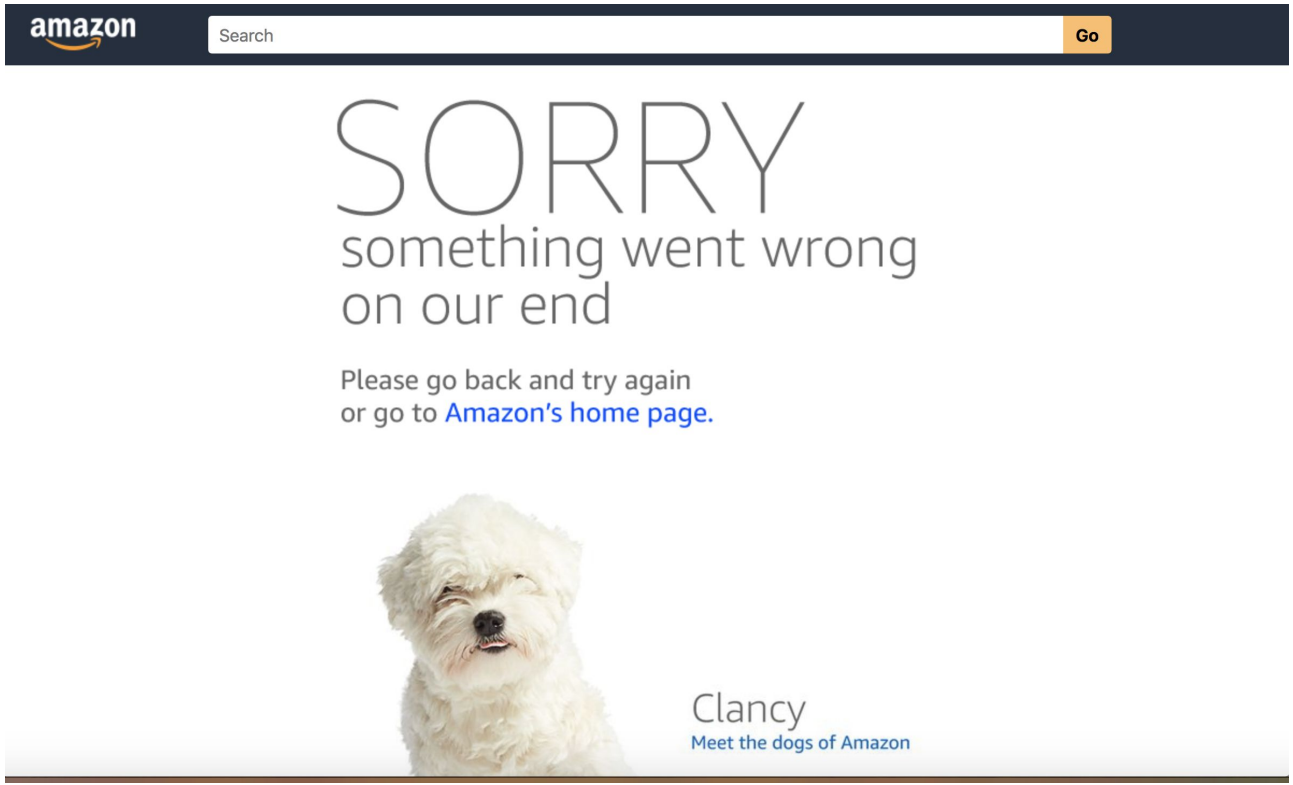
Reasoning About Replication: State Machine Approach & Chain Replication

Partial slides borrowed from Drew Zagieboylo and Chinasa T. Okolo
Presented by Yunhe Liu @ CS6410 10/24

Failure case: Service Unavailable



Picture source: https://fortune.com/2018/07/16/amazon-prime-day-2018-glitch-website-crashing-not-working/

Failure case: Critical Applications

Picture Source: http://clipart-library.com/cartoon-plane-images.html

Implementing Fault-Tolerant Service the State Machine Approach: A Tutorial

FRED B. SCHNEIDER

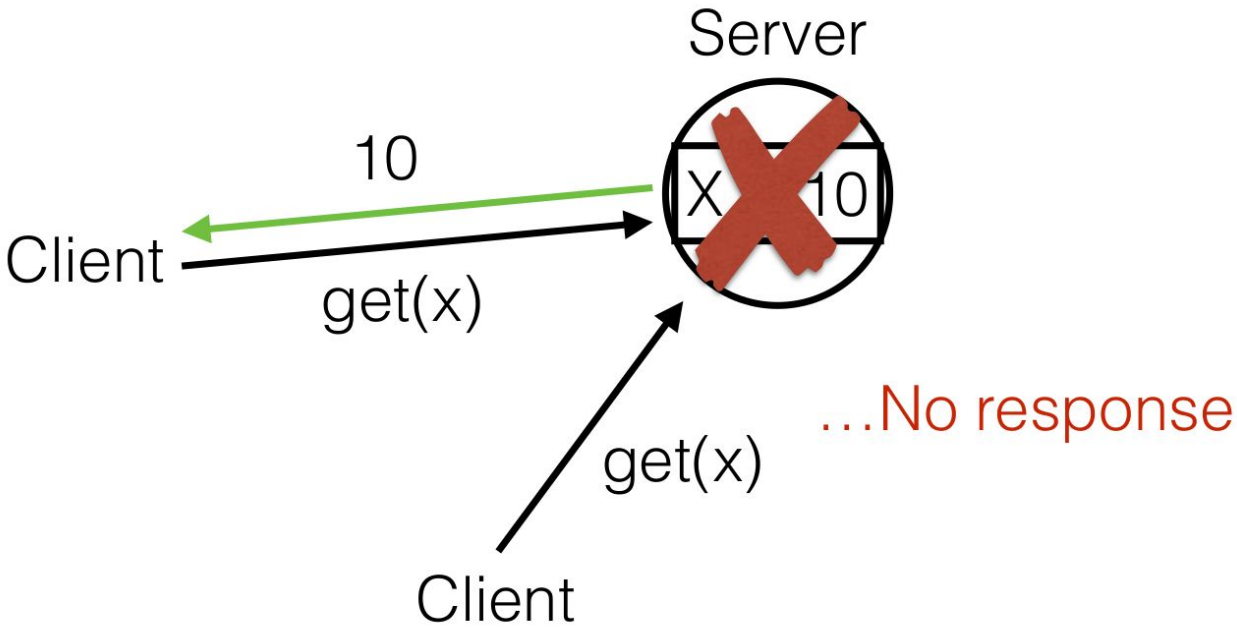
Published @ ACM Computing Surveys (1990)

Author

Fred B. Schneider
Cornell University

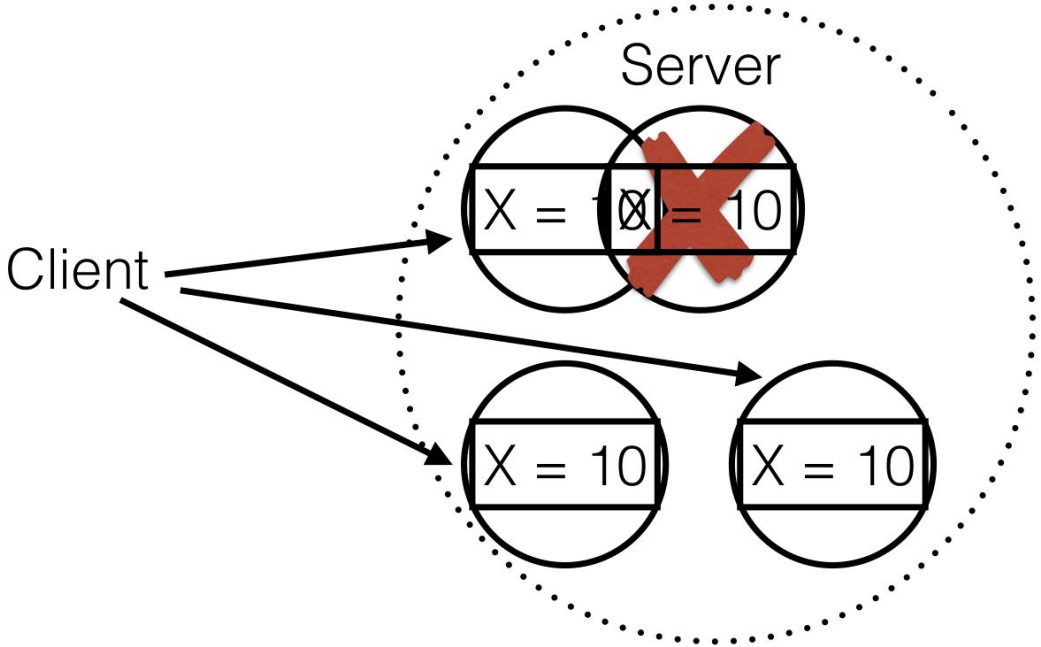
Samuel B. Eckert Professor of Computer Science
AAAS, ACM, and IEEE Fellow

Client-Server Model



- Client send commands to server. Server send response to client.
- If the server failed, the client get no response (service unavailable)
- Even worse, server send client wrong response.

Fault Tolerance



- Replicate the server. Each copy is called a replica.
- A mechanism to coordinate replicas so that certain failures does not affect correctness & availability of the service.

Roadmap

1. State Machine
2. Failure Tolerant State Machine
    a. Agreement
    b. Ordering
3. Bounds on Fault-Tolerance

State Machine

State Machine has two components:

1. State variables: encode its state.
2. Commands: transform its state.

We will see what is a state machine using an example.

State Machine: A Example

```
memory: state_machine
          var store: array[0 . . n] of word
  read: command(loc: 0 . . n)
          send store[loc] to client
          end read;
  write: command(loc: 0 . . n, value: word)
          store[loc] := value
          end write
    end memory
```

State variables: encode its state.

Figure 1. A memory.

State Machine: A Example

```
memory: state_machine
          var store: array[0 . . n] of word

  read: command(loc: 0 . . n)
          send store[loc] to client
          end read;

  write: command(loc: 0 . . n, value: word)
           store[loc] := value
           end write
    end memory
```

Commands (transform its state)

Figure 1. A memory.

State Machine: A Example

```
memory: state_machine
          var store: array[0 . . n] of word
  read: command(loc: 0 . . n)
          send store[loc] to client
          end read;
  write: command(loc: 0 . . n, value: word)
          store[loc] := value
          end write
    end memory
```

Figure 1. A memory.

