SE3331-1 (2023 Fall)

Consistency under crash: All-or-nothing atomicity

Xingda Wei, Yubin Xia

IPADS, Shanghai Jiao Tong University

https://www.sjtu.edu.cn

Credits: Rong Chen@IPADS

Review: what is a strong consistency model

It's easy for users to reason about correctness assuming

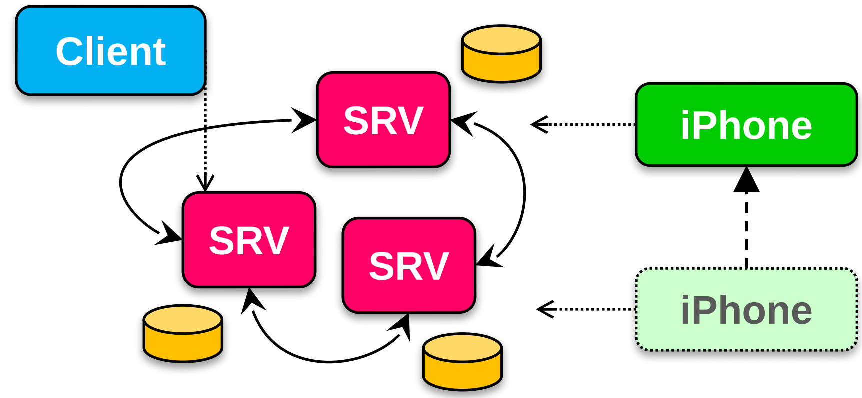
- Everything has only one-copy
- The overall behavior is equivalent to **some serial behavior**

Review: eventual consistency

A specific of weak consistency model, informally:

- All servers eventually receives all writes, and servers holding the same set of writes will have the same data contents
- Thus, if no new updates are made to the data, eventually all accesses will return the last update value

Review: basic execution of eventual consistency



Setup: replicated KVS

- Each device has a full copy of the KVS

Read

- Return the latest copy of the local KVS

Write

- Write to the local KVS and broadcast to the others
- May be rollbacked to ensure all the KVS copies are the same

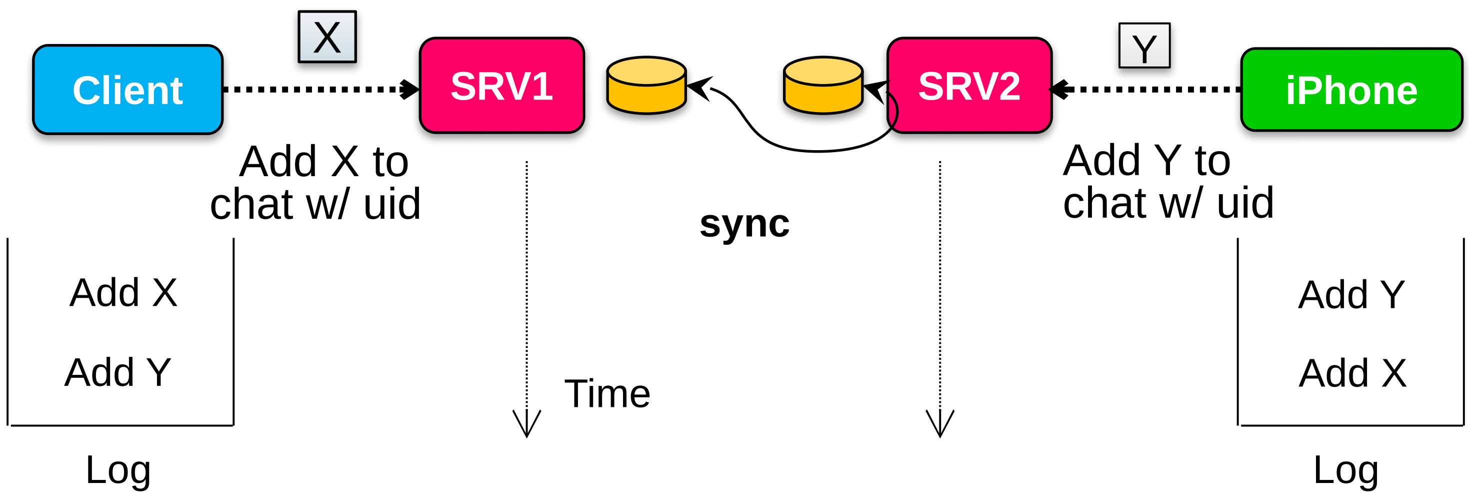
Review: basic ordered update log for single-copy value

Ordered list of updates at each node (updates are expressed as a function)

- Record the updates in a log, and sort it according to some order

Delay the updates, until we are sure that it can be ordered

Syncing: ensure both nodes have the same updates in log



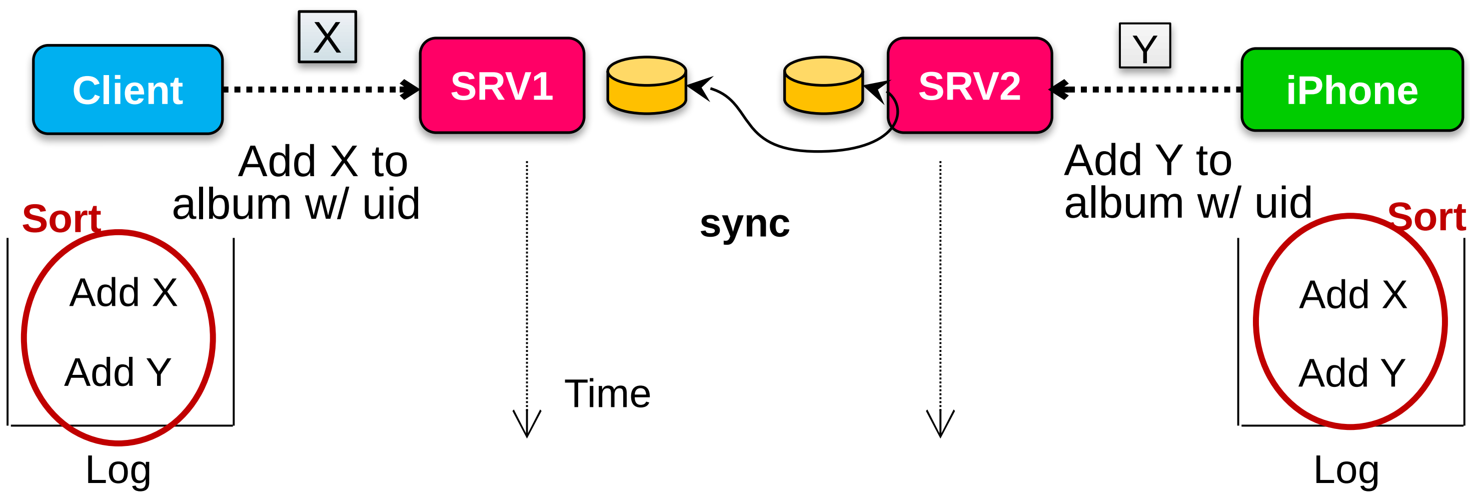
Review: basic ordered update log for single-copy value

Ordered list of updates at each node (updates are expressed as a function)

- Record the updates in a log, and sort it according to some order

Delay the updates, until we are sure that it can be ordered

Syncing: ensure both nodes have the same updates in log



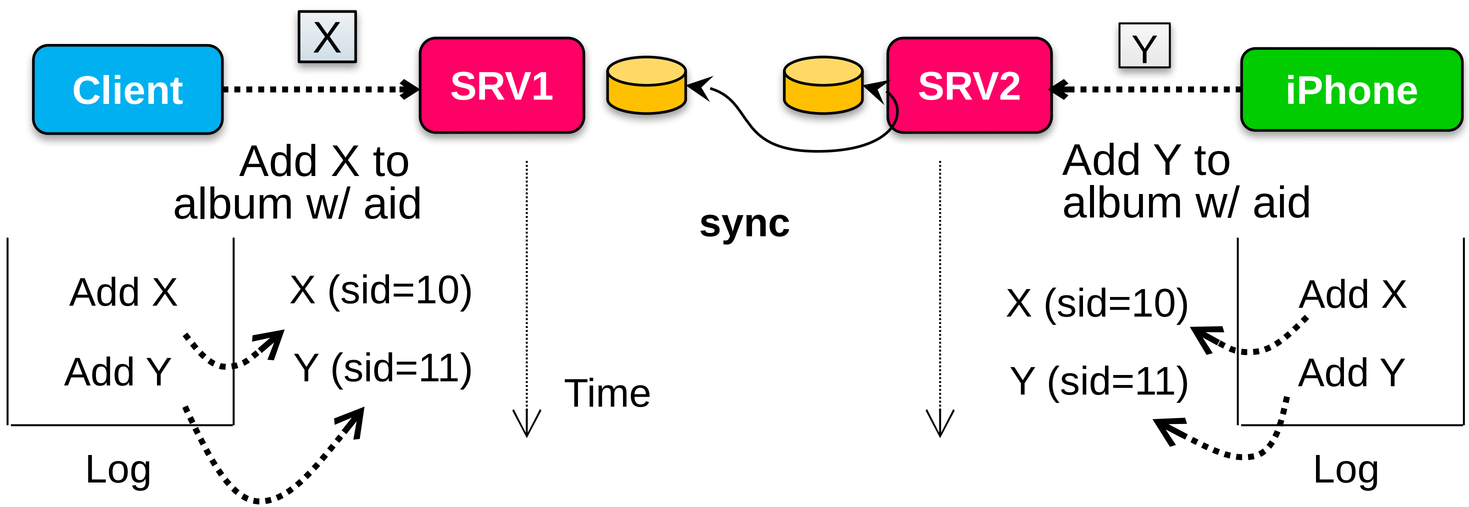
Review: basic ordered update log for single-copy value

Ordered list of updates at each node (updates are expressed as a function)

– Record the updates in a log, and sort it according to some order

Delay the updates, until we are sure that it can be ordered

Syncing: ensure both nodes have the same updates in log



Review: Lamport clock

We use lamport clock to sort the log entries

- Causality-preserving timestamp

Lamport logical clock

① Each server keeps a clock **T**

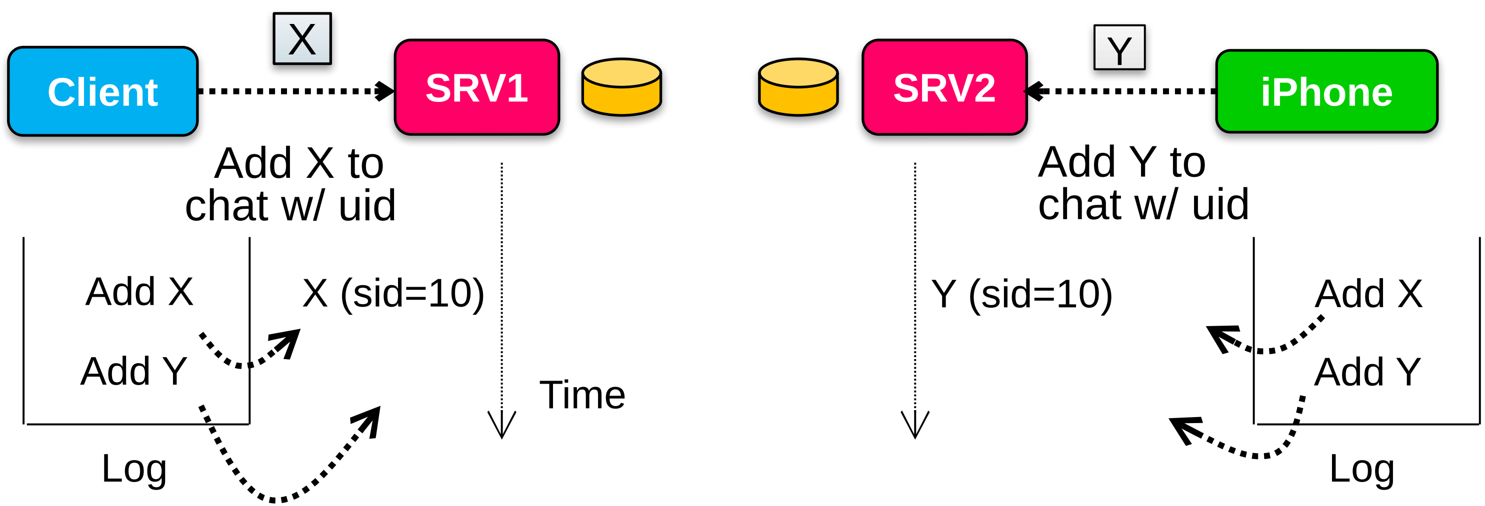
② Increments T as real time passes, e.g., one second per second

③ Modify **T = Max(T, T'+1)**
if sees **T'** from another server (e.g., from a message)

Review: local updates break the order of the log

Why delay the update? If we naively update the local storage, we cannot directly run the update functions after the sync

– Because the initial state of the servers are different!



9

Review: Rollback and Replay

Allow immediate update the storage for better read

Rollback all the updates before the sync

- Essentially clean the storage to an empty state

Re-run all update functions, starting from empty storage state

- After syncing, Srv1 and Srv2 have same set of updates (ordered logs)
- Srv1 and Srv2 arrive at same final state

Problems

- Slow sync process
- Large log size (If some device is out of the sync)

How to avoid storing all the logs?

Idea: distinguish tentative writes from stable ones

Each server's log consists of 2 portions:

- Stable writes, followed by
- Tentative writes

Stable writes are not rolled backup upon sync

- Tentative writes can be possibly been rolled back

Question

- How to determine which writes are stable? (hint: using the lamport clock!)

