SE 464
Week 7

DB Consistency, Reliable &
High-Throughput Systems

Distributed DB Consistency

The following slides were taken with permission from Professor Steve Tarzia of Northwestern University.

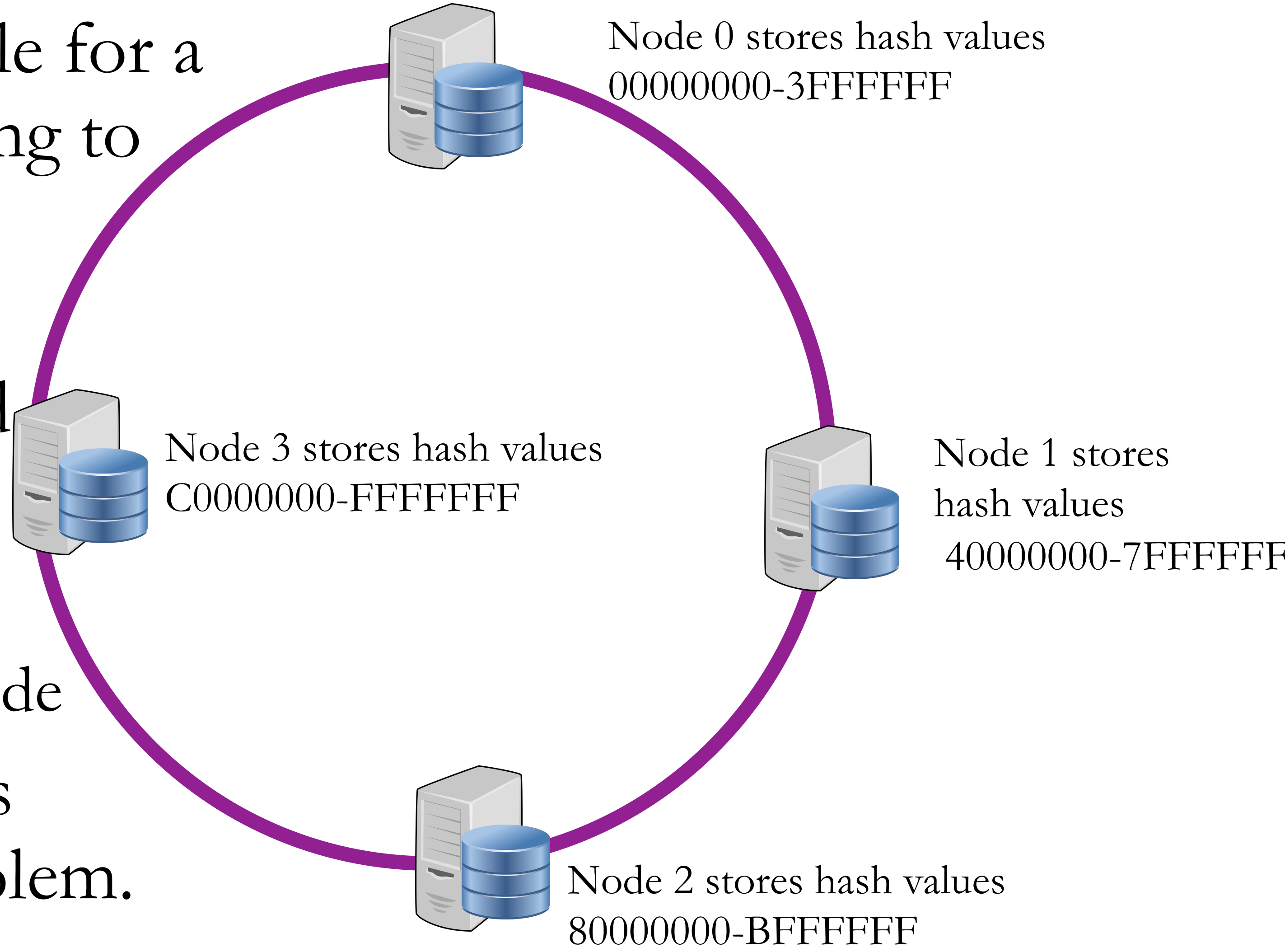
[Lecture 14: DB Consistency]

Last Time: NoSQL databases

- **Data partitioning** is necessary to divide write load among nodes.
  - Should minimize references between partitions.
  - Can be treated as a graph partitioning problem.
  - SQL sharding was a special case of data partitioning, done in app code.
- **NoSQL** databases make partitioning easy by eliminating references.
- Without references, data becomes **denormalized**.
  - Duplicated data consumes more space, can become inconsistent.
- **Distributed NoSQL databases** are very scalable, but they provide only a very simple **key-value** abstraction. One key is indexed.
- **Distributed Hash Table** can implement a NoSQL database.
  - The hash space is divided evenly between storage nodes.
  - Client computes hash of key to determine which node should store data.

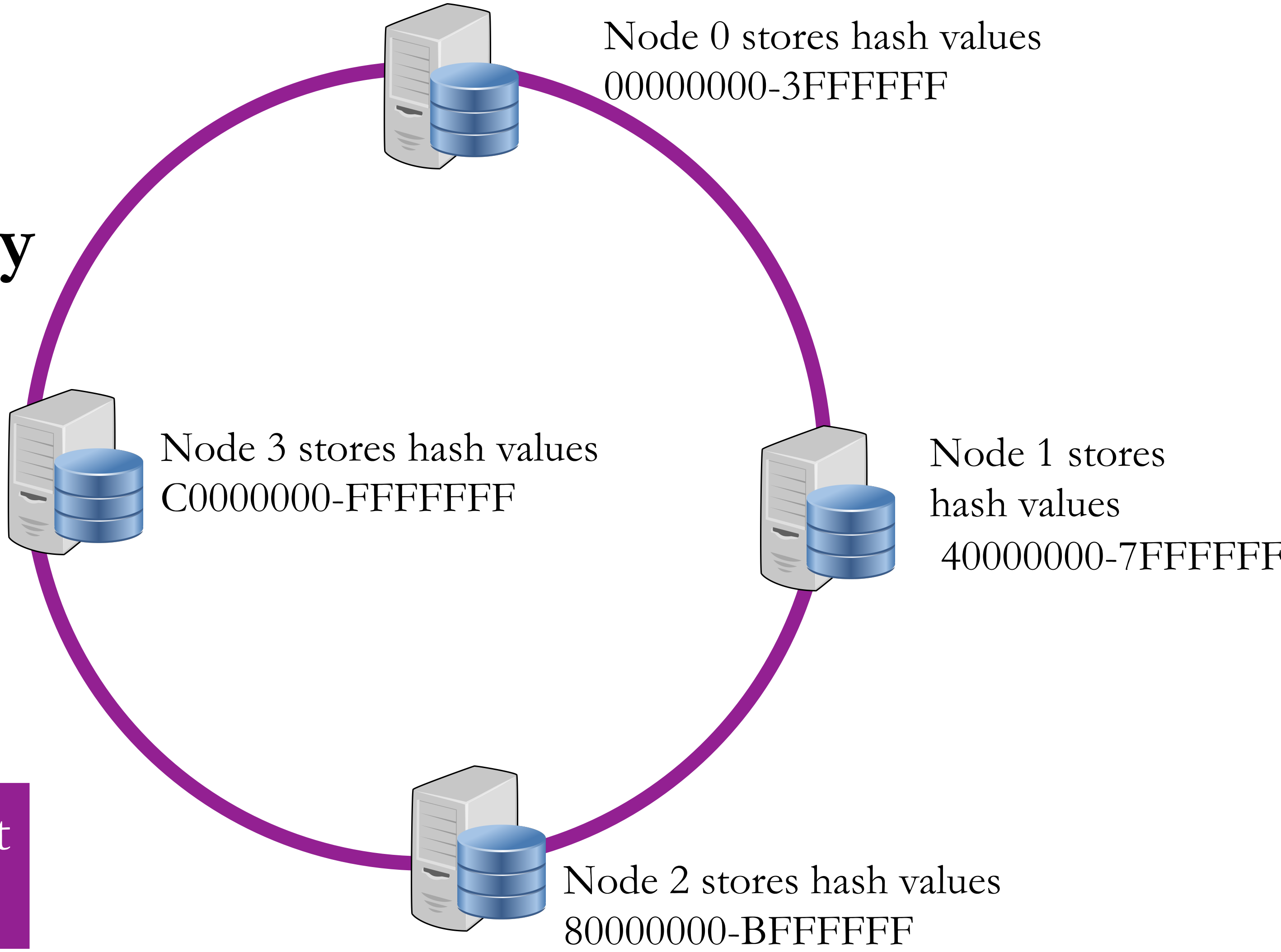
Hash-based partitioning of distributed DB

- aka, a Distributed Hash Table.
- Each cluster node is responsible for a *range of hash values* corresponding to an equal chunk of data
- Hash the **key** to determine where the (key, **value**) is stored
- To find data, client must have:
  - A list of all nodes.
  - <u>hash ranges</u> assigned to each node
- Sharing this node/range info is a **distributed consensus** problem.



A **Shared Nothing** architecture

- Each request is handled by **one** node.
  - There are no bottlenecks!
- Both **throughput** and **capacity** are directly proportional to the number of nodes.
- DHTs can scale to thousands of nodes.

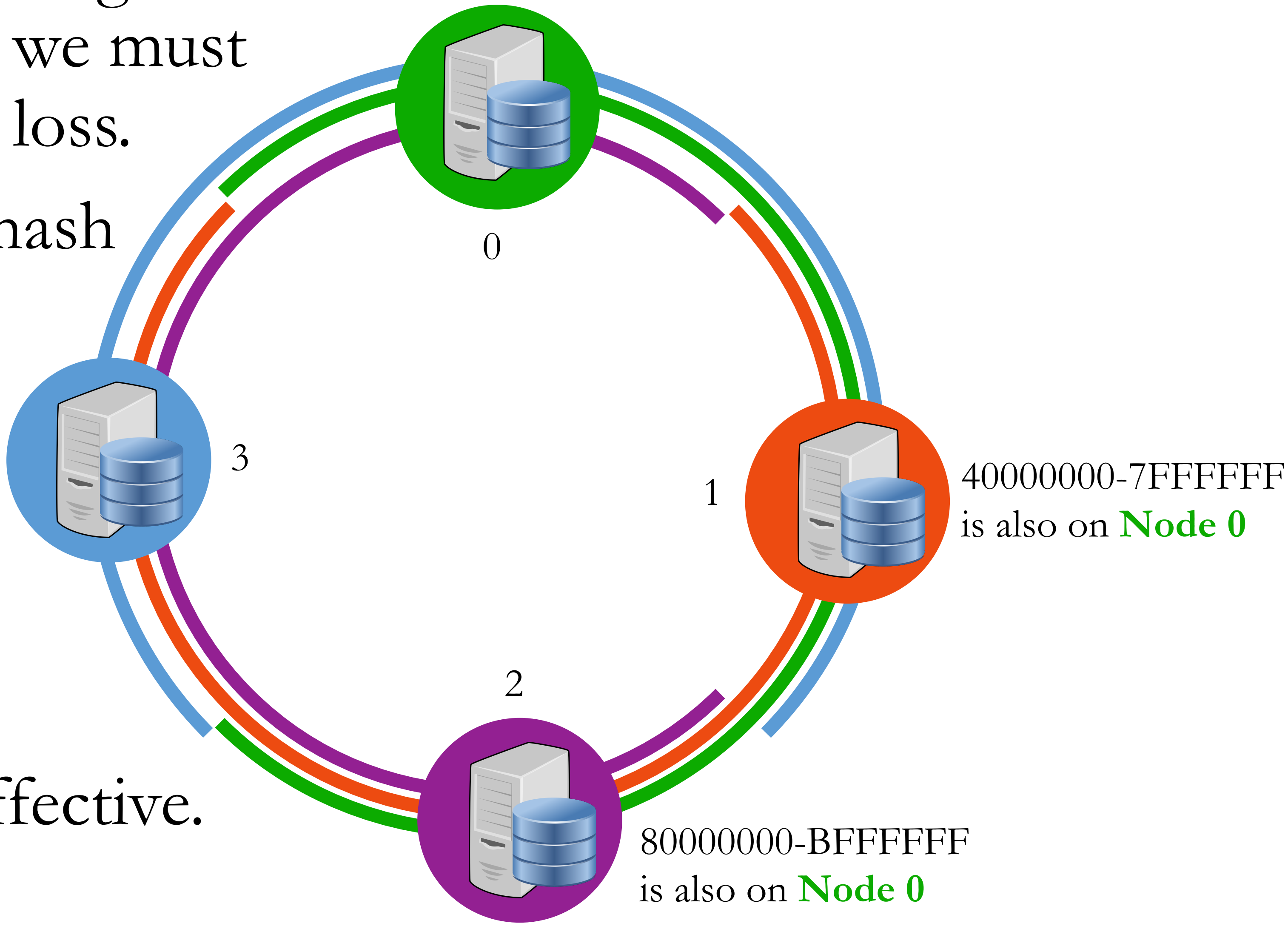


But what about **reliability?**

Making the DHT robust

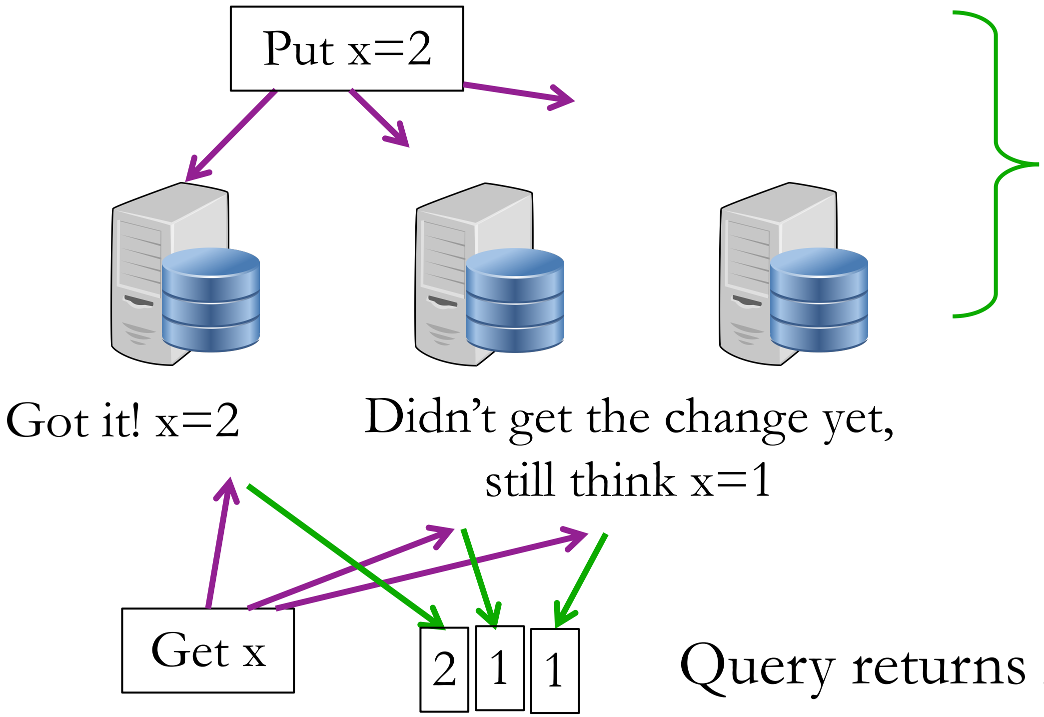
- Having Many nodes means a high chance of a node failure, so we must **replicate** data to avoid data loss.
- Create some overlap in the hash ranges covered by nodes.
  - *Node 0:* 0-7
  - *Node 1:* 3-F
  - *Node 2:* 0-3 and 8-F
  - *Node 3:* 0-7 and C-F
- Other schemes are possible, but this one is simple and effective.

