Distributed Consensus: Two-phase Commit

Rong Chen

Institute of Parallel and Distributed Systems

Shanghai Jiao Tong University

http://ipads.se.sjtu.edu.cn/rong_chen

Credits: some slides from Jinyang Li (NYU)

Concurrency so far

Crash recovery

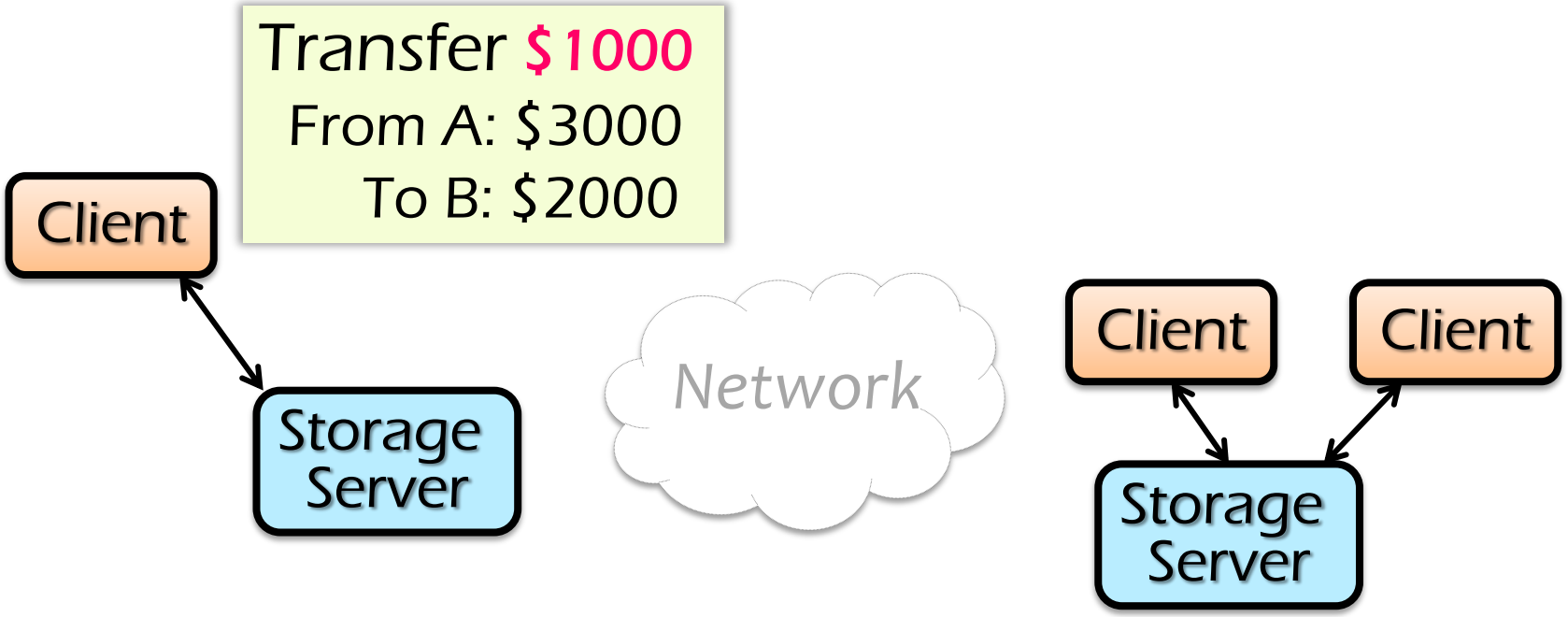
- A transaction **READ**/**WRITE** to many objects is atomic w.r.t failure

Concurrency control

- Serializability of multiple transactions
- **Two phase locking** (2PL)
- **Snapshot Isolation** (SI)

How about atomicity and concurrency control in distributed systems?

