cse138-notes

Andrew Zhu

Sep 01, 2022

CONTENTS:

1 Introduction **1**
- 1.1 Failures . . . 1
- 1.2 Timeouts . . . 2
- 1.3 Why? . . . 2

2 Time **3**
- 2.1 Logical Clocks . . . 4
- 2.2 Lamport Diagrams . . . 4
- 2.3 Network Models . . . 6
- 2.4 State and Events . . . 6
- 2.5 Partial Order . . . 7
- 2.6 Lamport Clocks . . . 8
- 2.7 Vector Clocks . . . 9

3 Protocol **13**
- 3.1 FIFO Delivery . . . 14
- 3.2 Causal Delivery . . . 15
- 3.3 Totally Ordered Delivery . . . 17

4 Snapshots of Distributed Systems **19**
- 4.1 Chandy-Lamport Algorithm . . . 20

5 Safety & Liveness **25**
- 5.1 Fault Models . . . 26
- 5.2 Two Generals Problem . . . 27
- 5.3 Fault Tolerance . . . 28
- 5.4 Reliable Delivery, Take 2 . . . 28
- 5.5 Reliable Broadcast . . . 29

6 Replication **33**
- 6.1 Primary-Backup Replication . . . 33
- 6.2 Chain Replication . . . 34
- 6.3 Total Order v. Determinism . . . 35
- 6.4 Bad Things . . . 36
- 6.5 Consistency . . . 38
- 6.6 Coordination . . . 38
- 6.7 Active v. Passive Replication . . . 39

7 Consensus **41**
- 7.1 Properties . . . 41
- 7.2 Paxos . . . 42

7.3 Multi-Paxos . . . 48
7.4 Paxos: Fault Tolerance . . . 49
7.5 Other Consensus Protocols . . . 49

8 Consistency **51**
8.1 Eventual Consistency . . . 51
8.2 Misc . . . 53
8.3 Quorum Consistency . . . 55

9 Dynamo **57**
9.1 Concepts . . . 57
9.2 Disagreements . . . 57
9.3 Merkle Trees . . . 58
9.4 Tail Latency . . . 58
9.5 Sharding . . . 59
9.6 Consistent Hashing . . . 63

10 Heterogeneous Distributed Systems **67**
10.1 First-Order Distributed Systems . . . 67
10.2 Heterogeneous . . . 68
10.3 Monolith . . . 68
10.4 Problems . . . 69
10.5 Iteration 2 . . . 70
10.6 Business . . . 71
10.7 Ideals + Rules of Thumb . . . 71
10.8 Conclusion . . . 72

11 MapReduce **73**
11.1 Online v. Offline Systems . . . 73
11.2 MapReduce . . . 74
11.3 Word Count . . . 76
11.4 What Could Go Wrong . . . 77
11.5 MapReduce @ Google . . . 77

12 Math **79**
12.1 The Cost of Consensus . . . 79
12.2 Strong Convergence . . . 79
12.3 Upper Bounds . . . 80
12.4 Examples . . . 80

13 Acknowledgement **83**

14 Indices and tables **85**

Index **87**

CHAPTER

ONE

INTRODUCTION

> What is a distributed system?

`distributed system`

A system running on several nodes connected by a network, characterized by partial failure.

1.1 Failures

There are two philosophies when dealing with failure:

- **High-Performance Computing (HPC) philosophy**
  - treat partial failure as total failure
  - checkpointing
- **"cloud computing" philosophy**
  - treat partial failure as expected
  - work around the failure
  - "everything fails, all the time"

Consider a simple system of two machines, M1 and M2. Suppose M1 asks M2 for the value of some variable *x*.

```
M1 -- x? --> M2
   <-- 5 ---
```

Some potential failures are:

- request from M1 is lost
- request from M1 is *slow*
- M2 crashes
- M2 is slow to respond
- response from M2 is slow or lost
- Byzantine faults - corrupt transmission/malicious response/etc

Important: If you send a request to another node and don't receive a response, it is impossible to know why.

1.2 Timeouts

Real systems try to cope with this using *timeouts*: after a certain amount of time, assume failure and try again. However, this isn't a perfect system - for example, consider if M1 told M2 to increment x:

```
M1 -- x++ --> M2 (x=5)
              x++
M1  X<-- 6 -- x=6
```

If we knew the max delay d and processing time r, we could set the timeout to $2d + r$ and rule out a lot of uncertainty. However, you can't get an accurate value for d and r in real life usually, just statistics. Therefore, distributed systems have to account for both *partial failure* and *unbounded latency*.

1.3 Why?

So why do we make systems distributed and deal with all of this?

- make things faster
- more data than can fit on one machine
- reliability (more copies of data)
- throughput (data is physically closer)

CHAPTER

TWO

TIME

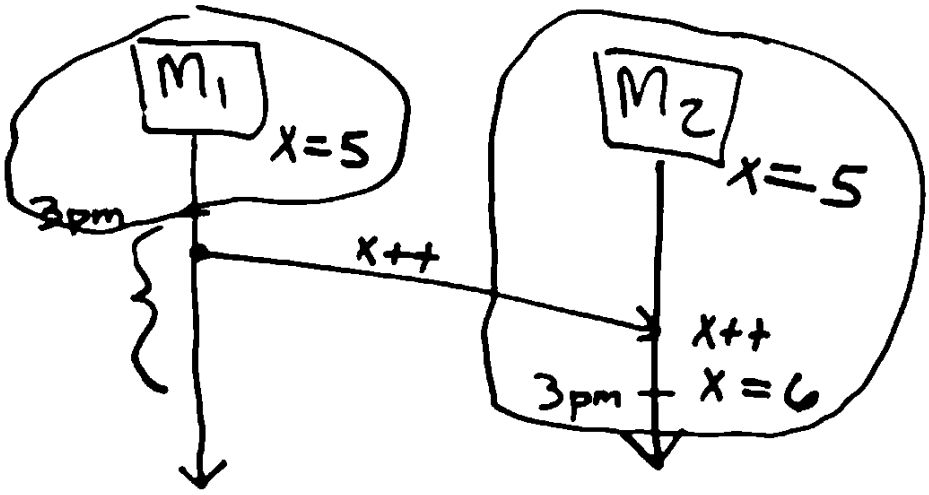
What do we use clocks for?

- **"scheduling": marking points in time**
  - "this class starts at 3:20pm PDT"
  - "this item in cache expires on May 21 at 8pm"
  - "this error message in errors.txt has a timestamp of X"
- **durations/intervals**
  - this class is 95 minutes long
  - this request times out in 10 seconds

Computers have 2 kinds of *physical* clocks:

- **time of day clock**
  - tells you what time of day it is
  - can be synchronized between machines (e.g. NTP)
  - can jump forward/backward (e.g. moving between time zones, DST, leap seconds)
  - not so great for intervals/durations
  - *ok* for points in time, but only ok
- **monotonic clock**
  - only goes forward
  - not comparable between machines (e.g. time since boot)
  - cannot mark specific points in time

Consider the following scenario, where two computers take snapshots of their state at a given time:



2.1 Logical Clocks

Logical clocks only measure the *order of events*.

e.g. `A -> B`: "A happened before B"

This tells us some things about causality:

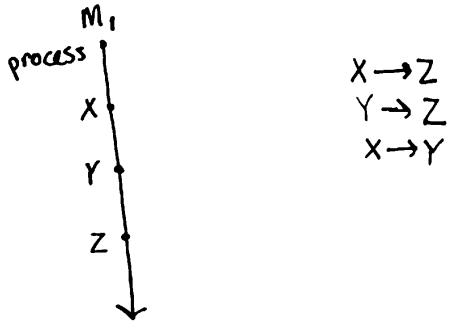
- A *might* have caused B
- B *could not* have caused A

2.2 Lamport Diagrams

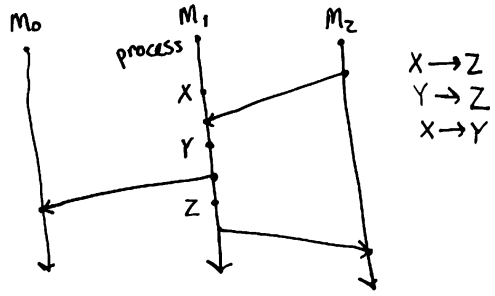
aka spacetime diagrams

Each process is represented by a line. It has a discrete beginning and goes on forever.

Events are represented by dots on the line.



