# Lecture 24: Distributed Systems

CS 105

December 10, 2019

# What is a distributed system?

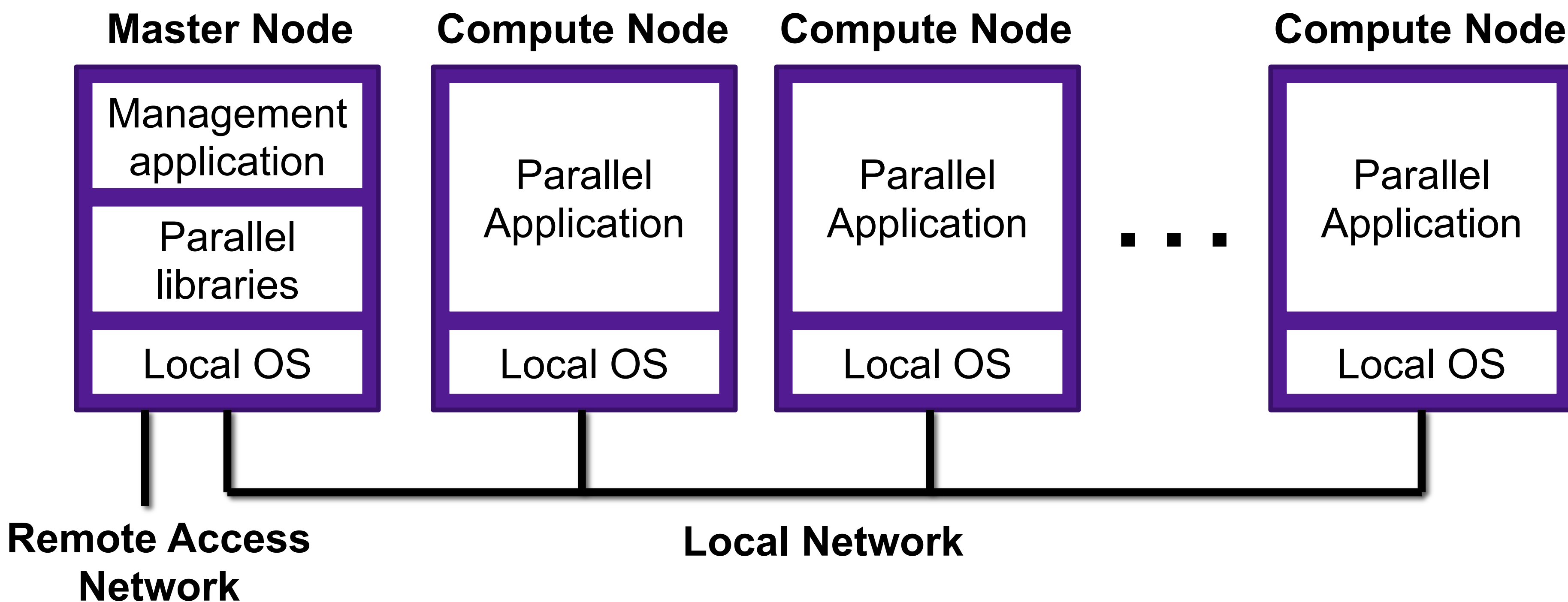
- A **distributed system** is a collection of autonomous computing elements that appears to its users as a single, coherent system

- A **distributed system** is several computers doing something together. Thus, a distributed system has three primary characteristics: multiple computers, interconnections, and shared state.

# Why not just use one computer?

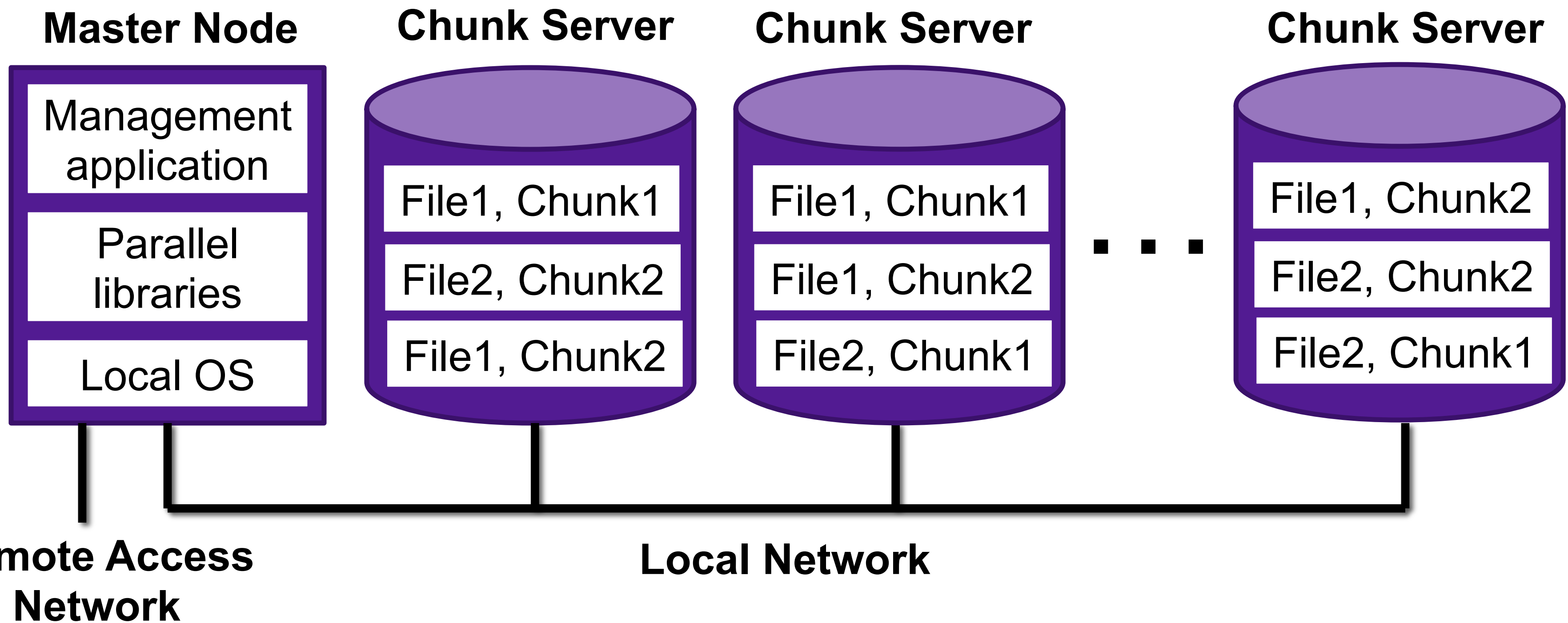
- computers fail
- limited resources
- physical location
- nonuniform hardware

# Example: Cluster Computing




- cluster computing is use for parallel programming in which a single (compute intensive) program is run in parallel on multiple machines
- Master node provides interface for users and is responsible for task scheduling
- Examples: MOSIX, MapReduce, Hadoop

# Example: Distributed File Systems




- files are divided into fixed-size chunks and stored on chuck servers
- each chunk is replicated
- master server stores chunk metadata

# Properties we want

- **Transparency:** Hide that resource is physically distributed across multiple computers
- **Consistency:** appears as all one system
- **Reliability:** system doesn't go down/go wrong when component(s) fail
- **Scalability:** can grow (add more nodes, memory, etc.)

# Replication

- In distributed systems, data are typically replicated
  - increases reliability
  - improves scalability
  - reduces latency for global systems
- Problem: how do you ensure consistency when data are replicated?

# The Model

- Shared data is kept in a data store
  - a register, a file system, a database, a distributed file system, a distributed key-value store
- Clients access the data store through read and write operations
- Consistency Semantics: a contract between the data store and its clients that specifies the results that clients can expect to obtain when accessing the data store
  - sequential consistency
  - causal consistency
  - eventual consistency

# Sequential Consistency

- “The result of any execution is the same as if the operations of all the processes were executed in some sequential order and the operations of each individual process appear in this sequence in the order specified by its program” (Lamport, 1979)

- In other words: create a total order that includes all the operations of the execution, such that:
  - the total order respects the local history of each process
  - every read returns the result of the latest write, according to the total order (data coherence)

# Example: Sequential Consistency

| | | | | | |
|---|---|---|---|---|---|
| P1: | **4** W(x): a | | | | |
| P2: | | **1** W(x): b | | | |
| P3: | | | **2** R(x): b | | **5** R(x): a |
| P4: | | | | **3** R(x): b | **6** R(x): a |

- Is this data store sequentially consistent?

# Example: Sequential Consistency

| | | | | | |
|---|---|---|---|---|---|
| P1: | W(x): a | | | | |
| P2: | | W(x): b | | | |
| P3: | | | R(x): b | | R(x): a |
| P4: | | | | R(x): a | R(x): b |

- Is this data store sequentially consistent?

# Exercise: Sequential Consistency

| | | | | | |
|---|---|---|---|---|---|
| P1: | W(x): a | | | W(x): c | |
| P2: | | R(x): a | W(x): b | | |
| P3: | | R(x): a | | R(x): c | R(x): b |
| P4: | | R(x): a | | R(x): b | R(x): c |

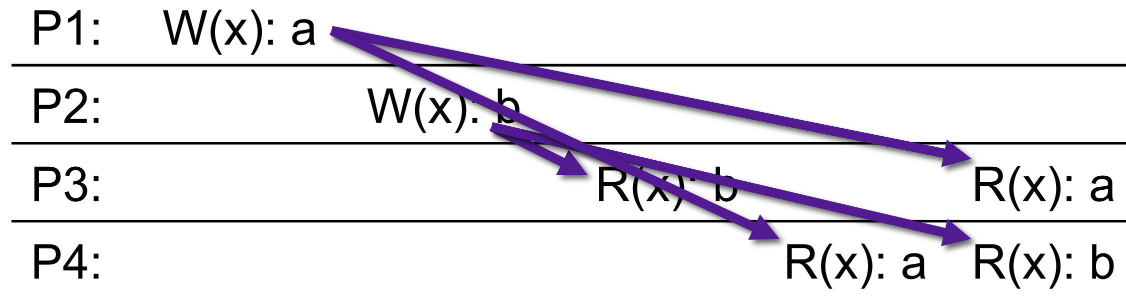
- Is this data store sequentially consistent?