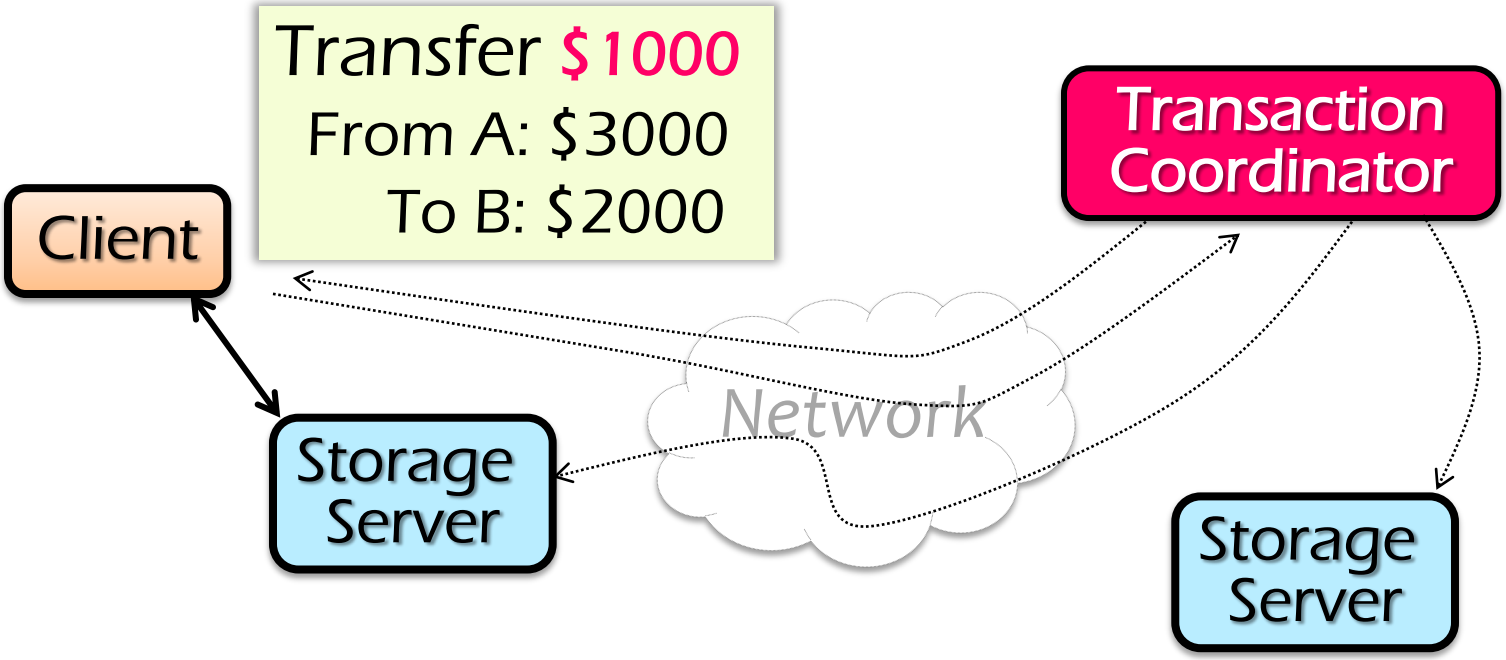
Example



Client desire

- Atomicity: transfer either happens or not at all
- Concurrency control: maintain serializability

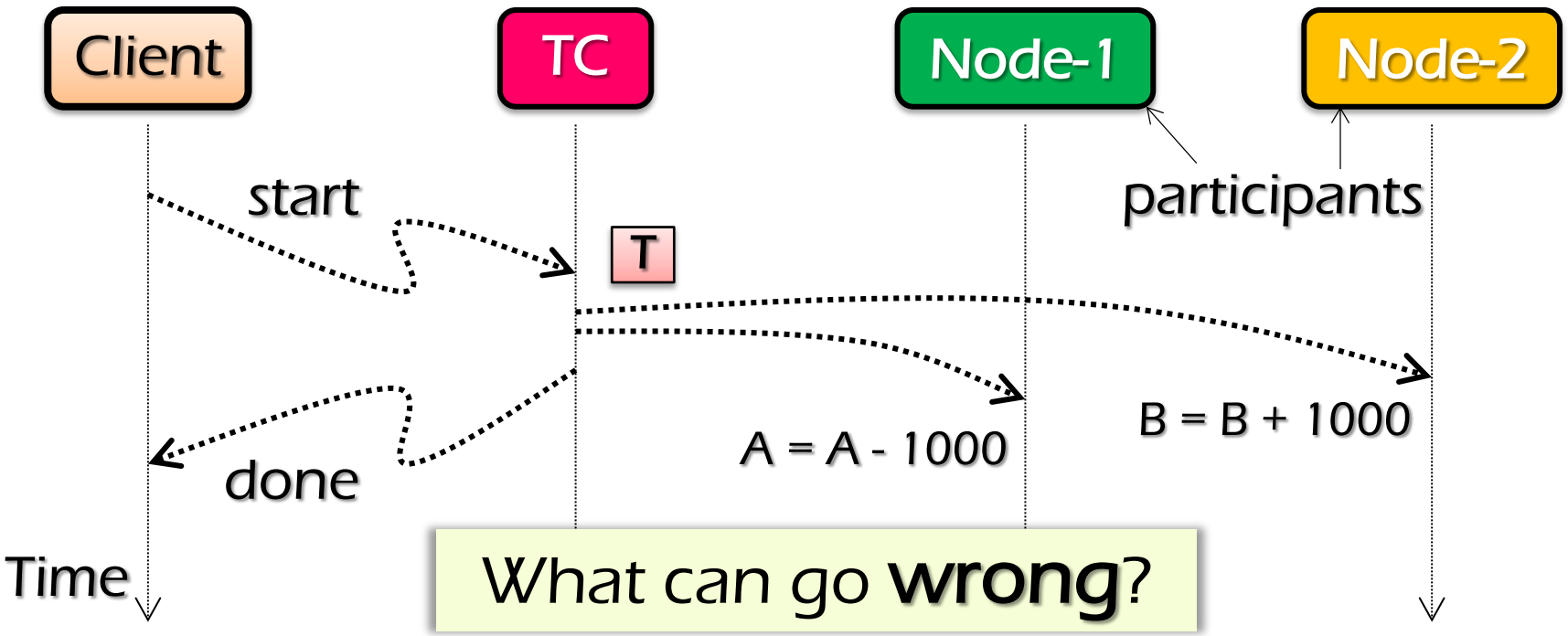
Example



Transaction Coordinator (TC) desire

- Begin transaction
- Responsible for commit/abort
- ...