Node 0 is assigned the hash values 00000000-3FFFFFF, but also stores replicas of data for the two next partitions.



Consistency

- Whenever data is replicated, there is a possibility of **inconsistency**.
  - Eg., an update was sent to three replicas, and one of them gets it first:



Communication and queueing delays are unpredictable. We can send messages, but don’t know exactly when they’ll arrive.

- What happens if we try to read while replicas are inconsistent?

CAP Theorem

The most famous result in distributed systems theory.
It says that a distributed system cannot achieve *all three* of the following:

- **Consistency**: reads always get the most recent write (or an error).
- **Availability:** every request received a non-error response.
- **Partition tolerance:** an arbitrary number of messages between nodes can be dropped (or delayed).

"Pick Two"

In other words:

- When distributed DB nodes are *out-of-sync* (partitioned), we must either accept **inconsistent** responses or **wait** for the nodes to resynchronize.
- To build a distributed DB where every request immediately gets a response that is globally correct, we need a network that is 100% reliable and has no delay.

Client-centric consistency models

- The CAP theorem gives us a tradeoff between **consistency** & **delay**.
- Inconsistency is bothersome. It can cause weird bugs.
- Fortunately, delay is usually something our apps can handle.
- If we really need both consistency and timeliness, then we must go back to a centralized database (probably a SQL relational DB).

- Distributed (NoSQL) DB designs give different options for handling the consistency/delay tradeoff.
- We'll consider a client connecting to the DB cluster.
- What consistency properties might we want to ensure?

Client-centric consistency properties

"More recently written" can include any write by another client.

Monotonic Reads

- If a client reads the value of **x**, later reads of x *by that same client* will always return the same value or a more recently written value.

Read your Writes

- If a client *writes* a value to **x**, later reads of x *by that same client* will always return the same value or a more recently written value.

Monotonic Writes

- If a client writes twice to **x**, the first write must happen before the second.

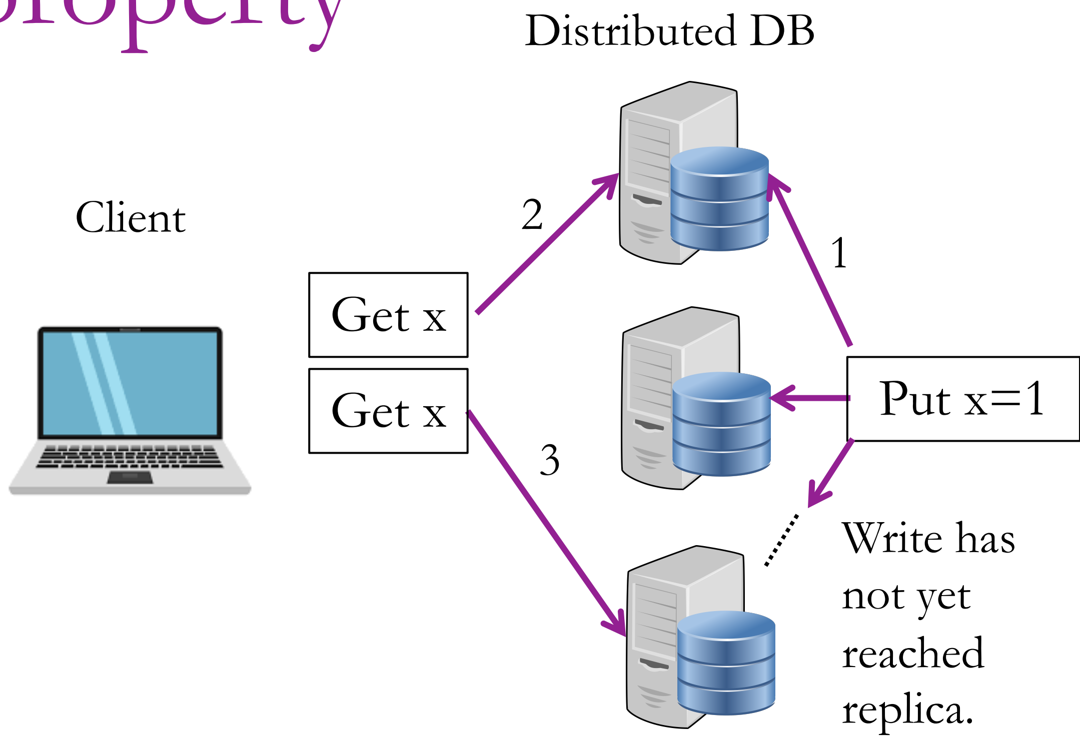
Failing the **Monotonic Read** property

Definition of Monotonic Reads:

- If a client reads the value of **x**, later reads of x *by that same client* will always return the same value or a more recently written value.

How might it fail?

- Read from two different nodes during an incomplete write.



How to prevent this problem?

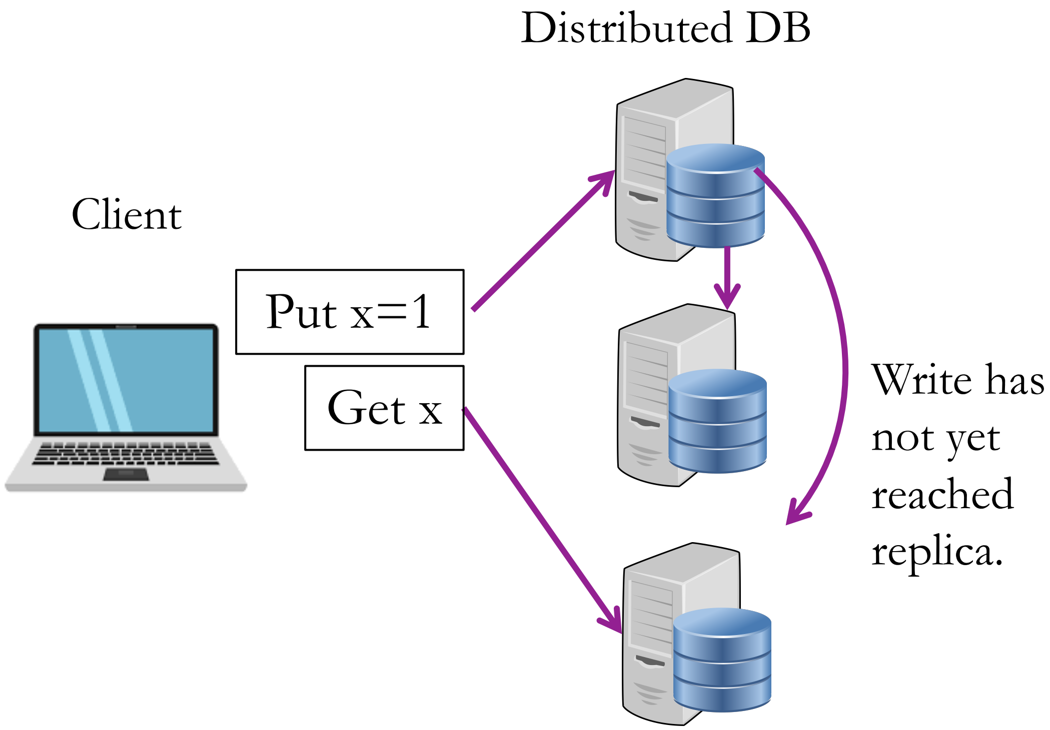
- Make client connect to same node for every request.
- Or delay the second request…

Failing to **Read your Writes**

Definition of Read your Writes:

- If a client *writes* a value to **x**, later reads of x *by that same client* will always return the same value or a more recently written value.

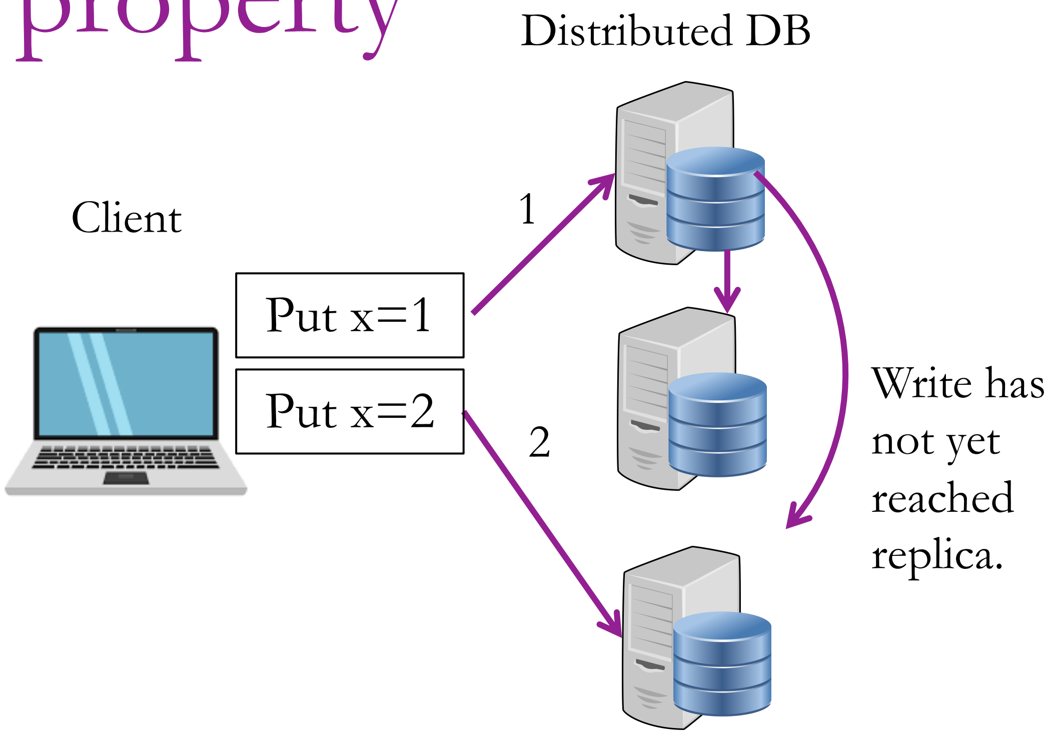
- If the system allows you to write on one node and read from another, you can get the old value if you read too quickly.
- Again, to fix this problem, stick with one node or "slow down."



Failing the **Monotonic Writes** property

Definition of Monotonic Writes:

- If a client writes twice to **x**, the first write must happen before the second.



- The second write can occur on a node before the first arrives.
- Does this matter?
  - Not unless the writes are cumulative. (eg., an increment operation)
  - Note that including a sequence number or timestamp would prevent the delayed write x=1 from being accepted on the third node.
- Solution: same as before.

Two alternatives for achieving Consistency

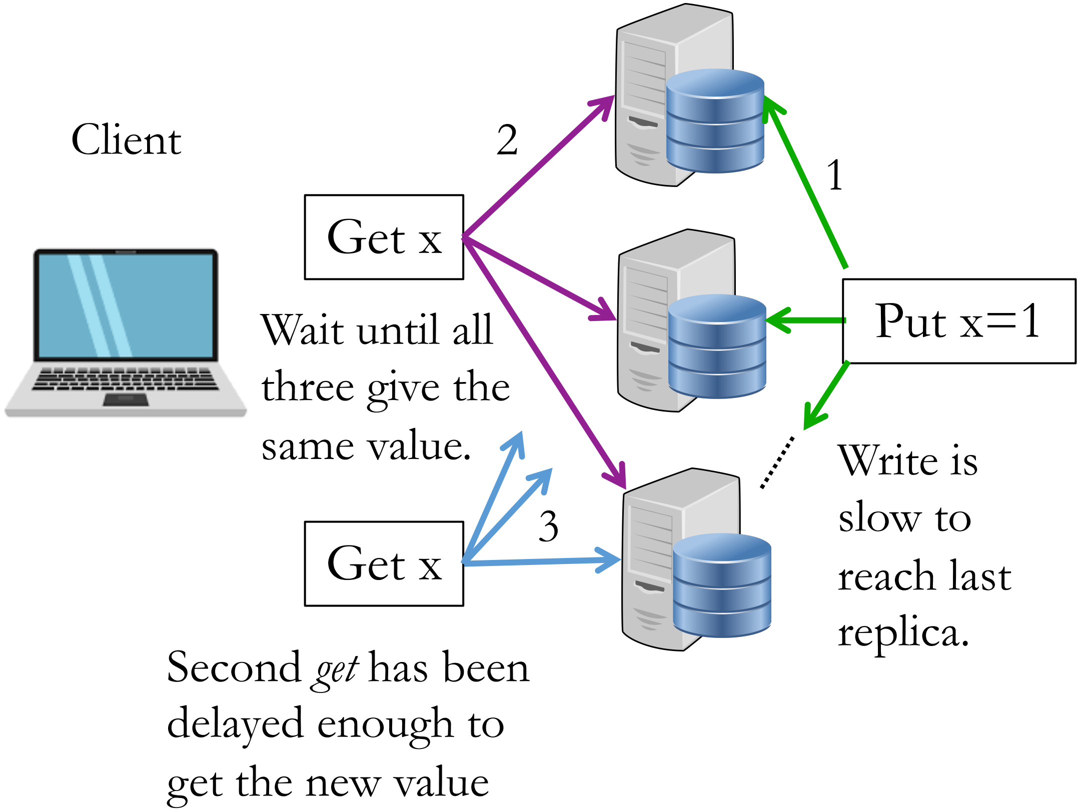
Set some rules for client and replication behavior to achieve consistency.

