CS 582: Distributed Systems

Spanner: Google's Globally Distributed Database

Dr. Zafar Ayyub Qazi

Fall 2024

ACK: Slides use some material from Robert Morris (MIT) and Wilson Hsieh (Google)

Google Spanner: Specific Learning Outcomes

By the end of today's lecture, you should be able to:

- ❑ Explain the difference between *operations* and *transactions*
- ❑ Explain *serializability* and *external consistency*
- ❑ Explain the design requirements of Google Spanner
- ❑ Explain the motivating application use case for Google Spanner
- ❑ Explain how *Read/Write transactions* are handled in Google Spanner
- ❑ Explain how *Read-Only transactions* are handled in Spanner
  - ❑ Understand the concepts of Snapshot Isolation and TrueTime
- ❑ Analyze the design choice made in Google Spanner and how they can provide serializability and external consistency as well as how they impact performance to execute transactions
- ❑ Analyze alternative design choices for Google Spanner and their impact on consistency and performance

Why Spanner?

- A rare example of a global-scale distributed system which can provide strong consistency
  - Specifically external consistency
    - Which is similar to linearizability
    - Highly desirable for several applications
  - In contrast to Dynamo and Memcache@Facebook
- Neat ideas
  - 2-PC over Paxos groups
  - Synchronized time for fast reads
- Used a lot inside Google
  - Google had also made it available as a product service for Cloud platforms

What was the motivating use case?

- A database for Google's ad bussiness
- Strong consistency was required
  - External consistency (linearizability)
- Workload was dominated by reads
  - Wanted to make them fast
- Support distributed transactions

Transactions vs Operations

- Transactions: a unit of work
  - May consist of multiple operations; reads and/or writes
- Example1: Read/Write (R/W) Transaction
  - Bank transfer from Y to X
    - BEGIN
      - X= X+1
      - Y= Y-1
    - END
- Example2: Read-Only (R/O) Transactions
    - BEGIN
      - Print X,Y
    - END

Distributed Transactions

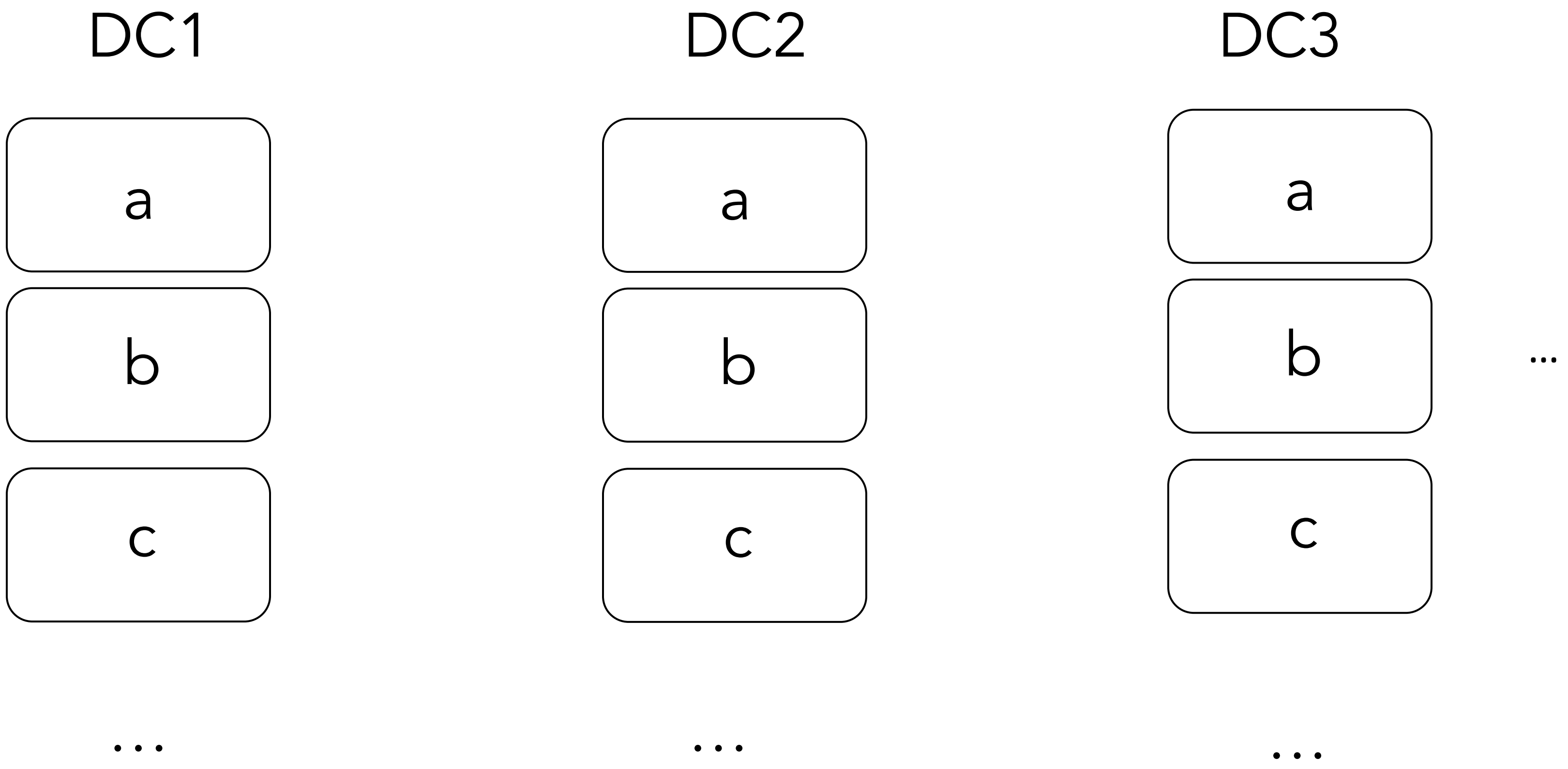
Important Required Property

- Atomicity: Transactions must commit or abort as a single atomic unit
  - BEGIN
    - X= X+1
    - Y= Y-1
  - END
- When commit, all updates performed on database are made permanent, visible to other transactions
- When abort, database restored to a state such that the aborting transaction never executed

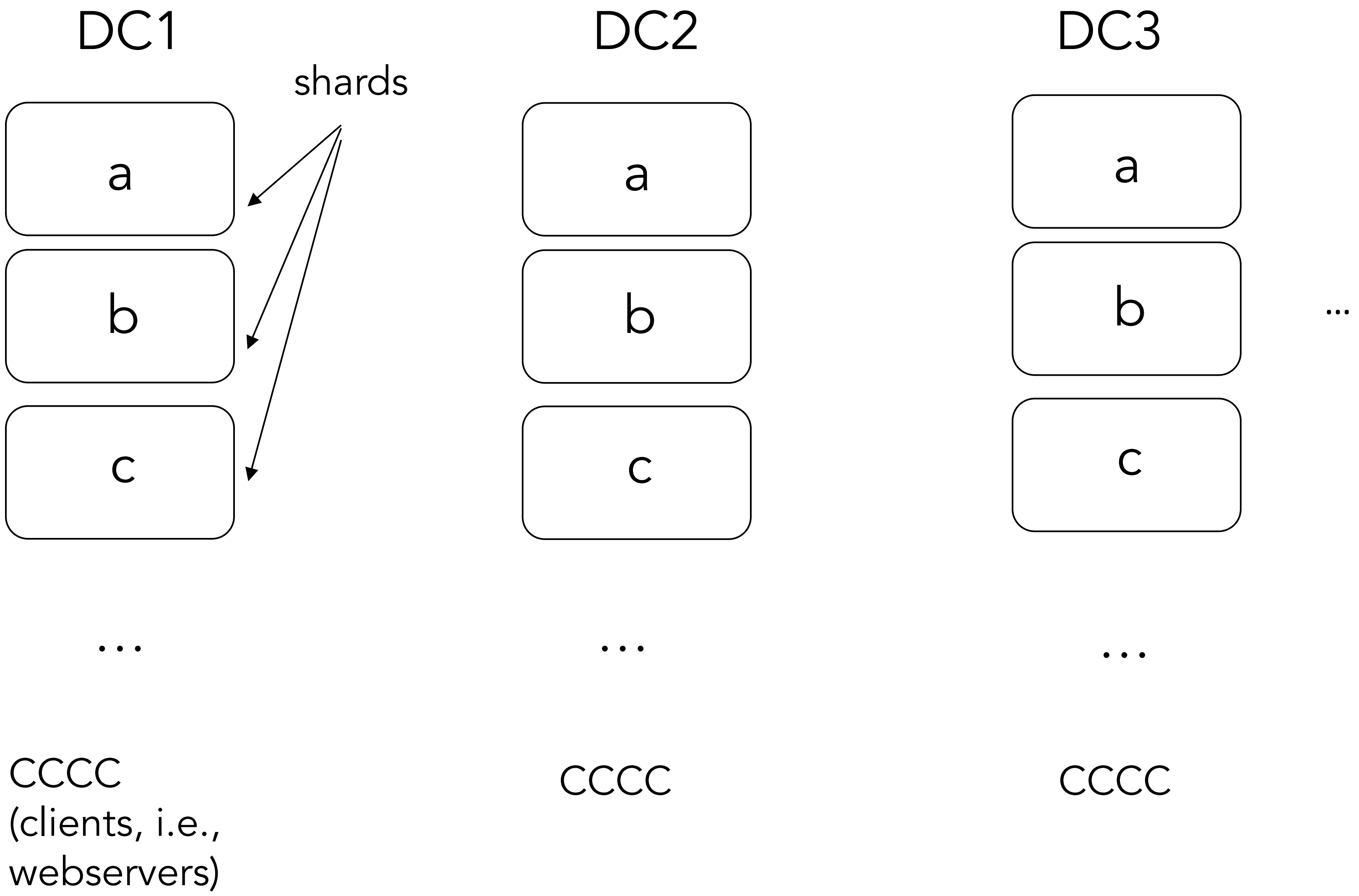
Spanner's Consistency Properties

- Even if the system handles concurrent transactions, their results must be serializable
  - Same results as if transactions executed one-by-one
    - There must be a serialization order
- If a transaction T2 starts after a transaction T1 has committed, T2 must see T1's writes
  - After refers to "wall-clock" time
  - Called External consistency (similar to linearizability)