One-phase Commit



1. A does not have enough money
2. Node has crashed
3. Coordinator crashed
4. Some other client is reading/writing A or B

Reasoning about Correctness

TC, Node-1 and Node-2 each has a notion of committing

Correctness

- If one **COMMIT**s, no one **ABORT**s
- If one **ABORT**s, no one **COMMIT**s

Performance

- If no failures, Node-1 and Node-2 can **COMMIT**, then commit
- If failures happen, find out outcome soon

Two-phase Commit (2PC)

The commit-step itself is two phase

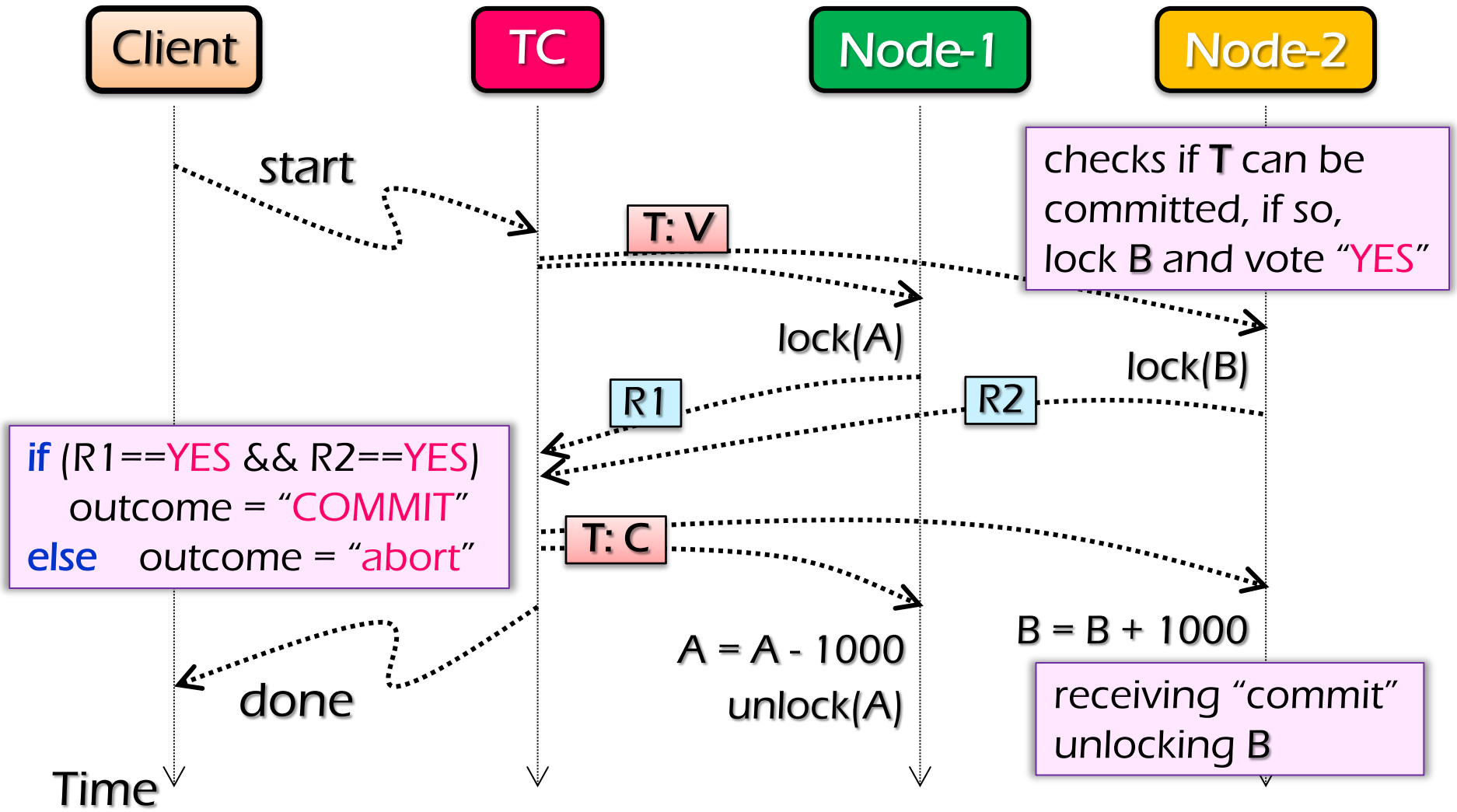
Phase-1: **Voting**

- Each participant **prepares** to commit, and **votes** on whether or not it can commit

Phase-2: **Committing**

- Each participant actually **commits** or **aborts**

Example


Client
TC
Node-1
Node-2
start
T: V
checks if T can be committed, if so, lock B and vote "YES"
lock(A)
lock(B)
R1
R2
if (R1==YES && R2==YES)
outcome = "COMMIT"
else outcome = "abort"
T: C
A = A - 1000
unlock(A)
B = B + 1000
done
receiving "commit" unlocking B
Time

The Voting Phase

TC asks each **participant** canCommit(T) ?