1. Make client send all requests to **one replica node**.
   - *Pro*: Simplicity.
   - *Con*: Consistency problems arise when a node fails.
     - Client must switch to another node, and the consistency problems are again possible.
     - *Note*: if don't care about fault tolerance, then avoid replication to get consistency. MongoDB does not replicate data and thus has Consistency and Partition Tolerance but lacks Availability because a failed node causes downtime (**C**A**P**).
2. Make client **wait** until the the read or write is synchronized across the whole system.
   - For efficiency, we only care about the single key/value being synchronized.
   - How do we know when the value is synchronized? 

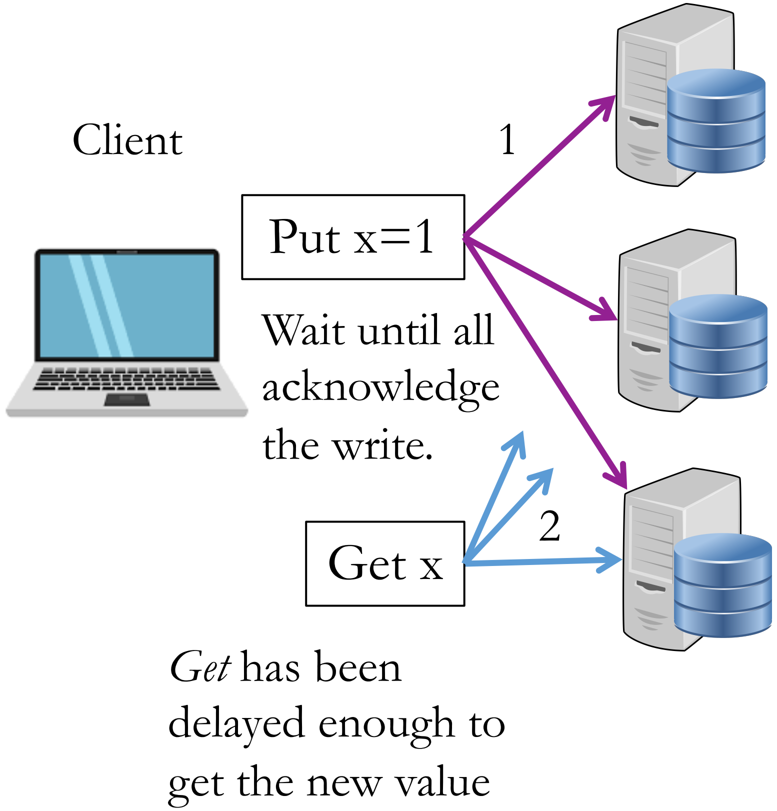
   - Simplest approach is for the client to send the request to all nodes and wait!

Waiting for Consistency

Monotonic Read:



- ## Read your Write:



Waiting for Consistency with **Quorums**

- A set of solutions for consistency in distributed DBs.
  - A **quorum** is a minimum percentage of a committee needed to act.
- Wait for an acknowledgement of consistent data from a certain number of replicas before considering the read/write completed.
  - **Prevents progress** until the replicas have a certain degree of consistency.

Write Quorum	Read Quorum	Optimized for
All	One	Fast reads
Majority	Majority	Balanced read/write performance
One	All	Fast writes

- We send requests to **all** nodes but wait for the prescribed # of responses.

Majority-read, majority-write example (three nodes)

- Client wants to write X=1.
  - Sends three write requests to three replicas.
  - When an **acknowledgement** from **two replicas** is received it can proceed.
  - The third/last write proceeds in the background.
- Client reads X
  - Sends three read requests to three replicas.
  - At this point, one of the replicas may still have old data, but that's OK!
  - Client will be satisfied when it receives two responses.
  - If they're different, use the most recent one.
    (Every write is timestamped by the client.)
- Because writes are not finished until at least two acknowledge, there is at most one old value being stored. At least one of two must be new.

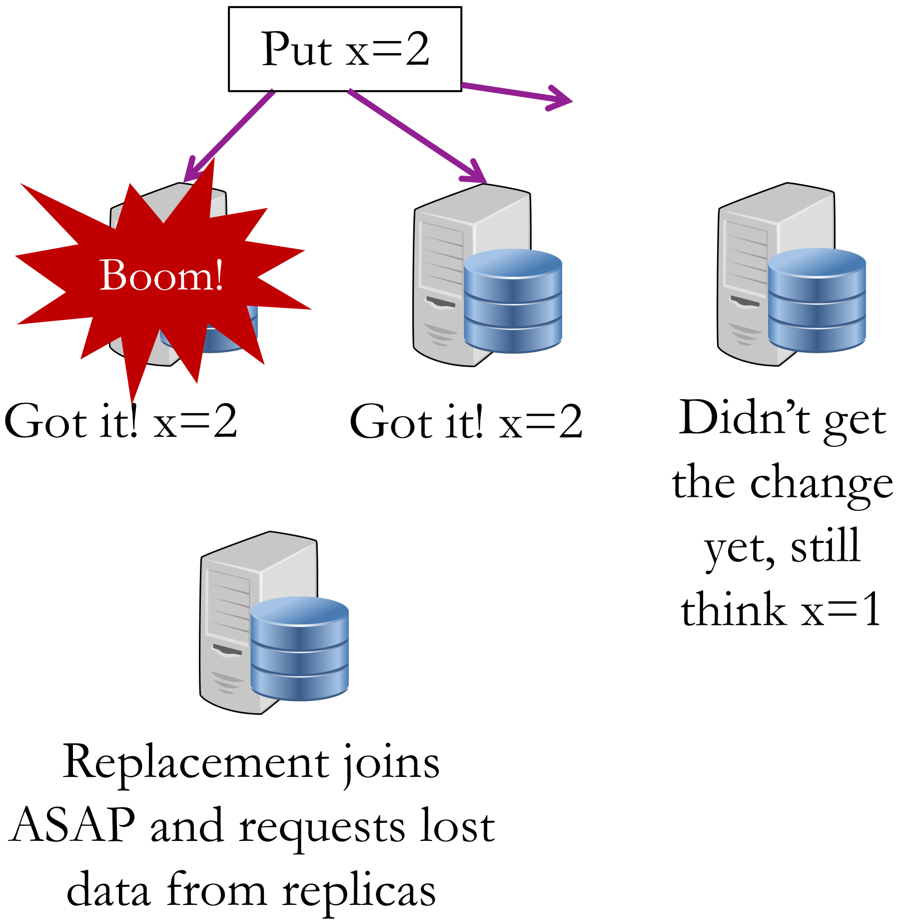
Single-read, unanimous-write example

- Client wants to write X=1.
  - Sends three write requests to three replicas.
  - Must wait until all **three replicas acknowledge** before proceeding.
- Client reads X
  - Sends three read requests to three replicas.
  - At this point, all three replicas must have received my previous write!
  - Client will be satisfied when it receives any **one** response.
  - Note that the responses from different nodes may be different (due to partial writes from other clients), but all will reflect data state after my own write.
    - Choose the latest value.
- Notice that writes are slow (max latency of the 3), but reads are fast (min latency of the 3).

Question: What happens if a DHT replica fails?

Example 1: write and read quorum of two (of three replicas).

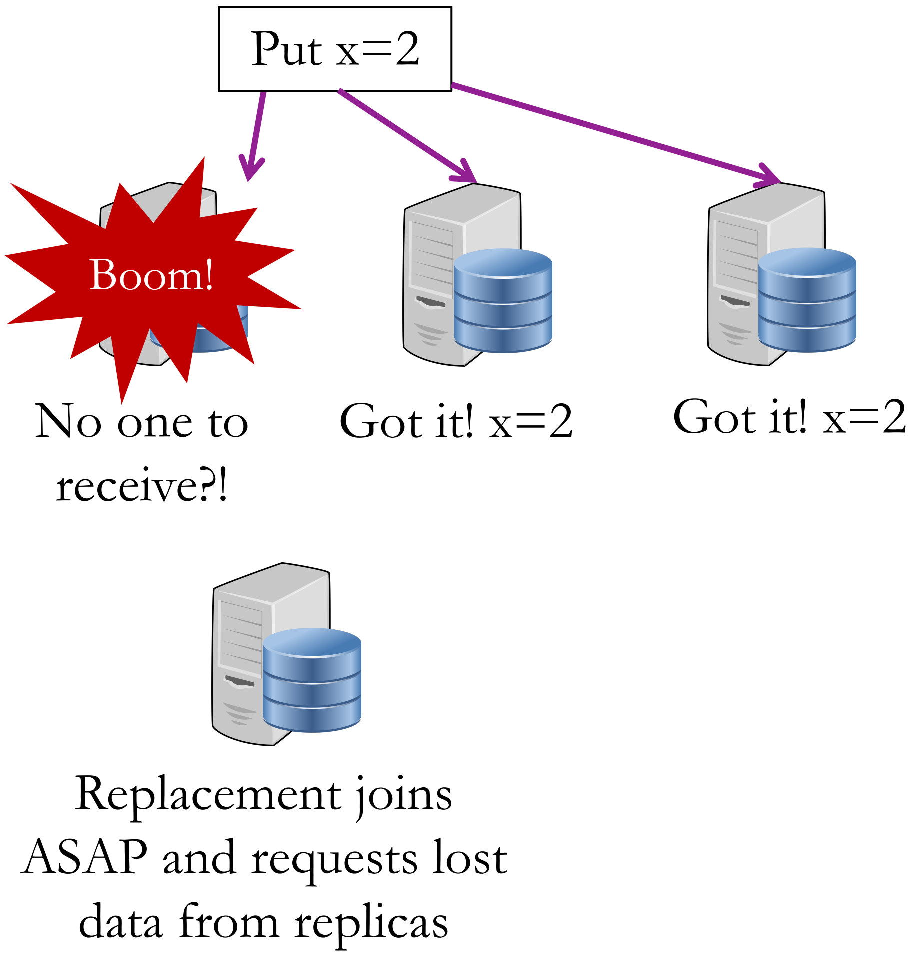
- Client performs a write, gets two ACKs and proceeds.
- At this point, replicas store two new values, and one old value.
- Now one of the written-to replicas fails!
- Can read and writes proceed?
- Yes. Two different values will be read, but client can choose the most recent one.
- The 3rd write will eventually be received, and two copies made available.



Question: What happens if a DHT replica fails?

Example 2: write quorum of three (read quorum of one)

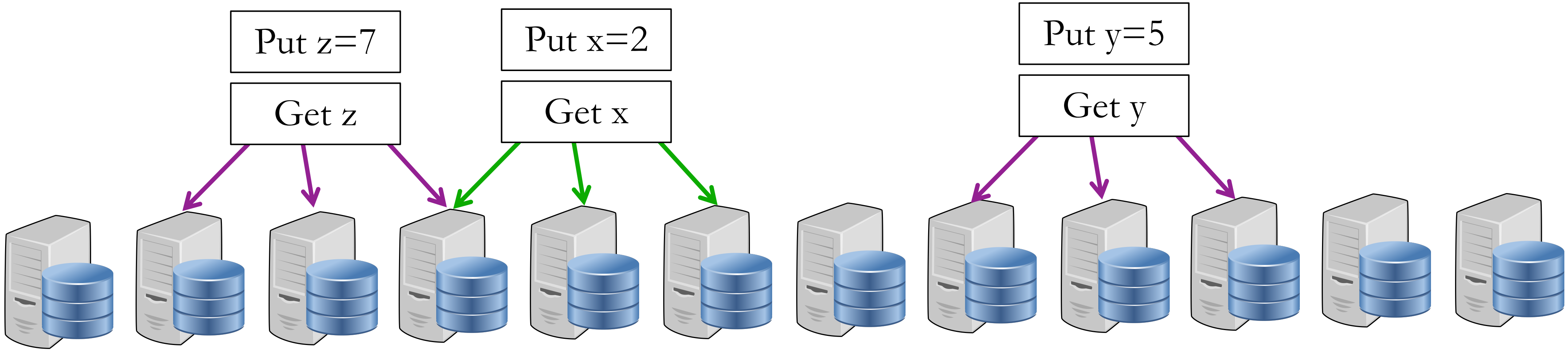
- A replica fails!
- Can reads and writes proceed?
- Client performs a write, and cannot get three ACKs.
  - Write is impossible! (but reads can proceed)
  - Part of the system is stalled, *temporarily*.
- The write can be retried after a replacement joins the DHT and gets copies of all the data.



Reminder: Why is this scalable?

- My consistency examples showed only three nodes == three replicas.
- This was not a scalable system because all nodes stored all data.
- In practice you can have a very large number N of nodes, and a constant number of replicas for each data key.
- Hashing will map each data key to a subset (often 3) of the N nodes.
- Quorum only apply to replica nodes. "Write to all" means all replicas (3 nodes).

Why use more than 3 replicas?