Answer

- An update (W) is stable iff no entries will have a lamport timestamp < W

De-centralized approach

Commit (write) scheme

- Each machine maintains the last seen lamport clock from the others
- Thus, we can calculate a global minimum
- If a log entry's write < global minimum, then it can be seemed as stable

Problem: If **any** node is **offline**, the **stable** portion of **all** logs stops growing

Centralized approach

Commit scheme

CSN: Commit-Seq-No

① One server designated “**primary**”

- Assign a total commit order: CSN to each write
- Complete timestamp: <**CSN**, local-TS, SrvID>
- Any write with a known CSN is stable

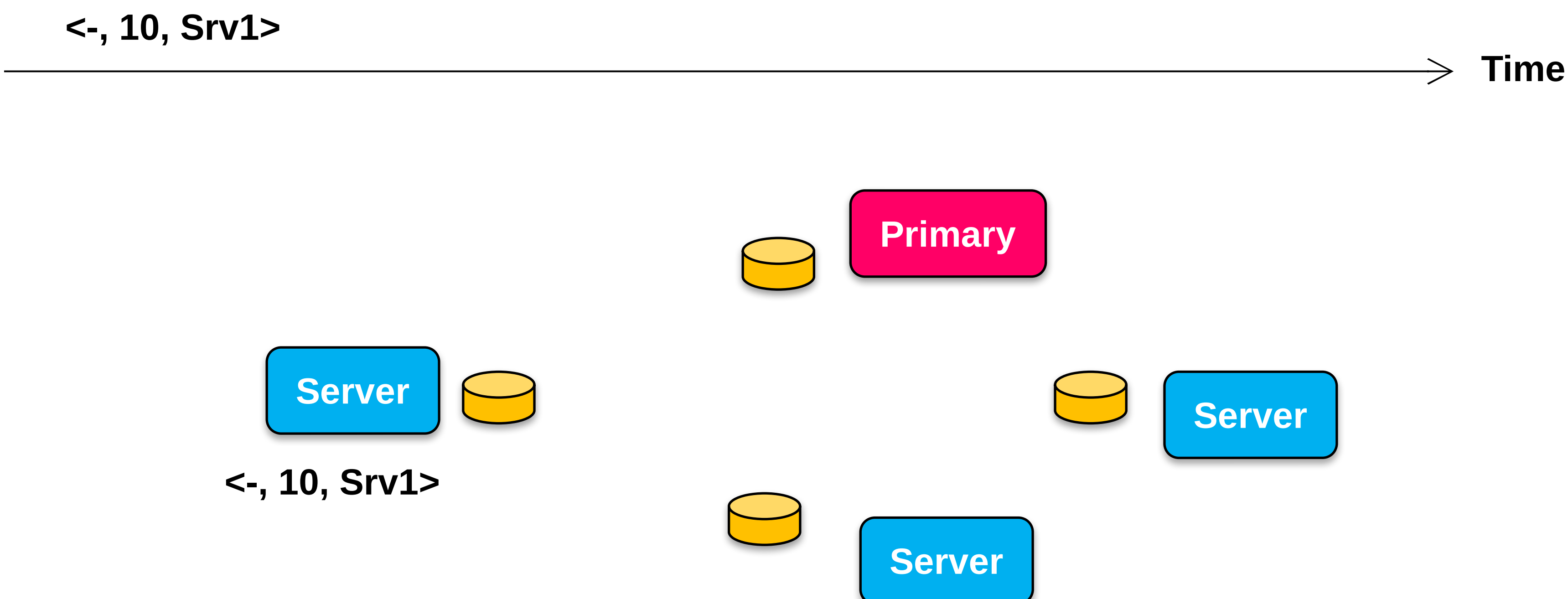
② All stable writes are ordered before tentative writes

③ CSNs are exchanged between servers

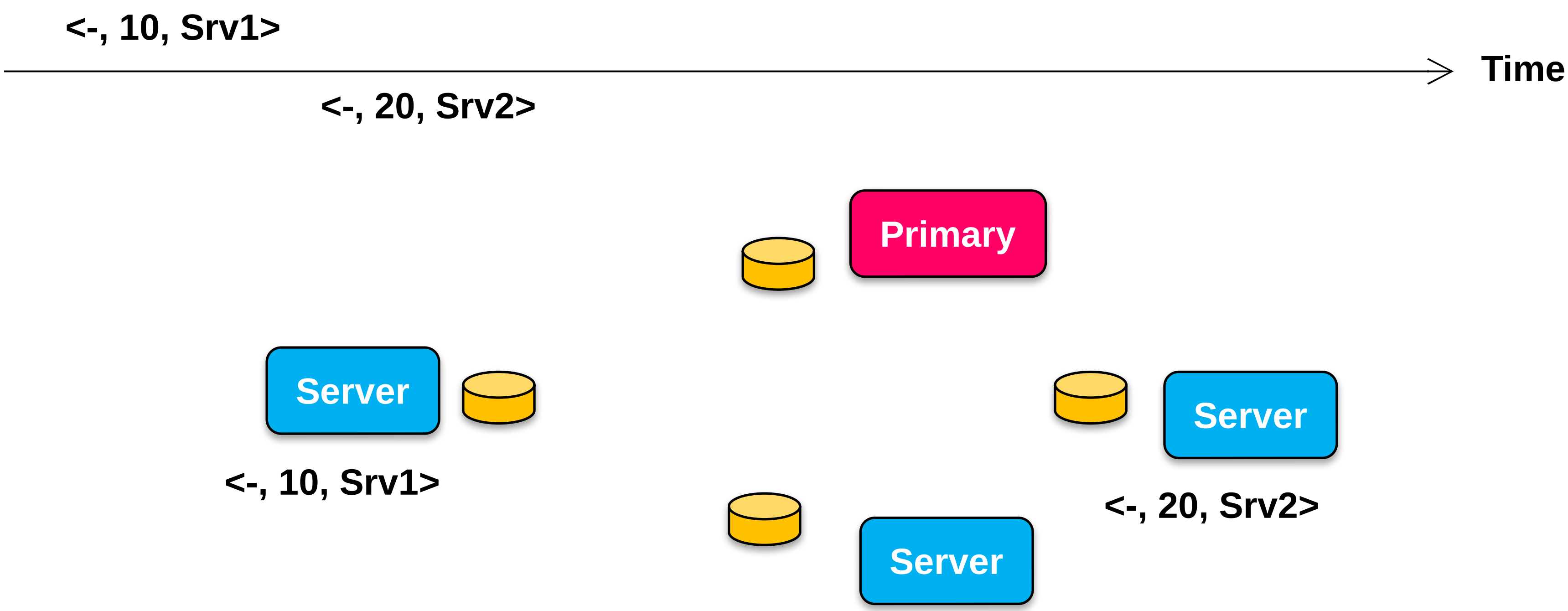
- CSNs define a total order for committed update

Advantage: as long as the primary is up, writes can be committed and stabilized

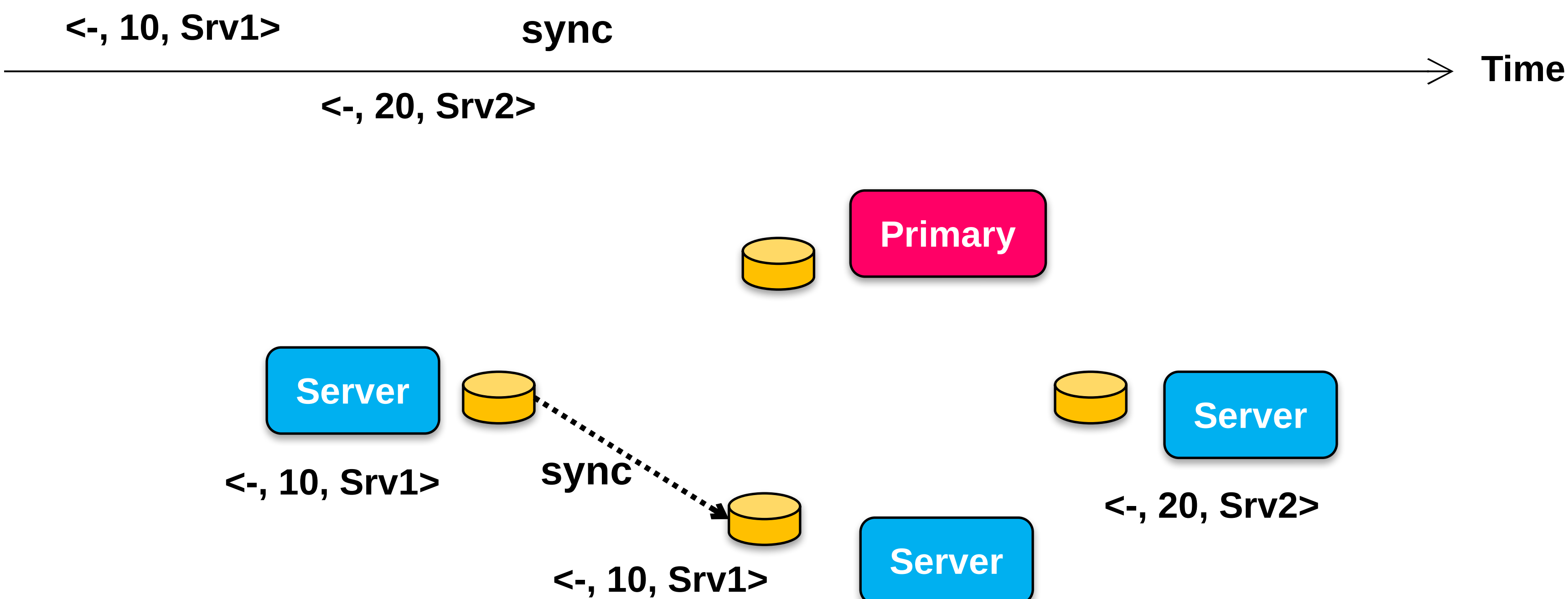
Example



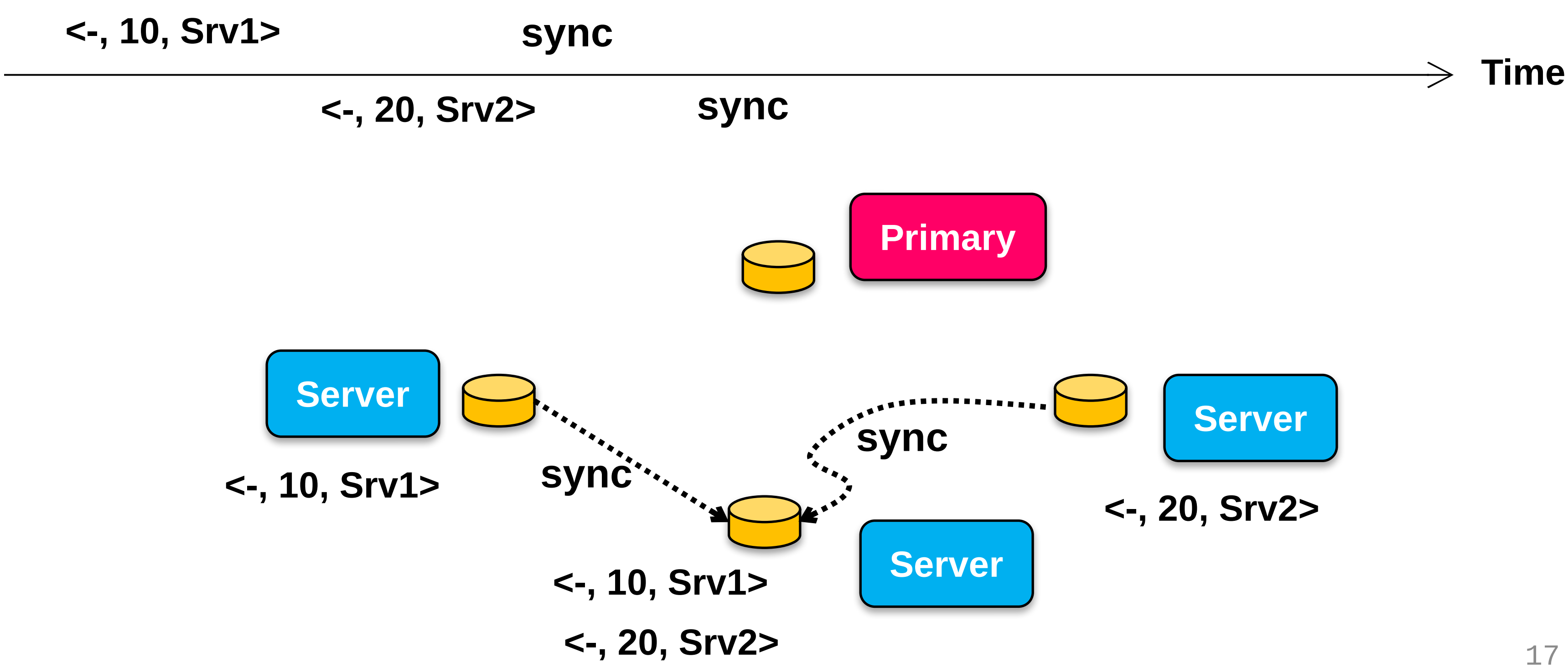
Example


<-, 10, Srv1>
Time
<-, 20, Srv2>
Primary
Server
Server
<-, 10, Srv1>
<-, 20, Srv2>
Server

Example

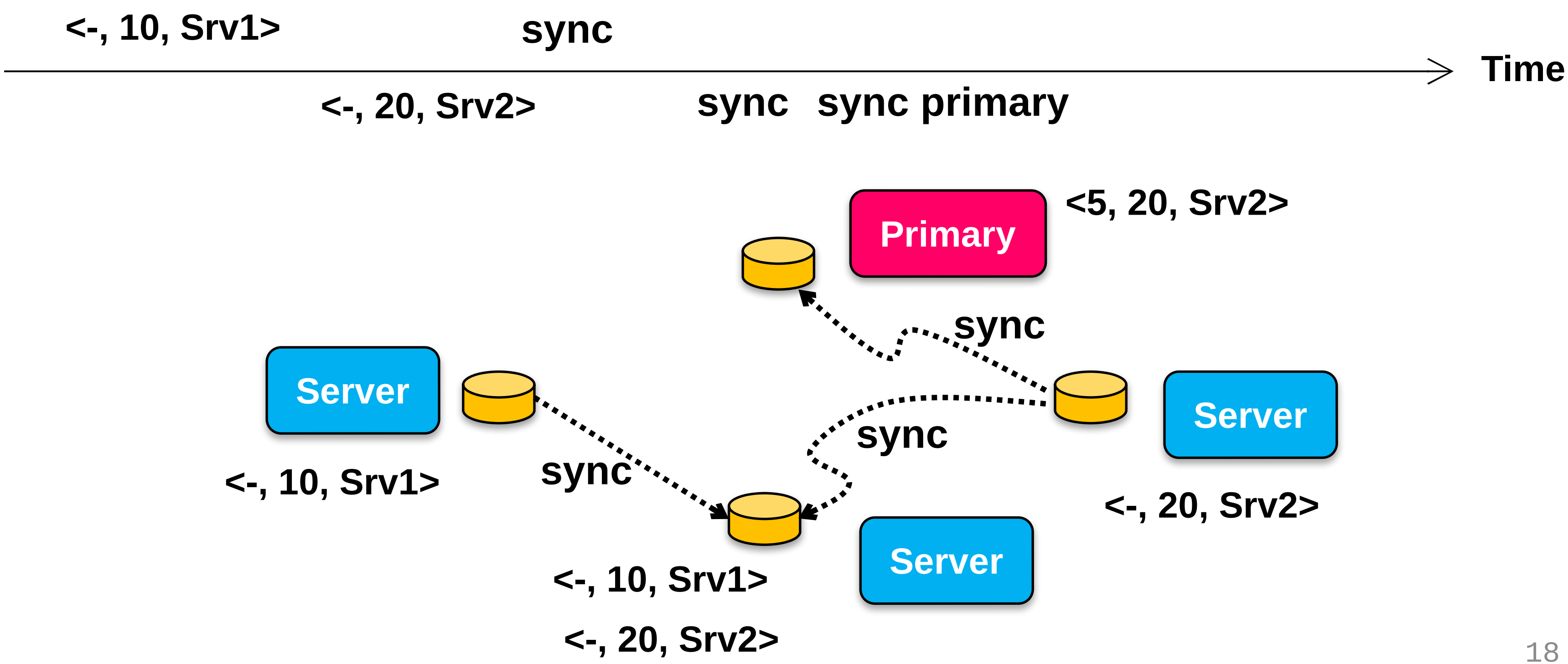

<-, 10, Srv1>
sync
Time
<-, 20, Srv2>
Primary
Server
<-, 10, Srv1>
sync
Server
<-, 20, Srv2>
Server
<-, 10, Srv1>