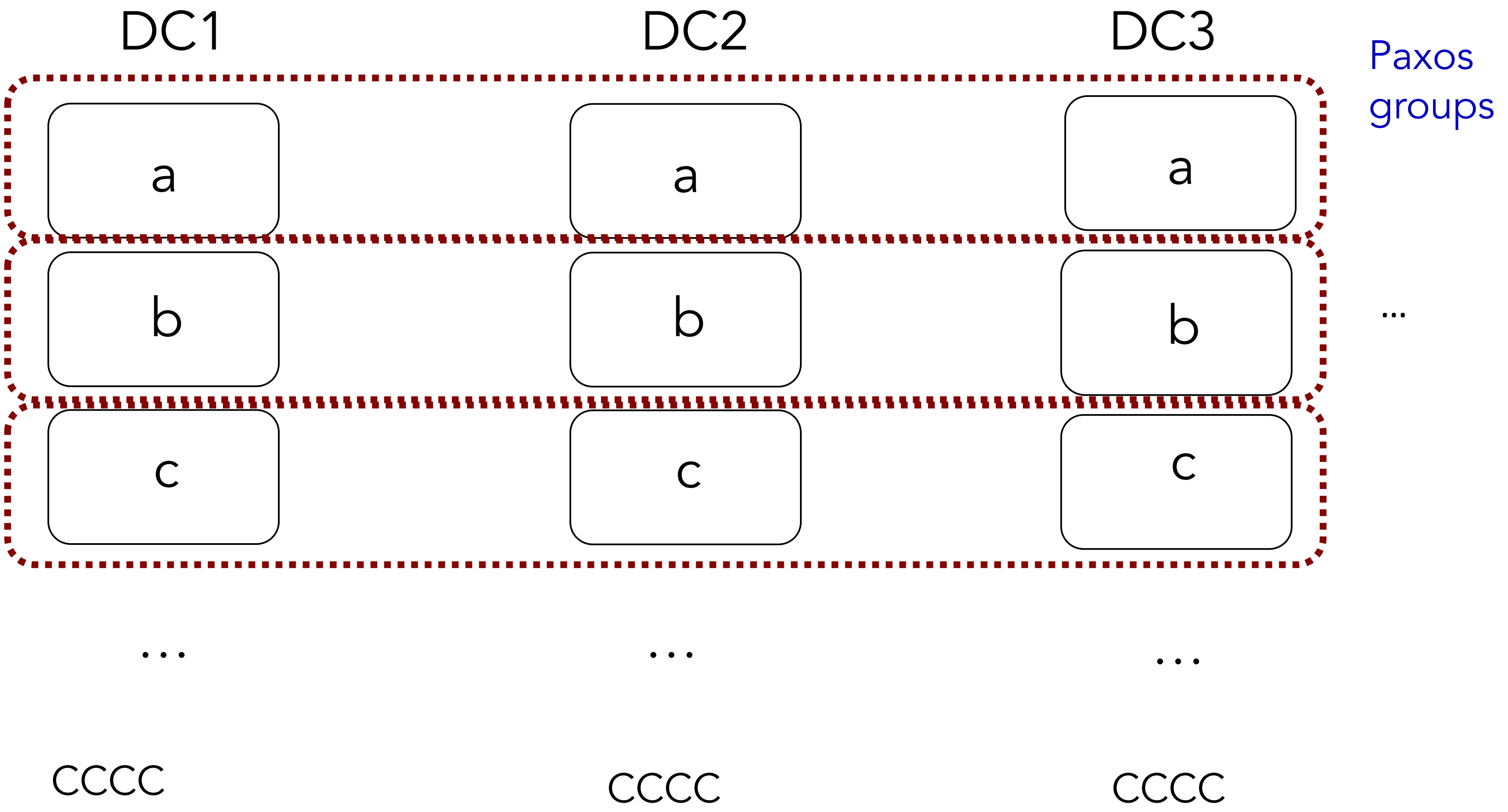
Spanner: Basic arrangement of servers



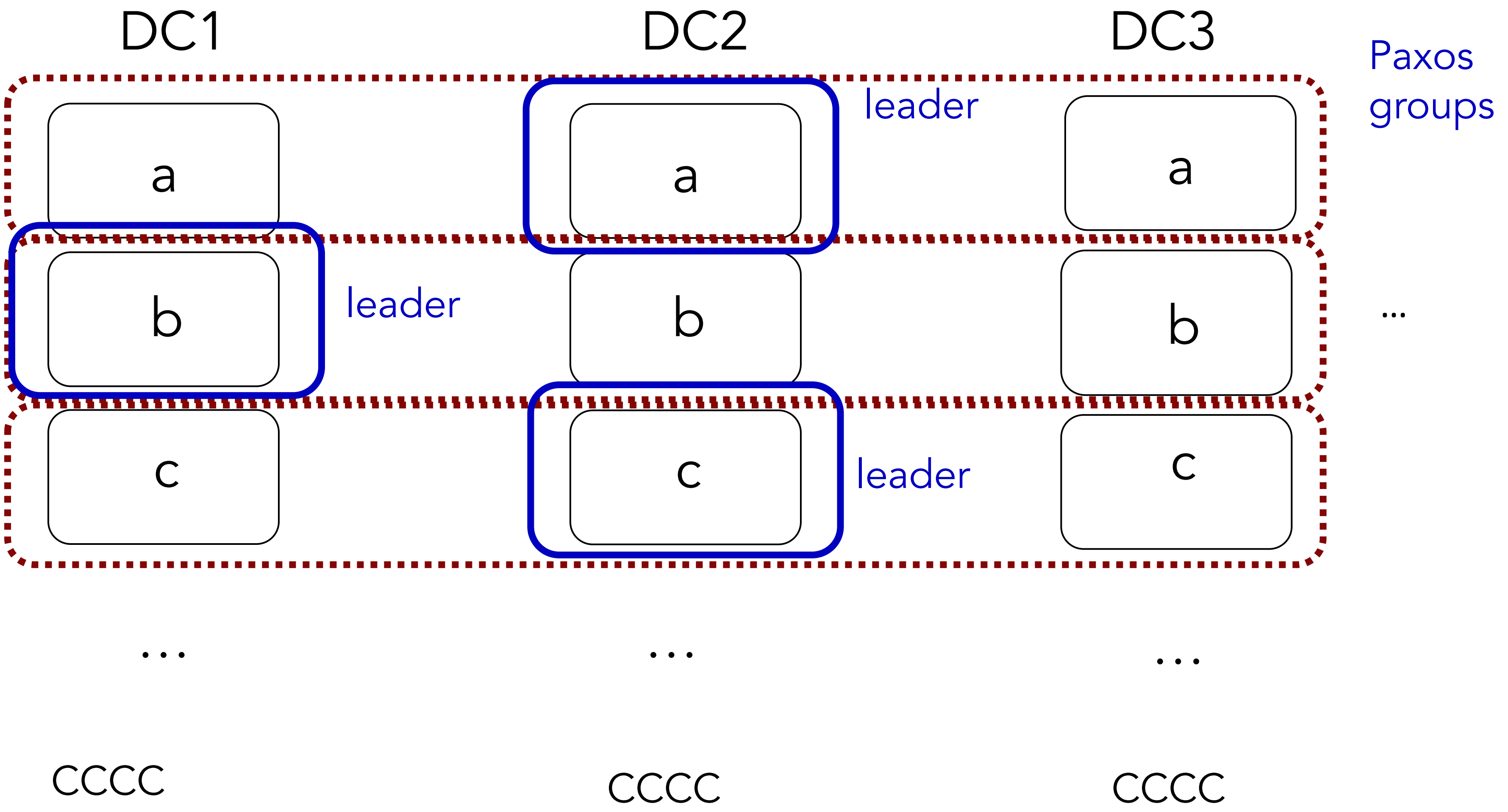
Spanner: Basic arrangement of servers



Spanner: Basic arrangement of servers



Spanner: Basic arrangement of servers



Why this arrangement?

- Why Paxos?
  - Only requires a majority of nodes
    - In this case, requires majority of DCs to be responding (fast)
- Why separate Paxos groups?
  - For high throughput – handle large number of requests in parallel
- Why multiple data centers?
  - Tolerate data center failures
  - Serve data from closeby datacenters

A Couple of Important Requirements

- They want to make R/O transactions fast
  - Allow clients to be able to read data from local DCs
  - Challenge: Local DC replica maybe a minority node, and may not have latest data & violate external consistency

- Support distributed transactions
  - Transactions that span multiple shards
  - Challenge: requires coordination among multiple Paxos groups

Spanner Overview

- Treats R/W and R/O transactions differently
- R/W Transaction handling
  - 2PC (with locks) over Paxos Group
- R/O Transaction handling (faster)
  - Snapshot Isolation
  - Time synchronization (via TrueTime API)

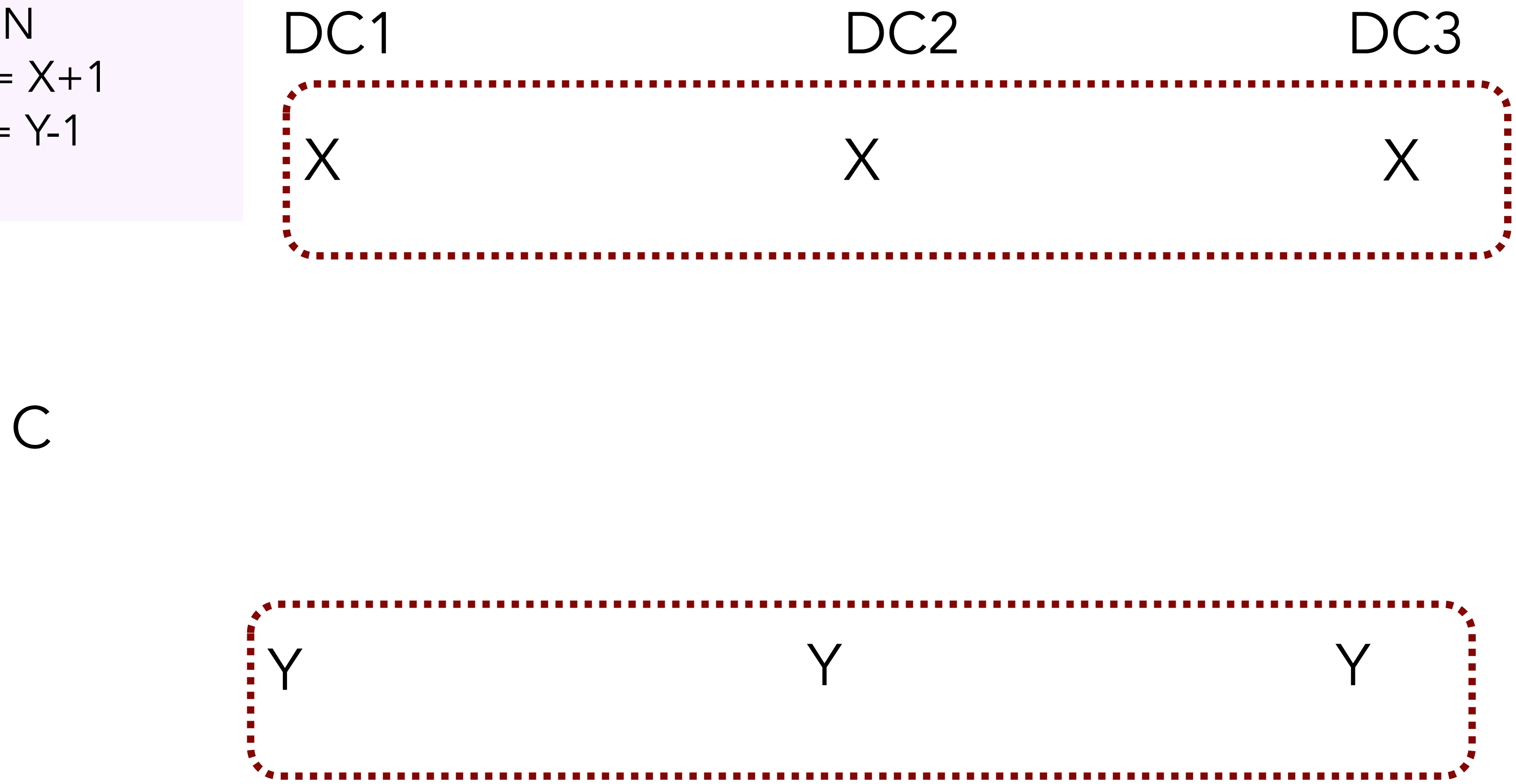
How are R/W Transactions handled?

- Lets understand through an example

How are R/W Transactions handled?

X & Y stored on different shards

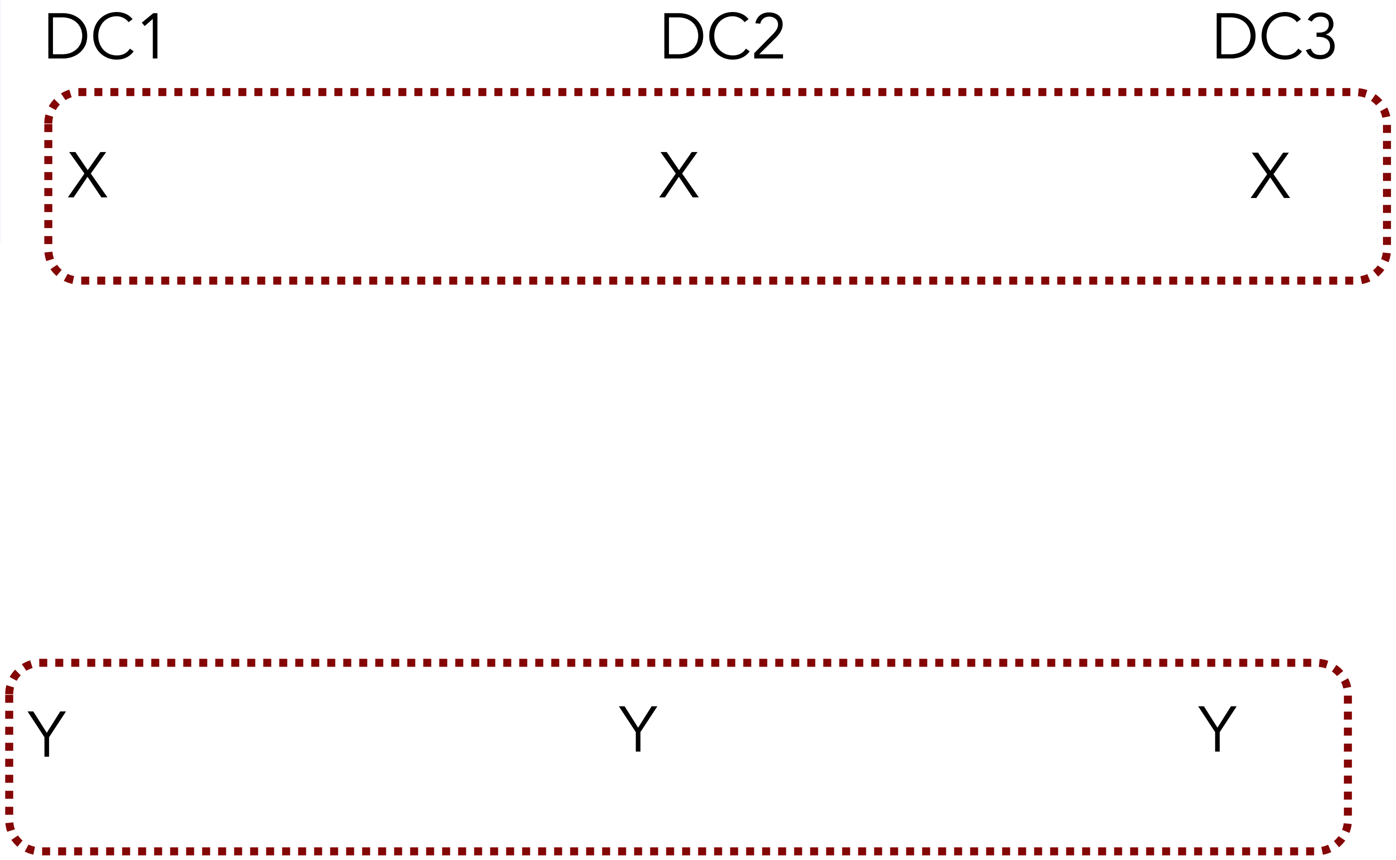
```
BEGIN
    X= X+1
    Y= Y-1
END
```



How are R/W Transactions handled?