Distributed DB

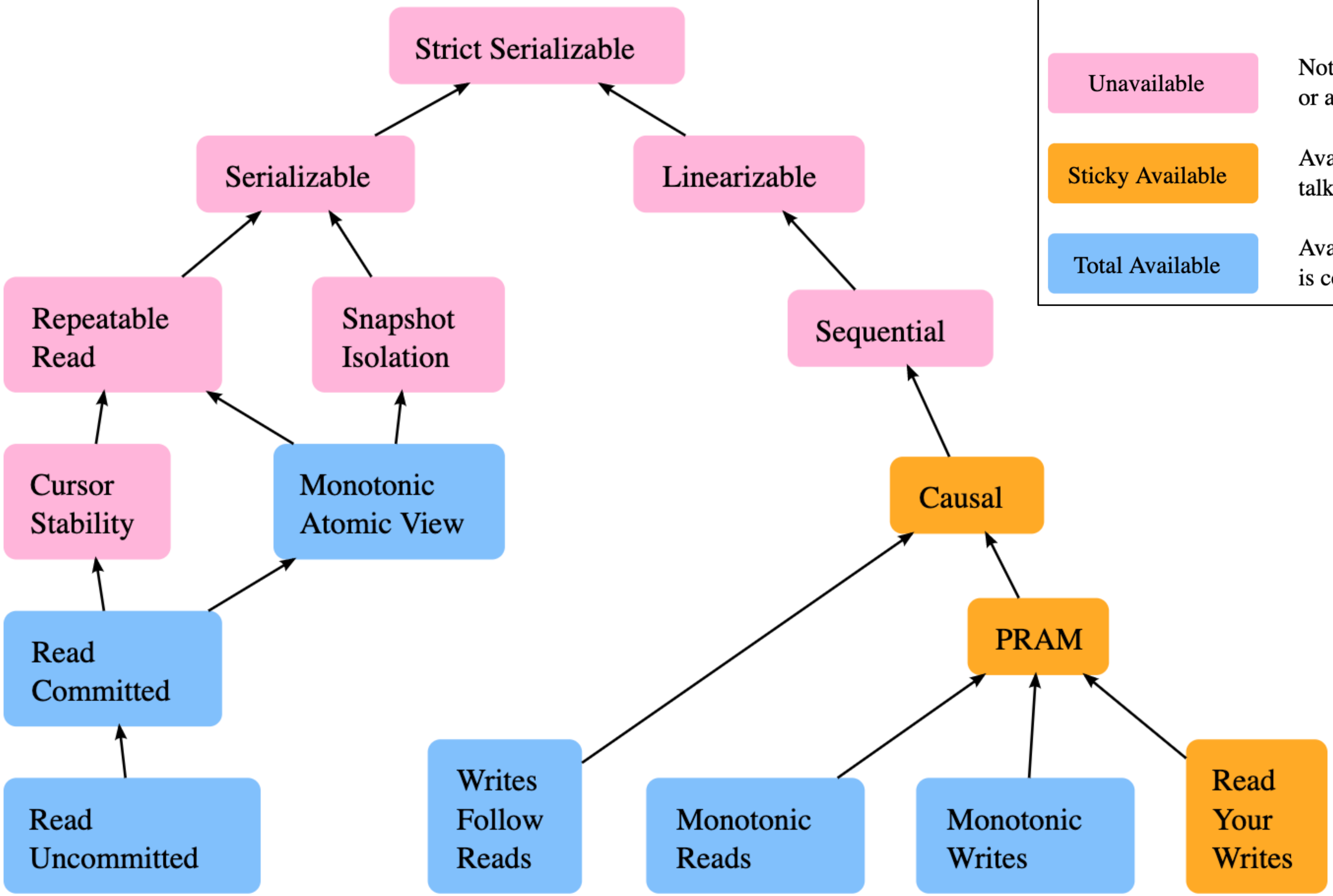
Another way of looking at consistency

- A distributed system is **linearizable** if the *partial ordering* of distributed actions is preserved.
  - The distributed actors each know the order of their **own** actions.
  - This certain knowledge must never be contradicted by the distributed system.
  - This creates a partial ordering of all the events in the distributed system
- For example:
  - if Anita does A, B, C (in that order)
  - and Sam does S, T, U, (in that order)

  These happen concurrently

  - Then no one should see B before A, nor U before T, etc.
- Every observed **serialization** of the parallel activity must be agreeable to the individual actors. Observations will vary across the system.
  - There are all valid: (A, B, C, S, T, U) (S, A, B, T, U, C) (S, T, U, A, B, C)

Consistency is a subtle topic, with [many models](#).



Legend

Unavailable	Not available during some types of network failures. Some or all nodes must pause operations in order to ensure safety.
Sticky Available	Available on every non-faulty node, so long as clients only talk to the same servers, instead of switching to new ones.
Total Available	Available on every non-faulty node, even when the network is completely down.

For more info on this fascinating topic

- CS-345 Distributed Systems
- Chapter 7 of Distributed Systems by van Steen and Tanenbaum.
- Part II (and Chapter 9 in particular) of the Designing Data Intensive Applications book by Kleppmann.
  - We covered the client-centric view of consistency.
  - Other models take a data-centric view.

- It's a nice mixture of CS theory and real system design.

NoSQL databases use DHTs or similar schemes

- Amazon DynamoDB
- Apache Cassandra
- ElasticSearch
- MongoDB (*hashed sharding* option)

Distributed filesystems can also use DHTs

- Filename/path is the key.
- Value is the file's contents.
- Hadoop HDFS, Google File System (Colossus, BigTable), Amazon S3

Recap: Distributed DB Consistency

- **Replication** of data ensures that a single failure does not lose data.
  - The more nodes you have, the more likely a failure!
- However, replication introduces **consistency** problems.
  - Tradeoff: must choose 2 of **C**onsistency, **A**vailability and **P**artition Tolerance.
- A distributed DB client, at very least, would want to achieve:
  - Montonic reads, monotonic writes, read your writes (together: linearizability).
- Ensure consistency by **waiting** for responses from multiple replicas.
- Different **quorum** levels (all, majority, one) trade delay of reads/writes and determine whether reads or writes are unavailable during recovery.
  - Cassandra DB lets programmer choose the quorum level for each read/write.
  - Other NoSQL databases are designed to use just one read/write strategy.

Principles of Reliable, High-Throughput Systems

The following content is sourced from Computer Systems Design from MIT OCW
https://ocw.mit.edu/courses/6-033-computer-system-engineering-spring-2018/pages/week-9/

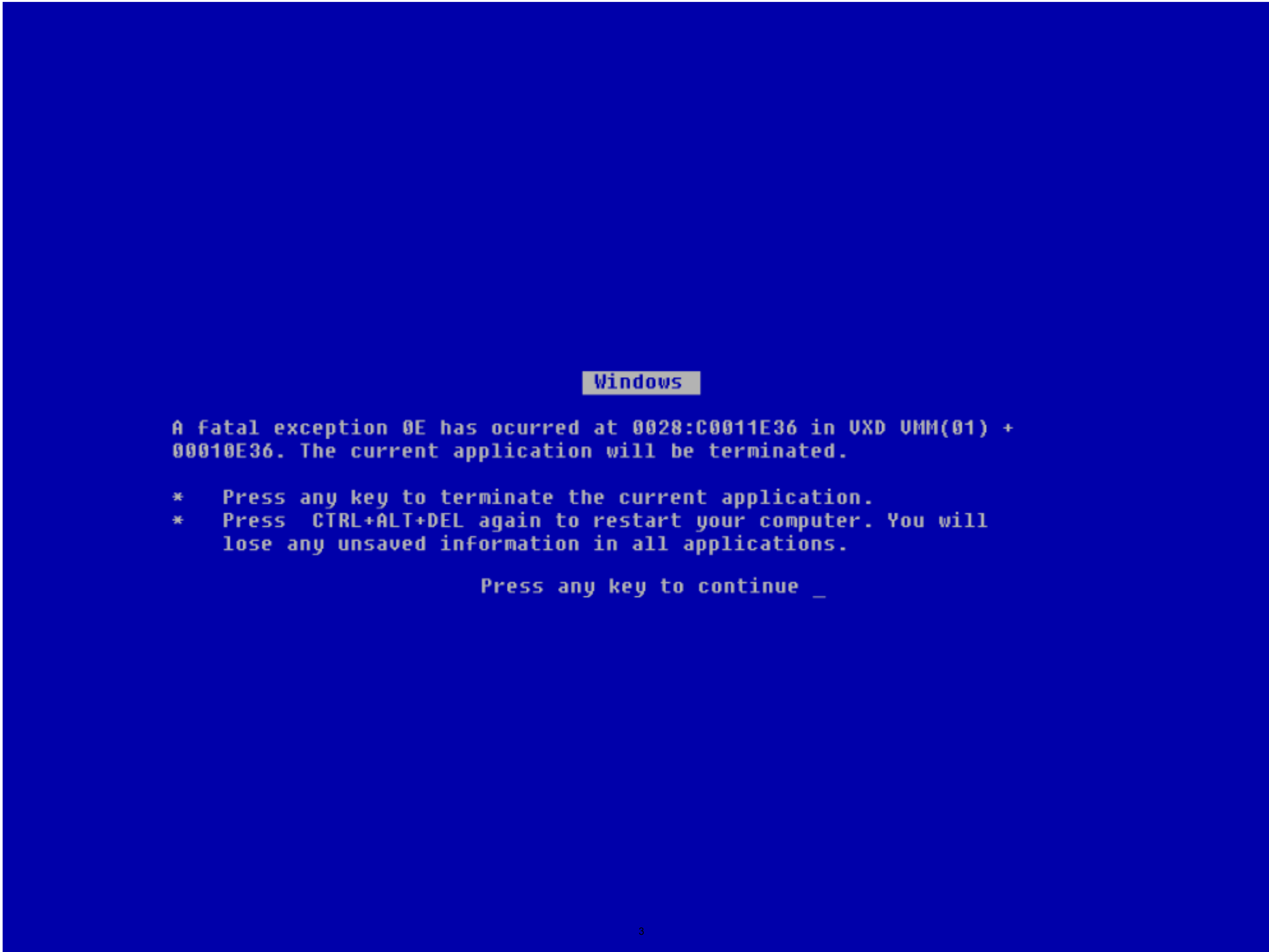
6.033 Spring 2018

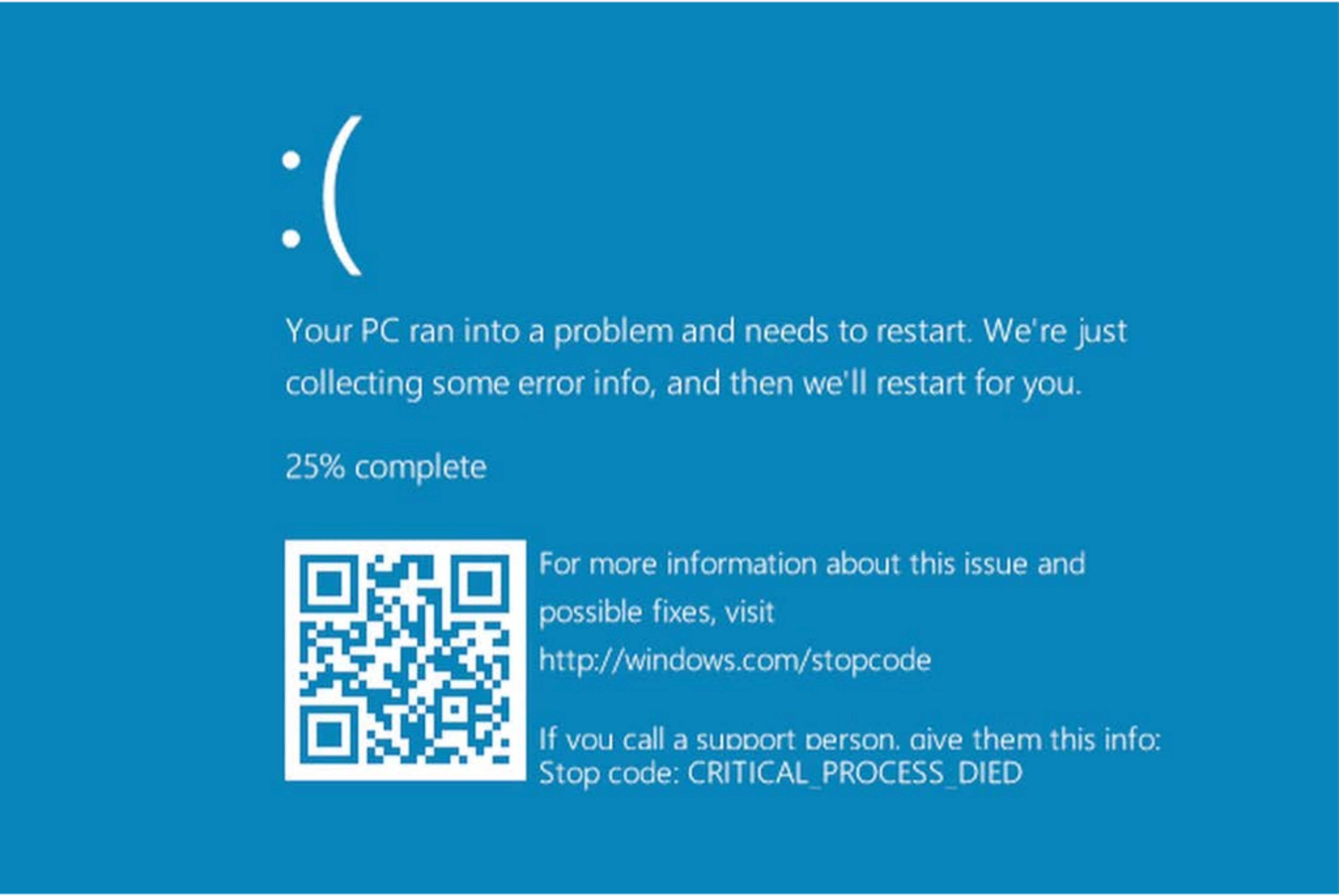
Lecture #14

- **Reliability via Replication**
  - **General approach to building fault-tolerance systems**
  - **Single-disk failures: RAID**

How to Design Fault-tolerant Systems in Three Easy Steps

1. identify all possible faults


Windows
A fatal exception 0E has ocurred at 0028:C0011E36 in VXD VMM(01) + 00010E36. The current application will be terminated.
* Press any key to terminate the current application.
* Press CTRL+ALT+DEL again to restart your computer. You will lose any unsaved information in all applications.
Press any key to continue _


:(
Your PC ran into a problem and needs to restart. We're just collecting some error info, and then we'll restart for you.
25% complete
For more information about this issue and possible fixes, visit http://windows.com/stopcode
If you call a support person, give them this info:
Stop code: CRITICAL_PROCESS_DIED


You need to restart your computer. Hold down the Power button for several seconds or press the Restart button.
Veuillez redémarrer votre ordinateur. Maintenez la touche de démarrage enfoncée pendant plusieurs secondes ou bien appuyez sur le bouton de réinitialisation.
Sie müssen Ihren Computer neu starten. Halten Sie dazu die Einschalttaste einige Sekunden gedrückt oder drücken Sie die Neustart-Taste.
コンピュータを再起動する必要があります。パワーボタンを数秒間押し続けるか、リセットボタンを押してください。

