# Distributed Systems

## Spring Semester 2020

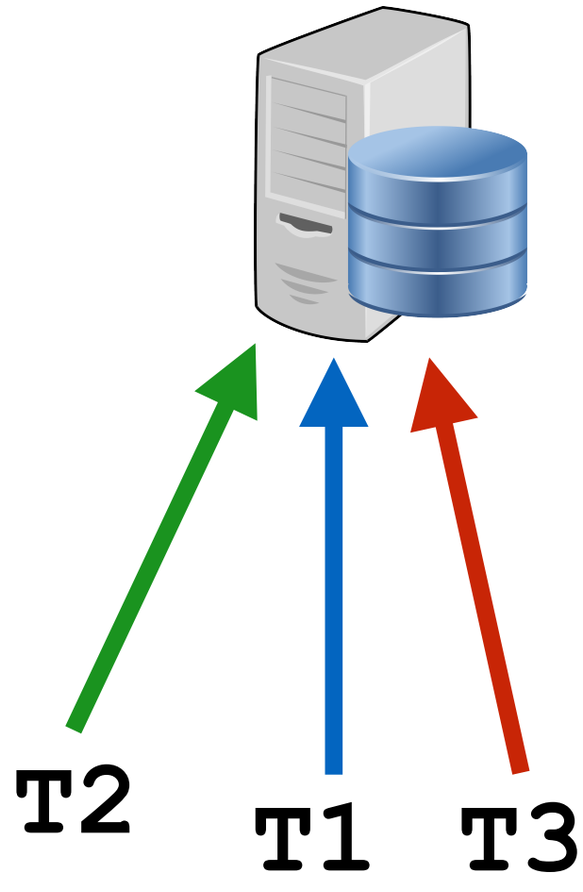
## Lecture 10: Optimistic Concurrency Control (Thor)

**John Liagouris**
**liagos@bu.edu**

# Scheme 1 : Centralized

## Example 2

```
  x=0   y=0
T1:  Rx0  Wx1
T2:  Rx0  Wy1
T3:  Ry0  Rx1
```

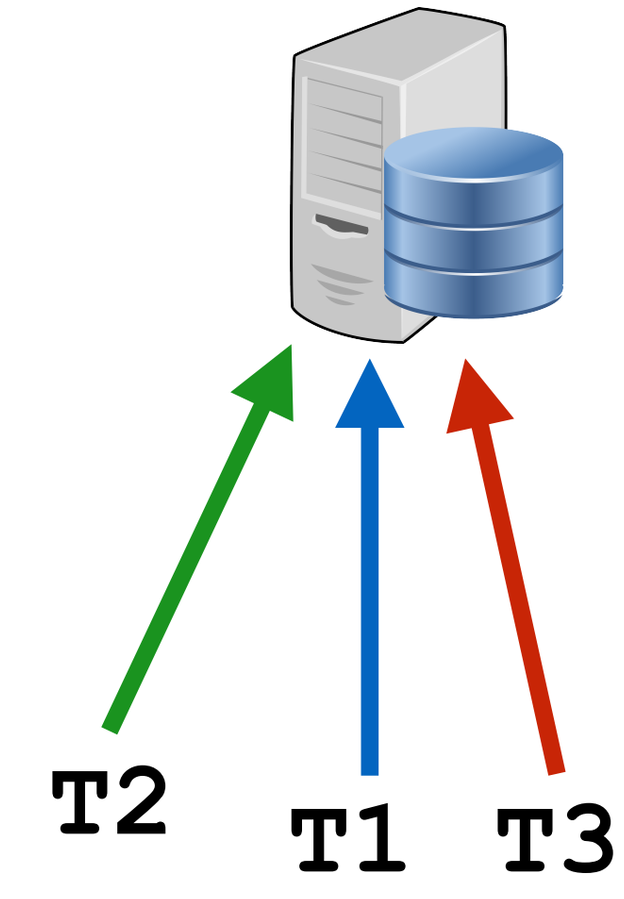


# Scheme 1 : Centralized

## Example 2

```
  x=0   y=0

T1:  Rx0  Wx1
T2:  Rx0  Wy1
T3:  Ry0  Rx1
```



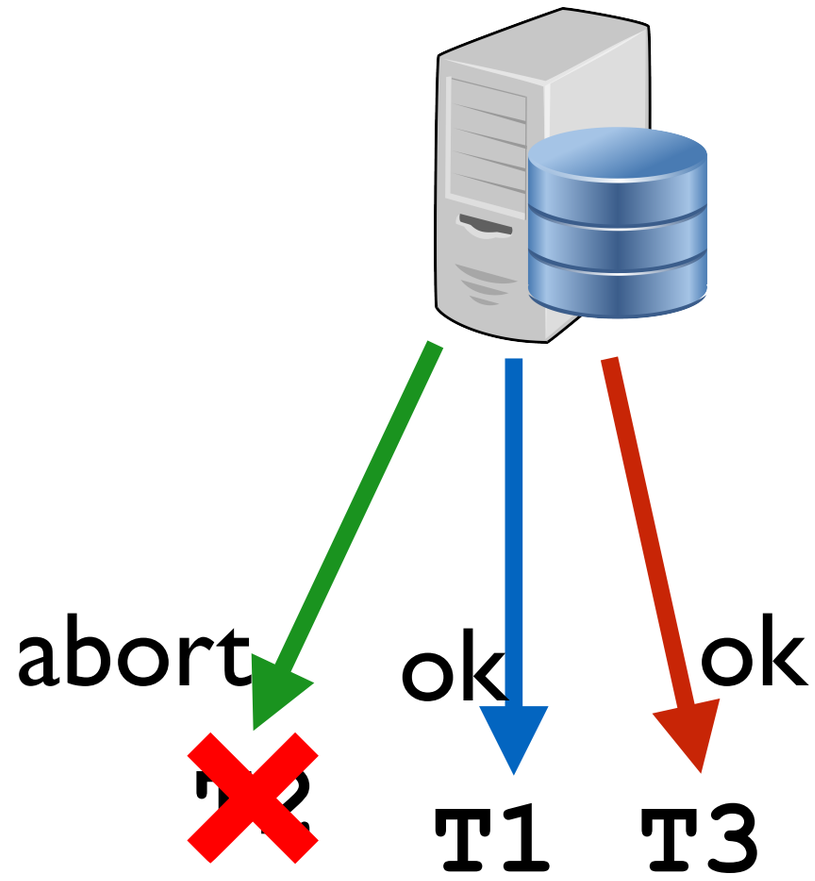

```
T1,T3: required (via x)
T3,T2: required (via y)
BUT T2,T1 required (via x)
Circular Dependency :-(
```

# Scheme 1 : Centralized

## Example 2

**x=0  y=0**

**T1: Rx0 Wx1**
**T2: Rx0 Wy1**
**T3: Ry0 Rx1**




Can not be Validated!
**Values** not consistent with any serial order!

Can fix things by replying Abort to one and OK to other two (eg. Abort T2 — T1, T3 OK)

# Scheme 1 : Centralized

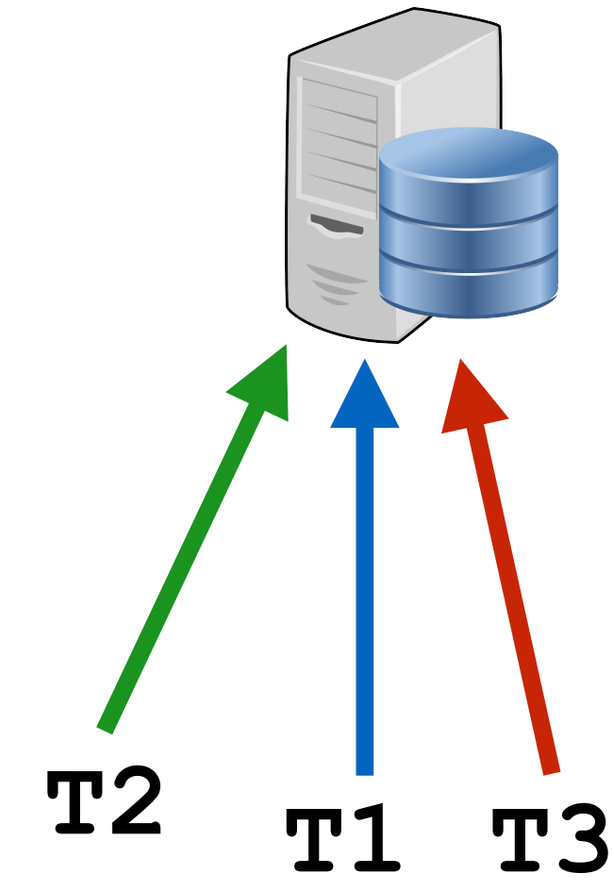
## Example 3: Readonly Transactions

**x=0  y=0**

**T1: Wx1**

**T2: Rx1 Wy2**

**T3: Ry2 Rx0**




Do we need to validate read-only transactions?