# Causal Consistency

- Writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines. (Hutto and Ahamad, 1990)

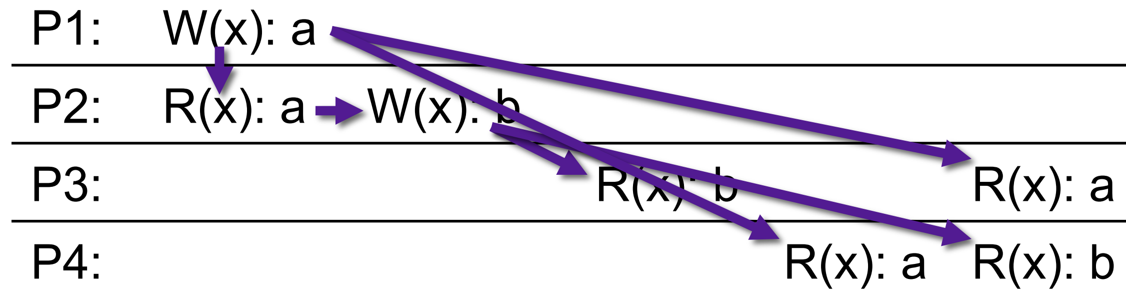
- The following pairs of operations are causally related:
  - Two writes by the same process to the same location
  - A read followed by a write of the same process (even if the write addresses a different location)
  - A read that returns the value of a write from any process
  - Two operations that are transitively related according to the above conditions

# Example: Causal Consistency



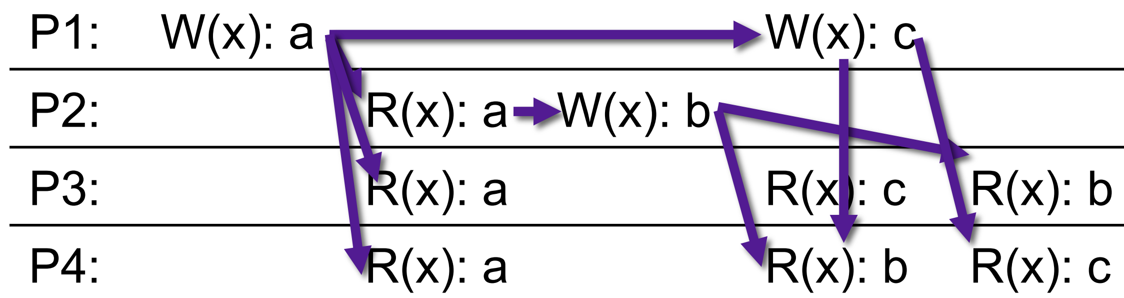

- Is this data store causally consistent?

# Example: Causal Consistency




- Is this data store causally consistent?

# Exercise: Causal Consistency




- Is this data store causally consistent?

# Eventual Consistency

- If no updates take place for a long time, all replicas will eventually have the same data stored (Vogels, 2009)

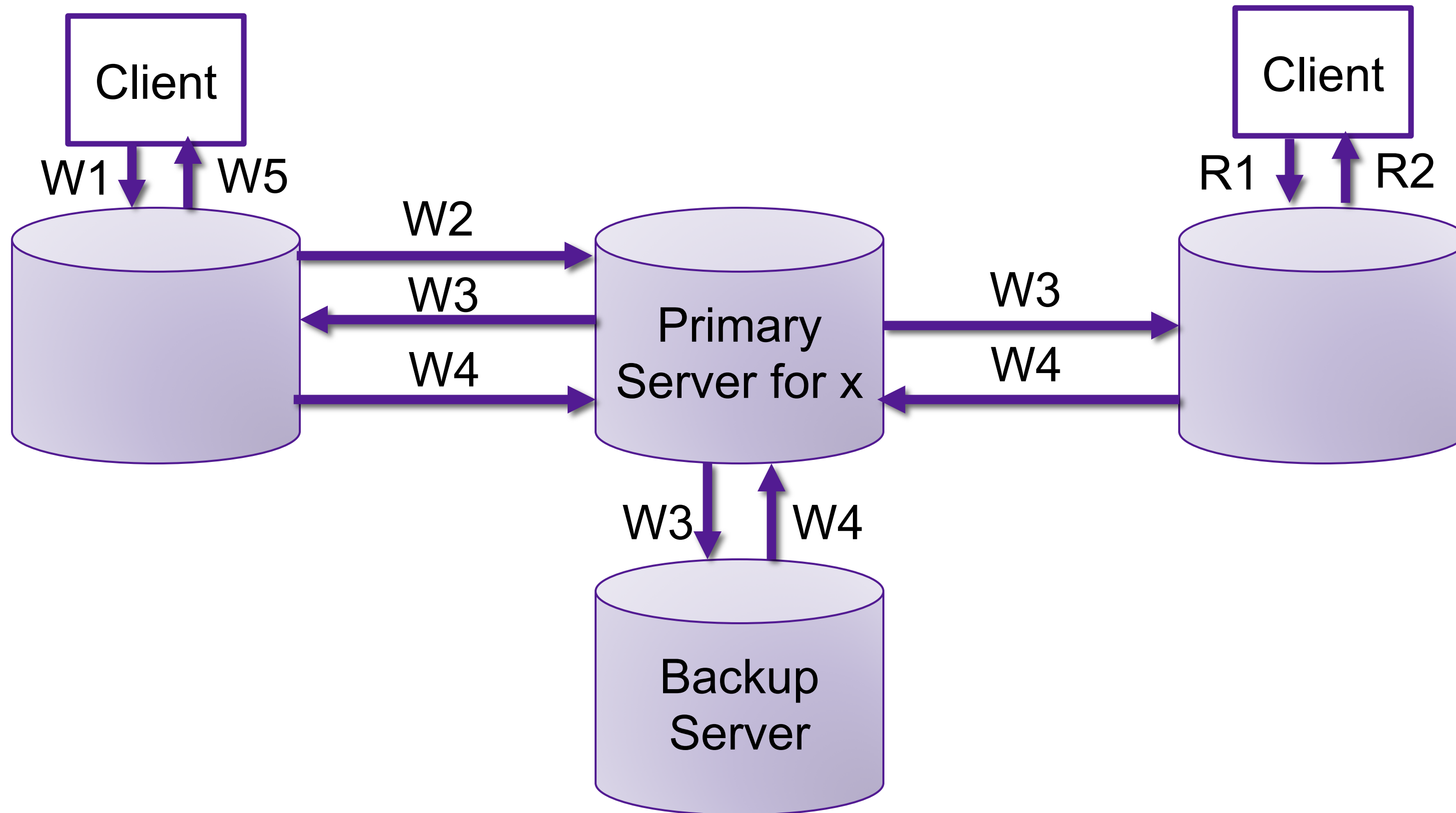
- In the absence of write-write conflicts, updates will eventually propagate to all replicas
- Convenient model for applications where updates are made by a single authority (e.g., web with caching)

# What semantics do you want

- want a consistency model that is easy to understand (otherwise it is hard to write correct programs that use the data store)

- need to be able to guarantee the consistency model

# Primary-based protocols



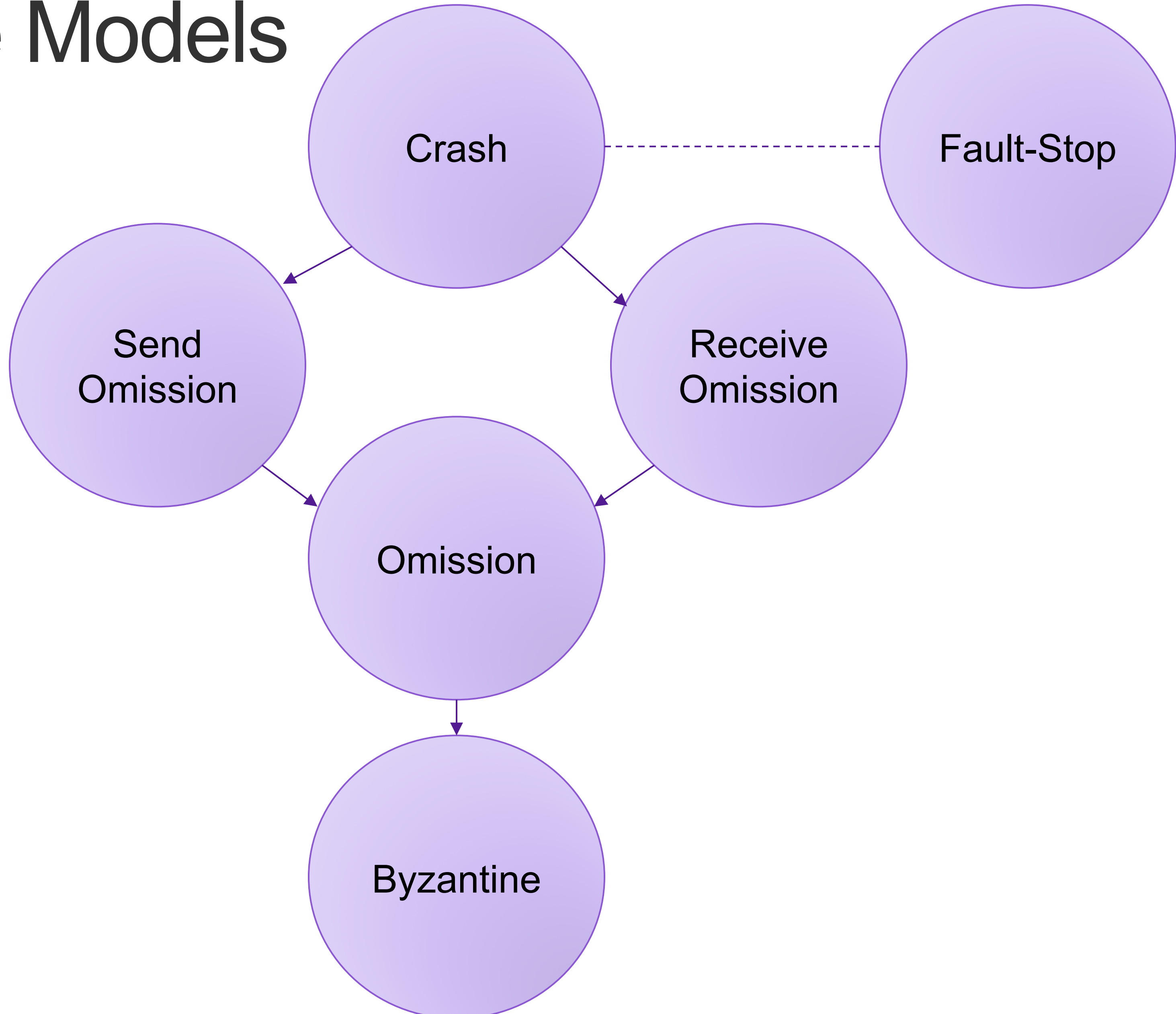

- straightforward implementation of sequential consistency
  - primary server orders writes
- performance vs. fault tolerance tradeoff

# Replicated-write protocols

- General idea: require clients to request and acquire the permission of multiple servers before either reading or writing a replicated data item