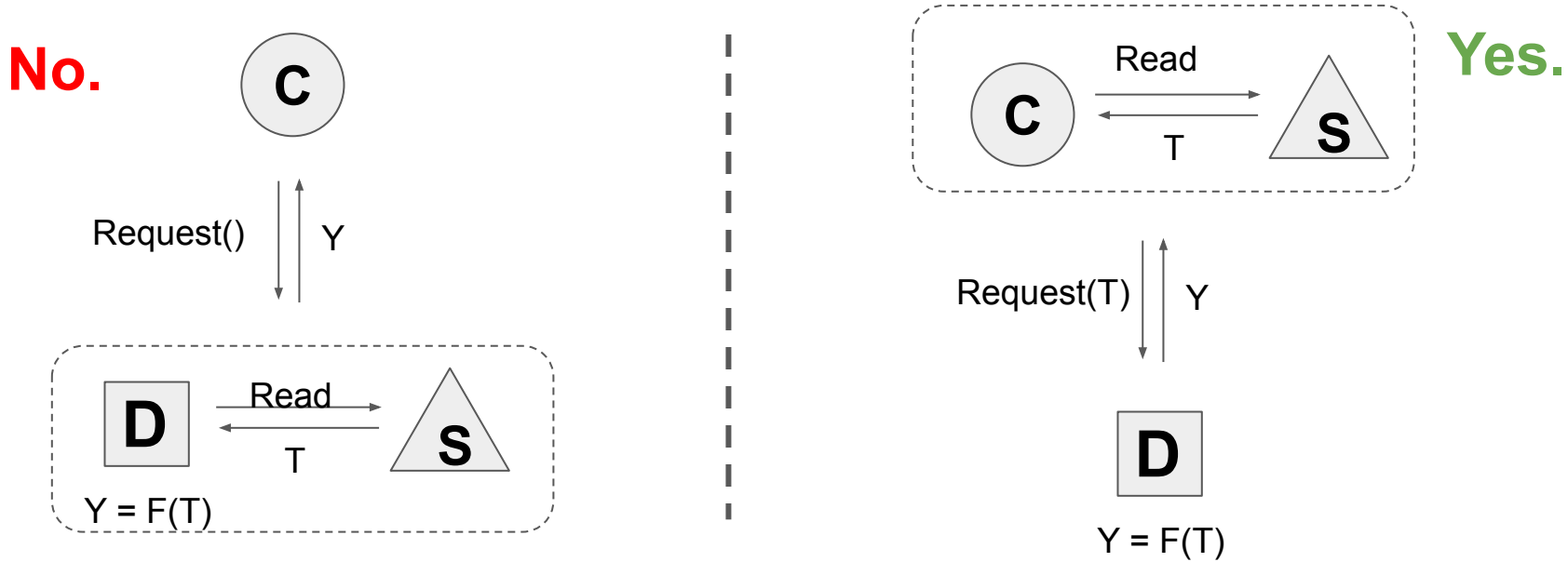
```
⟨memory.write, 100, 16.2⟩;
⟨memory.read, 100⟩;
receive v from memory
```

Semantic Characterization of a SM

Outputs of a state machine are:

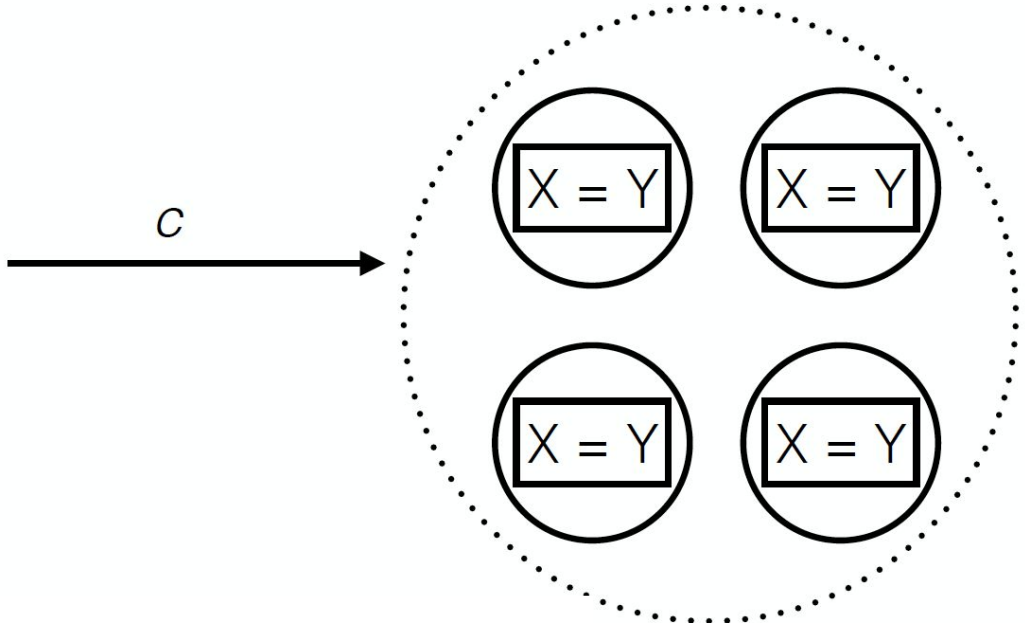
- Completely determined by the sequence of commands it processes.
- Independent of time and any other activity in the system.

Semantic Characterization of a SM: An example



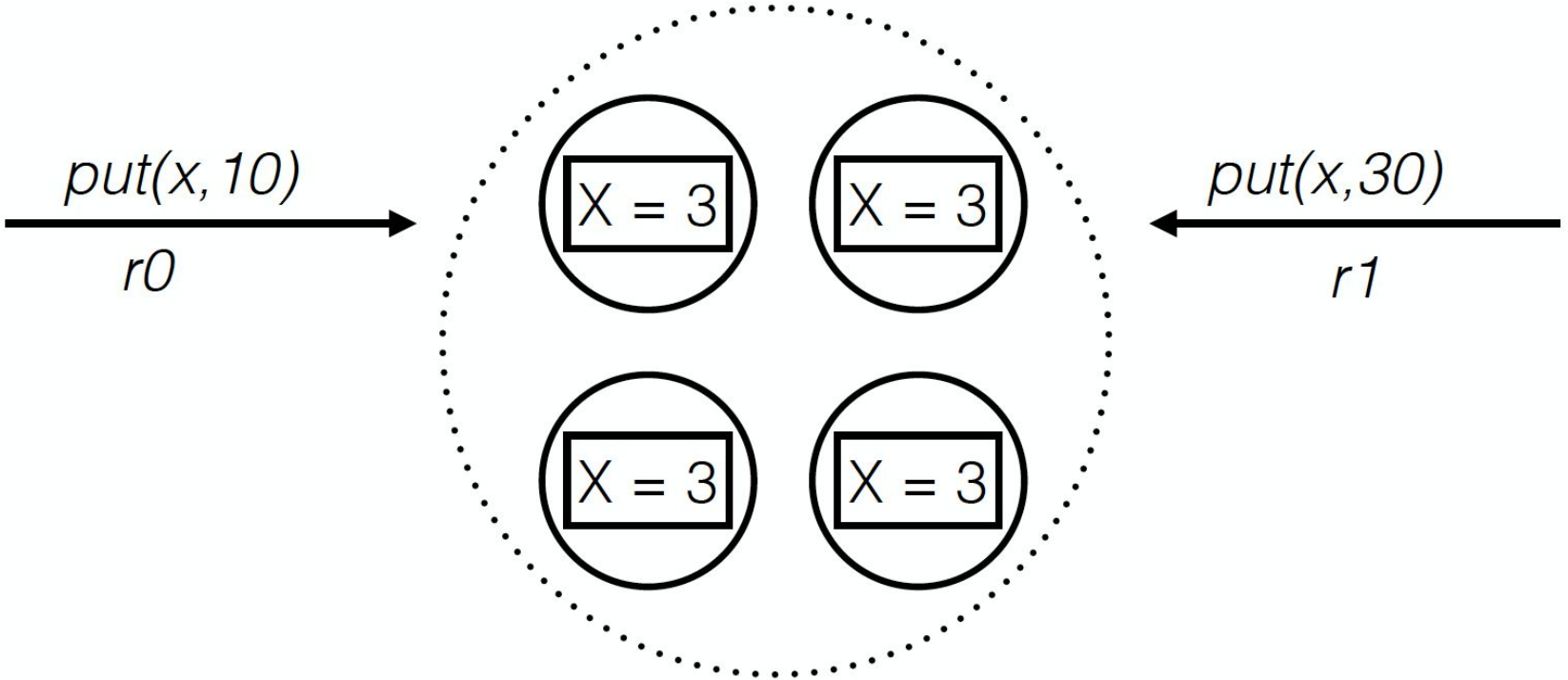
- S is a sensor, reading a value T that varies with real-time.
- D is a decision making state machine. C is a client.
- State machine output depend on input commands only. Not affected by time.

State Machine Approach

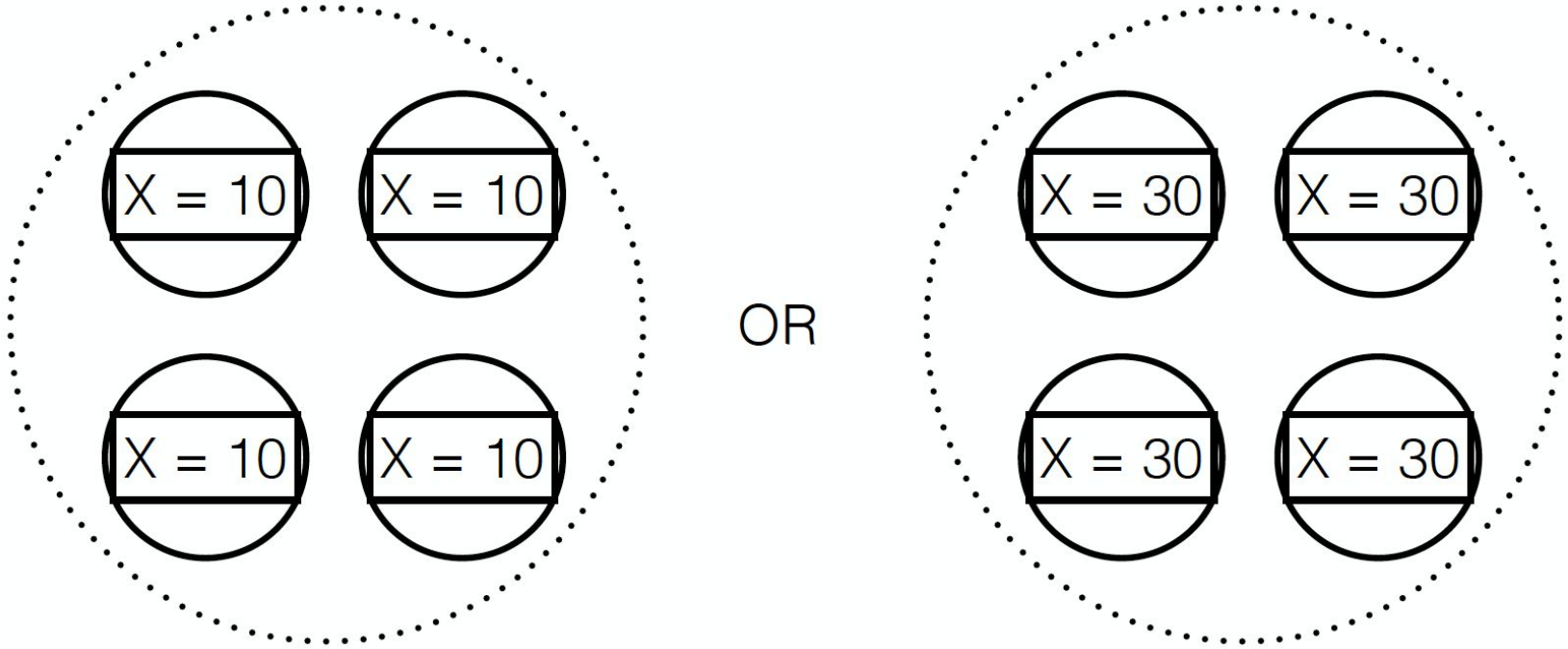


- Implement the server as replicated state machines (independent failures).
- Each replica processes the **same** commands in **the same order**.
- The service can function correctly as long as **some** replica(s) do not fail.