Example



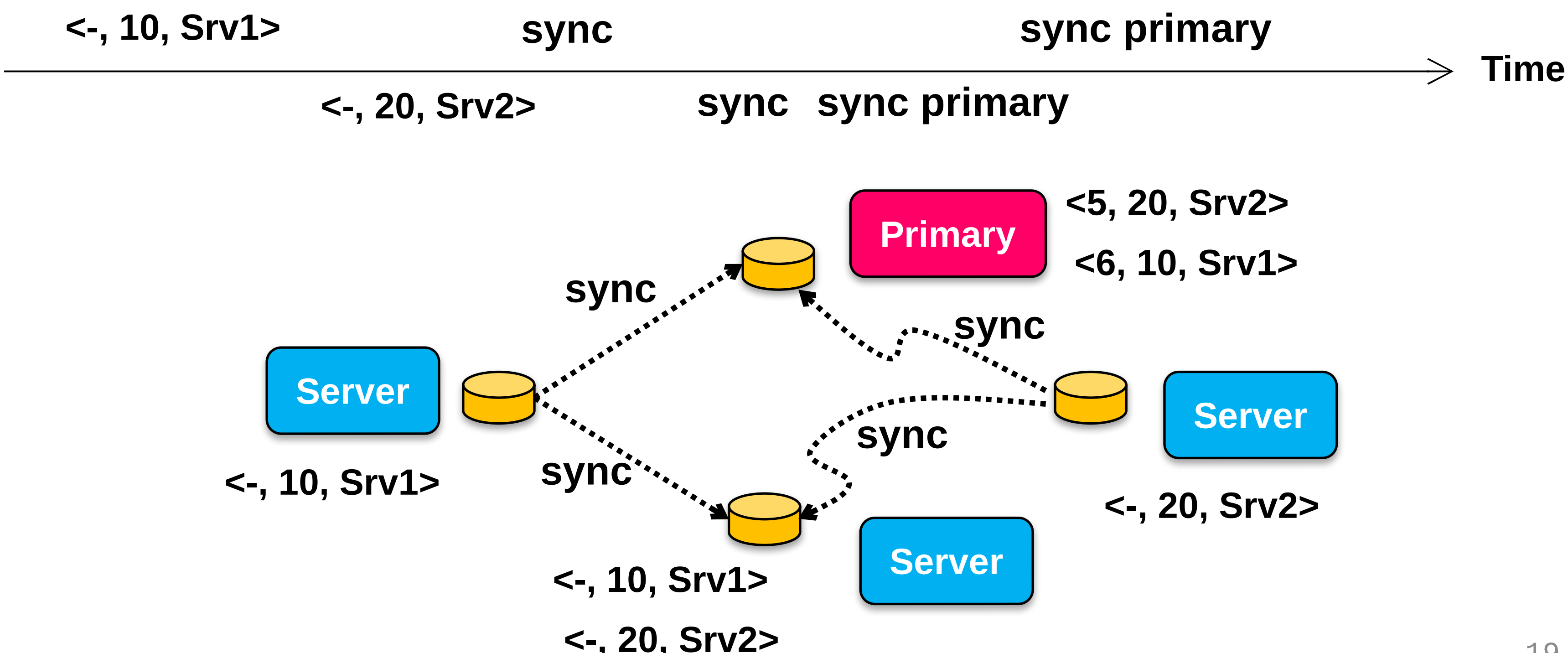
<-, 10, Srv1>
sync
Time
<-, 20, Srv2>
sync
Primary
Server
Server
sync
sync
<-, 10, Srv1>
<-, 20, Srv2>
Server
<-, 10, Srv1>
<-, 20, Srv2>

Example


<-, 10, Srv1>
sync
Time
<-, 20, Srv2>
sync
sync primary
Primary
<5, 20, Srv2>
sync
Server
sync
Server
<-, 10, Srv1>
sync
<-, 20, Srv2>
Server
<-, 10, Srv1>
<-, 20, Srv2>

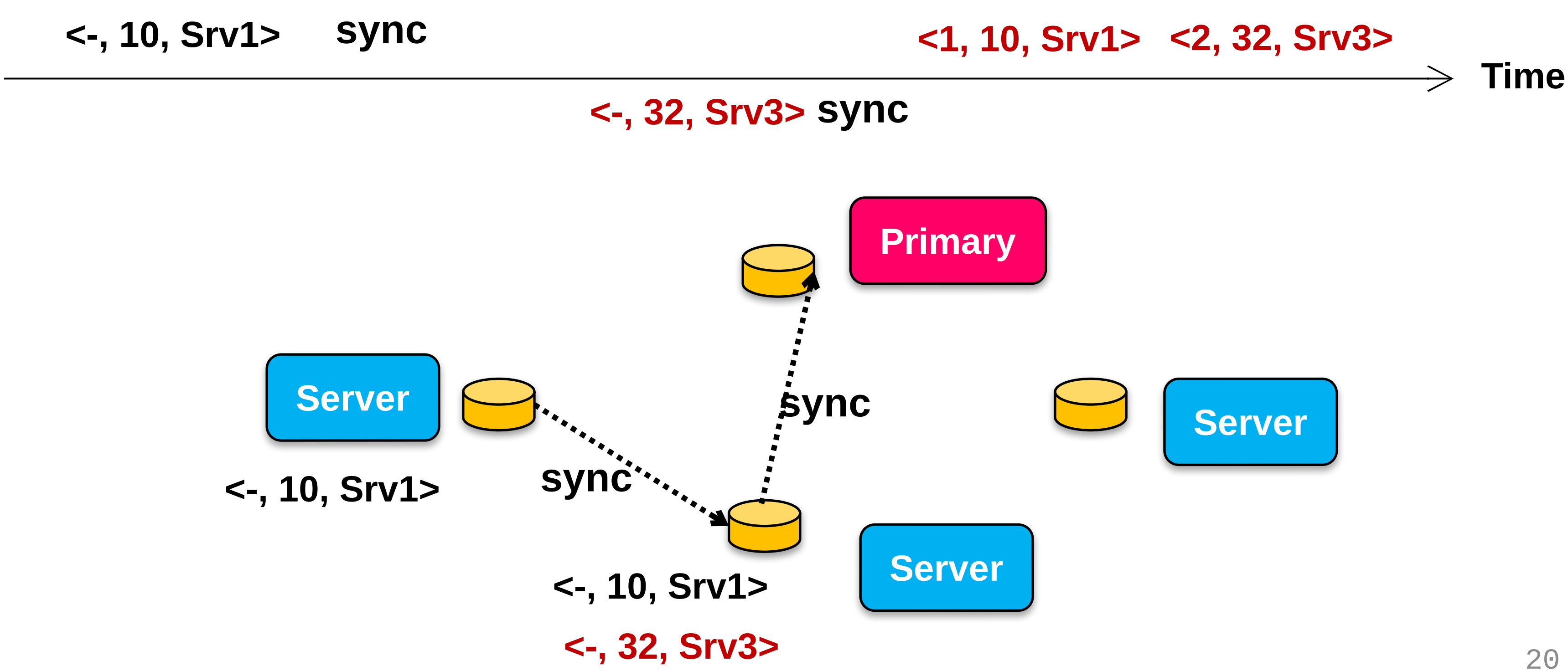
Example

Can stable <5,20,Srv2> as long as <4, xx, xx> has been stabled



Question: how to preserve causality?

Assigns CSNs for dependent tentative writes together



What about reads?

Possible read/write rule implementations

- Read: return the latest local copies of the data
- ~~Write: write locally (and directly returns), propagate the writes to all the servers in background (e.g., upon sync)~~
- Read may return
  - Tentative data
  - Outdated data
- **Trade** read/write consistency for high performance

Does eventual consistency anomalies matter?

Existential Consistency:
Measuring and Understanding Consistency at Facebook

Haonan Lu[*†], Kaushik Veeraraghavan[†], Philippe Ajoux[†], Jim Hunt[†],
Yee Jiun Song[†], Wendy Tobagus[†], Sanjeev Kumar[†], Wyatt Lloyd[*†]
[*]University of Southern California, [†]Facebook, Inc.

Abstract

Replicated storage for large Web services faces a trade-off between stronger forms of consistency and higher performance properties. Stronger consistency prevents anomalies, i.e., unexpected behavior visible to users, and reduces programming complexity. There is much recent work on improving the performance properties of systems with stronger consistency, yet the flip-side of this trade-off remains elu-

1. Introduction

Replicated storage is an important component of large Web services and the consistency model it provides determines the guarantees for operations upon it. The guarantees range from eventual consistency, which ensures replicas eventually agree on the value of data items after receiving the same set of updates to strict serializability [12] that ensures transactional isolation and external consistency [25]. Stronger

SOSP 2015

It depends on the application scenarios

- Frequencies of the anomalies
- Importance of the anomalies

Facebook has conducted a research to measure the frequencies of anomalies under eventual consistency. Some highlights are:

- Per-Object Results: 1 anomaly per million reads (user should see their writes)
- A social networking website can tolerate many anomalies

However, many scenarios (e.g., Bank) require stronger models (even linearizability is insufficient)

Consistency under single-machine faults

Recall: what is a strong consistency model

It's easy for users to reason about correctness assuming

- Everything has only one-copy
- The overall behavior is equivalent to some serial behavior

What about failure? Can it cause consistency issues?

Example: failure leaves operations in a partial state

e.g., writing to a file

- Question: what happen to `fputs` under crash?

What happens to fputs?

- **Not started** (in OS's in-memory cache)
- **Doing** (writing a large data, not finished yet)
- **Done** (fine)

```c
#include <stdio.h>

int main() {
    FILE *fp;

    fp = fopen("/tmp.txt", "w+");
    fputs("Hello world\n", fp);
    fclose(fp);
}
```

Application should deal with the fact that a single write could fail. Otherwise, may affect the correctness of the applications!

Example: bank transfer

Suppose bank accounts are stored in a single file

- The `transfer` is executed on a single machine with a single thread
- So it follows linearizability by default

```
transfer(bank, a, b, amt):
    bank[a] = bank[a] – amt
    bank[b] = bank[b] + amt
```

Application invariant that must preserve:
bank(a) + bank(b) never changes

Example: bank transfer

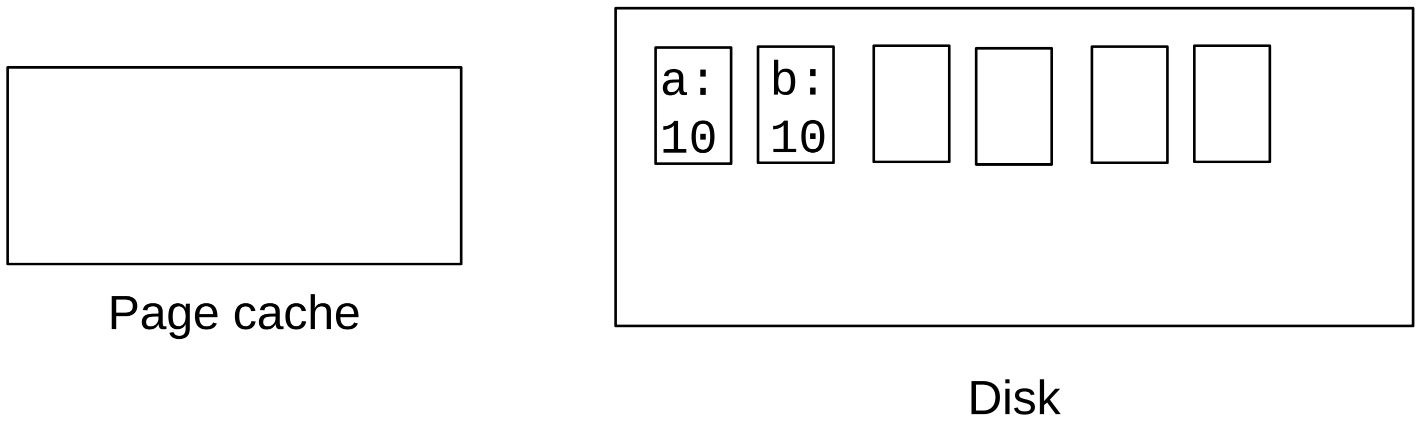
Implementation (Simplified)

```
transfer(bank, a, b, amt):
    records = mmap(bank, ...)
    records[a] = records[a] – amt
    records[b] = records[b] + amt
    fsync(bank, ...)
```

What if an error happens during the execution of `fsync`?

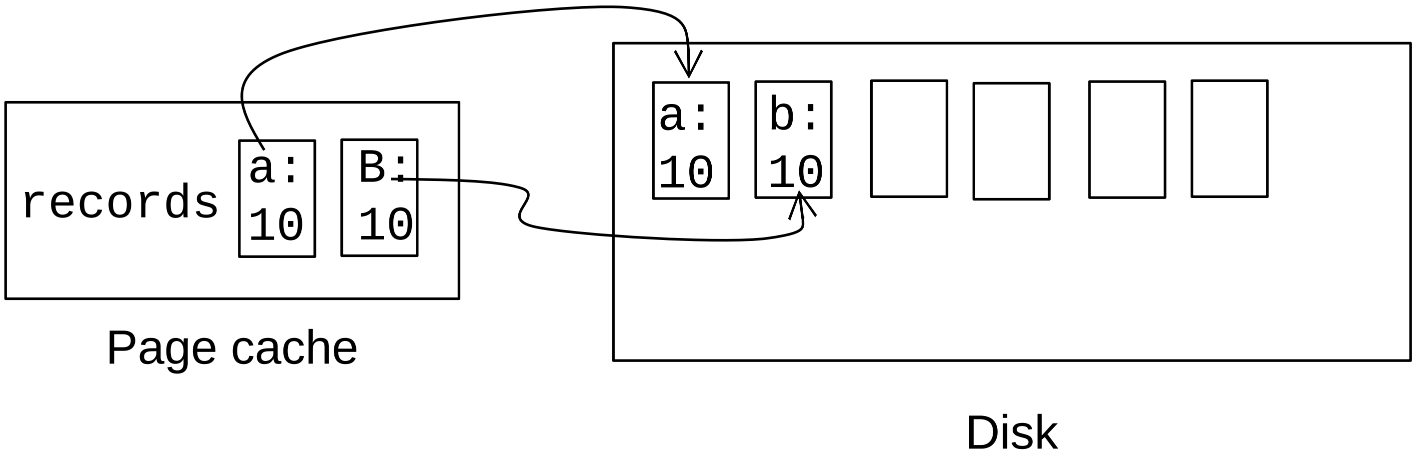
Deconstructing fsync (SYNC)

```
transfer(bank, a, b, amt): // amt=10
    records = mmap(bank, ...)
    records[a] = records[a] – amt
    records[b] = records[b] + amt
    fsync(bank, ...)
```



