Distributed Systems

Spring Semester 2020

Lecture 11: Distributed Programming in Argus

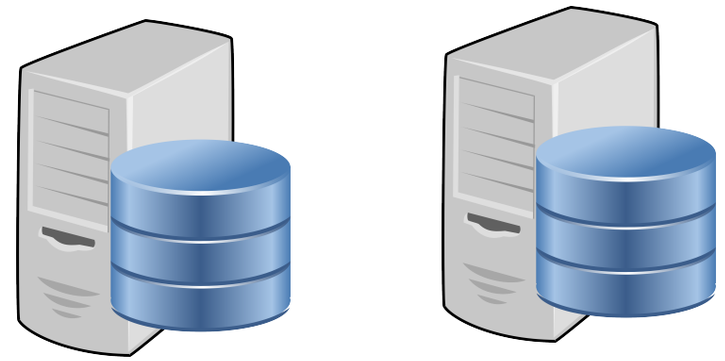
John Liagouris
liagos@bu.edu

Distributed Commit

A bunch of computers are cooperating on some task, e.g. bank transfer

Each computer in the system has a different role — Data

SRC:x **DST:y**



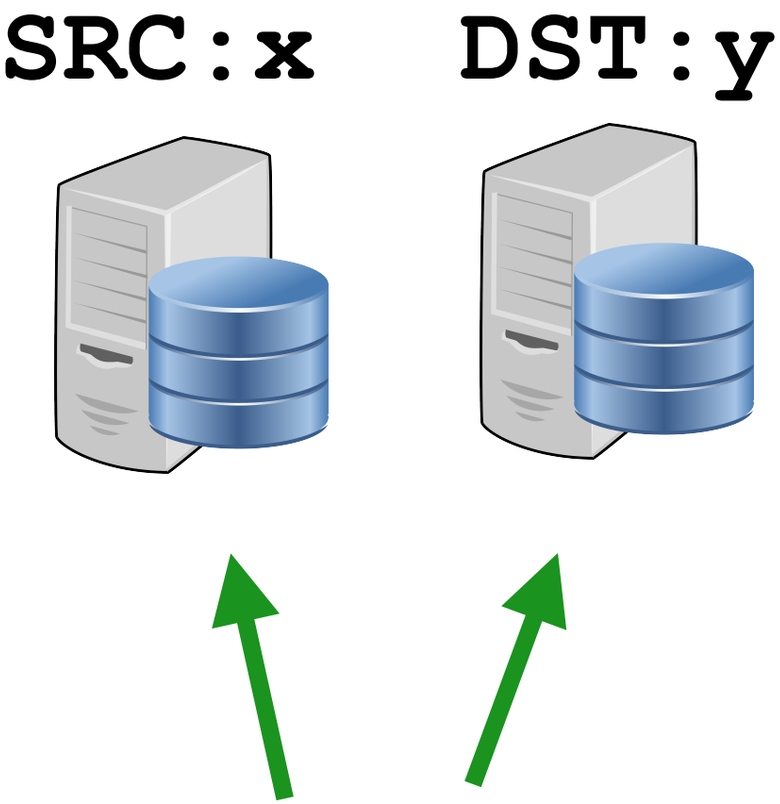
transfer:
src:x-=10,
dst:y+=10

Distributed Commit

A bunch of computers are cooperating on some task, e.g. bank transfer

Each computer in the system has a different role — Data

Need Atomicity:
all ops execute or none



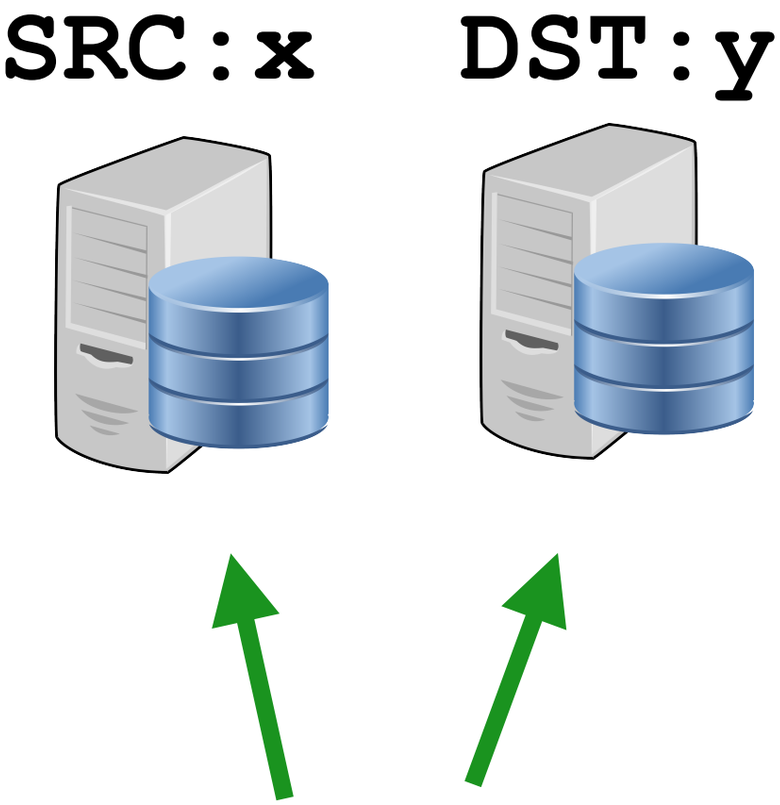
Distributed Commit

A bunch of computers are cooperating on some task, e.g. bank transfer

Each computer in the system has a different role — Data

Need Atomicity: all ops execute or none

Challenges: Failures, Concurrency, Performance



transfer:
src:x-=10,
dst:y+=10

The Idea

1) First tentative changes,
2) Later commit or undo (abort)

```
reserve_handler(u, t):
    if u[t] is free:
      temp_u[t] = taken -- A TMP VERSION
      return true
   else:
      return false
```

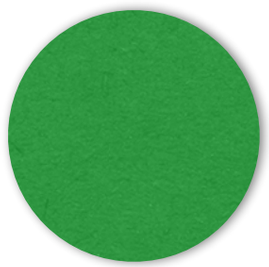
```
commit_handler():
   copy temp_u[t] to real u[t]
```

```
abort_handler():
   discard temp_u[t]
```