X & Y stored on different shards

BEGIN
X= X+1
Y= Y-1
END



C

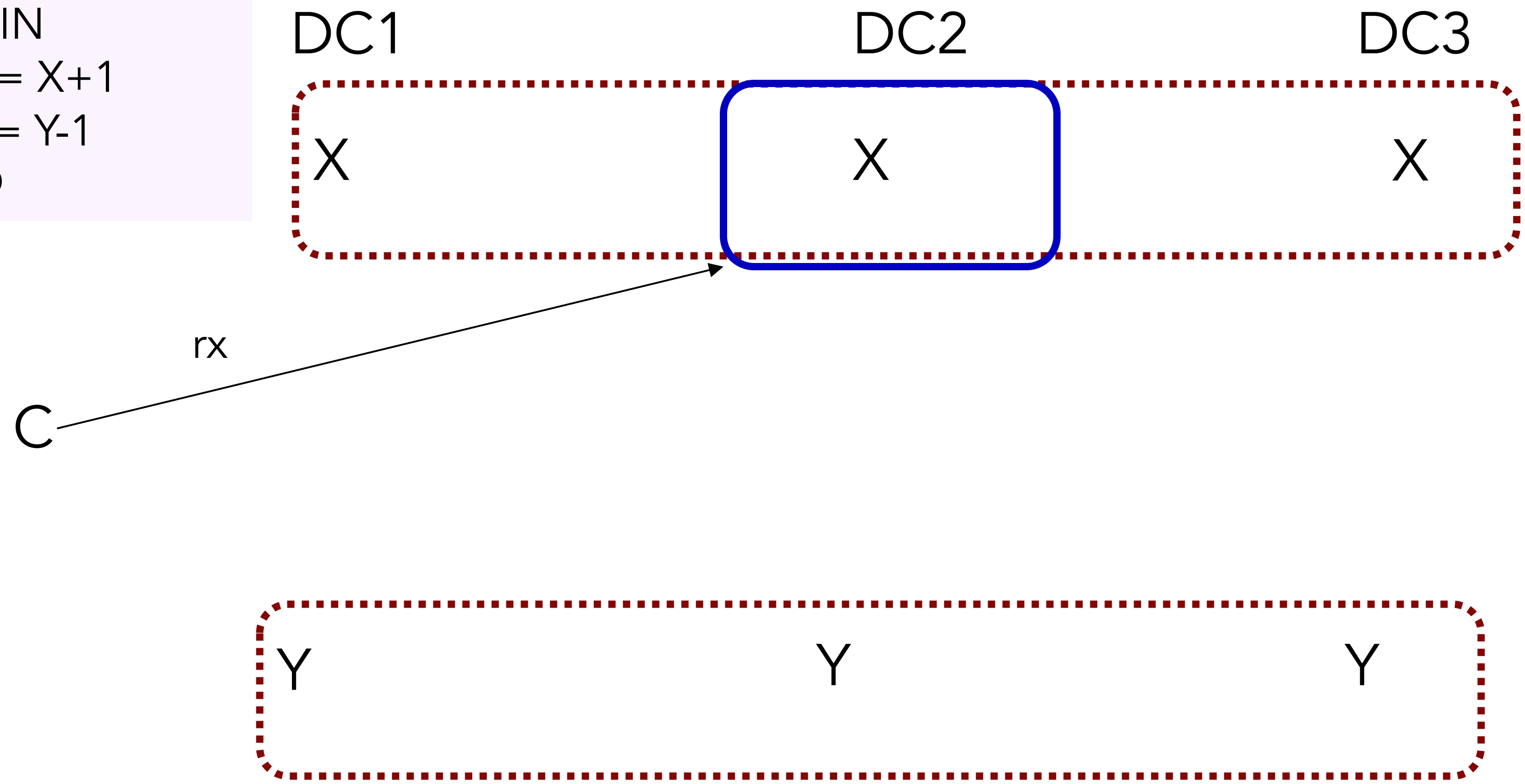
Client assigns a unique id for the transaction

And first performs the reads

How are R/W Transactions handled?

X & Y stored on different shards

```
BEGIN
    X= X+1
    Y= Y-1
END
```

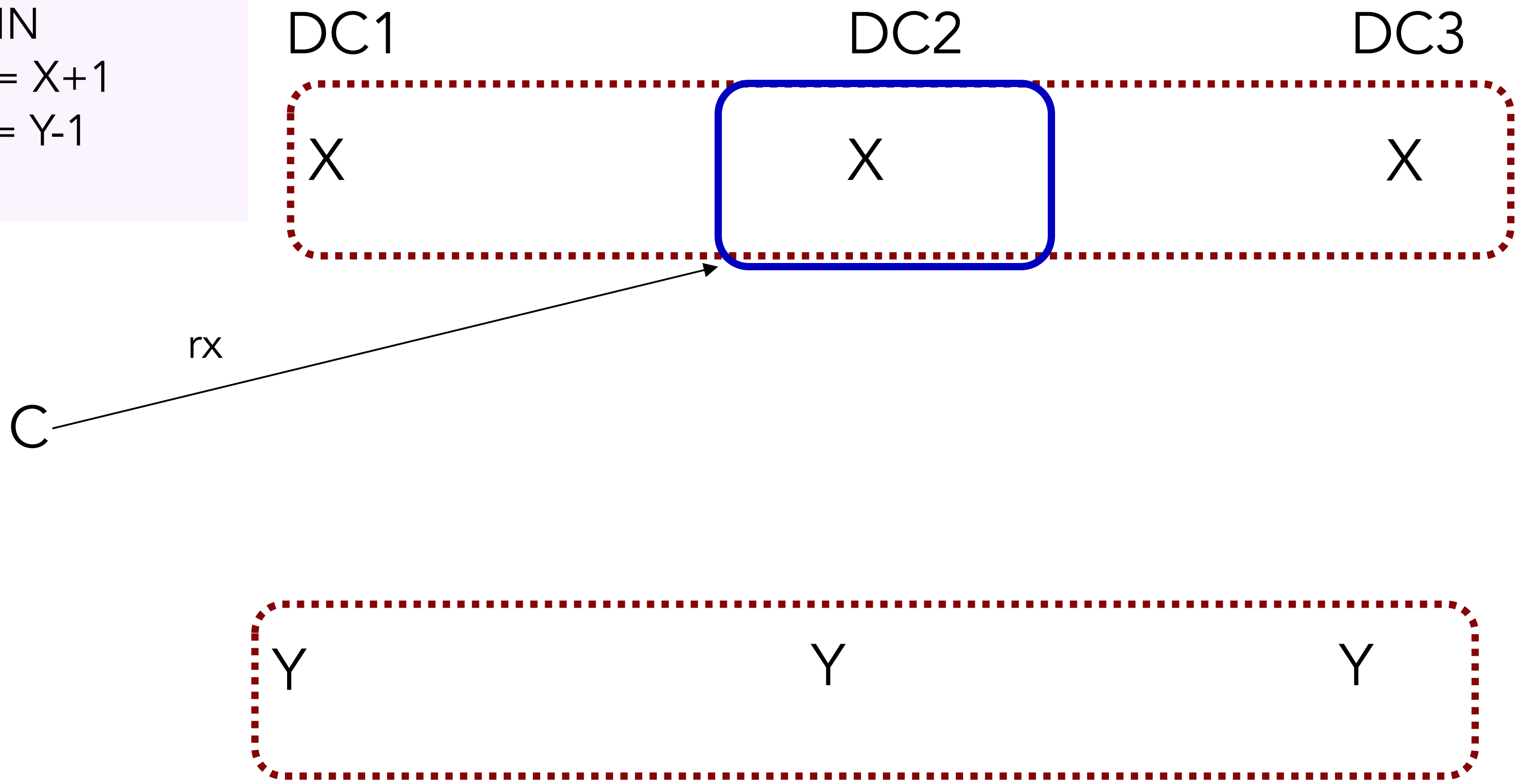


rx = read X
wx = Write to X

How are R/W Transactions handled?

X & Y stored on different shards

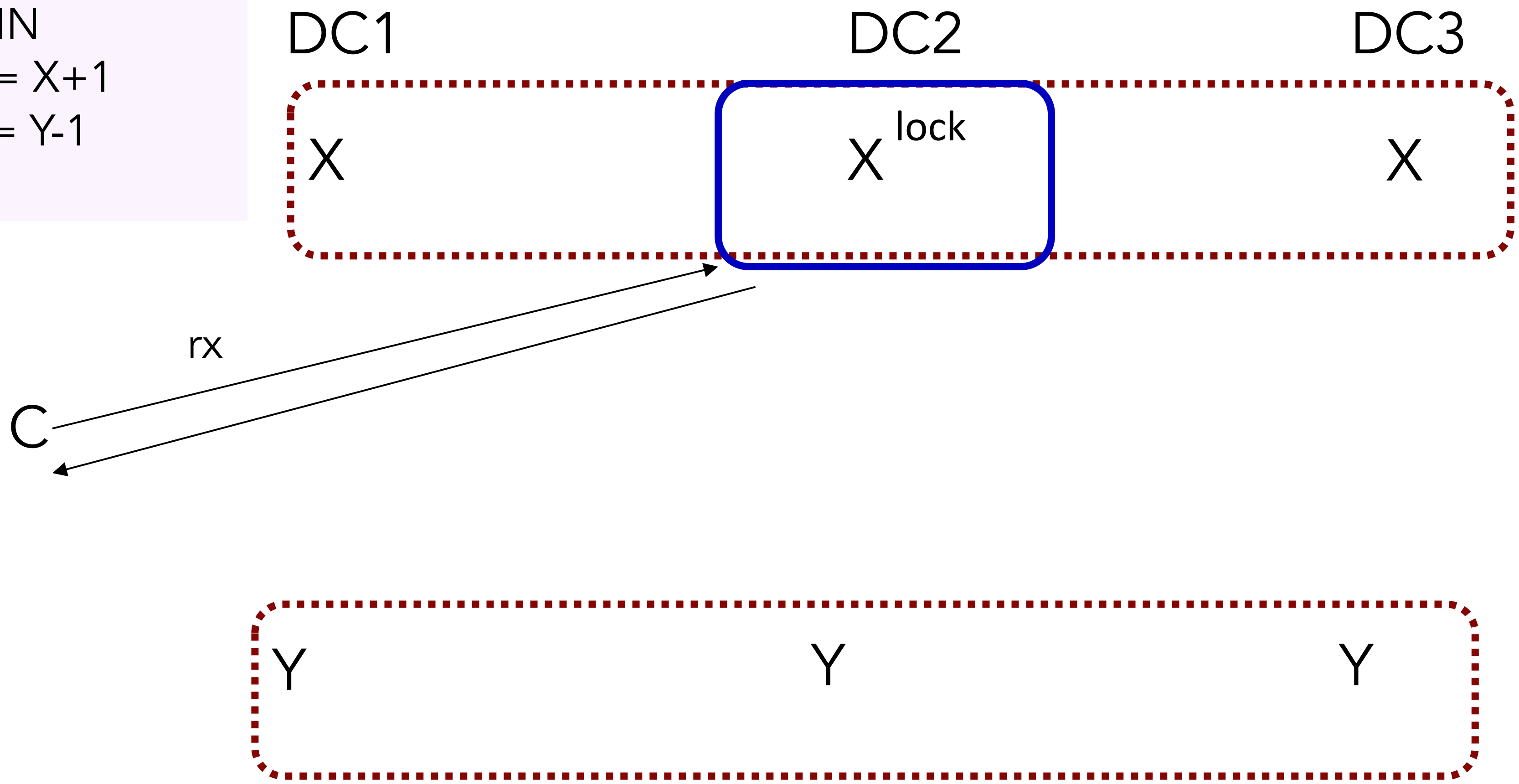
```
BEGIN
    X= X+1
    Y= Y-1
END
```



How are R/W Transactions handled?