Participants must prepare to commit using permanent storage before answering YES

- Objects are still locked
- Once a **participant** votes "**YES**"
  → it is not allowed to cause an **ABORT**

Outcome of **T** is uncertain until doCommit(T) or doAbort(T)

- Other **participants** might still cause an **ABORT**

Don't forget about their response after **REBOOT**

The Committing Phase

TC collects all votes

- If unanimous "**YES**", cause **COMMIT**
- If any **participant** voted "**NO**", cause **ABORT**

The fate of the **T** is decided atomically at the **TC**, once all **participants** vote

- **TC** records fate using permanent storage
- Then broadcasts *doCommit(T)* or *doAbort(T)* to **participants**

Don't forget about its decision after **REBOOT**

Performance Issues

What about **timeout**?

TC times out waiting for **participant**'s response	**Participant** times out waiting for **TC**'s outcome message

Handling **timeout** on Participants

- Can TC unilaterally decide to **COMMIT**?
- Can TC unilaterally decide to **ABORT**?

Handling **timeout** on TC

- If Participants responded with "**NO**", can it unilaterally **ABORT**?
- If Participants responded with "**YES**", can it unilaterally **COMMIT**/**ABORT**?

Termination Protocol

Execute termination protocol if Node-1 times out on TC and has voted "**YES**"

Node-1 sends "status" message Node-2

1. N-2 has received "**COMMIT**/**ABORT**" from TC
2. N-2 has not responded to TC
3. N-2 has responded with "**NO**"
4. N-2 has responded to "**YES**"

Resolve most failure cases on **Participant** except sometimes when **TC** fails

Crash and Reboot

TC and **participants** must log **protocol progress**

What and when does **TC** log to disk?

What and when does **Participants** log to disk?

if **TC** finds no "**COMMIT**" on disk → doAbort(T)

if **TC** finds "**COMMIT**" on disk → doCommit(T)

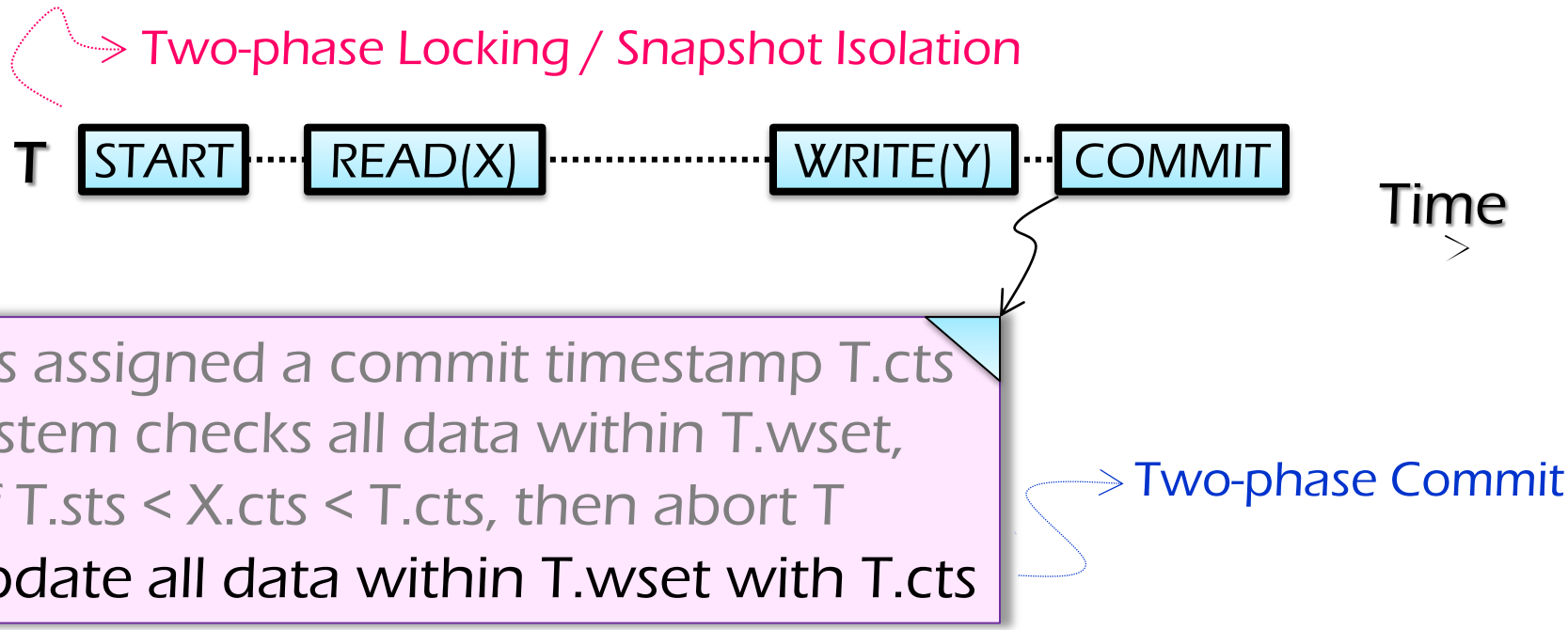
if **Participant** find no "**YES**" on disk → abort(T)