The Idea

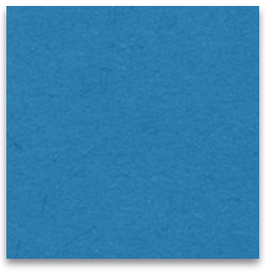
Single entity decides whether to commit
it prevents any chance of disagreement

Transaction Coordinator (TC)



Server A

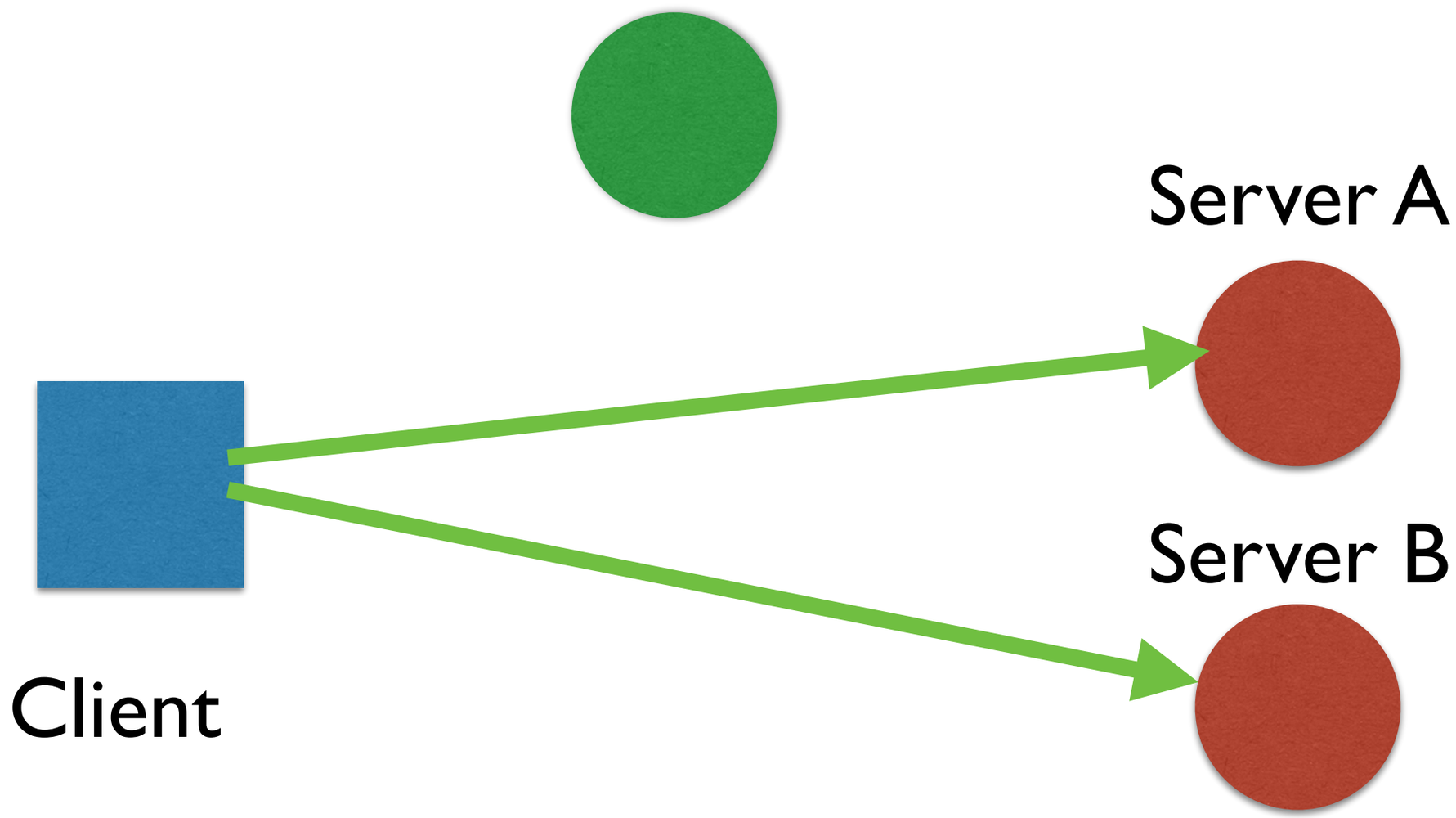
Server B



Client

The Idea

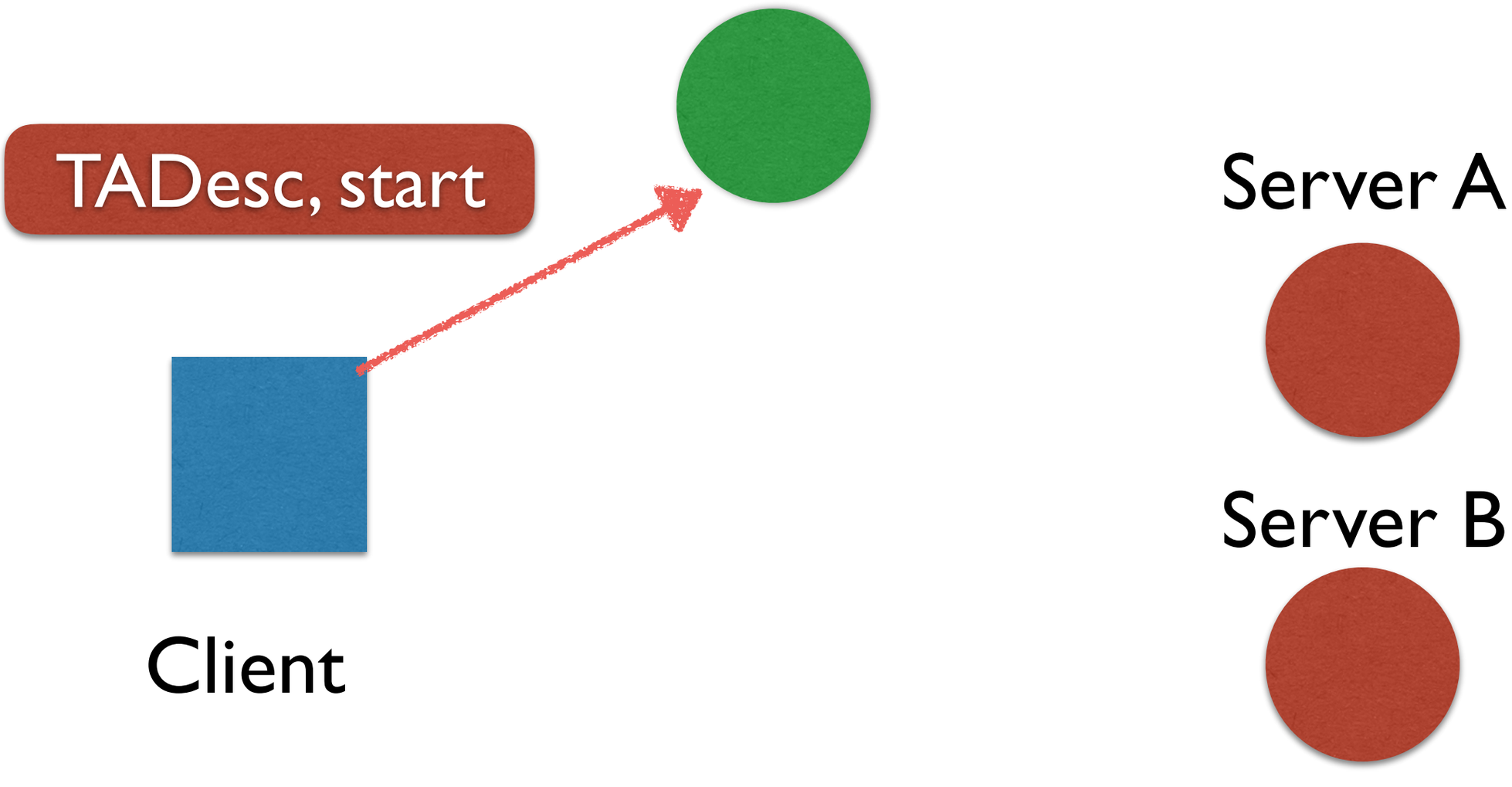
Single entity decides whether to commit
it prevents any chance of disagreement