X & Y stored on different shards

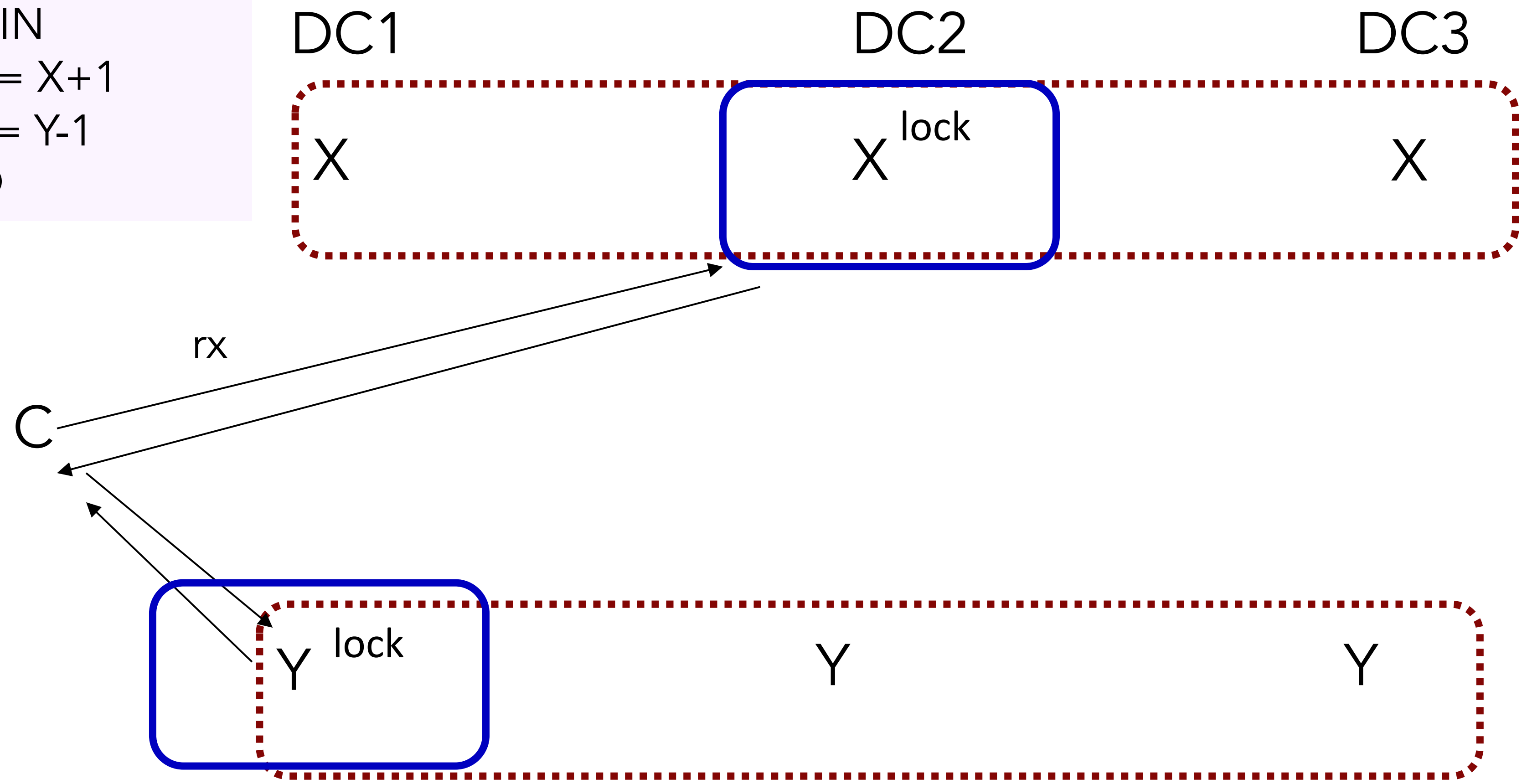
```
BEGIN
    X= X+1
    Y= Y-1
END
```



How are R/W Transactions handled?

X & Y stored on different shards

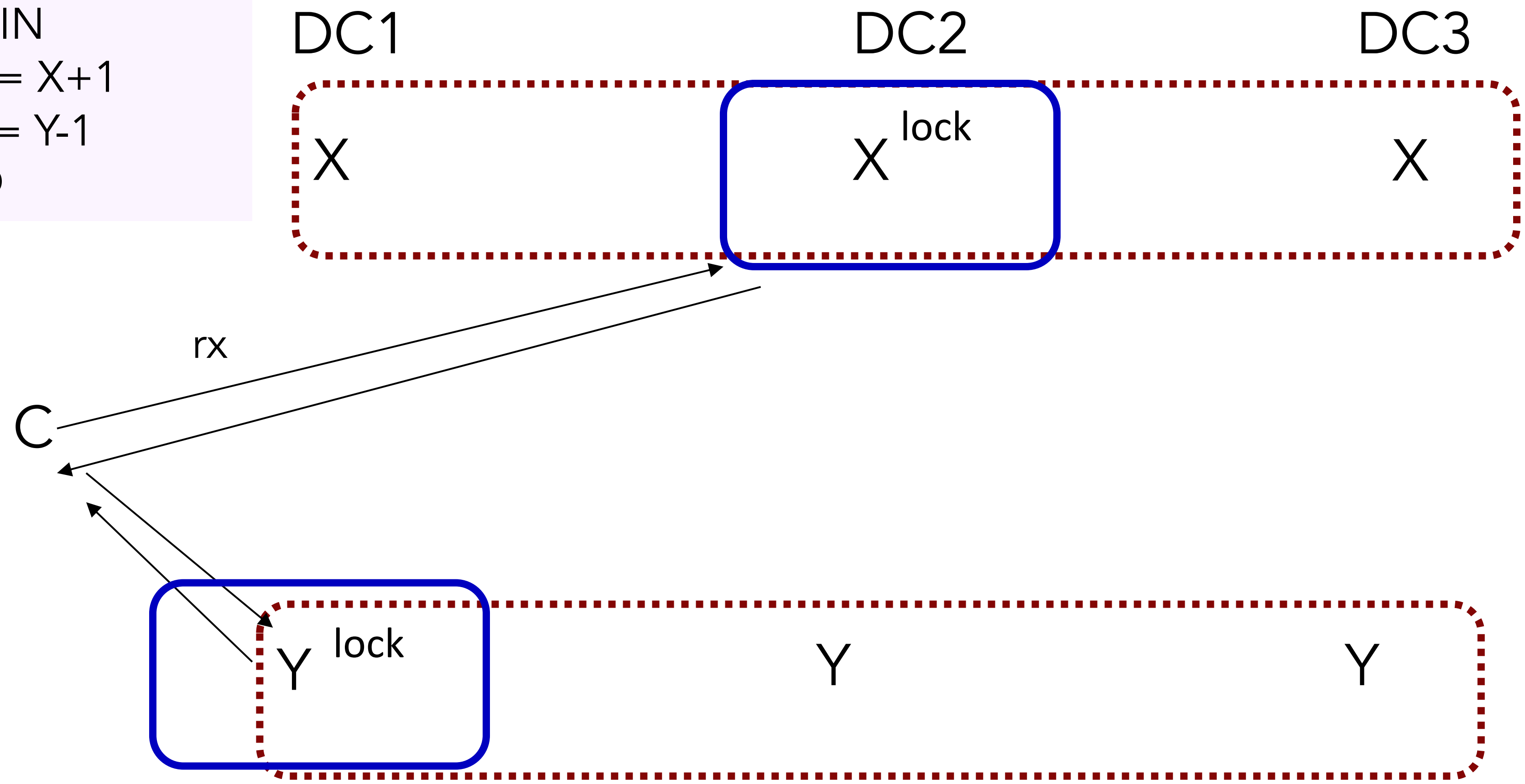
```
BEGIN
    X= X+1
    Y= Y-1
END
```



How are R/W Transactions handled?

X & Y stored on different shards

```
BEGIN
    X= X+1
    Y= Y-1
END
```

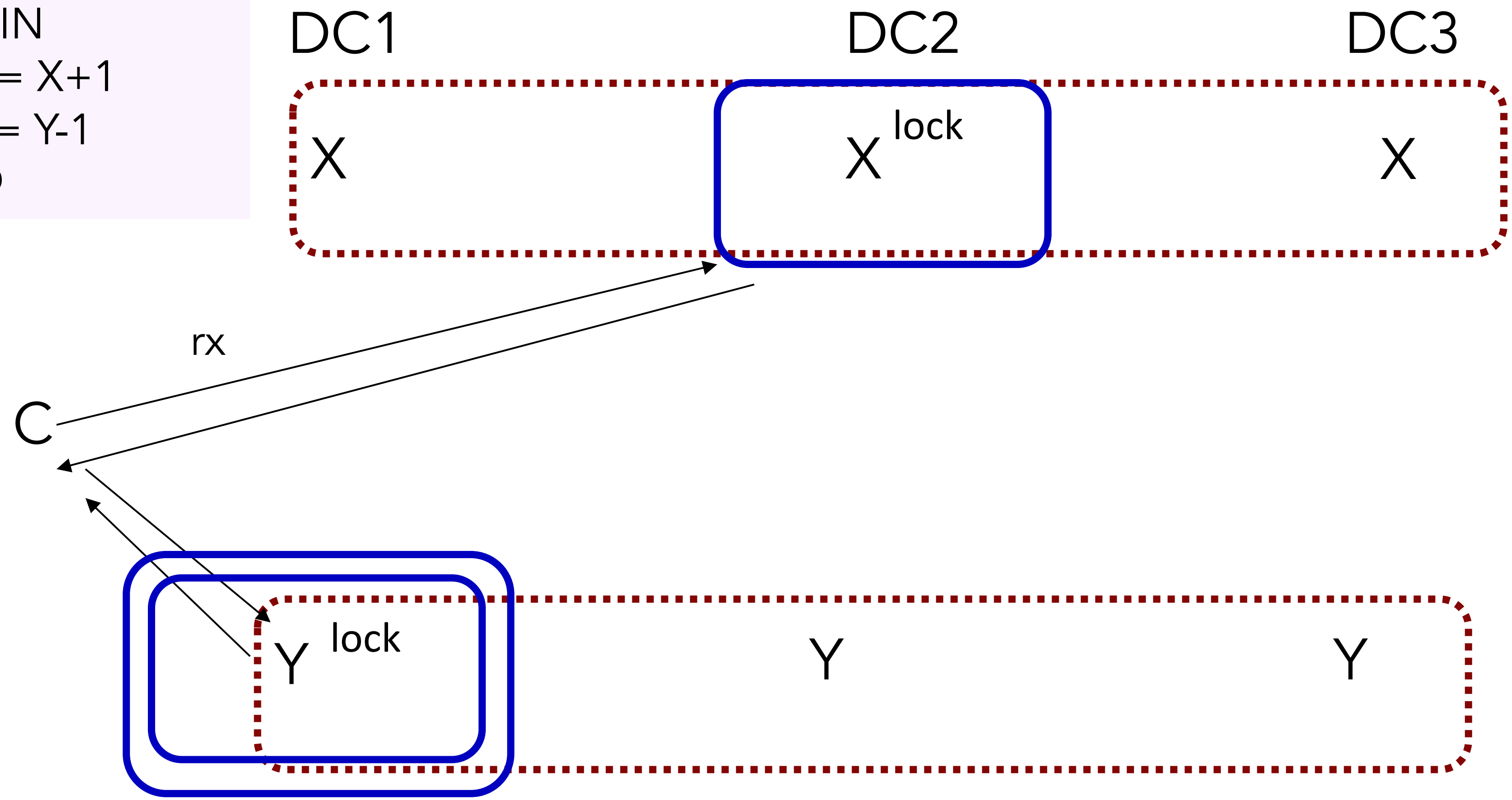


Client chooses one of the Paxos group as the 2-PC Coordinator, this is called the <u>Transaction Coordinator</u>

How are R/W Transactions handled?