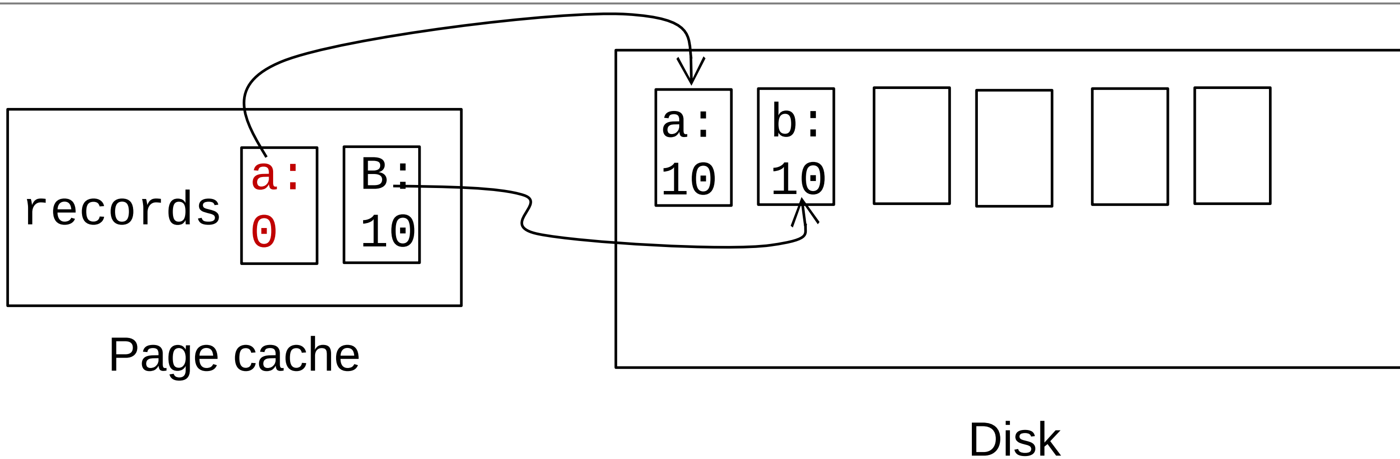
Deconstructing fsync (SYNC)

```
transfer(bank, a, b, amt): // amt=10
    records = mmap(bank, ...)
    records[a] = records[a] – amt
    records[b] = records[b] + amt
    fsync(bank, ...)
```



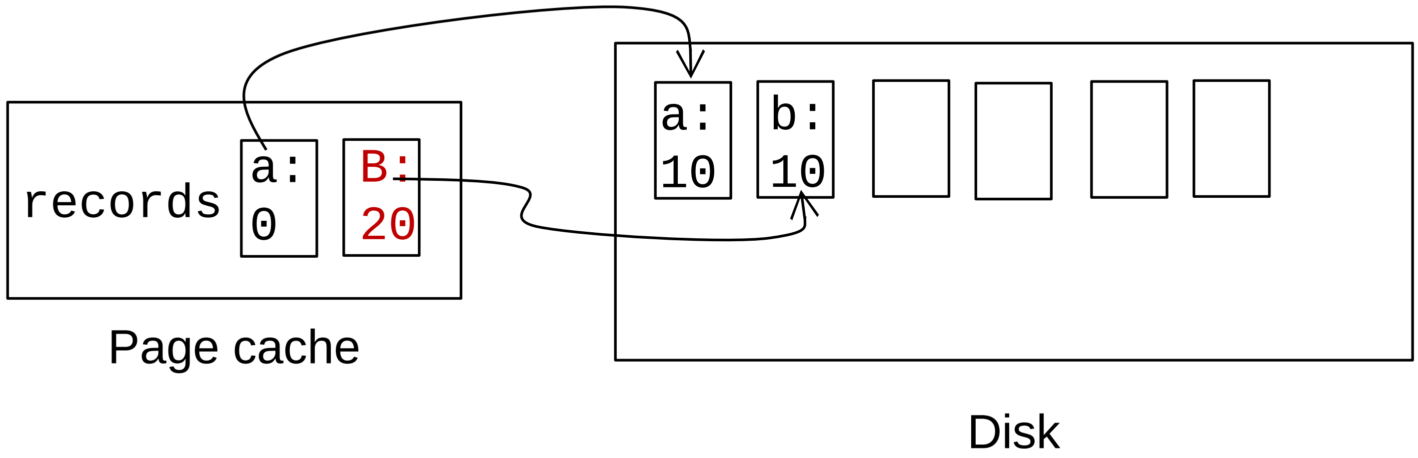
Deconstructing fsync (SYNC)

```
transfer(bank, a, b, amt): // amt=10
    records = mmap(bank, ...)
    records[a] = records[a] – amt
    records[b] = records[b] + amt
    fsync(bank, ...)
```



Deconstructing fsync (SYNC)

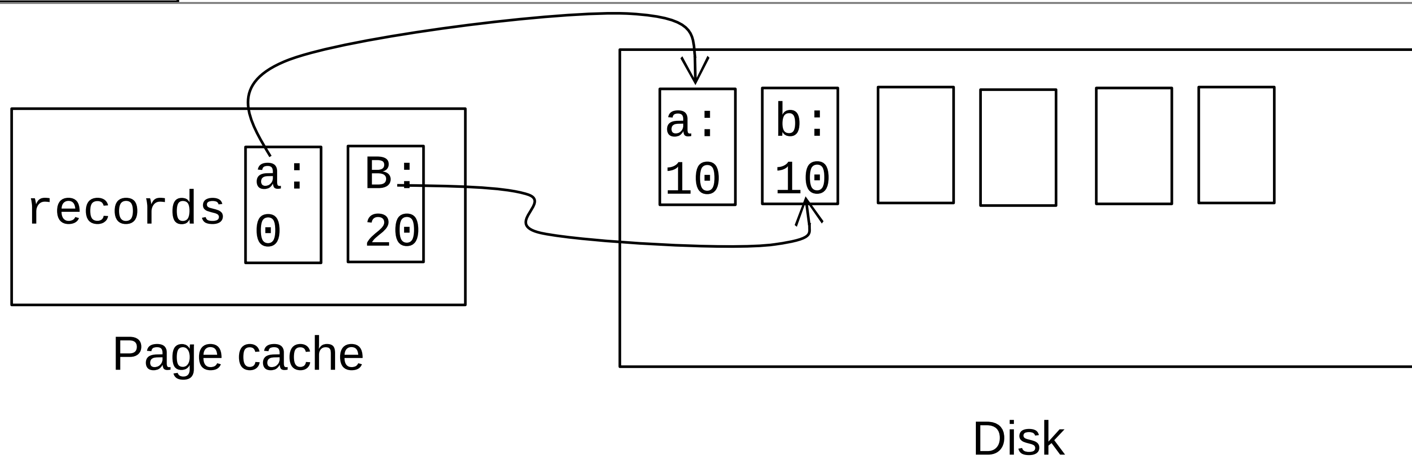
```
transfer(bank, a, b, amt): // amt=10
    records = mmap(bank, ...)
    records[a] = records[a] – amt
    records[b] = records[b] + amt
    fsync(bank, ...)
```



Deconstructing fsync (SYNC)

```
transfer(bank, a, b, amt): // amt=10
    records = mmap(bank, ...)
    records[a] = records[a] – amt
    records[b] = records[b] + amt
    fsync(bank, ...)
```

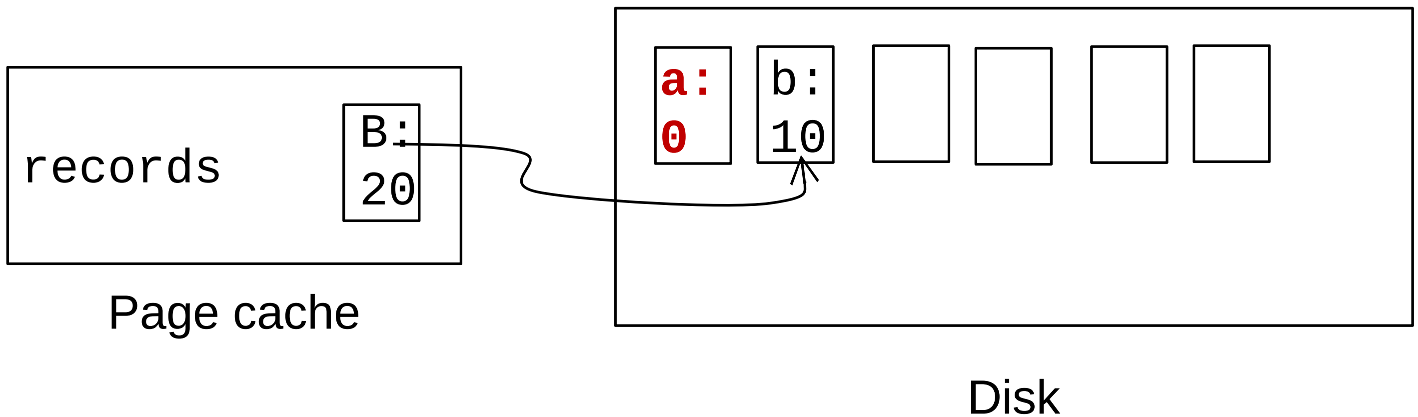
① Write(A)
② Write(B)



Deconstructing fsync (SYNC)

```
transfer(bank, a, b, amt): // amt=10
    records = mmap(bank, ...)
    records[a] = records[a] – amt
    records[b] = records[b] + amt
    fsync(bank, ...)
```

① Write(A)
② Write(B)

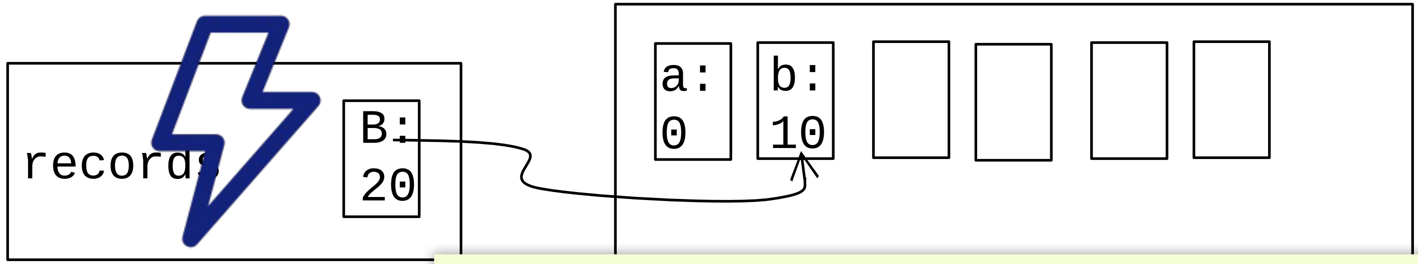


Pages are flushed to disk one-by-one

Deconstructing fsync (SYNC)

```
transfer(bank, a, b, amt): // amt=10
    records = mmap(bank, ...)
    records[a] = records[a] – amt
    records[b] = records[b] + amt
    fsync(bank, ...)
```

① Write(A)
② Write(B)



Page cache

Consistency is broken after the machine failure!

Application invariant that must preserve:
bank(a) + bank(b) never changes

What is a strong consistency model (continued)

It's easy for users to reason about correctness assuming

- Everything has only one-copy
- The overall behavior is equivalent to some serial behavior
- The operations that need to be executed in an atomic unit (usually called operations belonging to a transaction) are executed on a machine **that happens completely or not at all (all-or-nothing atomicity)**

All-or-nothing atomicity makes it much easier to reason about failures

- Need to think about the consequences of the action happening or not happening, but not about the action ***partially*** happening

Achieving atomicity: shadow copy

Ensure a set of operations written to a file is all-or-nothing

- E.g., writes records a & b to the bank file

High-level idea:

- Do not modify the old copy (there is always a consistent copy)
- Replace the origin file with the updates only if all the writes to are successful

Original

```
transfer(bank, a, b, amt):
    records = mmap(bank, ...)
    records[a] = records[a] – amt
    records[b] = records[b] + amt
    fsync(bank, ...)
```

Shadow copy

```
transfer(bank, a, b, amt):
    fcopy(bank, bank_temp)
    records =
mmap(bank_temp, ...)
    records[a] = records[a] – amt
    records[b] = records[b] + amt
    fsync(bank_temp, ...)
    rename(bank_temp,bank)
```

Shadow copy: analysis

Shadow copy

```
transfer(bank, a, b, amt):
    fcopy(bank, bank_temp)
    records =
mmap(bank_temp, ...)
    records[a] = records[a] – amt
    records[b] = records[b] + amt
    fsync(bank_temp, ...)
    rename(bank_temp,bank)
```

Question: what happen if a crash happens during fcopy or fsync?

– The origin bank file is always consistent

– We can just remove the `bank_temp`, so the transfer not happens

Shadow copy

Shadow copy

```
transfer(bank, a, b, amt):
    fcopy(bank, bank_temp)
    records =
mmap(bank_temp, ...)
    records[a] = records[a] – amt
    records[b] = records[b] + amt
    fsync(bank_temp, ...)
    rename(bank_temp,bank)
```

Question: what happen if a crash happens during rename?

– For the bank transfer, it is consistent

What about the filesystem state?

– Depends on the internal implementation of the filesystem!

rename(temp_bank, bank)

Directory data blocks:

- filename "bank" → inode 12
- filename " temp_bank " → inode 13

inode 12:

- data blocks: 3, 4, 5
- refcount: 1

inode 13:

- data blocks: 6, 7, 8
- refcount: 1

rename(temp_bank, bank)

Time

Directory data blocks:

- filename "bank" → inode **13**
- filename " temp_bank " → inode 13

inode 12:

- data blocks: 3, 4, 5
- refcount: 1

inode 13:

- data blocks: 6, 7, 8
- refcount: 1

rename(temp_bank, bank)

Time

Directory data blocks:

- filename "bank" → inode 13
- filename " temp_bank " → inode 13

inode 12:

- data blocks: 3, 4, 5
- refcount: 1

inode 13:

- data blocks: 6, 7, 8
- refcount: **2**

rename(temp_bank, bank)

Directory data blocks:

- filename "bank" → inode **13**
- filename " temp_bank " → inode 13

Time

inode 12:

- data blocks: 3, 4, 5
- refcount: **0**

inode 13:

- data blocks: 6, 7, 8
- refcount: 2

rename(temp_bank, bank)

Directory data blocks:

- filename "bank" → inode **13**
- ~~**filename " temp_bank " → inode 13**~~

inode 12:

- data blocks: 3, 4, 5
- refcount: 0

inode 13:

- data blocks: 6, 7, 8
- refcount: 2

Time

rename(temp_bank, bank)

Directory data blocks:

- filename "bank" → inode **13**
- ~~filename " temp_bank " → inode 13~~

inode 12:

- data blocks: 3, 4, 5
- refcount: 0

inode 13:

- data blocks: 6, 7, 8
- refcount: **1**

Time

If crash, what step will cause problem?

Problem

Directory data blocks:

- filename "bank" → **inode 13**
- filename ”temp_bank" → **inode 13**

inode 12:

- data blocks: 3, 4, 5
- refcount: 1

inode 13:

- data blocks: 6, 7, 8
- **refcount: 1**

Time

Two names point to fnew’s inode, but refcount is 1
which reference is the correct one?

Naïve solution

Ask application to remove the unnecessary inodes

- i.e., the application knows that "bank_temp" should be removed

Typically, not a good idea

- Problem #1. The filesystem consistency depends on the application. What if the application is buggy or malicious?
- Problem #2. What about more complex applications?

