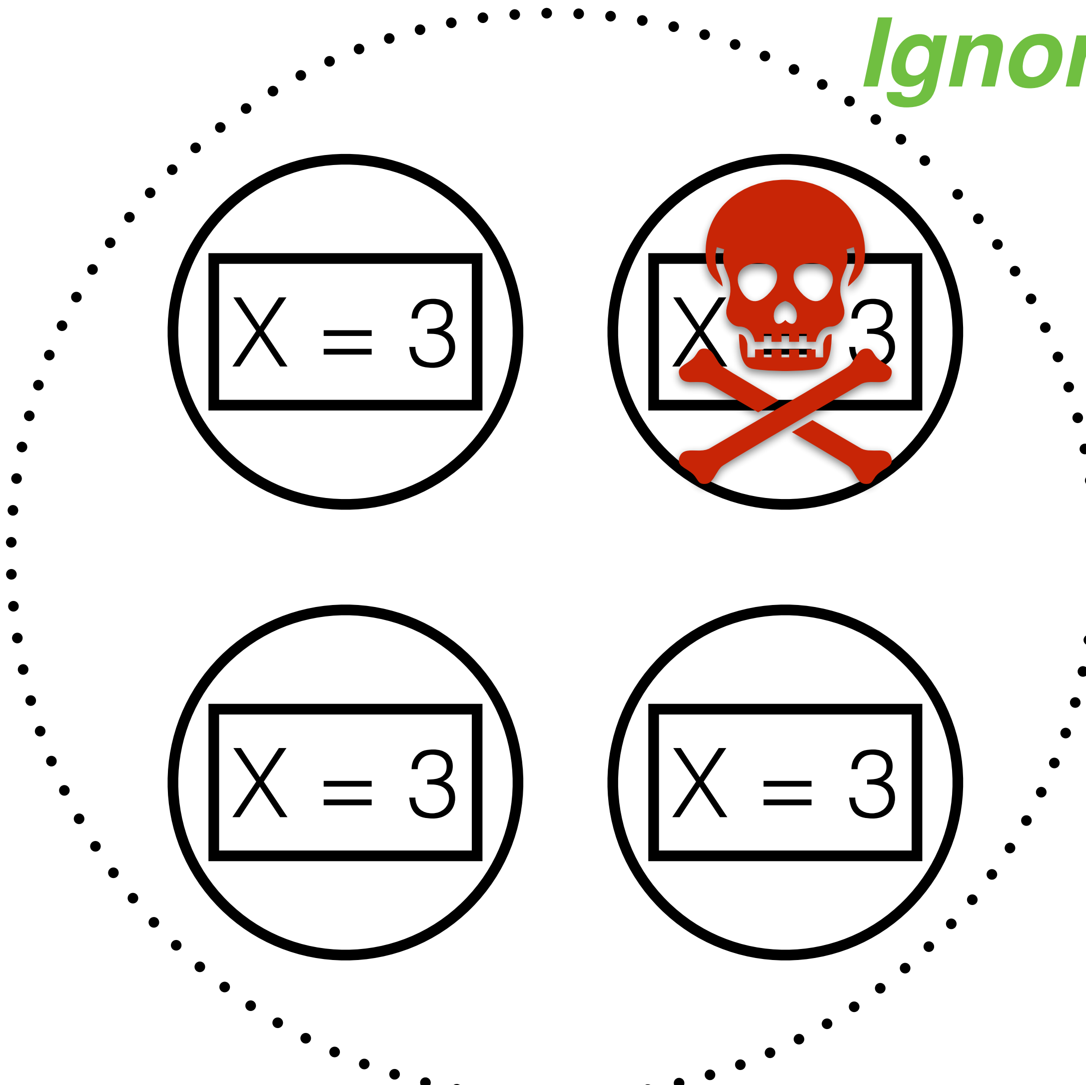
Byzantine Tolerance
Small ID

$uid = max(cuid(sm_i, r))$

Ignore low candidates!

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

Req.	*CUID*	*UID*
r0	***-5***	*???*



Req.	*CUID*	*UID*
r0	*1.2*	

Req.	*CUID*	*UID*
r0	*1.4*	

2) *Accept r0*

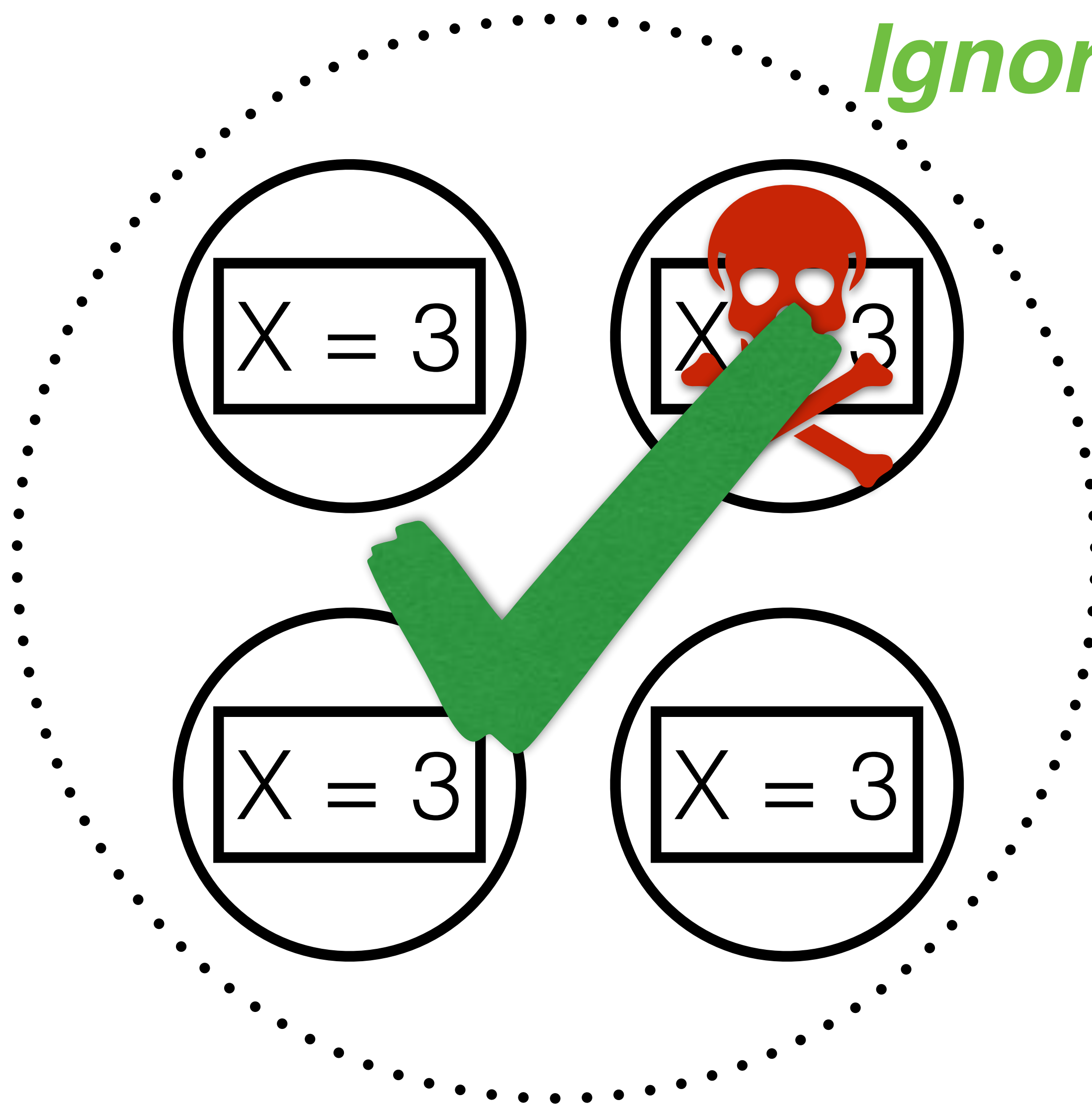
Byzantine Tolerance
Small ID

$uid = max(cuid(sm_i,r))$

Ignore low candidates!