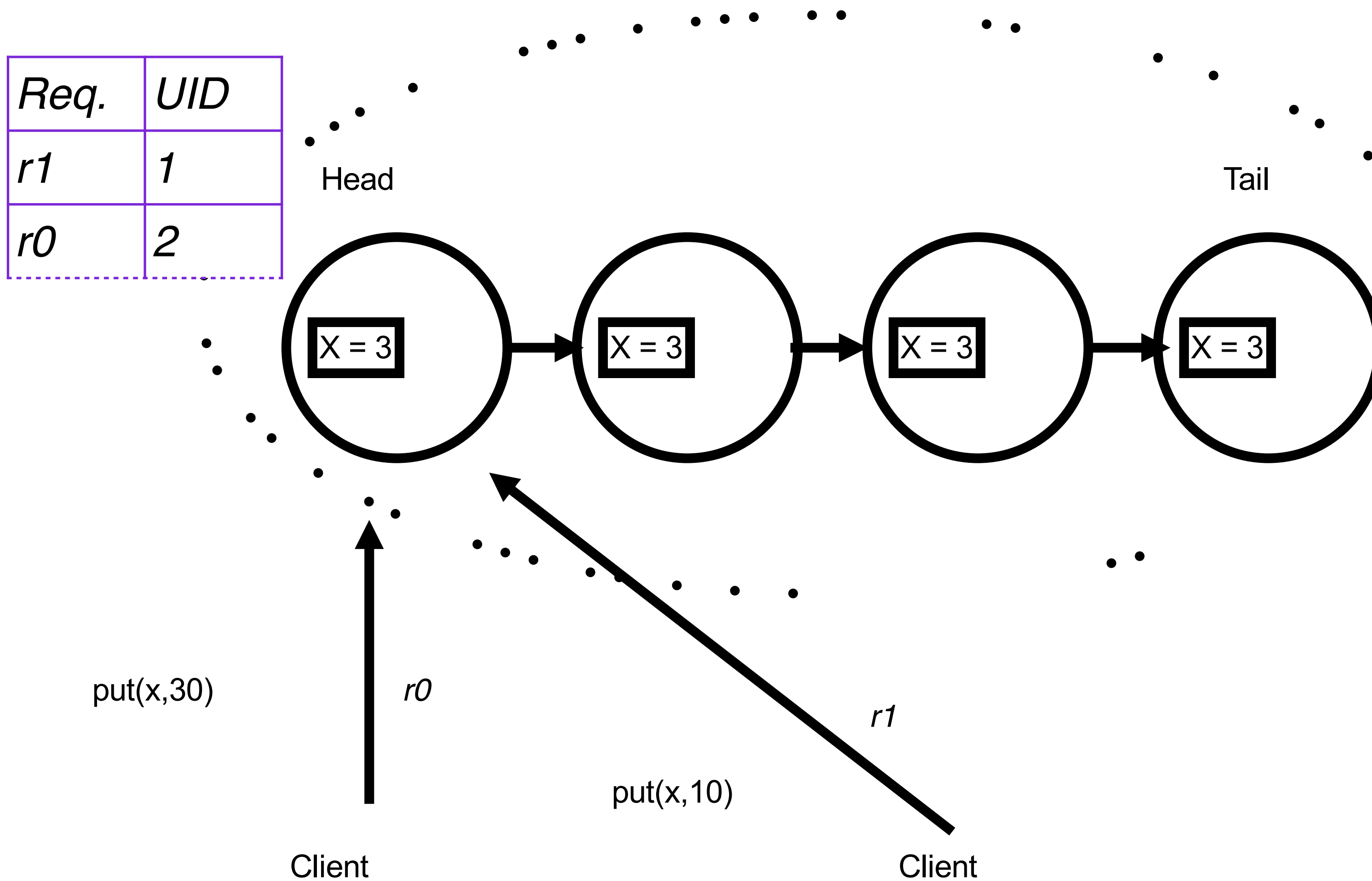
- To write a replicated item:
  - client must first contact a majority of servers and get them to agree to the update
  - once majority of servers have agreed, file is changed and a new version number is associated with
- To read a replicated file:
  - client must obtain same version of file from a majority of servers

# Failure Models

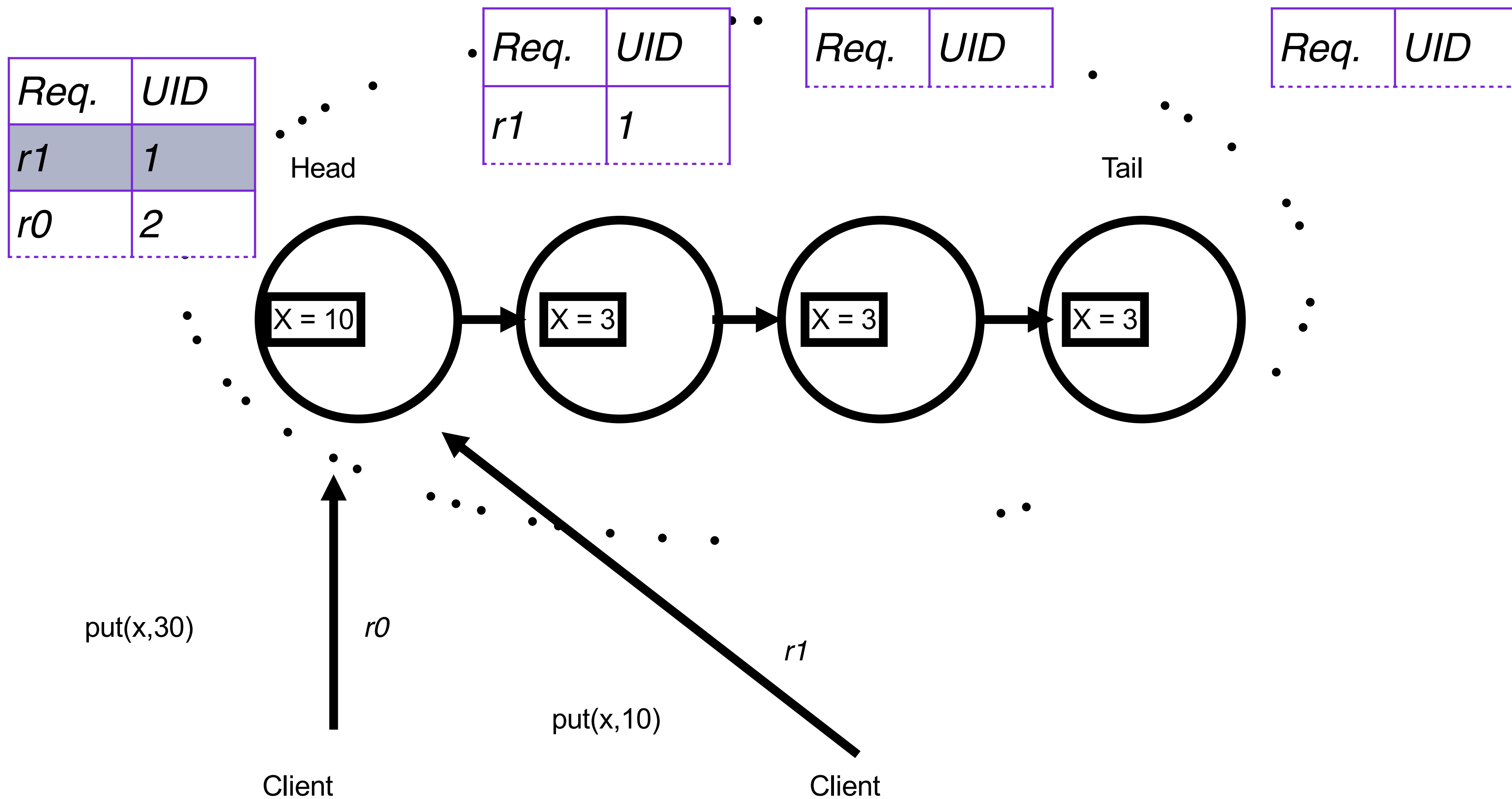


# Chain Replication

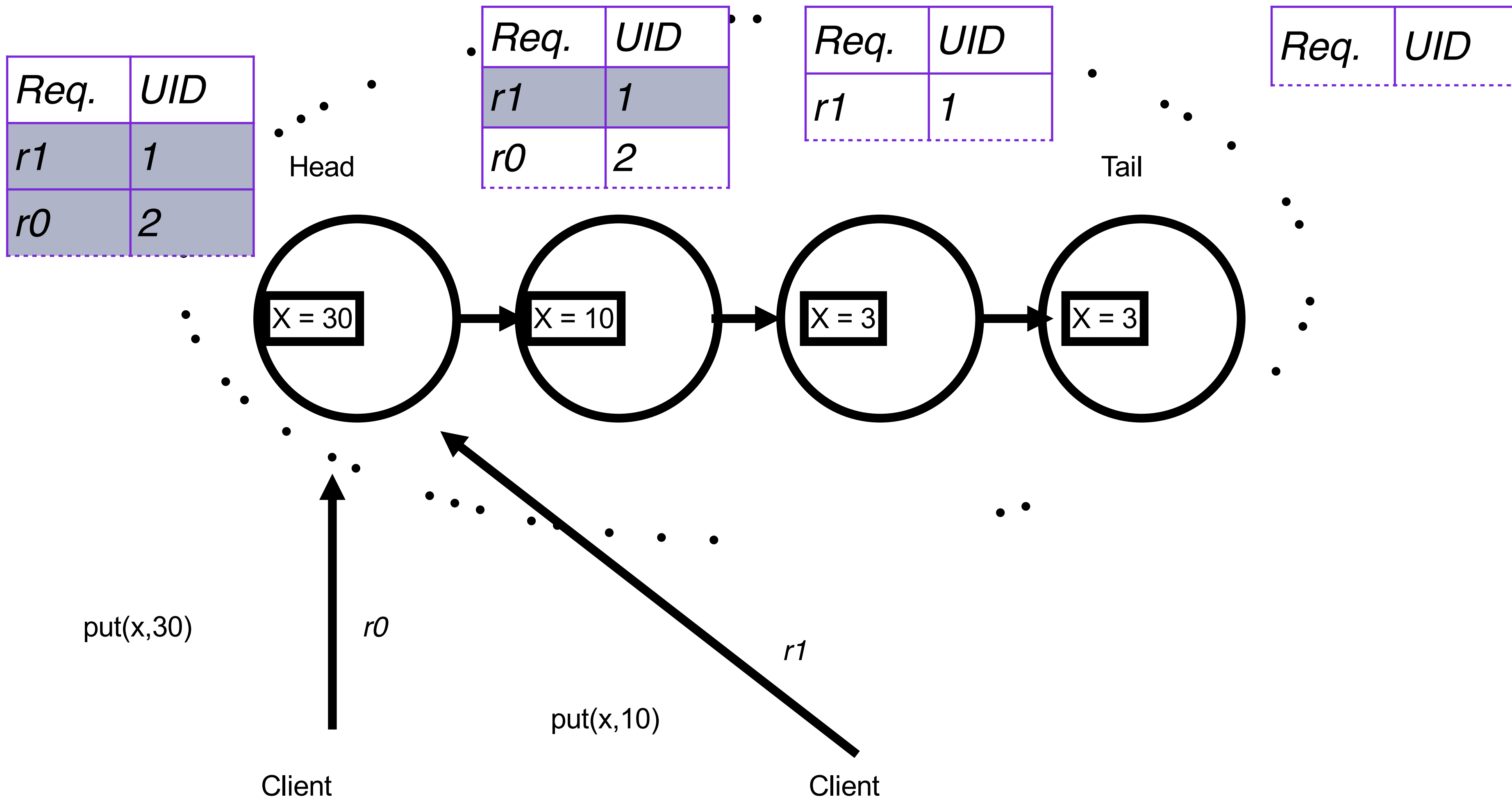
| *Req.* | *UID* |
| --- | --- |
| *r1* | *1* |
| *r0* | *2* |