Ideal semantic: the application’s recovery method should be decoupled from the filesystem recovery method

Solution: file system ensures the rename is atomic

To decouple the atomicity of shadow copy from the application

Key idea: journaling

Journaling overview

Logging

– A concept from database (and will talk about more details in this lecture)

Record before update

1. Record changes in journal
2. Commit journal
3. Update

Crash before commit:
- No data is changed
- Discard journal

Crash after commit:
- Journal is complete
- Redo changes in journal

Rename via journaling

Directory data blocks:

- filename "bank" → inode 12 **-> 13**
- ~~filename " temp_bank " → inode 13~~

Inside of 12

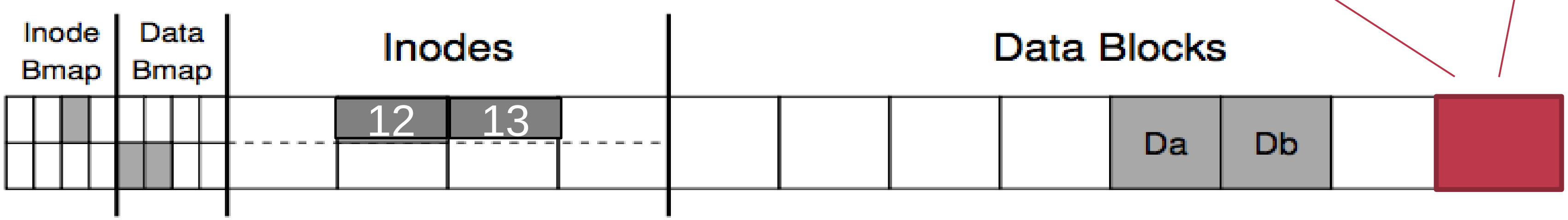
- data blocks: 3, 4, 5
- refcount: 1 -> **0**

Inside of 13

- data blocks: 6, 7, 8
- refcount: 1

Directory data blocks:
- “bank” -> inode 13
- Delete ”temp_bank”

Inode 12: refcount = 0
Inode 13: refcount = 1
COMMITTED

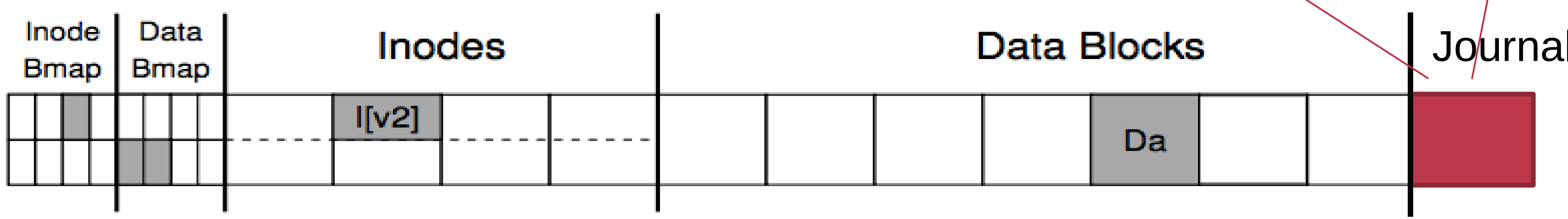


Append a File via Journaling

Inside of Inode:

- owner : yubin
- permissions : read-write
- size : 1 **=> 2**
- pointer : 4
- pointer : null **=> 5**
- pointer : null

Inode:
size: 1=>2
2^{nd} pointer: null => 5
Db:
Old-data… => Appended-data…
COMMITTED



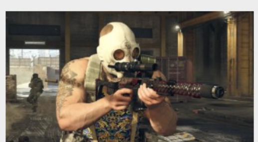
Journaling Drawbacks

Everything is written to the disk twice

- Once in the journal
- The other at the home location

The problem: files are large

The largest game install sizes (in descending order)



231GB - Call of Duty: Modern Warfare

Don't act surprised, we all knew this was coming. With both Modern Warfare and Warzone combined, today's CoD is one of the most chonky games in existence. If not the chonkiest.

Mitigating Journaling Drawbacks

Observation:

- Not everything in file system has **equal importance**
- Usually, the **metadata** is more important
    - E.g., inode data updated by rename

Mitigation:

- Only protect metadata via Journaling
    - Data is written only once
- What if data is also important?
    - Ext4 options: data=journal/ordered/writeback
    - Application can also handle the write itself, e.g., wait for fsync to flush the data synchronously before proceed to the next

What if crash during commit journal?

Assume that writing to one sector on disk is all-or-nothing

- Disk saves enough energy to complete one sector write
    - Small capacitor suffices to power disk for a few microsecond
    - Assumption: time spent writing a sector is small
- If write did not start, no need to complete it
    - Still all-or-nothing

So write to the journal is always all-or-nothing

Back to shadow copy

Write to a copy of data, atomically switch to new copy

Switching can be done with one all-or-nothing operation

- Rename with journaling

Only requires a simple recovery procedure of the file system

- i.e., remove the temporal file after the recovery
- The filesystem itself is kept consistent via journaling
  - i.e., the filesystem itself has a recovery process

Drawbacks of shadow copy

Question: what would happen if multiple clients share the same file?

Example:

Client 0

```
transfer(bank, a, b, amt):
    fcopy(bank, bank_temp)
    records = mmap(bank_temp, ...)
    records[a] = records[a] – amt
    records[b] = records[b] + amt
    fsync(bank_temp, ...)
    rename(bank_temp,bank)
```

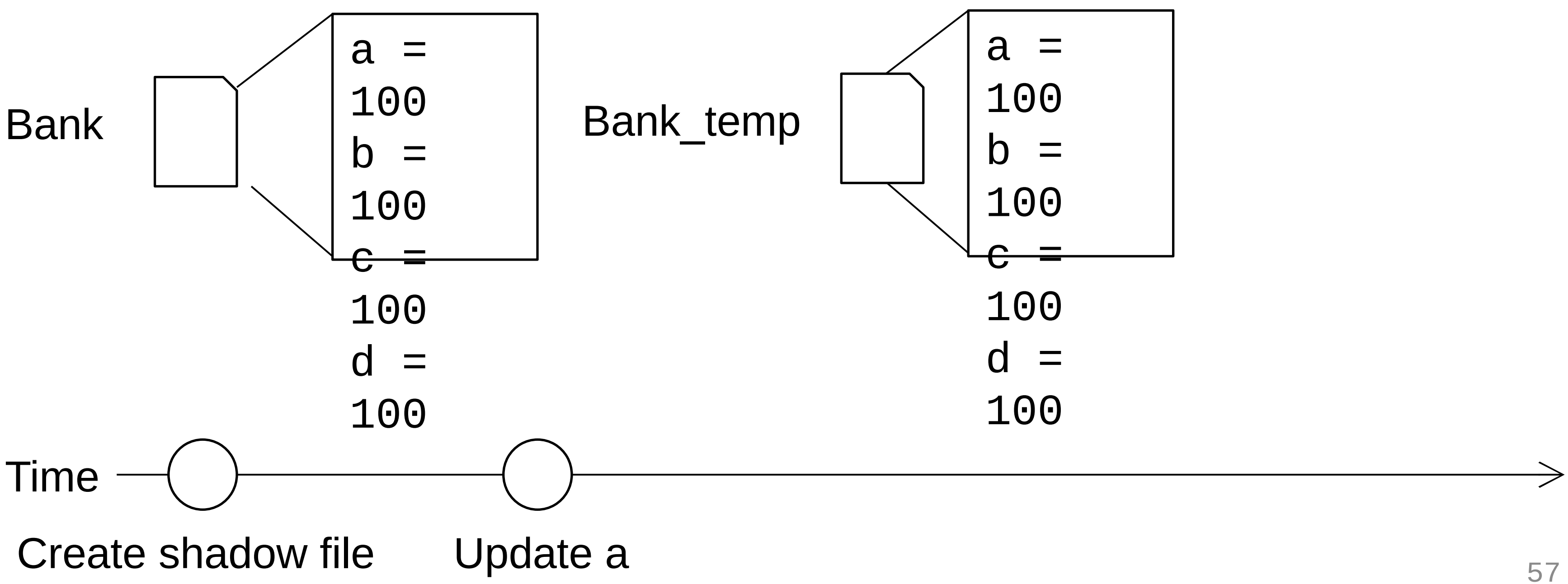
Client 1

```
transfer(bank, c, d, amt):
    fcopy(bank, bank_temp)
    records = mmap(bank_temp, ...)
    records[a] = records[c] – amt
    records[b] = records[d] + amt
    fsync(bank_temp, ...)
    rename(bank_temp,bank)
```

Drawbacks of shadow copy

Question: what would happen if multiple clients share the same file?

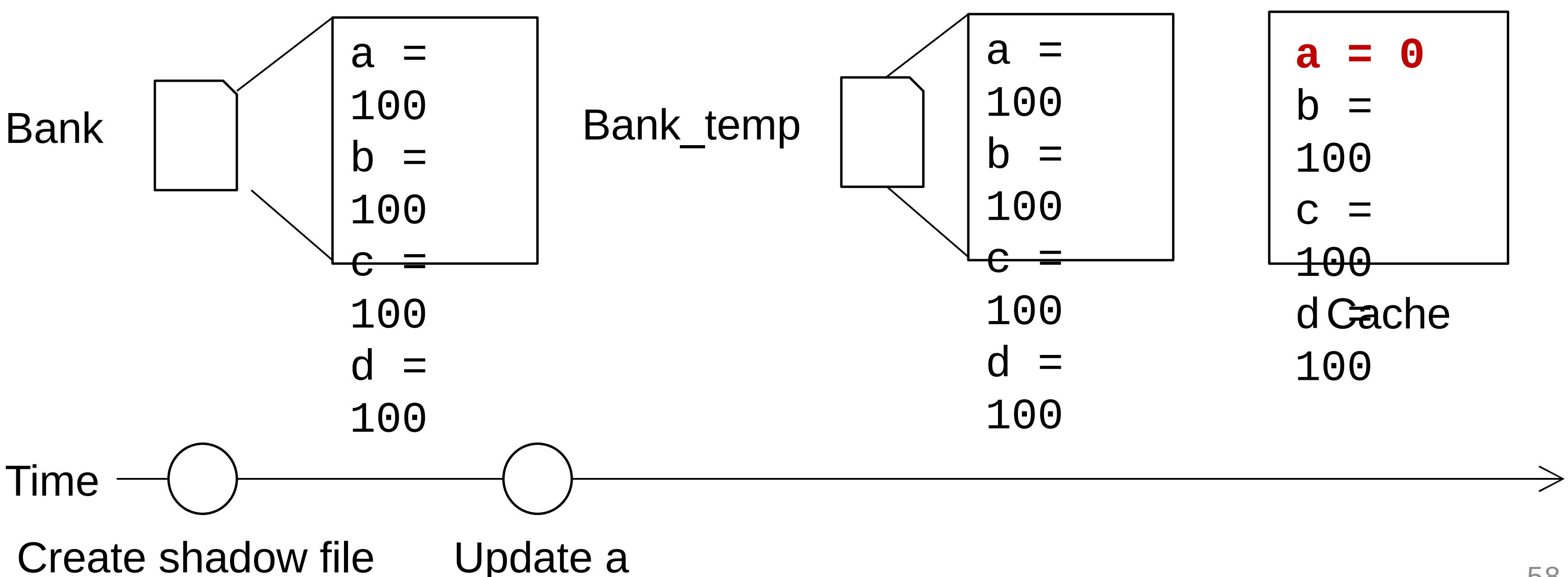
① Client 0 transfers 100 from a to b



Drawbacks of shadow copy

Question: what would happen if multiple clients share the same file?

① Client 0 transfers 100 from a to b

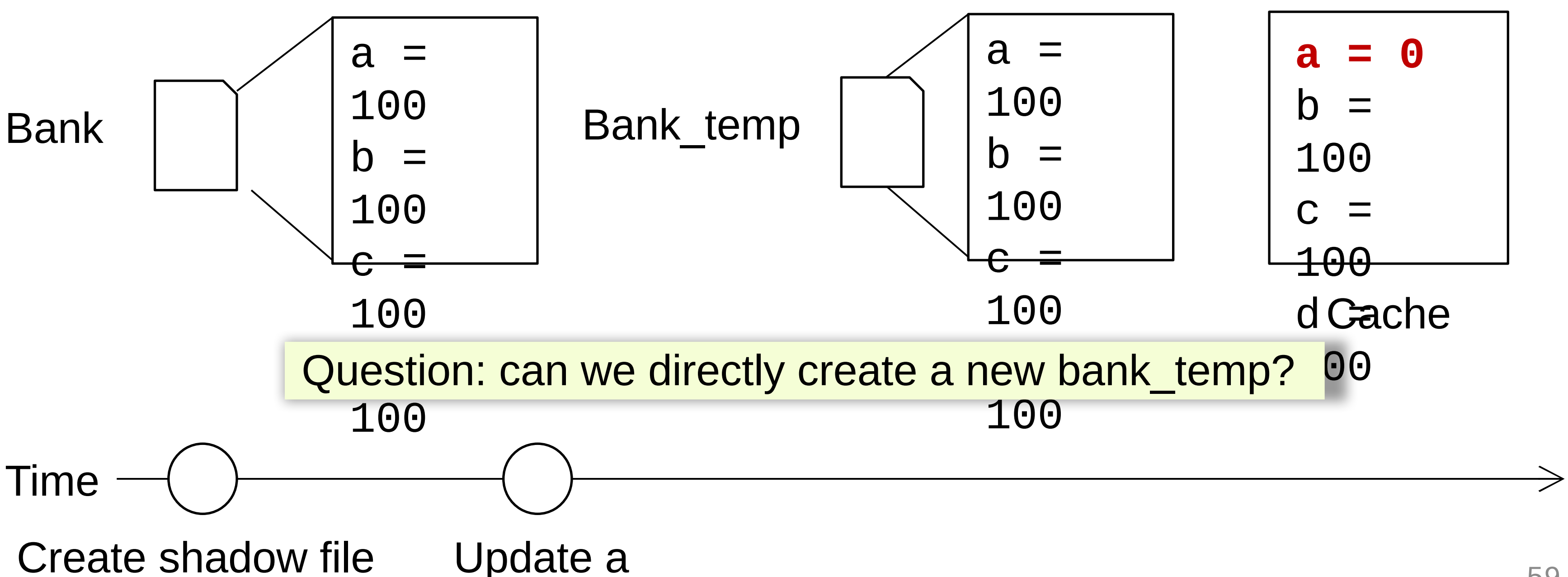


Drawbacks of shadow copy

Question: what would happen if multiple clients share the same file?