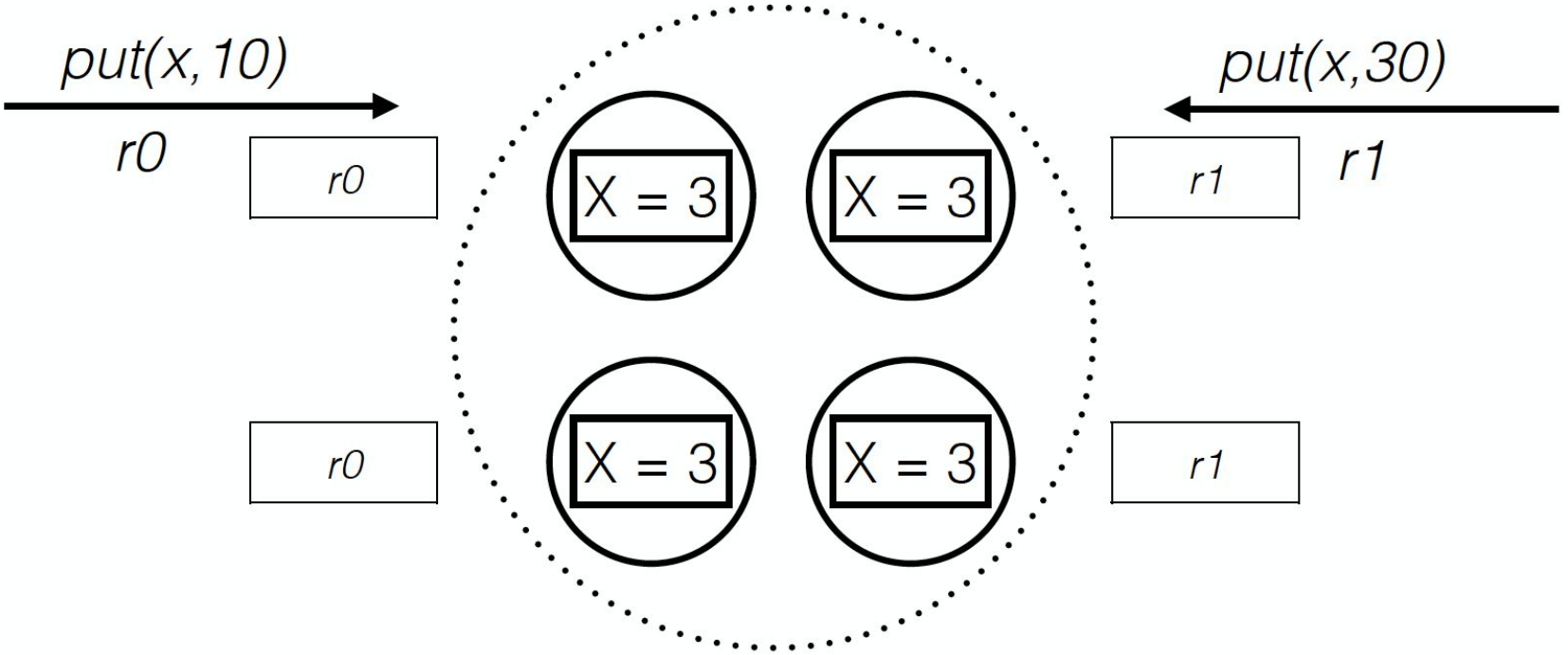
Process Same Commands in the Same Ordering



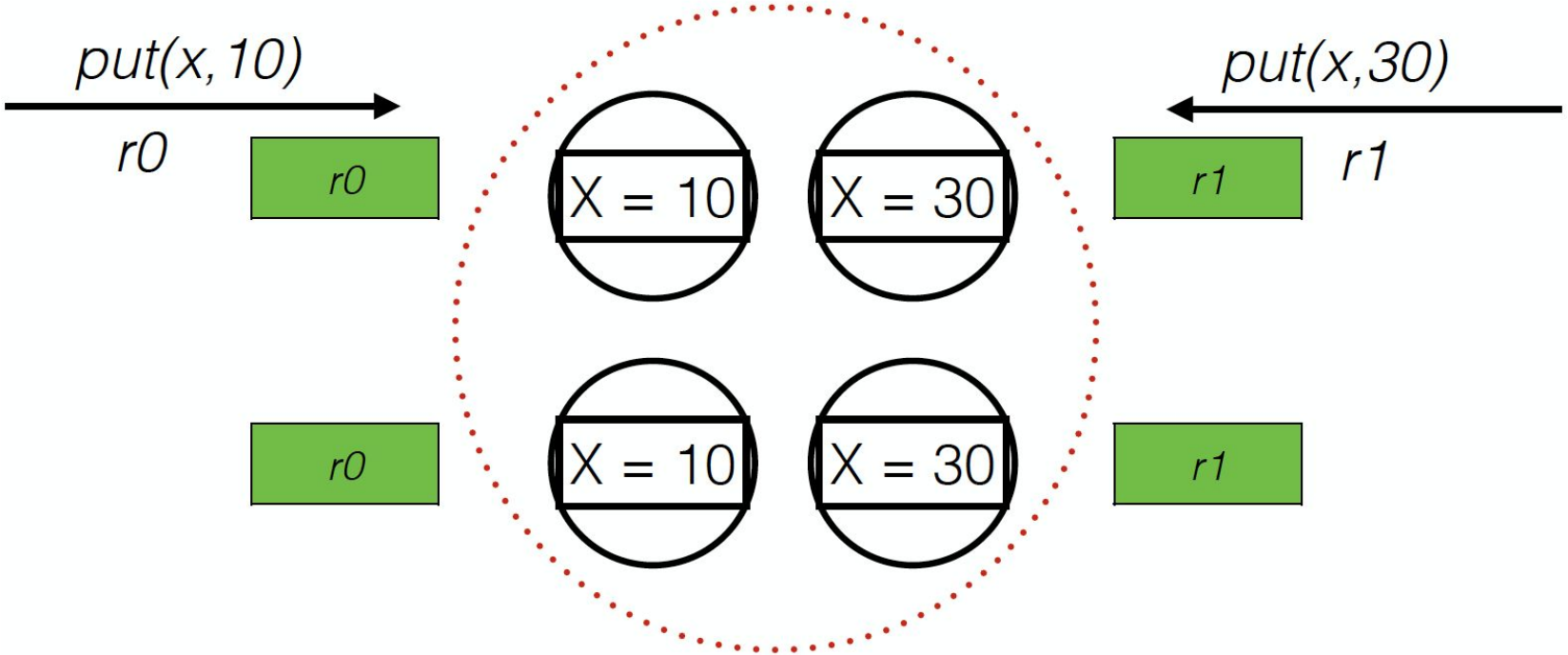
Process Same Commands in the Same Ordering



Not Receiving the Same Commands



Not Receiving the Same Commands



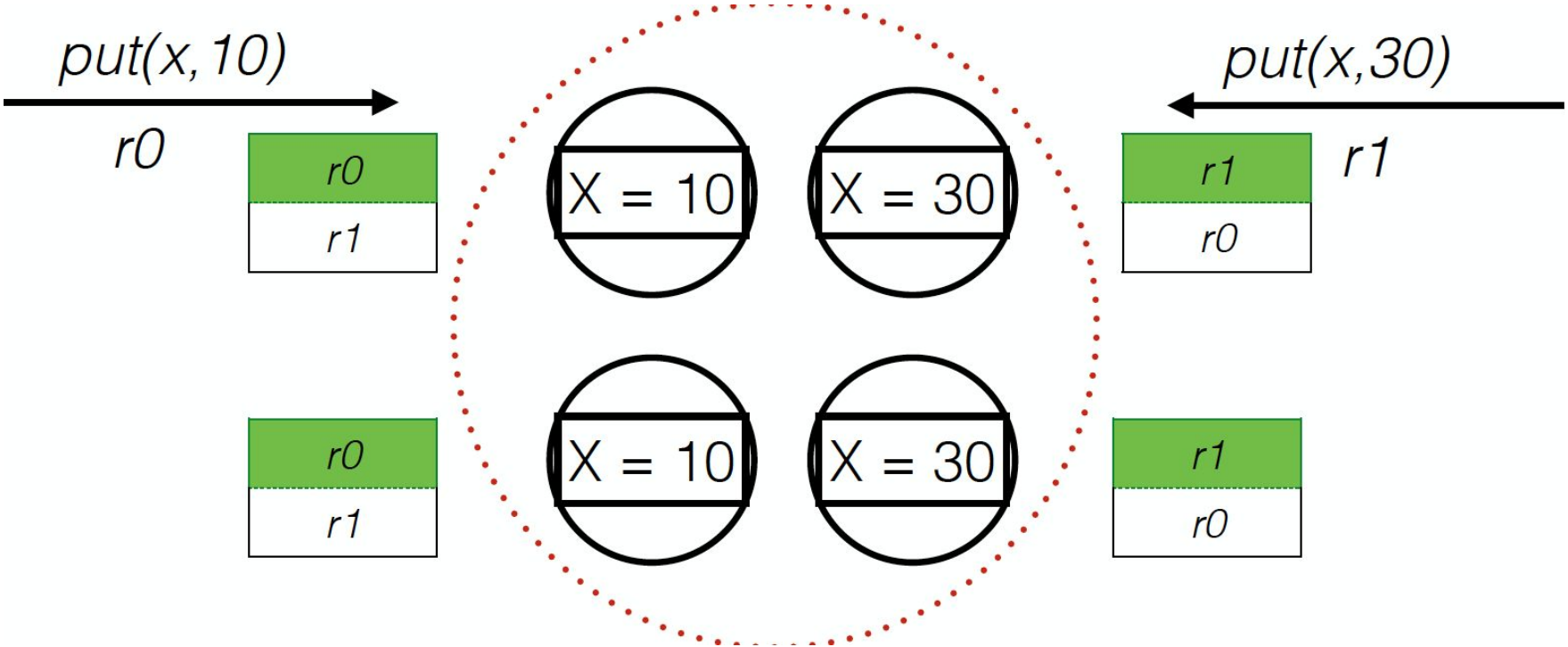
Agreement: Same Commands

A client sends a command; if that client is non-faulty, all state machine replicas will receive the command.