X & Y stored on different shards

BEGIN
X= X+1
Y= Y-1
END

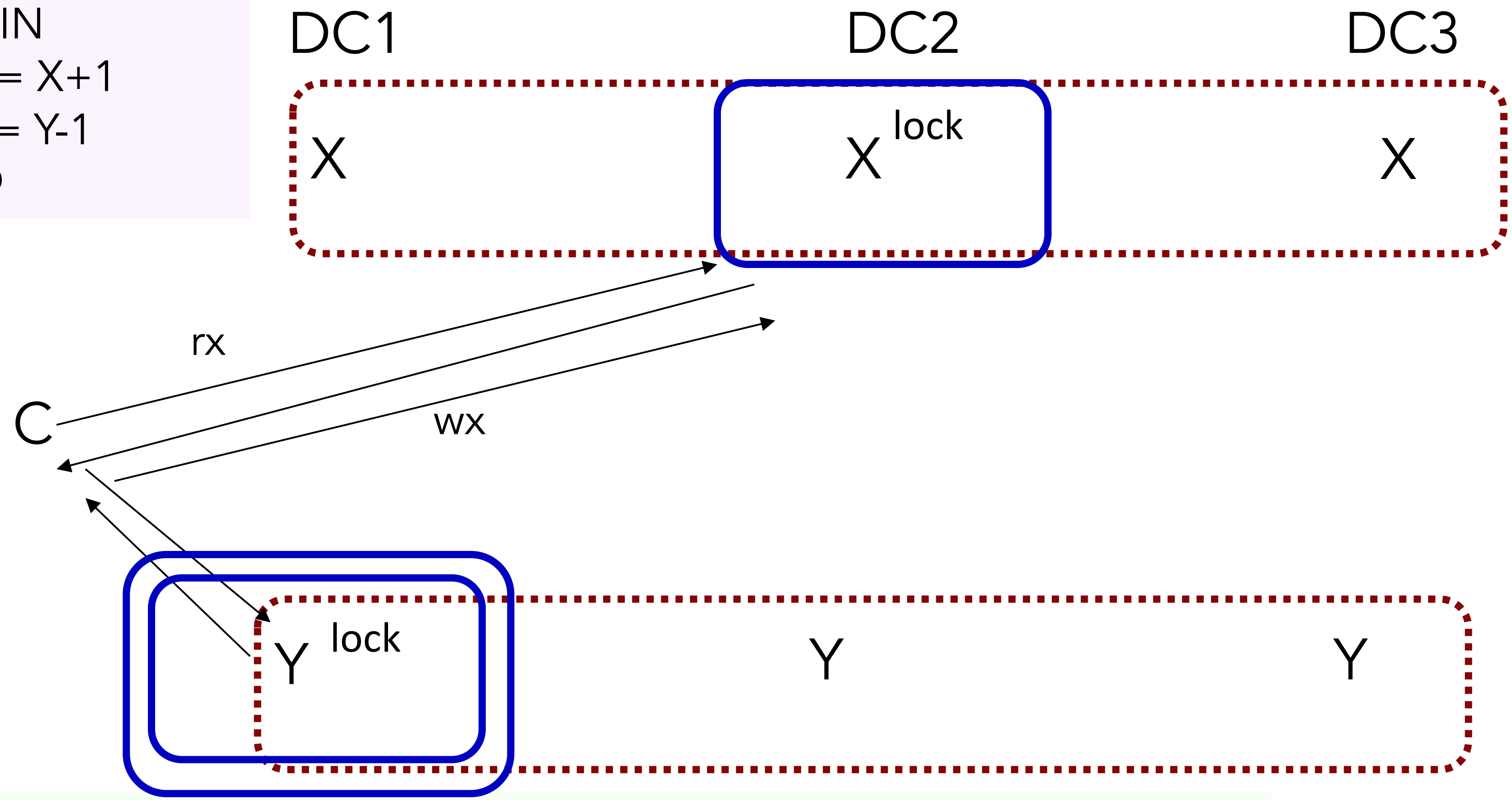


Client chooses one of the Paxos group as the 2-PC Coordinator, this is called the Transaction Coordinator (TC)

How are R/W Transactions handled?

X & Y stored on different shards

BEGIN
    X= X+1
    Y= Y-1
END

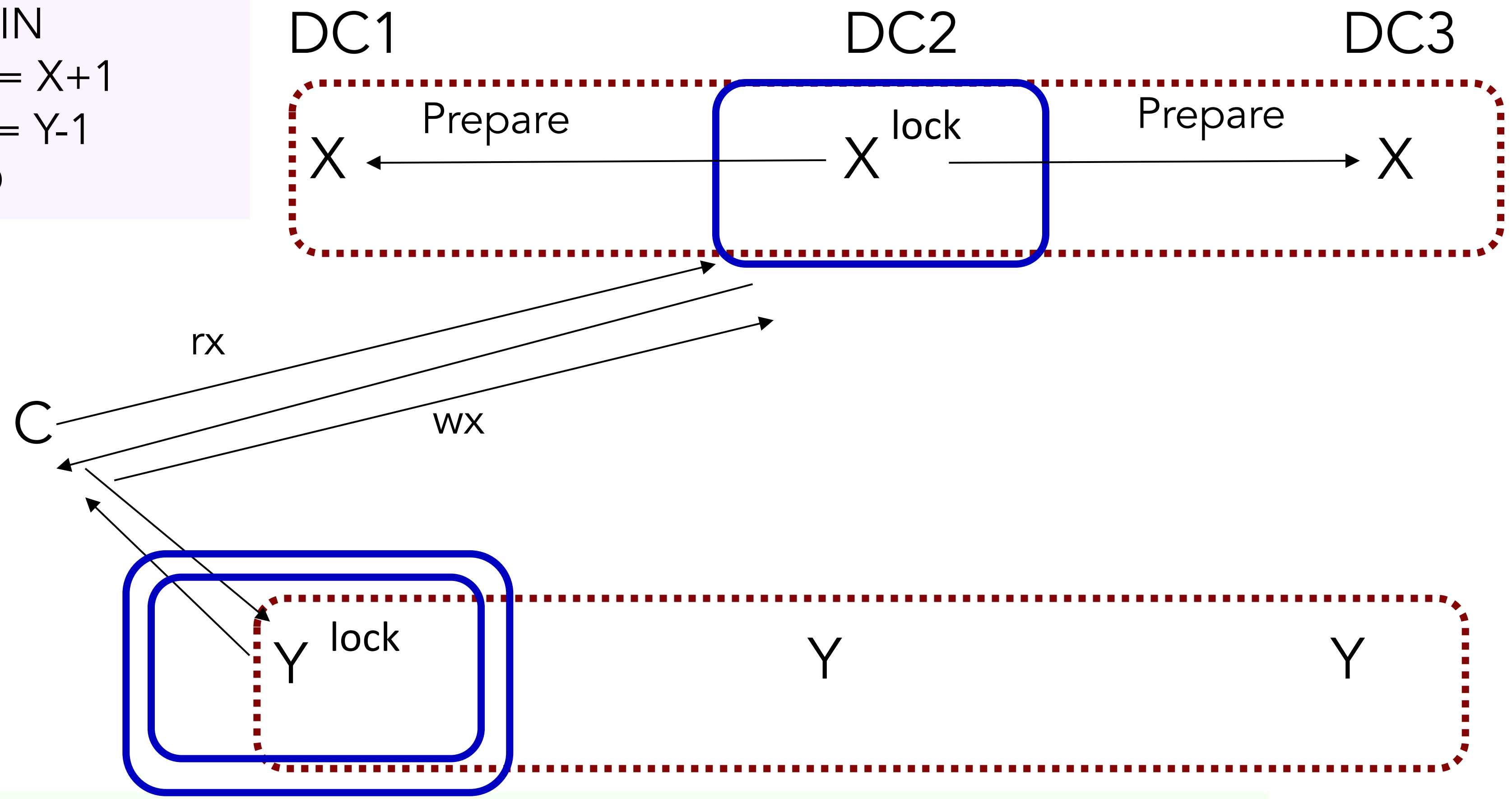


Sends the Updated values

How are R/W Transactions handled?

X & Y stored on different shards

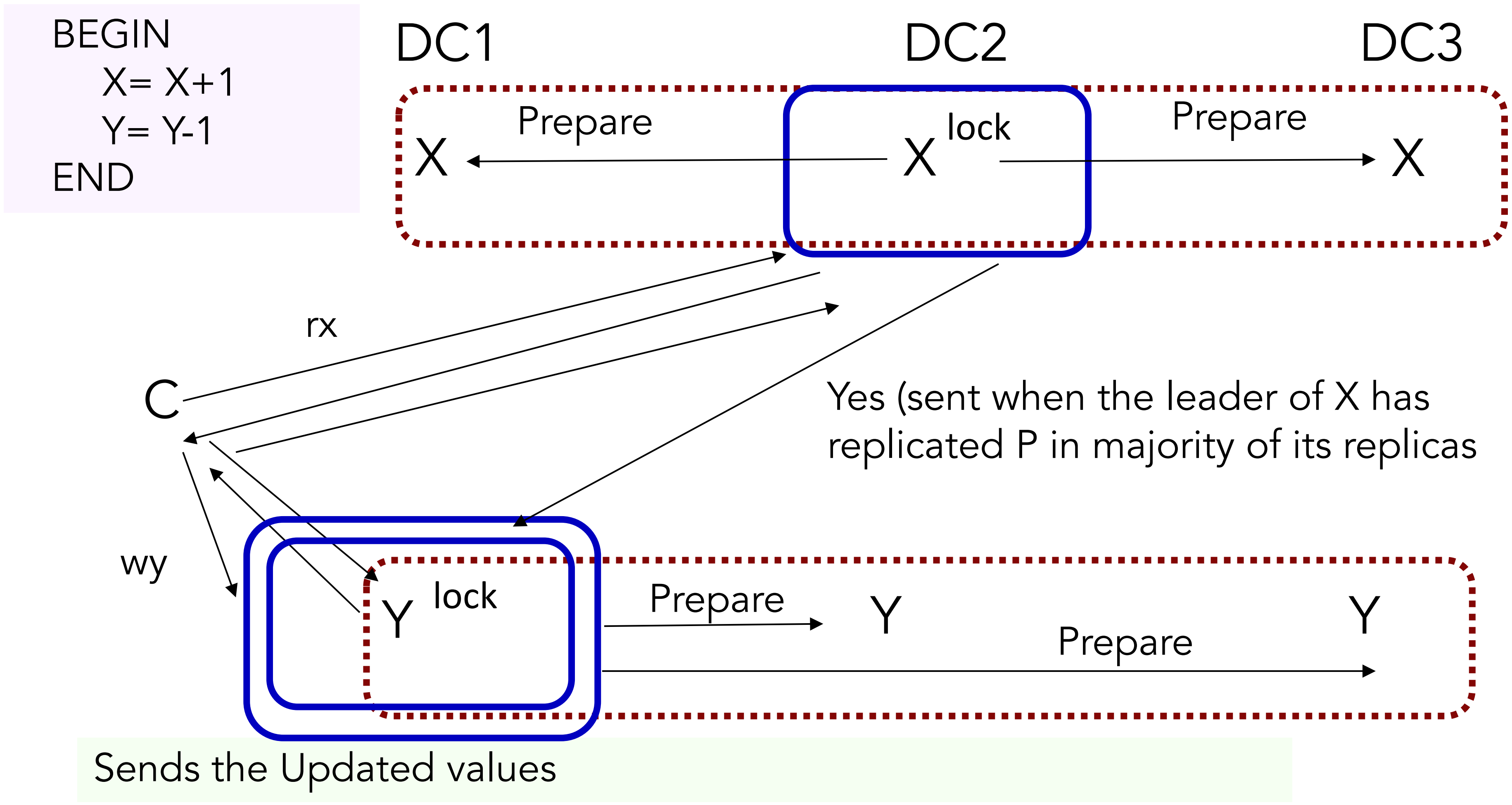
BEGIN
X= X+1
Y= Y-1
END



Sends the Updated values

How are R/W Transactions handled?