# Scheme 1 : Centralized

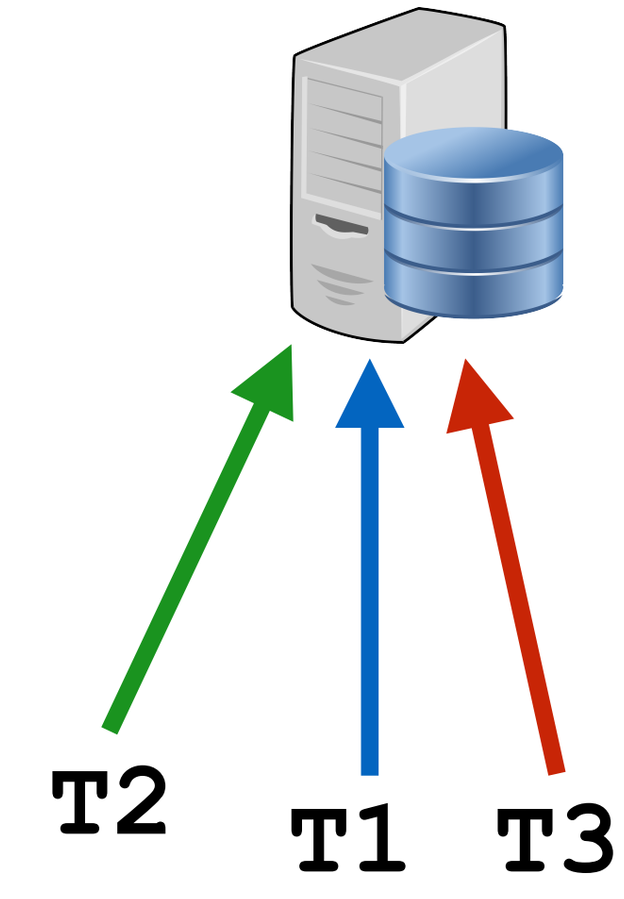
## Example 3: Readonly Transactions

**x=0  y=0**

**T1: Wx1**

**T2: Rx1 Wy2**

**T3: Ry2 Rx0**



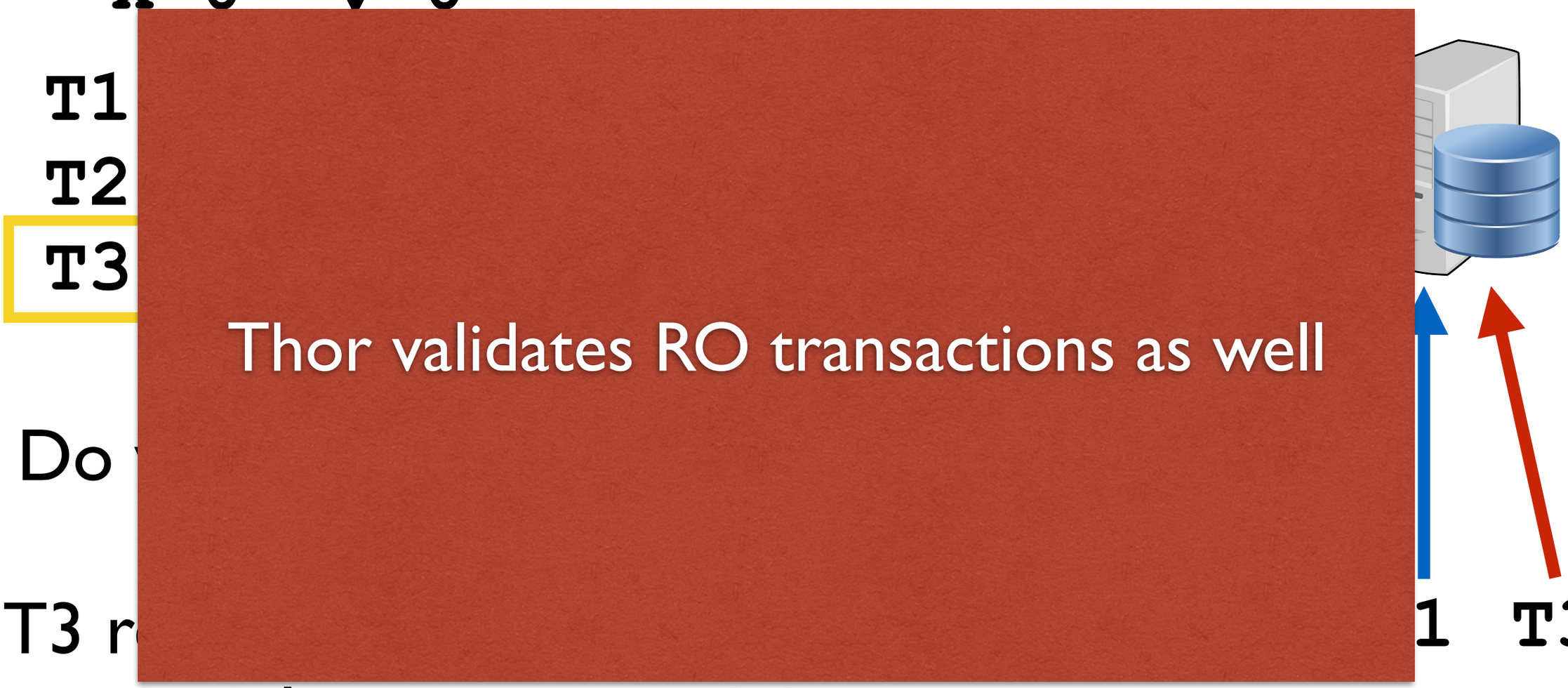

Do we need to validate read-only transactions?

T3 read a stale x=0 from its cache, hadn't yet seen invalidate.

need to validate in order to abort T3.