Req.	*CUID*	*UID*
r0	*1.1*	*1.4*

Req.	*CUID*	*UID*
r0	*1.2*	*1.4*



Req.	*CUID*	*UID*
r0	***-5***	*???*

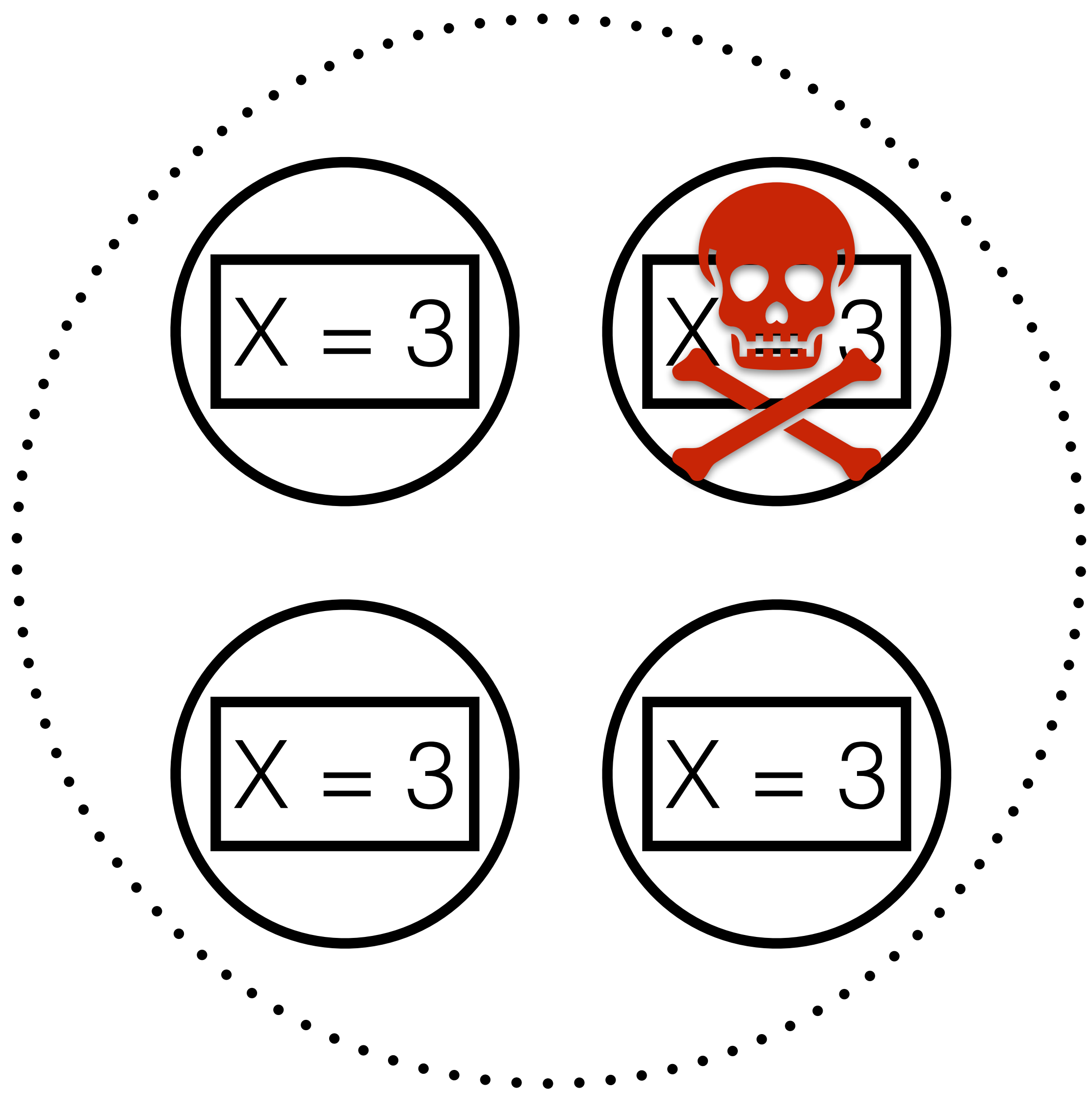
Req.	*CUID*	*UID*
r0	*1.4*	*1.4*

2) *Accept r0*

Byzantine Tolerance
Large ID

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

Req.	*CUID*	*UID*
r0	*1.2*	



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

Req.	*CUID*	*UID*
r0	*1.4*	

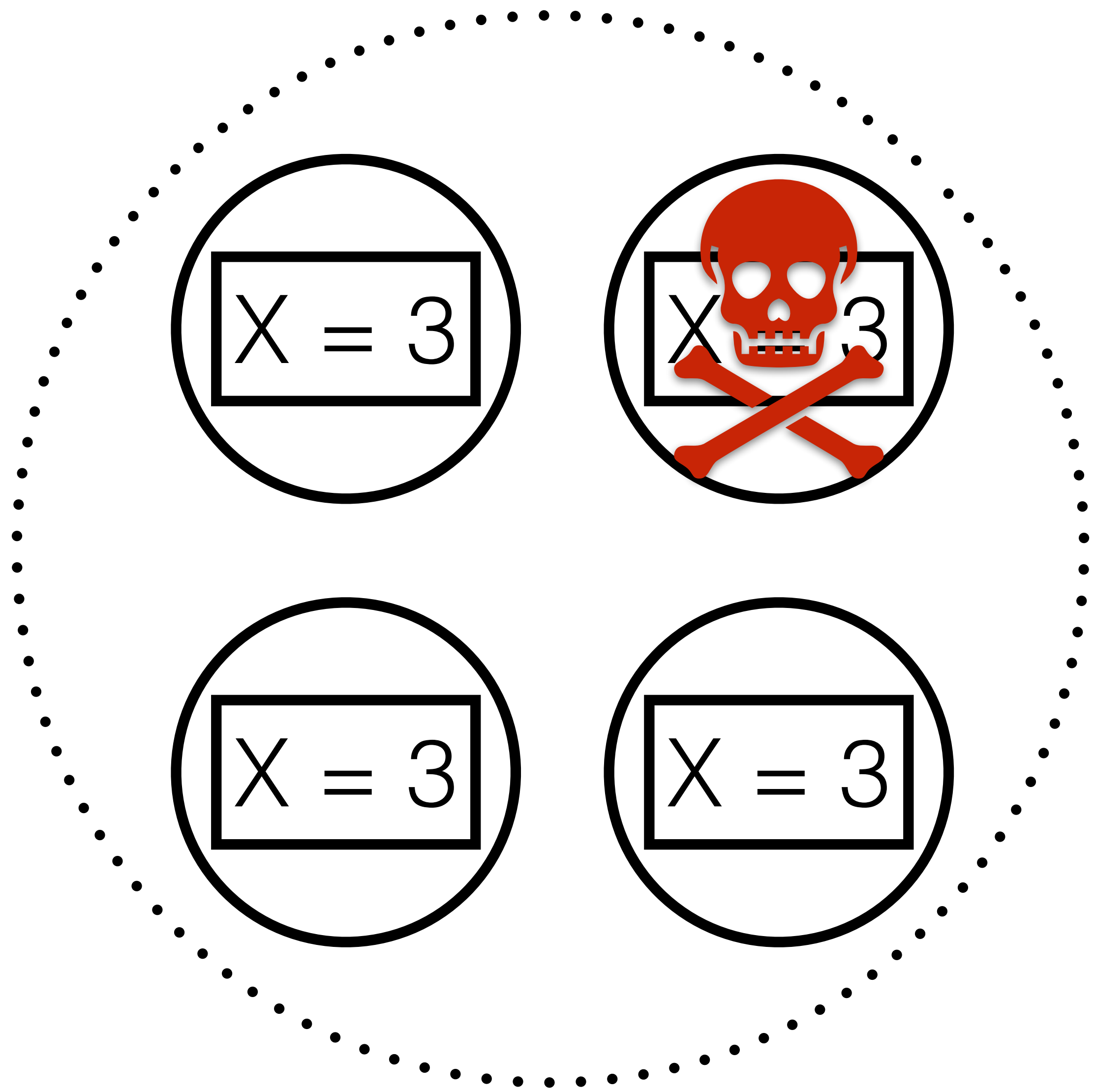
1) Propose Candidates

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**	???

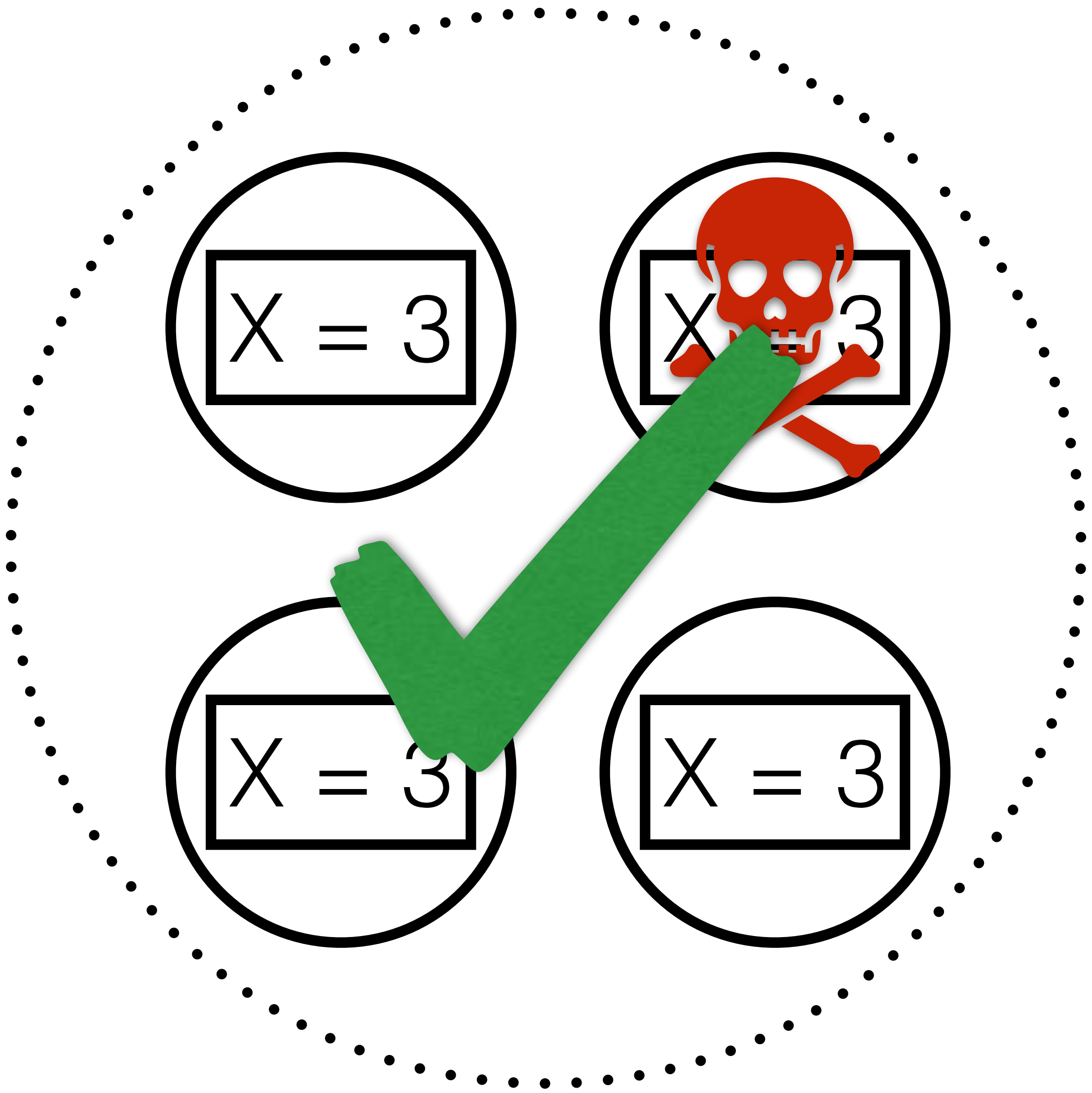
Req.	CUID	UID
r0	1.4	

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

Other Contributions

- Tolerating Faulty Output Devices
  - (e.g. a faulty network, or user-facing i/o)
- Tolerating Faulty Clients
- Reconfiguration

Takeaways

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*,

Lamport 1978

Takeaways

- 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*

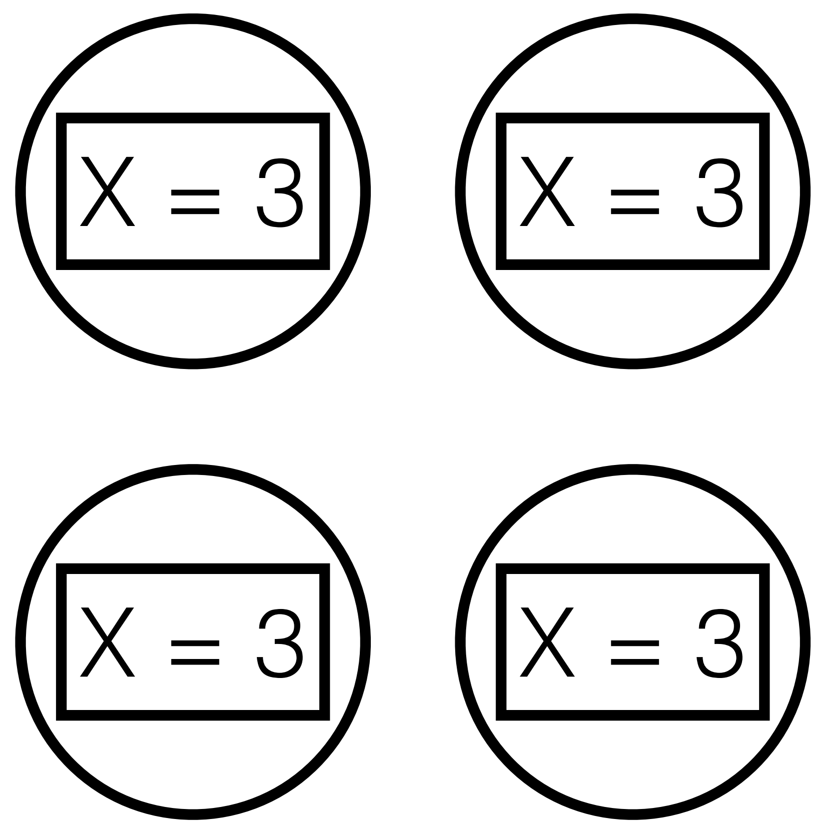
Chain Replication

- Authors
  - Robert Van Renesse (RVR)
  - Fred Schneider

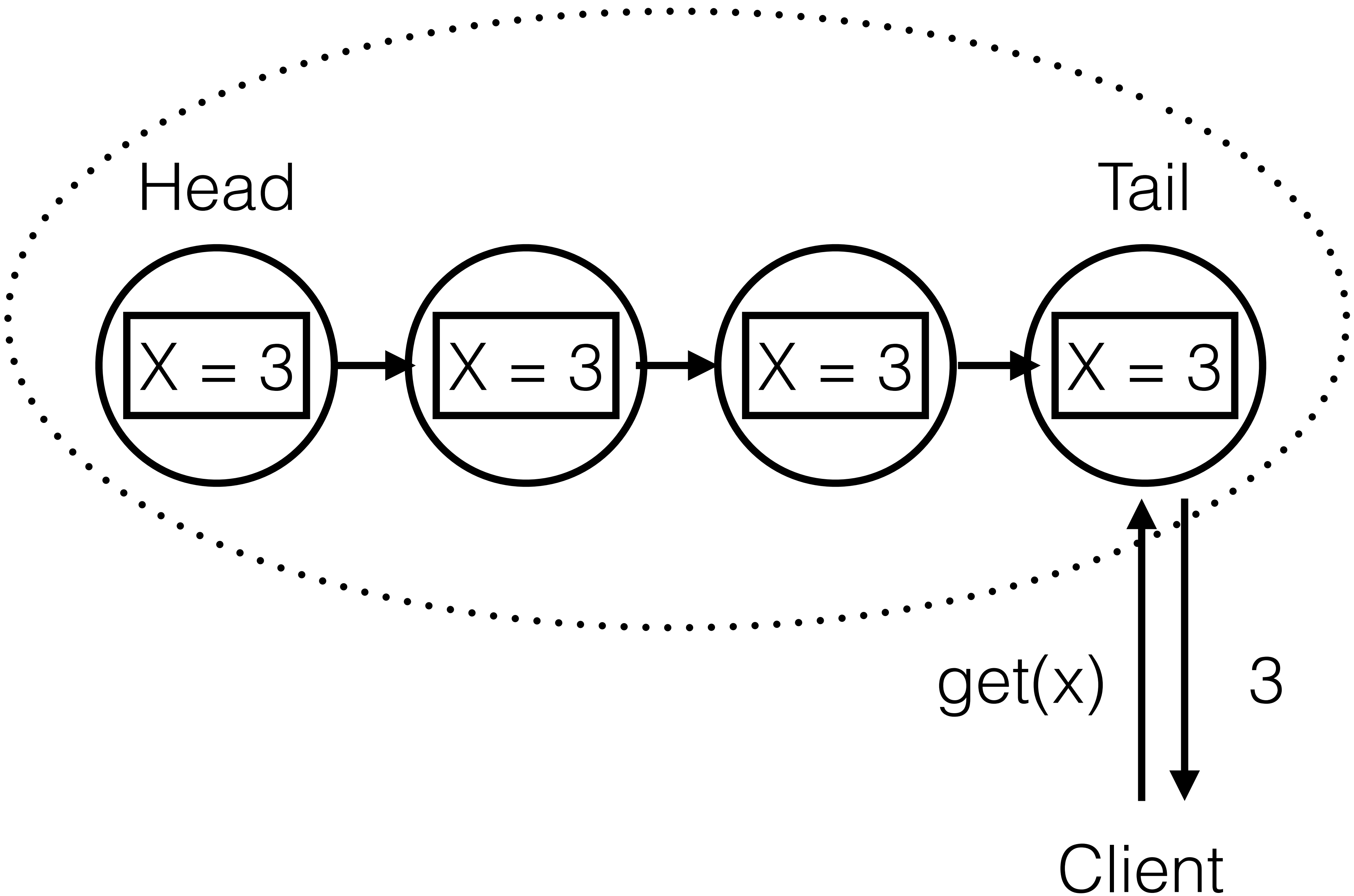
Chain Replication

- Fault Tolerant Storage Service (Fail-Stop)
- Requests:
  - Update(x, y) => set object *x* to value *y*
  - Query(x) => read value of object *x*