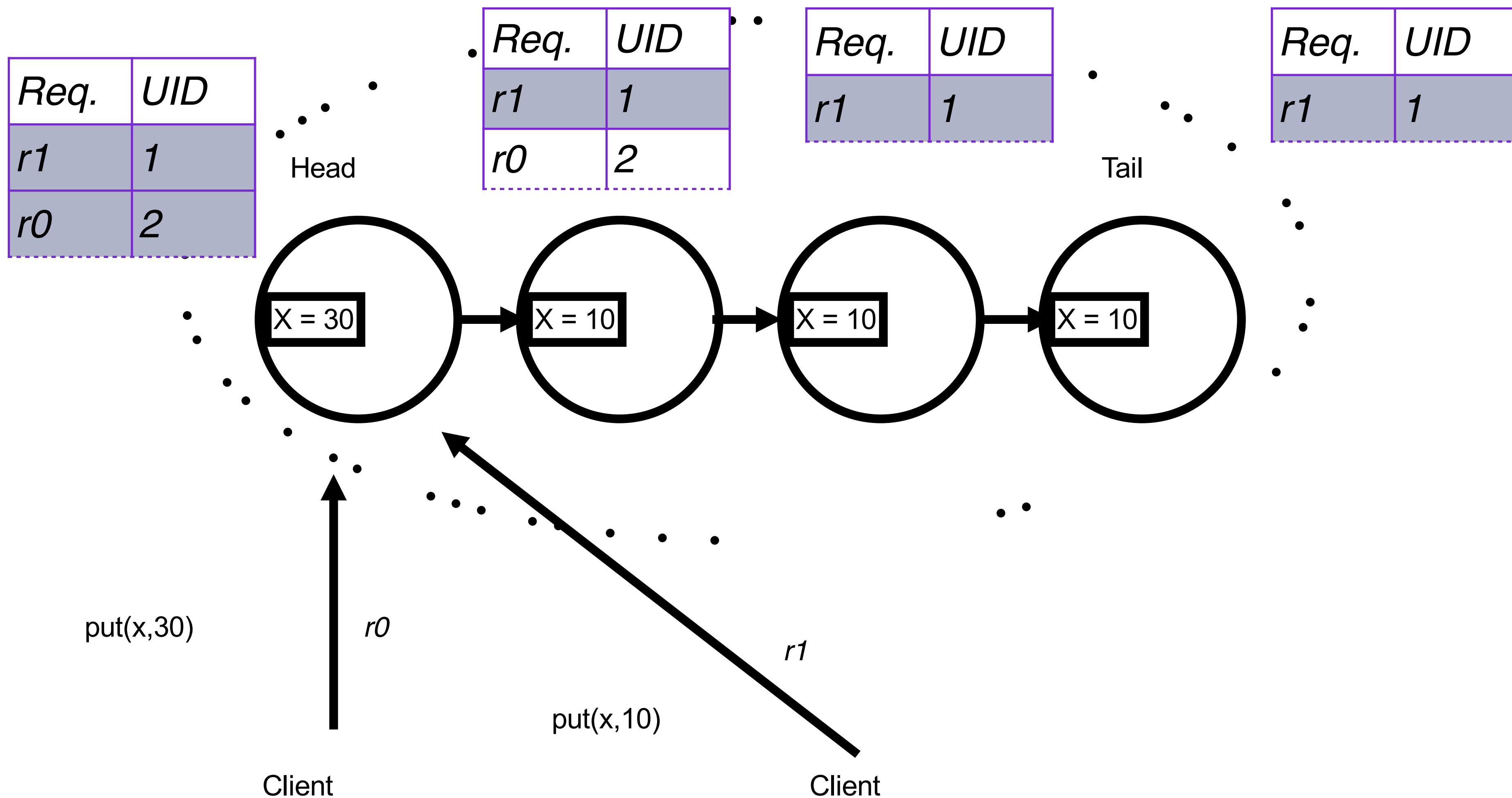
# Chain Replication



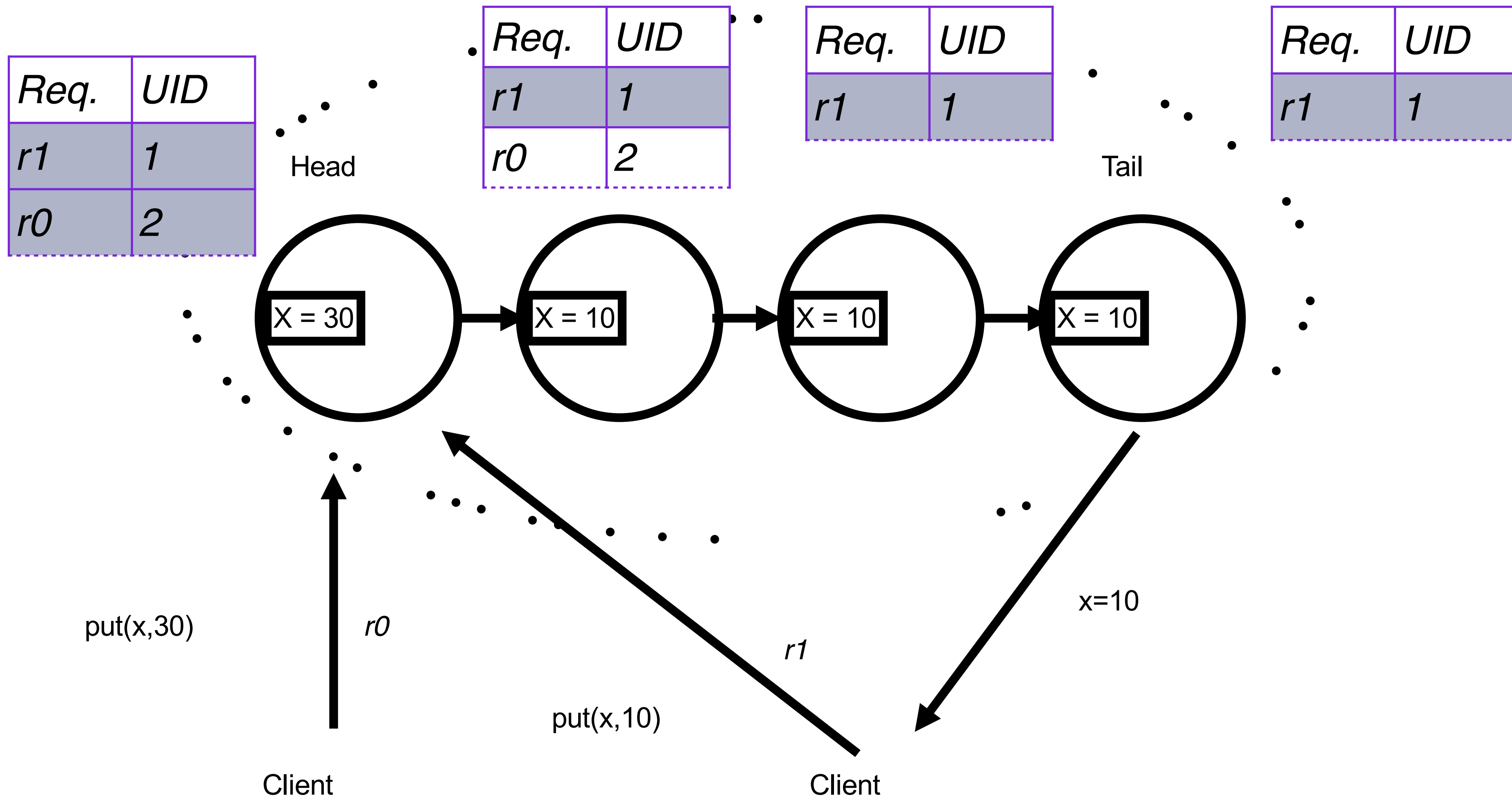
# Chain Replication



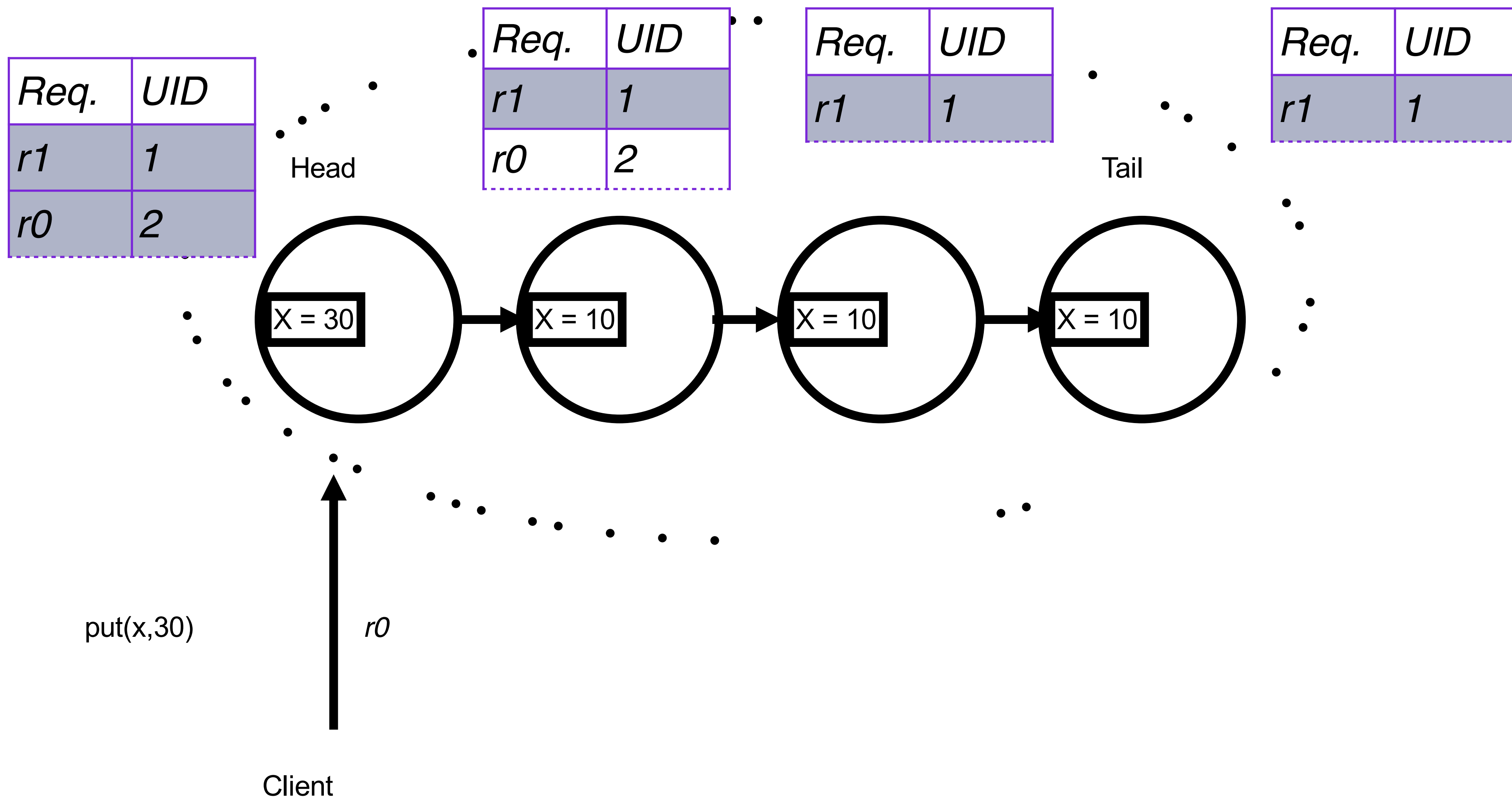
# Chain Replication



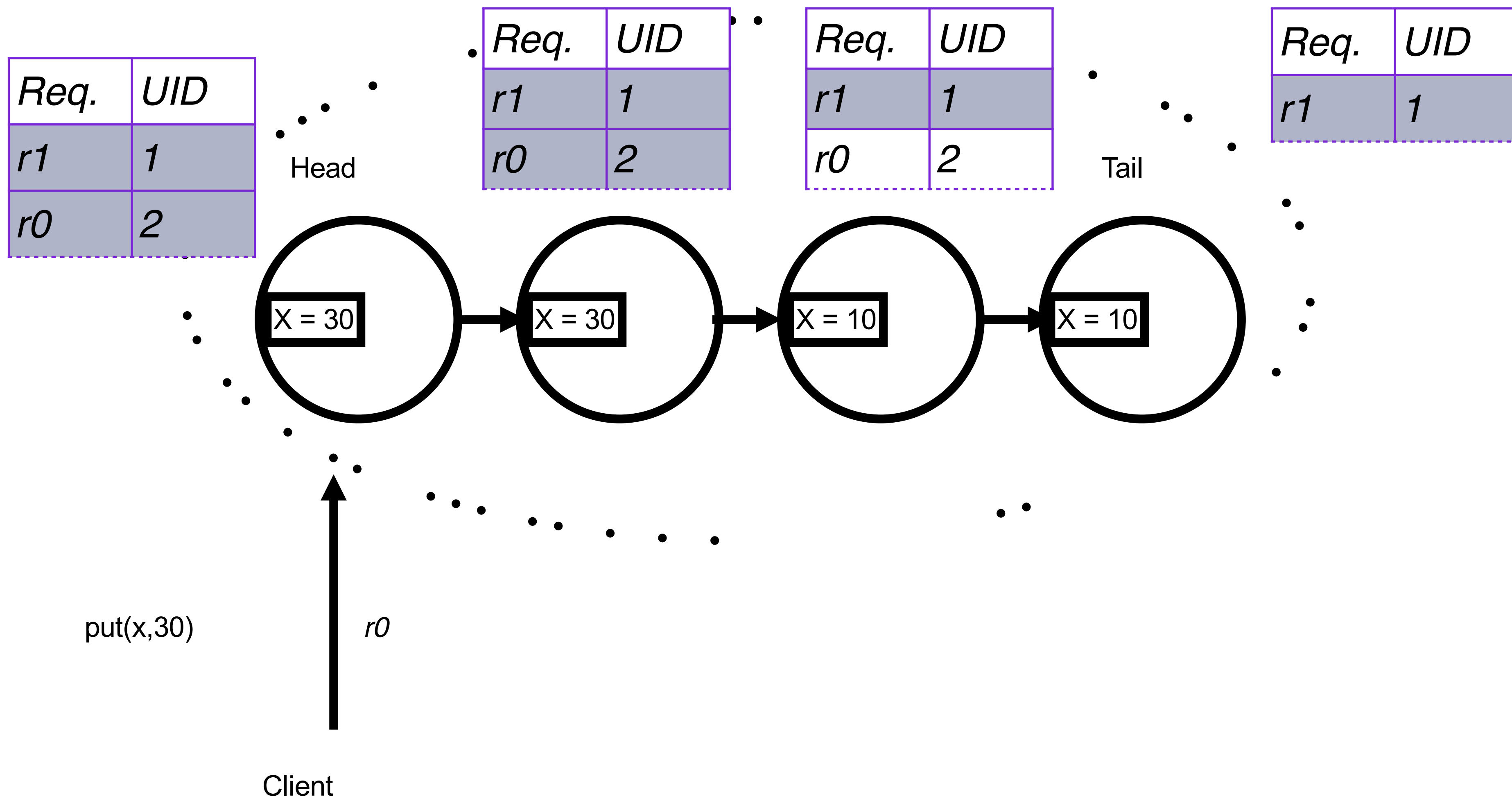
# Chain Replication



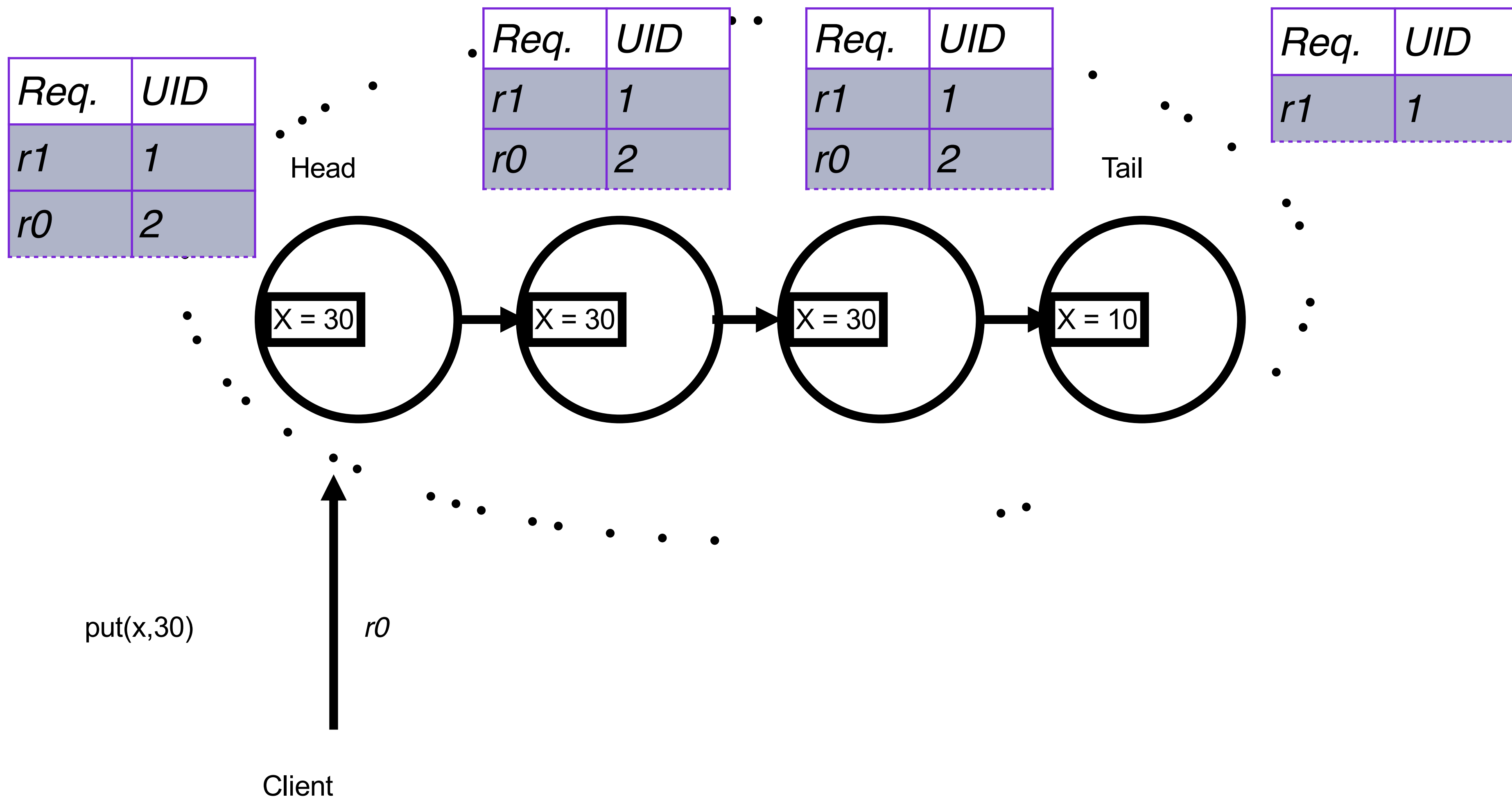
# Chain Replication



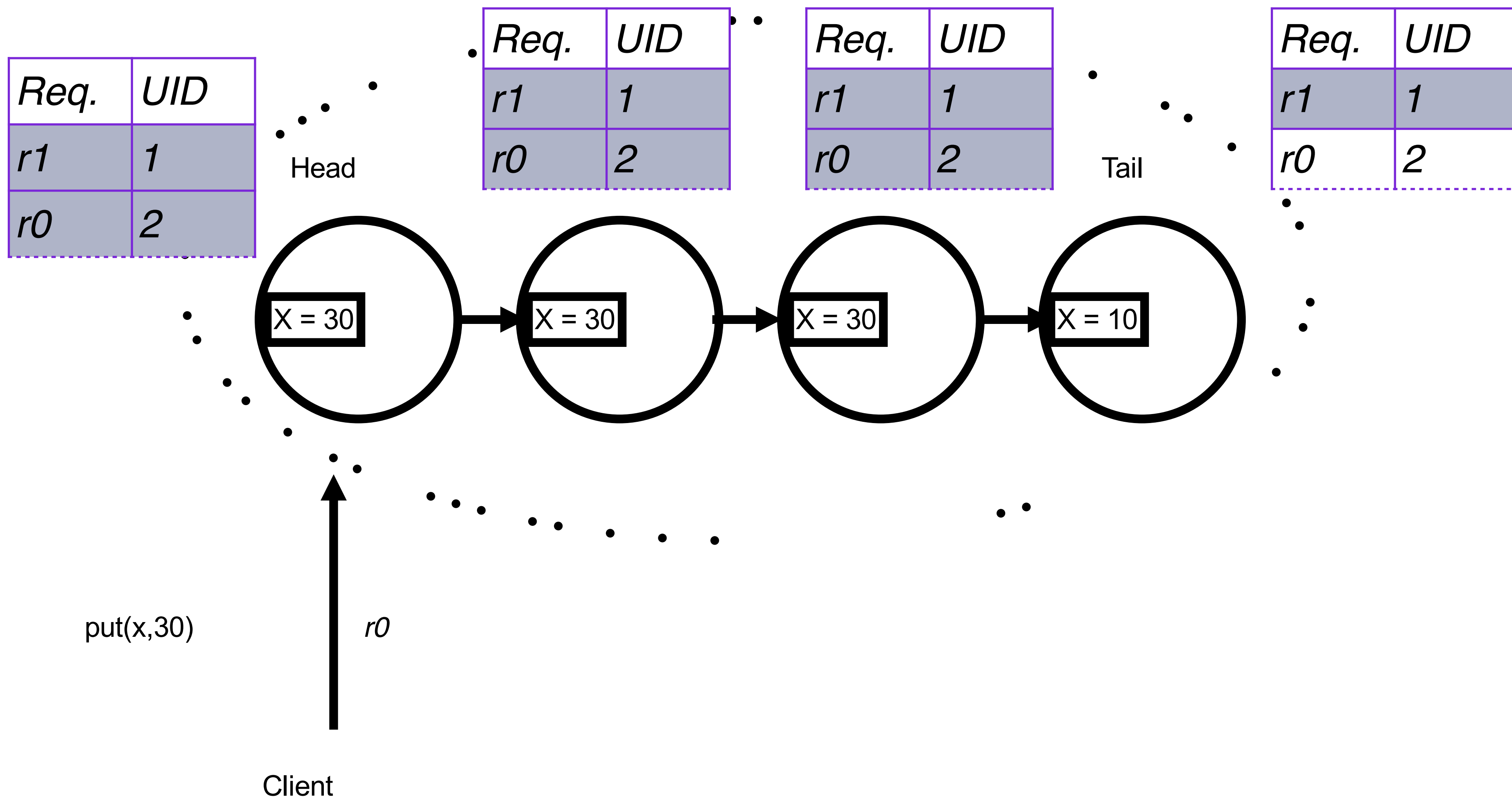