# Scheme 1 : Centralized

## Example 3: Readonly Transactions




Do

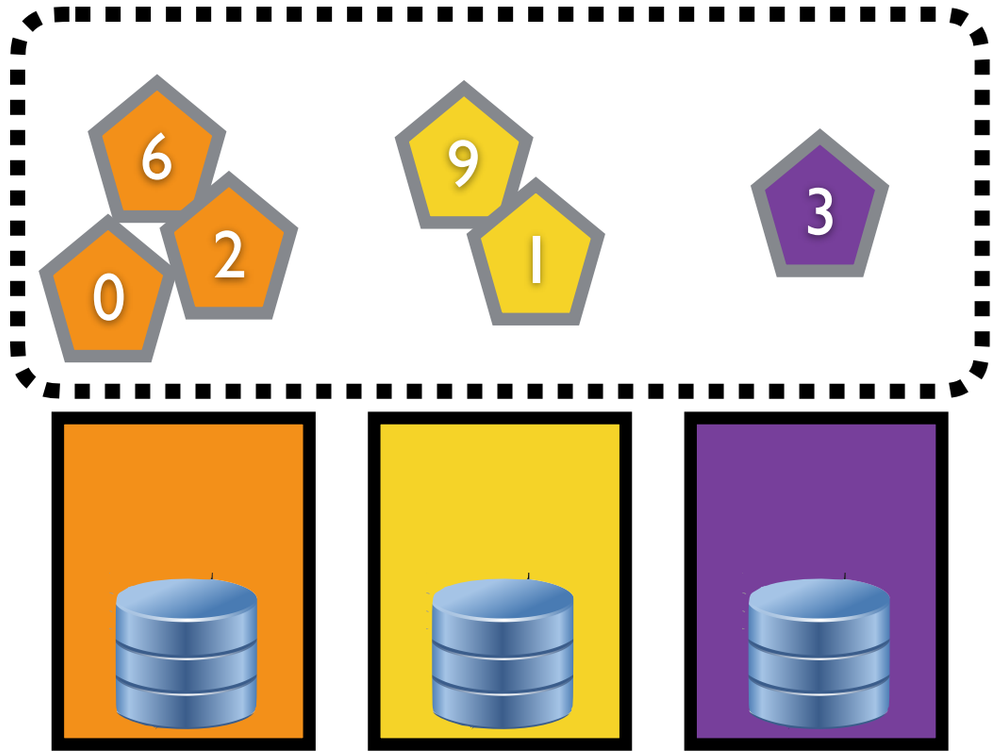
T3 r

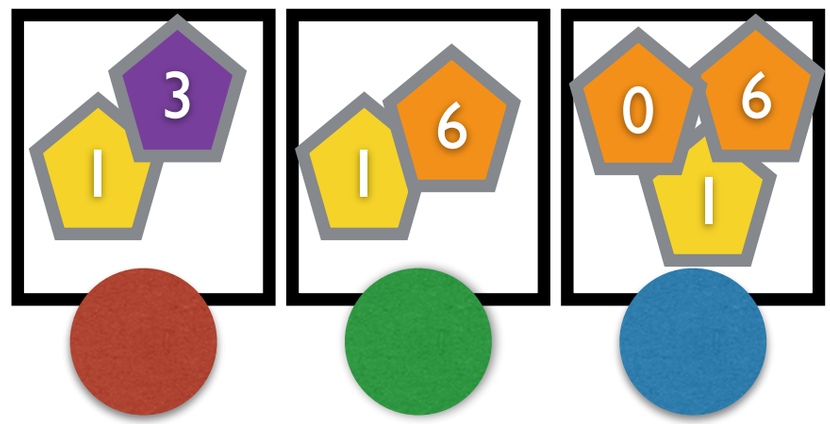
hadn't yet seen invalidate.

need to validate in order to abort T3.

# Distributed

- Of course what we really want is distributed OCC where storage and validation work is split across servers
- Each server only sees transactions that use its data
- Each server only validates just its part of the transaction
- Aggregate using two phase commit to see that all relevant servers say yes then only commit



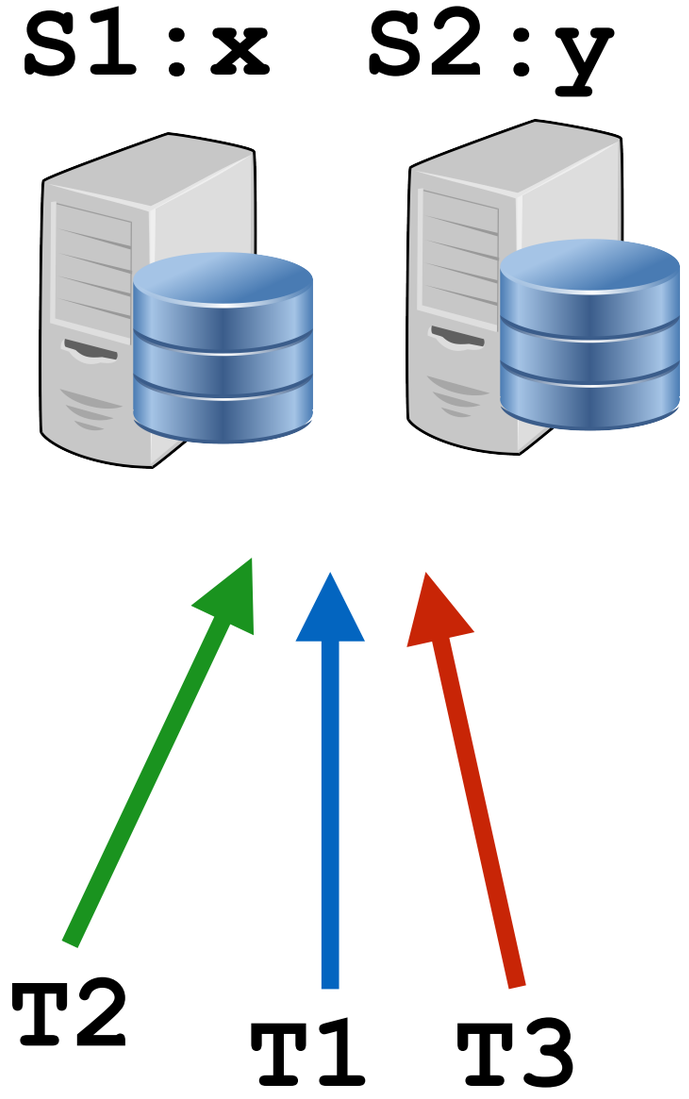

T:<R/W,id,Value>,…

# Distributed (naive)

## Example 2 again

```
  x=0   y=0

T1: Rx0 Wx1
T2: Rx0 Wy1
T3: Ry0 Rx1
```



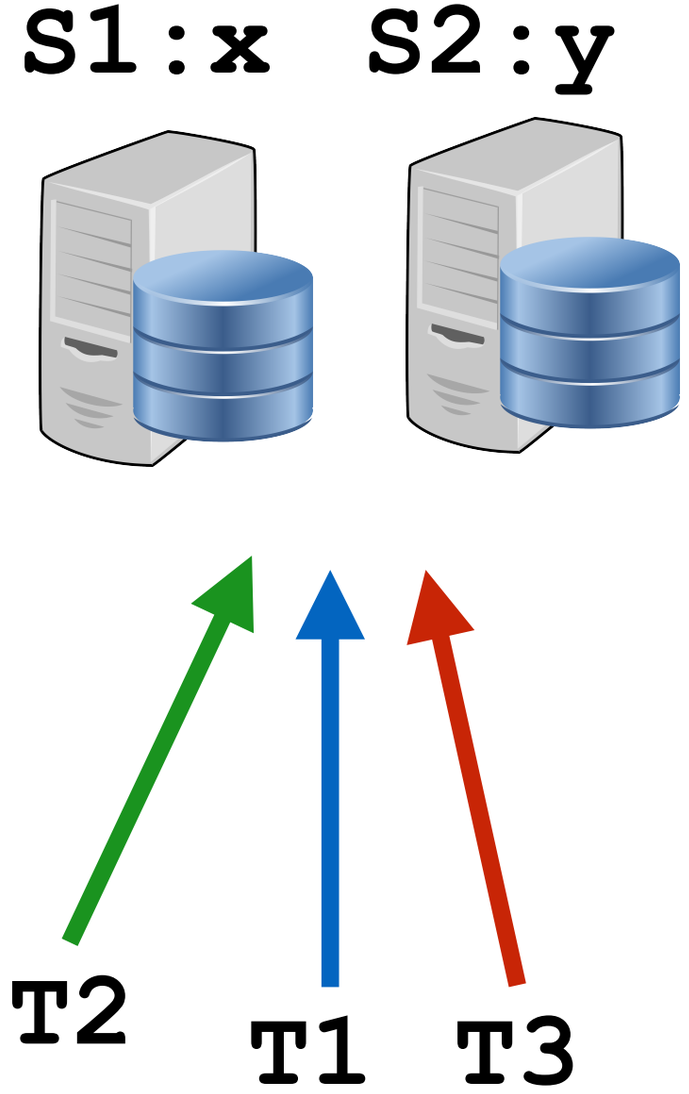
# Distributed (naive)