Step 1: Client still sends RPCs to A and B

The Idea

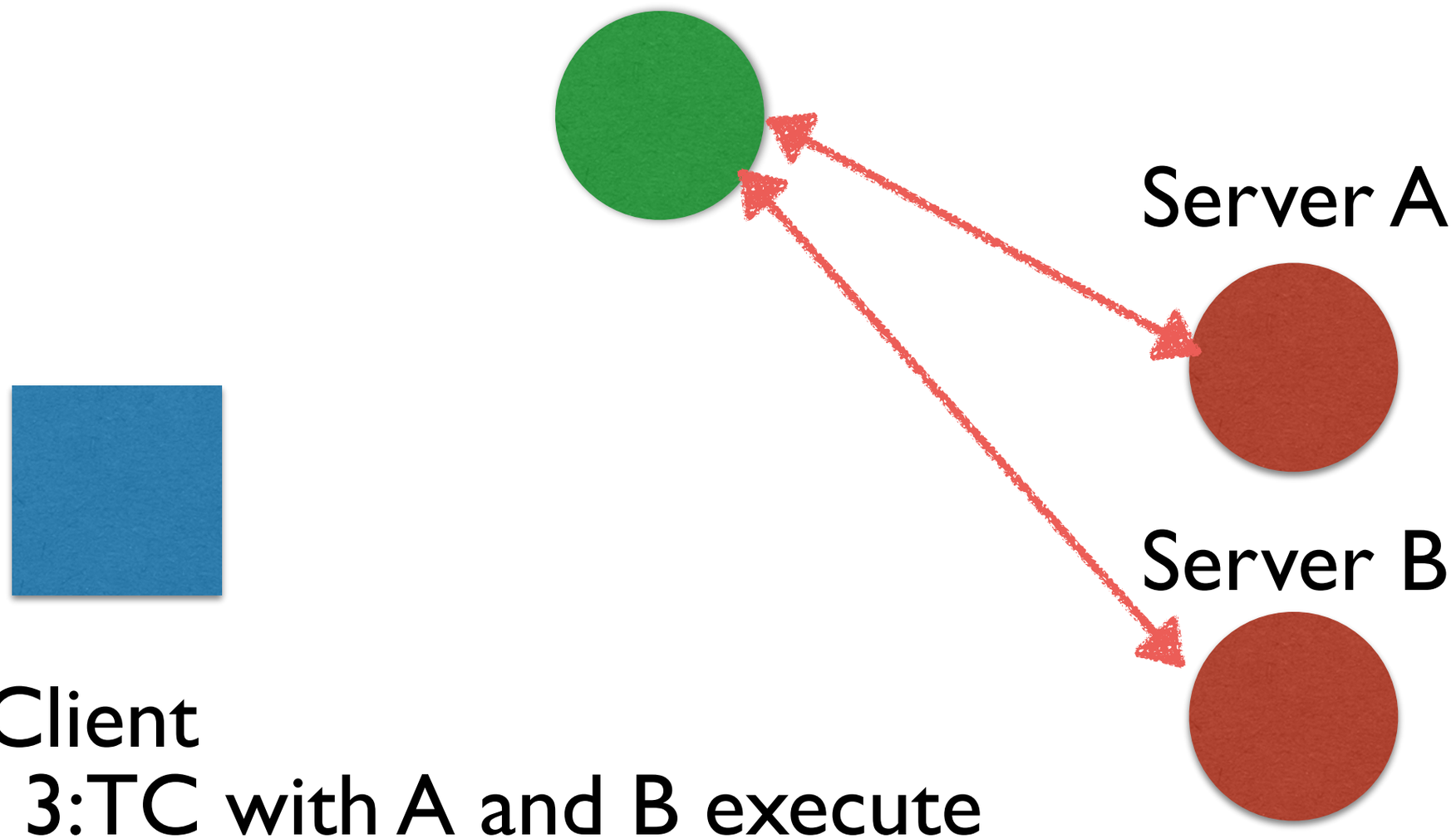
Single entity decides whether to commit
it prevents any chance of disagreement