# Chain Replication



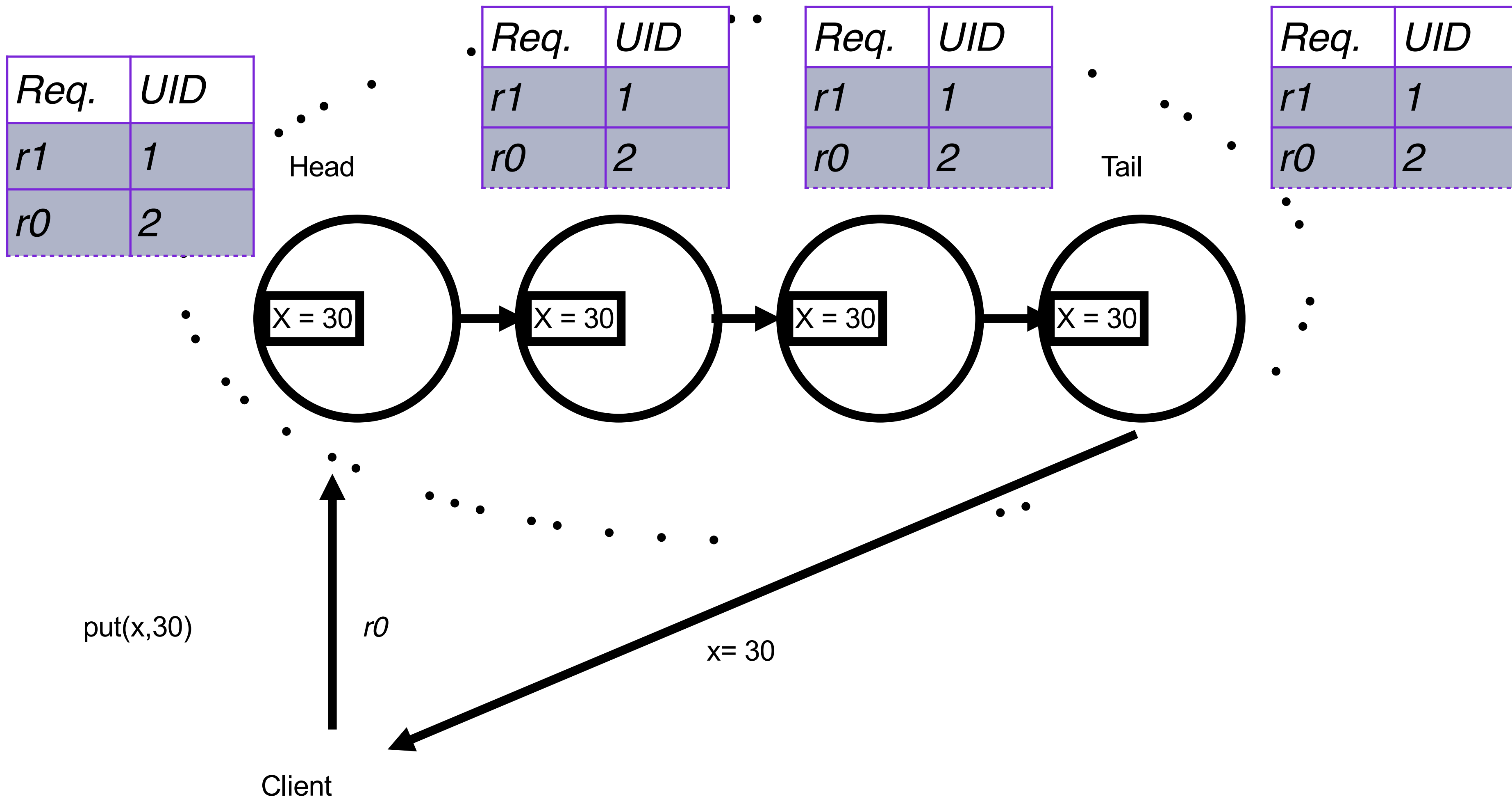
# Chain Replication



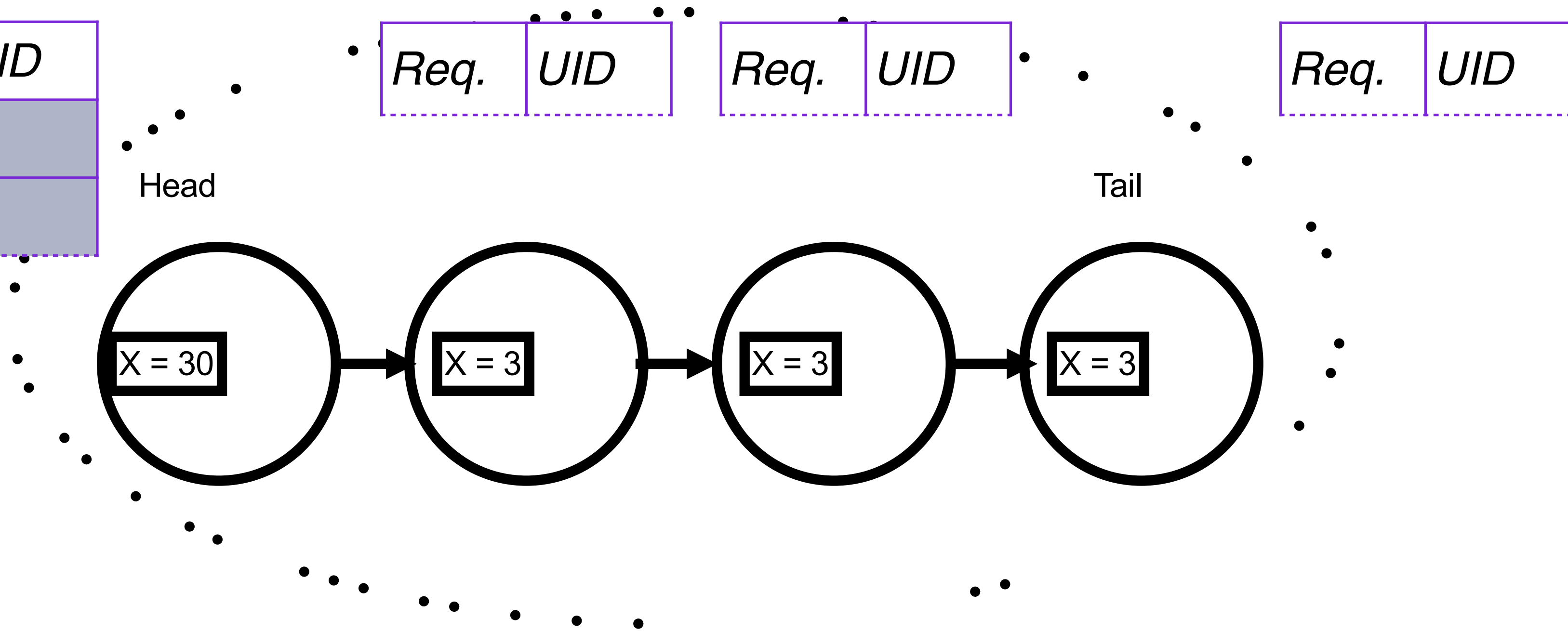
# Chain Replication



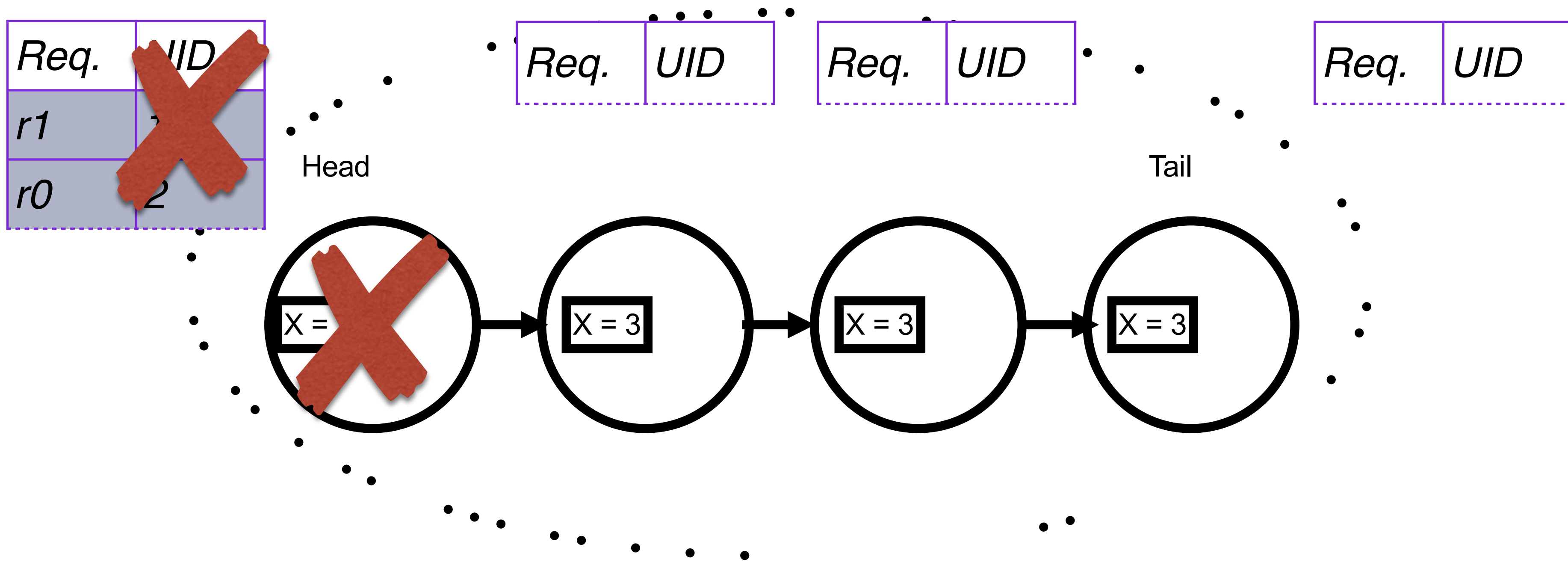
# Chain Replication



# Fault Tolerance

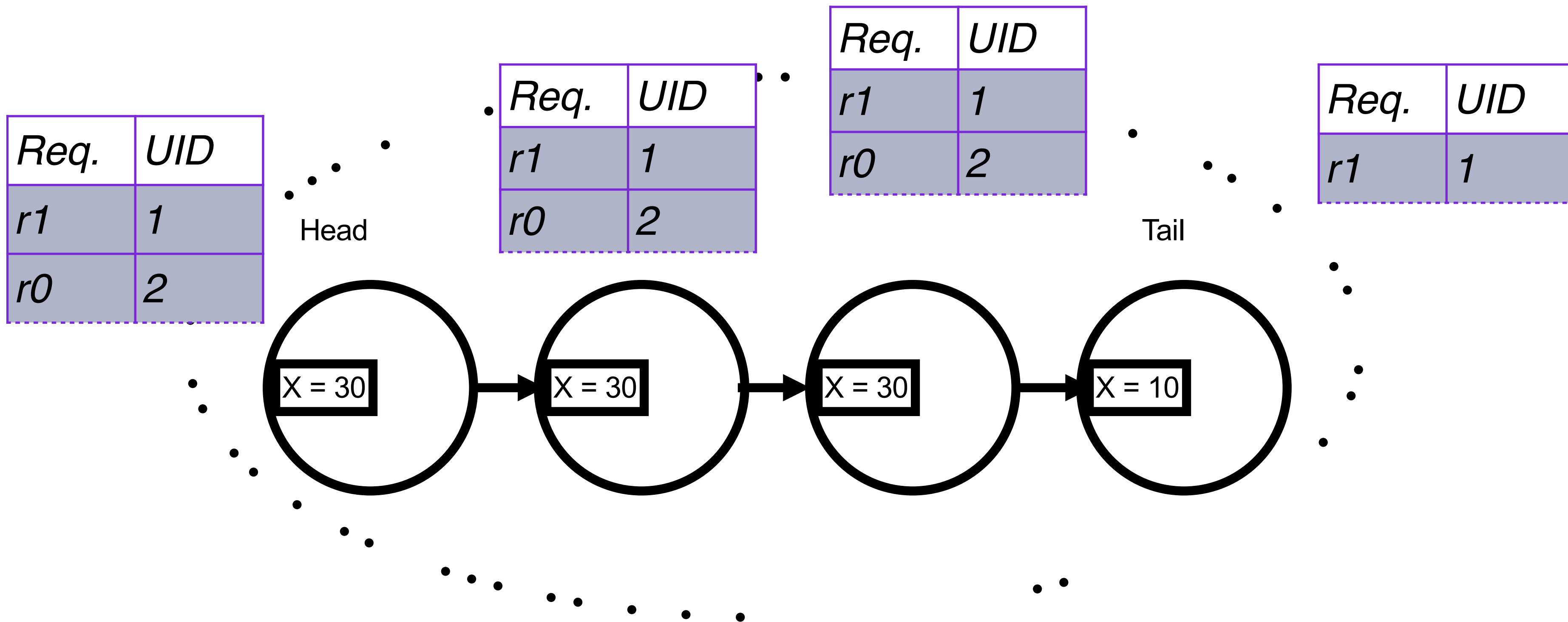


# Fault Tolerance

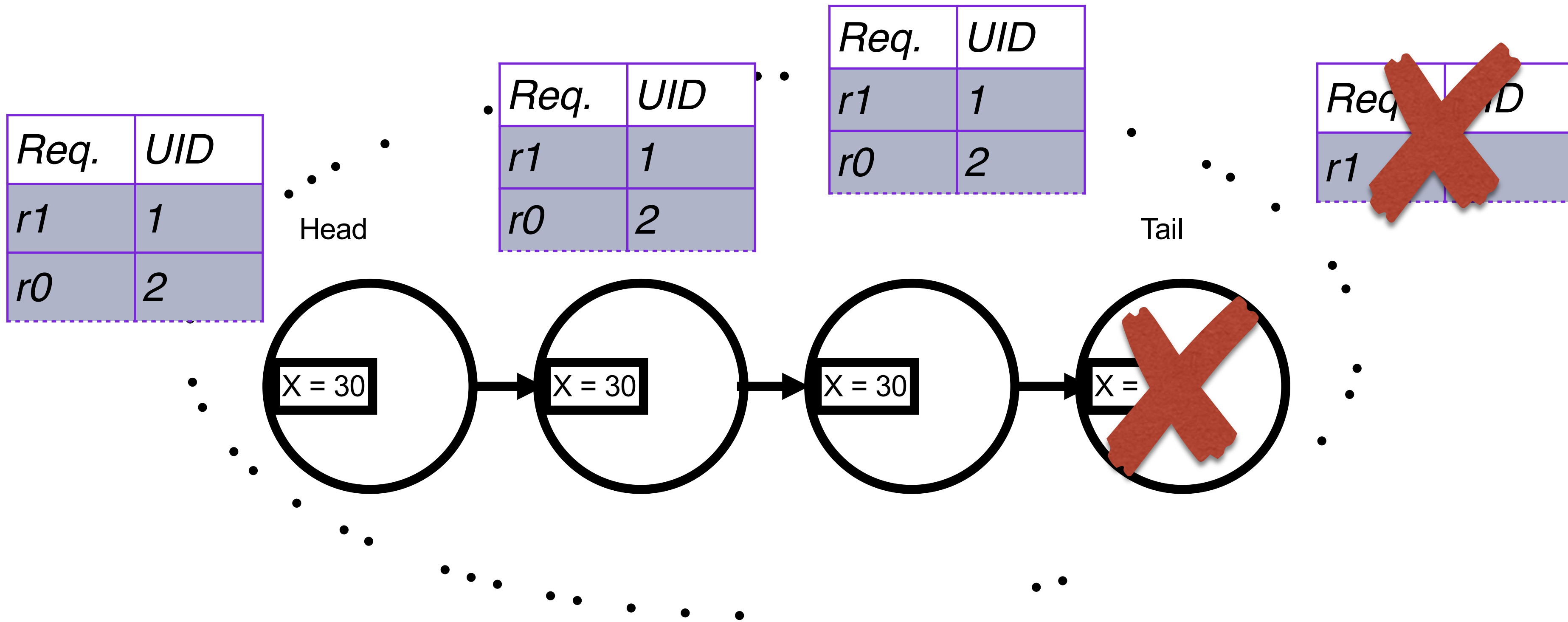