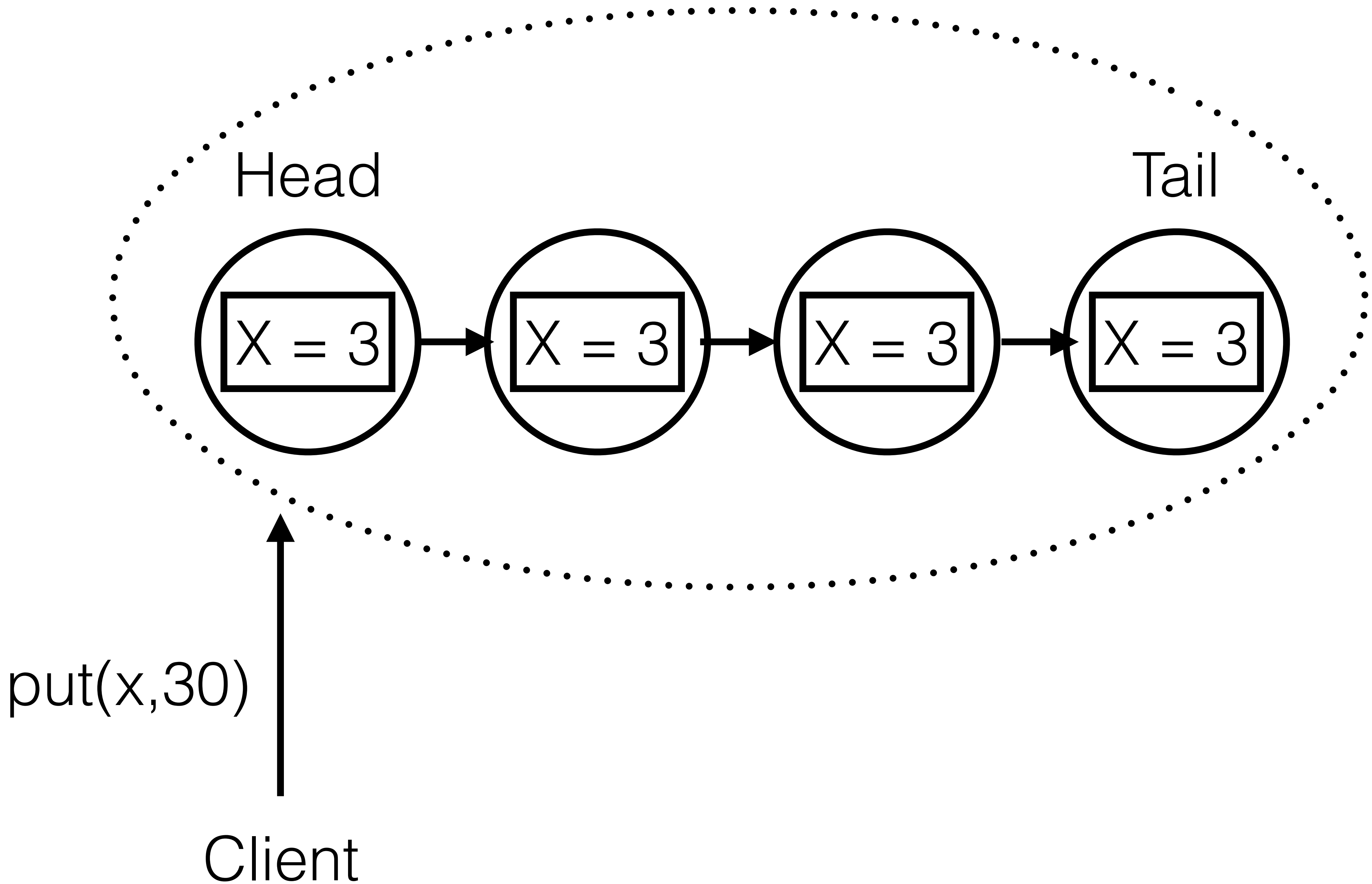
Chain Replication



Chain Replication

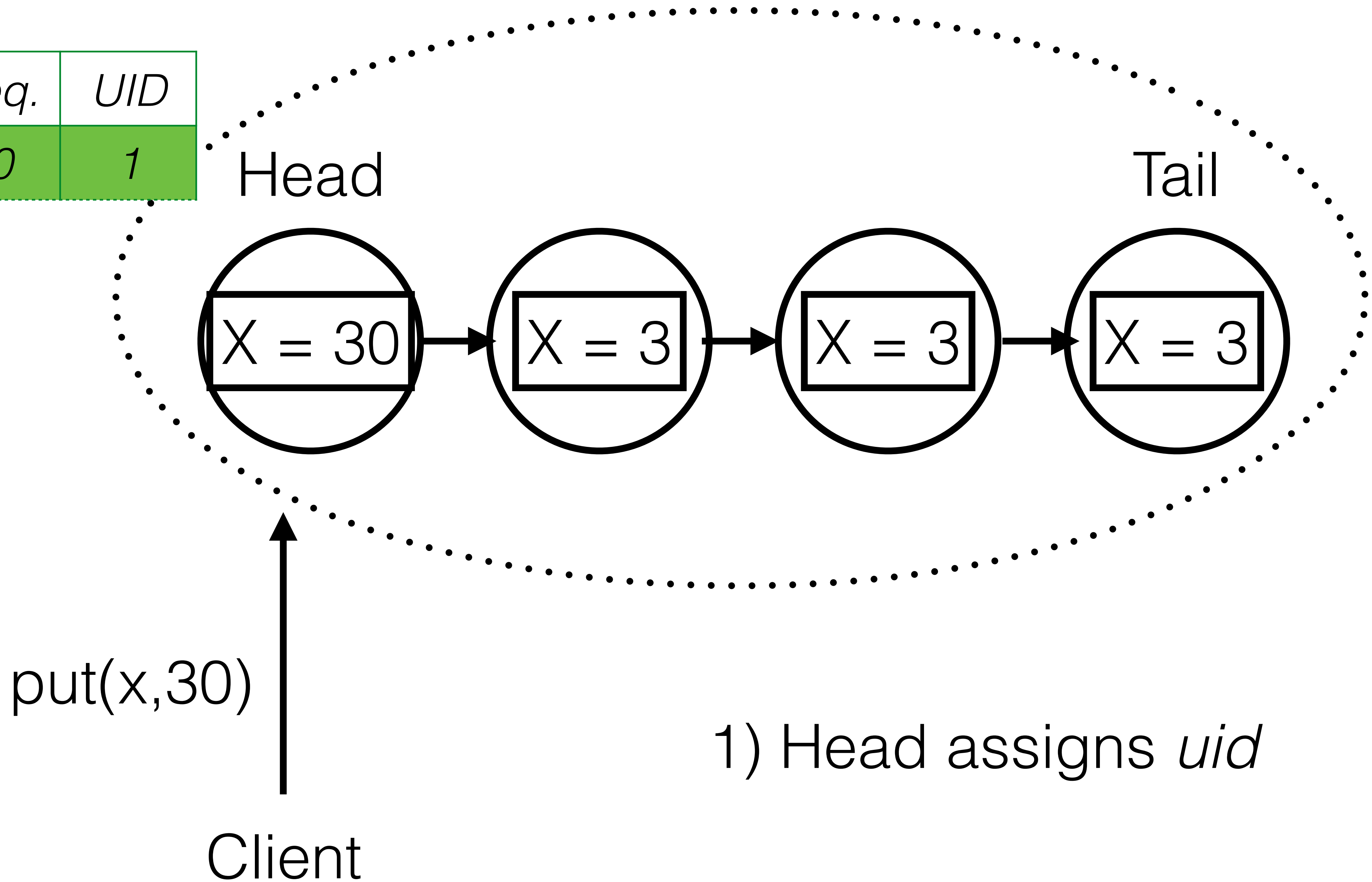


Chain Replication



Chain Replication

Req.	*UID*
r0	*1*

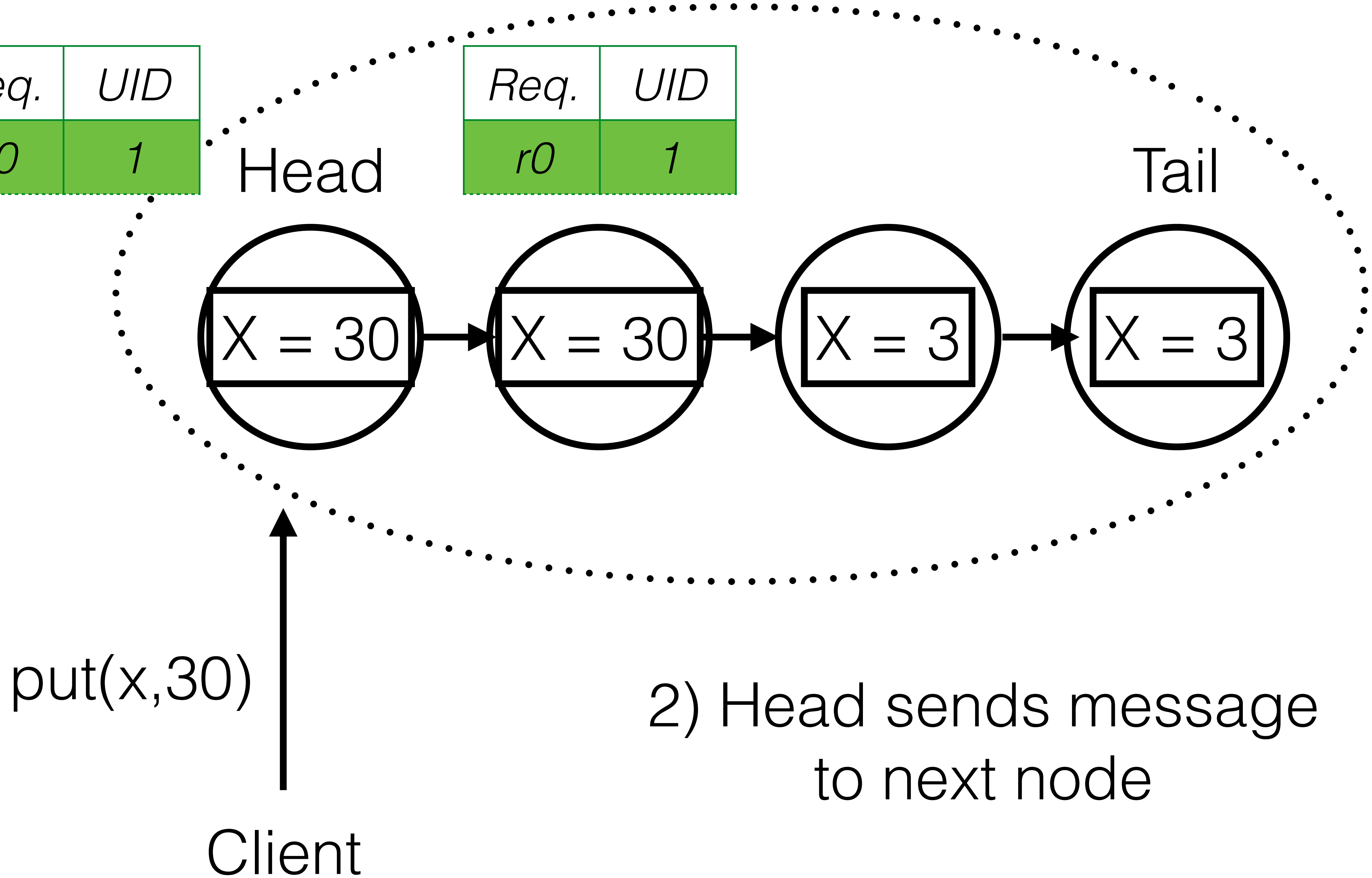


1) Head assigns *uid*

Chain Replication

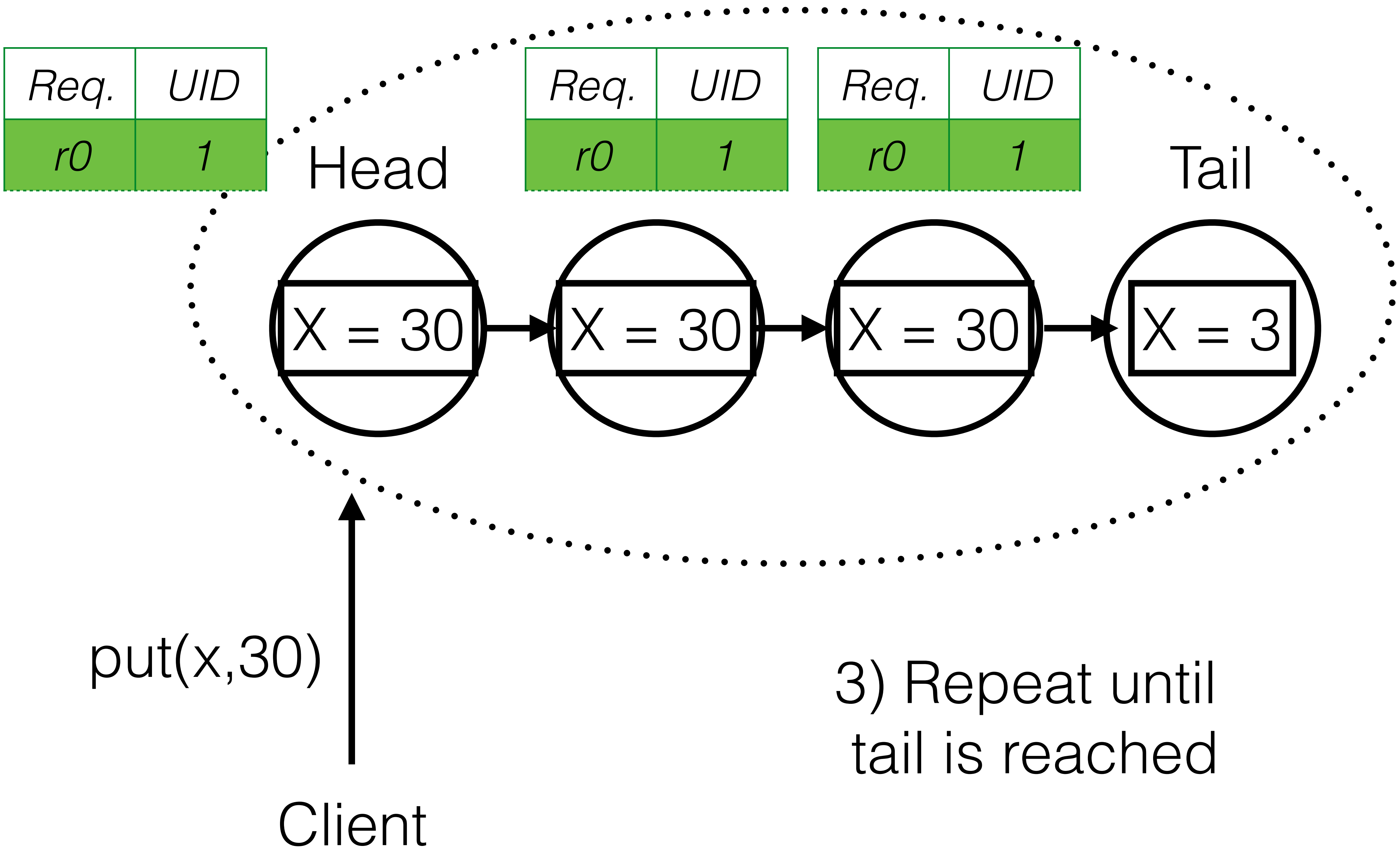
Req.	*UID*
r0	*1*

Req.	*UID*
r0	*1*

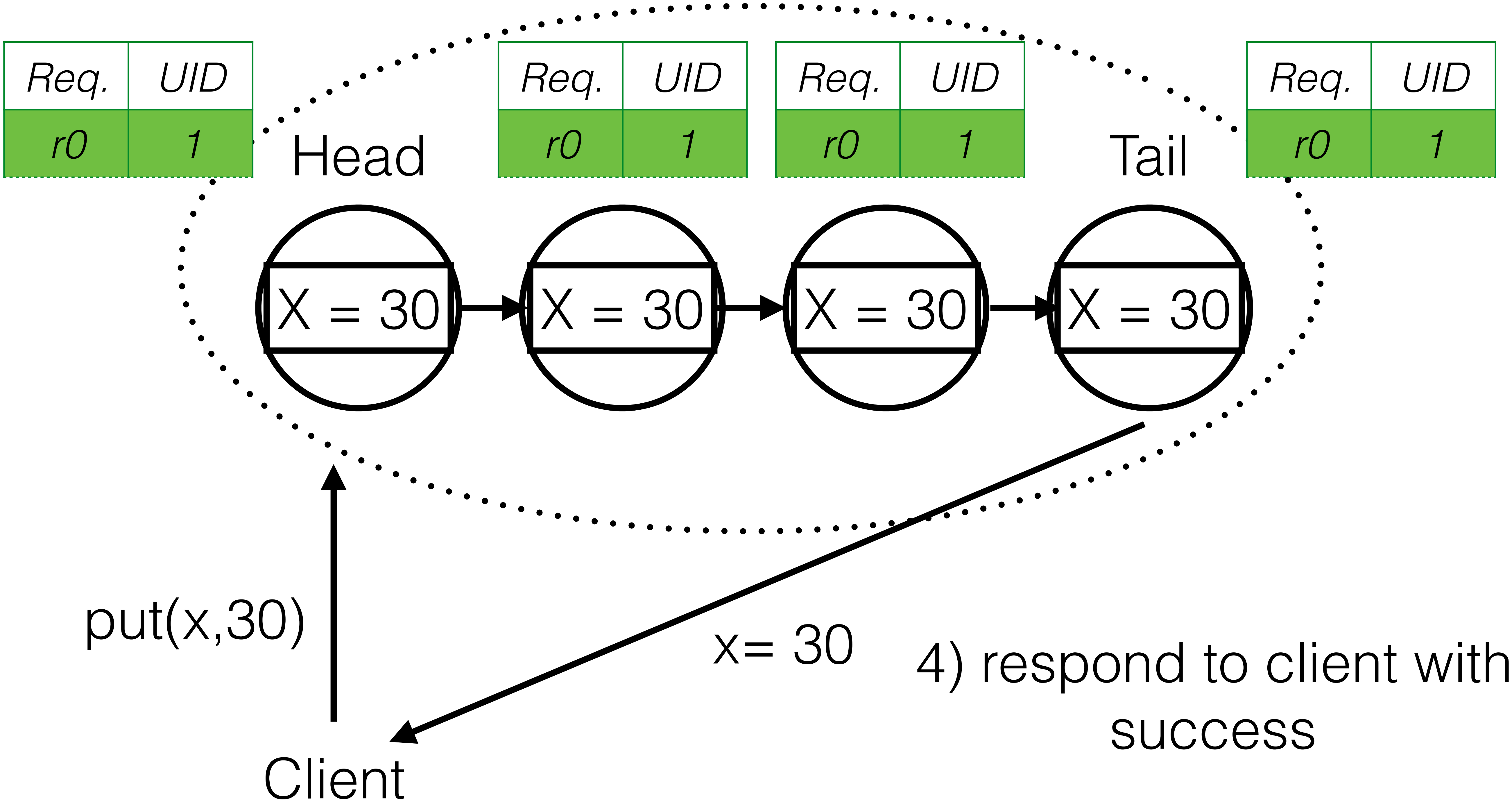


2) Head sends message to next node

Chain Replication



Chain Replication



Chain Replication

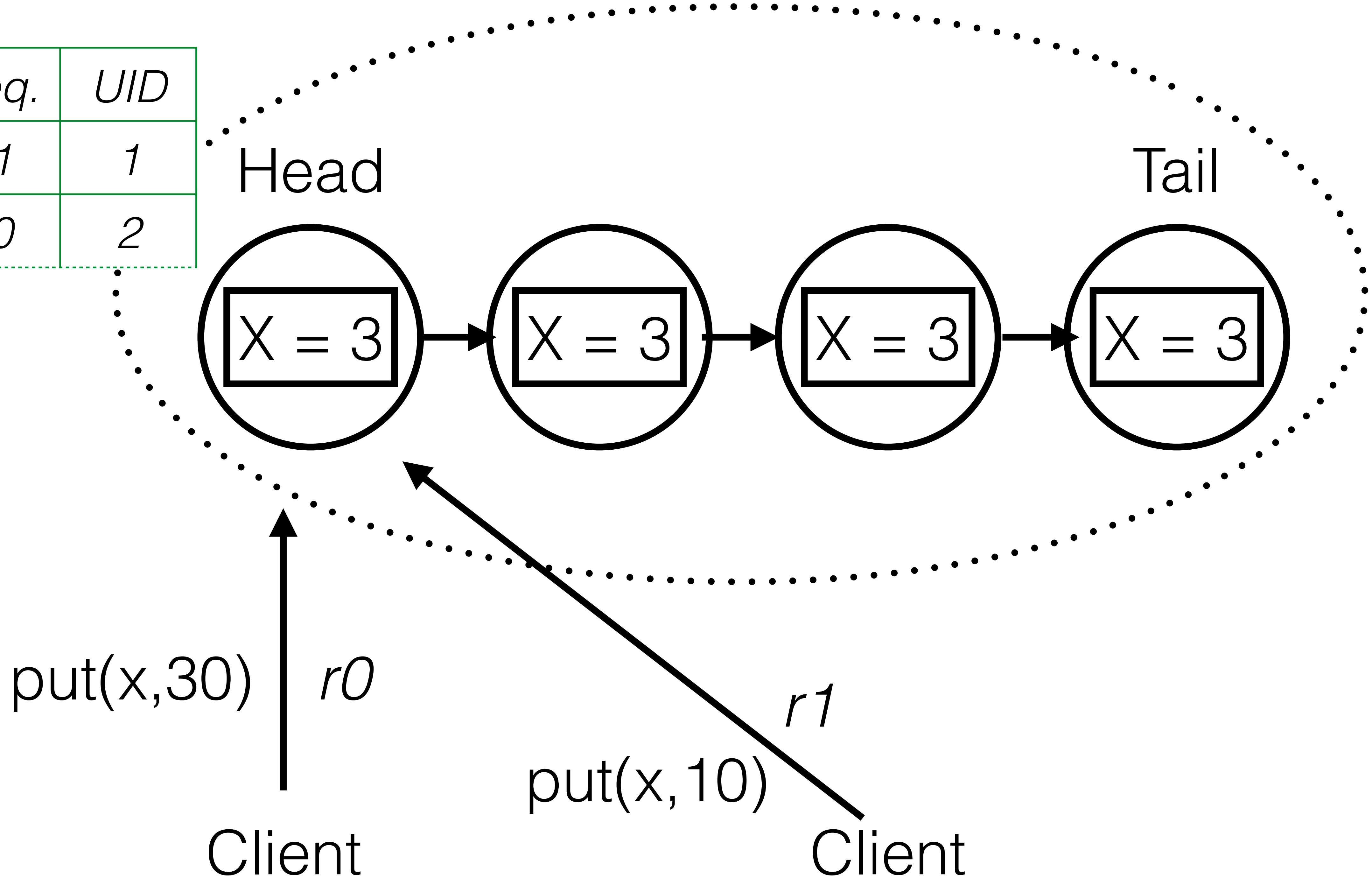
- How does Chain Replication implement State Machine Replication?