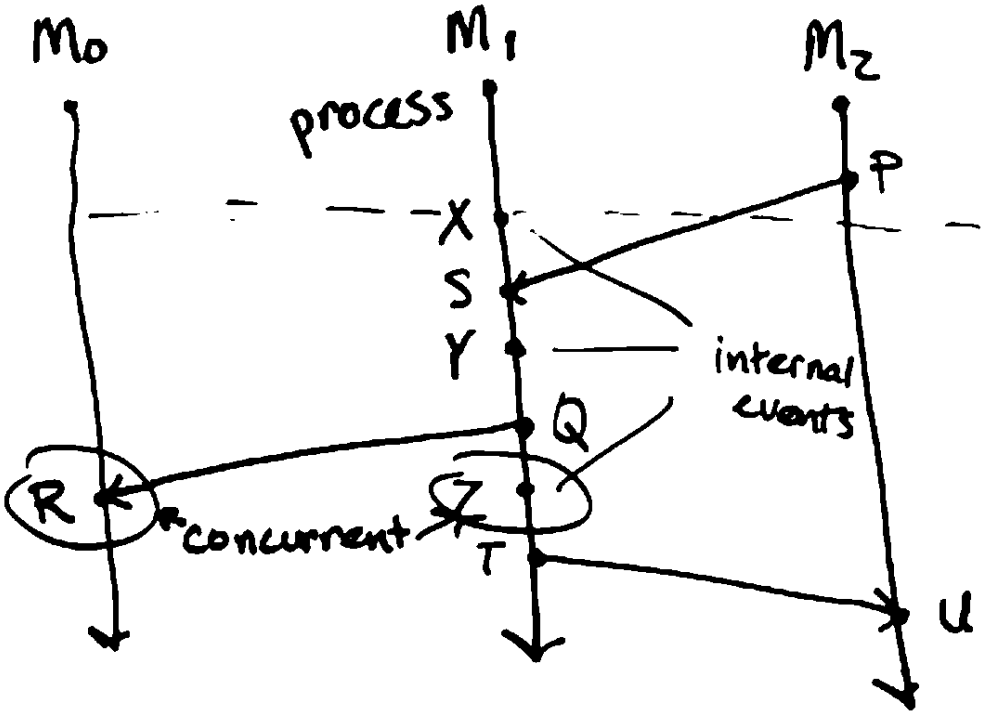
You can represent systems of machines and messages with lines and arrows:



Given events *A* and *B*, we say $A \rightarrow B$ if:

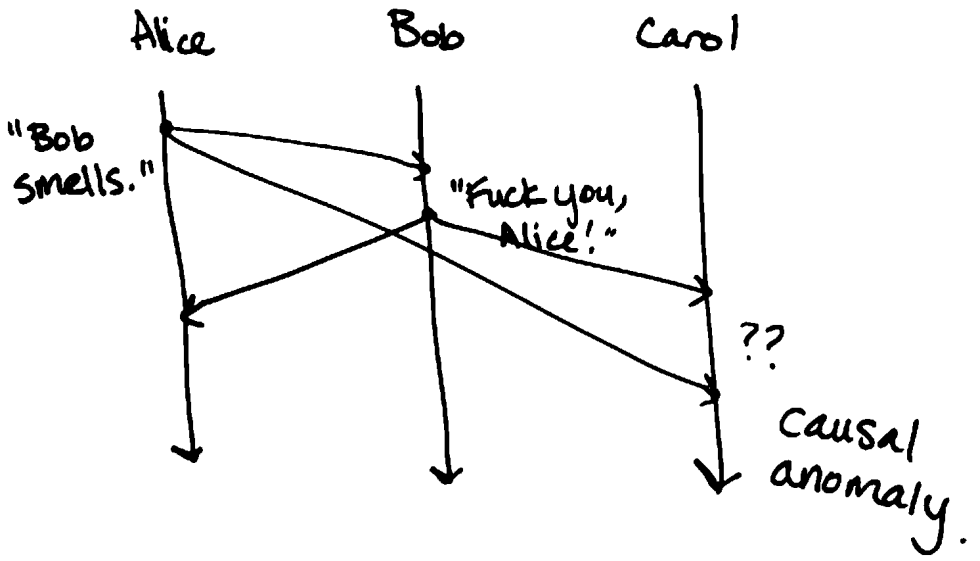
- A and B occur on the same process with A before B
- A is a send event and B is the corresponding receive event
- $A \rightarrow C$ and $C \rightarrow B$

If we can't be sure that given a pair of events, one happens before the other, they are *concurrent*:



$R||T$

We can use logical clocks to counteract causal anomalies like this, caused by unbounded latency:



2.3 Network Models

synchronous network

a network where there exists an n such that no message takes longer than n units of time to be delivered.

We won't talk about this type of network.

asynchronous network

a network with unbounded latency, i.e. there does not exist such n such that no message takes longer than n units of time to be delivered.

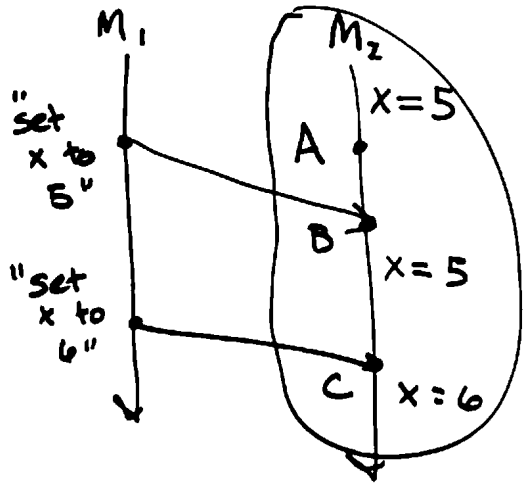
2.4 State and Events

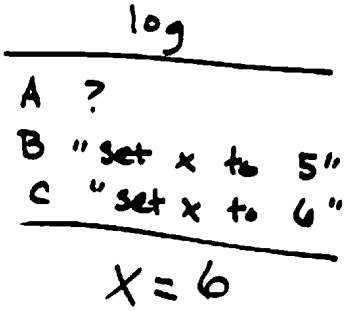
state

something that a given machine knows - e.g. contents of memory/disk

represented by a dot on a Lamport diagram

We can determine the current state by looking at the sequence of events leading up to it:





However, we can't do the opposite.

2.5 Partial Order

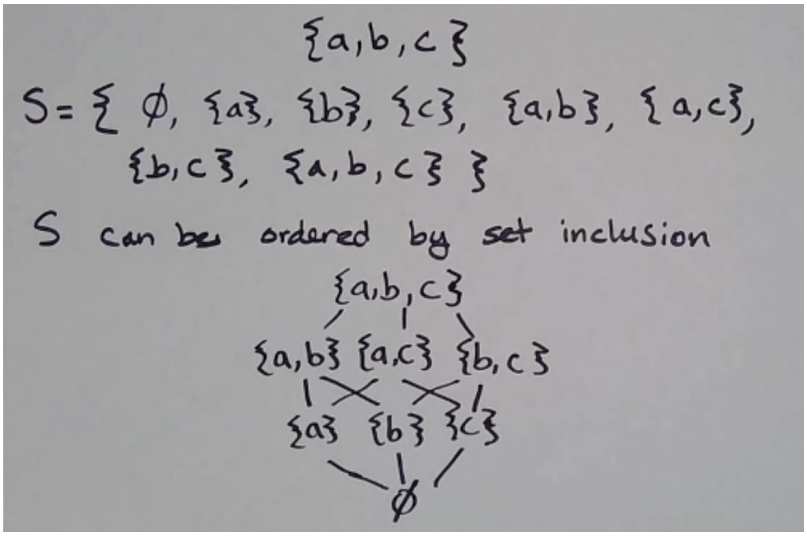
Let's take a brief detour to talk about partial orders.

Partial orders are:

- a set S
- **a binary relation, often written , that lets you compare elements of S, and has the following properties:**
  - reflexive: $\forall a \in S, a \leq a$
  - antisymmetry: $\forall a, b \in S, a \leq b \wedge b \leq a \implies a = b$
  - transitivity: $\forall a, b, c \in S, a \leq b \wedge b \leq c \implies a \leq c$

The "happens-before" relation is *not* a partial order! Considering a set of events, transivity holds and antisymmetry holds (vacuously), but events are not reflexive (it is, however, an *irreflexive partial order*)!

Note: An actual partial order is set containment:



Note that in a partial order, some elements of S may not be comparable (in the example above, {A} and {B} are not related). If all elements in S are comparable, it's called a *total order* (e.g. natural numbers).

2.6 Lamport Clocks

a type of logical clock

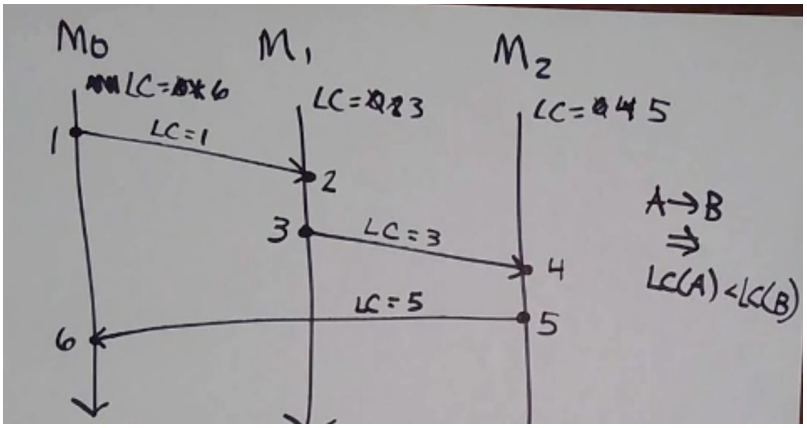
$LC(A)$ - the Lamport clock of event A.

clock condition

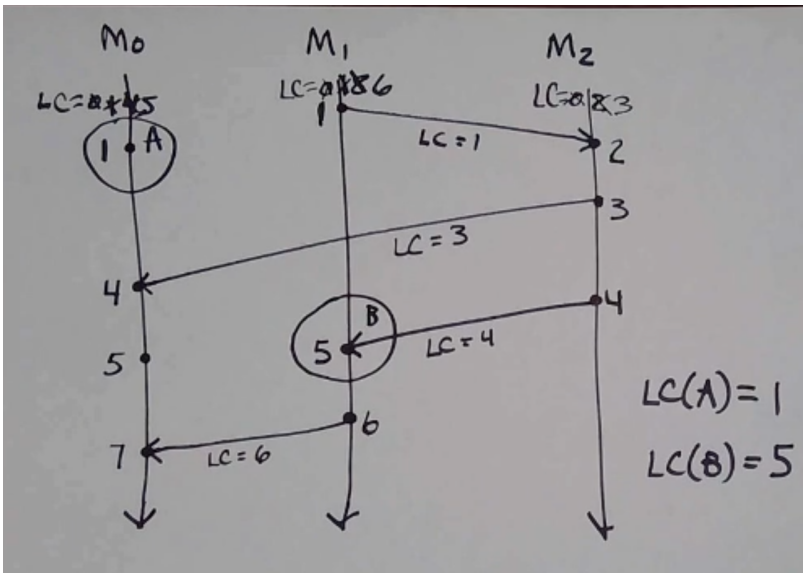
if $A \rightarrow B$, then $LC(A) < LC(B)$.