How to Design Fault-tolerant Systems in Three Easy Steps

1. identify all possible faults
2. detect and contain the faults
3. handle the fault

quantifying reliability

dealing with disk failures

Barracuda 7200.10

Experience the industry's proven flagship perpendicular 3.5-inch hard drive

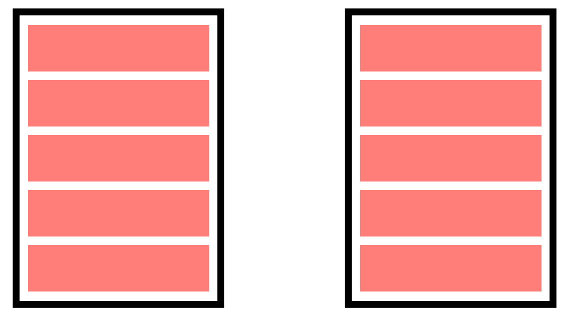
Seagate

Specifications	750 GB[1]	500 GB[1]	400 GB[1]	320 GB[1]	250 GB[1]		160 GB[1]	80 GB[1]
Model Number	ST3750640A ST3750640AS	ST3500630A ST3500630AS	ST3400620A ST3400620AS	ST3320620A ST3320620AS	ST3250620A ST3250620AS ST3250820A ST3250820AS	ST3250410AS ST3250310AS	ST3160815A ST3160815AS ST3160215A ST3160215AS	ST380815AS ST380215A ST380215AS
Interface Options	Ultra ATA/100 SATA 3Gb/s NCQ SATA 1.5Gb/s NCQ	Ultra ATA/100 SATA 3Gb/s NCQ SATA 1.5Gb/s NCQ	Ultra ATA/100 SATA 3Gb/s NCQ SATA 1.5Gb/s NCQ	Ultra ATA/100 SATA 3Gb/s NCQ SATA 1.5Gb/s NCQ	Ultra ATA/100 SATA 3Gb/s NCQ SATA 1.5Gb/s NCQ	SATA 3Gb/s NCQ SATA 1.5Gb/s NCQ	Ultra ATA/100 SATA 3Gb/s NCQ SATA 1.5Gb/s NCQ	Ultra ATA/100 SATA 3Gb/s NCQ SATA 1.5Gb/s NCQ
Performance								
Transfer Rate, Max Ext (MB/s)	100/300	100/300	100/300	100/300	100/300	100/300	100/300	100/300
Cache (MB)	16	16	16	16	16, 8	16, 8	8, 2	8, 2
Average Latency (msec)	4.16	4.16	4.16	4.16	4.16	4.16	4.16	4.16
Spindle Speed (RPM)	7200	7200	7200	7200	7200	7200	7200	7200
Configuration/Organization								
Heads/Disks[2]	8/4	6/3	5/3	4/2	3/2	2/1	2/1	1/1
Bytes per Sector	512	512	512	512	512	512	512	512
Reliability/Data integrity								
Contact Start-Stops	50,000	50,000	50,000	50,000	50,000	50,000	50,000	50,000
Nonrecoverable Read Errors per Bits Read	1 per 10^{14}	1 per 10^{14}	1 per 10^{14}	1 per 10^{14}	1 per 10^{14}	1 per 10^{14}	1 per 10^{14}	1 per 10^{14}
Mean Time Between Failures (MTBF, hours)	700,000	700,000	700,000	700,000	700,000	700,000	700,000	700,000
Annualized Failure Rate (AFR)	0.34%	0.34%	0.34%	0.34%	0.34%	0.34%	0.34%	0.34%
Limited Warranty (years)	5	5	5	5	5	5	5	5

700,000 hours ≈ 80 years

dealing with disk failures

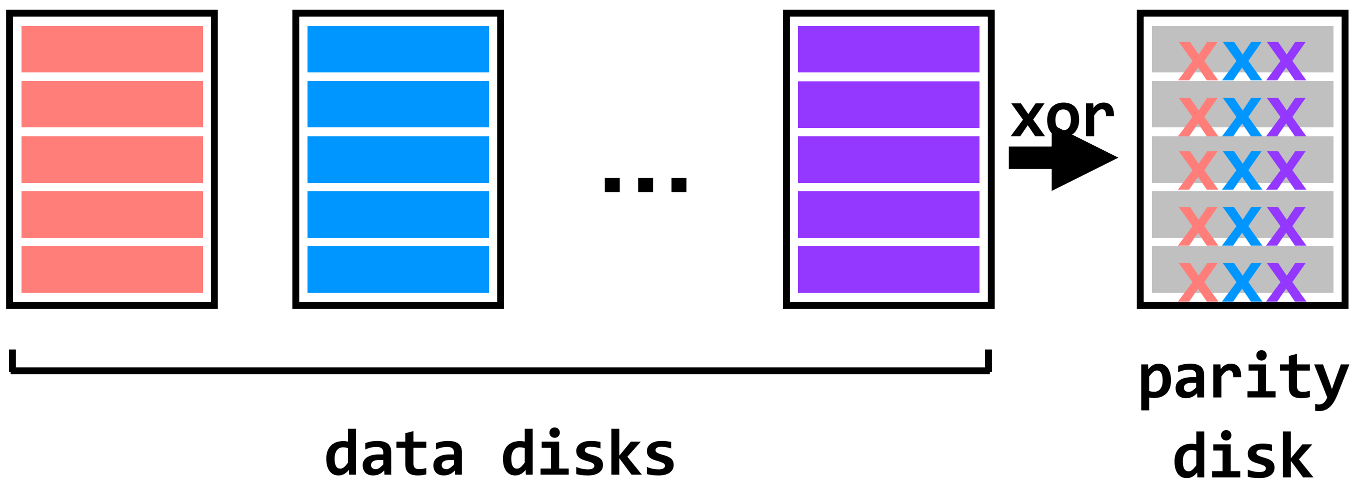
RAID 1 (mirroring)



😀 can recover from single-disk failure

😭 requires 2N disks

RAID 4 (dedicated parity disk)



sector i of the parity disk is the xor of sector i from all data disks

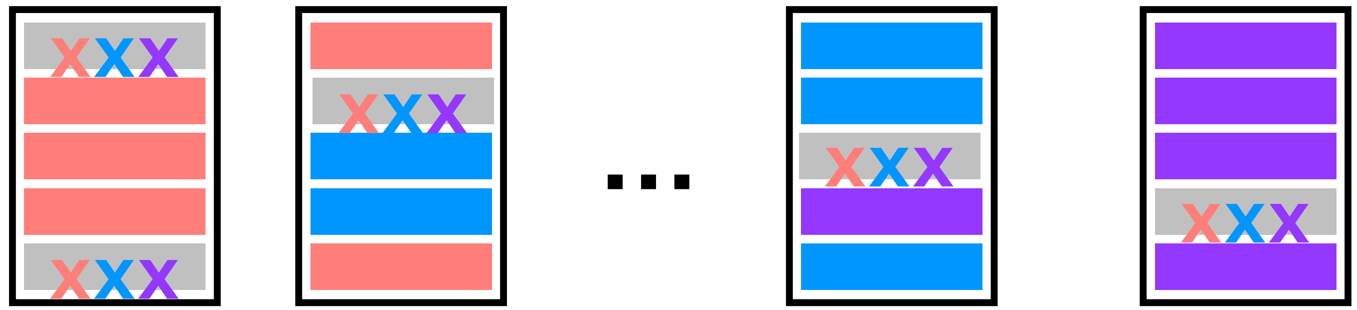
😀 can recover from single-disk failure

😀 requires N+1 disks (not 2N)

😀 performance benefits if you stripe a single file across multiple data disks

😭 all writes hit the parity disk

RAID 5 (spread out the parity)



😀 can recover from single-disk failure

😀 requires N+1 disks (not 2N)

😀 performance benefits if you stripe a single file across multiple data disks

😀 writes are spread across disks

- Systems have faults. We have to take them into account and build reliable, **fault-tolerant systems**. Reliability always comes at a cost — there are tradeoffs between reliability and monetary cost, reliability and simplicity, etc.

- Our main tool for improving reliability is **redundancy**. One form of redundancy is **replication**, which can be used to combat many things including disk failures (important, because disk failures mean lost data).

- **RAID** replicates data across disks in a smart way: RAID 5 protects against single-disk failures while maintaining good performance.

6.033 Spring 2018

Lecture #15

- **When replication fails us**
  - **Atomicity via shadow copies**
  - **Isolation**
  - **Transactions**

high-level goal: build reliable systems from unreliable components

this is difficult because reasoning about failures is difficult. we need some abstractions that will let us simplify.

atomicity

an action is atomic if it **happens completely or not at all**. if we can guarantee atomicity, it will be much easier to reason about failures

```
transfer (bank, account_a, account_b, amount):
    bank[account_a] = bank[account_a] - amount
    bank[account_b] = bank[account_b] + amount
```

← **crash!** 💥

problem: `account_a` lost **amount** dollars, but `account_b` didn't gain **amount** dollars

```
transfer (bank, account_a, account_b, amount):
    bank[account_a] = bank[account_a] - amount
    bank[account_b] = bank[account_b] + amount
```

← **crash!** 💥

solution: make this action atomic. ensure that we complete both steps or neither step.

quest for atomicity: attempt 1

```
transfer (bank_file, account_a, account_b, amount):
    bank = read_accounts(bank_file)
    bank[account_a] = bank[account_a] - amount
    bank[account_b] = bank[account_b] + amount
    write_accounts(bank_file)
```

crash! 💥

crash! 💥

problem: a crash during `write_accounts` leaves `bank_file` in an intermediate state

quest for atomicity: attempt 2

(shadow copies)

```
transfer (bank_file, account_a, account_b, amount):
    bank = read_accounts(bank_file)
    bank[account_a] = bank[account_a] - amount
    bank[account_b] = bank[account_b] + amount
    write_accounts(tmp_file)
    rename(tmp_file, bank_file)
```

crash! 💥

crash! 💥

crash! 💥

problem: a crash during `rename` potentially leaves `bank_file` in an intermediate state

quest for atomicity: attempt 2

(shadow copies)

```
transfer (bank_file, account_a, account_b, amount):
    bank = read_accounts(bank_file)
    bank[account_a] = bank[account_a] - amount
    bank[account_b] = bank[account_b] + amount
    write_accounts(tmp_file)
    rename(tmp_file, bank_file)  ← crash! 💥
```

solution: make `rename` atomic

quest for atomicity: making `rename` atomic

```
directory entries
   filename "bank_file" -> inode 1
   filename "tmp_file" -> inode 2
```

quest for atomicity: making `rename` atomic

```
directory entries
    filename "bank_file" -> inode 1
    filename "tmp_file" -> inode 2
```

```
inode 1: // old data
    data blocks: [..]
[..]
    refcount: 1
```

```
inode 2: // new data
    data blocks:
    refcount: 1
```

```
rename(tmp_file, orig_file):
    // point bank_file's dirent at inode 2
    // delete tmp_file's dirent
    // remove refcount on inode 1
```

quest for atomicity: making rename atomic

```
directory entries
    filename "bank_file" -> inode 2
    filename "tmp_file" -> inode 2

inode 1: // old data                 inode 2: // new data
    data blocks: [..]                    data blocks: [..]
    refcount: 1                          refcount: 1

rename(tmp_file, orig_file):
    tmp_inode = lookup(tmp_file)   // = 2
    orig_inode = lookup(orig_file) // = 1

    orig_file dirent = tmp_inode
    // delete tmp_file's dirent
    // remove refcount on inode 1
```

quest for atomicity: making rename atomic

```
directory entries
   filename "bank_file" -> inode 2


inode 1: // old data                inode 2: // new data
    data blocks: [..]                   data blocks:
[..]
    refcount: 1                         refcount: 1
```

```
rename(tmp_file, orig_file):
    tmp_inode = lookup(tmp_file)   // = 2
    orig_inode = lookup(orig_file) // = 1

    orig_file dirent = tmp_inode
    remove tmp_file dirent
    // remove refcount on inode 1
```

quest for atomicity: making rename atomic

```
directory entries
    filename "bank_file" -> inode 2

inode 1: // old data                 inode 2: // new data
    data blocks: [..]                    data blocks:
[..]
    refcount: 0                          refcount: 1
```

```
rename(tmp_file, orig_file):
    tmp_inode = lookup(tmp_file)    // = 2
    orig_inode = lookup(orig_file) // = 1

    orig_file dirent = tmp_inode
    remove tmp_file dirent
    decref(orig_inode)
```

quest for atomicity: making `rename` atomic

```
directory entries
    filename “bank_file” -> inode 1
    filename “tmp_file” -> inode 2
```

```
inode 1: // old data
    data blocks: [..]
[..]
    refcount: 1
```

```
inode 2: // new data
    data blocks:

    refcount: 1
```

```
rename(tmp_file, orig_file):
    tmp_inode = lookup(tmp_file)   // = 2
    orig_inode = lookup(orig_file) // = 1

    // point bank_file’s dirent at inode 2
    // delete tmp_file’s dirent
    // remove refcount on inode 1
```

quest for atomicity: making rename atomic

```
directory entries
    filename "bank_file" -> inode 1
    filename "tmp_file" -> inode 2

inode 1: // old data
    data blocks: [..]
[..]
    refcount: 1

inode 2: // new data
    data blocks:
    refcount: 1
```

```
rename(tmp_file, orig_file):
    tmp_inode = lookup(tmp_file)   // = 2
    orig_inode = lookup(orig_file) // = 1
                                   ← crash! 💥
    orig_file dirent = tmp_inode
    remove tmp_file dirent
    decref(orig_inode)
```

rename didn't happen

quest for atomicity: making rename atomic

```
directory entries
    filename "bank_file" -> inode 2
    filename "tmp_file" -> inode 2
```

```
inode 1: // old data
    data blocks: [..]
[..]
    refcount: 1
```

```
inode 2: // new data
    data blocks:

    refcount: 1
```

```
rename(tmp_file, orig_file):
    tmp_inode = lookup(tmp_file)   // = 2
    orig_inode = lookup(orig_file) // = 1

    orig_file dirent = tmp_inode
    remove tmp_file dirent
    decref(orig_inode)
```

crash! 💥

rename happened,
but refcounts are wrong

quest for atomicity: making rename atomic

```
directory entries
    filename "bank_file" -> inode ?
    filename "tmp_file" -> inode 2

inode 1: // old data
    data blocks: [..]
[..]
    refcount: 1

inode 2: // new data
    data blocks:

    refcount: 1
```

```
rename(tmp_file, orig_file):
    tmp_inode = lookup(tmp_file)   // = 2
    orig_inode = lookup(orig_file) // = 1

    orig_file dirent = tmp_inode
    remove tmp_file dirent
    decref(orig_inode)
```

crash! 💥

crash *during* this line seems bad..

quest for atomicity: making rename atomic

```
directory entries
    filename "bank_file" -> inode ?
    filename "tmp_file" -> inode 2

inode 1: // old data                inode 2: // new data
    data blocks: [..]                   data blocks:
[..]
    refcount: 1                         refcount: 1
```

```
rename(tmp_file, orig_file):
    tmp_inode = lookup(tmp_file)    // = 2
    orig_inode = lookup(orig_file) // = 1

    orig_file dirent = tmp_inode
    remove tmp_file dirent
    decref(orig_inode)
```