- Agreement
  - Only *Update* modifies state, can ignore *Query*
  - Client always sends *update* to *Head*. *Head* propagates request down chain to *Tail*.
  - Everyone accepts the request!

Chain Replication

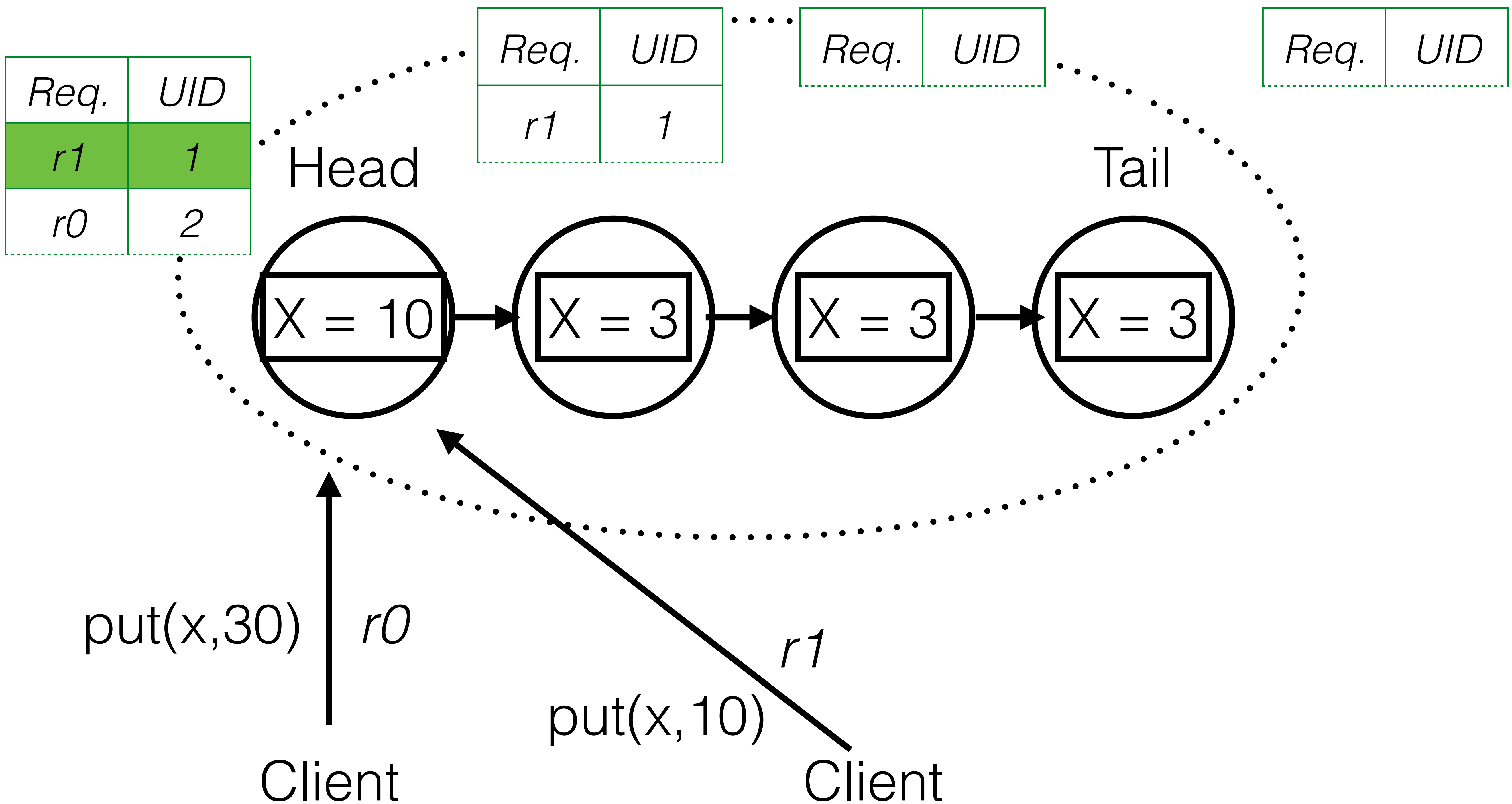
- How does Chain Replication implement State Machine Replication?
- Order
  - Unique IDs generated implicitly by *Head*'s ordering
  - FIFO order preserved down the chain
  - Tail interleaves *Query* requests
  - How can clients test stability? (How can clients tell when their *Updates* have been handled)