Dropped requests *r1* and *r0*

# Fault Tolerance

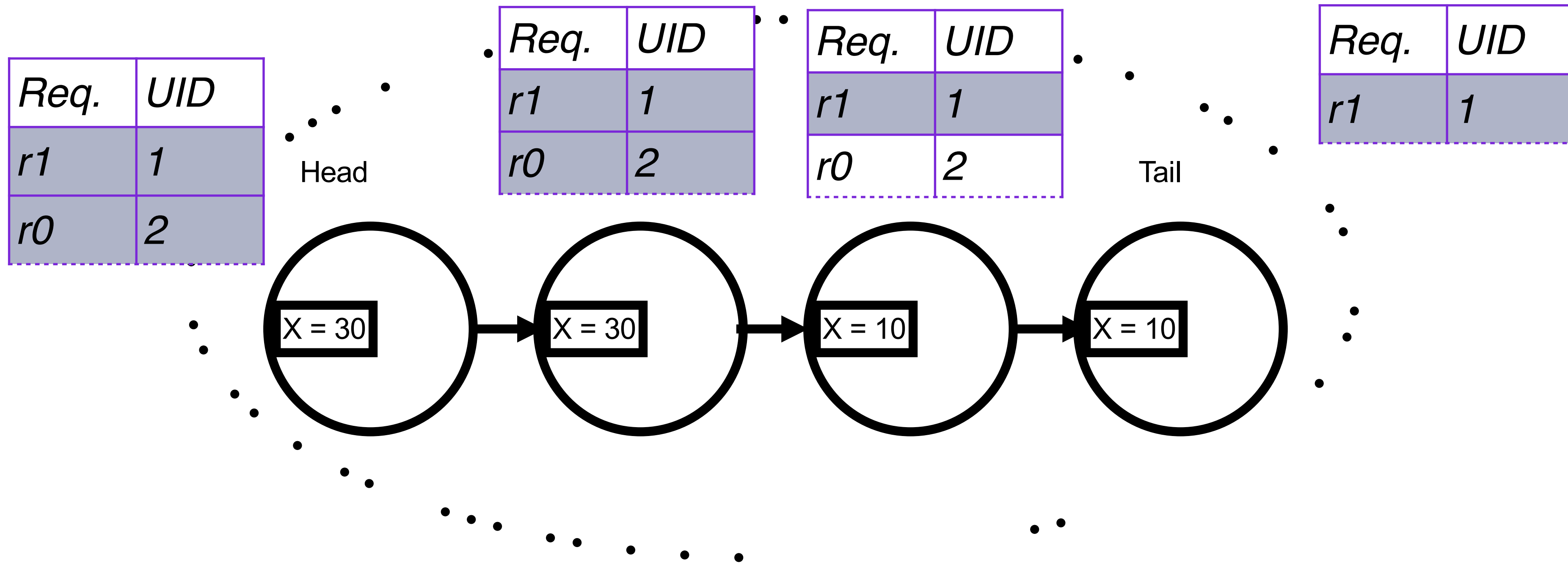


# Fault Tolerance

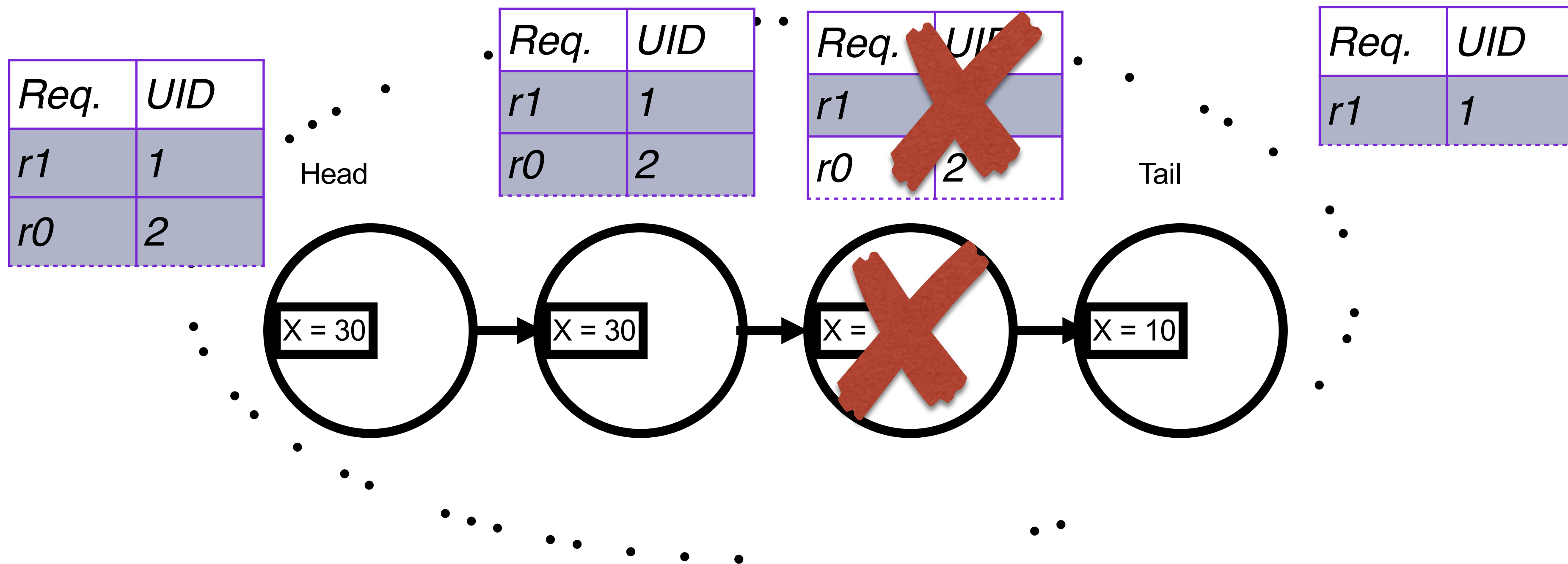



New tail is *stable* for superset of old tail's requests

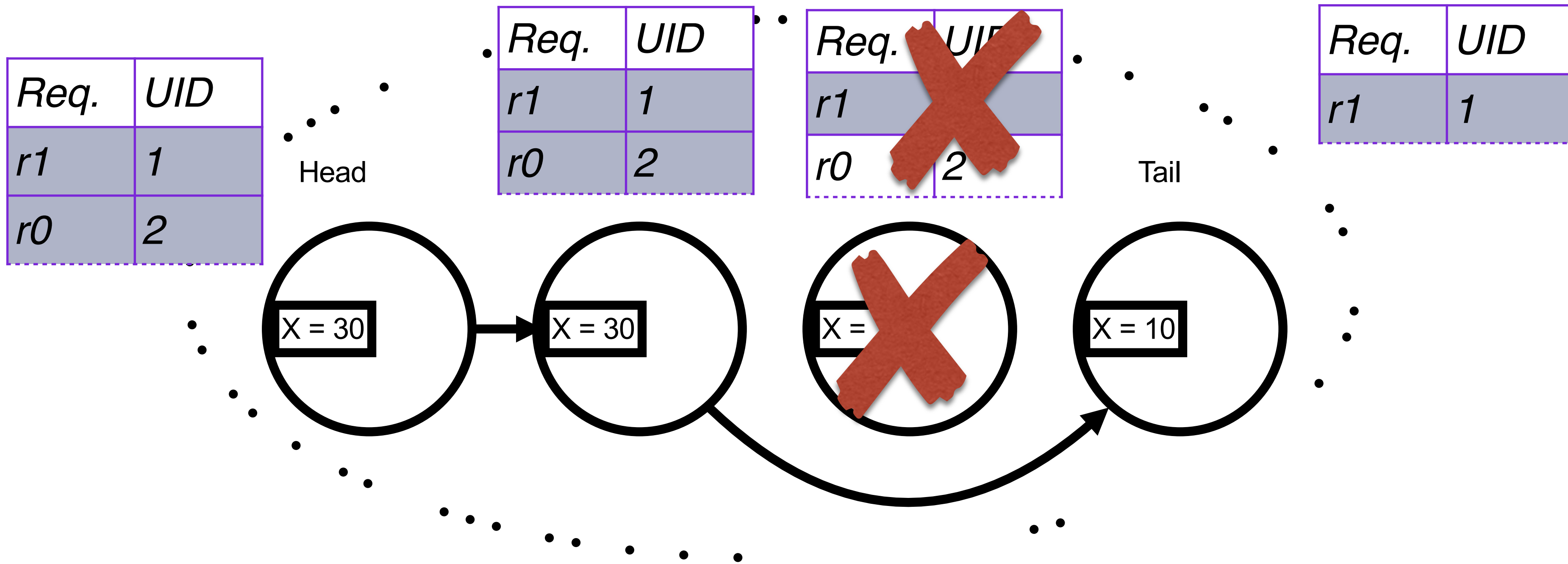
# Fault Tolerance



# Fault Tolerance

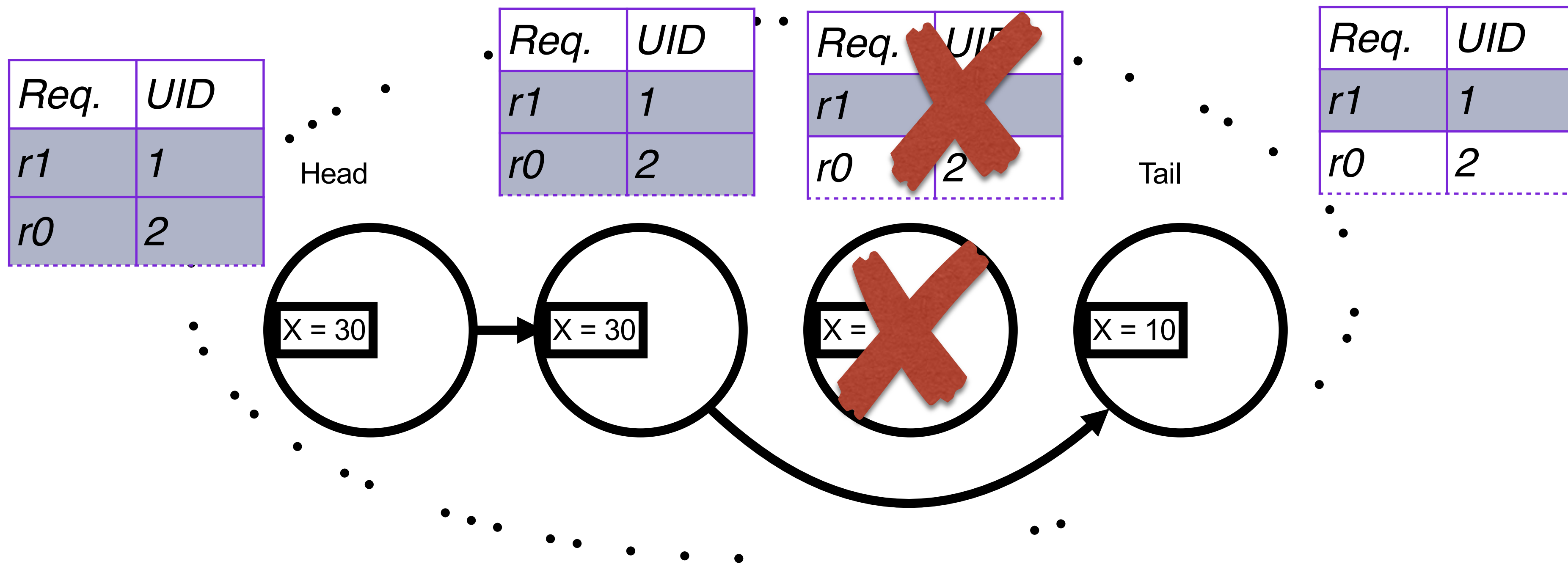