← **crash!** 💥

crash *during* this line seems bad..
but is okay because single-sector writes are themselves atomic

quest for atomicity: making rename atomic

```
directory entries
    filename "bank_file" -> inode 2
    filename "tmp_file" -> inode 2

inode 1: // old data                 inode 2: // new data
    data blocks: [..]                    data blocks: [..]
    refcount: 1                          refcount: 1
```

```
rename(tmp_file, orig_file):
    tmp_inode = lookup(tmp_file)   // = 2
    orig_inode = lookup(orig_file) // = 1

    orig_file dirent = tmp_inode
    remove tmp_file dirent
    decref(orig_inode)
```

crash! 💥

rename happened, but refcounts are wrong

quest for atomicity: making rename atomic

```
directory entries
    filename "bank_file" -> inode 2
    filename "tmp_file" -> inode 2

inode 1: // old data                inode 2: // new data
    data blocks: [..]                   data blocks:
[..]
    refcount: 1                         refcount: 1
```

```
rename(tmp_file, orig_file):
    tmp_inode = lookup(tmp_file)    // = 2
    orig_inode = lookup(orig_file) // = 1
    incref(tmp_inode)
    orig_file dirent = tmp_inode
    decref(orig_inode)
    remove tmp_file dirent
    decref(tmp_inode)
```

problem: this is a mess, and is still incorrect

solution: recover from failure

(clean things up)

```
recover(disk):
    for inode in disk.inodes:
        inode.refcount = find_all_refs(disk.root_dir, inode)
    if exists("tmp_file"):
        unlink("tmp_file")
```

atomicity

(first abstraction)

not quite solved; shadow copies perform poorly even for a single user and a single file, and we haven't even talked about concurrency

isolation

(second abstraction)

if we guarantee isolation, then two actions A1 and A2 will appear to have run **serially** even if they were executed concurrently

(i.e., A1 before A2, or vice versa)

transactions: provide atomicity and isolation

```
Transaction 1
  begin
  transfer(A, B, 20)
  withdraw(B, 10)
  end
```

```
Transaction 2
  begin
  transfer(B, C, 5)
  deposit(A, 5)
  end
```

atomicity: each transaction will appear to have run to completion, or not at all

isolation: when multiple transactions are run concurrently, it will appear as if they were run sequentially (serially)

atomicity and isolation — and thus, transactions — make it easier to reason about failures (and concurrency)

```
transfer (bank_file, account_a, account_b, amount):
    acquire(lock)
    bank = read_accounts(bank_file)
    bank[account_a] = bank[account_a] - amount
    bank[account_b] = bank[account_b] + amount
    write_accounts("tmp_file")
    rename("tmp_file", bank_file)
    release(lock)
```

couldn't we just put locks around everything?

(isn't that what locks are *for*?)

```
transfer (bank_file, account_a, account_b, amount):
    acquire(lock)
    bank = read_accounts(bank_file)
    bank[account_a] = bank[account_a] - amount
    bank[account_b] = bank[account_b] + amount
    write_accounts(“tmp_file”)
    rename(“tmp_file”, bank_file)
    release(lock)
```

this particular strategy will perform poorly

(would force a single transfer at a time)

```
transfer (bank_file, account_a, account_b, amount):
    acquire(lock)
    bank = read_accounts(bank_file)
    bank[account_a] = bank[account_a] - amount
    bank[account_b] = bank[account_b] + amount
    write_accounts("tmp_file")
    rename("tmp_file", bank_file)
    release(lock)
```

this particular strategy will perform poorly

(would force a single transfer at a time)

locks sometimes require global reasoning, which is messy

eventually, we'll incorporate locks, but in a systematic way

goal: to implement **transactions**, which provide atomicity and isolation, while not hindering performance

atomicity → **shadow copies.** work, but perform poorly and don't allow for concurrency

isolation → **?** (coarse-grained locks perform poorly, finer-grained locks are difficult to reason about)

eventually, we also want transaction-based systems to be **distributed**: to run across multiple machines

- **Transactions** provide **atomicity** and **isolation**, both of which make it easier for us to reason about failures because we don't have to deal with intermediate states.

- **Shadow copies** are one way to achieve atomicity. The work, but perform poorly: require copying an entire file even for small changes, and don't allow for concurrency.

6.033 Spring 2018

Lecture #16

- **Atomicity via Write-ahead logging**

goal: build reliable systems from unreliable components

the abstraction that makes that easier is

transactions, which provide **atomicity** and **isolation**, while not hindering **performance**

atomicity → **shadow copies** (simple, poor performance)

isolation → **?**

eventually, we also want transaction-based systems to be **distributed**: to run across multiple machines

goal: build reliable systems from unreliable components

the abstraction that makes that easier is

transactions, which provide **atomicity** and **isolation**, while not hindering **performance**

atomicity → **shadow copies** (simple, poor performance) or **logs** (better performance, a bit more complex)

isolation → **?**

eventually, we also want transaction-based systems to be **distributed**: to run across multiple machines

```
transfer(bankfile, account_a, account_b, amount):
  bank = read_accounts(bankfile)
  bank[account_a] = bank[account_a] – amount
  bank[account_b] = bank[account_b] + amount
  write_accounts(tmp_bankfile)
  rename(tmp_bankfile, bankfile)
```

using shadow copies to abort on error

```
transfer(bankfile, account_a, account_b, amount):
  bank = read_accounts(bankfile)
  bank[account_a] = bank[account_a] – amount
  bank[account_b] = bank[account_b] + amount
  if bank[account_a] < 0:
    print "Not enough funds"
  else:
    write_accounts("tmp_bankfile")
    rename(tmp_bankfile, bankfile)
```

with transaction syntax

```
transfer(account_a, account_b, amount):
  begin
  write(account_a, read(account_a) - amount)
  write(account_b, read(account_b) + amount)
  if read(account_a) < 0: // not enough funds
    abort
  else:
    commit
```

```
begin    // T1
A = 100
B = 50
commit   // A=100; B=50

begin    // T2
A = A-20
B = B+20
commit   // A=80; B=70

begin    // T3
A = A+30
```

crash! 💥

problem: after crash, **A=110**, but **T3** never committed

we need a way to revert to A's previous committed value

TID	T1 UPDATE	T1 UPDATE	T1 COMMIT	T2 UPDATE	T2 UPDATE	T2 COMMIT	T3 UPDATE
OLD	A=0	B=0		A=100	B=50		A=80
NEW	A=100	B=50		A=80	B=70		A=110

```
begin     // T1
A = 100
B = 50
commit    // A=100; B=50

begin     // T2
A = A-20
B = B+20
commit    // A=80; B=70

begin     // T3
A = A+30
```

```
    +--------+--------+--------+--------+--------+--------+--------+
TID | T1     | T1     | T1     | T2     | T2     | T2     | T3     |
    | UPDATE | UPDATE | COMMIT | UPDATE | UPDATE | COMMIT | UPDATE |
OLD | A=0    | B=0    |        | A=100  | B=50   |        | A=80   |
NEW | A=100  | B=50   |        | A=80   | B=70   |        | A=110  |
    +--------+--------+--------+--------+--------+--------+--------+
```

```
read(log, var):
  commits = {}
  // scan backwards
  for record r in log[len(log) - 1] .. log[0]:
    // keep track of commits
    if r.type == commit:
      commits.add(r.tid)
    // find var's last committed value
    if r.type == update and
       r.tid in commits and
       r.var == var:
        return r.new_value
```

```
    +--------+--------+--------+
TID |  T1    |  T1    |  T1    |
    | UPDATE | UPDATE | COMMIT |
OLD | A=0    | B=0    |        |
NEW | A=100  | B=50   |        |
    +--------+--------+--------+
```

```
begin     // T2
A = A-20
```

```
read(log, var):
  commits = {}
  // scan backwards
  for record r in log[len(log) - 1] .. log[0]:
    // keep track of commits
    if r.type == commit:
      commits.add(r.tid)
    // find var's last committed value
    if r.type == update and
       r.tid in commits and
       r.var == var:
        return r.new_value
```

```
commits = {}
```

```
    +---------+---------+---------+
TID |   T1    |   T1    |   T1    |
    | UPDATE  | UPDATE  | COMMIT  |
OLD | A=0     | B=0     |         |
NEW | A=100   | B=50    |         |
    +---------+---------+---------+
```

```
begin     // T2
A = A-20
```

```
read(log, var):
  commits = {}
  // scan backwards
  for record r in log[len(log) - 1] .. log[0]:
    // keep track of commits
    if r.type == commit:
      commits.add(r.tid)
    // find var's last committed value
    if r.type == update and
       r.tid in commits and
       r.var == var:
        return r.new_value
```

```
commits = {T1}
```

```
    +--------+--------+--------+--------+
TID |   T1   |   T1   |   T1   |   T2   |
    | UPDATE | UPDATE | COMMIT | UPDATE |
OLD | A=0    | B=0    |        | A=100  |
NEW | A=100  | B=50   |        | A=80   |
    +--------+--------+--------+--------+
```

```
begin      // T2
A = A-20
```

```
read(log, var):
  commits = {}
  // scan backwards
  for record r in log[len(log) - 1] .. log[0]:
    // keep track of commits
    if r.type == commit:
      commits.add(r.tid)
    // find var's last committed value
    if r.type == update and
       r.tid in commits and
       r.var == var:
        return r.new_value
```

```
    +--------+--------+--------+--------+
TID |   T1   |   T1   |   T1   |   T2   |
    | UPDATE | UPDATE | COMMIT | UPDATE |
OLD | A=0    | B=0    |        | A=100  |
NEW | A=100  | B=50   |        | A=80   |
    +--------+--------+--------+--------+
```

```
begin     // T2
A = A-20
A = A-30
```

```
read(log, var):
  commits = {}
  // scan backwards
  for record r in log[len(log) - 1] .. log[0]:
    // keep track of commits
    if r.type == commit:
      commits.add(r.tid)
    // find var's last committed value
    if r.type == update and
       r.tid in commits and
       r.var == var:
        return r.new_value
```

```
    +--------+--------+--------+--------+
TID | T1     | T1     | T1     | T2     |
    | UPDATE | UPDATE | COMMIT | UPDATE |
OLD | A=0    | B=0    |        | A=100  |
NEW | A=100  | B=50   |        | A=80   |
    +--------+--------+--------+--------+
```

```
begin      // T2
A = A-20
A = A-30
```

```
read(log, var):
  commits = {}
  // scan backwards
  for record r in log[len(log) - 1] .. log[0]:
    // keep track of commits
    if r.type == commit:
      commits.add(r.tid)
    // find var's last committed value
    if r.type == update and
       (r.tid in commits or r.tid == current_tid) and
       r.var == var:
        return r.new_value
```

```
    +--------+--------+--------+--------+--------+--------+--------+
TID |   T1   |   T1   |   T1   |   T2   |   T2   |   T2   |   T3   |
    | UPDATE | UPDATE | COMMIT | UPDATE | UPDATE | COMMIT | UPDATE |
OLD | A=0    | B=0    |        | A=100  | B=50   |        | A=80   |
NEW | A=100  | B=50   |        | A=80   | B=70   |        | A=110  |
    +--------+--------+--------+--------+--------+--------+--------+
```

```
begin      // T1
A = 100
B = 50
commit

begin      // T2
A = A-20
B = B+20
commit

begin      // T3
A = A+30
crash! 💥
```

after a crash, the log is still correct; uncommitted updates will not be read

TID	T1	T1	T1	T2	T2	T2	T3
	UPDATE	UPDATE	COMMIT	UPDATE	UPDATE	COMMIT	UPDATE
OLD	A=0	B=0		A=100	B=50		A=80
NEW	A=100	B=50		A=80	B=70		A=110

performance?

problem: reads are slow

```
     +--------+--------+--------+--------+--------+--------+--------+
 TID |   T1   |   T1   |   T1   |   T2   |   T2   |   T2   |   T3   |
     | UPDATE | UPDATE | COMMIT | UPDATE | UPDATE | COMMIT | UPDATE |
 OLD | A=0    | B=0    |        | A=100  | B=50   |        | A=80   |
 NEW | A=100  | B=50   |        | A=80   | B=70   |        | A=110  |
     +--------+--------+--------+--------+--------+--------+--------+
```

cell storage (on disk) A 110 B 70

```
read(var):
  return cell_read(var)


write(var, value):
  log.append(current_tid, update, var, read(var), value)
  cell_write(var, value)
```

```
     +--------+--------+--------+--------+--------+--------+--------+
 TID |   T1   |   T1   |   T1   |   T2   |   T2   |   T2   |   T3   |
     | UPDATE | UPDATE | COMMIT | UPDATE | UPDATE | COMMIT | UPDATE |
 OLD | A=0    | B=0    |        | A=100  | B=50   |        | A=80   |
 NEW | A=100  | B=50   |        | A=80   | B=70   |        | A=110  |
     +--------+--------+--------+--------+--------+--------+--------+
```

cell storage (on disk) $\boxed{\texttt{A 110}}$ $\boxed{\texttt{B 70}}$

```
recover(log):
  commits = {}
  for record r in log[len(log)-1] .. log[0]:
    if r.type == commit:
      commits.add(r.tid)
    if r.type == update and r.tid not in commits:
      cell_write(r.var, r.old_val) // undo
```

TID	T1 UPDATE	T1 UPDATE	T1 COMMIT	T2 UPDATE	T2 UPDATE	T2 COMMIT	T3 UPDATE
OLD	A=0	B=0		A=100	B=50		A=80
NEW	A=100	B=50		A=80	B=70		A=110

cell storage (on disk) A 110 B 70

commits = {}

```
recover(log):
  commits = {}
  for record r in log[len(log)-1] .. log[0]:
    if r.type == commit:
      commits.add(r.tid)
    if r.type == update and r.tid not in commits:
      cell_write(r.var, r.old_val) // undo
```

TID	T1	T1	T1	T2	T2	T2	T3
	UPDATE	UPDATE	COMMIT	UPDATE	UPDATE	COMMIT	UPDATE
OLD	A=0	B=0		A=100	B=50		A=80
NEW	A=100	B=50		A=80	B=70		A=110

cell storage (on disk)