## Example 2 again

**x=0 y=0**

**T1: Rx0 Wx1**
**T2: Rx0 Wy1**
**T3: Ry0 Rx1**

**S1:x only**
**T1: Rx0 Wx1**
**T2: Rx0**
**T3: Rx1**
**YES(T2,T1,T3)**



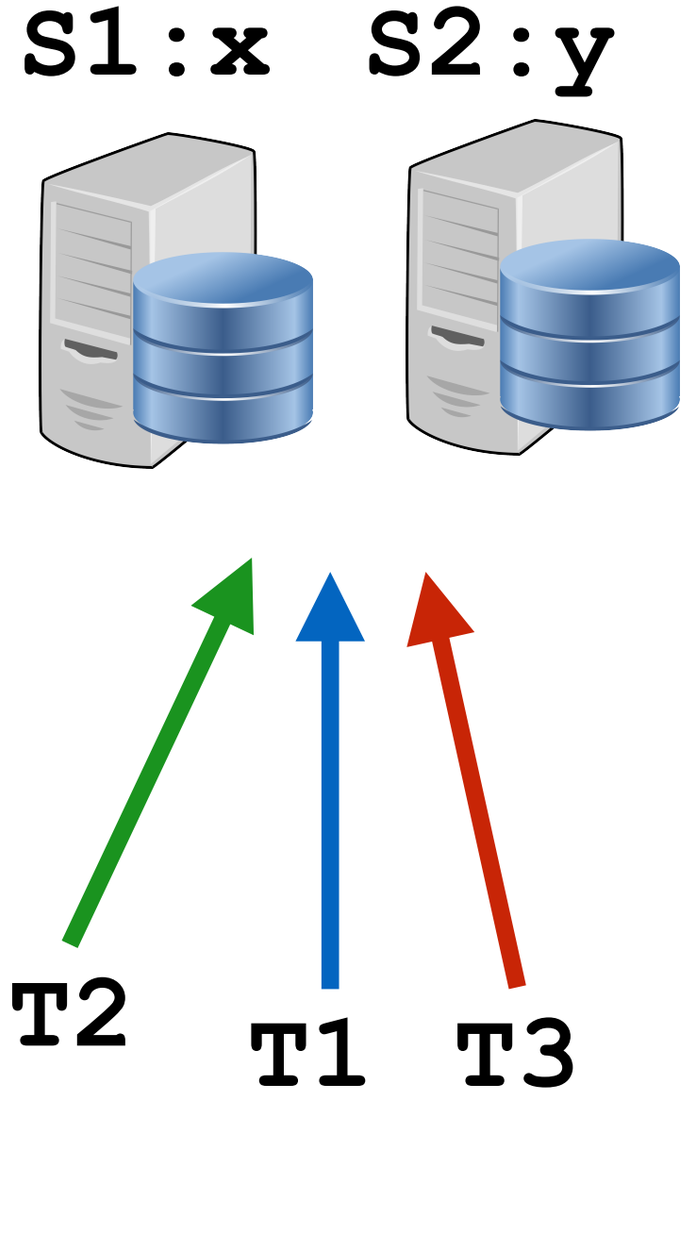
# Distributed (naive)

## Example 2 again

```
  x=0   y=0

T1: Rx0 Wx1
T2: Rx0 Wy1
T3: Ry0 Rx1
```

| S1:x only | S2:y only |
|---|---|
| T1: Rx0 Wx1 | T2: Wy1 |
| T2: Rx0 | T3: Ry0 |
| T3: Rx1 | |
| YES(T2,T1,T3) | YES (T3,T2) |



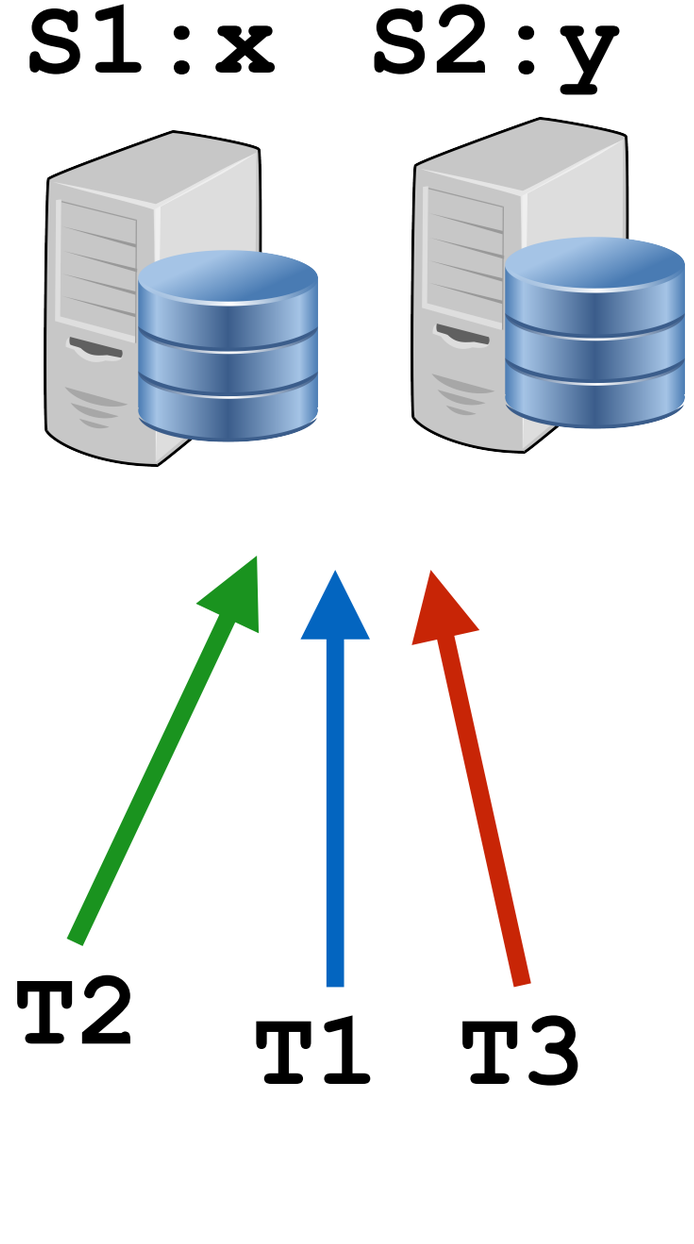
# Distributed (naive)

## Example 2 again

```
  x=0   y=0

T1: Rx0 Wx1
T2: Rx0 Wy1
T3: Ry0 Rx1
```

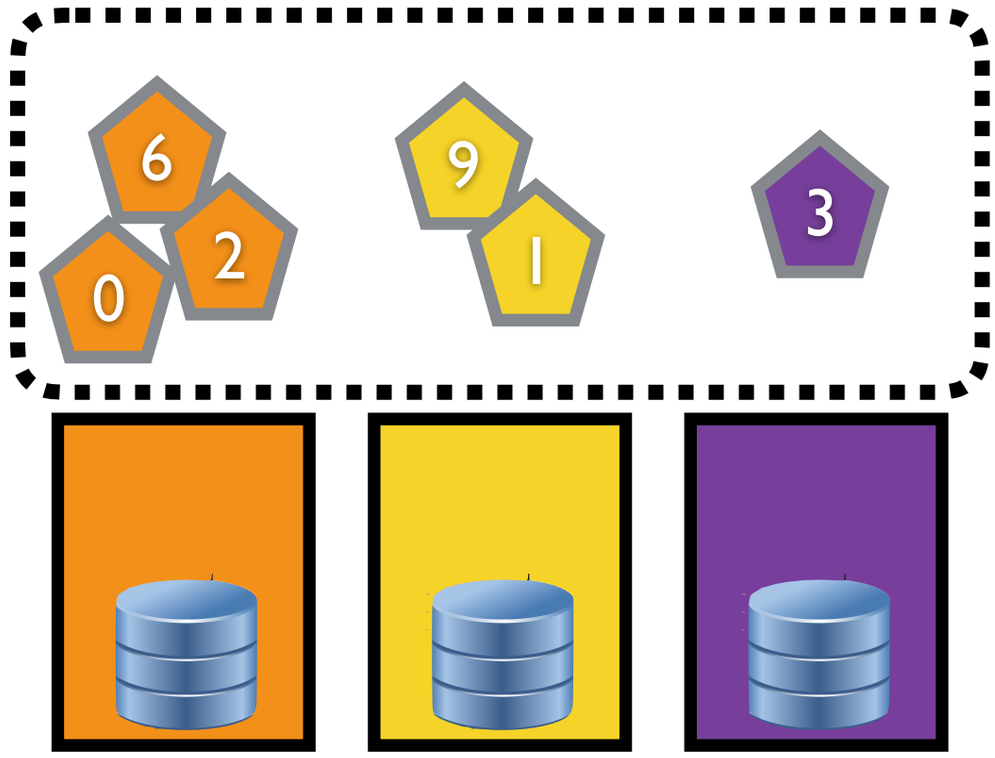
| S1:x only | S2:y only |
|---|---|
| T1: Rx0 Wx1 | T2: Wy1 |
| T2: Rx0 | T3: Ry0 |
| T3: Rx1 | |
| YES(T2,T1,T3) | YES (T3,T2) |

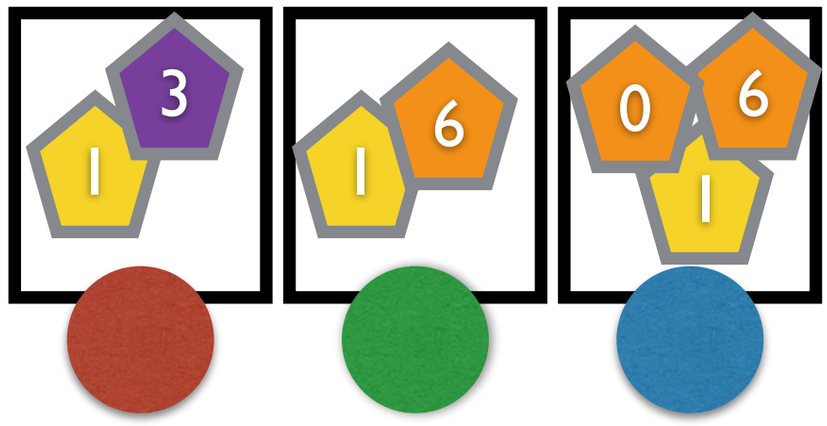



Says Yes but we know the real answer is NO!