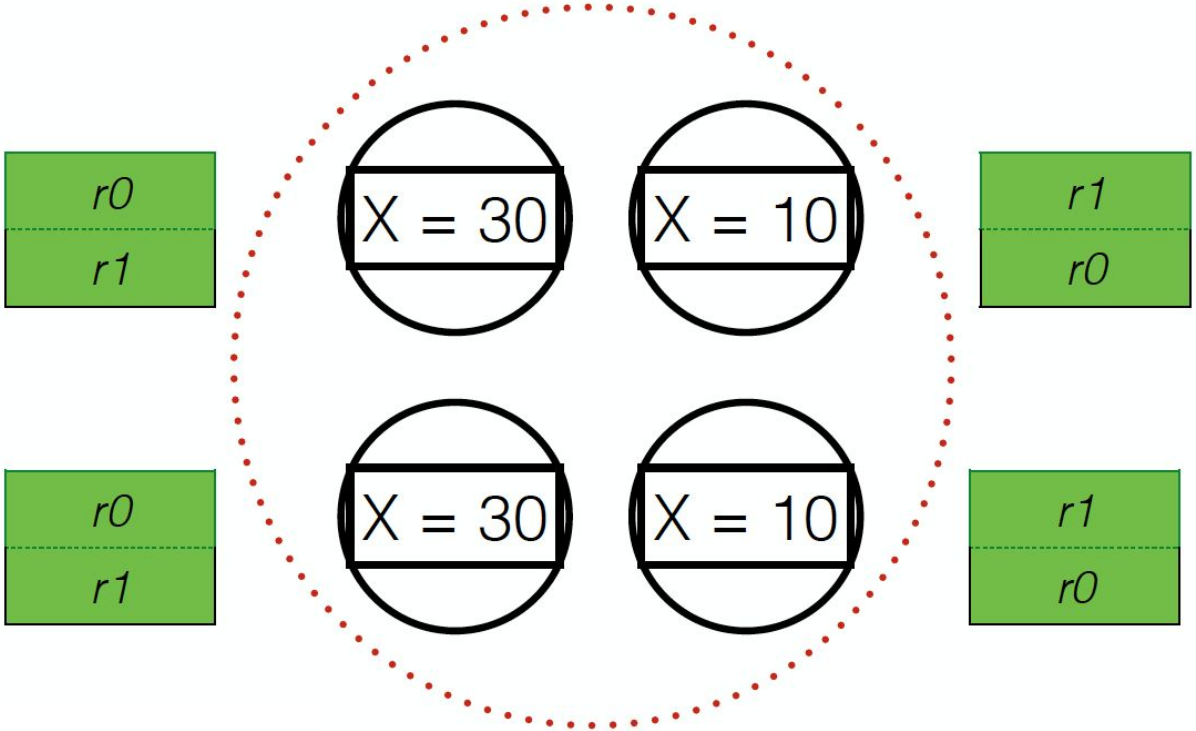
The Paper Referred to Literature for existing protocols:

- Byzantine Agreement protocols, reliable broadcast protocols, agreement protocols
- Strong and Dolev [1983], Schneider et al. [1984]

Not Processing Commands in the Same Order



Not Processing Commands in the Same Order



Order: Process Commands in the Same Order

- Assign unique ids (total ordering) to requests, process them in ascending order.
- How do we assign unique IDs (total ordering)?

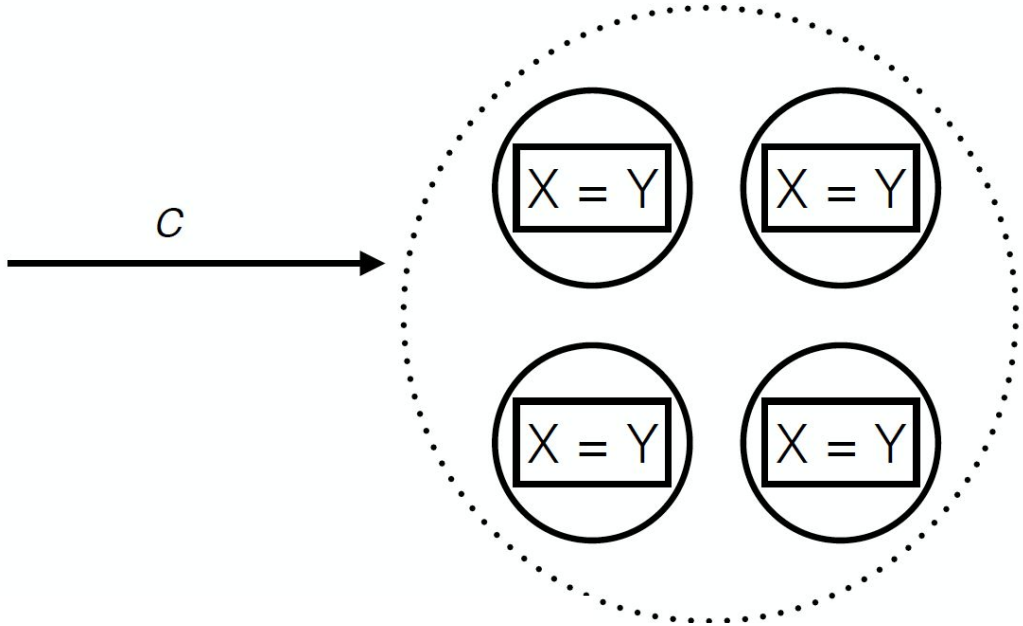
Assigning Total Order to Commands

- Logical Clock (We saw this Tuesday)
  - Logical Clock + Processor ID -> produce total order.

- Real-time clock
  - Clock need to have fine granularity so that no two commands can be issued on the same clock tick.
  - Clock need to have finer granularity than the minimum message delay time.

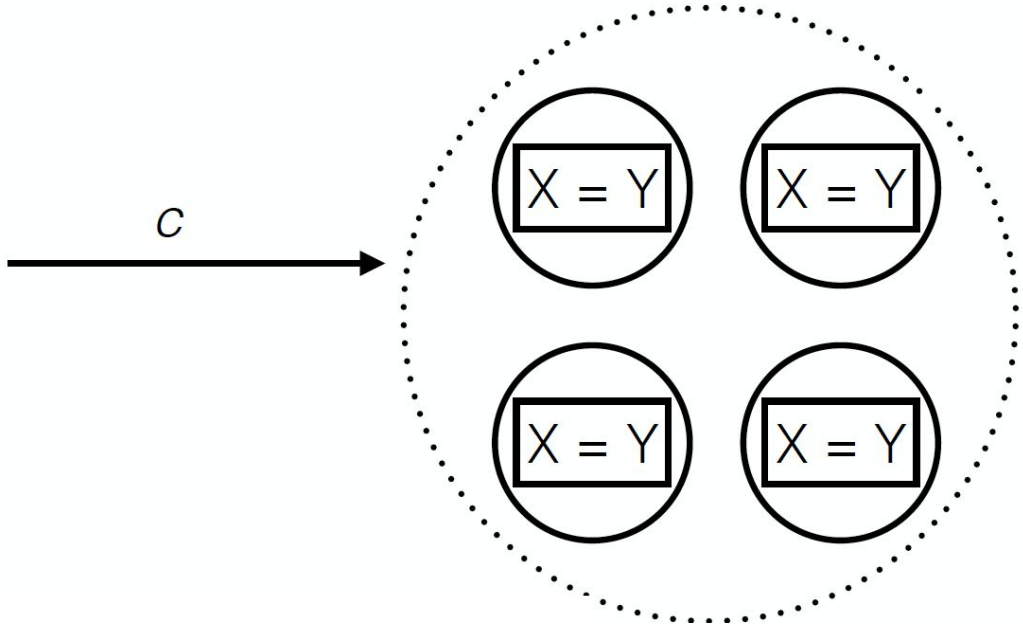
- Replica Generated Ids (2-phase)
  - Phase1: Every replica propose a candidate
  - Phase2: One candidate is chosen and agreed upon by all replicas

State Machine Approach



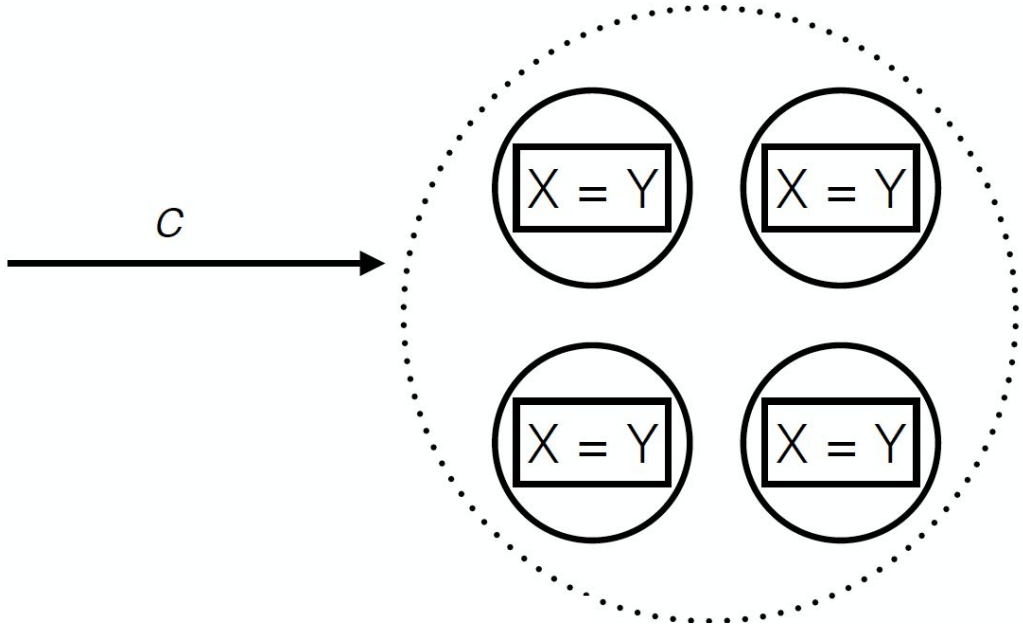
- Implement the server as replicated state machines (independent failures).
- Each replica execute the **same** commands in **the same order** independently.
- The service can function correctly as long as **some** replica(s) are not failed.

Failure Model: Fail-Stop



- Fail-Stop: Faulty replicas can be detected.
- As long as 1 replica is correct, the service is correct and available.
- Need at least t + 1 replicas to tolerant t failures.

Failure Model: Byzantine Failure



- Byzantine Failure: Faulty servers can do arbitrary, perhaps malicious things.
- Need to vote when different replica output different result.
- Need at least 2t + 1 replicas to tolerant t failures.