Step 2: on end_transaction, client sends "go" to TC

The Idea

Single entity decides whether to commit
it prevents any chance of disagreement

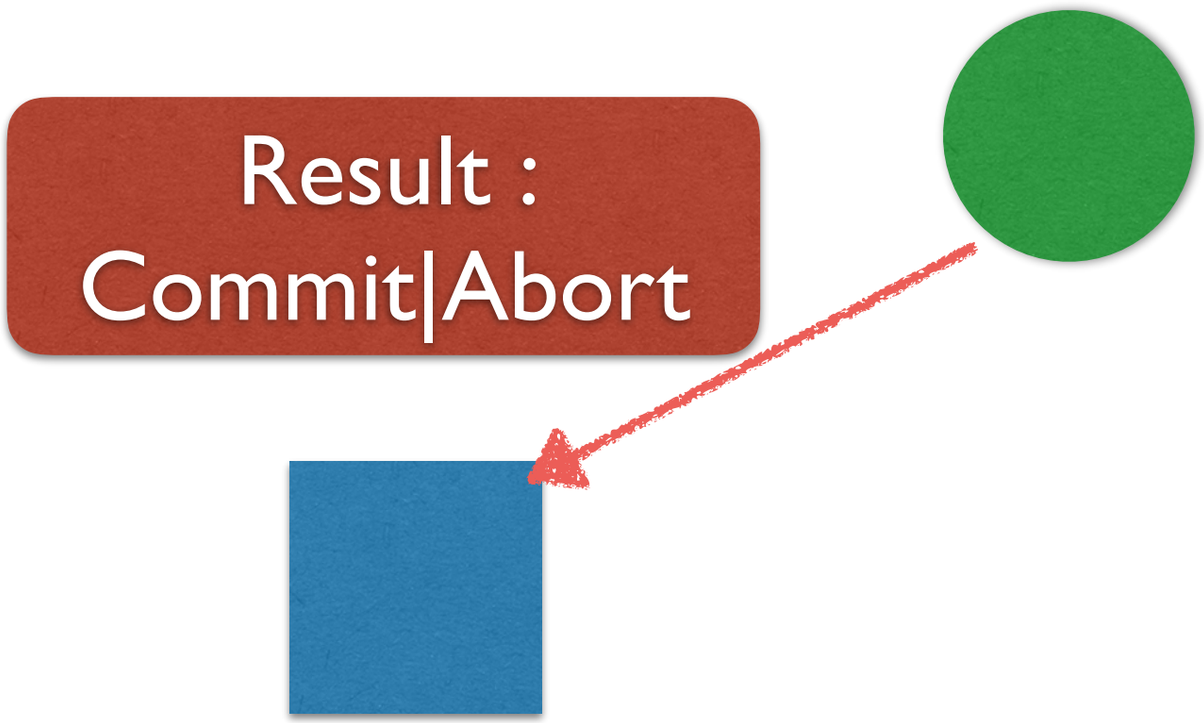


Step 3:TC with A and B execute distributed commit protocol

The Idea

Single entity decides whether to commit
it prevents any chance of disagreement

Transaction Coordinator (TC)



Server A

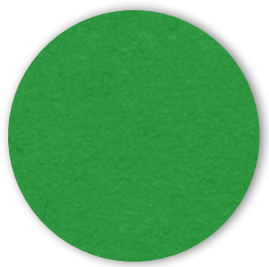
Server B

Client

Step 3:TC with A and B execute distributed commit protocol

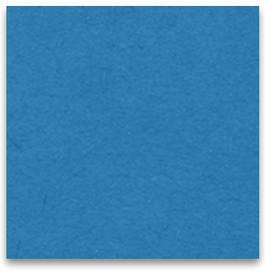
2PC No Failures

Transaction Coordinator (TC)