How do we assign Lamport clocks to events?

1. every process keeps a counter initialized to 0
2. on every event on a process, increment process' counter by 1
3. when you send a message, include the current counter with the message
4. when you receive a message, set counter to max(local, recv) + 1



Important: The converse is not necessarily true (i.e. $LC(A) < LC(B) \neg \implies A \rightarrow B$).



Even though $LC(A) < LC(B)$, there is no path from *A* to *B* - so there is no guarantee that $A \rightarrow B$.

Specifically, while Lamport clocks are *consistent* with causality, they do not *characterize* causaility.

We can use it for it's contrapositive, though! ($\neg LC(A) < LC(B) \implies \neg A \rightarrow B$ - either $B \rightarrow A$ or $A||B$)

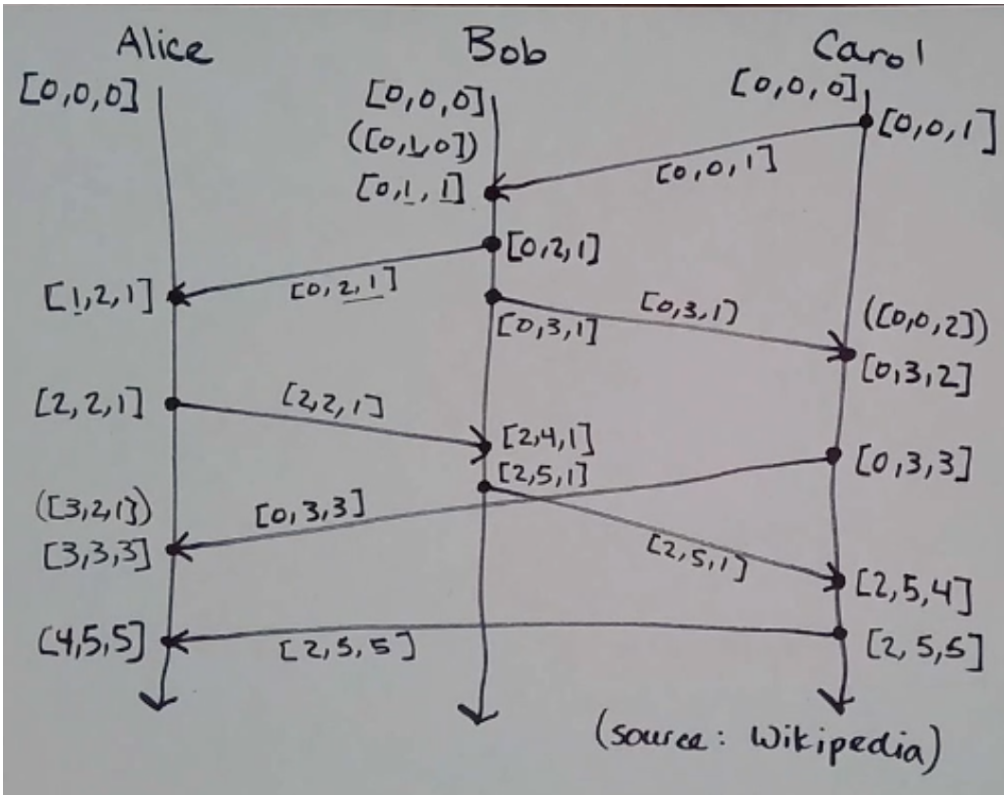
2.7 Vector Clocks

While Lamport clocks don't character causaility, vector clocks do!

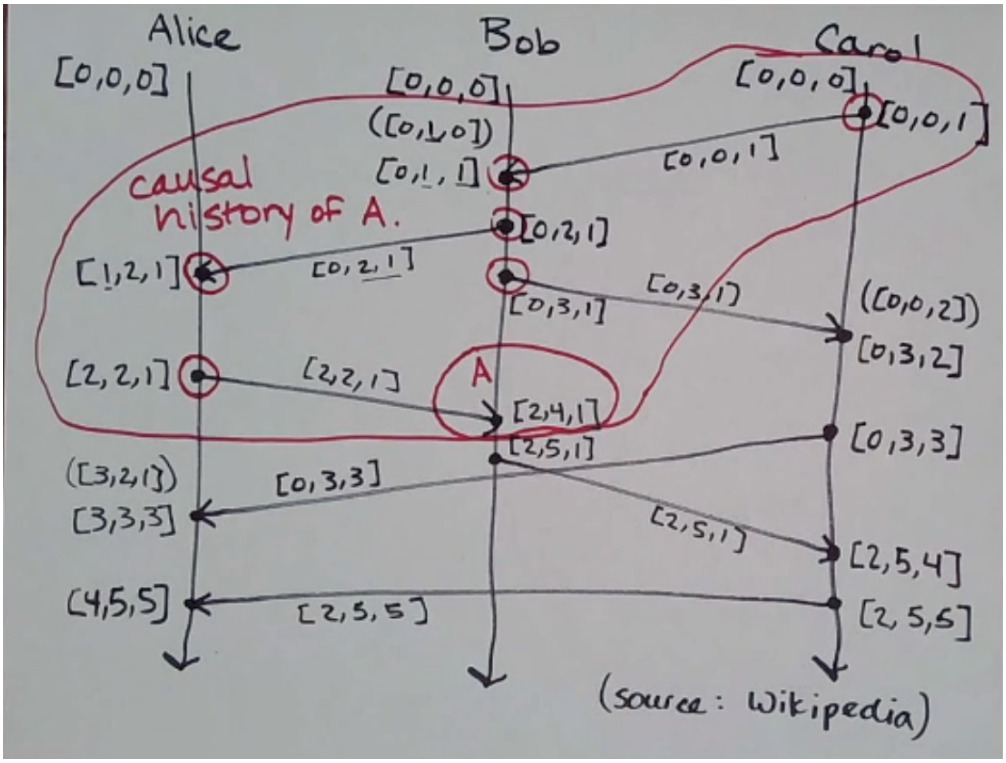
$A \rightarrow B \iff VC(A) < VC(B)$

1. Every process keeps a vector of integers initialized to 0 (size = # of processes)
2. On every event (including receive), a process increments its own position in its vector
3. When sending a message, a process includes its current vector clock
4. When receiving a message, a process updates its vector clock to max(local, recv) (element-wise) and increments its position

Note: max() is element-wise: e.g. max([1, 12, 4], [7, 0, 2]) = [7, 12, 4]



Now, say we wanted to find all events that happened before some event *A* (its causal history):



Notice that all vector clocks of events in A's causal history are less than or equal to A's VC! (Similarly, all events that happen after A have VCs greater than or equal to A's VC.)

Any events that do not satisfy either of these are concurrent with/causally independent of A. (In the example above, some examples are `[0, 3, 3]` and `[3, 3, 3]`).

CHAPTER

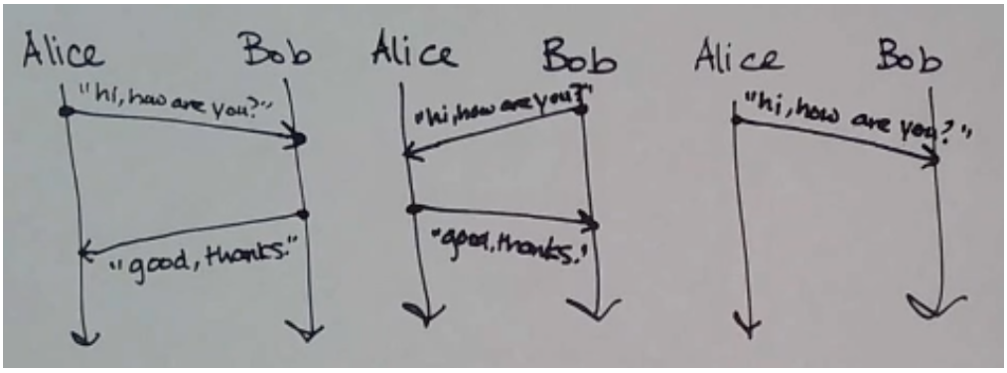
THREE

PROTOCOL

protocol

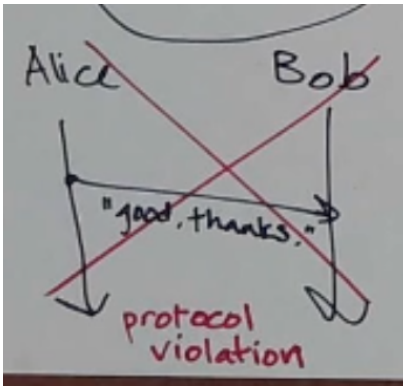
A set of rules that processes use to communicate with each other.

e.g. "At any point a machine can wake up and send 'hi, how are you?', and the receiver must reply with 'good, thanks'."