# Distributed (TimeStamps)

- Coordinator/Client stamps a transaction with a timestamp (Thor loosely synchronized clocks)
- Validation now checks to see if reads and writes are in timestamp order
- Solves distributed problem: validating the same order — yes will now refer to same order being valid



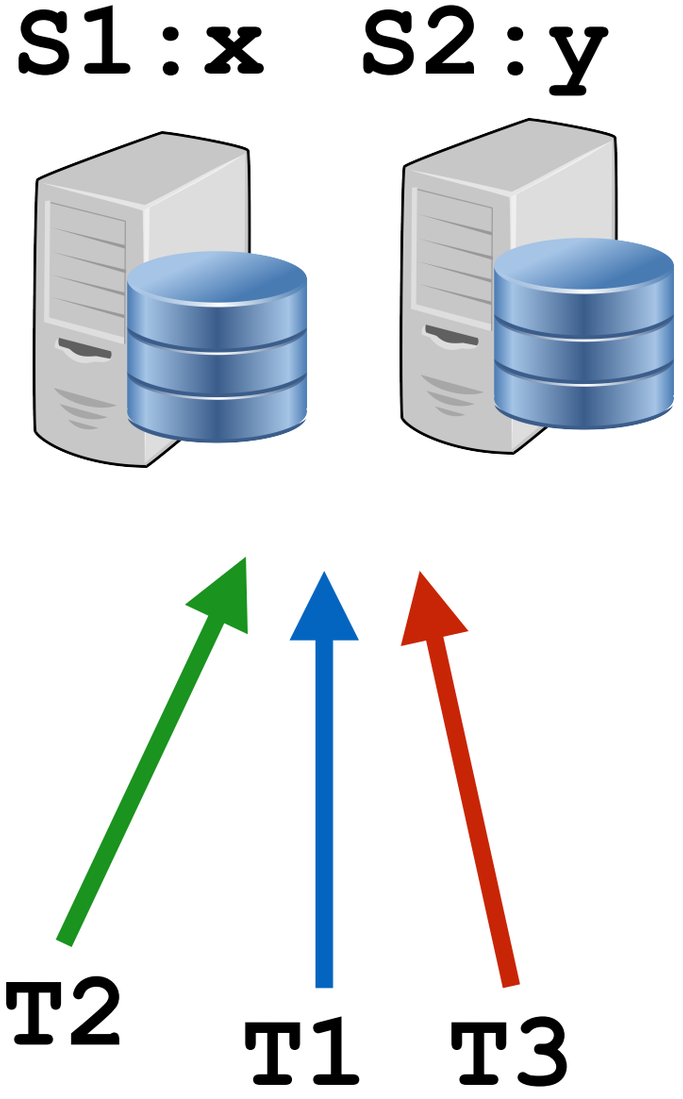

T:<R/W,id,Value>,…

# Distributed (TS)

## Example 2 again

```
    x=0   y=0

T1@100: Rx0 Wx1
T2@110: Rx0 Wy1
T3@105: Ry0 Rx1
```



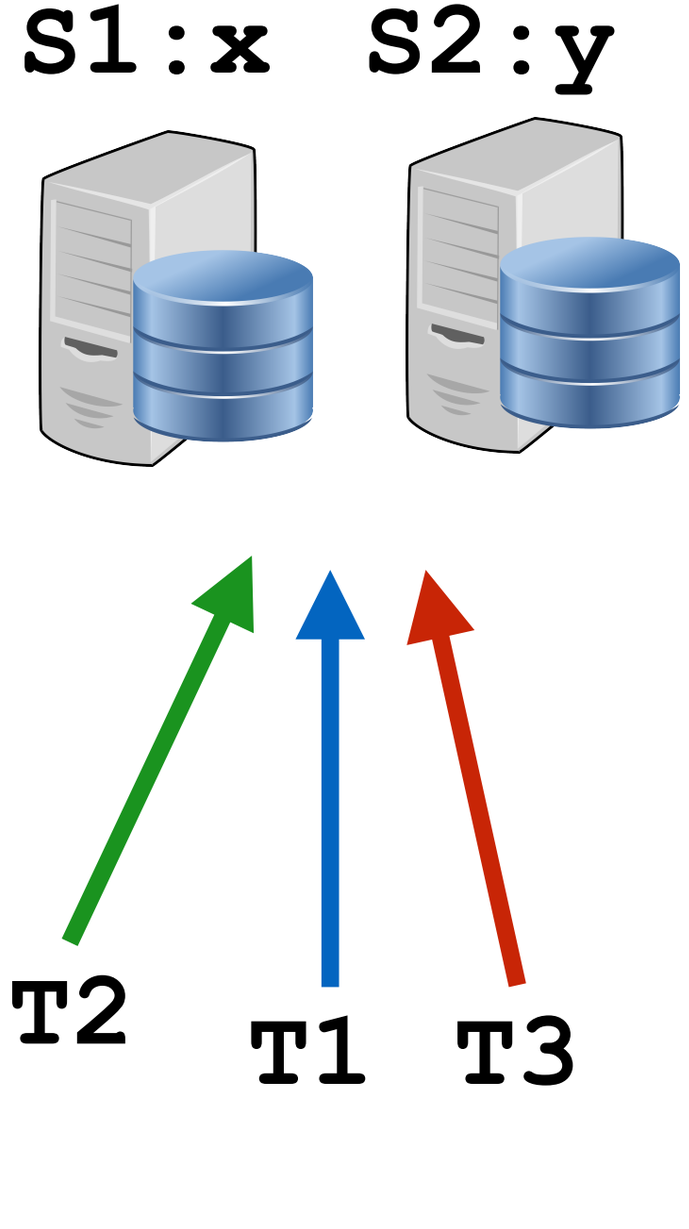
# Distributed (TS)

## Example 2 again

```
    x=0   y=0

T1@100: Rx0 Wx1
T2@110: Rx0 Wy1
T3@105: Ry0 Rx1
```

```
S1:x only
T1@100: Rx0 Wx1
T2@110: Rx0
T3@105: Rx1
TS impose
T1,T3,T2 -> NO
```

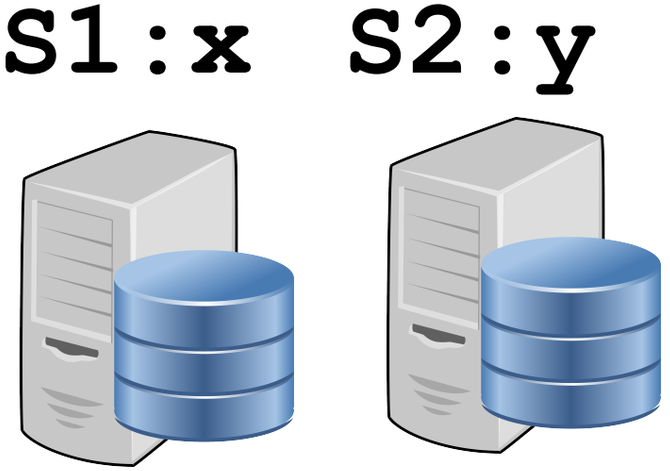


# Distributed (TS)

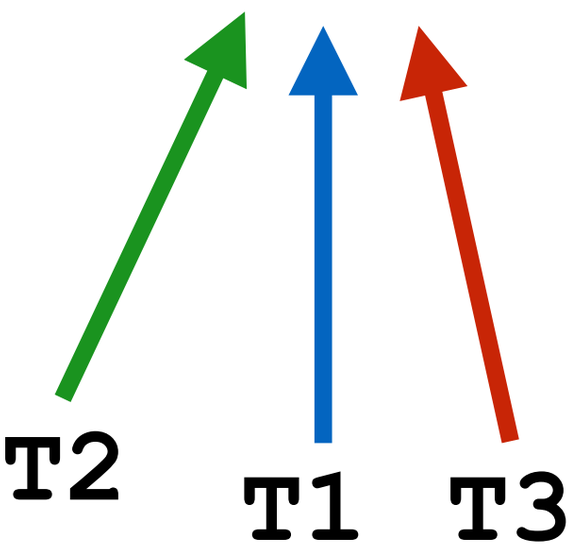
## Example 2 again

```
     x=0   y=0

T1@100: Rx0 Wx1
T2@110: Rx0 Wy1
T3@105: Ry0 Rx1
```




| S1:x only | S2:y only |
|---|---|
| T1@100: Rx0 Wx1 | T2@110: Wy1 |
| T2@110: Rx0 | T3@105: Ry0 |
| T3@105: Rx1 | |
| TS impose | TS impose |
| T1,T3,T2 -> NO | YES (T3,T2) |