if **Participant** find "**YES**" on disk

→ Run termination protocol to decide

Concurrency Control with 2PC

Each **participant** uses two-phase locking (2PL) or snapshot isolation (SI) for the **READ**/**WRITE** objects, and two-phase commit (2PC) for the **COMMIT** process



Summary

All **Participants** that decide reach the same decision

No **COMMIT** unless everyone says "**YES**"

No failure and all "**YES**", then **COMMIT**

If failures, then repair

- Wait long enough (timeout) for recovery
  → some decisions (**COMMIT**/**ABORT**)

SINFONIA: A New Paradigm for Building Scalable DS

Marcos Aguilera, "*Sinfonia: a New Paradigm for Building Scalable Distributed Systems*". SOSP, **2007**

Problems Addressing

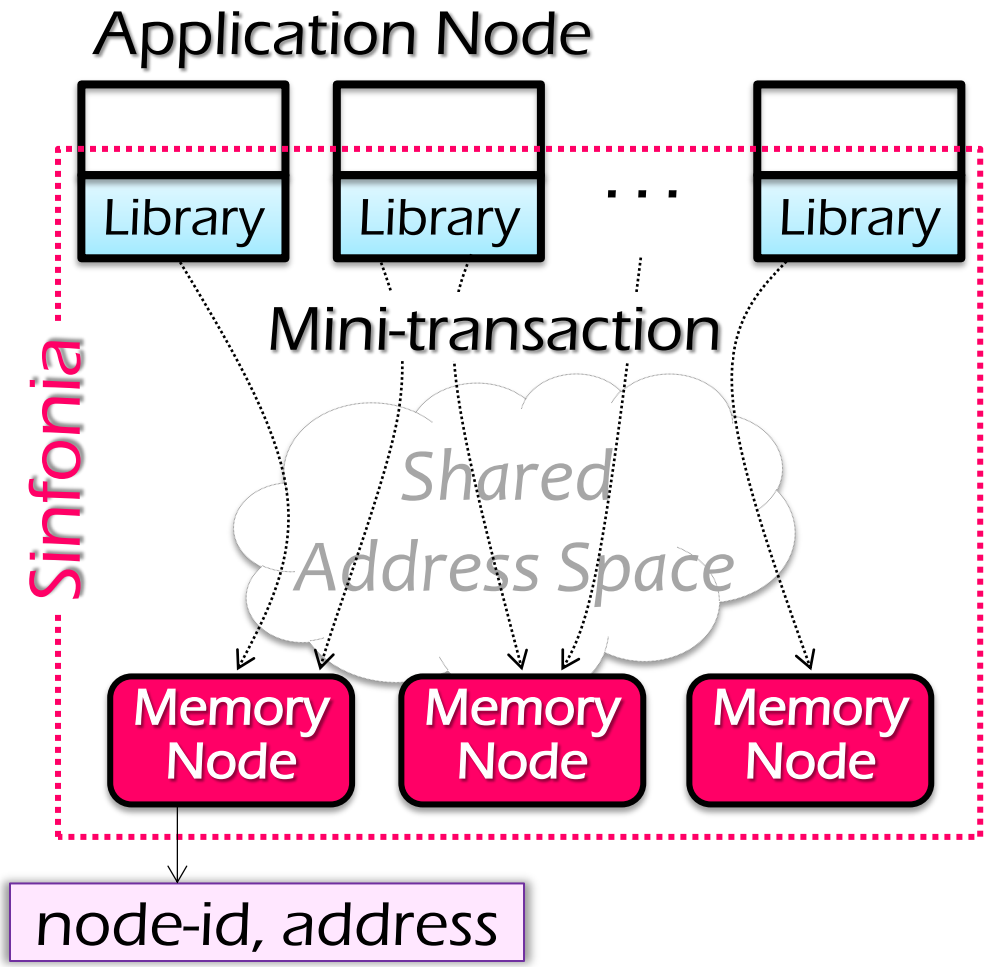
Target: the systems or infrastructural apps within a datacenter