Takeaway

- Can represent deterministic distributed system as Replicated State Machine.
- Each replica reaches the same conclusion about the system independently.
- Formalizes notions of fault-tolerance in SMR.

Next

We will look at a specific instance of state machine replication: Chain Replication.

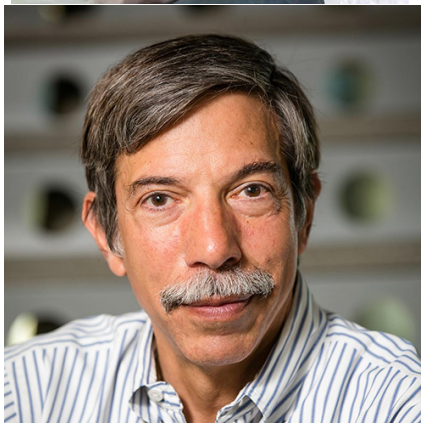
Chain Replication for Supporting High Throughput and Availability

Robbert van Renesse & Fred B. Schneider
Published @ OSDI'04

Authors

Robbert van Renesse
Cornell University

ACM Fellow and Ukelele enthusiast



Fred B. Schneider
Cornell University

State Machine Approach

Background

- Chain replication (CR) is a replication protocol coordinating large-scale storage servers.

- CR becomes a popular topic of research
  - Geambasu et al. DSN'08, Andersen et al. SOSP'09, Terrace et al. ATC'09, and many more.

- CR has been used widely in commercial products
  - MongoDB, MySQL, Microsoft Azure Blob Store, EMC Centera Clusters, CouchBase, and Ceph/RADOS etc.

Background

- The Goal of CR is to provide:
  - High throughput
  - High availability
  - Strong Consistency

- At the time, strong consistency were considered “in-tension” with high throughput and high availability
  - For example, GFS (We have seen this paper too!)

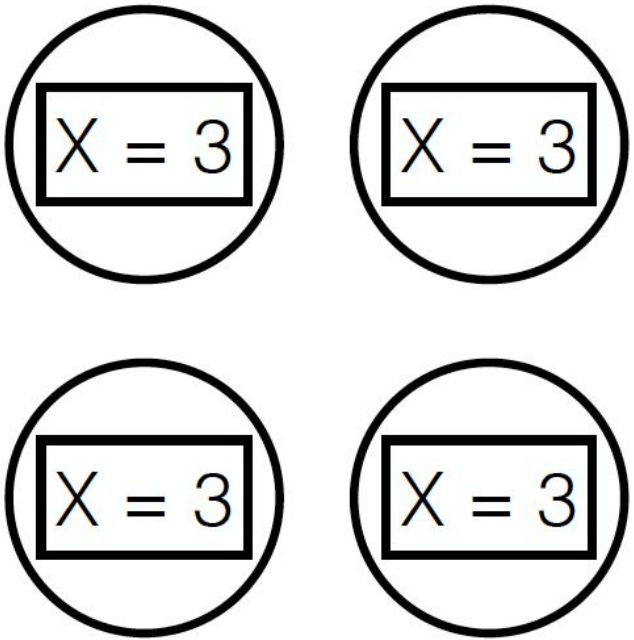
Storage System Interface

Requests:

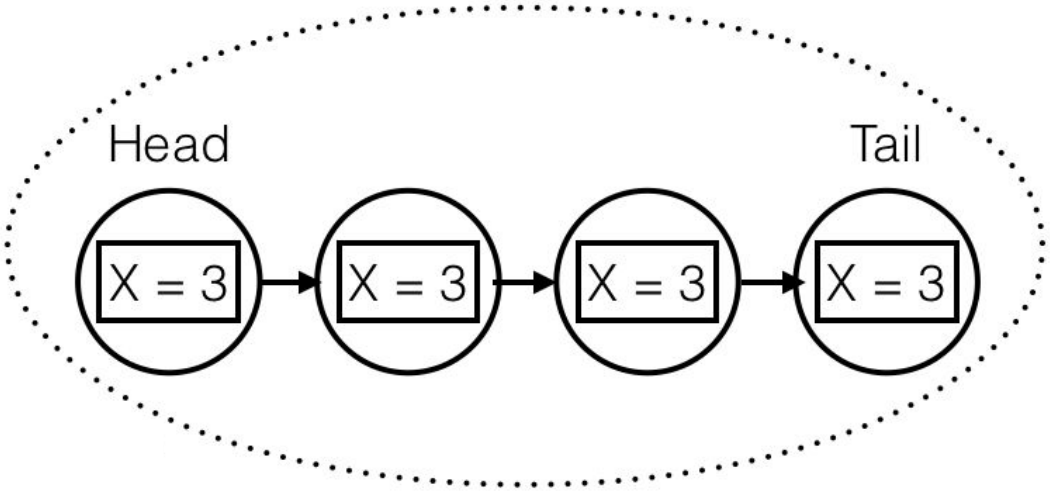
- Update(x, y) => set object x to value y
- Query(x) => read value of object x

Chain Replication assumps fail-stop failure model.