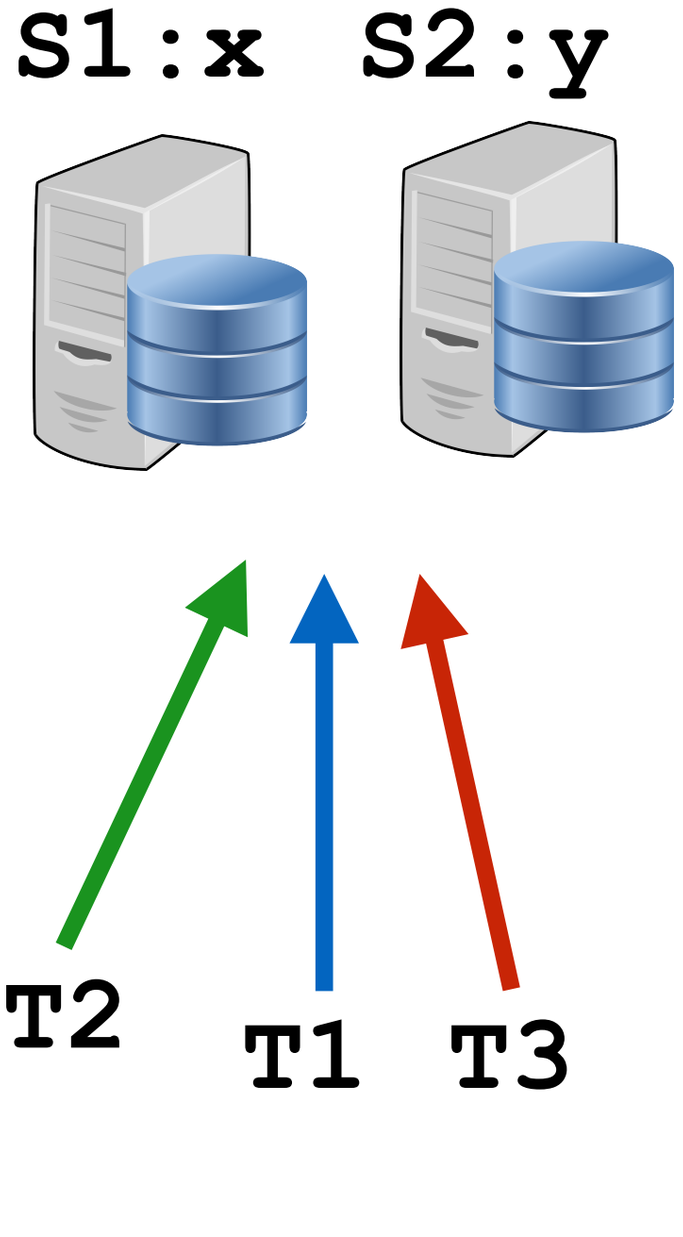
# Distributed (TS)

## Example 2 again

```
    x=0   y=0

T1@100: Rx0 Wx1
T2@110: Rx0 Wy1
T3@105: Ry0 Rx1
```

```
S1:x only          |  S2:y only
T1@100: Rx0 Wx1    |  T2@110: Wy1
T2@110: Rx0        |  T3@105: Ry0
T3@105: Rx1        |
TS impose          |  TS impose
T1,T3,T2 -> NO     |  YES (T3,T2)
```



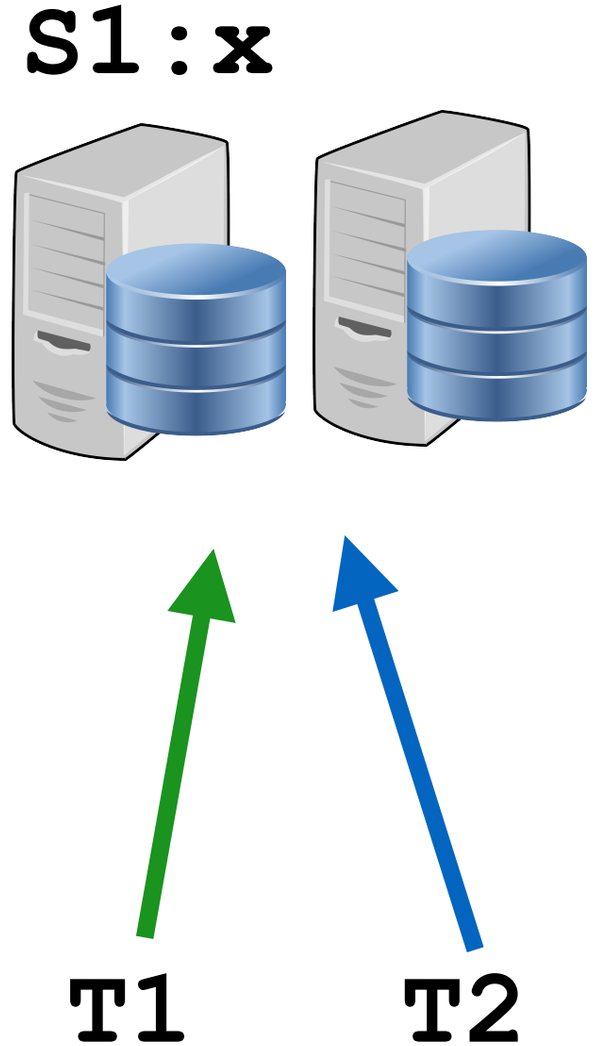

Says NO correctly

# What have we given up

## Example

**x=0**

**T1@100: Rx0 Wx1**
**T2@50: Rx1 Wx2**

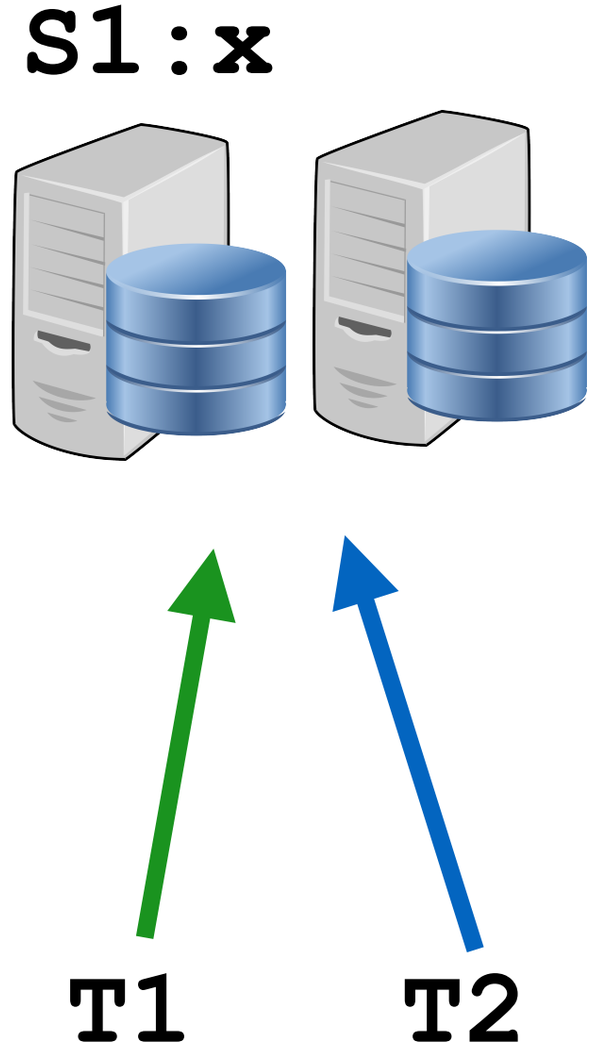


# What have we given up

## Example

**x=0**

**T1@100: Rx0 Wx1**
**T2@50: Rx1 Wx2**




T2 follows T1 in real time and sees T1's write but T2 will abort since TS says order ids T2,T1 — T2 comes first, so T1 should have seen x=2
Even though T1,T2 order works does not consider it