① Client 0 transfers 100 from a to b

② Client 1 transfers 100 from c to d

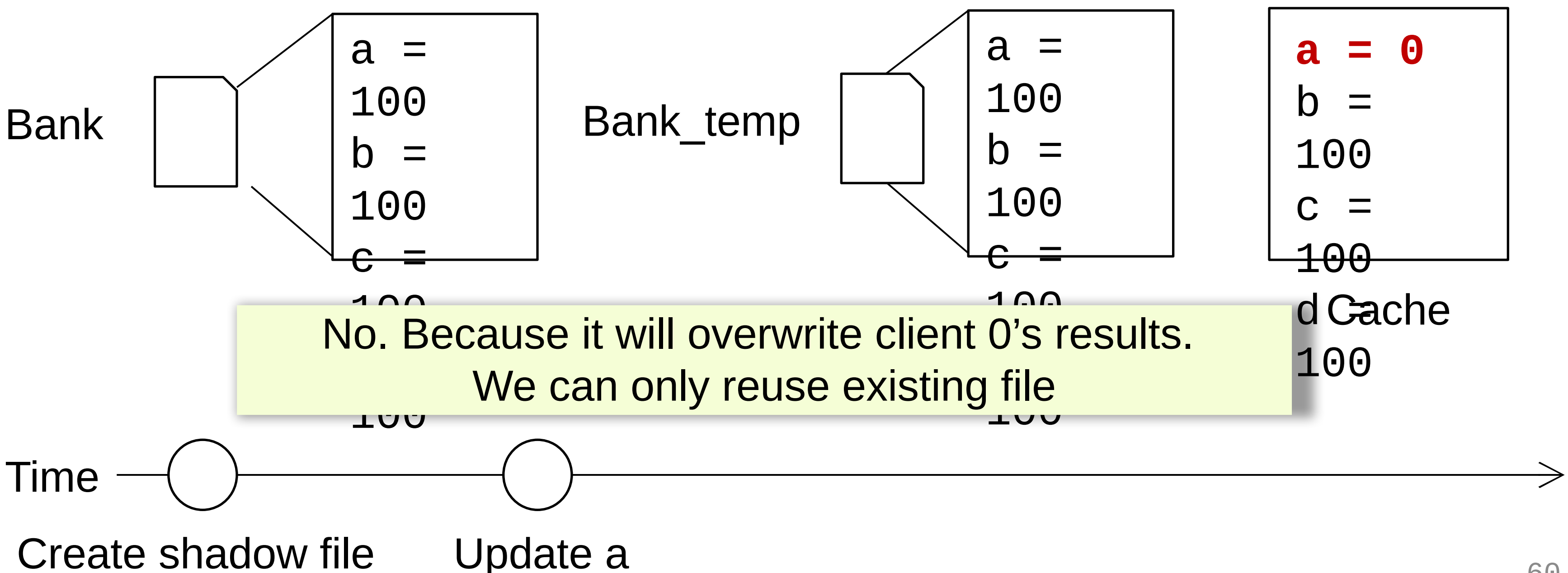


Drawbacks of shadow copy

Question: what would happen if multiple clients share the same file?

① Client 0 transfers 100 from a to b

② Client 1 transfers 100 from c to d



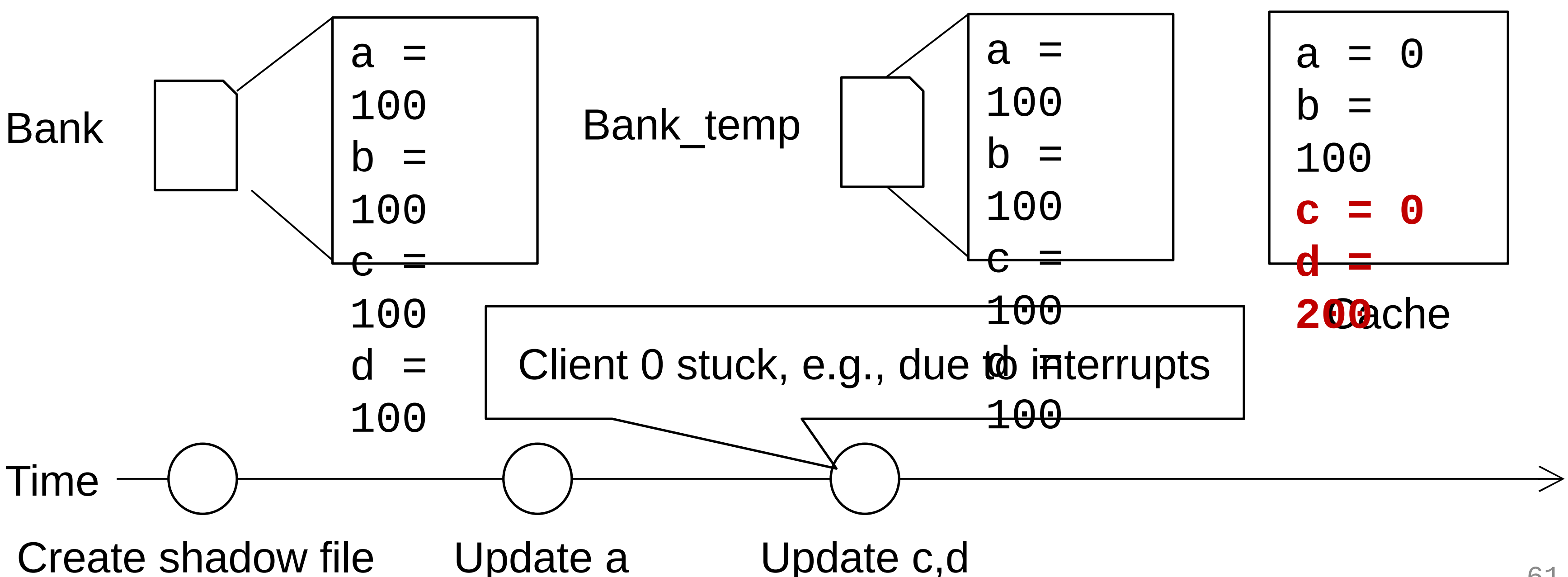
No. Because it will overwrite client 0's results.
We can only reuse existing file

Drawbacks of shadow copy

Question: what would happen if multiple clients share the same file?

① Client 0 transfers 100 from a to b

② Client 1 transfers 100 from c to d

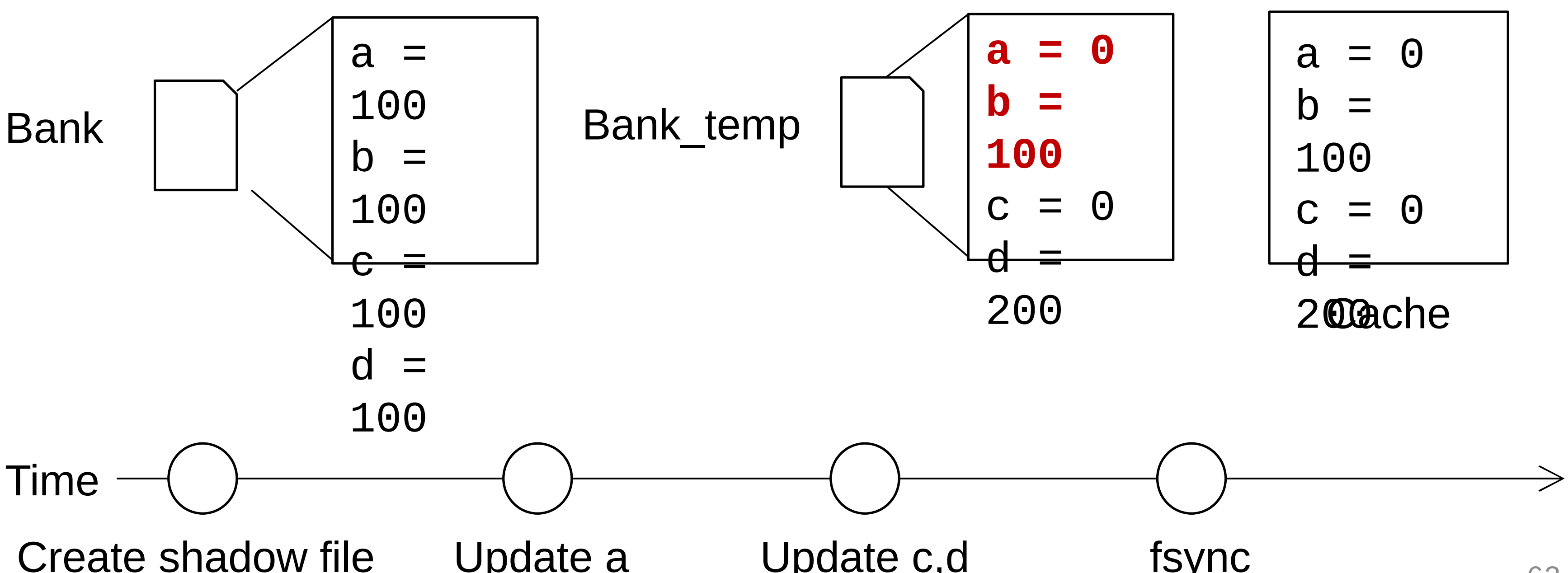


Drawbacks of shadow copy

Question: what would happen if multiple clients share the same file?

① Client 0 transfers 100 from a to b

② Client 1 transfers 100 from c to d

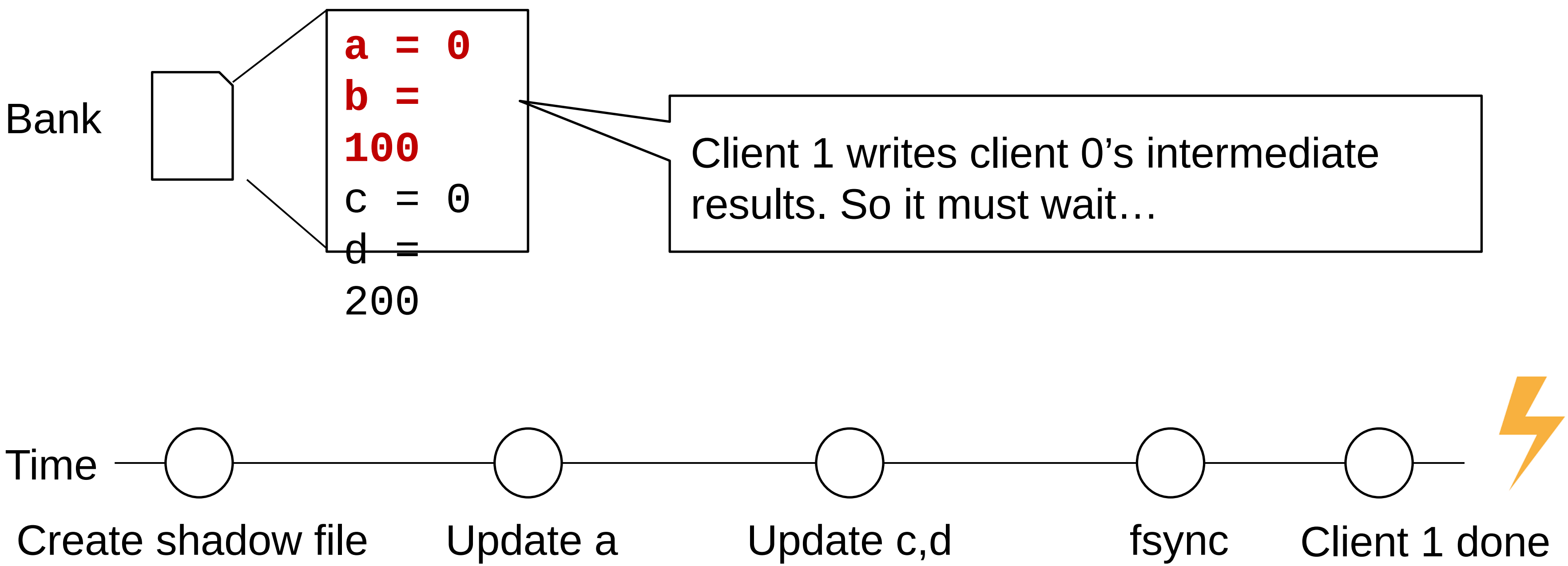


Drawbacks of shadow copy

Question: what would happen if multiple clients share the same file?

① Client 0 transfers 100 from a to b

② Client 1 transfers 100 from c to d

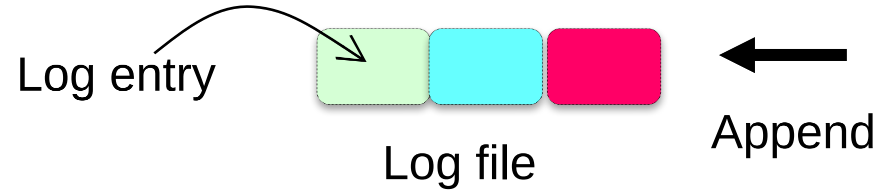


Drawbacks of shadow copy

- ✗ Only one operation can happen at a time
  - Naively executing concurrent transfer can cause consistency issue
  - Even these operations in principle can run concurrently
- ✗ Hard to **generalize to multiple files** or directories
  - Have to place all files in a single directory, or rename subdirs
- ✗ Requires copying the entire file for any (small) change

Logging for atomicity

Logging



Key idea

– Avoid updating the disk states until we can recovery it after failure

How to achieve so in shadow copy or journaling?

– Shadow copy buffers the updates in **a copy of the origin file**

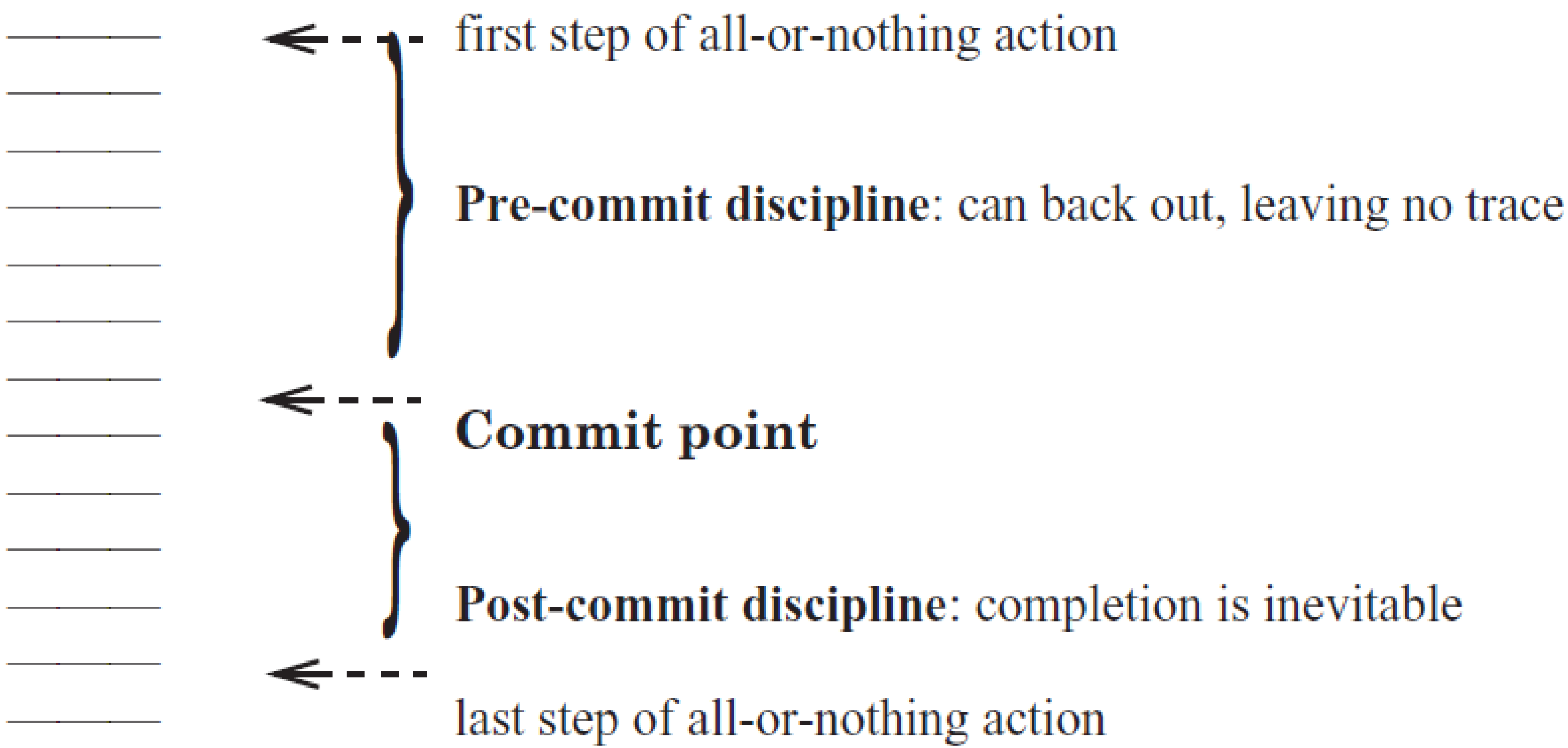
– Journaling buffers the updates **in a log file**