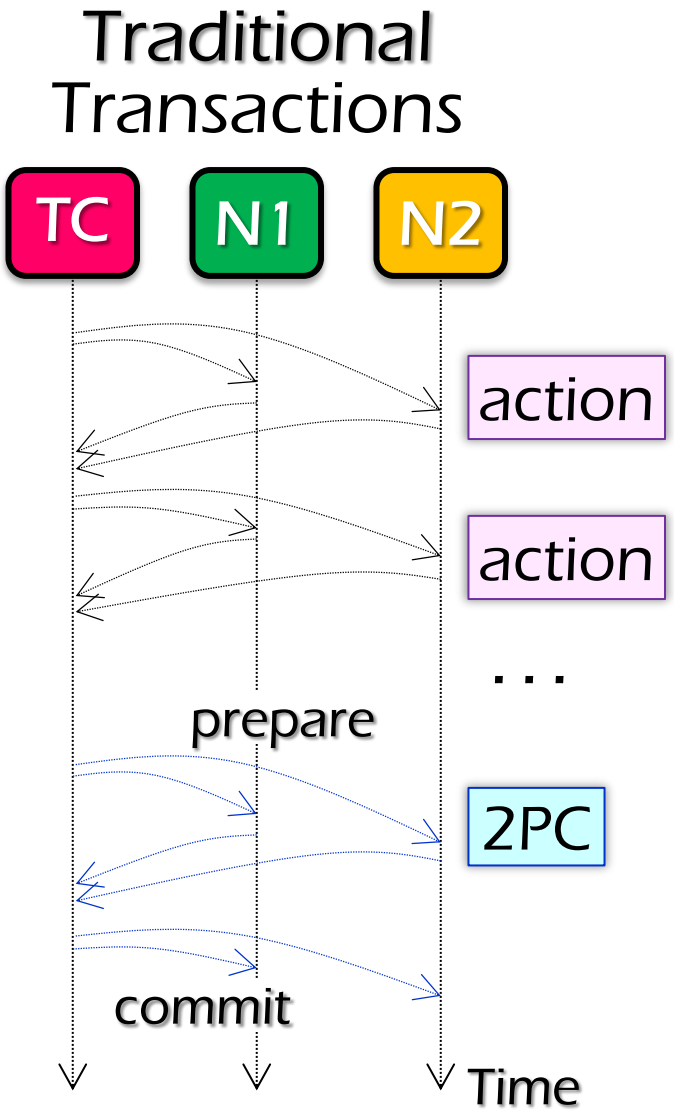
Sinfonia: a shared data store service

- Span multiple servers
- Address space sharding
- Replicated with consistency guarantees

Goal: reduce development efforts for system programmers

Sinfonia Architecture





Sinfonia's Mini-transactions

Provide **atomicity** and **concurrency** control

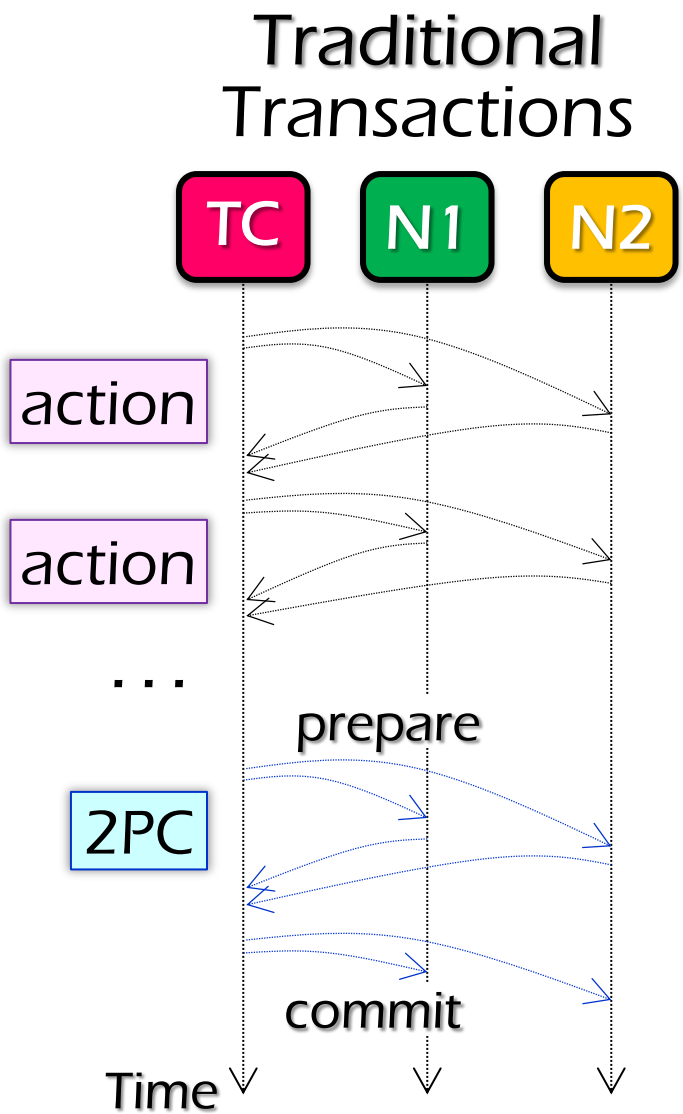
Trade off **expressiveness** for **efficiency**