Chain Replication

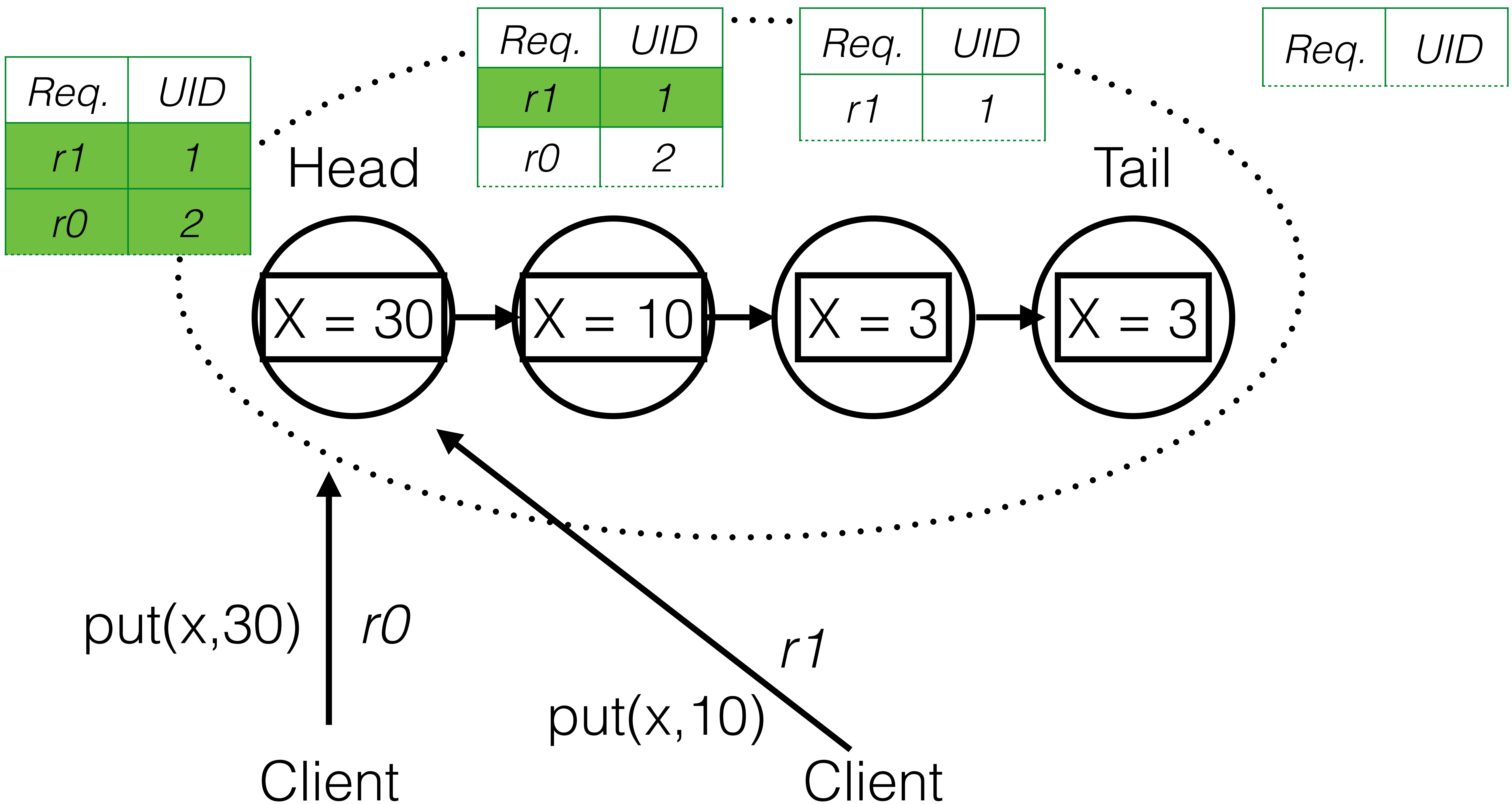
Req.	*UID*
r1	*1*
r0	*2*



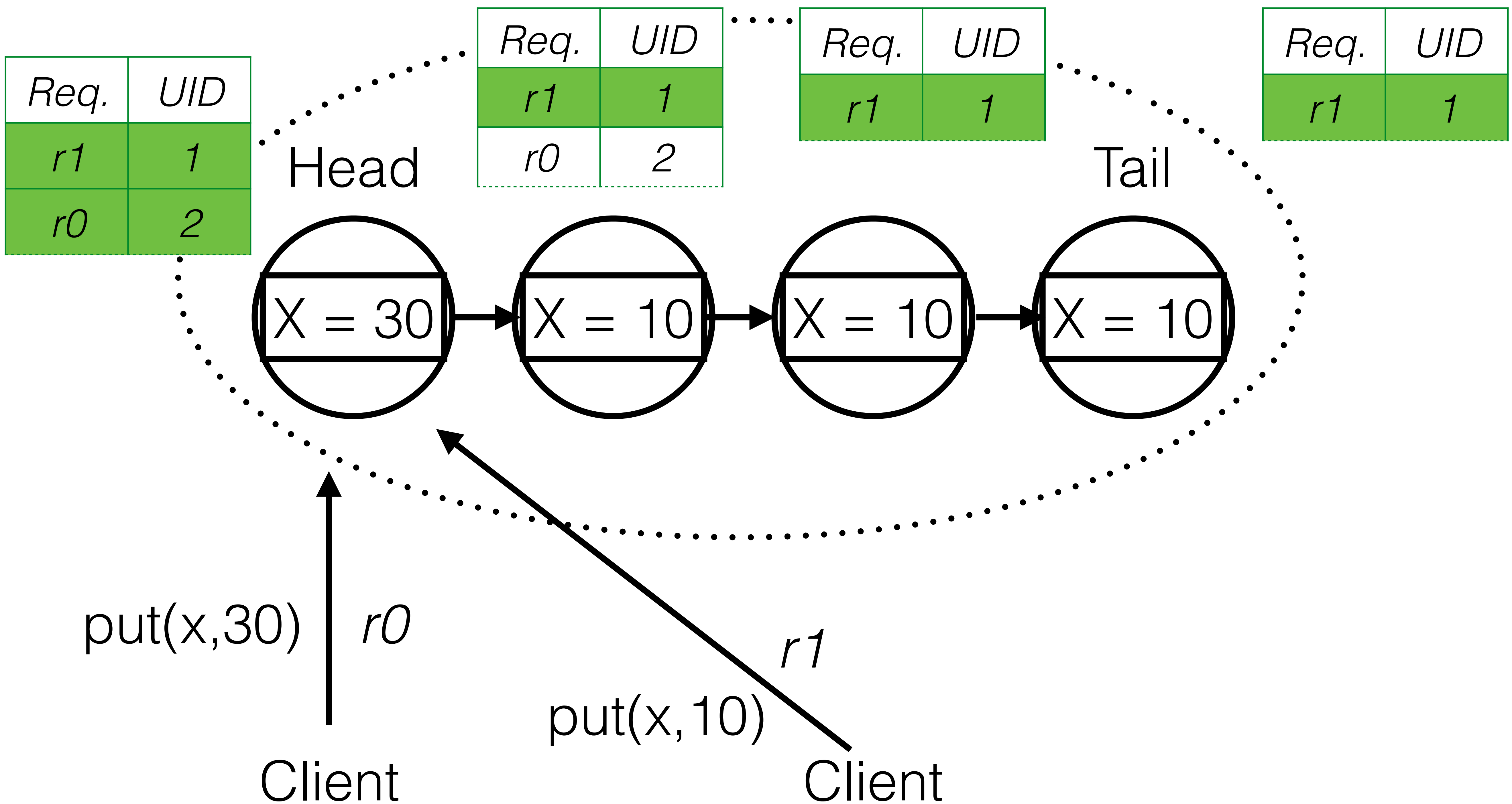
Chain Replication



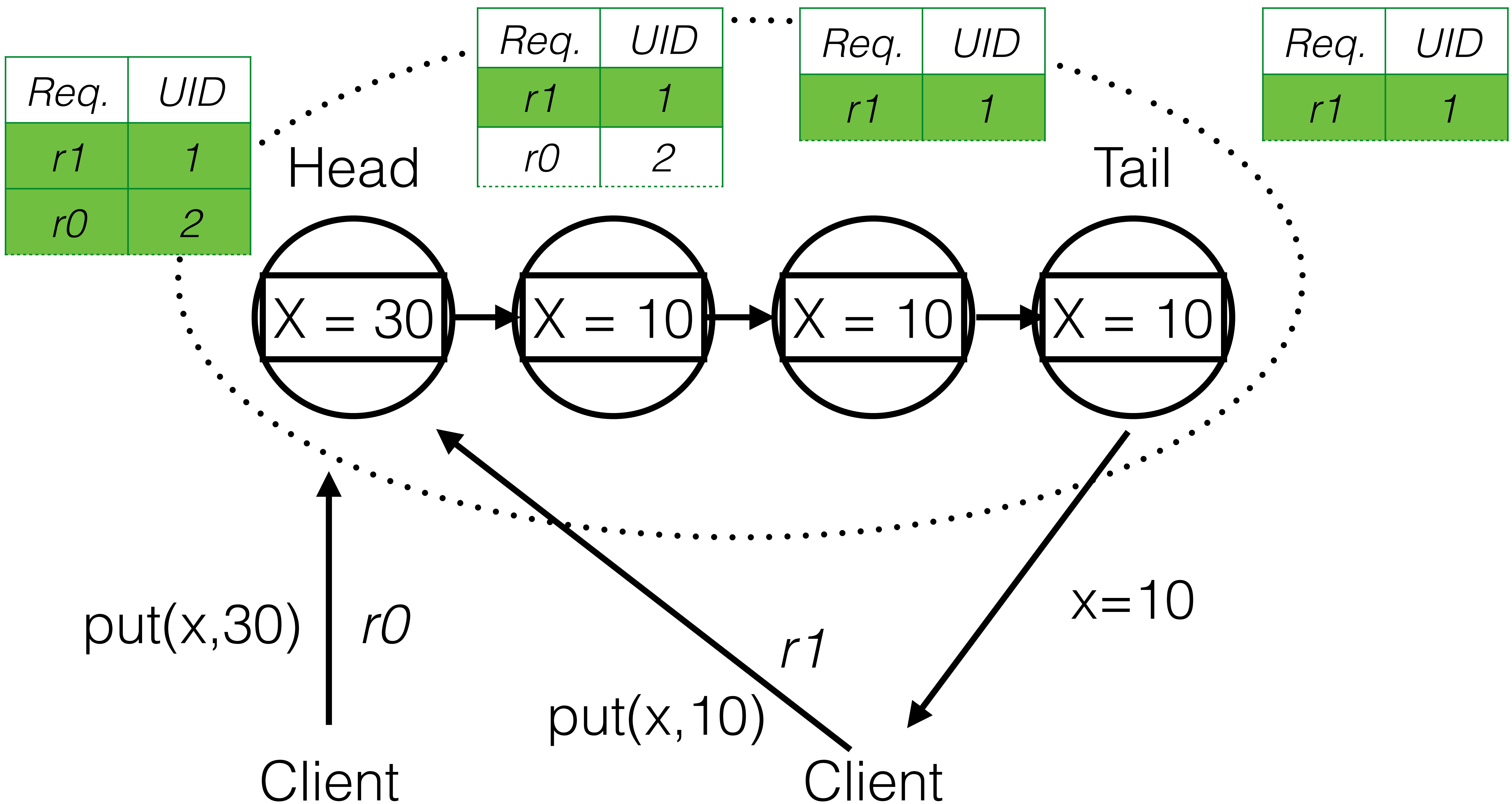
Chain Replication



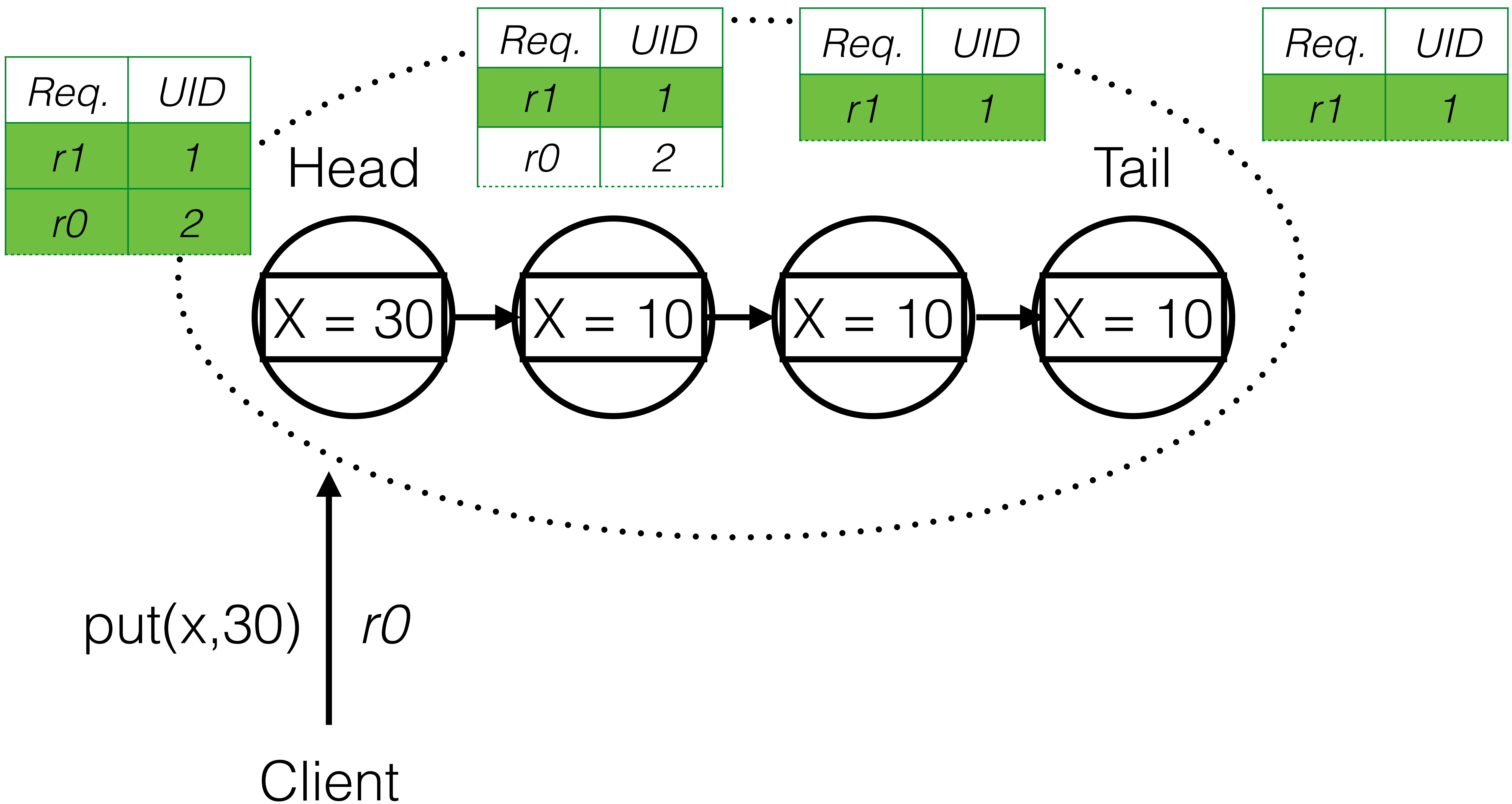
Chain Replication



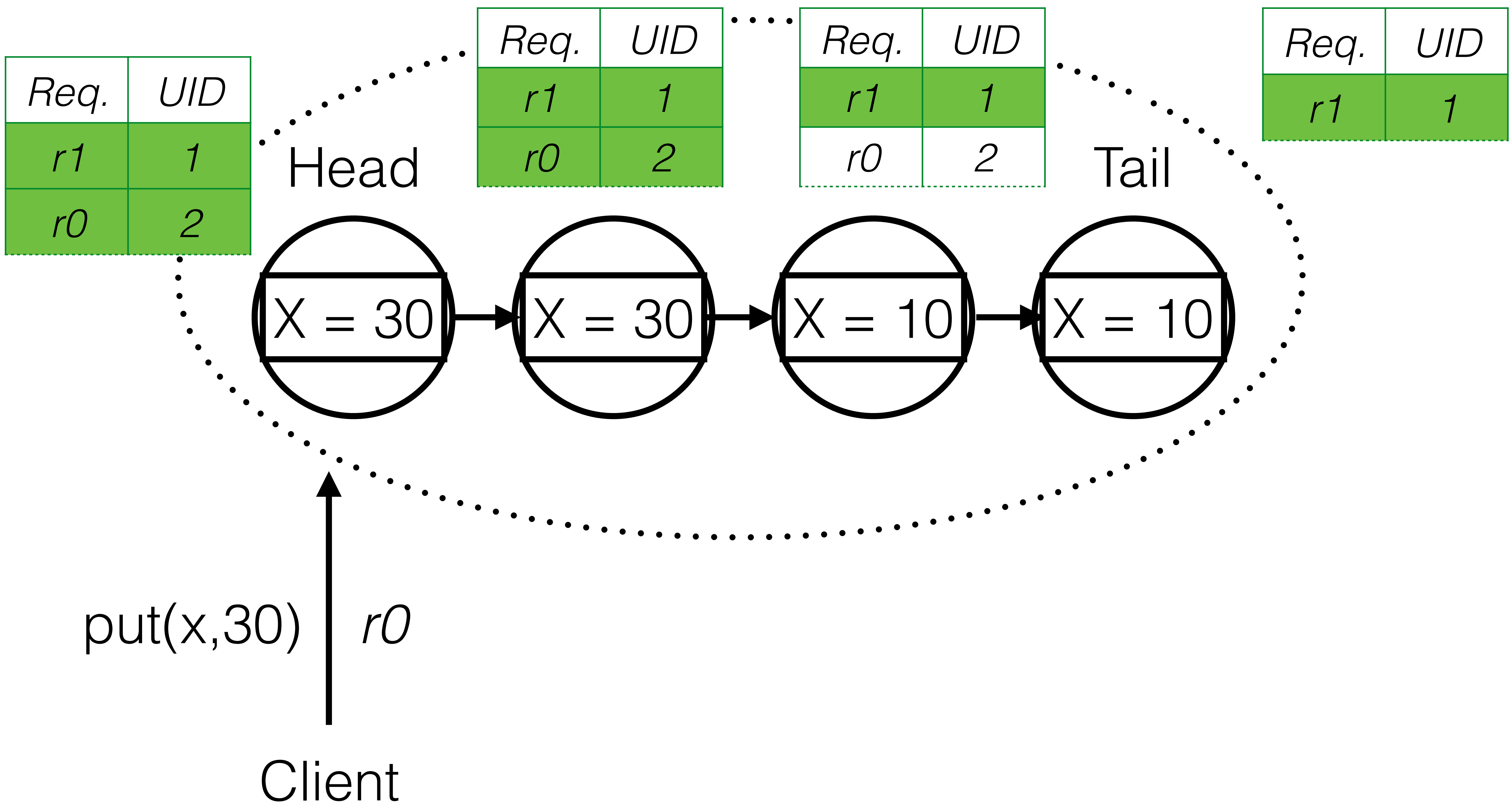
Chain Replication



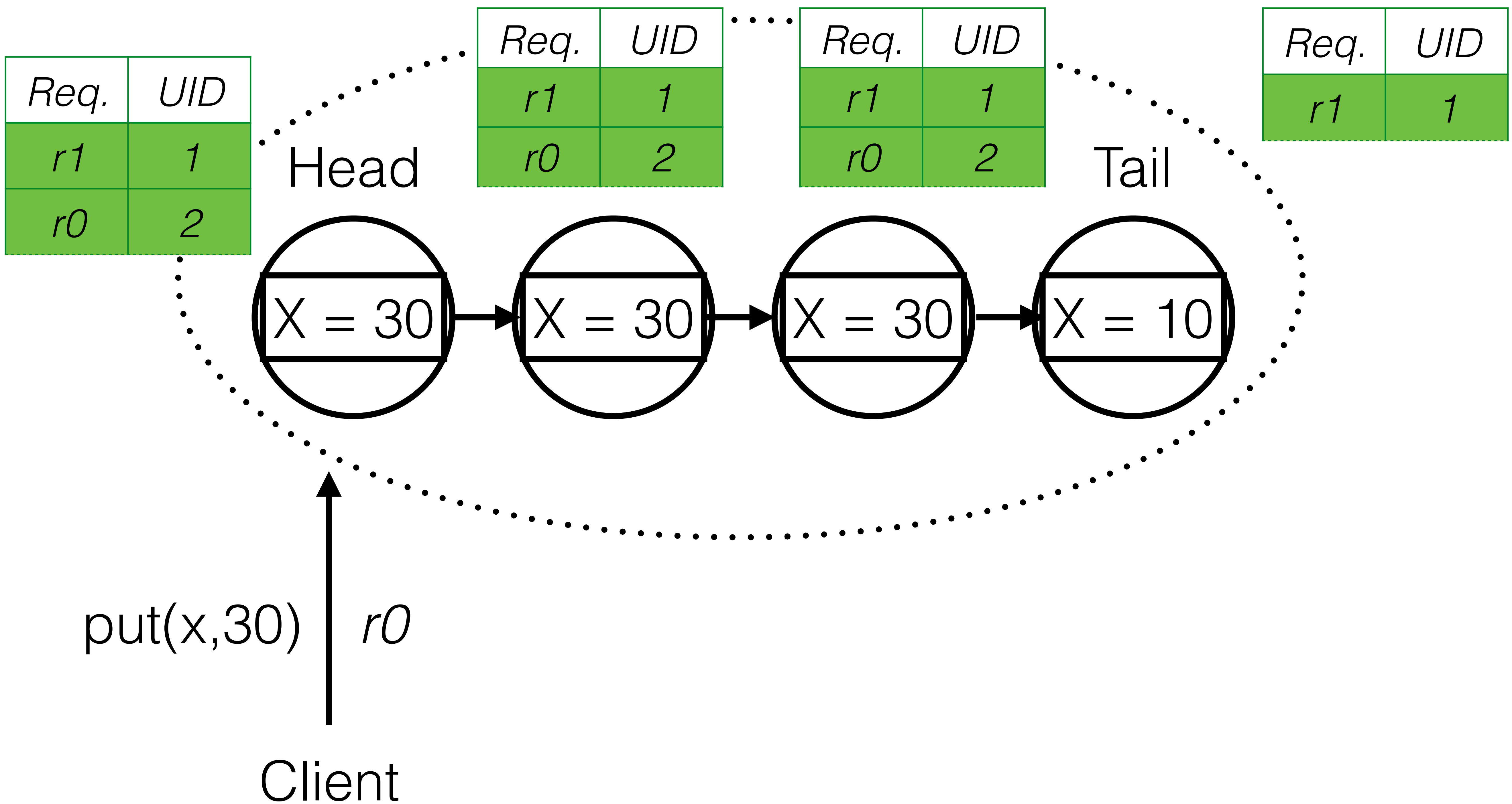
Chain Replication



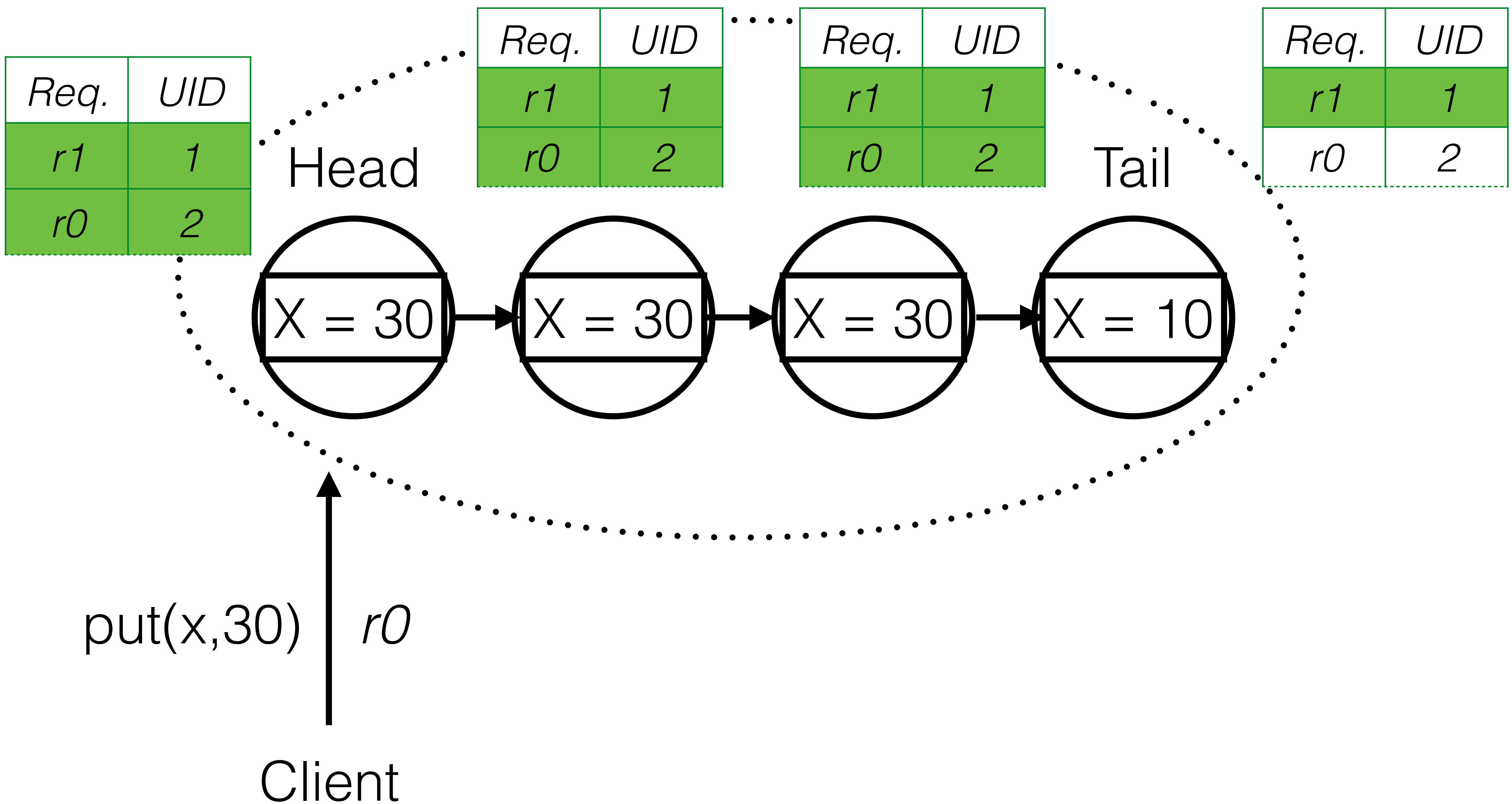
Chain Replication



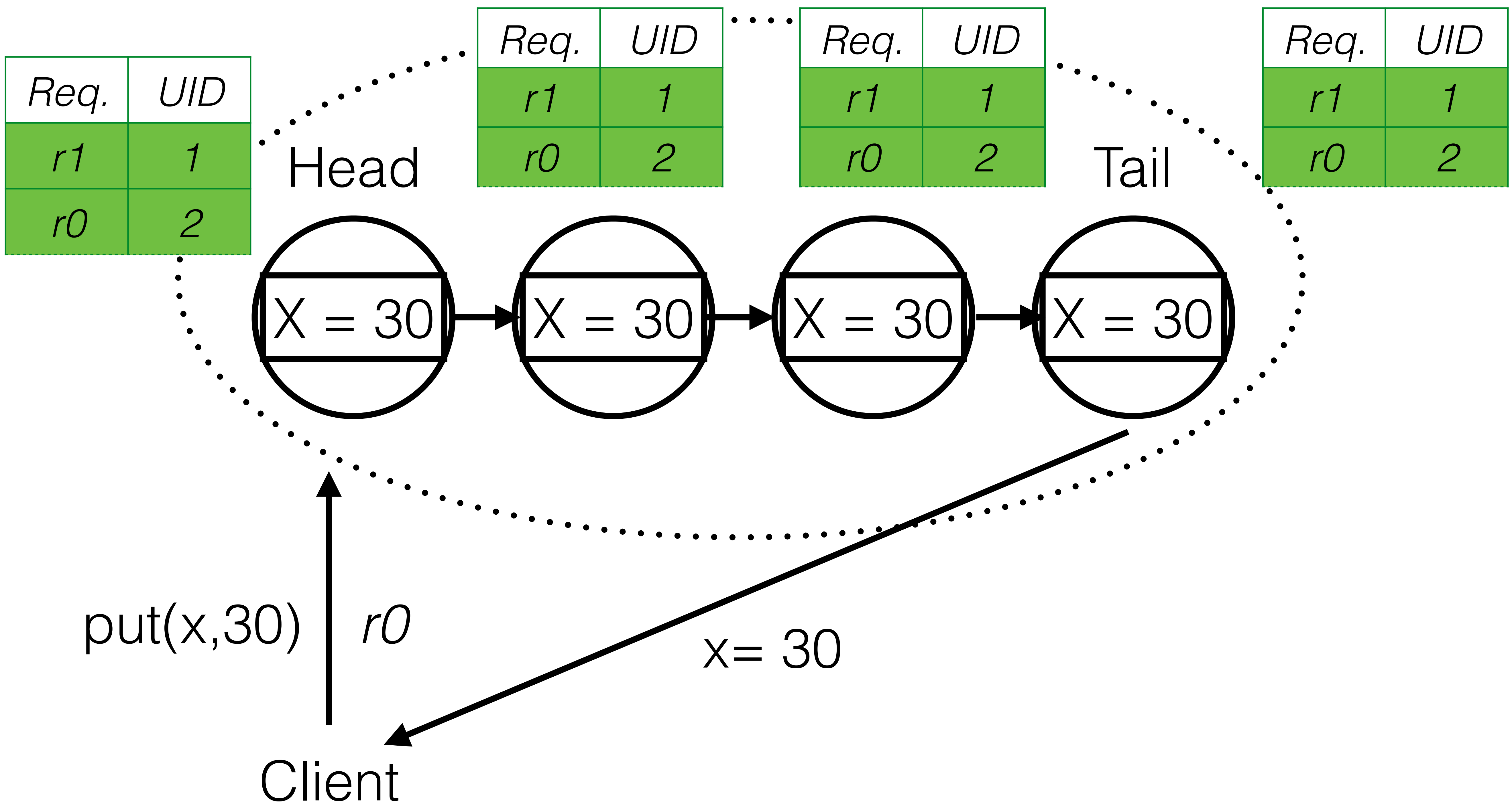
Chain Replication



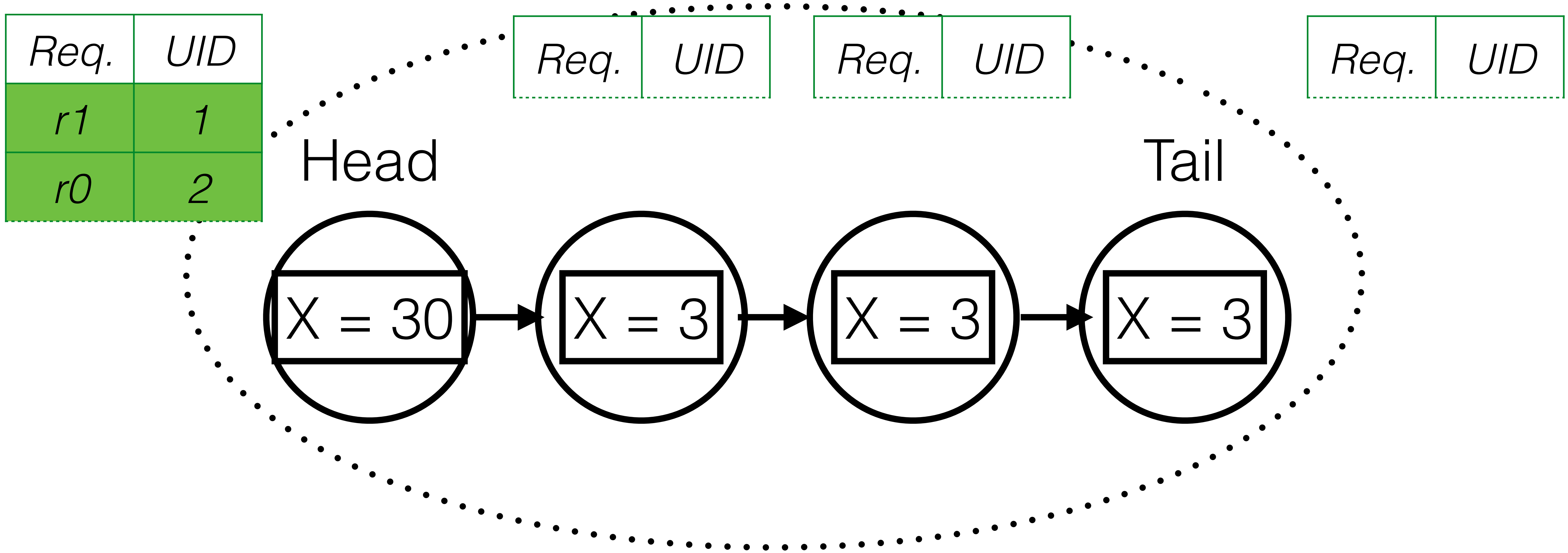