A 110	B 70

commits = {}

```
recover(log):
  commits = {}
  for record r in log[len(log)-1] .. log[0]:
    if r.type == commit:
      commits.add(r.tid)
    if r.type == update and r.tid not in commits:
      cell_write(r.var, r.old_val) // undo
```

TID	T1 UPDATE	T1 UPDATE	T1 COMMIT	T2 UPDATE	T2 UPDATE	T2 COMMIT	T3 UPDATE
OLD	A=0	B=0		A=100	B=50		A=80
NEW	A=100	B=50		A=80	B=70		A=110

cell storage (on disk)

A 80

B 70

commits = {}

```
recover(log):
  commits = {}
  for record r in log[len(log)-1] .. log[0]:
    if r.type == commit:
      commits.add(r.tid)
    if r.type == update and r.tid not in commits:
      cell_write(r.var, r.old_val) // undo
```

TID	T1	T1	T1	T2	T2	T2	T3
	UPDATE	UPDATE	COMMIT	UPDATE	UPDATE	COMMIT	UPDATE
OLD	A=0	B=0		A=100	B=50		A=80
NEW	A=100	B=50		A=80	B=70		A=110

cell storage (on disk)

A 80	B 70

commits = {}

```
recover(log):
  commits = {}
  for record r in log[len(log)-1] .. log[0]:
    if r.type == commit:
      commits.add(r.tid)
    if r.type == update and r.tid not in commits:
      cell_write(r.var, r.old_val) // undo
```

```
    +---------+---------+---------+---------+---------+---------+---------+
TID |   T1    |   T1    |   T1    |   T2    |   T2    |   T2    |   T3    |
    | UPDATE  | UPDATE  | COMMIT  | UPDATE  | UPDATE  | COMMIT  | UPDATE  |
OLD | A=0     | B=0     |         | A=100   | B=50    |         | A=80    |
NEW | A=100   | B=50    |         | A=80    | B=70    |         | A=110   |
    +---------+---------+---------+---------+---------+---------+---------+
```

cell storage (on disk)

A	80

B	70

commits = {T2}

```
recover(log):
  commits = {}
  for record r in log[len(log)-1] .. log[0]:
    if r.type == commit:
      commits.add(r.tid)
    if r.type == update and r.tid not in commits:
      cell_write(r.var, r.old_val) // undo
```

```
    +---------+---------+---------+---------+---------+---------+---------+
TID |  T1     |  T1     |   T1    |  T2     |  T2     |   T2    |  T3     |
    | UPDATE  | UPDATE  | COMMIT  | UPDATE  | UPDATE  | COMMIT  | UPDATE  |
OLD | A=0     | B=0     |         | A=100   | B=50    |         | A=80    |
NEW | A=100   | B=50    |         | A=80    | B=70    |         | A=110   |
    +---------+---------+---------+---------+---------+---------+---------+
```

cell storage (on disk)

A	80

B	70

commits = {T2}

```
recover(log):
  commits = {}
  for record r in log[len(log)-1] .. log[0]:
    if r.type == commit:
      commits.add(r.tid)
    if r.type == update and r.tid not in commits:
      cell_write(r.var, r.old_val) // undo
```

TID	T1 UPDATE	T1 UPDATE	T1 COMMIT	T2 UPDATE	T2 UPDATE	T2 COMMIT	T3 UPDATE
OLD	A=0	B=0		A=100	B=50		A=80
NEW	A=100	B=50		A=80	B=70		A=110

cell storage (on disk)

A 80	B 70

commits = {T2}

```
recover(log):
  commits = {}
  for record r in log[len(log)-1] .. log[0]:
    if r.type == commit:
      commits.add(r.tid)
    if r.type == update and r.tid not in commits:
      cell_write(r.var, r.old_val) // undo
```

TID	T1	T1	T1	T2	T2	T2	T3
	UPDATE	UPDATE	COMMIT	UPDATE	UPDATE	COMMIT	UPDATE
OLD	A=0	B=0		A=100	B=50		A=80
NEW	A=100	B=50		A=80	B=70		A=110

cell storage (on disk)

A 80	B 70

commits = {T2}

```
recover(log):
  commits = {}
  for record r in log[len(log)-1] .. log[0]:
    if r.type == commit:
      commits.add(r.tid)
    if r.type == update and r.tid not in commits:
      cell_write(r.var, r.old_val) // undo
```

TID	T1	T1	T1	T2	T2	T2	T3
	UPDATE	UPDATE	COMMIT	UPDATE	UPDATE	COMMIT	UPDATE
OLD	A=0	B=0		A=100	B=50		A=80
NEW	A=100	B=50		A=80	B=70		A=110

cell storage (on disk)

A 80	B 70

commits = {T2, T1}

```
recover(log):
  commits = {}
  for record r in log[len(log)-1] .. log[0]:
    if r.type == commit:
      commits.add(r.tid)
    if r.type == update and r.tid not in commits:
      cell_write(r.var, r.old_val) // undo
```

```
    +---------+---------+---------+---------+---------+---------+---------+
TID |   T1    |   T1    |   T1    |   T2    |   T2    |   T2    |   T3    |
    | UPDATE  | UPDATE  | COMMIT  | UPDATE  | UPDATE  | COMMIT  | UPDATE  |
OLD | A=0     | B=0     |         | A=100   | B=50    |         | A=80    |
NEW | A=100   | B=50    |         | A=80    | B=70    |         | A=110   |
    +---------+---------+---------+---------+---------+---------+---------+
```

cell storage (on disk)

A 80	B 70

commits = {T2, T1}

```
recover(log):
  commits = {}
  for record r in log[len(log)-1] .. log[0]:
    if r.type == commit:
      commits.add(r.tid)
    if r.type == update and r.tid not in commits:
      cell_write(r.var, r.old_val) // undo
```

TID	T1	T1	T1	T2	T2	T2	T3
	UPDATE	UPDATE	COMMIT	UPDATE	UPDATE	COMMIT	UPDATE
OLD	A=0	B=0		A=100	B=50		A=80
NEW	A=100	B=50		A=80	B=70		A=110

cell storage (on disk)

A	80

B	70

commits = {T2, T1}

```
recover(log):
  commits = {}
  for record r in log[len(log)-1] .. log[0]:
    if r.type == commit:
      commits.add(r.tid)
    if r.type == update and r.tid not in commits:
      cell_write(r.var, r.old_val) // undo
```

```
    +--------+--------+--------+--------+--------+--------+--------+
TID |   T1   |   T1   |   T1   |   T2   |   T2   |   T2   |   T3   |
    | UPDATE | UPDATE | COMMIT | UPDATE | UPDATE | COMMIT | UPDATE |
OLD | A=0    | B=0    |        | A=100  | B=50   |        | A=80   |
NEW | A=100  | B=50   |        | A=80   | B=70   |        | A=110  |
    +--------+--------+--------+--------+--------+--------+--------+
```

cell storage (on disk) A 80 B 70

```
read(var):
  return cell_read(var)

write(var, value):
  log.append(current_tid, update, var, read(var), value)
  cell_write(var, value)
```

```
    +--------+--------+--------+--------+--------+--------+--------+
TID |  T1    |  T1    |   T1   |  T2    |  T2    |   T2   |  T3    |
    | UPDATE | UPDATE | COMMIT | UPDATE | UPDATE | COMMIT | UPDATE |
OLD | A=0    | B=0    |        | A=100  | B=50   |        | A=80   |
NEW | A=100  | B=50   |        | A=80   | B=70   |        | A=110  |
    +--------+--------+--------+--------+--------+--------+--------+
```

cell storage (on disk) | A 110 | B 70 |

performance?

problem: read performance is now great, but writes got (a little bit) slower and recovery got (a lot) slower

```
     +--------+--------+--------+--------+--------+--------+--------+
TID  |   T1   |   T1   |   T1   |   T2   |   T2   |   T2   |   T3   |
     | UPDATE | UPDATE | COMMIT | UPDATE | UPDATE | COMMIT | UPDATE |
OLD  | A=0    | B=0    |        | A=100  | B=50   |        | A=80   |
NEW  | A=100  | B=50   |        | A=80   | B=70   |        | A=110  |
     +--------+--------+--------+--------+--------+--------+--------+
```

cell storage (on disk)	A 110	B 70	cache (memory)	A 110	B 70

```
read(var):
  if var in cache:
    return cache[var]
  else:
    // may evict others from cache to cell storage
    cache[var] = cell_read(var)
    return cache[var]

write(var, value):
  log.append(current_tid, update, var, read(var), value)
  cache[var] = value

flush(): // called "occasionally"
  cell write(var, cache[var]) for each var
```

```
    +--------+--------+--------+--------+--------+--------+--------+
TID |   T1   |   T1   |   T1   |   T2   |   T2   |   T2   |   T3   |
    | UPDATE | UPDATE | COMMIT | UPDATE | UPDATE | COMMIT | UPDATE |
OLD | A=0    | B=0    |        | A=100  | B=50   |        | A=80   |
NEW | A=100  | B=50   |        | A=80   | B=70   |        | A=110  |
    +--------+--------+--------+--------+--------+--------+--------+
```

```
cell storage   +-------+  +-------+     cache    .......  .......
 (on disk)     | A 100 |  | B  50 |   (memory)   :A 110:  :B  70:
               +-------+  +-------+              .......  .......
```

suppose we flushed the cache after T1 committed,
but have not flushed it since then

```
    +--------+--------+--------+--------+--------+--------+--------+
TID |   T1   |   T1   |   T1   |   T2   |   T2   |   T2   |   T3   |
    | UPDATE | UPDATE | COMMIT | UPDATE | UPDATE | COMMIT | UPDATE |
OLD | A=0    | B=0    |        | A=100  | B=50   |        | A=80   |
NEW | A=100  | B=50   |        | A=80   | B=70   |        | A=110  |
    +--------+--------+--------+--------+--------+--------+--------+
```

cell storage (on disk) A 100 B 50 **cache (memory)**

```
recover(log):
  commits = {}
  for record r in log[len(log)-1] .. log[0]:
    if r.type == commit:
      commits.add(r.tid)
    if r.type == update and r.tid not in commits:
      cell_write(r.var, r.old_val) // undo
```

```
    +--------+--------+--------+--------+--------+--------+--------+
TID |   T1   |   T1   |   T1   |   T2   |   T2   |   T2   |   T3   |
    | UPDATE | UPDATE | COMMIT | UPDATE | UPDATE | COMMIT | UPDATE |
OLD | A=0    | B=0    |        | A=100  | B=50   |        | A=80   |
NEW | A=100  | B=50   |        | A=80   | B=70   |        | A=110  |
    +--------+--------+--------+--------+--------+--------+--------+
```

cell storage (on disk) | A 80 | B 50 | **cache (memory)**

```
recover(log):
  commits = {}
  for record r in log[len(log)-1] .. log[0]:
    if r.type == commit:
      commits.add(r.tid)
    if r.type == update and r.tid not in commits:
      cell_write(r.var, r.old_val) // undo
```

TID	T1	T1	T1	T2	T2	T2	T3
	UPDATE	UPDATE	COMMIT	UPDATE	UPDATE	COMMIT	UPDATE
OLD	A=0	B=0		A=100	B=50		A=80
NEW	A=100	B=50		A=80	B=70		A=110

cell storage (on disk) | A 80 | B 50

cache (memory)

```
recover(log):
  commits = {}
  for record r in log[len(log)-1] .. log[0]:
    if r.type == commit:
      commits.add(r.tid)
    if r.type == update and r.tid not in commits:
      cell_write(r.var, r.old_val) // undo
```

```
     +--------+--------+--------+--------+--------+--------+--------+
TID  |  T1    |  T1    |  T1    |  T2    |  T2    |  T2    |  T3    |
     | UPDATE | UPDATE | COMMIT | UPDATE | UPDATE | COMMIT | UPDATE |
OLD  | A=0    | B=0    |        | A=100  | B=50   |        | A=80   |
NEW  | A=100  | B=50   |        | A=80   | B=70   |        | A=110  |
     +--------+--------+--------+--------+--------+--------+--------+
```

cell storage (on disk) | A 80 | B 50 | **cache (memory)**

```
recover(log):
  commits = {}
  for record r in log[len(log)-1] .. log[0]:
    if r.type == commit:
      commits.add(r.tid)
    if r.type == update and r.tid not in commits:
      cell_write(r.var, r.old_val) // undo
```

all other updates were committed; **B**'s value won't ever be changed

TID	T1	T1	T1	T2	T2	T2	T3
	UPDATE	UPDATE	COMMIT	UPDATE	UPDATE	COMMIT	UPDATE
OLD	A=0	B=0		A=100	B=50		A=80
NEW	A=100	B=50		A=80	B=70		A=110

cell storage (on disk) | **A 80** | **B 50** | **cache (memory)**

```
recover(log):
  commits = {}
  for record r in log[len(log)-1] .. log[0]:
    if r.type == commit:
      commits.add(r.tid)
    if r.type == update and r.tid not in commits:
      cell_write(r.var, r.old_val) // undo
  for record r in log[0] .. log[len(log)-1]:
    if r.type == update and r.tid in commits:
      cell_write(r.var, r.new_value) // redo
```

```
    +--------+--------+--------+--------+--------+--------+--------+
TID |   T1   |   T1   |   T1   |   T2   |   T2   |   T2   |   T3   |
    | UPDATE | UPDATE | COMMIT | UPDATE | UPDATE | COMMIT | UPDATE |
OLD | A=0    | B=0    |        | A=100  | B=50   |        | A=80   |
NEW | A=100  | B=50   |        | A=80   | B=70   |        | A=110  |
    +--------+--------+--------+--------+--------+--------+--------+
```

cell storage (on disk) | **A 80** | **B 70** | **cache (memory)**

```
recover(log):
  commits = {}
  for record r in log[len(log)-1] .. log[0]:
    if r.type == commit:
      commits.add(r.tid)
    if r.type == update and r.tid not in commits:
      cell_write(r.var, r.old_val) // undo
  for record r in log[0] .. log[len(log)-1]:
    if r.type == update and r.tid in commits:
      cell_write(r.var, r.new_value) // redo
```

```
    +--------+--------+--------+--------+--------+--------+--------+
TID |  T1    |  T1    |   T1   |  T2    |  T2    |   T2   |  T3    |
    | UPDATE | UPDATE | COMMIT | UPDATE | UPDATE | COMMIT | UPDATE |
OLD | A=0    | B=0    |        | A=100  | B=50   |        | A=80   |
NEW | A=100  | B=50   |        | A=80   | B=70   |        | A=110  |
    +--------+--------+--------+--------+--------+--------+--------+
```

cell storage (on disk) $\boxed{\texttt{A 80}}$ $\boxed{\texttt{B 70}}$ **cache (memory)**

performance?

problem: recovery is still slow

```
    +--------+--------+--------+--------+--------+--------+--------+
TID |   T1   |   T1   |   T1   |   T2   |   T2   |   T2   |   T3   |
    | UPDATE | UPDATE | COMMIT | UPDATE | UPDATE | COMMIT | UPDATE |
OLD | A=0    | B=0    |        | A=100  | B=50   |        | A=80   |
NEW | A=100  | B=50   |        | A=80   | B=70   |        | A=110  |
    +--------+--------+--------+--------+--------+--------+--------+
```

```
cell storage   +------+  +------+    cache
 (on disk)     | A 80 |  | B 70 |  (memory)
               +------+  +------+
```

performance?

solution: write checkpoints and truncate the log

- **(Write-ahead) logs** provide **atomicity** with better performance than shadow copies. The primary benefit is making small appends for each update, rather than copying and entire file over for every change.