- Fewer **network** round-trip-times to execute
- Less **flexible**, **general-purpose** than **traditional** transactions

Mini-transactions

- A **lightweight**, **short-lived** type of transaction
- Over **unstructured** data

Transaction Comparison

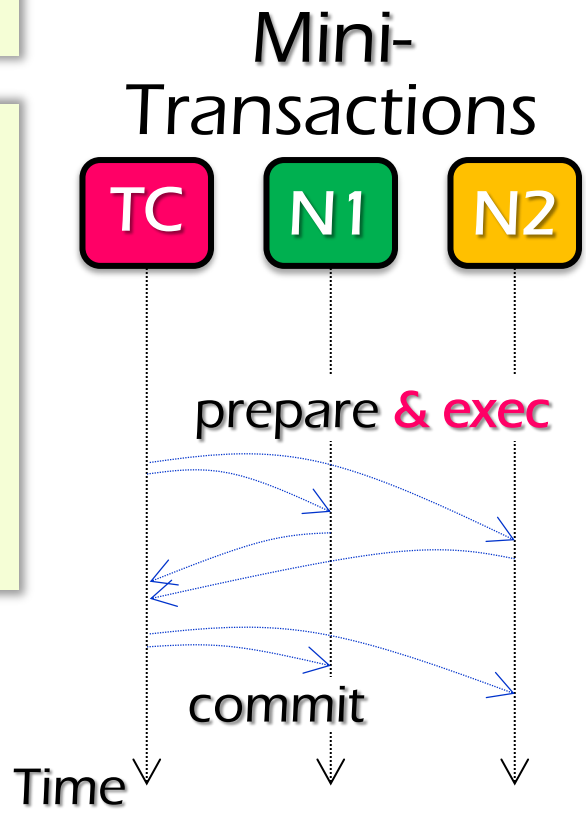



```
START T
  if (a > 0)  b = a * a;
  for (i = 0; i < a; i++)
      b += i;
END T
```

general but expensive

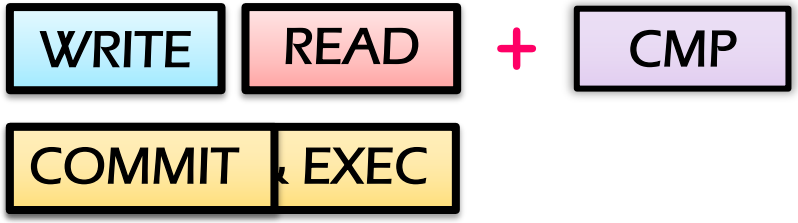
```
START T
  if (a == 3000 &&
          b == 2000) {
      a = 2000;
      b = 3000;
  }
END T
```

less general but efficient



Mini-transactions Detail

Mini-Transaction:




```
t = new tx()
a = t->read(n1:A, 4)
b = t->read(n2:B, 4)
if (a==3000 && b==2000){
  t->write(n1:A, 4, 2000)
  t->write(n2:B, 4, 3000)
}
stat = t->commit()
```

```
t = new mini_tx()
t->cmp(n1:A, 4, 3000)
t->cmp(n2:B, 4, 2000)
t->write(n1:A, 4, 2000)
t->write(n2:B, 4, 3000)
Stat = t->exec_commit()
```

Semantics of a mini-transaction

1. Check data by **CMP** ops
2. if all match then → retrieve data by **READ** ops
   modify data by **WRITE** ops

Potential Uses of Mini-Tx

1. **Atomic** swap operation
2. **Atomic** read of multiple data
3. **Try** to acquire a lease
4. **Try** to acquire multiple leases **atomically**
5. **If** lease is held then change data
6. **Validate** cache then change data
7. **…**

Sinfonia's 2P Protocol

Transaction coordinator is at application node instead of memory node

- Saves one round-trip-time (RTT)

Problems: crashed **TC** blocks transaction progress

- Application nodes are less reliable than memory nodes

Sinfonia's 2P Protocol

TC keeps no log *transfer between app-nodes (failure)*

A transaction is committed iff
all **participants** have "**YES**" in their logs

Recovery **TC** cleans up

- Ask all **participants** for existing vote
  (or vote "**NO**" if not voted "**YES**")
- **COMMIT** iff all vote "**YES**"

Transaction blocks if a memory node crashes

- Must wait for memory node to recovery

Sinfonia's 2P Protocol

TC keeps no log

transfer between app-nodes (failure)

A transaction is committed iff
all **participants** have "**YES**" in their logs

Rec

- **COMMIT** iff all vote "**YES**"

Question: What's the difference between coordinator in mini-transaction's 2PC protocol and standard 2PC protocol?