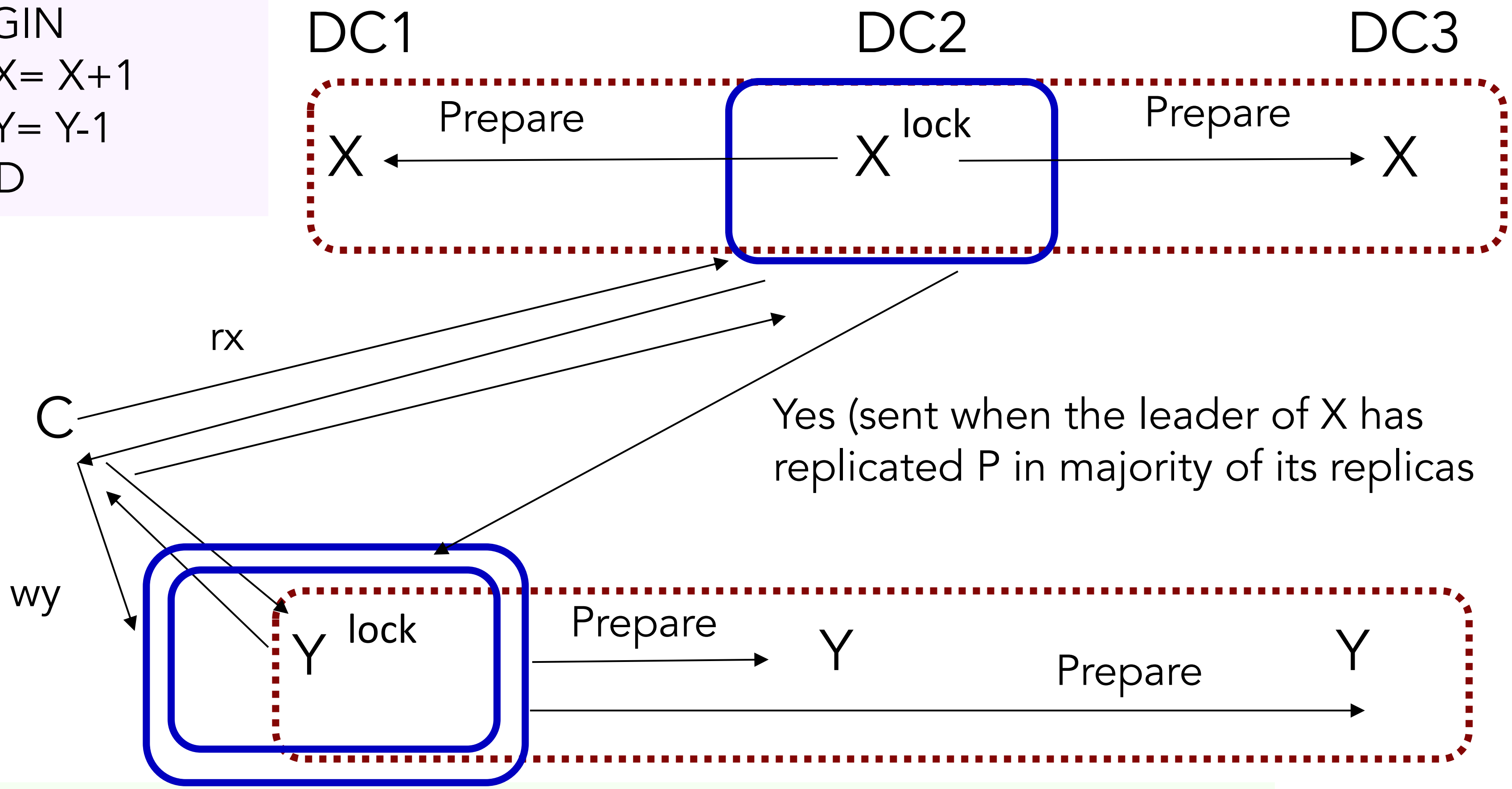
X & Y stored on different shards



How are R/W Transactions handled?

X & Y stored on different shards

```
BEGIN
    X= X+1
    Y= Y-1
END
```



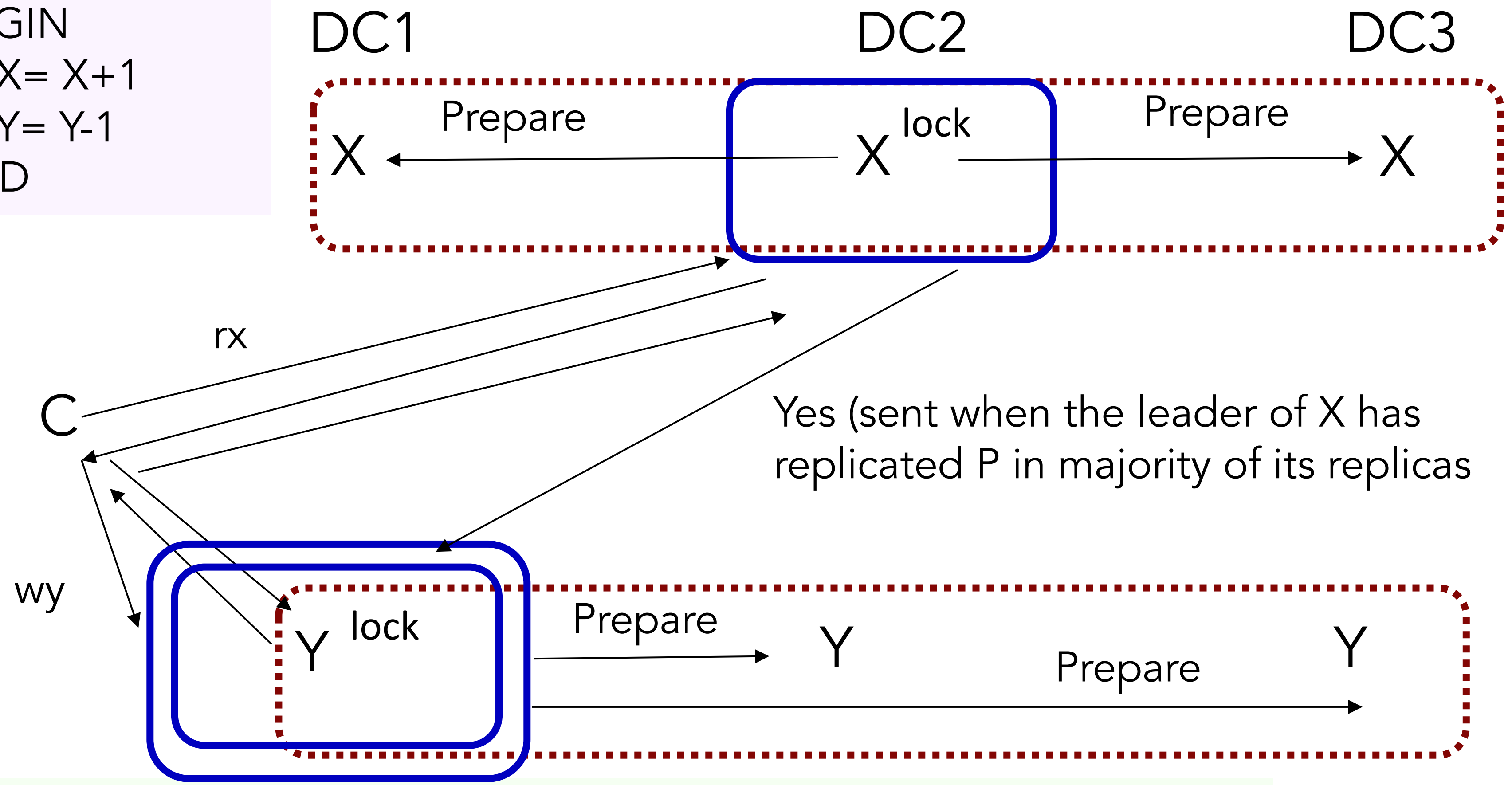
Sends the Updated values

If all the Paxos leaders (of different shards), say Yes, only then TC decides to commit

How are R/W Transactions handled?

X & Y stored on different shards

BEGIN
X= X+1
Y= Y-1
END



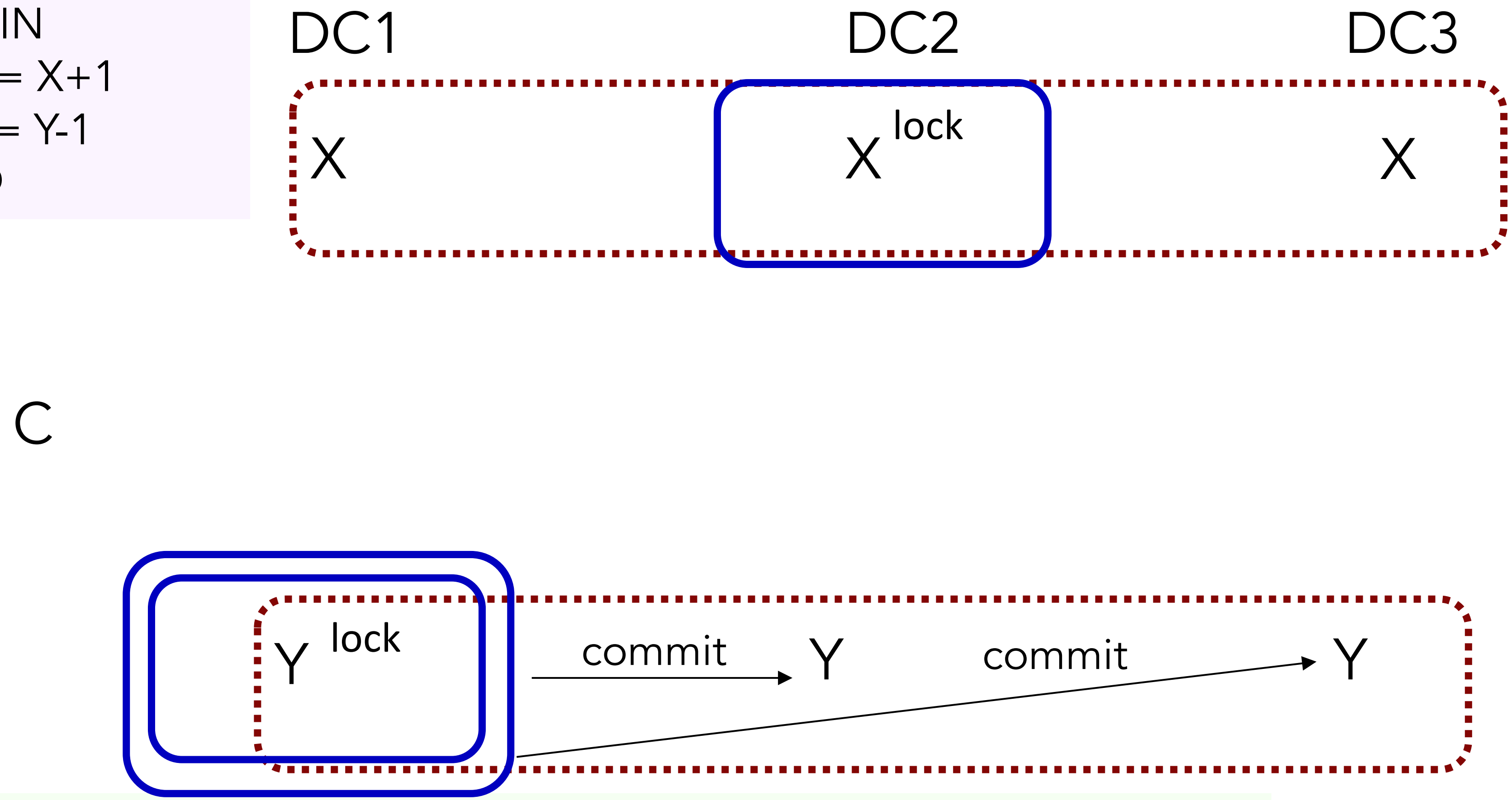
Sends the Updated values

If all the Paxos leaders (of different shards), say Yes, only then TC decides to commit

How are R/W Transactions handled?