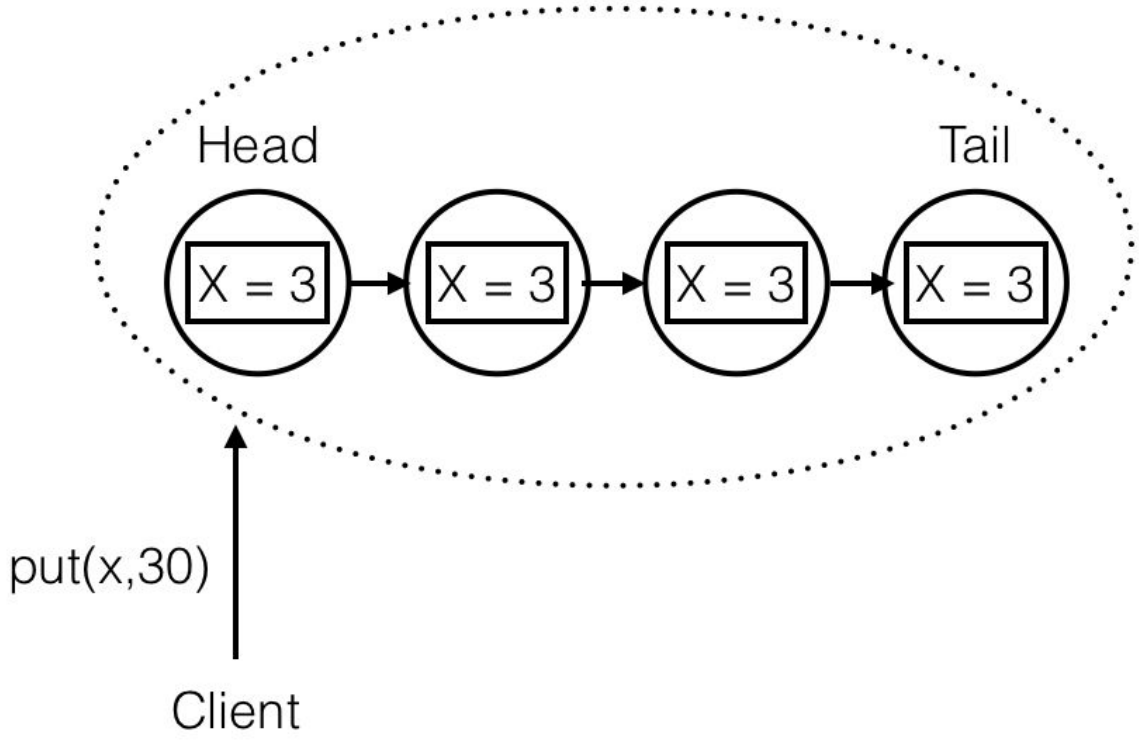
Chain Replication



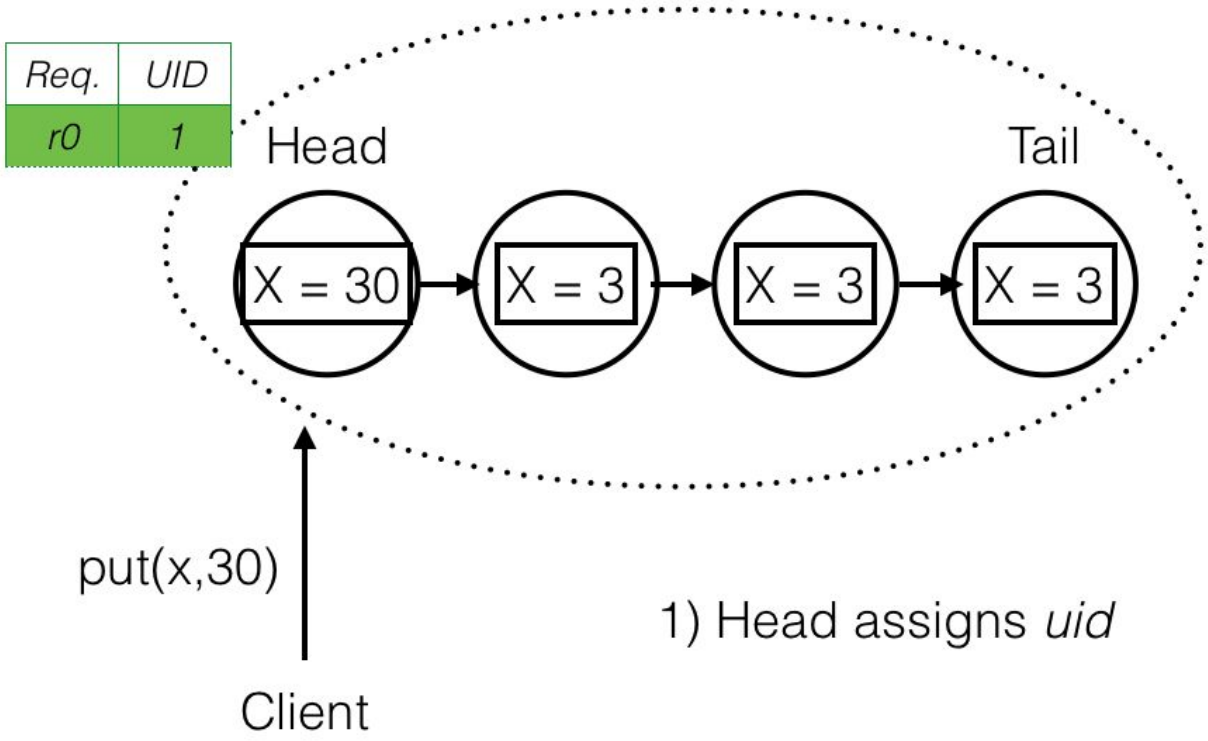
Chain Replication



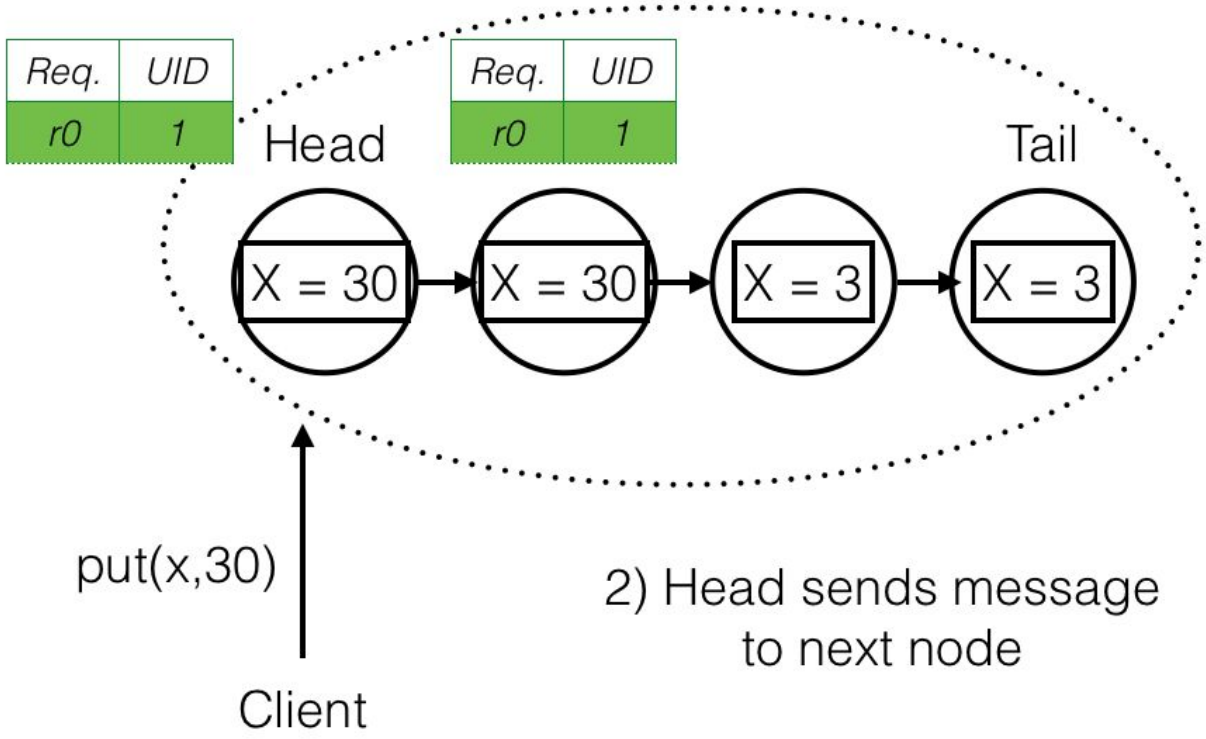
Update



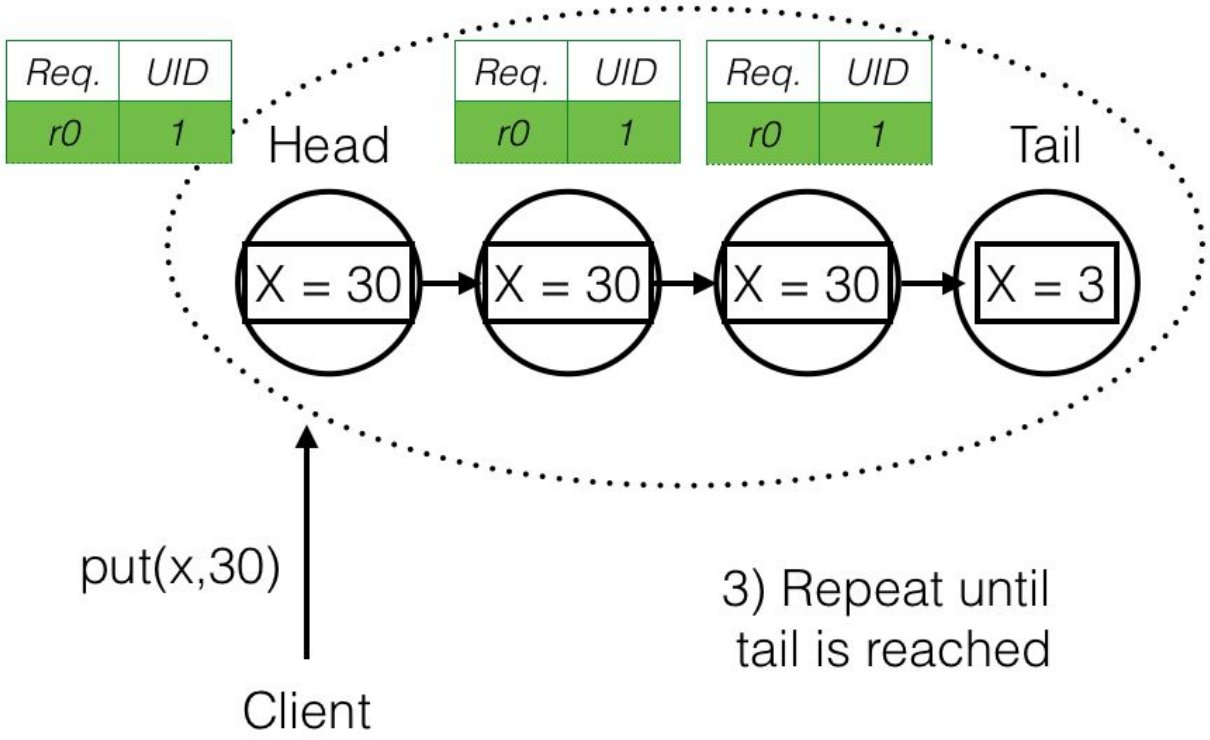
Update


Req.
UID
r0
1
Head
Tail
X = 30
X = 3
X = 3
X = 3
put(x,30)
1) Head assigns *uid*
Client

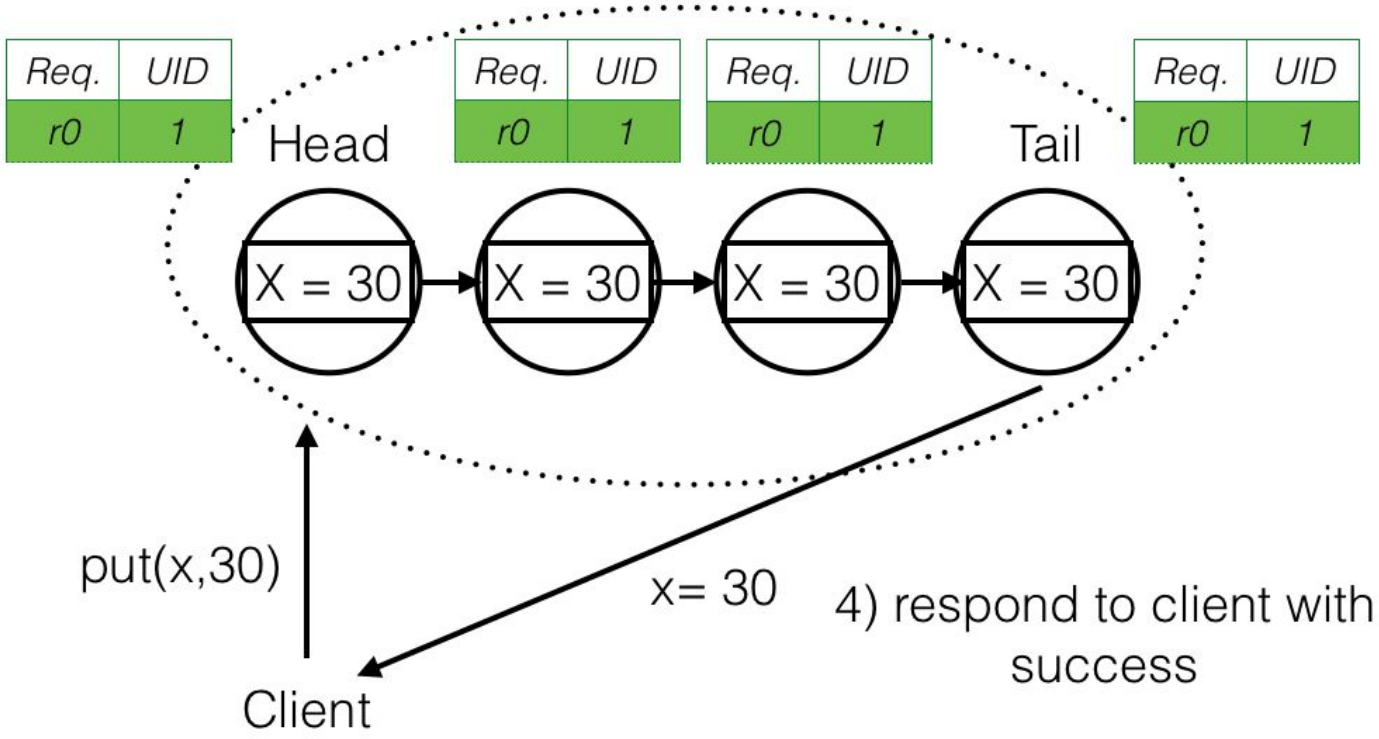
Update


Req.
UID
r0
1
Head
Req.
UID
r0
1
Tail
X = 30
X = 30
X = 3
X = 3
put(x,30)
Client
2) Head sends message
to next node

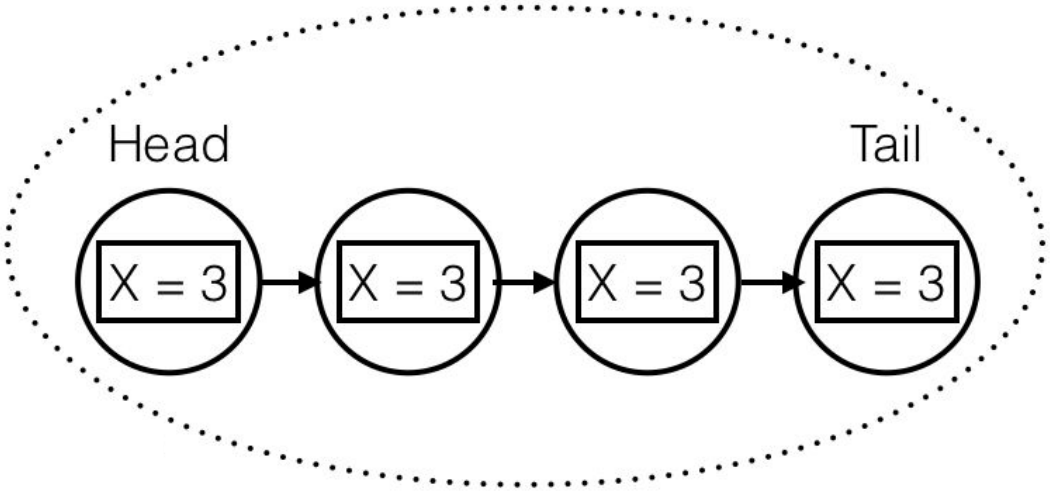
Update


Req.
UID
r0
1
Head
Req.
UID
r0
1
Req.
UID
r0
1
Tail
X = 30
X = 30
X = 30
X = 3
put(x,30)
Client
3) Repeat until
tail is reached