# What have we given up

## Example

x=0

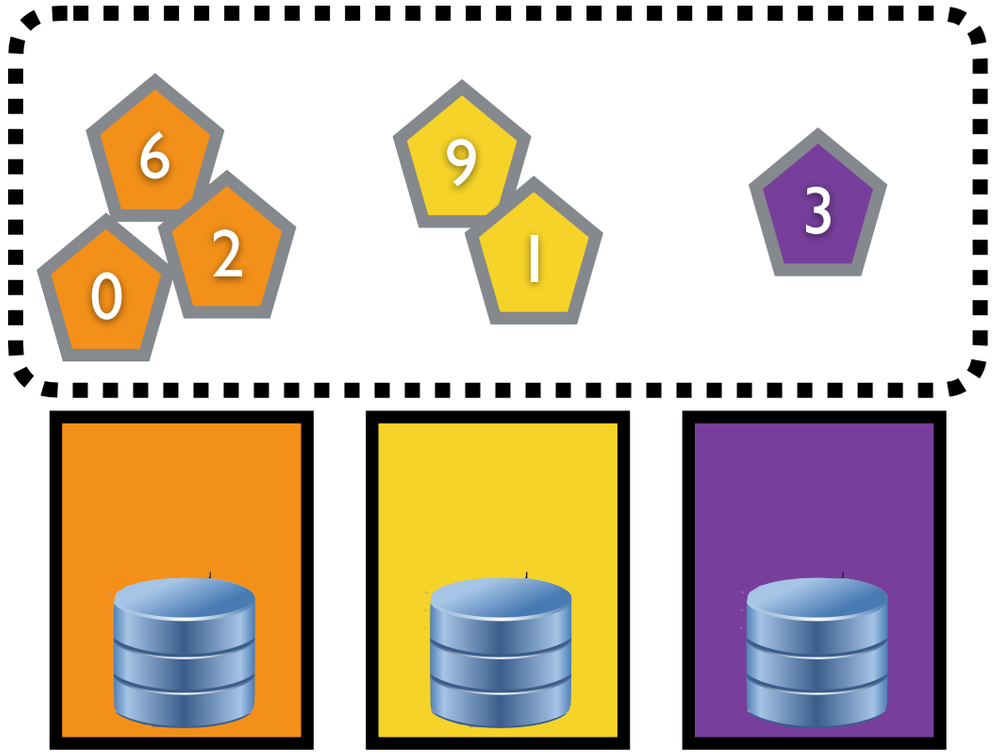
Requiring TS order can abort unnecessarily:
Validation no longer *searching* for an order that works — instead merely *checking* that TS order consistent w/ reads, writes

We have given up some optimism by requiring TS order
- Maybe: not a problem if clocks closely synched
- Maybe: not a problem if conflicts are rare

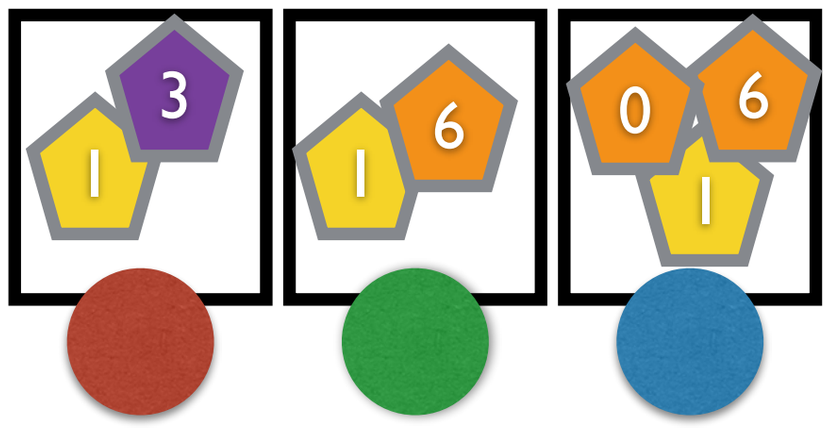
not consider it

T1 T2

# Distributed (TimeStamps)

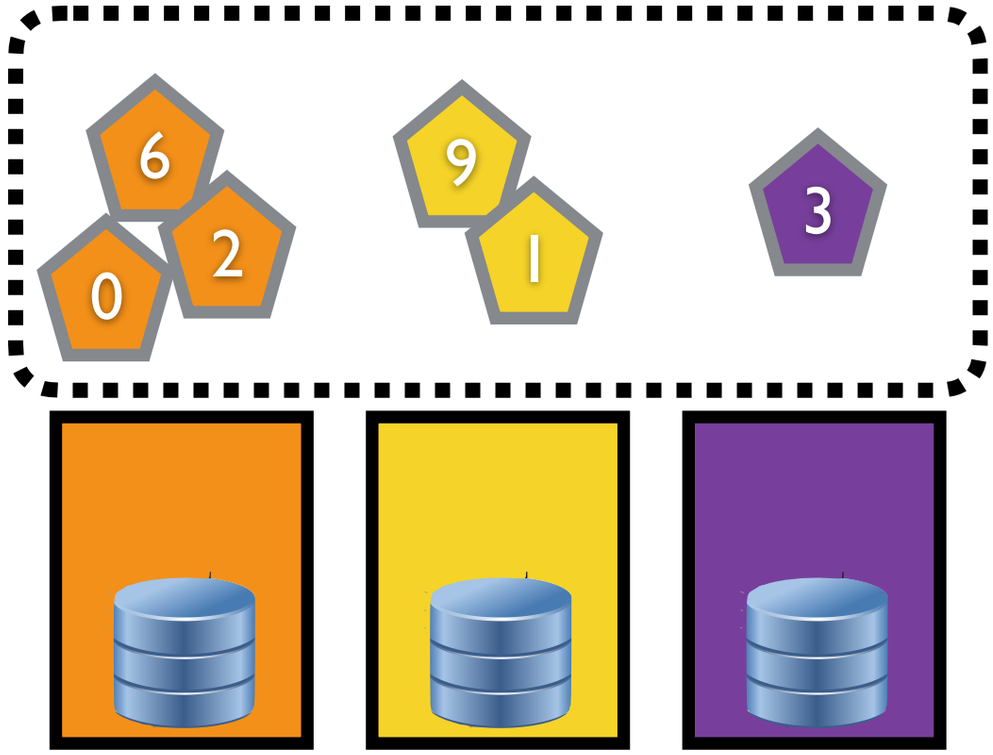



- Besides the spurious aborts there is another problem



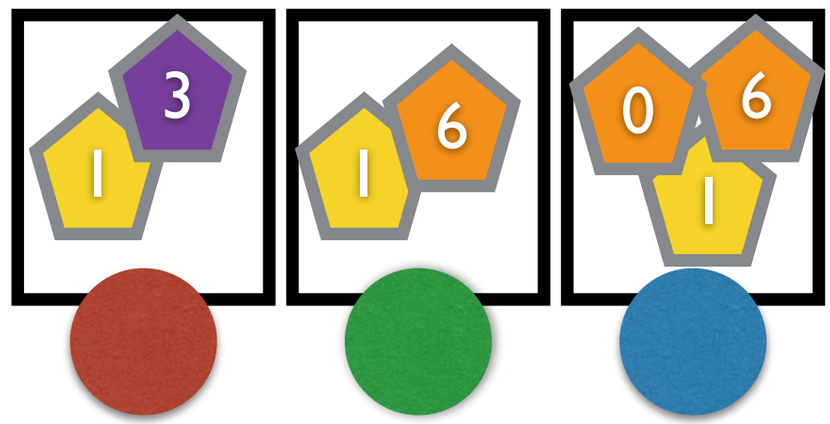

T:<R/W,id,Value>,…

# Distributed (TimeStamps)




- Besides the spurious aborts there is another problem
- Commit messages need to send around Values which could be big

```
T1@100: Rx0 Wx1
T2@50:  Rx1 Wx2
```