X & Y stored on different shards

```
BEGIN
    X= X+1
    Y= Y-1
END
```



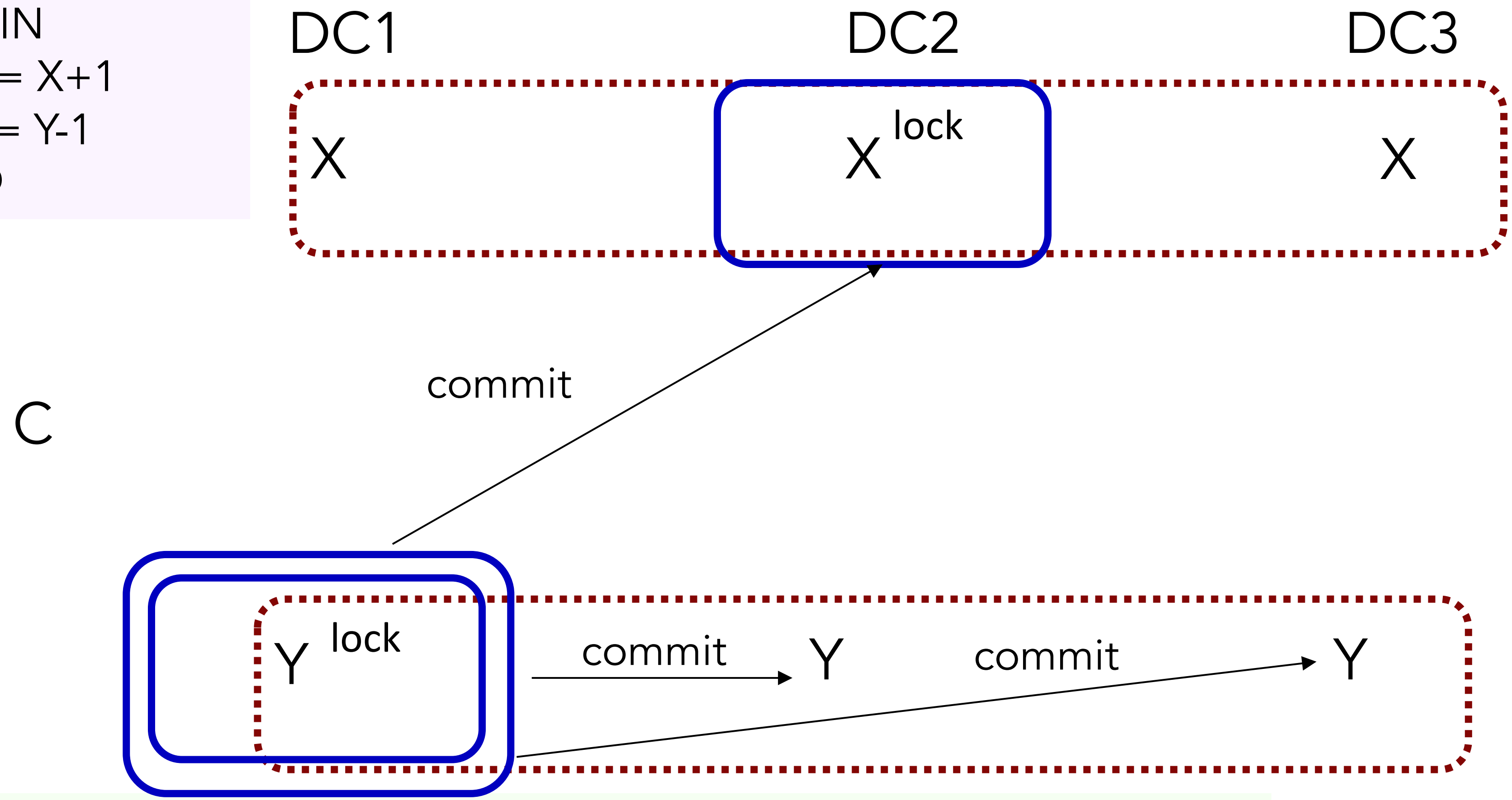
Sends the Updated values

If TC decides to commit, it first sends a commit message to its followers

How are R/W Transactions handled?

X & Y stored on different shards

```
BEGIN
    X= X+1
    Y= Y-1
END
```



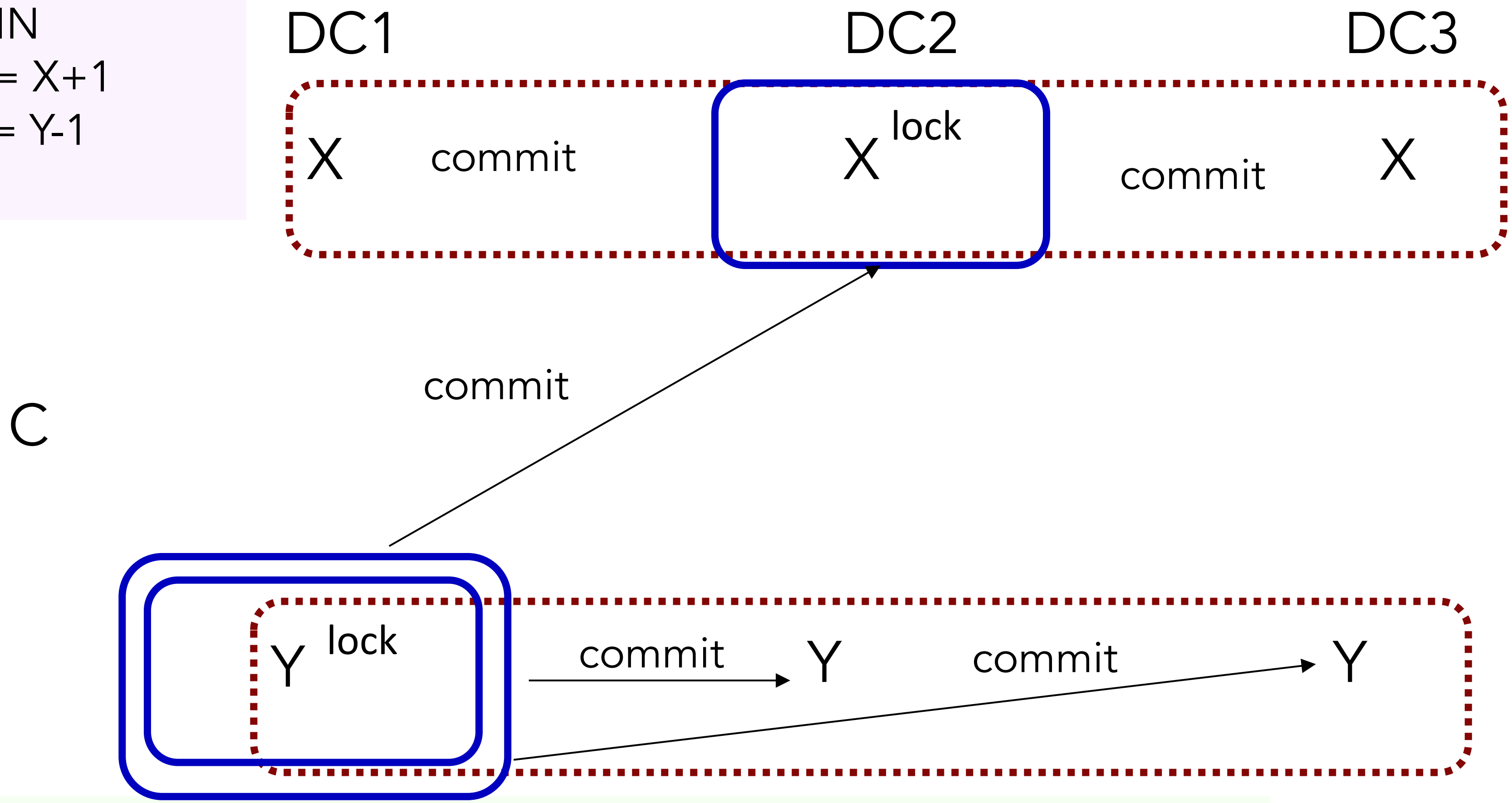
Sends the Updated values

Once a majority of nodes have replicated the log in TC's shard, it sends a commit to the other shards

How are R/W Transactions handled?

X & Y stored on different shards

```
BEGIN
    X= X+1
    Y= Y-1
END
```



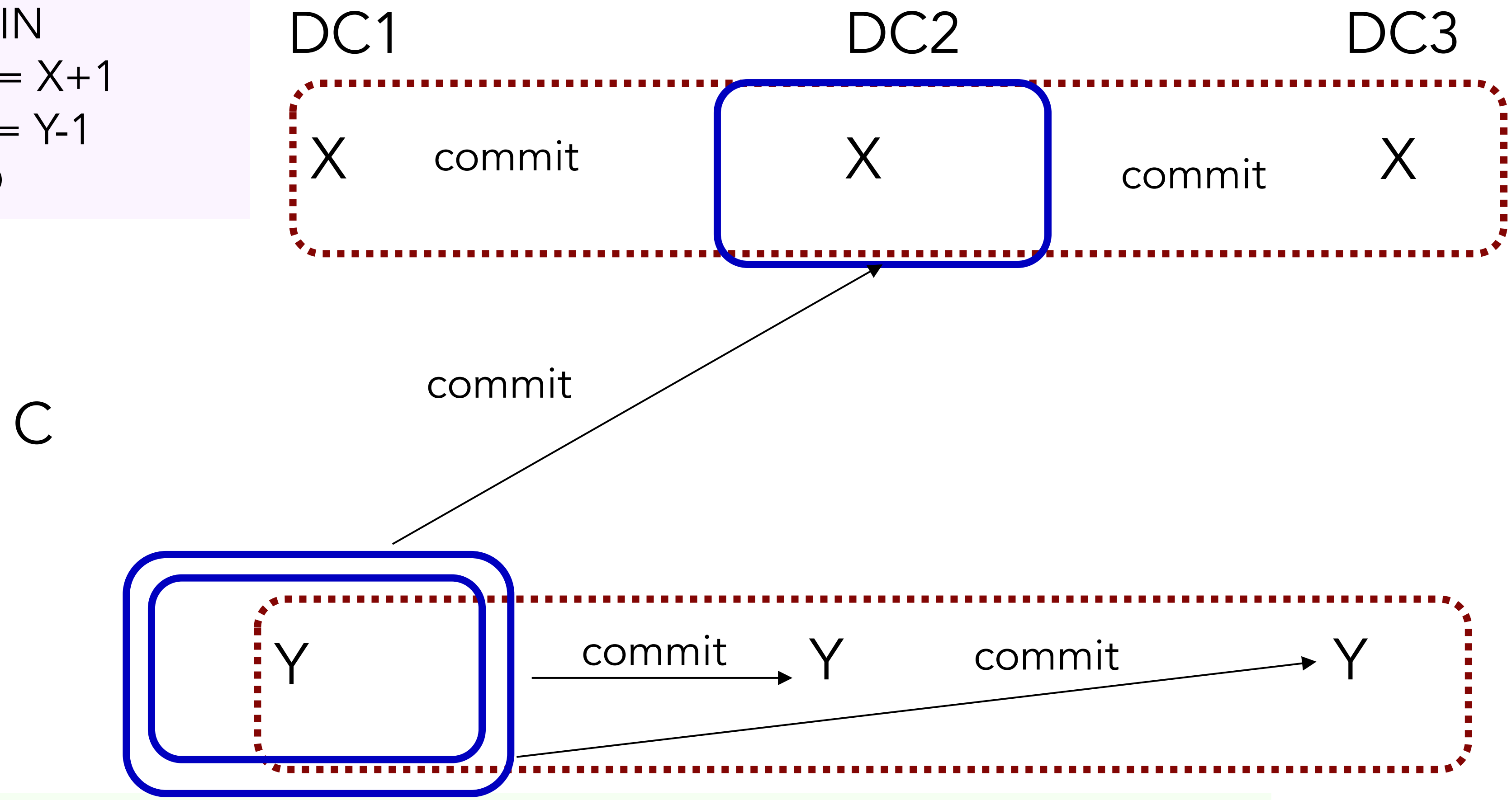
Sends the Updated values

If all the Paxos leaders (of different shards), say Yes, only then TC decides to commit

How are R/W Transactions handled?

X & Y stored on different shards

```
BEGIN
    X= X+1
    Y= Y-1
END
```



Sends the Updated values

Once commit is safely replicated on a majority, a shard leader releases the lock

Paxos Group

- Replicates shard data
- Replicates two-phase commit state

Observations about the design so far

- Locking ensures serializability
  - Two transactions conflict, one has to wait for the other
- 2PC widely hated b/c it blocks with locks held if TC fails
  - Replicating the TC with Paxos solves this problem!
- R/W transactions take a long time
  - Many inter-data-center messages.
  - Table 6 suggests about 100 ms for cross-USA r/w transaction
    - Much less for cross-city (Table 3)
  - But lots of parallelism: many clients, many shards. So total throughput could be high if busy

Next …

- R/O Transactions in Spanner
    - Snapshot Isolation
    - Replica Safe Time
    - TrueTime

R/O Transactions

- These may involve multiple reads
  - Perhaps from multiple shards
- Want these to be much faster then R/W Trans.
  - Read from local replicas
    - Fast, but local replica maybe not be up-to-date
    - If part of Paxos minority
- For R/O, Spanner doesn't use locks, 2-PC & TC
  - Avoid inter-DC messages, makes reads faster
  - Table 3/6 shows R/O transactions <u>~10X faster</u> as compared to R/W transactions
- Challenge: how to satisfy consistency reqs. with such reads?

Correctness constraints on R/O Trans.