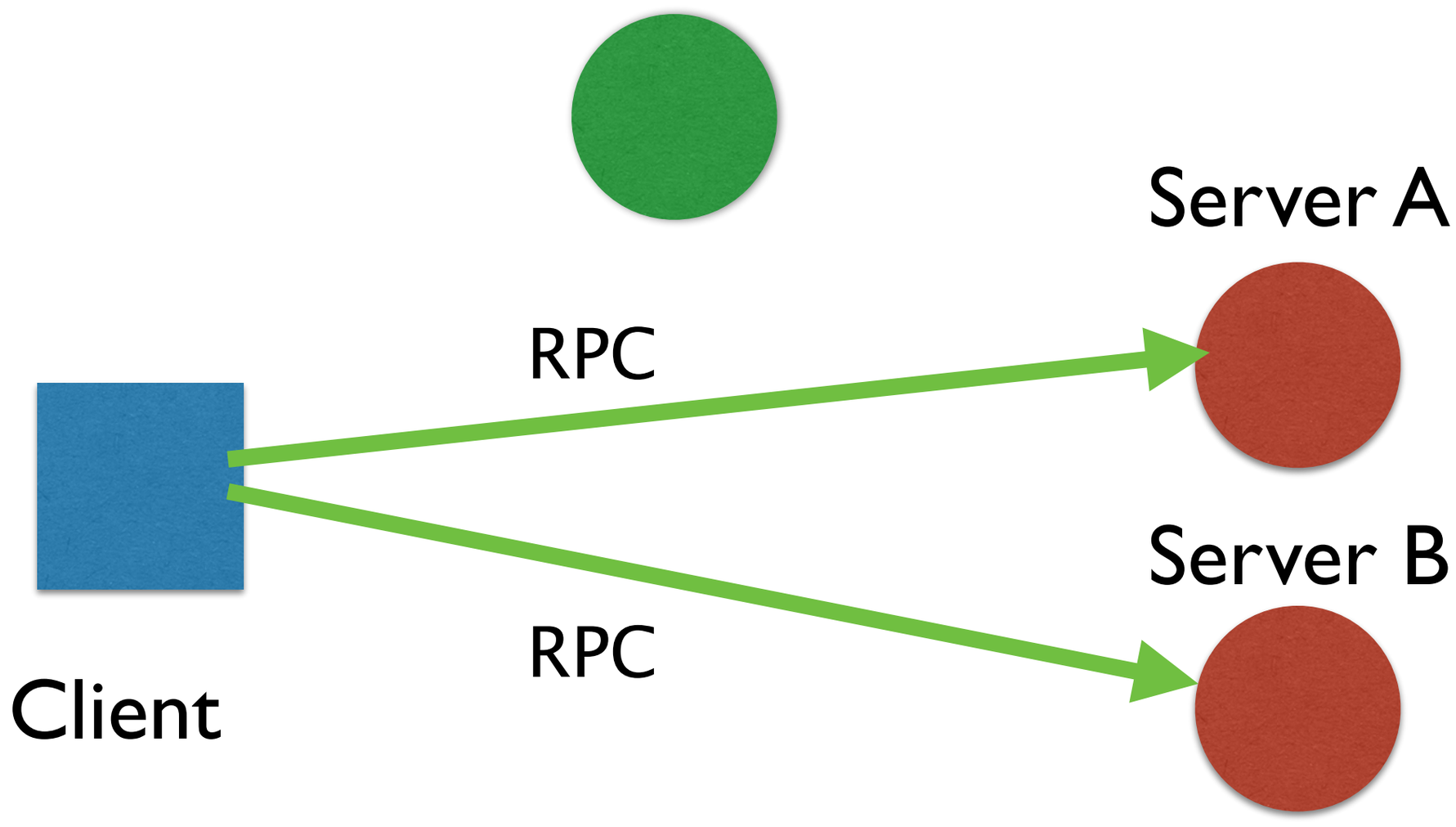
Server A

Server B

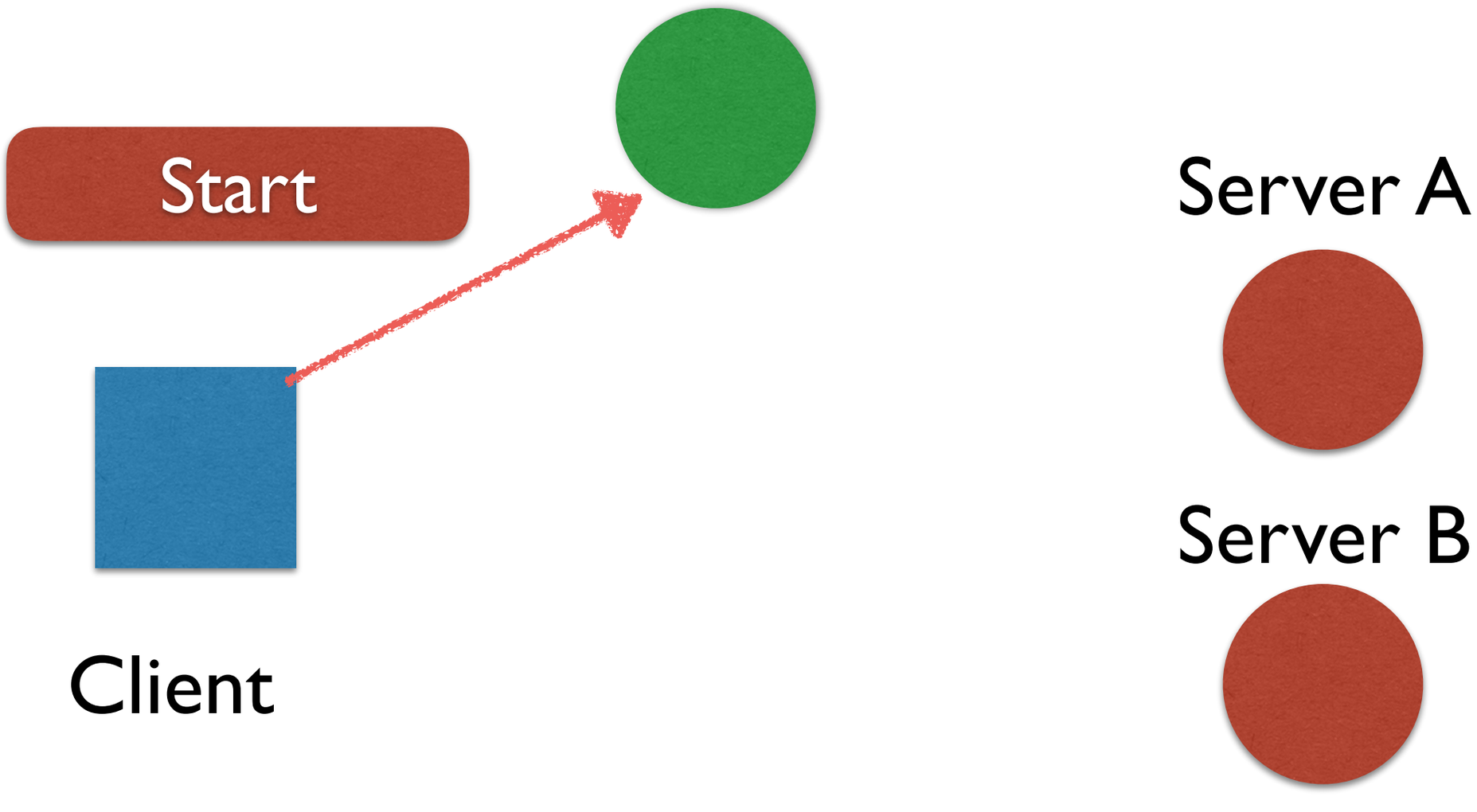


Client

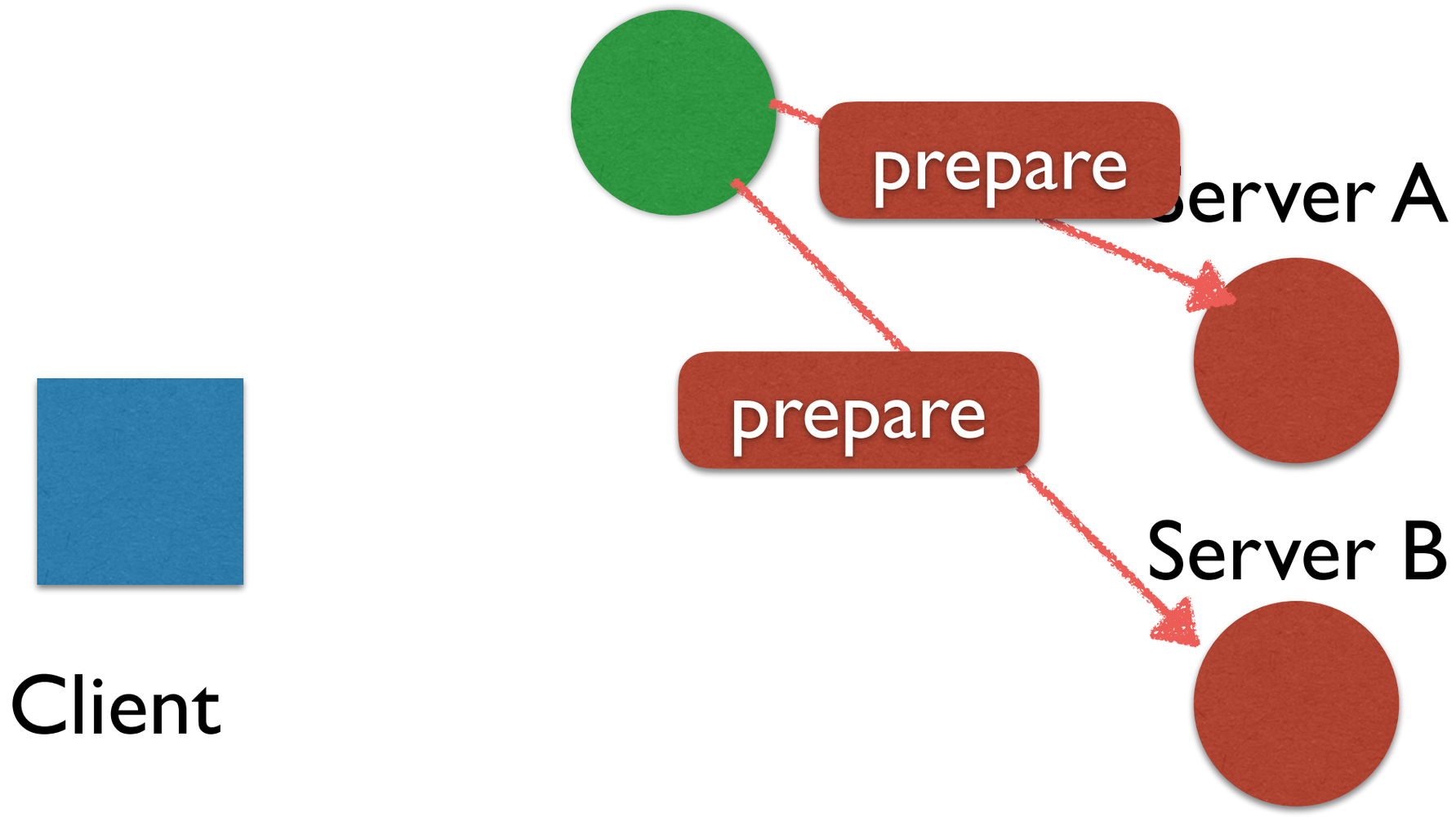
2PC No Failures



2PC No Failures



2PC No Failures

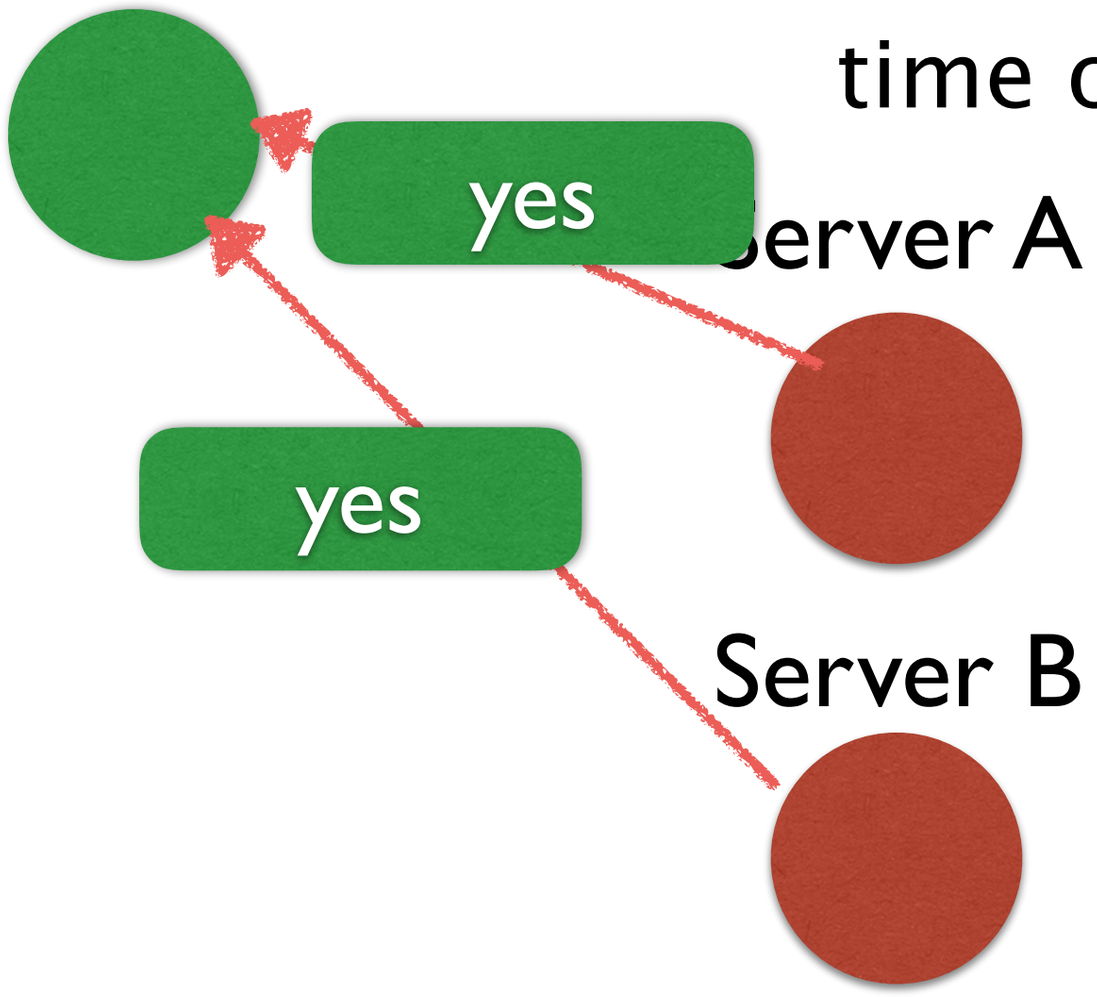


TC sends "prepare" messages to A and B

2PC No Failures

Transaction Coordinator (TC)

Respond "yes" if haven't crashed, no time outs, etc



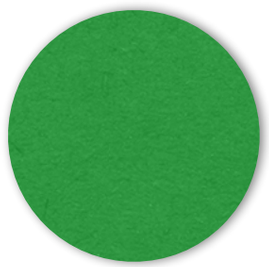
Client

A and B respond, saying whether they're willing to commit.