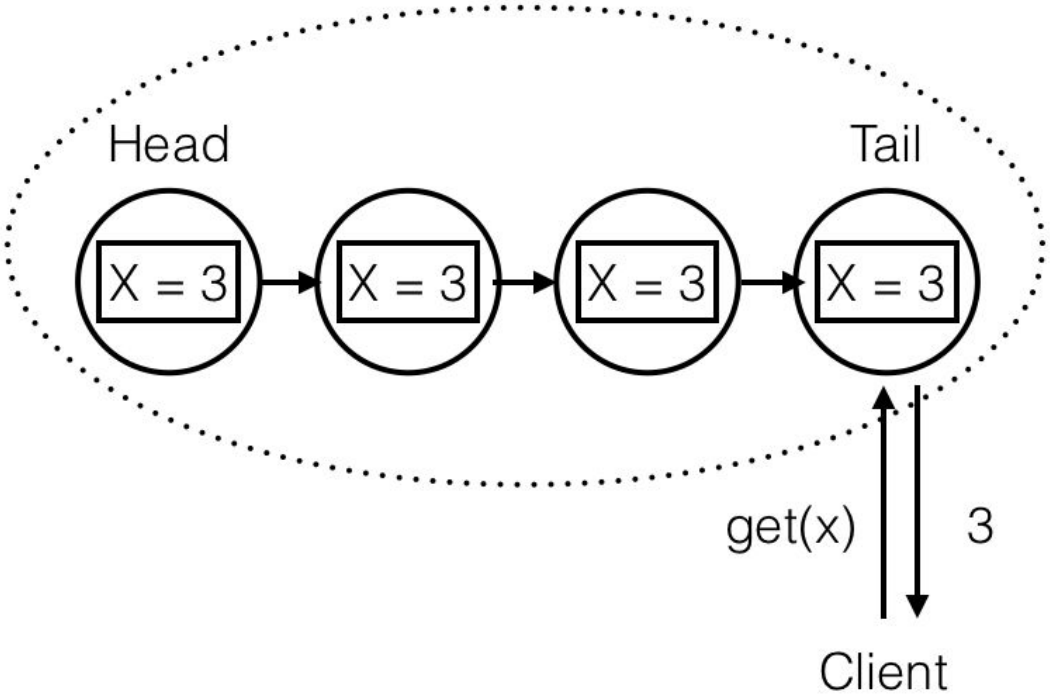
Update


Req.
UID
r0
1
Head
Req.
UID
r0
1
Req.
UID
r0
1
Tail
Req.
UID
r0
1
X = 30
X = 30
X = 30
X = 30
put(x,30)
x= 30
4) respond to client with success
Client

Query



Query



How did CR Implement State Machine Replication?

Agreement (Every replica process the same set of commands):

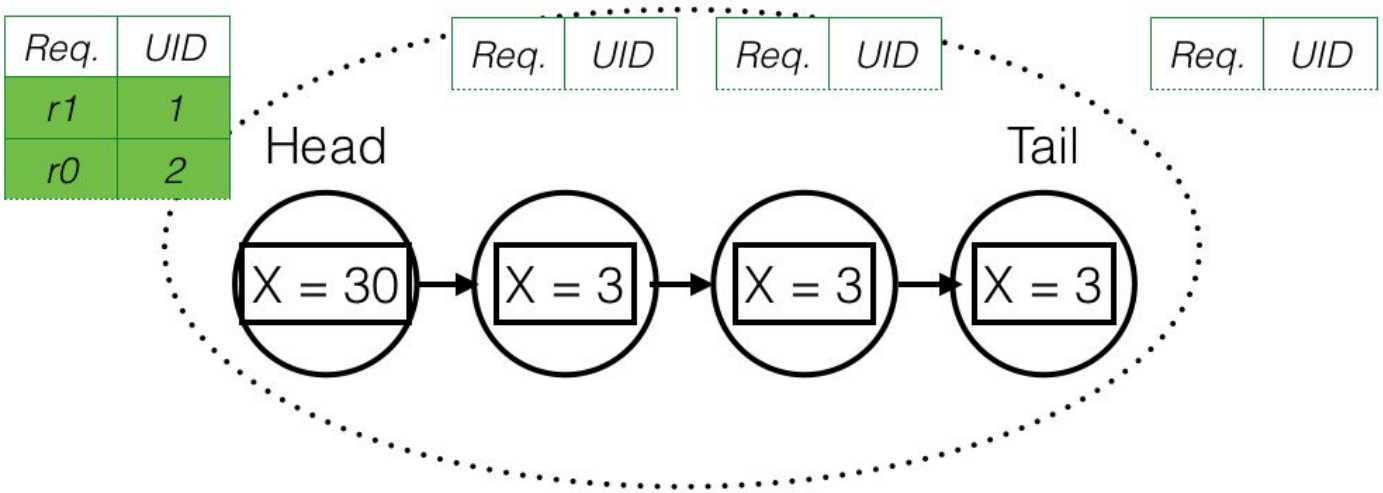
- Only Update modifies state, can ignore Query
- Client always sends update to Head.
- Head propagates request down chain to Tail.
- Every replica receives every update request.

How did CR inplement State Machine Replication?

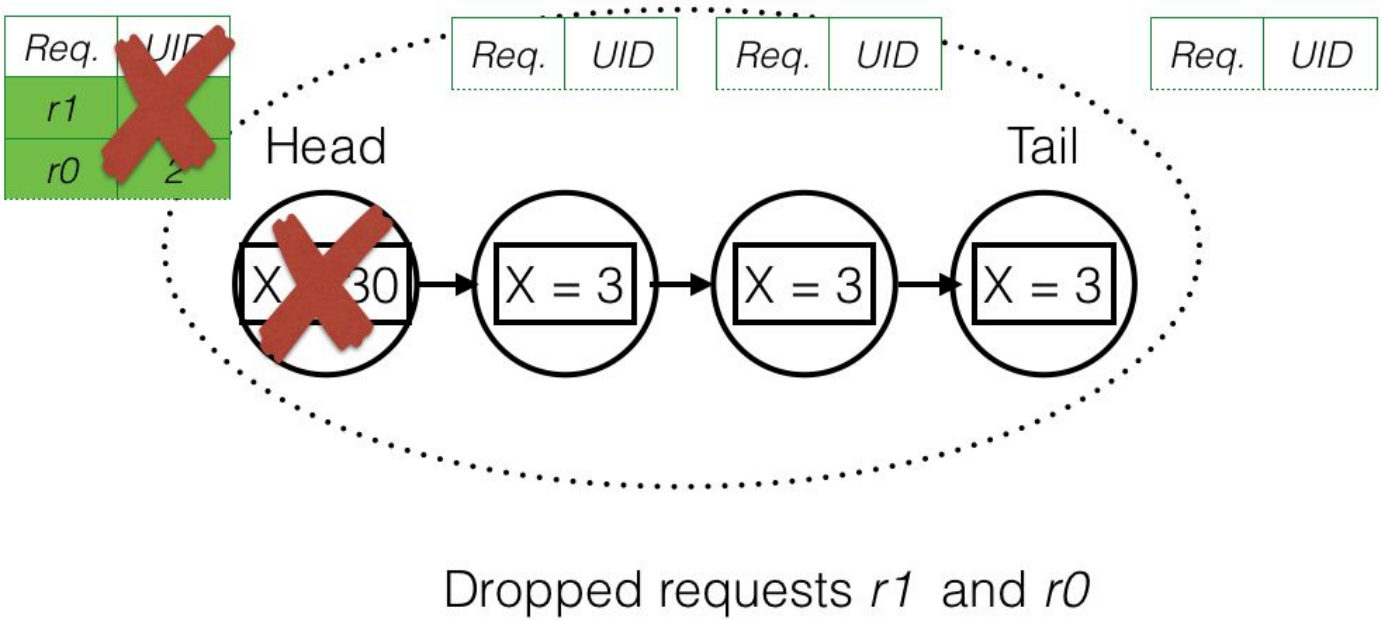
Order (Every replica process the same set of commands):

- Only Update modifies state, can ignore Query
- Unique IDs generated implicitly by Head’s ordering
- FIFO order preserved down the chain
- Every update request propagates down the chain in the same order.