We can generalize this by storing all the updates in a log

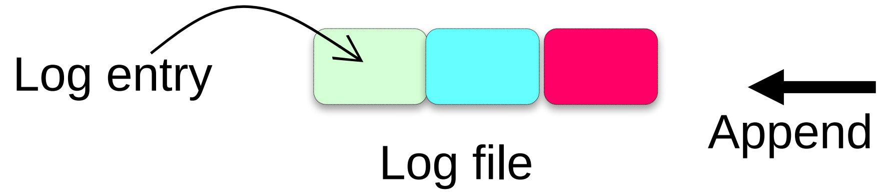
– Log file: a file only contains the updated results

– Log entries: contain the updated values of an atomic unit (e.g., transfer)

Transaction and commit Point: marking atomic units



Transaction and Commit Point



We call a set of operations that needs to be atomic "transaction"

Transaction typically provides interfaces for applications to mark the atomicity granularity of operations

- `TX.begin()`
- `TX.commit()`

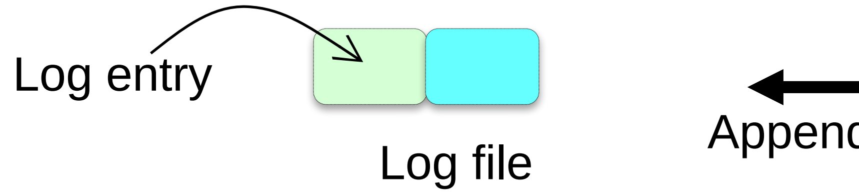
Each update between a `begin()` & `commit()` are stored in a log entry

- Can be replayed via replaying each entry of the log

Commit point

- The time when we are sure the operation is **"all"**

First try: commit logging

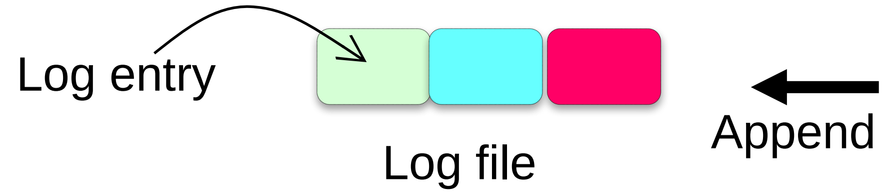



```
transfer(bank, a, b, amt, log): // amt=10
    records = mmap(bank, ...)
    new_a = records[a] – amt
    new_b = records[b] + amt

    commit_log = "log start: a:" + new_a + "\b:" + 
new_b
    log.append(commit_log).sync()

    record[a] = new_a
    record[b] = new_b
```

Updates are buffered in the memory

– To prevent writing a temporal value to the disk

First try: commit logging

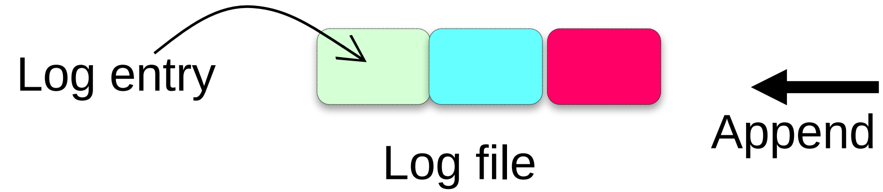



```
transfer(bank, a, b, amt, log): // amt=10
    records = mmap(bank, ...)
    new_a = records[a] – amt
    new_b = records[b] + amt

    commit_log = "log start: a:" + new_a + "\b:" + 
new_b
    log.append(commit_log).sync()

    record[a] = new_a
    record[b] = new_b
```

Before we write the disk, write the log to the disk synchronously

– Question: do we need these two steps to be atomic? How to achieve so?

First try: commit logging

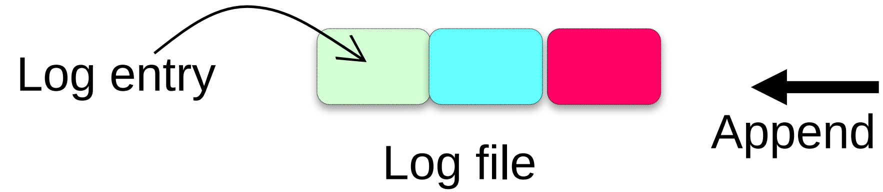



```
transfer(bank, a, b, amt, log): // amt=10
    records = mmap(bank, ...)
    new_a = records[a] – amt
    new_b = records[b] + amt

    commit_log = "log start: a:" + new_a + "\b:" +
new_b
    log.append(commit_log).sync()

    record[a] = new_a
    record[b] = new_b
```

Before we write the disk, write the log to the disk synchronously

- Yes. We need the log content to be atomically write to the disk
- Can be simply achieved by adding a checksum to the commit_log

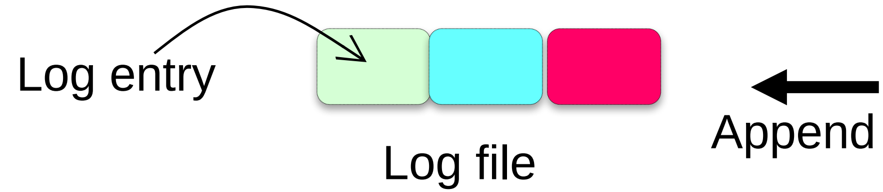
First try: commit logging




```
transfer(bank, a, b, amt, log): // amt=10
    records = mmap(bank, ...)
    new_a = records[a] – amt
    new_b = records[b] + amt

    commit_log = "log start: a:" + new_a + "\b:" + new_b
    log.append(commit_log).sync()

    record[a] = new_a
    record[b] = new_b
```

After the logging succeed, we can update the disk states

First try: commit logging

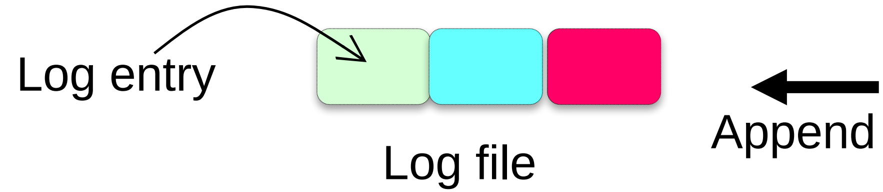



```
transfer(bank, a, b, amt, log): // amt=10
    records = mmap(bank, ...)
    new_a = records[a] – amt
    new_b = records[b] + amt

    commit_log = "log start: a:" + new_a + "\b:" +
new_b
    log.append(commit_log).sync()

    record[a] = new_a
    record[b] = new_b
    fsync(bank) // ?
```

Question: do we need to add fsync to flush the modifications to the bank file?

– Not necessary

Crash recovery of commit log



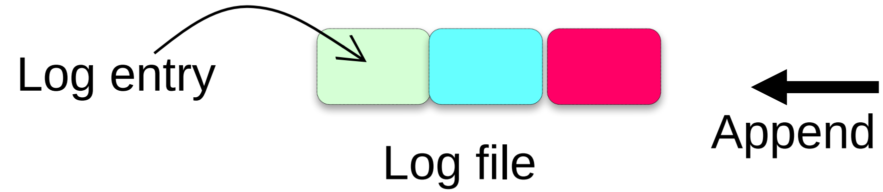
After reboot, we need to recover the systems to a consistent state

– Based on the log entries stored in the log file

Rules

1. Travel from start to end
2. Re-apply the updates recorded in a complete log entry

First try: commit logging




```
transfer(bank, a, b, amt, log): // amt=10
    records = mmap(bank, ...)
    new_a = records[a] – amt
    new_b = records[b] + amt

    commit_log = "log start: a:" + new_a + "\b:" +
new_b
    log.append(commit_log).sync()

    record[a] = new_a
    record[b] = new_b
```

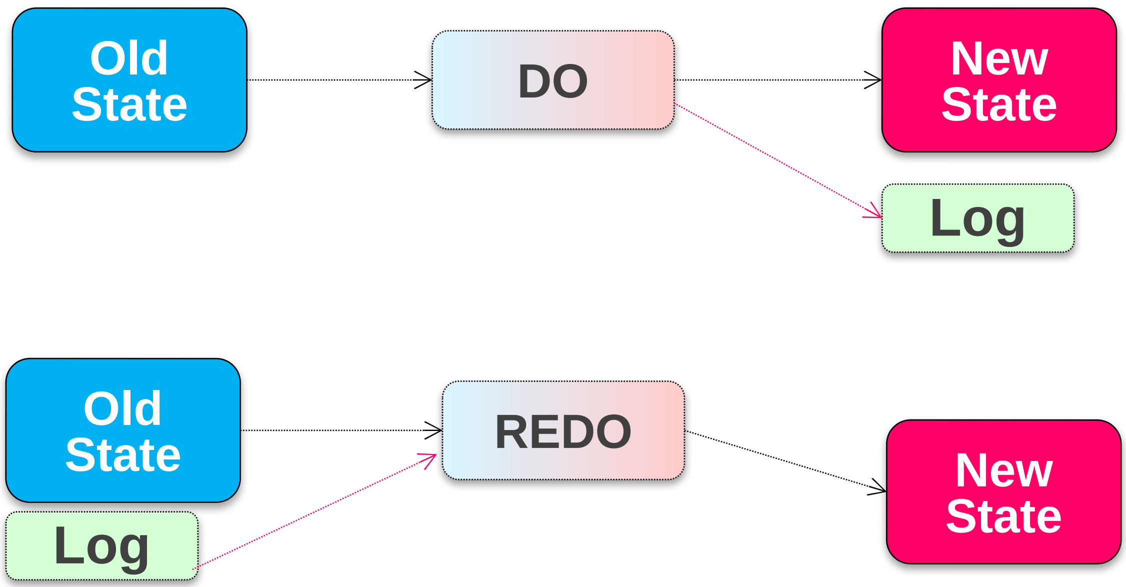
Question: What is the commit point of this transaction? (`transfer`)

– The line after `log.append(commit_log).sync()`

Quick summary: commit logging (redo-only logging)

Keep a log of all update actions

– Redo all the updates upon recovery



Journaling can be viewed as apply commit logging to filesystem

Pros & Cons of redo-only logging so far

Pros

- The commit is extremely efficient: only one file append operations (w/ updated data)
  - Other methods, e.g., shadow copy copies the entire file

Cons

- Wastes of disk I/O: all disk operations must happen at the commit point
- All updates must be buffered in the memory until the transaction commits
  - What if there is insufficient memory?
- The log file is continuously growing while most its updates are already flushed to the disk (unless the machine is rebooted or crashed, and we do the recovery)

Pros & Cons of redo-only logging so far

Pros

- The commit is extremely efficient: only one file append operations (w/ updated data)
  - Other methods, e.g., shadow copy copies the entire file

Cons

- Wastes of disk I/O: all disk operations must happen at the commit point
- All updates must be buffered in the memory until the transaction commits
  - What if there is insufficient memory?
- The log file is continuously growing while most its updates are already flushed to the disk

Unlike filesystem journaling, the user can commit a lot of entries in a transaction

Basic idea

We allow the transaction directly writing uncommitted values to the disk

- Before the commit point to free-up memory space & utilize disk I/O

```
transfer(bank, a, b, amt, log): // amt=10
    records = mmap(bank, ...)

    records[a] = records[a] – amt

    records[b] = records[b] + amt
```

The OS will flush the page back if out of the memory

Problem

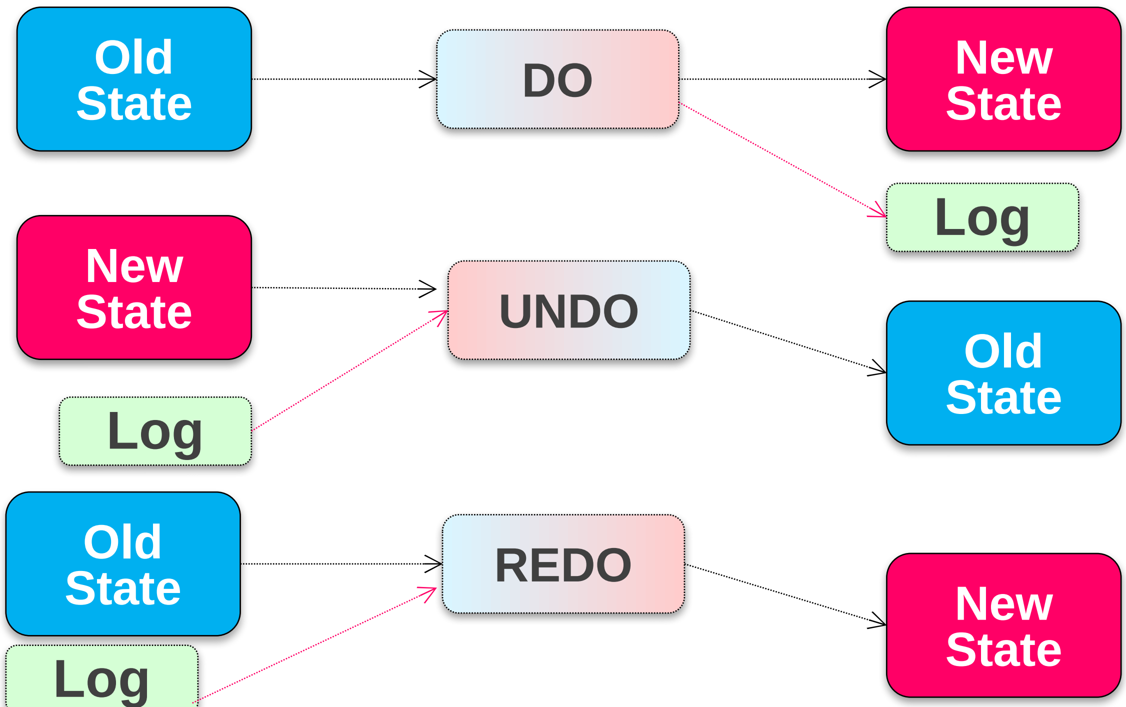
- How to prevent a partial updates from uncommitted transactions?

Idea: use log to **undo** updates of uncommitted transactions!

Undo logging

Keep a log of all update actions

– The log can undo the (partial) updates of a DO operation



Logging w/ undo

Before updates, write an undo log record to the log file

- Should contain sufficient information to undo uncommitted transactions
- E.g., old values

```
transfer(bank, a, b, amt, log): // amt=10
    records = mmap(bank, ...)
    log.append(...).sync()
    records[a] = records[a] – amt
    log.append(...).sync()
    records[b] = records[b] + amt
```

Action (file name, offset, **old value**)

Question: do we need the redo entry?

Logging w/ undo-redo logging

Question: do we need the redo entry?