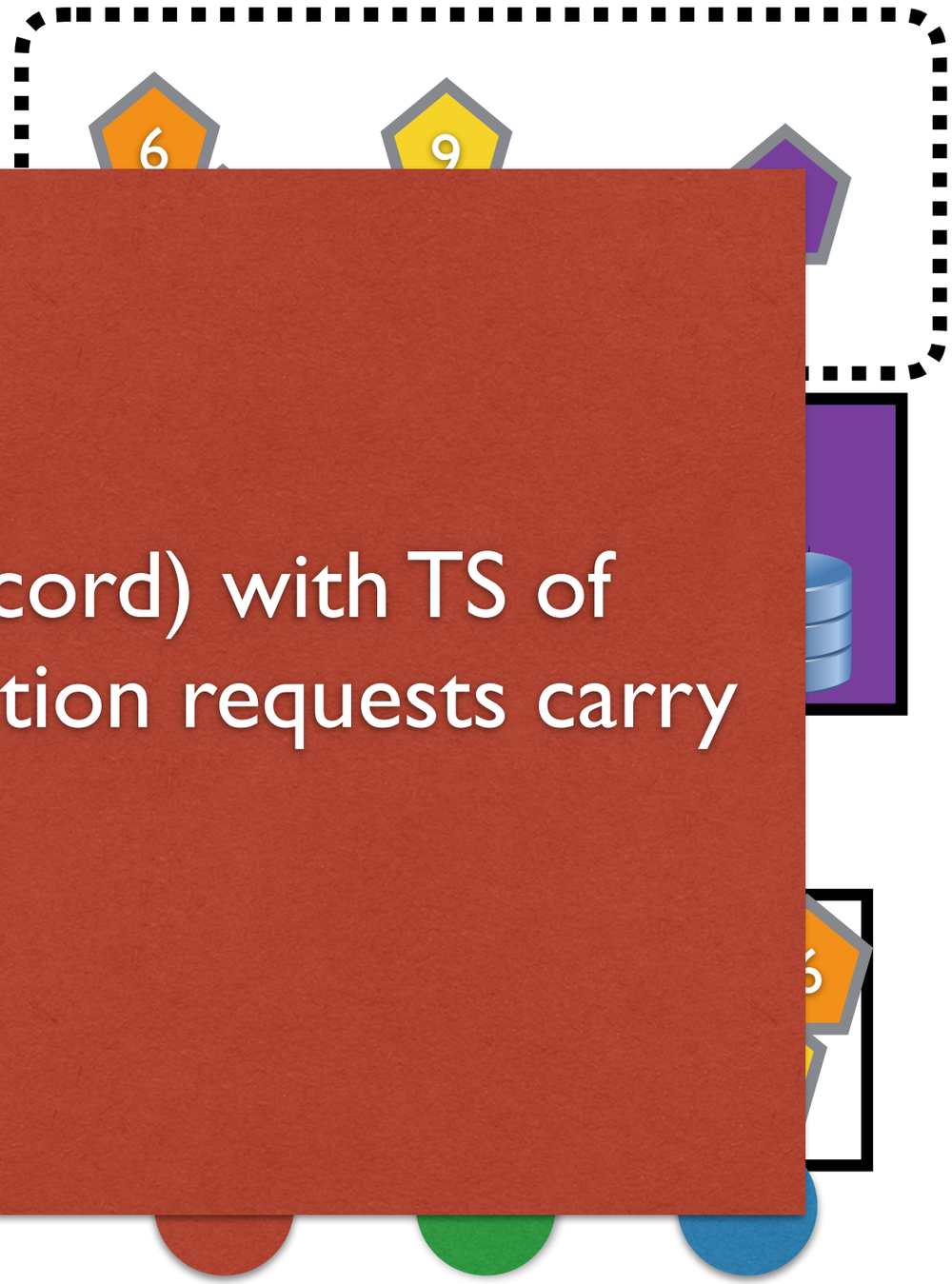



T:<R/W,id,Value>,…

# Distributed (TimeStamps)

- Besides the spurious aborts

TimeStamp + V#
Tag each DB record (and cached record) with TS of transaction that last wrote it. Validation requests carry TS of each record read




version number

T:<R/W,id,Value>,…

# Distributed (Timestamp + V#)

## Example 2 again

**x=0@0  y=0@0**

**T1@100: Rx@0 Wx**

**T2@110: Rx@0 Wy**

**T3@105: Ry@0 Rx@100**

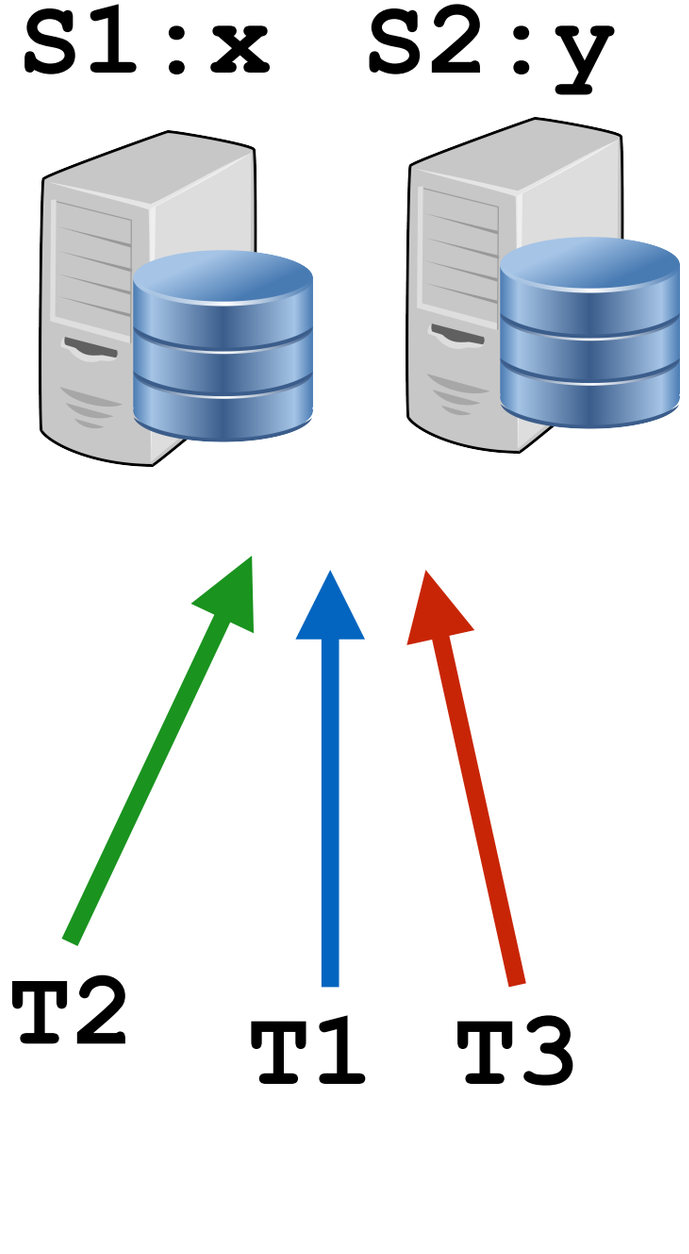
**S1:x only**

**T1@100: Rx@0 Wx**

**T3@105: Rx@100**

**T2@110: Rx@0**

**Does each record see most recent write — NO T2 is stale**



# Distributed (Timestamp + V#)

## Example 2 again

```
   x=0@0   y=0@0
T1@100: Rx@0 Wx
T2@110: Rx@0 Wy
T3@105: Ry@0 Rx@100
```

| S1:x only (in TS order) | S2:x only |
|---|---|
| T1@100: Rx@0 Wx | T3@105: Ry@0 |
| T3@105: Rx@100 | T2@110: Wy |
| T2@110: Rx@0 | |
| Does each record see most recent write — | |
| NO — T2 is stale | YES |

# Distributed (Timestamp + V#)

## Example 2 again

```
   x=0@0   y=0@0
T1@100: Rx@0 Wx
T2@110: Rx@0 Wy
T3@105: Ry@0 Rx@100
```

```
S1:x or
T1@100: Rx@0 Wx
T3@105: Rx@100
T2@110: Rx@0
Does each record see
most recent write —
NO — T2 is stale
```

Says NO correctly

```
T3@105: Ry@0
T2@110: Wy
```

YES

# Distributed (Timestamp + V#)

What have we give up by thinking about version #s rather than values? Maybe version numbers are different but values are the same

e.g.
T1@100:Wx1
T2@110:Wx2
T3@120:Wx1
T4@130: Rx1@100

Timestamps say we should abort T4 because read a stale version should have read T3's write — so scheme will abort
But T4 read the correct value — x=1 so abort wasn't necessary

# Distributed (Timestamp + V#)

Thor was concerned that storing per-record timestamps might use too much storage space — Thor avoids space overhead

Not clear this really is important

Thor: no per-record timestamps rather coupled validation testing with cache invalidation — invalidation sets used to detect if read is stale