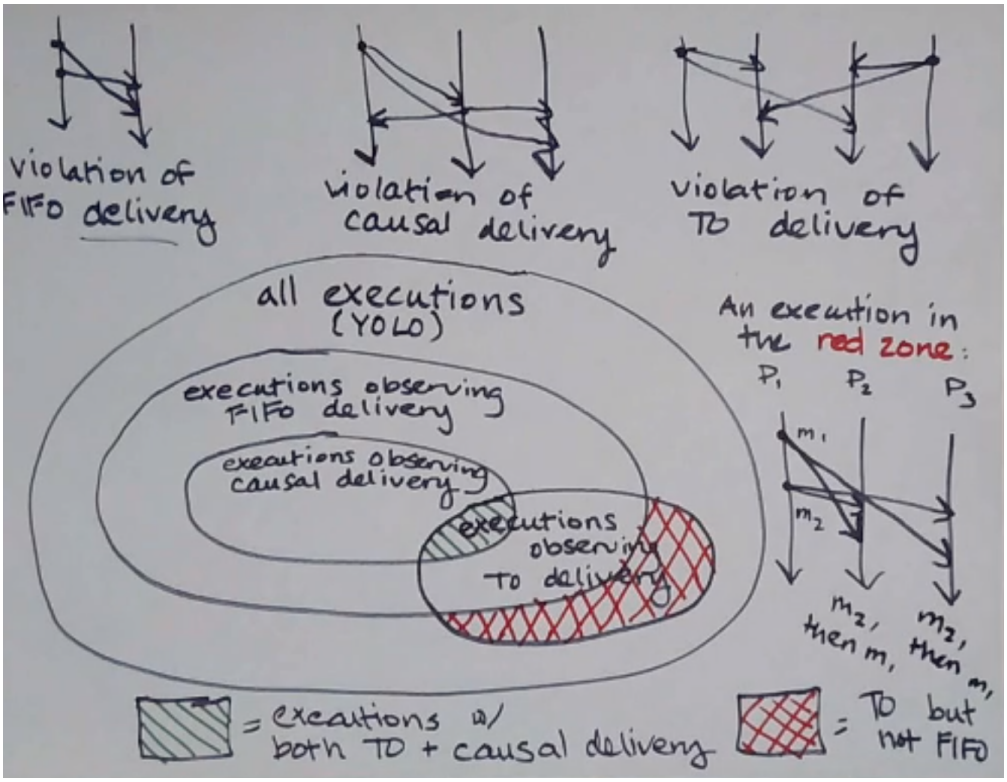


These are all Lamport diagrams of valid runs of the protocol - even though the third doesn't have a response yet, we may have just caught it before it has had a chance to reply.

There are in fact *infinite* diagrams representing a valid run of the protocol! We can also draw diagrams representing a violation of the protocol:



The following are three different correctness properties of executions:

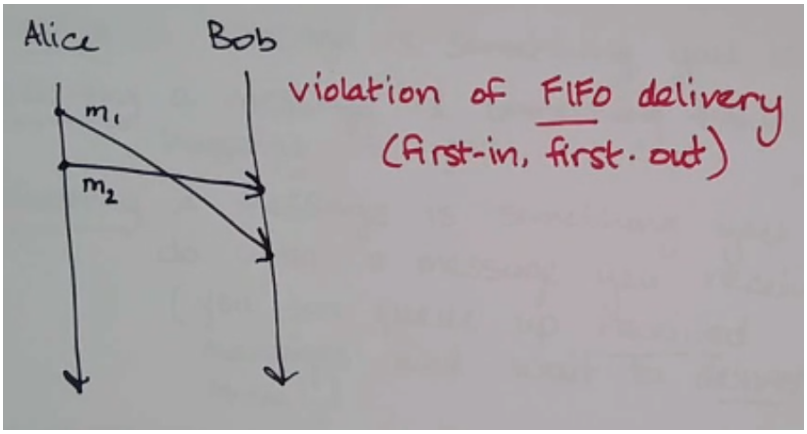


3.1 FIFO Delivery

if a process sends message m_2 after message m_1, any process *delivering* both delivers m_1 first.

Note:

- *Sending* a message is something you do
- *Receiving* a message is something that happens to you
- *Delivering* a message is something you *can* do with a message you receive (you can queue up receives and wait to deliver them)



Most systems programmers don't have to worry about this often - it's already part of TCP!

3.1.1 Sequence Numbers

This can be implemented using *sequence numbers*:

- messages are tagged with the sender ID, and sender sequence number
- senders increment their sequence number after sending
- if a received message's sequence number is (previously received seq num) + 1, deliver it

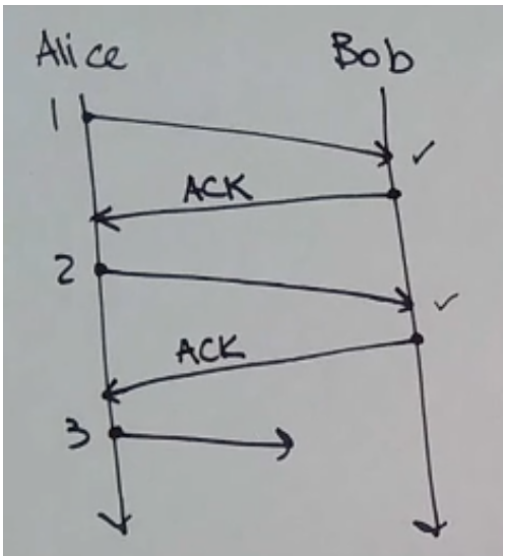
However, this only works well if you also have *reliable delivery* - if not, and the receiver misses one, it will buffer all future messages forever.

There are some solutions to this - like ignoring the missed message and delivering all buffered ones - but if the missed one shows up later, it has to be dropped.

Also, just dropping every single message satisfies FIFO vacuously.

3.1.2 ACK

Another implementation is ACK - Alice waits for Bob to send an acknowledgement before sending the next message.

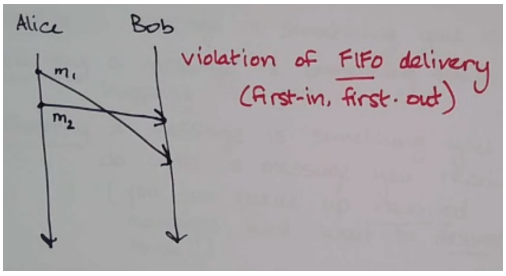


However, it's a lot slower since it requires a full round trip per message.

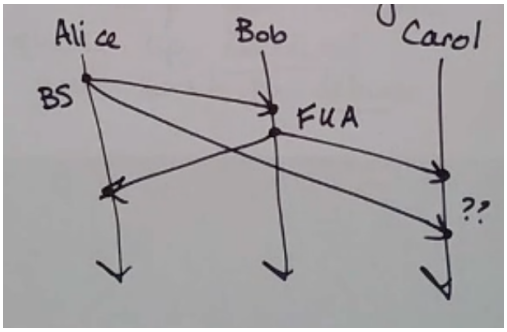
3.2 Causal Delivery

If m_1's send happened before m_2's send, then m_1's delivery must happen before m_2's delivery.

The violation of FIFO delivery above is also a violation of causal delivery:



Note that however, this diagram does not violate FIFO delivery, since FIFO delivery only accounts for messages sent from a single process.



It is a violation of *causal delivery*, though.

3.2.1 Implementing Causal Broadcast

`unicast`
: 1 sender, 1 receiver (aka point-to-point)

`multicast`
: 1 sender, many receivers

`broadcast`
: 1 sender, *all* processes in the system receive

For all of these above, no matter how many receivers there are, each send is considered one message.

First, we'll examine the **vector clocks algorithm** with a twist: *message receives don't count as events*.

- Every process keeps a VC, initially 0's
- When a process sends a message, it increments its own position in the VC, and includes the updated VC with the message
- When a process *delivers* a message, it updates its VC to the pointwise maximum of its local VC and the message's

`causal delivery`
: the property of executions that we care about today

`causal broadcast`
: an algorithm that gives you causal delivery in a setting where all messages are broadcast messages

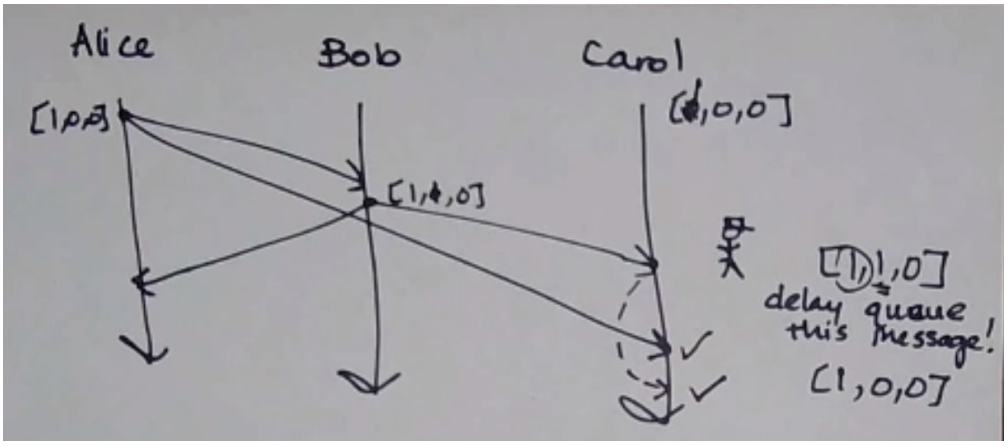
We want to define a deliverability condition that tells us whether or not a received message is os is not OK to deliver. This deliverability condition will use the vector clock on the message.