# Fault Tolerance




Need to *re-send r0*

# Fault Tolerance

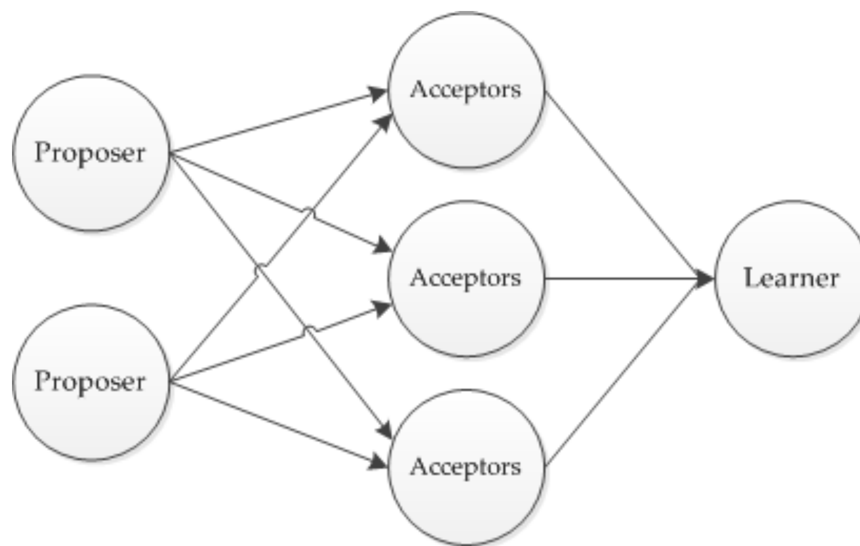



Need to *re-send r0*

How is all of this assignment managed?

# Paxos

- Assumptions:
  - communication is unreliable. messages may be lost, duplicated, or reordered
  - messages that are corrupted can be detected
  - all operations are deterministic
  - processes might exhibit crash failures but not arbitrary failures
  - no collusion between processes



# Byzantine Fault Tolerance

# Byzantine Fault Tolerance

- want consistent state (consensus) even in the presence of arbitrary (possibly malicious) failures

- need to make assumptions on how many failures can occur