- **Cell storage** is used with the log to improve read-performance, and **caches** and **truncation** can be used to improve write- and recovery-performance.

6.033 Spring 2018

Lecture #17

- **Isolation**
  - **Conflict serializability**
  - **Conflict graphs**
  - **Two-phase locking**

goal: build reliable systems from unreliable components

the abstraction that makes that easier is

transactions, which provide **atomicity** and **isolation**, while not hindering **performance**

atomicity → **shadow copies** (simple, poor performance) or **logs** (better performance, a bit more complex)

isolation → **?**

eventually, we also want transaction-based systems to be **distributed**: to run across multiple machines

goal: build reliable systems from unreliable components

the abstraction that makes that easier is

transactions, which provide **atomicity** and **isolation**, while not hindering **performance**

atomicity → **shadow copies** (simple, poor performance) or **logs** (better performance, a bit more complex)

isolation → **two-phase locking**

eventually, we also want transaction-based systems to be **distributed**: to run across multiple machines

goal: run transactions `T1`, `T2`, .., `TN` concurrently, and have it "appear" as if they ran sequentially

```
T1
begin
read(x)
tmp = read(y)
write(y, tmp+10)
commit
```

```
T2
begin
write(x, 20)
write(y, 30)
commit
```

naive approach: actually run them sequentially, via (perhaps) a single global lock

goal: run transactions `T1`, `T2`, .., `TN` concurrently, and have it "appear" as if they ran sequentially

what does this even mean?

```
T1
begin
read(x)
tmp = read(y)
write(y, tmp+10)
commit
```

```
T2
begin
write(x, 20)
write(y, 30)
commit
```

```
T1                        T2
begin                     begin
read(x)                   write(x, 20)
tmp = read(y)             write(y, 30)
write(y, tmp+10)          commit
commit
```

possible sequential schedules

```
T1 -> T2: x=20, y=30
T2 -> T1: x=20, y=40
```

```
T2: write(x, 20)
T1: read(x)
T2: write(y, 30)
T1: tmp = read(y)
T1: write(y, tmp+10)
```

at end:

x=20, y=40

```
T1: read(x)
T2: write(x, 20)
T1: tmp = read(y)
T2: write(y, 30)
T1: write(y, tmp+10)
```

at end:

x=20, y=10

(assume x, y initialized to zero)

T1

begin

```
read(x)
tmp = read(y)
write(y, tmp+10)
```

commit

T2

begin

```
write(x, 20)
write(y, 30)
```

commit

possible sequential schedules

T1 -> T2: x=20, y=30

T2 -> T1: x=20, y=40

```
T2: write(x, 20)
T1: read(x)
T2: write(y, 30)
T1: tmp = read(y)
T1: write(y, tmp+10)
```

at end:

x=20, y=40

```
T1: read(x)
T2: write(x, 20)
T2: write(y, 30)
T1: tmp = read(y)
T1: write(y, tmp+10)
```

at end:

x=20, y=40

```
T1
begin
read(x)
tmp = read(y)
write(y, tmp+10)
commit
```

```
T2
begin
write(x, 20)
write(y, 30)
commit
```

possible sequential schedules

T1 -> T2: x=20, y=30
T2 -> T1: x=20, y=40

```
T2: write(x, 20)
T1: read(x)
T2: write(y, 30)
T1: tmp = read(y)
T1: write(y, tmp+10)
```

at end:
x=20, y=40

```
T1: read(x) // x=0
T2: write(x, 20)
T2: write(y, 30)
T1: tmp = read(y) // y=30
T1: write(y, tmp+10)
```

at end:
x=20, y=40

In the second schedule, T1 reads x=0 *and* y=30; those two reads together aren't possible in a sequential schedule.
is that okay?

it depends.

there are many ways for multiple transactions to "appear" to have been run in sequence; we say there are different notions of **serializability**. what type of serializability you want depends on what your application needs.

conflicts

two operations conflict if they operate on the same object and at least one of them is a write.

```
T1
begin
T1.1 read(x)
T1.2 tmp = read(y)
T1.3 write(y, tmp+10)
commit
```

```
T2
begin
T2.1 write(x, 20)
T2.2 write(y, 30)
commit
```

conflicts

`T1.1 read(x)` and `T2.1 write(x, 20)`

`T1.2 tmp = read(y)` and `T2.2 write(y, 30)`

`T1.3 write(y, tmp+10)` and `T2.2 write(y, 30)`

conflicts

two operations conflict if they operate on the same object and at least one of them is a write.

in any schedule, two conflicting operations A and B will have an order: either A is executed before B, or B is executed before A. we'll call this the **order** of the conflict (in that schedule).

```
T1
begin
T1.1 read(x)
T1.2 tmp = read(y)
T1.3 write(y, tmp+10)
commit
```

```
T2
begin
T2.1 write(x, 20)
T2.2 write(y, 30)
commit
```

---

conflicts

`T1.1 read(x)` and `T2.1 write(x, 20)`

`T1.2 tmp = read(y)` and `T2.2 write(y, 30)`

`T1.3 write(y, tmp+10)` and `T2.2 write(y, 30)`

```
T1
begin
T1.1 read(x)
T1.2 tmp = read(y)
T1.3 write(y, tmp+10)
commit
```

```
T2
begin
T2.1 write(x, 20)
T2.2 write(y, 30)
commit
```

conflicts

```
T1.1 read(x)           -> T2.1 write(x, 20)
T1.2 tmp = read(y)     -> T2.2 write(y, 30)
T1.3 write(y, tmp+10)  -> T2.2 write(y, 30)
```

if we execute **T1** before **T2**, within any conflict, **T1**'s operation will occur first

```
T1
begin
T1.1 read(x)
T1.2 tmp = read(y)
T1.3 write(y, tmp+10)
commit
```

```
T2
begin
T2.1 write(x, 20)
T2.2 write(y, 30)
commit
```

---

conflicts

```
        T1.1 read(x)        <-   T2.1 write(x, 20)
   T1.2 tmp = read(y)       <-   T2.2 write(y, 30)
T1.3 write(y,               <-   T2.2 write(y, 30)
tmp+10)
```

if we execute **T2** before **T1**, within any conflict, **T2**'s operation will occur first

conflicts

two operations conflict if they operate on the same object and at least one of them is a write.

conflict serializability

a schedule is **conflict serializable** if the order of all of its conflicts is the same as the order of the conflicts in some sequential schedule.

conflicts

T1.1, T2.1
T1.2, T2.2
T1.3, T2.2

a schedule is **conflict serializable** if the order of all of its conflicts is the same as the order of the conflicts in some sequential schedule.

(here, that means we will see one transaction's — T1's or T2's — operation occurring first in each conflict)

```
T2.1: write(x, 20)
T1.1: read(x)
T2.2: write(y, 30)
T1.2: tmp = read(y)
T1.3: write(y, tmp+10)
```

```
T2.1 -> T1.1
T2.2 -> T1.2
T2.2 -> T1.3
```

```
T1.1: read(x)
T2.1: write(x, 20)
T2.2: write(y, 30)
T1.2: tmp = read(y)
T1.3: write(y, tmp+10)
```

```
T1.1 -> T2.1
T2.2 -> T1.2
T2.2 -> T1.3
```

conflict graph

edge from **Ti** to **Tj** iff **Ti** and **Tj** have a conflict between them and the first step in the conflict occurs in **Ti**

```
T2: write(x, 20)
T1: read(x)
T2: write(y, 30)
T1: tmp = read(y)
T1: write(y, tmp+10)
```

```
T2.1 -> T1.1
T2.2 -> T1.2
T2.2 -> T1.3
```

```
T1: read(x)
T2: write(x, 20)
T2: write(y, 30)
T1: tmp = read(y)
T1: write(y, tmp+10)
```

```
T1.1 -> T2.1
T2.2 -> T1.2
T2.2 -> T1.3
```

conflict graph

edge from T_i to T_j iff T_i and T_j have a conflict between them and the first step in the conflict occurs in T_i

```
T2: write(x, 20)
T1: read(x)
T2: write(y, 30)
T1: tmp = read(y)
T1: write(y, tmp+10)
```

T2 → T1

```
T1: read(x)
T2: write(x, 20)
T2: write(y, 30)
T1: tmp = read(y)
T1: write(y, tmp+10)
```

T2 ⇄ T1

a schedule is conflict serializable iff it has an acyclic conflict graph

problem: how do we generate schedules that are conflict serializable? generate all possible schedules and check their conflict graphs?

solution: two-phase locking (2PL)

1. each shared variable has a lock

2. before **any** operation on a variable, the transaction must acquire the corresponding lock

3. after a transaction releases a lock, it may **not** acquire any other locks

we will usually release locks after commit or abort, which is technically ***strict*** two-phase locking

2PL produces a conflict-serializable schedule

(equivalently, 2PL produces a conflict graph without a cycle)

proof: suppose not. then a cycle exists in the conflict graph

$$\begin{tikzcd} T_1 \arrow[r, "x_1"] & T_2 \arrow[r, "x_2"] & T_3 \arrow[r, "x_3"] & \dots \arrow[r, "x_{k-1}"] & T_k \arrow[llll, bend left, "x_k"] \end{tikzcd}$$

to cause the conflict, each pair of conflicting **transactions** must have some **shared variable** that they conflict on

T_1 acquires x_1.lock
T_2 acquires x_1.lock

T_2 acquires x_2.lock
T_3 acquires x_2.lock

...

T_k acquires x_k.lock
T_1 acquires x_k.lock

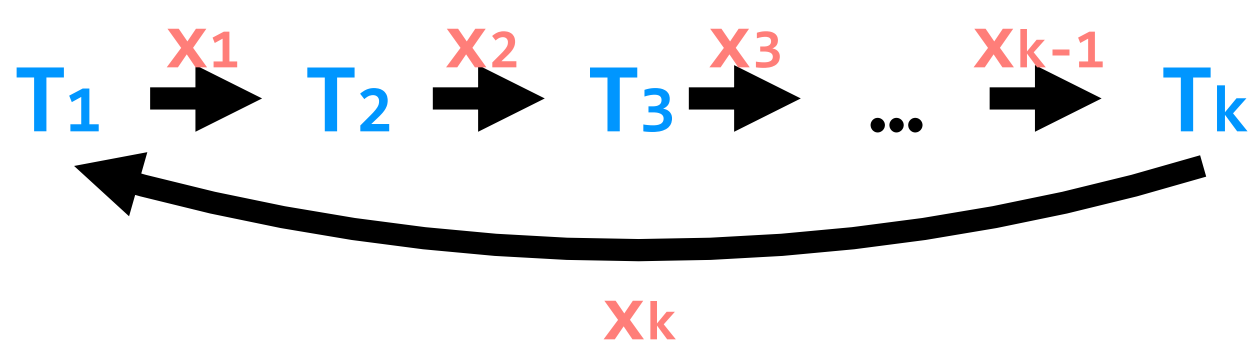
in the schedule, each pair of **transactions** needs to acquire a lock on their **shared variable**

in order for the schedule to progress, T_1 must have released its lock on x_1 before T_2 acquired it

2PL produces a conflict-serializable schedule

(equivalently, 2PL produces a conflict graph without a cycle)

proof: suppose not. then a cycle exists in the conflict graph



to cause the conflict, each pair of conflicting **transactions** must have some **shared variable** that they conflict on

T_1 **acquires** x_1**.lock**
T_1 **releases** x_1**.lock**
T_2 **acquires** x_1**.lock**

T_2 **acquires** x_2**.lock**
T_3 **acquires** x_2**.lock**

...

T_k **acquires** x_k**.lock**
T_1 **acquires** x_k**.lock**

in the schedule, each pair of **transactions** needs to acquire a lock on their **shared variable**

in order for the schedule to progress, T_1 must have released its lock on x_1 before T_2 acquired it

contradiction: this is not a valid 2PL schedule

T1	**T2**
acquire(x.lock)	acquire(y.lock)
read(x)	read(y)
acquire(y.lock)	acquire(x.lock)
read(y)	read(x)
release(y.lock)	release(x.lock)
release(x.lock)	release(y.lock)

problem: 2PL can result in deadlock

T1	**T2**
`acquire(x.lock)`	`acquire(y.lock)`
`read(x)`	`read(y)`
`acquire(y.lock)`	`acquire(x.lock)`
`read(y)`	`read(x)`
`release(y.lock)`	`release(x.lock)`
`release(x.lock)`	`release(y.lock)`

"solution": global ordering on locks

T1	**T2**
acquire(x.lock)	acquire(y.lock)
read(x)	read(y)
acquire(y.lock)	acquire(x.lock)
read(y)	read(x)
release(y.lock)	release(x.lock)
release(x.lock)	release(y.lock)

better solution: take advantage of atomicity and abort one of the transactions!

performance improvement: allow concurrent reads with reader- and writer-locks

T1
```
acquire(x.reader_lock)
read(x)
acquire(y.writer_lock)
write(y)
release(y.writer_lock)
release(x.reader_lock)
```

T2
```
acquire(x.reader_lock)
read(x)
acquire(y.writer_lock)
write(y)
release(y.writer_lock)
release(x.reader_lock)
```

multiple transactions can hold reader locks for the same variable at once. a transaction can only hold a writer lock for a variable if there are *no* other locks held for that variable

- Different types of **serializability** allow us to specify precisely what we want when we run transactions in parallel. **Conflict-serializability** is common in practice.

- **Two-phase locking** allows us to generate conflict serializable schedules. We can improve its performance by allowing concurrent reads via reader- and writer-locks.