deliverability

a message *m* is deliverable at a process *p* if:

- $VC(m)[k] = VC(p)[k] + 1$, where *k* is the sender's position
- $VC(m)[k] \leq VC(p)[k]$, for every other *k*

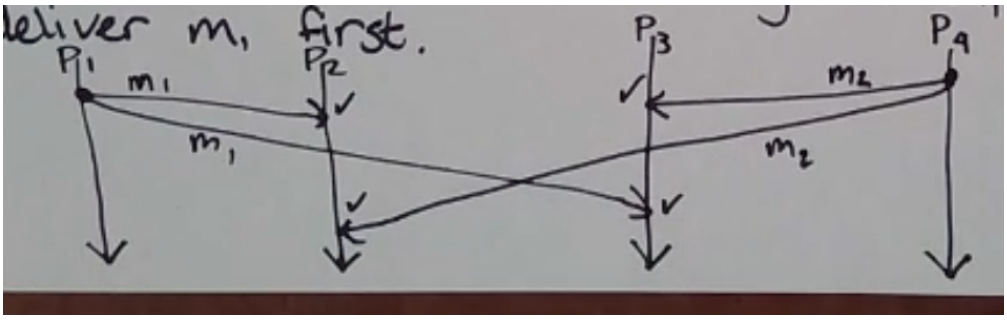
If a message is not deliverable, add it to the delivery queue, and check for deliverability each time you receive a new message (update your VC).



3.3 Totally Ordered Delivery

If a process delivers m_1 then m_2, then *all* processes delivering both m_1 and m_2 deliver m_1 first.

The image below is a violation, since P2 delivers 1 then 2, but P3 delivers 2 then 1.



CHAPTER

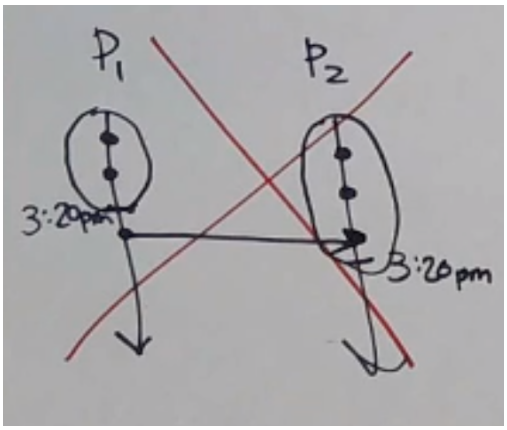
FOUR

SNAPSHOTS OF DISTRIBUTED SYSTEMS

Processes have individual *state*, which is pretty much all the events that have happened up to a given point.

How do we get the state of an entire distributed system (a "global snapshot") at a given time?

We can't use time-of-day clocks, since they aren't guaranteed to be synchronized across all processes.



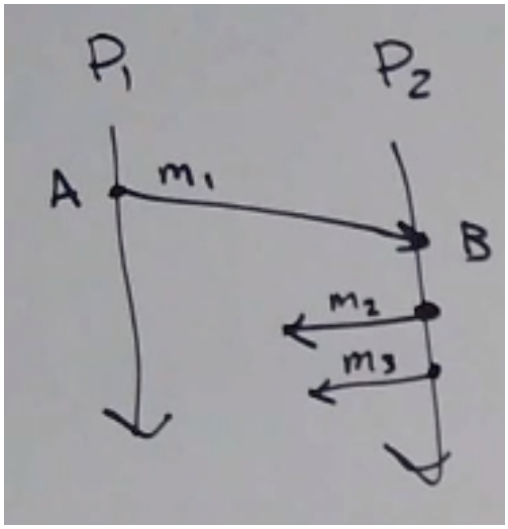
`consistent`

> **Property that we want**: If we have events *A* and *B* where $A \rightarrow B$, and *B* is in the snapshot, then *A* is also in the snapshot.

4.1 Chandy-Lamport Algorithm

channel

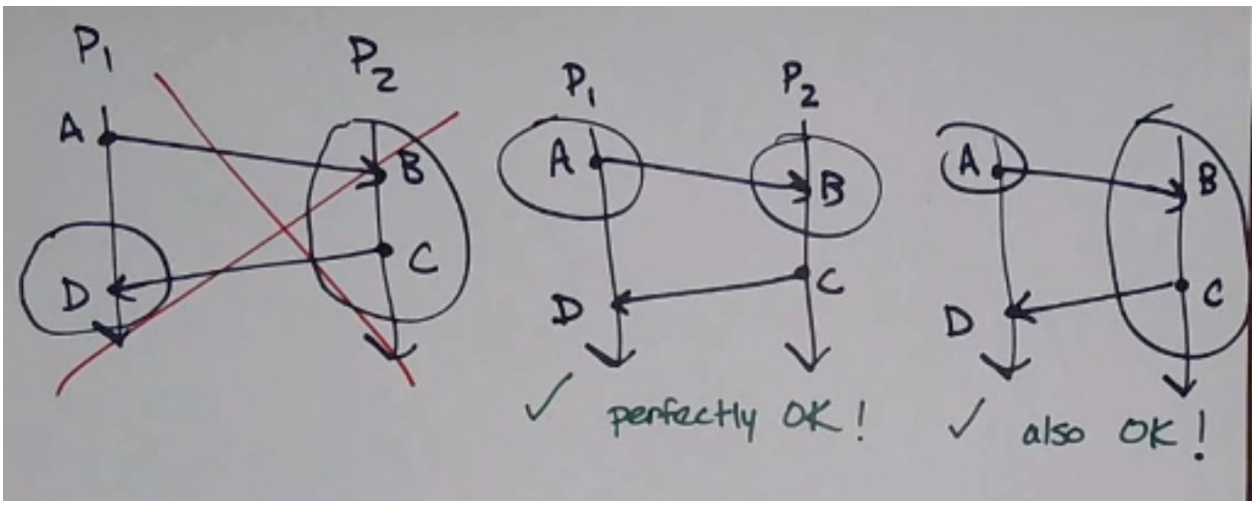
connection from one process to another, with FIFO ordering (e.g. C_{12}: the channel from P_1 to P_2)



m_2 is *in* C_{21}, $C_{21} = [m_2, m_3]$ (in-transit)

Note that a process graph must be *strongly connected* for C-L to work.

Some examples of snapshots:



How it works:

First, an initiator process (1 or more):

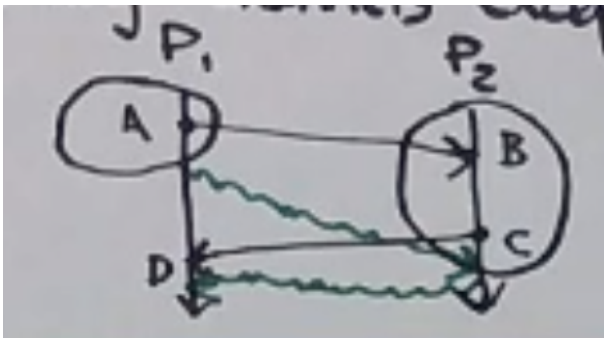
- records its own state
- sends a *marker message* out on all its outgoing channels
- starts *recording* the messages it receives on all its incoming channels

then, when process P_i receives a marker message on C_{ki}:

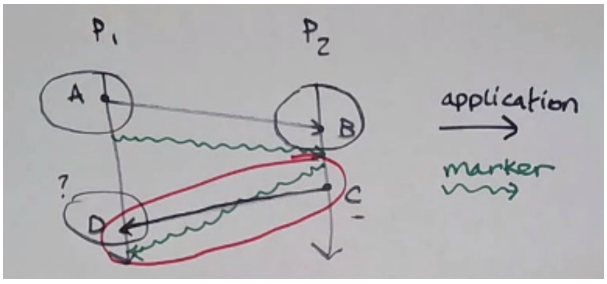
- **if it is the first marker P_i has seen (sent or received):**

  - it records its state
  - marks channel C_{ki} as empty
  - sends a marker out on all its outgoing channels
  - starts recording on all incoming channels *except* C_{ki}