Chain Replication



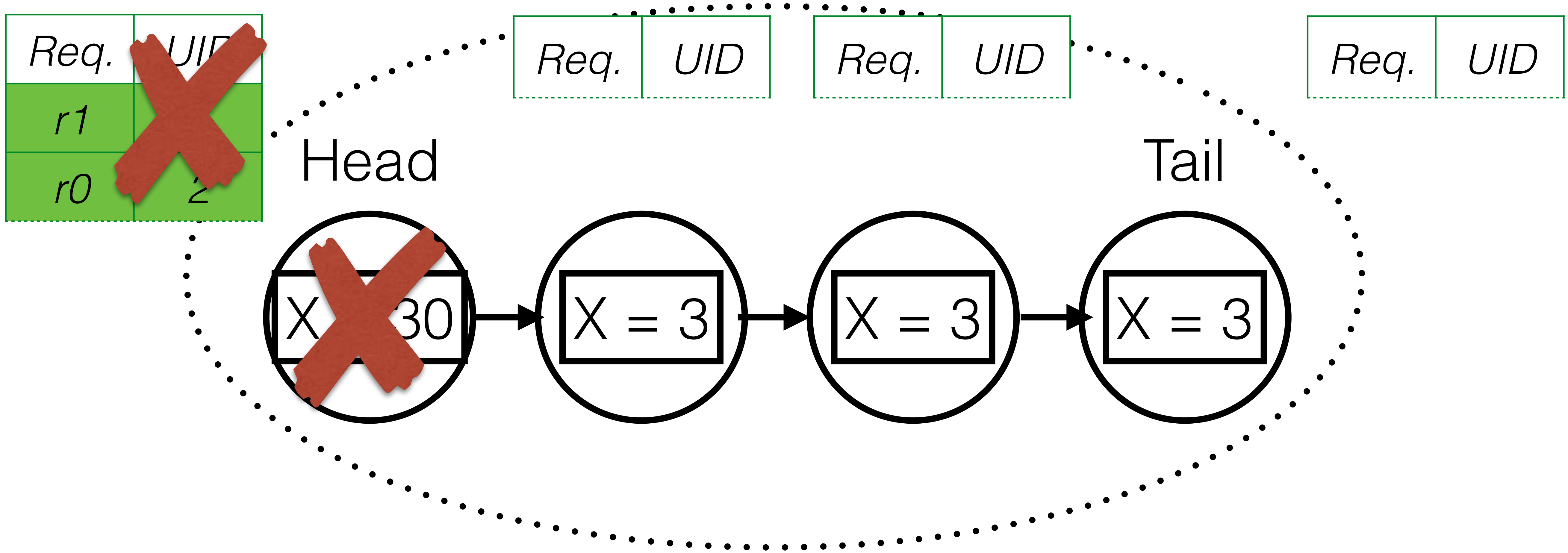
Chain Replication



Fault Tolerance

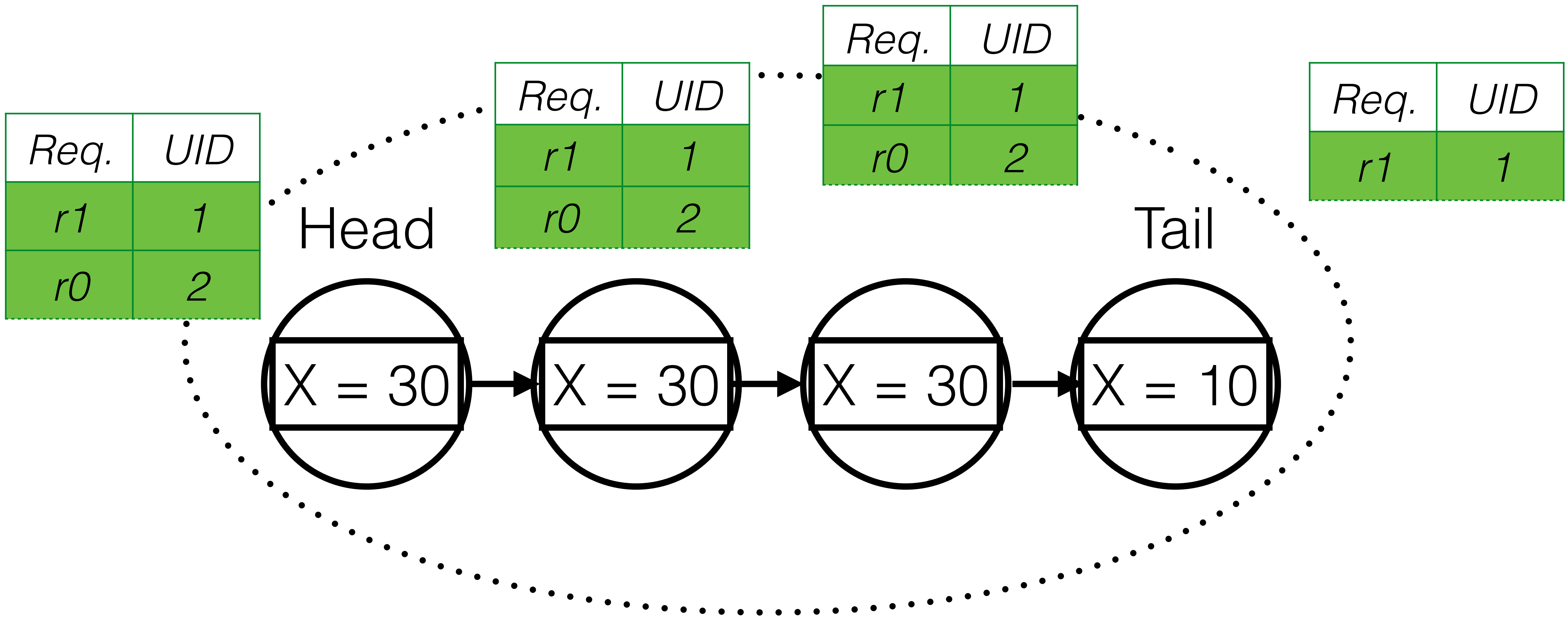


Fault Tolerance

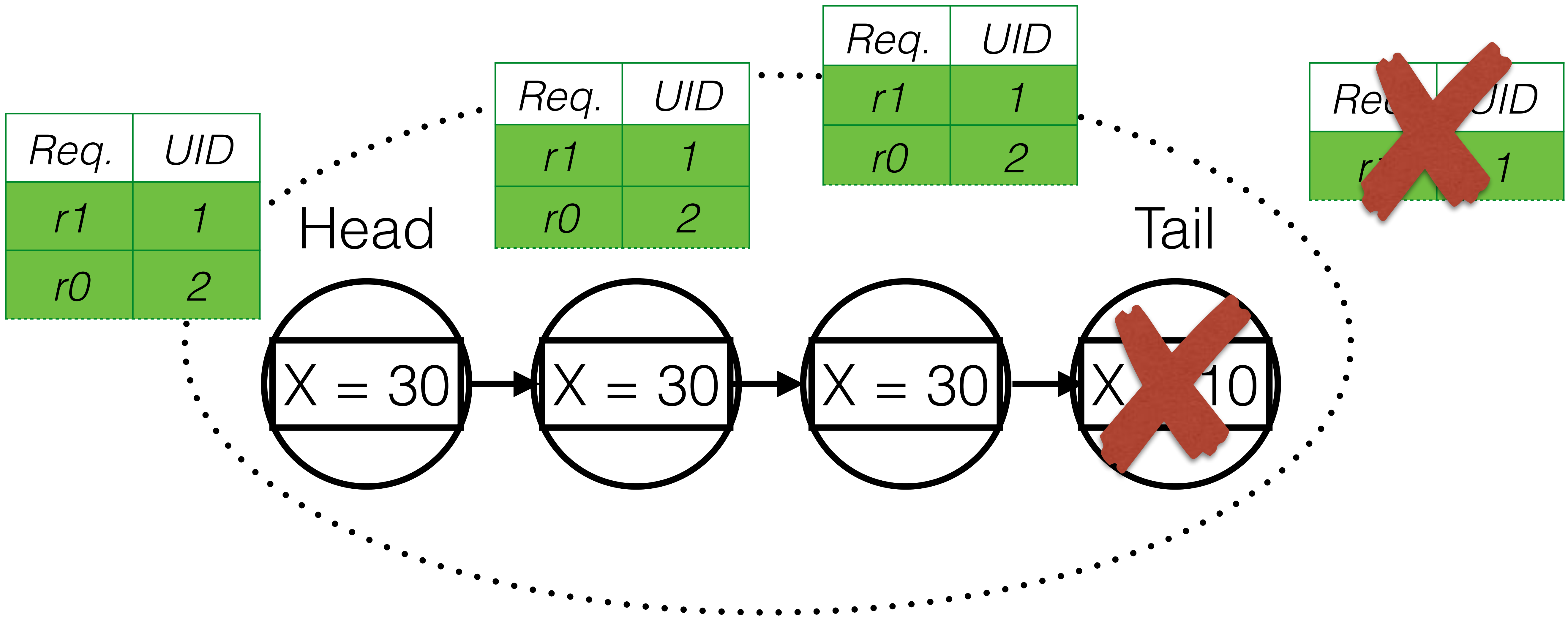


Dropped requests *r1* and *r0*

Fault Tolerance

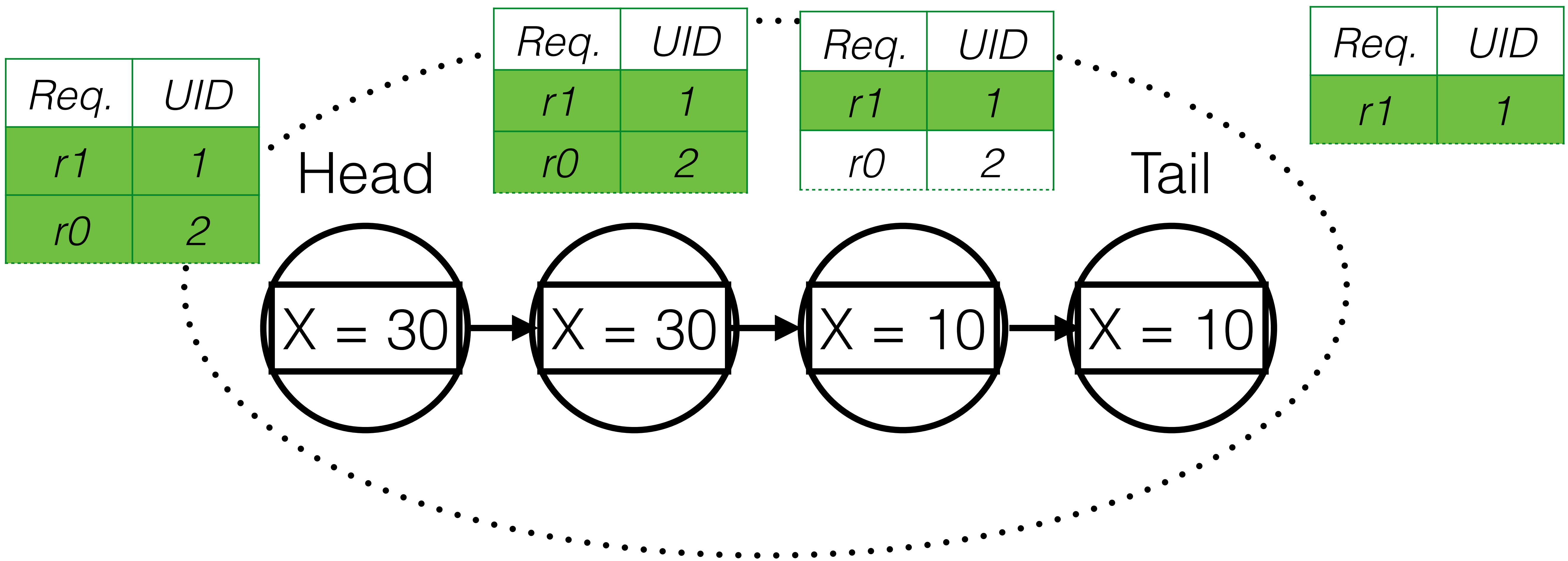


Fault Tolerance

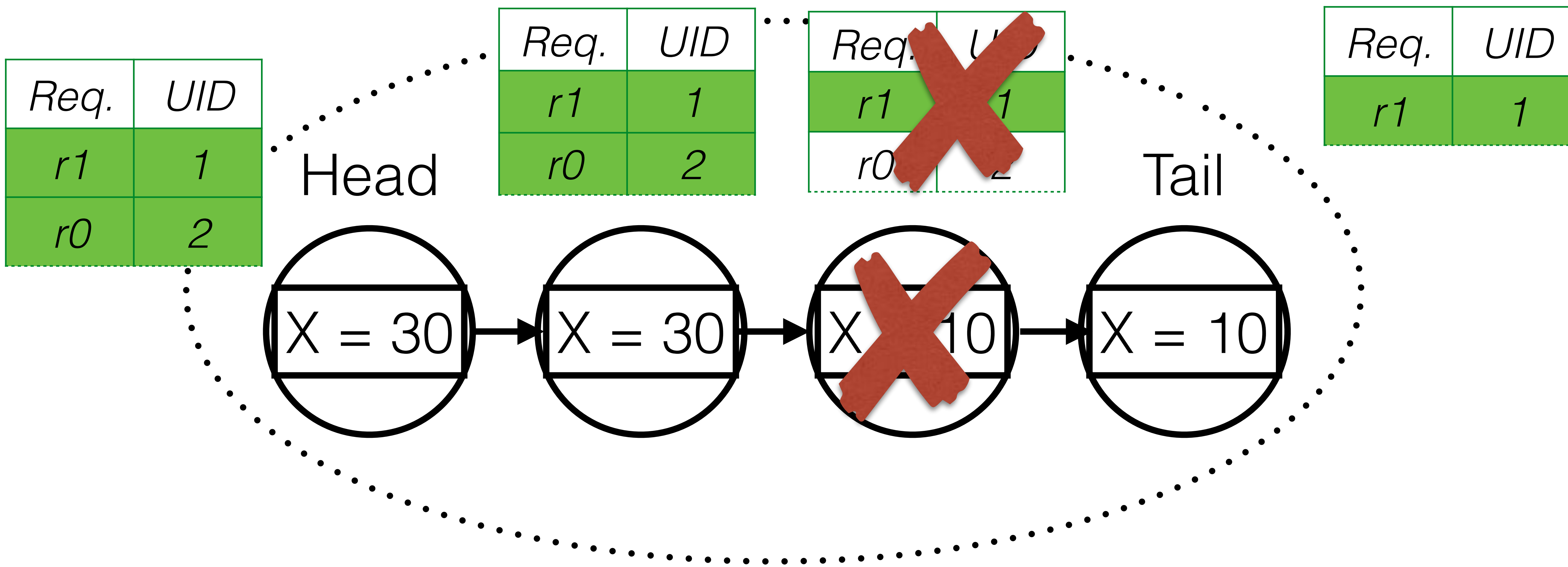


New tail is *stable* for superset of old tail's requests

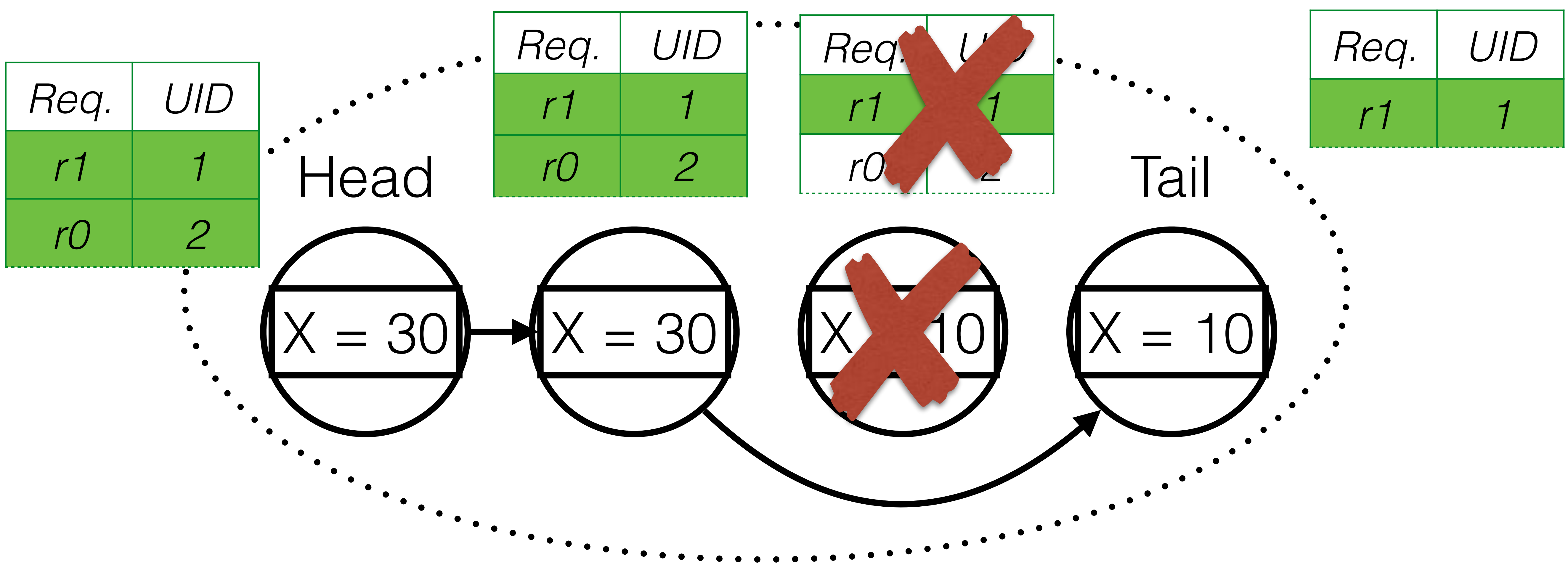
Fault Tolerance



Fault Tolerance

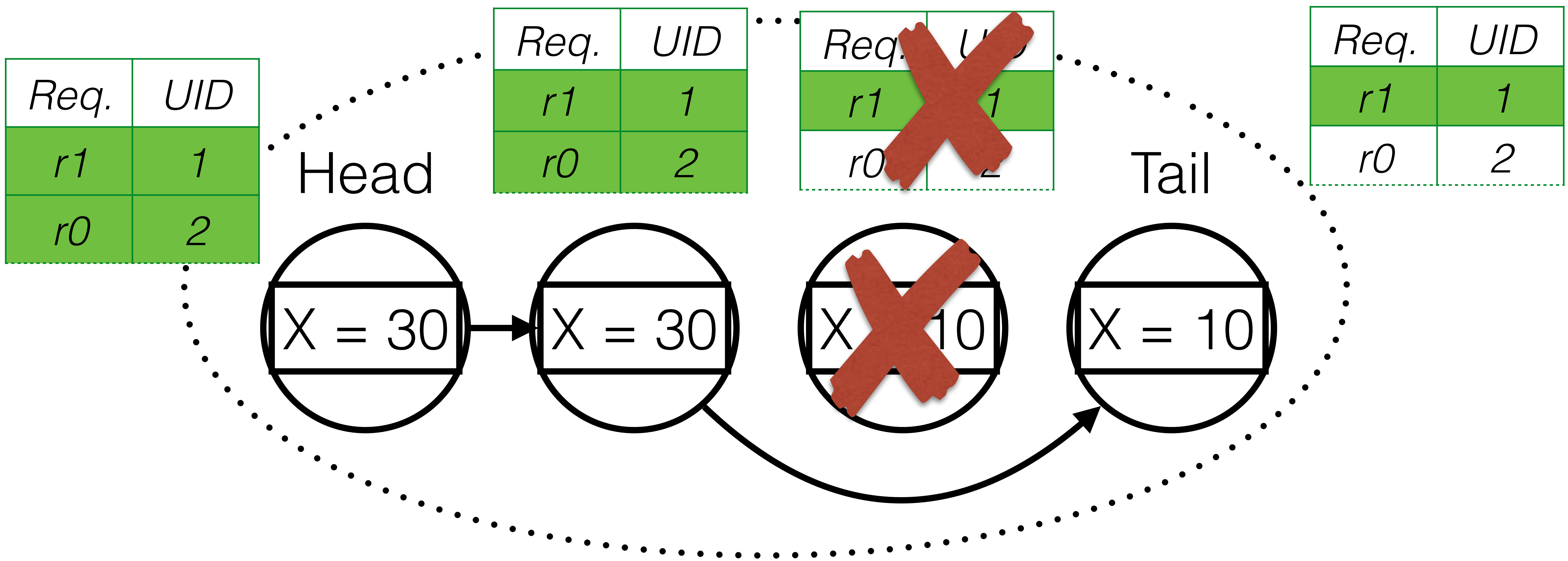


Fault Tolerance



Need to *re-send r0*

Fault Tolerance



Need to *re-send r0*

How is all of this assignment managed?

Chain Replication
Fault Tolerance

- Trusted Master
    - *Fault-tolerant state machine*
    - Trusted by all replicas
    - Monitors all replicas & issues commands

Chain Replication Fault Tolerance

- Failure cases:
  - Head Fails
    - *Master* assigns 2nd node as Head
  - Tail Fails
    - *Master* assigns 2nd to last node as Tail
  - Intermediate Node Fails
    - *Master* coordinates chain link-up

Chain Replication Evaluation

- Compare to other primary/backup protocols
- Tradeoffs?
  - Latency
  - Consistency
- *Trusted Master*

Conclusions

- Implements the “exercise left to the reader” hinted at by Lamport’s paper
- Provides *some* of the concrete details needed to actually implement this idea
    - But still a fair number of details in real implementations that would need to be considered
    - Chain replication illustrates a “simple” example with fully concrete details
- Does some work to justify why such synchronization might be useful (plane actuators)
- A key contribution that bridges the gap between academia and practicality for SMR