Fault Tolerance

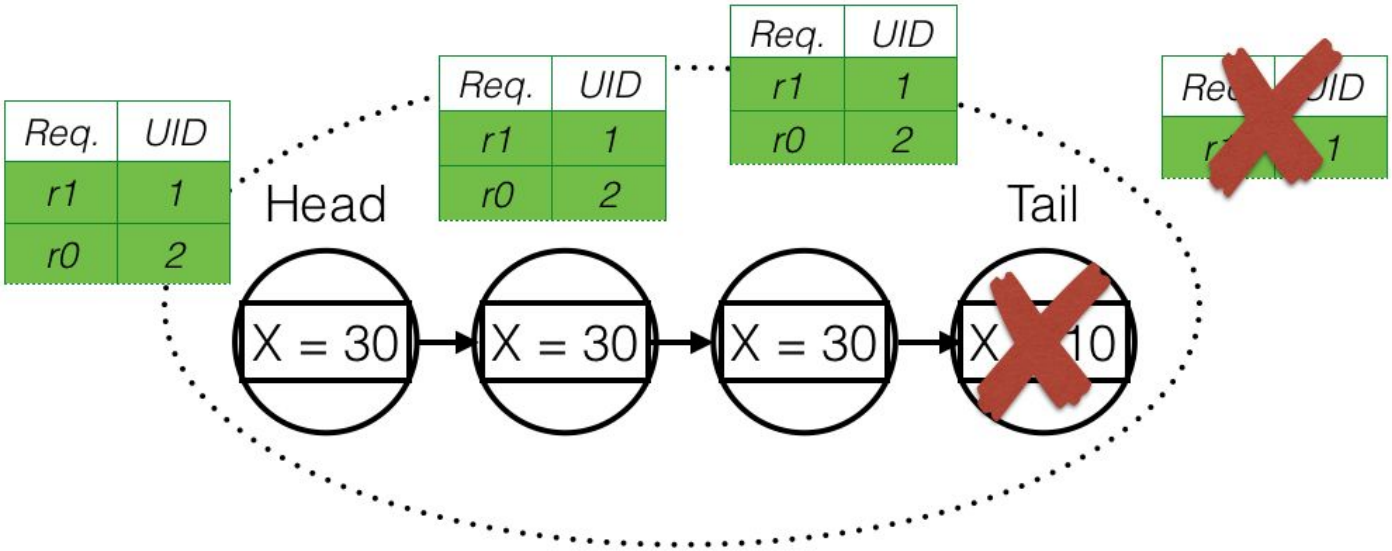


Fault Tolerance: Head



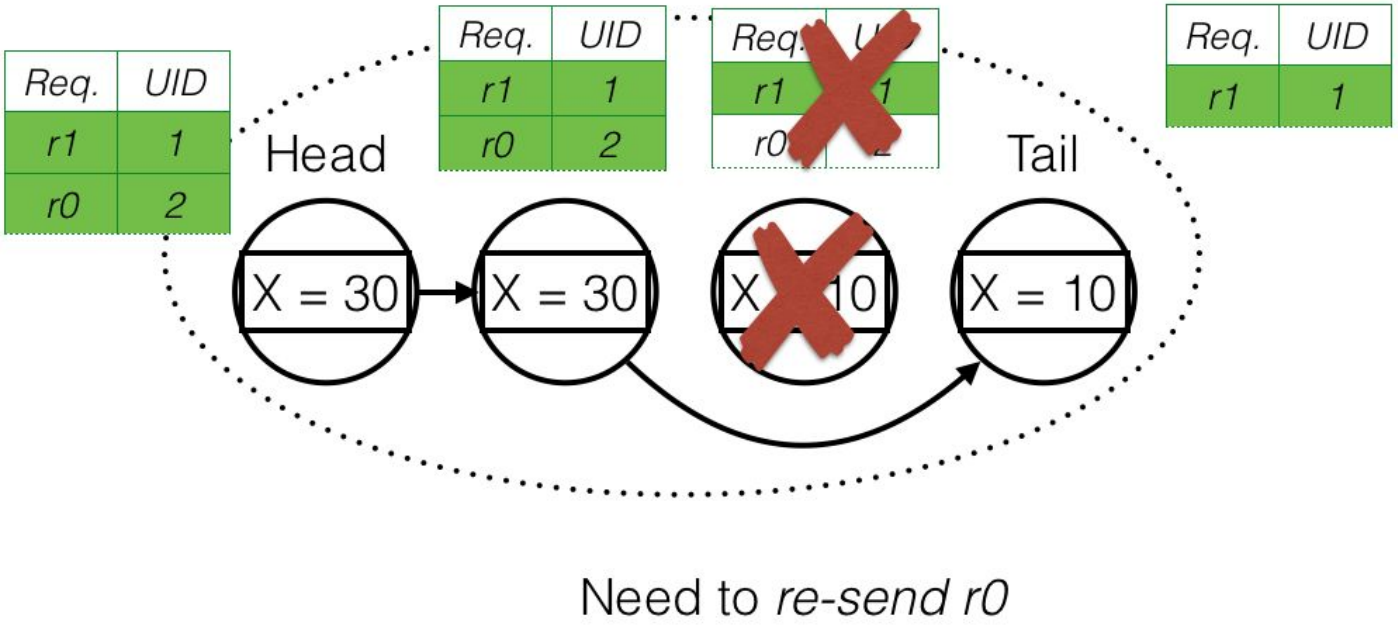
2nd replica now becomes head.

Fault Tolerance: Tail



2nd last replica now becomes tail.

Fault Tolerance: Replica in the Middle

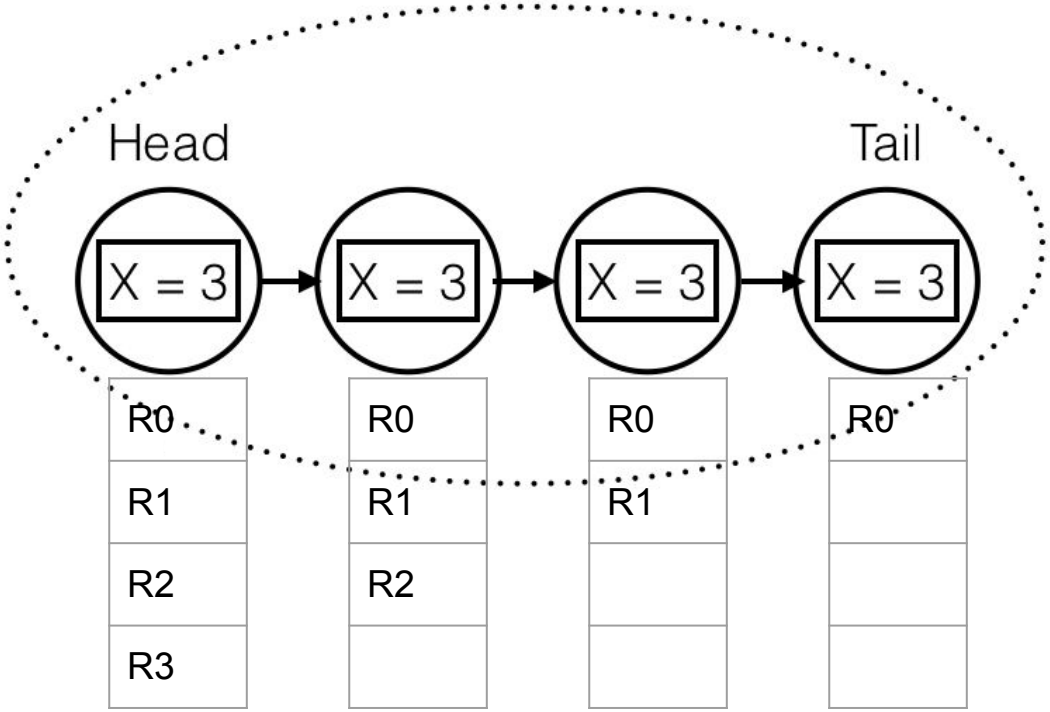


Connect the predecessor of the failed node to the successor of the failed node.

Design Goal

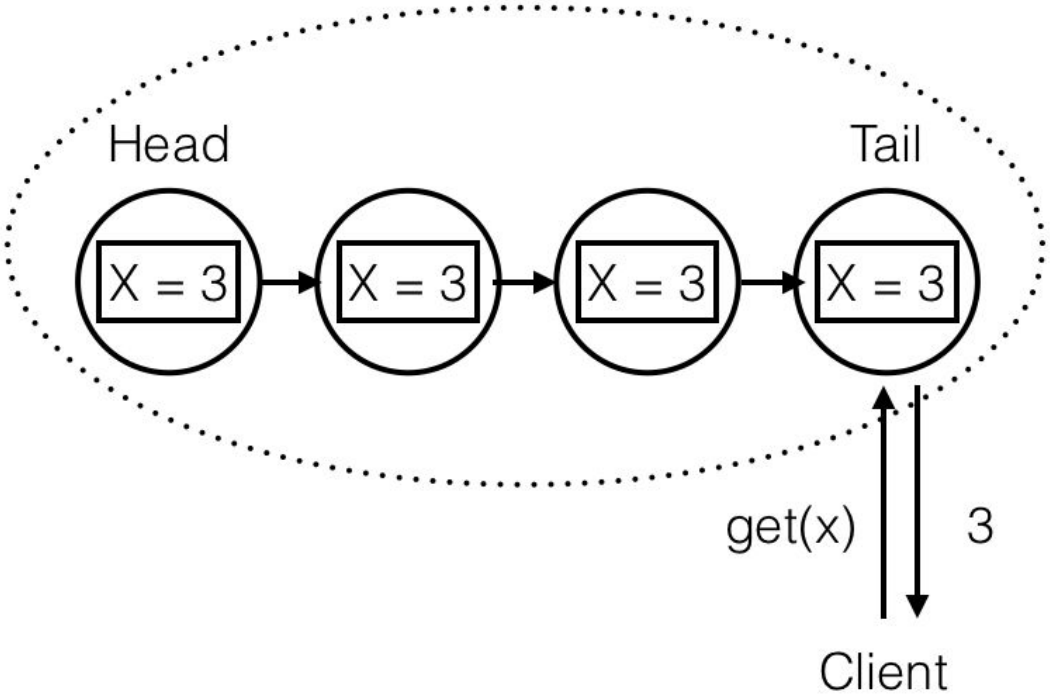
Is the design achieve high throughput, strong consistency and high availability at the same time?

Design Goal: High Throughput

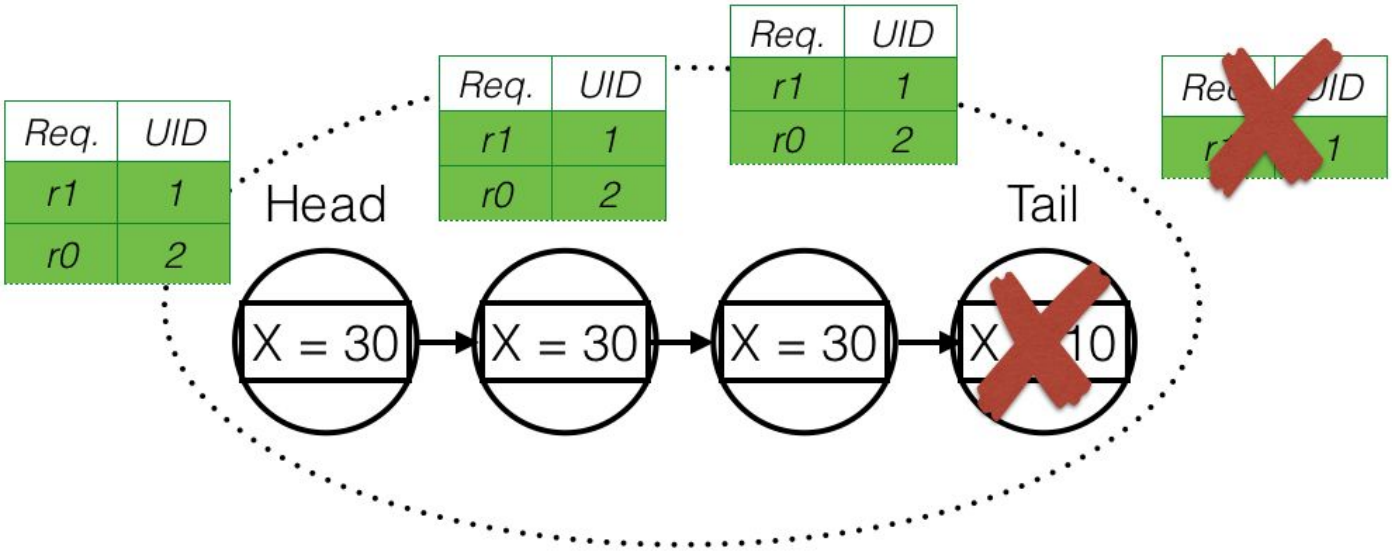


Requests can be pipelined

Design Goal: Consistency



Design Goal: High Availability



Worst failure case: tail failure. Service unavailable for 2 message delays (Notify new tail that it has became tail and notify client of the new tail).

Trade off?

- Latency
- The assumption of reliable master service.

CR’s connection to State Machine Approach

- State Machine Approach provided 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.
- A key contribution that bridges the gap between academia and practicality for SMR.

The End & Acknowledgements