- Depends on whether we wait for records[a] to be written to the disk (e.g., sync)
- **Typically, yes:** waiting two disk syncs are slow!
  - Especially for non-logging writes: log is a fast sequential disk write

```
transfer(bank, a, b, amt, log): // amt=10
    records = mmap(bank, ...)
    log.append(...).sync()
    records[a] = records[a] – amt
    log.append(...).sync()
    records[b] = records[b] + amt

    log.append("TX {id} commit").sync()
```

Action (file name, offset, **old & new values**)

Log entry vs. log record

Redo-only logging appends log entry to the log file

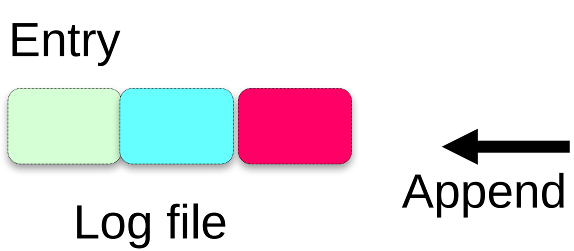
– Containing all the updates of the transaction

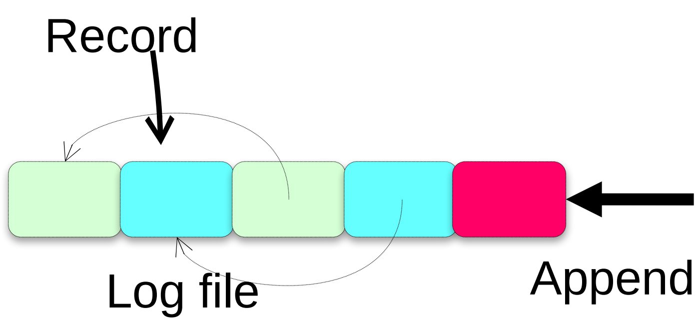
Und—redo logging appends log records to the log file

– Containing the updates of a single operation

– Log records from different transaction (TX) may possibly interleave

- E.g., the OS schedules the transaction out
- Therefore, we further need pointer to trace operations from the same TX

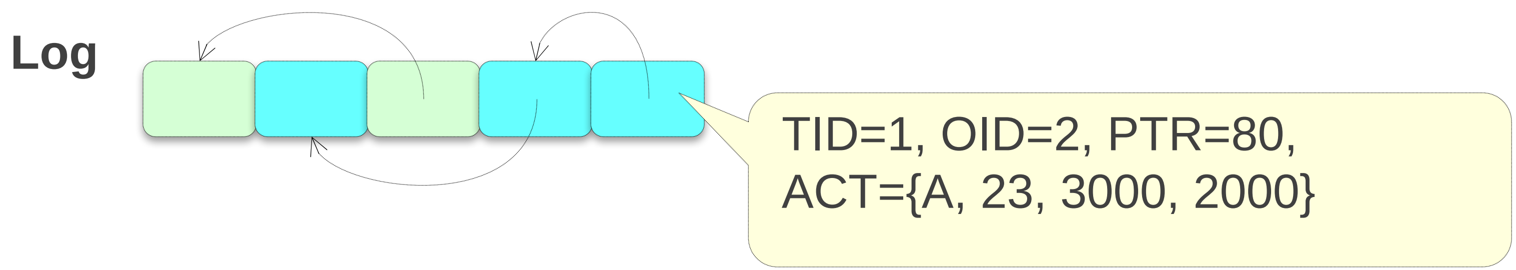




Put it all together: log record in undo-do logging

Each log record consists of

1. Transaction ID
2. Operation ID
3. Pointer to previous record in this transaction
4. Value (file name, offset, old & new value)
5. …

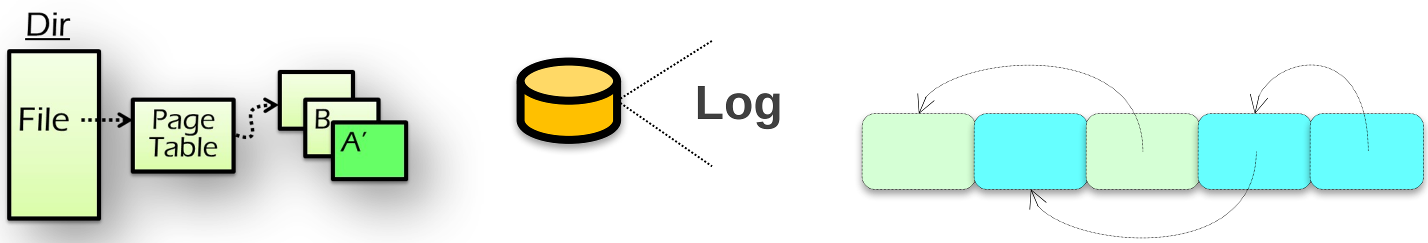


Put it all together: logging rules

```
transfer(bank, a, b, amt, log): // amt=10
  records = mmap(bank, ...)
  log.append(...).sync()
  records[a] = records[a] – amt
  log.append(...).sync()
  records[b] = records[b] + amt
  log.append("TX {id} commit”).sync()
```

Write log record to disk before modifying persistent state

- (e.g., replace A’s value)
- Write Ahead Log (WAL) protocol



Put it all together: logging rules

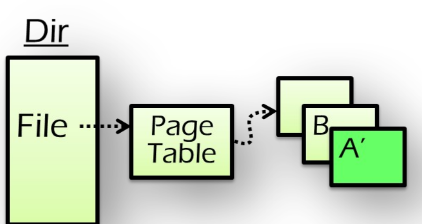
Write log record to disk before modifying persistent state

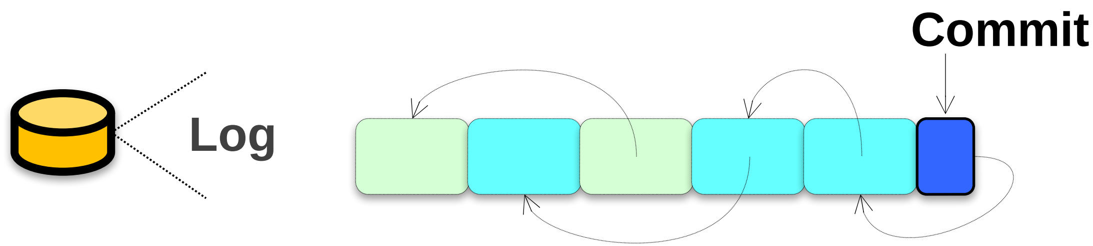
- (e.g., replace A’s value)
- Write Ahead Log (WAL) protocol

At commit point, append a commit record to the log last

- E.g., when user calls commit

```
transfer(bank, a, b, amt, log): // amt=10
  records = mmap(bank, ...)
  log.append(...).sync()
  records[a] = records[a] – amt
  log.append(...).sync()
  records[b] = records[b] + amt
  log.append("TX {id} commit”).sync()
```





Recovery rules

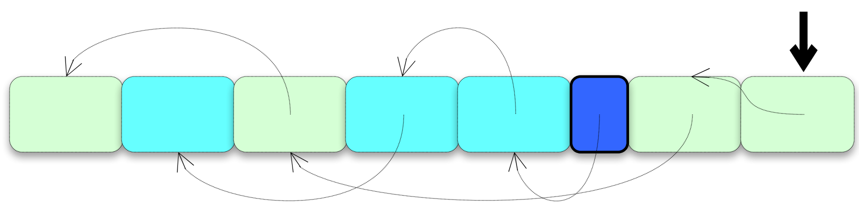
How to **recovery from crash**?

Read the log and recover states according to its content

Rules:

1. Travel from end to start

Log



Recovery rules

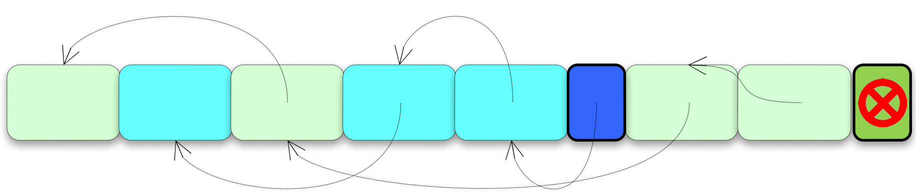
How to **recovery from crash**?

Read the log and recover states according to its content

Rules:

1. Travel from end to start
2. Mark all transaction's log record w/o CMT log and append ABORT log

Log



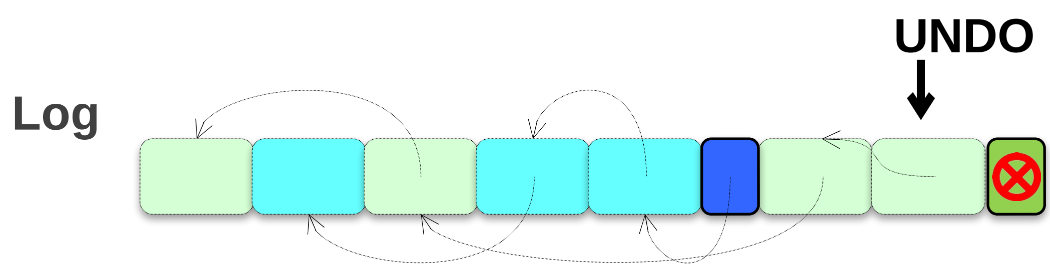
Recovery rules

How to **recovery from crash**?

Read the log and recover states according to its content

Rules:

1. Travel from end to start
2. Mark all transaction's log record w/o CMT log and append ABORT log
3. UNDO ABORT logs from end to start



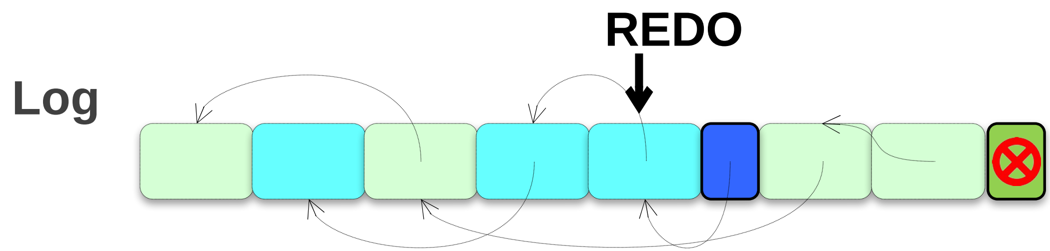
Recovery rules

How to recovery from crash?

Read the log and recover states according to its content

Rules:

1. Travel from end to start
2. Mark all transaction's log record w/o CMT log and append ABORT log
3. UNDO ABORT logs from end to start
4. REDO CMT logs from start to end



Problem: continuously growing of the log file

Both redo-only logging & undo-redo logging append to the log file

- The log file is continuously growing while most its updates are already flushed to the disk
- A large log file also makes recovery slow

Typically, a machine fails less frequent

- E.g., one per day

We need checkpoint the log file to reduce the log file size!

- Checkpoint: Determining which parts of the log can be discarded, then discarded them

Checkpoint the log

Naïve solution

- Run the recovery process. If it is done, then we can discard all the log file
- Problem: too slow

Observation

- Uncommitted updates are only in the page cache
- If we can flush all the page cache to the disk, then the committed TX's log can be discarded

Problem

- What if a TX is ongoing?

How to checkpoint?

Basic approach

1. Wait till no transactions are in progress
2. Flush the page cache
3. Discard all the logs

Question

- What if a TX is doing a long time? Can we allow ongoing TXs?
- Idea: can we wait till no **operations** are in progress?
- We need to reserve the log for ongoing TXs !

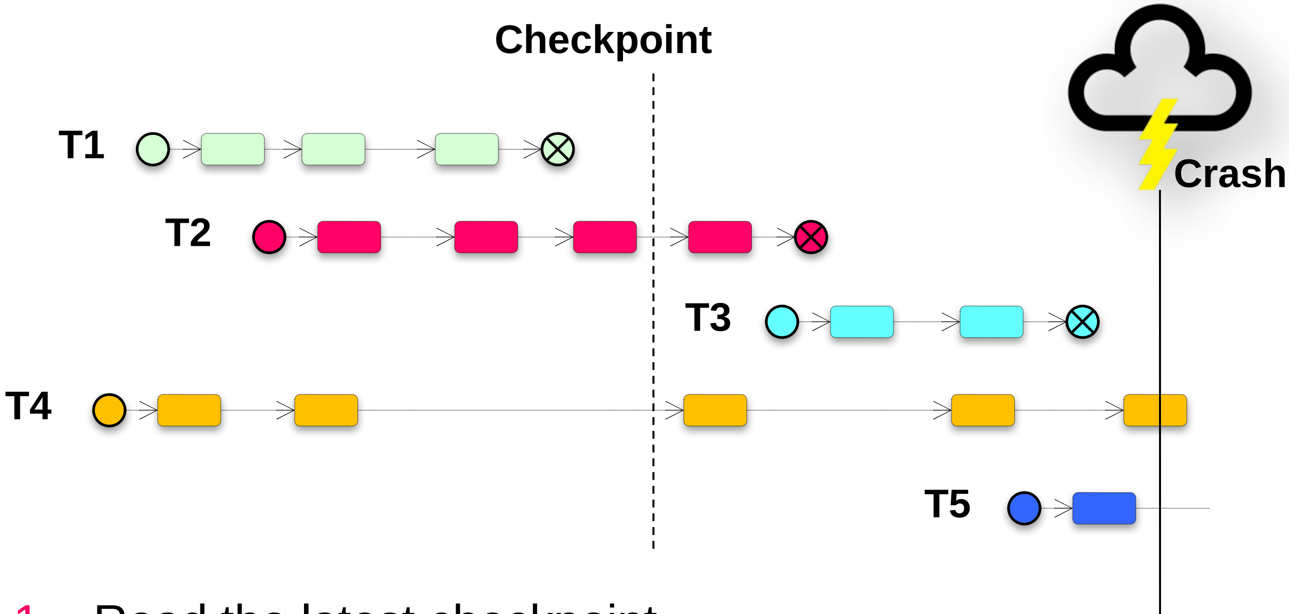
How to checkpoint?

actions

How to checkpoint?

1. Wait till no ~~transactions~~ are in progress
2. Write a **CKPT** record to log
    - Contains a list of all transaction in process and their logs
3. Flush the page cache
4. Discard all the log records except the CKPT record

Recovery with checkpoint



1. Read the latest checkpoint
   ✉ T2, T4 are ongoing transactions
2. Read log ✉ T2, T3 are committed, T5 are ongoing
3. Undo ongoing TXs & redo committed TXs

Undo-redo logging vs. redo-only logging

Question:

- Which one is faster during execution?
- Which one is faster during recovery?

Redo-only logging

- Less disk operations compared with undo-redo logging
- Only need one scan of the entire log file

Redo-only logging is typically preferred except for TXs with large in-memory states

UNDO-only Logging

Logging rules

- Append **UNDO** log record **before** flushing state modification
- State modification must be flushed before transaction committed
  - w/o REDO

Rarely used

- Much slower than UNDO-REDO logging during **execution**
- Though the **recovery** speed is faster

Summary

Systematic methods to support all-or-nothing atomicity

- Shadow copy
- Logging (including journaling)

Logging

- REDO logging
- UNDO-REDO logging
- UNDO-only logging

Question

Do log necessary be stored in a log file?

- No. It can be any mechanisms that supports atomic append & fault tolerant
  - E.g., in-memory logs replicated on many servers(see later lectures)

Logging: widely used in large-scale websites

All can use logging to ensure atomicity!