- Serializable
  - Same results as if transactions executed one-by-one
    - Even though they may actually execute concurrently
- External consistency
  - If a transaction T2 starts after a transaction T1 has committed, T2 must see T1's writes
- Challenge: If we get the latest-value when we read, we still may not have a serialization order
    - For now lets assume local node among Paxos majority

Read the latest committed values

```
BEGIN
    Print X,Y
END
```

Suppose R/O transactions just read the latest committed values

Read the latest committed values

BEGIN
    Print X,Y
END

Suppose R/O transactions just read the latest committed values

T1 wx wy C

T2 wx wy C

T3 rx ry

Time

Read the latest committed values

Suppose R/O transactions just read the latest committed values

```
BEGIN
    Print X,Y
END
```

T1	wx	wy	C						
T2					wx	wy	C		
T3				rx				ry	

Time →

The results don't match any serial order
Not T1 T2 T3
Not T1, T3, T2

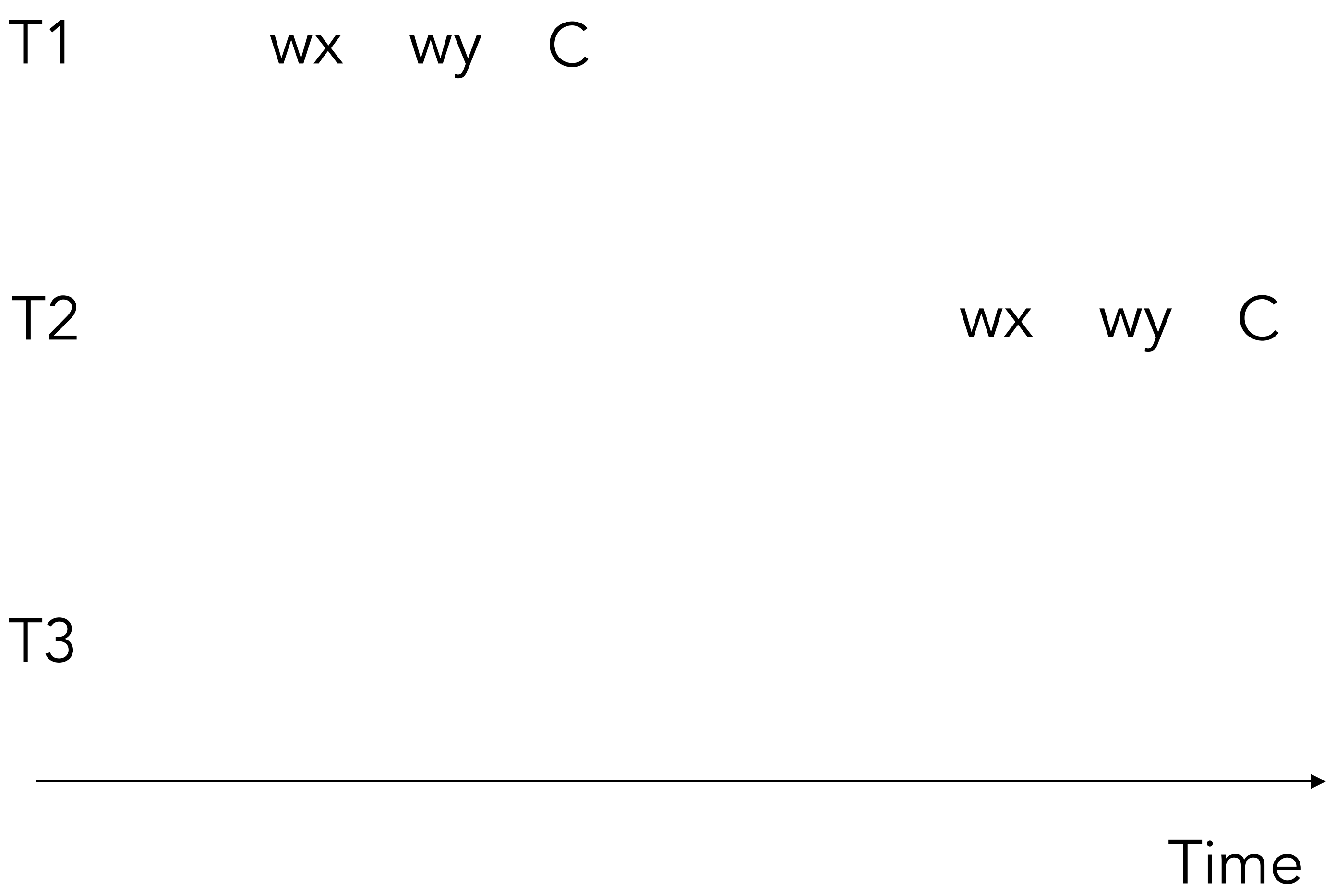
Read the latest committed values

- We want T3 to see both of T2's writes, or none

Idea: Snapshot Isolation

- Assume clocks are synchronized
  - We will relax this constraint later
- Assign every transaction a timestamp
  - R/W: TS = COMMIT TIME
  - R/O: TS = START TIME
  - Execute as if one-at-a-time in time-stamp order
    - Even if actual reads occur in different order
- Multi-version DB
  - Each replica stores multiple time-stamped versions of each record
    - All of a R/W transaction's writes get the same time-stamp
    - For R/O transactions, find the record version with the highest timestamp less than the R/O transaction timestamp

Snapshot Isolation



Snapshot Isolation

x@10 = 9
y@10 = 11

x@20 = 8
y@20 = 12

T1@10 wx wy C

T2@20 wx wy C

T3

Time

Snapshot Isolation

x@10 = 9
y@10 = 11

x@20 = 8
y@20 = 12

T1@10 wx wy C

T2@20 wx wy C

T3@15 rx ry

Time

Snapshot Isolation

x@10 = 9
y@10 = 11

x@20 = 8
y@20 = 12

T1@10 wx wy C

T2@20 wx wy C

T3@15 rx9 ry11

Time

Why is it OK to read stale value of Y?

- Its because T2 and T3 are concurrent. And linearizability says that if two transactions are concurrent, we can put either one first. Spanner puts T3 first

Another Challenge

- What if the local replica is from the minority and didn't see some writes?

Idea: Replica Safe time

- Paxos leaders send writes in timestamp order
- Before serving a read at time 20, replica must see Paxos write for time > 20. So it knows it has seen all writes < 20
- Problem: What if clocks are not perfectly synchronized?

What could go wrong if clocks aren't synchronized?

- If a R/O transaction's timestamp (TS) is too large:
  - Its TS will be higher than replica safe times, and reads will block
  - Correct but slow -- delay increased by amount of clock error

- If a R/O transaction's timestamp (TS) is too small:
  - It will miss writes that committed before the R/O transaction started
    - The low TS will cause it to use old versions of records
    - This violates external consistency

Example

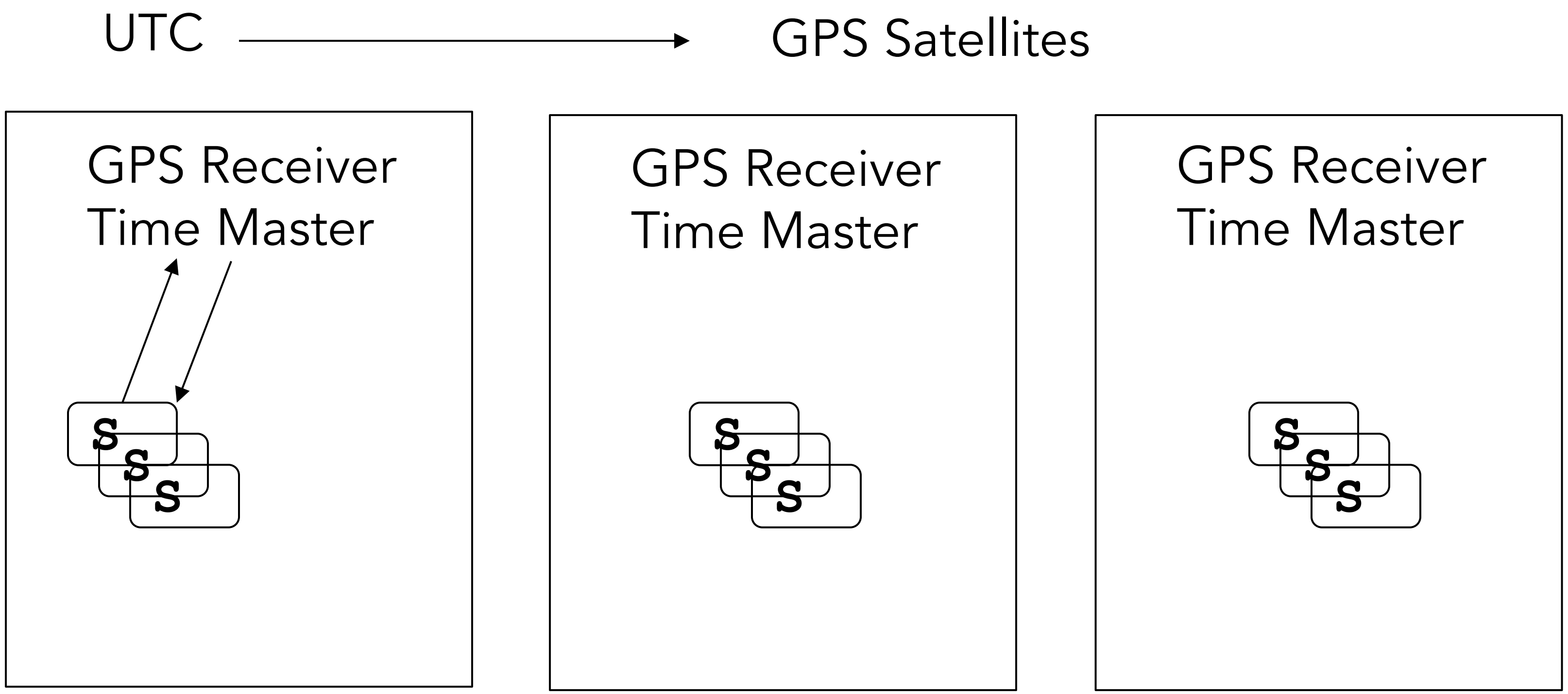
T1@0 wx1 C

T2@ 10 wx2 C

T3@5 rx

This would cause T3 to read the version of x at time 0, which was 1. But T3 started after T2 commited (in real time). So external consistency requires T2 see x=2

Google's Time Reference System



Server also talk to nearby time master

Uncertainty due to network delays, drift between checks

TrueTime

- Time service yields a TTinterval = [earliest, latest]
  - The correct time is guaranteed to be somewhere in the interval
  - Interval width computed from measured network delays, clock hardware specifications
- Figure 6: intervals are usually microseconds, but sometimes 10+ milliseconds
- Conclusion: server clocks aren't exactly synchronized, but TrueTime provides guaranteed bounds on how wrong a server's clock can be

How is TrueTime Used?

- Two Rules

- Start Rule (how timestamps are choosen?)
  - Timestamp (TS) = TT.now(). latest
  - R/O – start
  - R/W – commit

- Commit Wait – R/W transaction
  - Delay until TS < TT.now().earliest
  - After this delay TS guaranteed to be in the past

How TrueTime Enforces External Consistency?

T1

T2

T3

How TrueTime Enforces External Consistency?

T1@1 wx1 C

1 10

T2@10 wx2 C

T3

How TrueTime Enforces External Consistency?

T1@1 wx1 C

1 10

T2@10 wx2 C

10 12

T3@12 rx

How TrueTime Enforces External Consistency?

T1@1 wx1 C

1 10 11 20

T2@10 wx2 C

10 12

T3@12 rx

Why this provides external consistency?

- Commit wait means R/W TS is guaranteed to be in the past

- R/O TS = TT.now().latest is guaranteed to be >= correct time
  - Thus >= TS of any previous committed transaction (due to its commit wait)

More generally

- Snapshot Isolation gives you serializable R/O transactions
  - Timestamps set an order
  - Snapshot versions (and safe time) implement consistent reads at a timestamp
  - Transaction sees all writes from lower-TS transactions, none from higher. Any number will do for TS if you don't care about external consistency
- Synchronized timestamps yield external consistency
  - Even among transactions at different data centers
  - Even though reading from local replicas that might lag

Why is all this useful?

- Fast R/O transactions
  - Read from replica in client's datacenter
  - No locking, no two-phase commit
  - Thus the 10x latency improvement in Tables 3 and 6

- Although:
  - R/O transaction reads may block due to safe time, to catch up
  - R/W transaction commits may block in Commit Wait. Accurate (small interval) time minimizes these delays

Summary

- Rare to see deployed systems offer distributed transactions with strong consistency over geographically distributed data
- Spanner was a surprising demonstration that it can be practical
- Timestamping scheme and 2-PC over Paxos are the two most interesting aspects
- Widely used within Google; a commercial Google service; influential