2PC No Failures

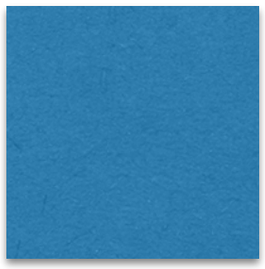
If A:”yes" && B:”yes” {
  "commit" messages.
}

Transaction Coordinator (TC)



Server A

Server B

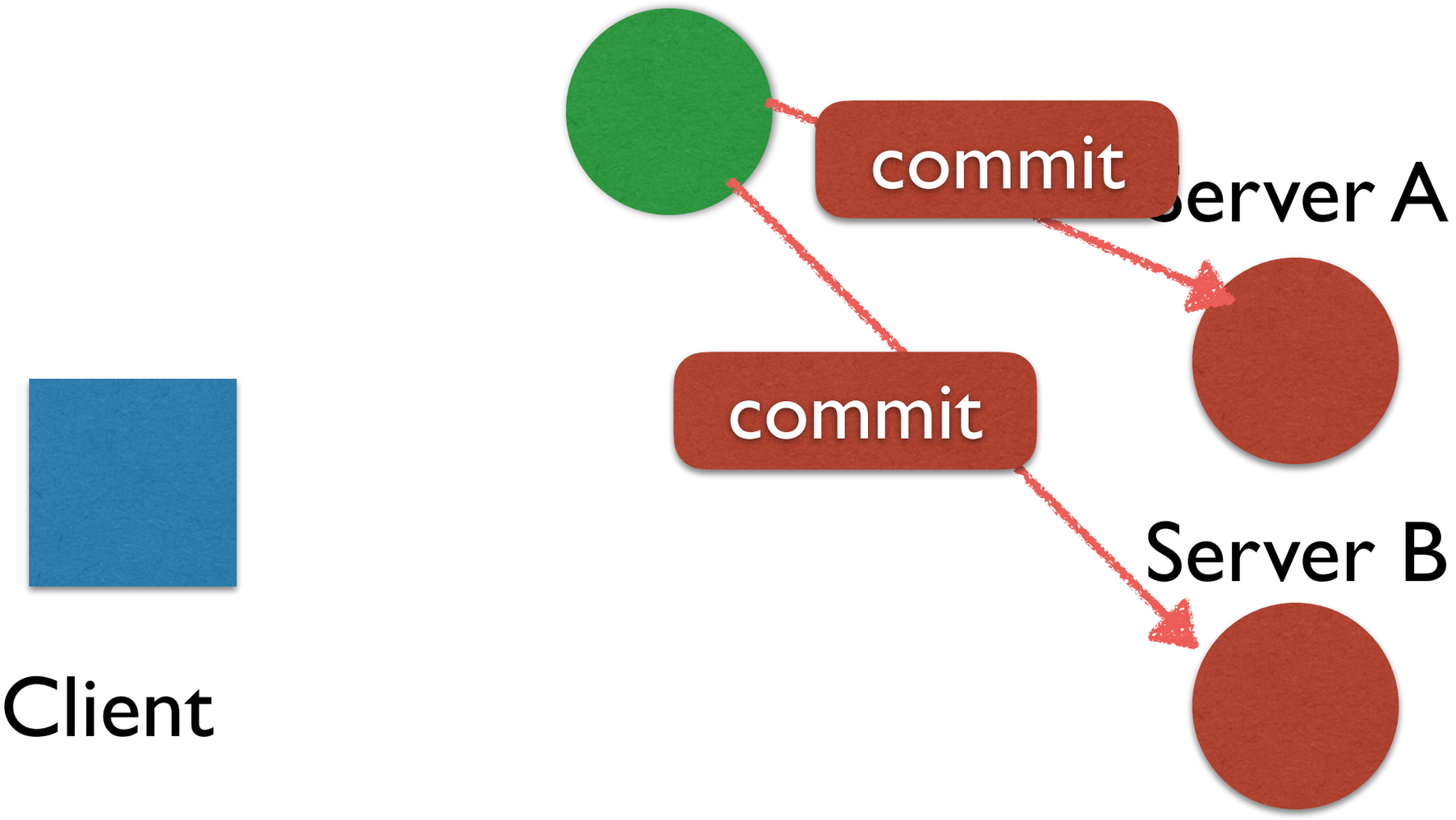


Client

A and B respond, saying whether they're willing to commit.

2PC No Failures

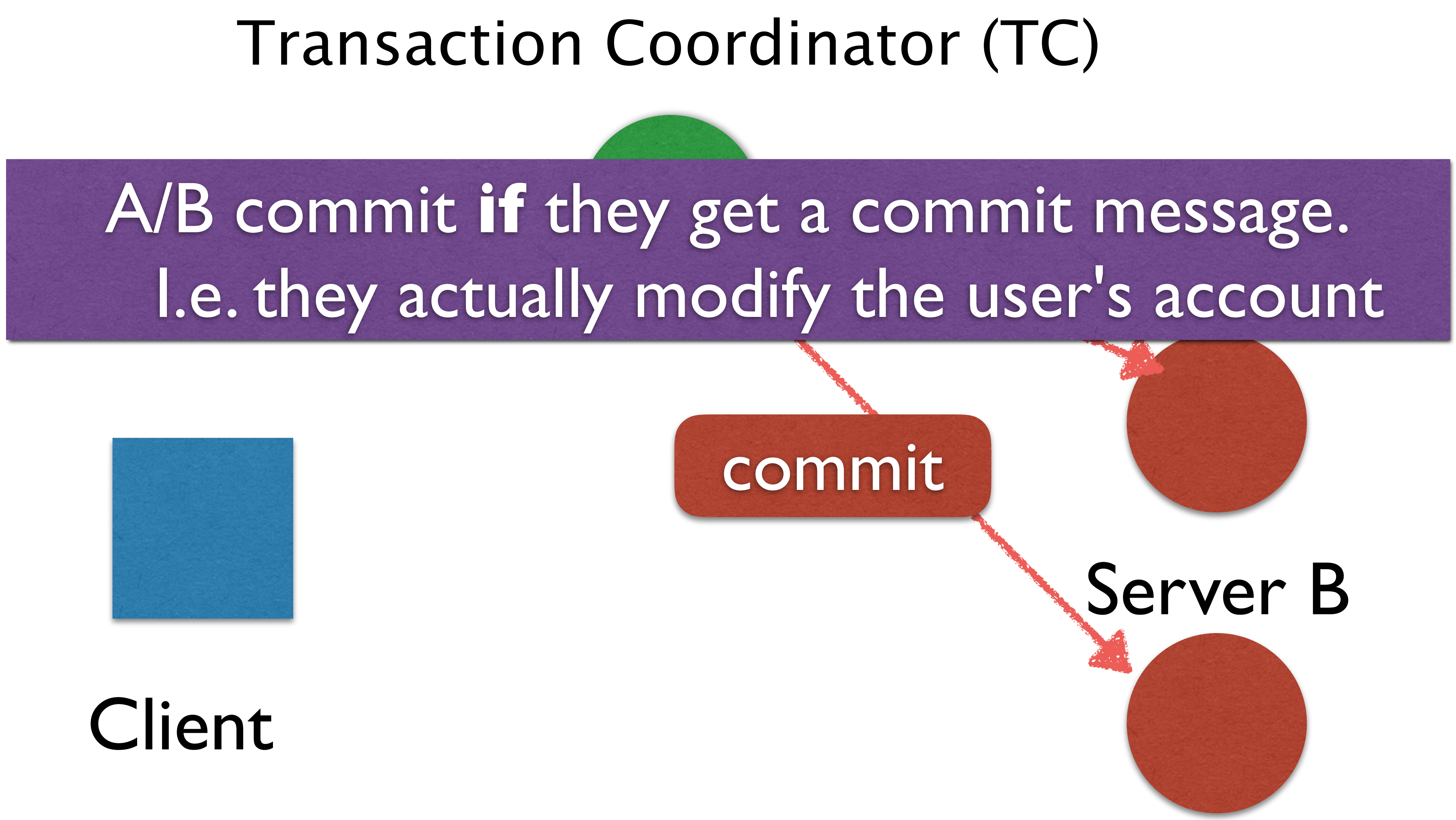
If A:”yes" && B:”yes” { send “commit” }
else { send “abort” }