- **otherwise:**
  - it stops recording on channel C_{ki}

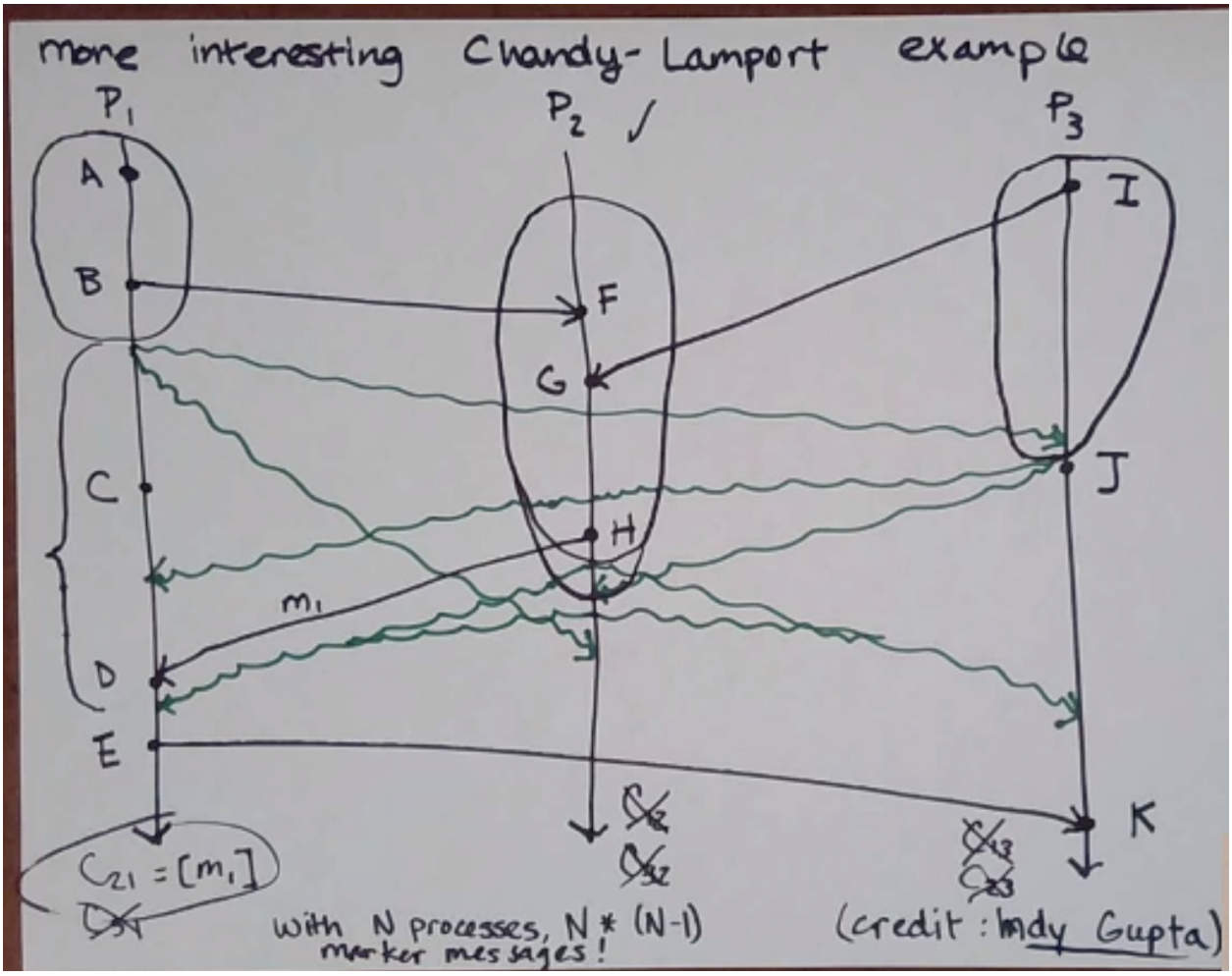


^ this is a consistent snapshot!



^ Note that this cannot happen since channels are FIFO!

Note: Since each process sends a marker message to every other message, this algorithm sends $n(n-1)$ messages in total.



4.1.1 The Big Picture

in Chandy-Lamport Snapshots

The bad:

- **channels** are required to be FIFO, and messages sent down them can be represented as an ordered list (e.g. $C_{12} = [m_1, m_2]$)
- because channels are FIFO, you never have to pause applications sending messages
- C-L snapshotting also assumes you have *guaranteed delivery* (messages are not lost, corrupted, or duplicated)
- it also assumes that processes don't crash

The good:

- snapshots it takes are consistent, regardless of how long it takes to send the marker messages
- guaranteed to terminate (given assumptions above)
- works fine with more than one initiator! (decentralized - a centralized alg can only be initiated by one process)

Why?

- checkpointing

- deadlock detection
- detection of any *stable property*

CHAPTER

FIVE

SAFETY & LIVENESS

Previously:

- FIFO delivery
- Causal delivery
- Totally ordered delivery

These are all safety properties: properties that say a "bad" thing *won't* happen; on the other hand, a *liveness* property says that a "good" thing *will* happen (e.g. eventual delivery).

Note: The word *eventually* is often an indicator of a liveness property.

Now, formally:

`safety property`

- say that a "bad" thing won't happen
- can be violated in a finite execution (can point to where it went wrong)

`liveness property`

- says that a "good" thing will (eventually) happen
- cannot be violated in a finite execution (may take infinite time to satisfy)

It's the combination of safety and liveness that makes systems useful - for example, otherwise, you could just drop every message to satisfy FIFO.

In fact, every property is either a safety property, a liveness property, or both.

Let's try and define reliable delivery:

(Take 1): Let P_1 be a process that sends a message to process P_2. If neither P_1 not P_2 crashes (and not all messages are lost), then P_2 eventually delivers *m*.

Well, it's a start, but the whole "not crashing" thing is weak.

5.1 Fault Models

A *fault model* tells you which kind of faults can occur. In the example system of two processes, one asking the other for a value, we have the following faults:

- message from M_1 is lost - omission fault
- message from M_1 is slow - timing fault
- M_2 crashed - crash fault
- M_2 is slow - timing fault
- message from M_2 is slow - timing fault
- message from M_2 is lost - omission fault
- M_2 lies - byzantine fault

`crash fault`
: a process fails by halting (stops sending/receiving messages) and not everyone necessarily knows about it

  (There is actually a subset, called "fail-stop faults", where a process halts and everyone *knows* it halts)

`omission fault`
: a message is lost (a process fails to send/receive a single message)

`timing fault`
: a process responds too late (or too early)

`Byzantine fault`
: a process behaves in an arbitrary or malicious way

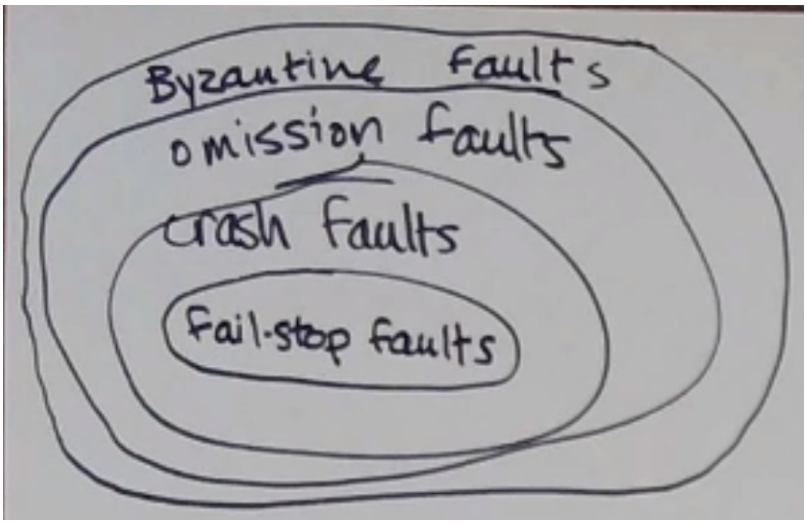
Note: There's a subset of Byzantine faults between omission and Byzantine called "authentication-detectable Byzantine faults" - i.e. message corruptions that can be detected, and "downgraded" to an omission fault

---

If protocol X tolerates crash faults and protocol Y tolerates omission faults, does Y also tolerate crash faults?

Yes! Crash faults are just a special case of omission faults where *all* messages to/from a process are lost.

In fact, Byzantine faults are also a superset of crash (and omission) faults.



Note: We left out timing faults in the above diagram, since we're dealing only in async in this class.

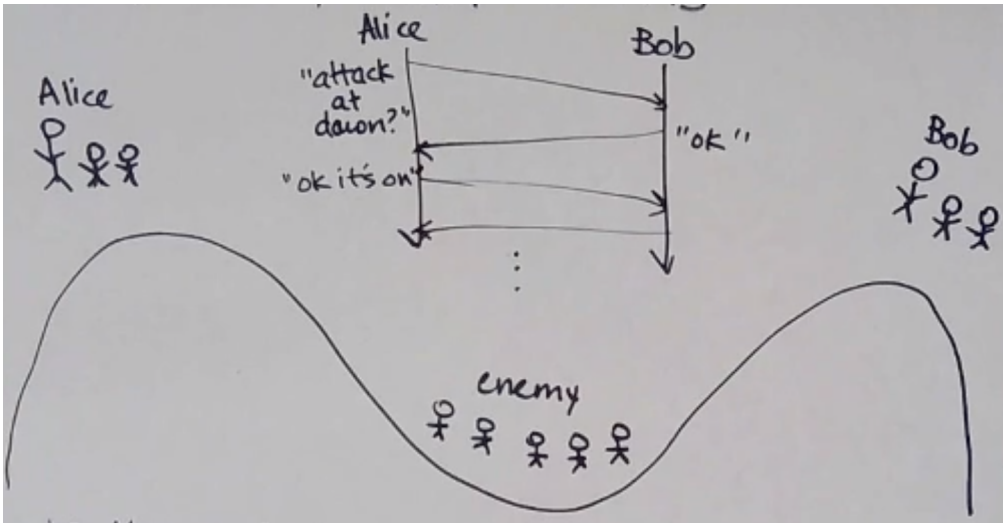
`fault model`

A *fault model* is a specification that says what kind of faults a system can exhibit, and this tells you what kinds of faults need to be tolerated.

In this class, we focus on the *omission model* (which includes the *crash model*).

