Transactions

Distributed Transactions

- Models above dictate results of **single** operation
- What about **multiple** operations?
- Transaction is a **sequences of operations** which appears as a **single unit of work** (i.e., a composite operation)
- Expresses more complex semantics
  - Transferring money between two accounts:
    ```
    account1.balance -= sum;
    account2.balance += sum;
    ```
  - Trip reservation (flight + hotel)

ACID Properties

A Transaction Processing System Fulfills a Number of Properties

- **Atomicity**: either the **entire transaction** is executed, or not at all
- **Consistency**: transaction must execute operations in accordance to its **consistency contract**
- **Isolation**: effect of a transaction must **not be visible** to others until it is completed
- **Durability**: Effects of committed transaction remains in system even in presence of **failures**

OPTIMISTIC CONCURRENCY CONTROL

Example: Violation of consistency

Client 1

get(Courses, ECE419, &r) **r.Mark++** set(Courses, ECE419, r)

<ECE419, 96>

Time

In storage server: {ECE419, 95} {ECE419, **96**}

Client 2

get(Courses, ECE419, &r) **r.Mark++** set(Courses, ECE419, r)

<ECE419, 96>

Time

In storage server: {ECE419, 95} {ECE419, **96**}

Desired:

Initially: Mark is 95

r.Mark++: Mark is 96

r.Mark++: Mark is **97**

Courses (initial state):

Course	Mark
ECE419	**95**
ECE344	92
ECE451	87

Optimistic Concurrency Control

Version Records

```
struct storage_record {
        char value[MAX_VALUE_LEN];

        uintptr_t version[8];
};
```

- Track record versions across updates
- Each record update creates a new version

Versioned Records

set(Courses, ECE419, r)

r.v1

set(Courses, ECE419, r)

r v2

set(Courses, ECE419, r)

r v3

We can represent record versions with integers

0, 1, 2, 3, 4 …

Record Version Management

- When a record is **first created**, its **version is initialized**
- When a record is **updated**, its **version is incremented**
- When a **record is read**, its **version is returned** to the caller (the application)

Reading Versioned Records

- get(table, key, &r)
  - If the call succeeds:
    - r is {$value_1$, …, $value_n$, ***version***}
  - Otherwise r is undefined

In the application

Updating Versioned Records

get(table, key, &***r***)

Updates to r's values based on application logic

set(table, key, ***r***)

- Outcomes of the set call
  - Successfully completed
  - An ERROR condition
  - **ERR_TRANSACTION_ABORT**
    - **Handle by re-driving the update sequence.**

Example

Client 1

get(Courses, ECE419, &r) **r.Mark++** set(Courses, ECE419, r)

<ECE419, 96, v1>

{ECE419, 95, v1} {ECE419, **96, v2**}

Client 2
(1st attempt)

get(Courses, ECE419, &r) **r.Mark++** set(Courses, ECE419, r)

<ECE419, 96, v1>

{ECE297, 95, v1}

Transaction Abort Error

Client 2
(2nd attempt
AFTER
Client 1's set())

get(Courses, ECE419, &r) **r.Mark++** set(Courses, ECE419, r)

{ECE419, 96, v2} {ECE419, **97, v3**}

Courses

Course	Mark
ECE419	95
ECE344	92
ECE451	87

Desired:

Initially: Mark is 95
r.Mark++: Mark is 96
r.Mark++: Mark is **97**

In the server

Updating Versioned Records

- Server receives a **set(table, key, r)**
- Find table, find key, …
- Check r’s version against the version stored
- If there is a mismatch
  - Return with **ERR_TRANSACTION_ABORT**
  - Otherwise
    - Perform the update
      - By incrementing the version & storing
    - Return success

DISTRIBUTED COMMIT PROTOCOL

Distributed Commit Protocols

- A transaction consist of three steps:
  - **Begin**: start transaction
  - **Operations**: sequence of operations is executed at various machines (e.g., a distributed DB)
  - **Commit/Abort**: operations are applied at every machine, or cancelled/aborted
- To guarantee atomicity, we require that **every node** involved in transaction to either **commit or abort**
- Sounds familiar? (consensus…)

2-Phase Commit (2-PC)

- A commit protocol with **two phases**
- Requires a **coordinator**
- **Participants** are nodes involved in transaction
- Phase 1: **Voting Phase**
  - Each **participant prepares** to commit and **votes** whether it can commit or not
- Phase 2: **Commit Phase**
  - Each **participant commits** or **aborts**

Voting Phase

- Coordinator asks each participant: ***canCommit***?
- Participant replies *YES* or *NO*
  - May vote *NO* if it cannot apply operations it received for some reason
- Participant is then standing by until it receives an instruction in next phase
  - May require participant to lock some data

Commit Phase

- Coordinator collects all votes
- If votes are unanimous “YES”:
  - Coordinator sends ***doCommit*** to all participants
- Else:
  - Coordinator sends ***doAbort*** to all participants

- Unlike Paxos, 2-PC requires unanimity!