A and B respond, saying whether they're willing to commit.

2PC No Failures

If A:"yes" && B:"yes" { send "commit" }
else { send "abort" }



A and B respond, saying whether they're willing to commit.

2PC No Failures

If A:”yes" && B:”yes” { send “commit” }
else { send “abort” }

Why is this correct so far?

Neither can commit unless they both agreed.

But crucial that neither changes mind after responding to prepare
Not even if failure!

Client

A and B respond, saying whether they're willing to commit.

2PC WITH Failures

What about failures?

- Network broken/lossy/slow
- Server crashes

What is our goal w.r.t. failure?

- Resume correct operation after repair
- I.e. recovery, *not* availability

Single symptom: timeout when expecting a message.

2PC WITH Failures

Where do hosts wait for messages?

1. TC waits for yes or no response to prepare message
2. A and B wait for prepare messages and commit/ abort messages

Terminating

CASE	ACTION
TC timeout for yes/no	abort
B timeout for prepare	abort
B timeout for commit/abort and B voted NO	abort
B timeout for commit/abort and B voted YES	BLOCK

Terminating

CASE	ACTION
TC timeout for yes/no	abort

TC has not sent any "commit" messages.
So TC can safely abort, and send "abort" messages.

A/B timeout while waiting for prepare from TC
have not yet responded to prepare, so TC can't have decided commit
so A/B can unilaterally abort
respond "no" to future prepare

B timeout for prepare	abort

Terminating

B timeout for commit/abort and B voted NO	abort
B timeout for commit/abort and B voted YES	BLOCK

If B voted "no", **it can** unilaterally abort.

Terminating

B timeout for commit/abort and B voted YES	BLOCK

If B voted “yes” Can unilaterally abort?

Terminating

B timeout for commit/abort and B voted YES	BLOCK

If B voted “yes” Can unilaterally abort?

NO!
TC might have gotten "yes" from both, and sent out "commit" to A, but crashed before sending to B.
So then A would commit and B would abort: incorrect.

Terminating

B timeout for commit/abort and B voted YES	BLOCK

If B voted “yes” Can unilaterally commit?

NO!

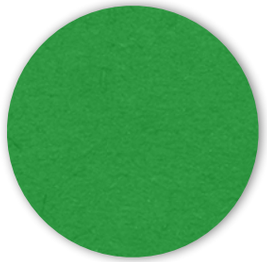
B can't unilaterally commit, either:

A might have voted “no".

So: if B voted "yes", it must "block": wait for TC decision.

2PC With Failures

Transaction Coordinator (TC)



Server A

Server B

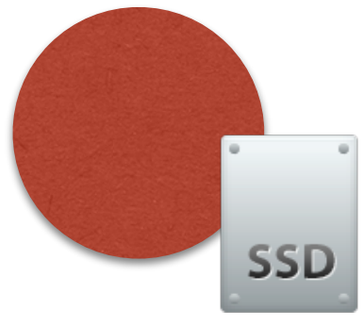
What if B crashes and restarts?
If B sent "yes" before crash, B must remember!
Can't change to "no" (and thus abort) after restart
Since TC may have seen previous yes and told A to commit

2PC With Failures

Transaction Coordinator (TC)

Server A

Server B



Participants must write persistent (on-disk) state:
B must remember on disk before saying "yes", including modified data.
If B reboots, disk says "yes" but no "commit", B must ask TC.
If TC says "commit", B copies modified data to real data.

2PC With Failures

Transaction Coordinator (TC)

What if TC crashes and restarts?
If TC might have sent "commit" or "abort" before crash,
TC must remember!
And repeat that if anyone asks
(i.e. if A/B/client didn't get msg).
Thus TC must write "commit" to disk before sending commit msgs.
TC can't change its mind since A/B/client may have already acted.

This protocol is called "two-phase commit".

* All hosts that decide reach the same decision
* No commit unless everyone says "yes".
* TC failure can make servers block until repair.

Two Phase Commit

Used in sharded DBs when a transaction uses data on multiple shards

But it has a bad reputation:

slow because of multiple phases / message exchanges

locks are held over the prepare/commit exchanges; blocks other xactions

TC crash can cause indefinite blocking, with locks held

Thus usually used only in a single small domain

E.g. not between banks, not between airlines, not over wide area

Better transaction schemes are an active area of research

Raft and two-phase commit solve different problems!

Use Raft to get high availability by replicating
i.e. to be able to operate when some servers are crashed
the servers all do the *same* thing

Use 2PC when each participant does something different
And *all* of them must do their part

2PC does not help availability
since all servers must be up to get anything done

Raft does not ensure that all servers do something
since only a majority have to be alive