5.2 Two Generals Problem

Tom Scott: https://www.youtube.com/watch?v=IP-rGJKSZ3s



In the omission model, it is impossible for Alice and Bob to attack and know for sure that the other will attack.

How do we mitigate?

5.2.1 Probabilistic Certainty

One option is to have Alice constantly send until she receives an ACK; then the longer Bob goes without receiving a message, the more sure he is that she has received the ACK. Note that it's not 100% guaranteed; every single message since then could have failed, but it works.

5.2.2 Common Knowledge

There is common knowledge of p when everyone knows p, everyone knows that everyone knows p, everyone knows that everyone knows that everyone knows p...

5.3 Fault Tolerance

What does it mean to tolerate a class of faults? Usually it's defined by how/how much your program reacts to a fault.

A *correct* program satisfies both its safety and liveness properties, but often satisfying both is impossible during a fault.

So really, it's about *how wrong* it goes in the presence of a fault.

	live	not live
safe	masking	fail-safe
not safe	non-masking	:(

5.4 Reliable Delivery, Take 2

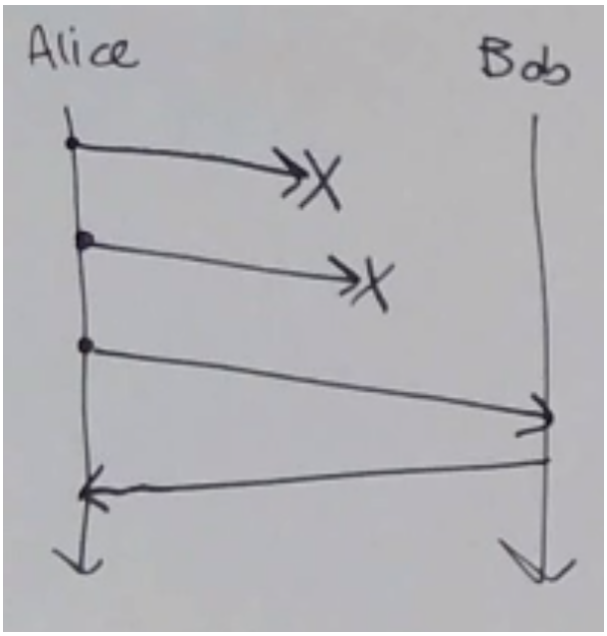
Previously: Let P_1 be a process that sends a message m to P_2. If neither P_1 nor P_2 crashes (and not all messages are lost), then P_2 eventually delivers m.

Do we need `not all messages are lost`? Yes, if we're working under the omission model.

So how do we implement it?

One implementation: repeat until ack

- Alice puts a message in the send buffer
- on timeout, send what's in the buffer
- when ack received, delete message from buffer



One problem is if Bob's ACK gets dropped, and he receives the message twice. This may or may not be an issue, depending on the message. For example, if the message is something like "increase value by 1", that's an issue!

Note: Client A sending a set request, then client B, then a repeated send of client A is not actually an issue here - it's the same if client A's initial message was delayed (one client will be sad).

In this scenario, your messages should be *idempotent* - sending them multiple times should have the same effect as if it was sent once.

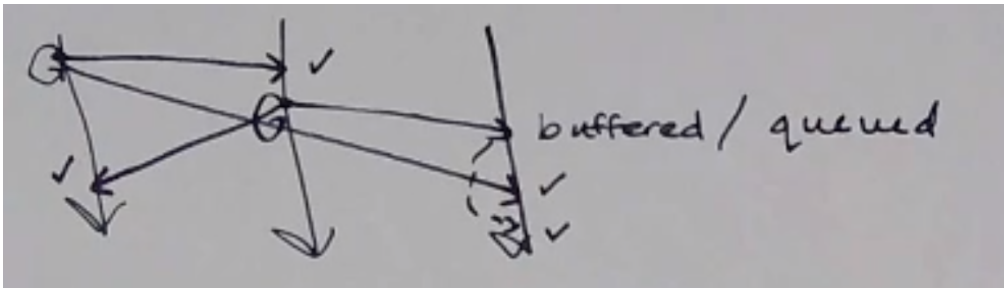
So this is actually *at-least-once* delivery! But is *exactly-once* delivery possible?

Not really… most systems that claim exactly-once delivery in real life are one of the following:

- the messages were idempotent anyway
- they're making an effort to deduplicate messages on the receiver

5.5 Reliable Broadcast

broadcast: one sender, everyone receives



Note: If you have a *unicast* primitive, you can implement broadcast by sending multiple unicasts in very quick succession.

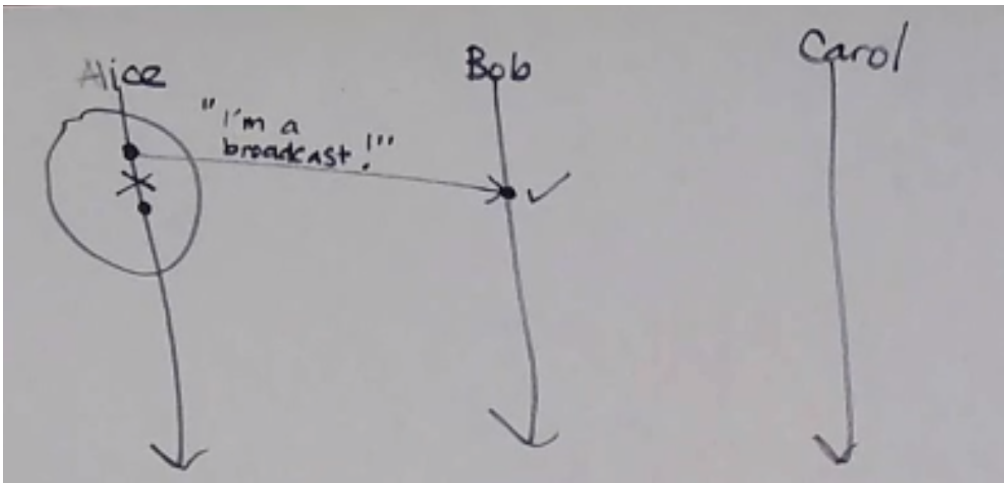
`reliable broadcast`

If a *correct process* delivers a broadcast message *m*, then all correct processes deliver *m*.

Note: A correct process is a process that is acting correctly in a given fault model (e.g. not crashed in crash model).

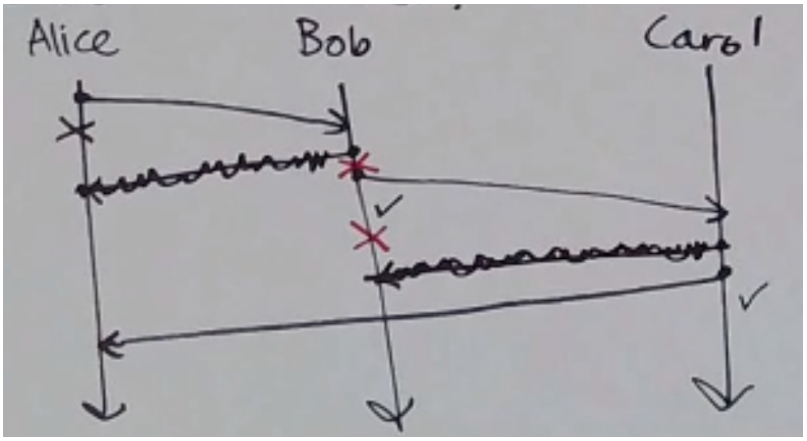
Ex. If Alice sends a message to Bob and Carol in the crash model, Bob delivers it, but Carol crashes before she can, reliable broadcast is not violated because Carol is not correct.

Ex. If Alice is sending a message to Bob and Carol in the crash model but crashes after Bob's is sent but before Carol's is, reliable broadcast is violated, since Bob is a correct process that delivered the message, but Carol did not.

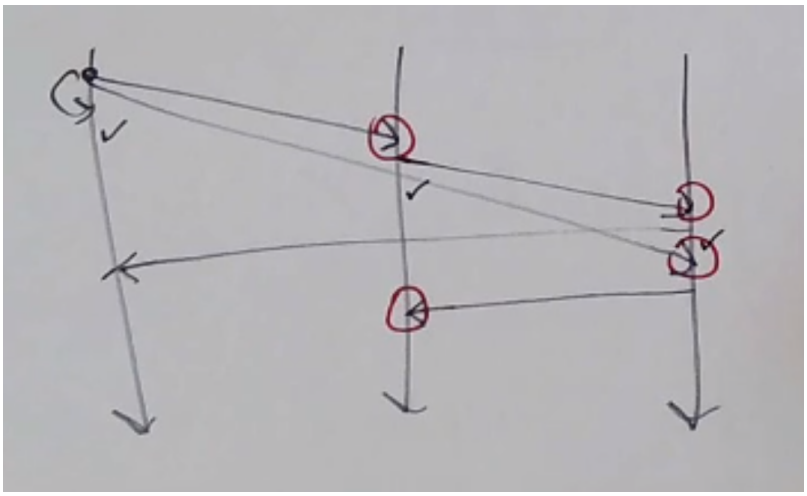


If a process can crash in the middle of a broadcast, how do we get reliable broadcast?

If you receive a broadcast message, forward it to everyone else (or, at least, everyone but the sender) before delivering it:



But if no one crashes, each process receives it twice:



But each process can keep track of messages they've already delivered/forwarded to not do it twice.

Important: Fault tolerance often involves making copies!

CHAPTER

SIX

REPLICATION

Why do we replicate state/data?