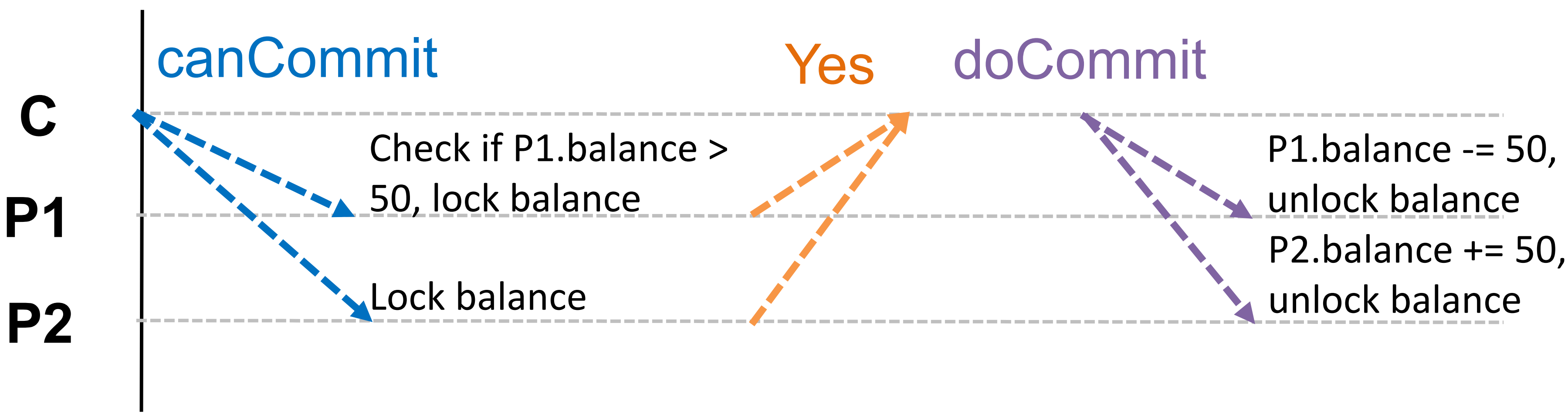
2-PC Example

Bank transfer:

P1.balance = 100

P2.balance = 50

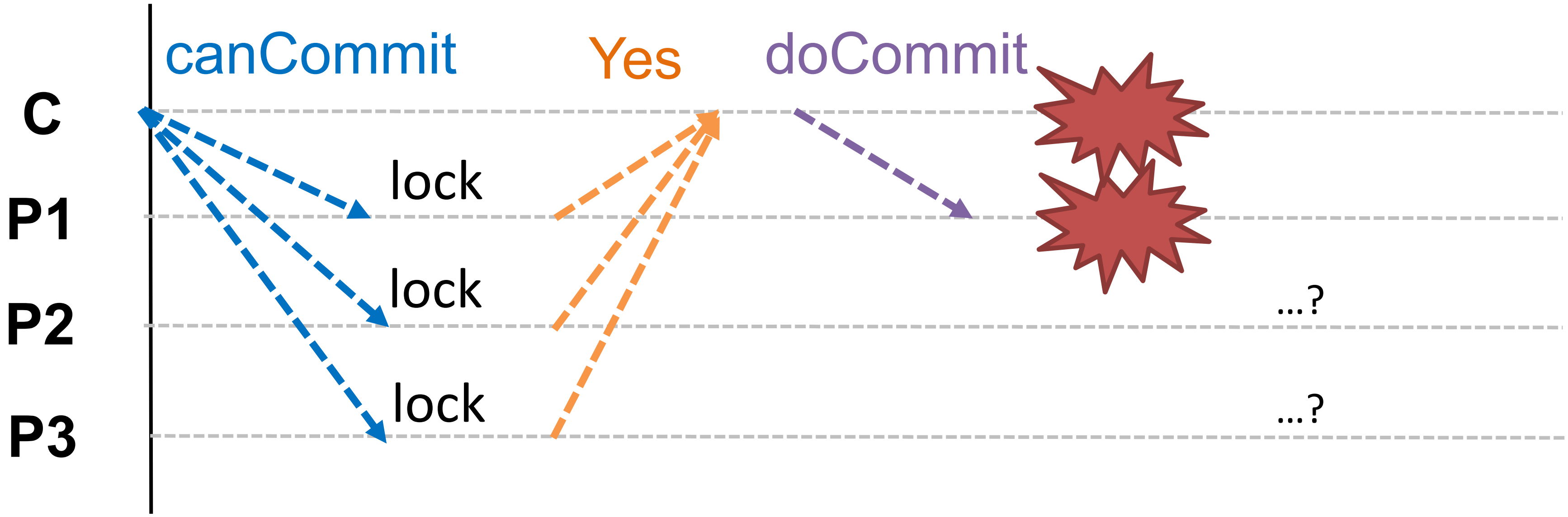
Transfer 50 from P1 to P2



Problems with 2-PC

- 2-PC is **safe**, but not **live** (like Paxos)
  - Using **write-ahead logs**, 2-PC can progress if nodes **eventually recover**
  - Participants could also communicate with each other to share coordinator's decision if it failed
- However, it is a **blocking protocol:**
  - In some situations (cf. next slide), participants which are still alive must **continue locking** until failures are resolved
  - This severely limits **availability**!
- Fortunately, Paxos could be used to support **transaction commit** (Paxos Commit) → **How?**

2-PC Blocking Example



P2 and P3 are alive, are aware that P1 and C failed, but cannot decide to **commit or abort** because they do not know if P1 committed or not! If they make the wrong decision now, outcome may be inconsistent with P1's decision when P1 recovers. **Thus, P2, P3 must block until C or P1 recovers**.