Transaction blocks if a memory node crashes

- Must wait for memory node to recovery

Example: SinfoniaFS

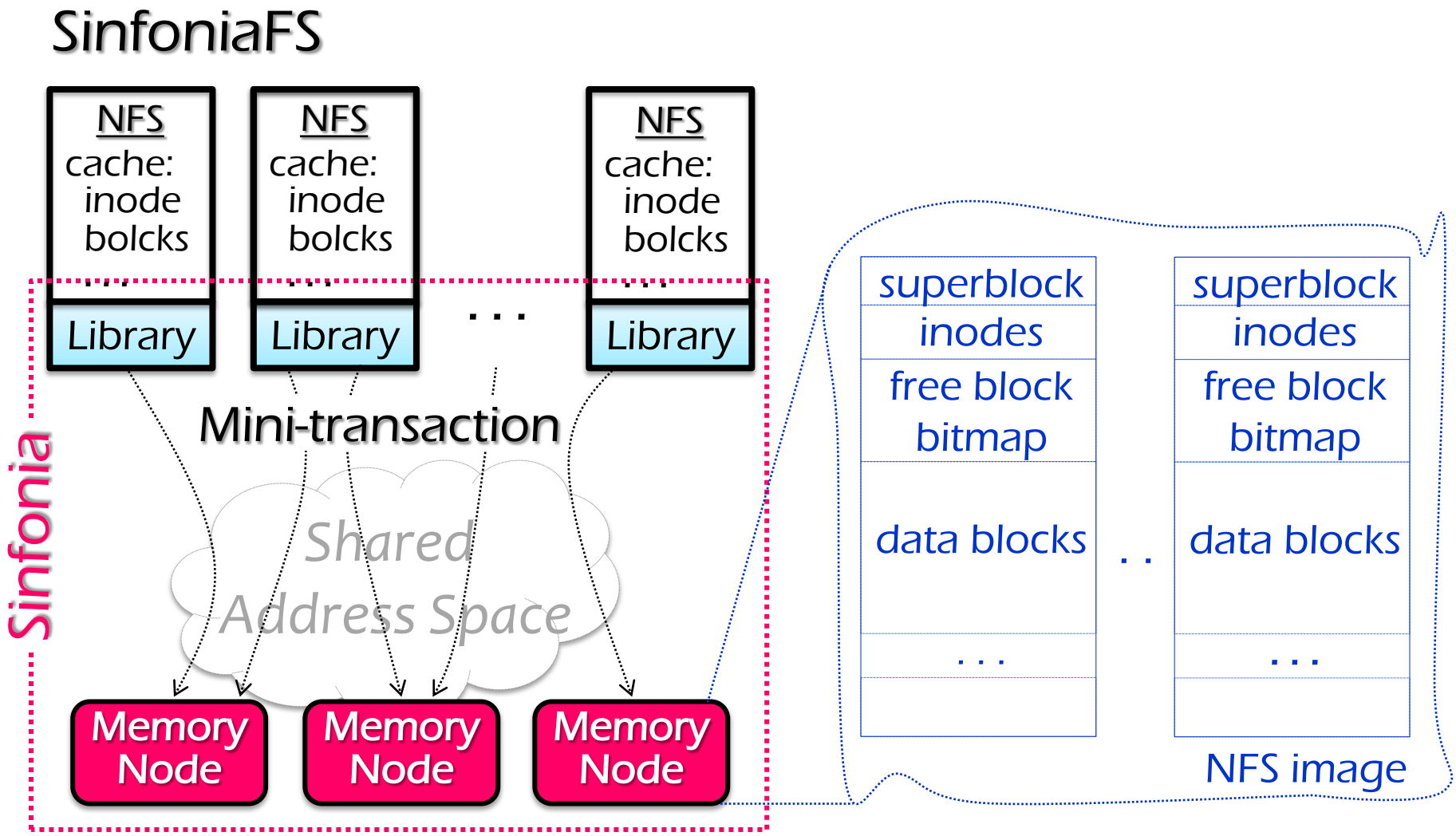
SinfoniaFS: file system based on Sinfonia

- Hosts share the same set of files (in Sinfonia)
- Scalability: performance improves with more memory nodes
- Fault tolerance

SinfoniaFS exports a **NFS** interface

- Each NFS operation → one mini-transaction

SinfoniaFS Architecture



Example: setattr

```
void setattr(ino_t inum, sattr_t newattr){
  do {
    addr = ... // get address of inode
    curr_version = inode->version;
    t = new mini_transaction();
    t->cmp(addr, 4, curr_version);
    t->write(addr, 4, curr_version + 1);
    t->write(addr, 20, newattr);
  } while(t->status == fail);
}
```

General use of mini-transaction in **SinfoniaFS**

1. If local cache is empty, load it
2. Make modifications to local cache
3. Issue a mini-transaction to check the validity of cache, apply modification
4. If mini-transaction fails, reload cached item and try again

Next Lecture

Topic: Distributed Consensus (Paxos)

THANKS