- fault tolerance: prevent data loss
- data locality (keep data close to clients that need it)
- dividing the work

But:

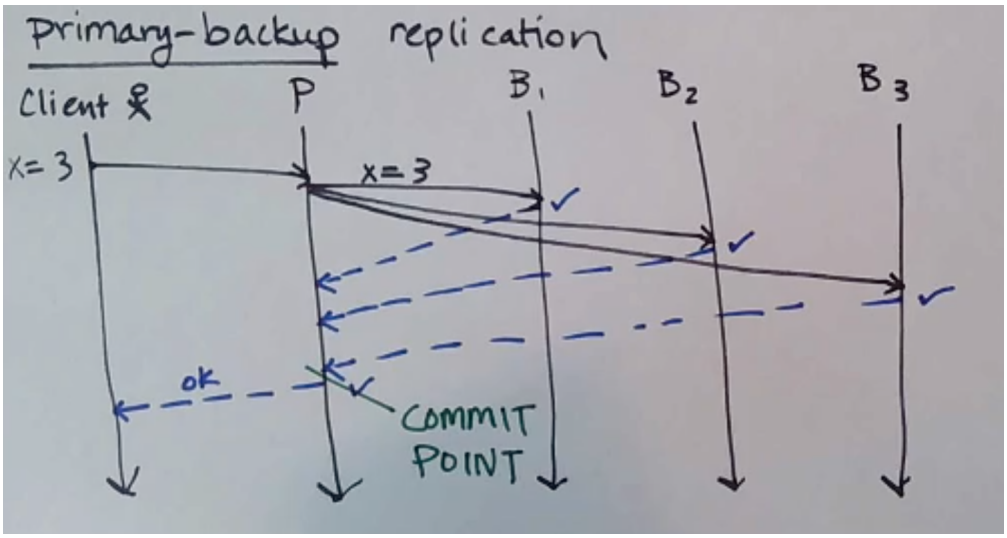
- higher cost
- hard to keep state consistent

Informally, a replicated storage system is *strongly consistent* when you cannot tell that the data is replicated.

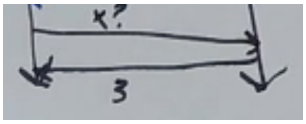
6.1 Primary-Backup Replication

First, pick a process to be the primary. All other processes are then backups. In this scenario, clients can only connect to the primary

When a client makes a write to the primary, the primary sends that write to all backups - when it then receives an ACK from all backups, that is the *commit point* and it returns OK to the client.



On a read, the primary just returns the value to the client without consulting with any backups.



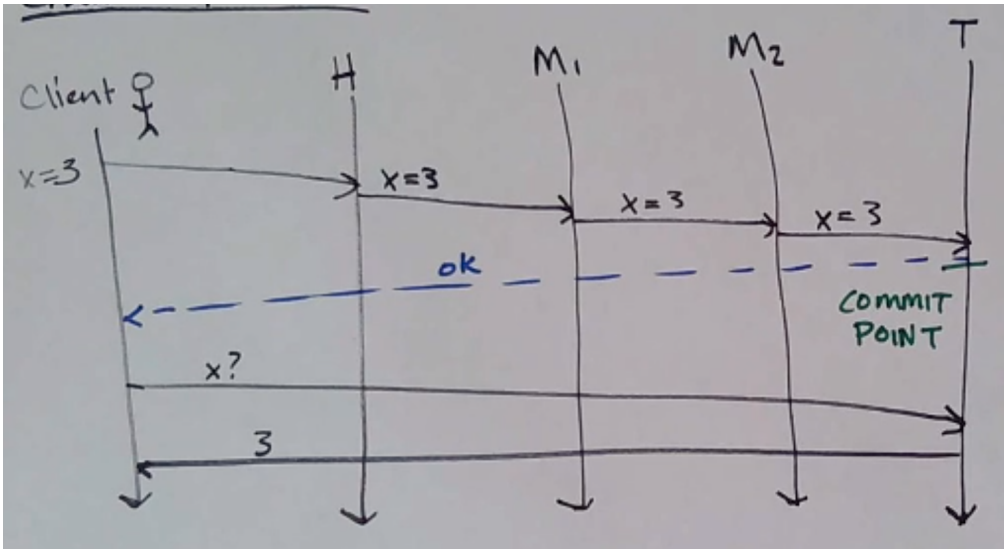
Note: The client must ask the *primary* instead of a backup because there might be an uncommitted write in flight - a backup might know it, but the primary might not have committed it yet.

How well does it match our replication criteria?

- [x] fault tolerance
- [ ] data locality
- [ ] dividing work

6.2 Chain Replication

Chain Replication for Supporting High Throughput and Availability - van Renesse, Schneider, 2004



Writes go to the head, which forwards to the next one in the chain, and so on to the tail. The tail ACKs the write.

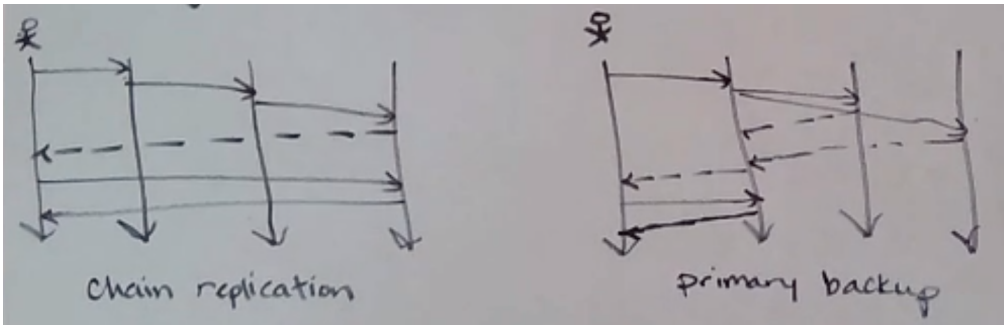
Reads go to the tail.

Well, how well does it do?

- [x] fault tolerance
- [~] data locality
- **[x] dividing work**
  - slightly better - reads and writes go to different processes

`throughput`
: number of actions per unit of time

Depending on the workload, CR could give you better throughput than PB.



For CR, the optimal workload ratio is about 15% writes to 85% reads.

So what's the downside of CR? Well, CR has a **higher write latency** (depending on # of nodes in chain). (the two have about the same read latency)

`latency`

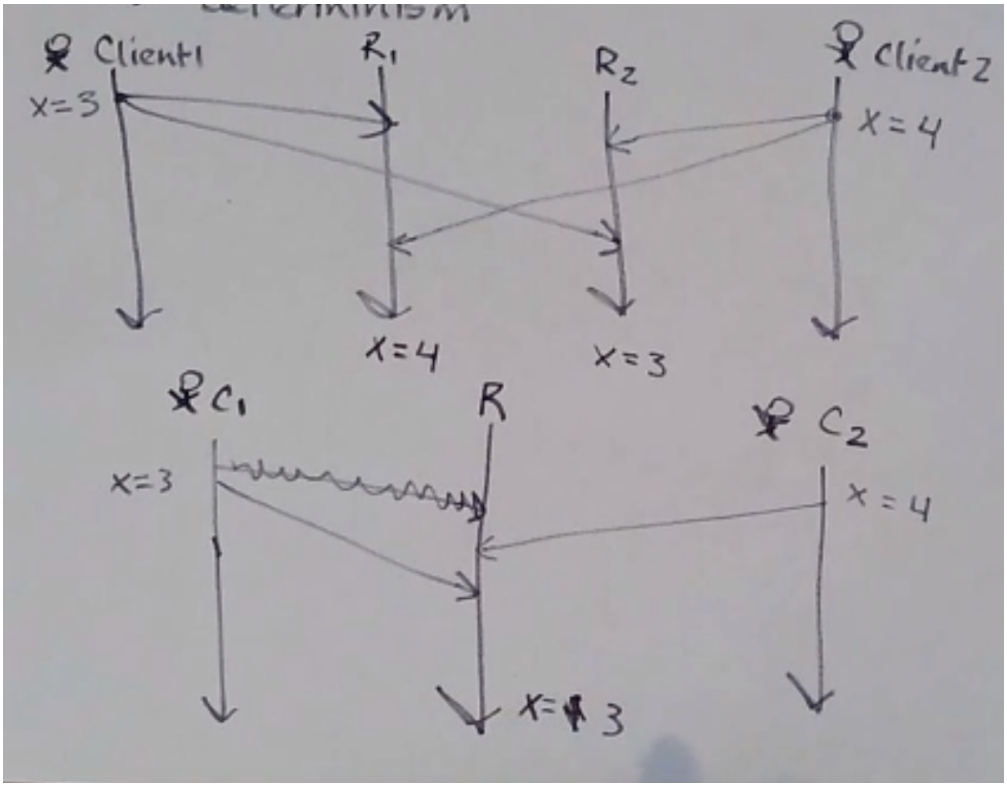
time between start and end of one action

Since PB broadcasts the write, it's processed in parallel by the backups, and it can be ACKed as soon as all backups ACK. For CR, the write message is forwarded, and has to be processed by each process in the chain in series.

Note: Regardless of which replication scheme you choose, the client and replicas have to agree on who's the primary/head/tail/etc, or else you lose the guarantees of replication!

6.3 Total Order v. Determinism

Messages sent by different clients at the same time can arrive at different times:



In the second example, there's no violation of TO delivery, but the result is not the same depending on which client's message receives first!

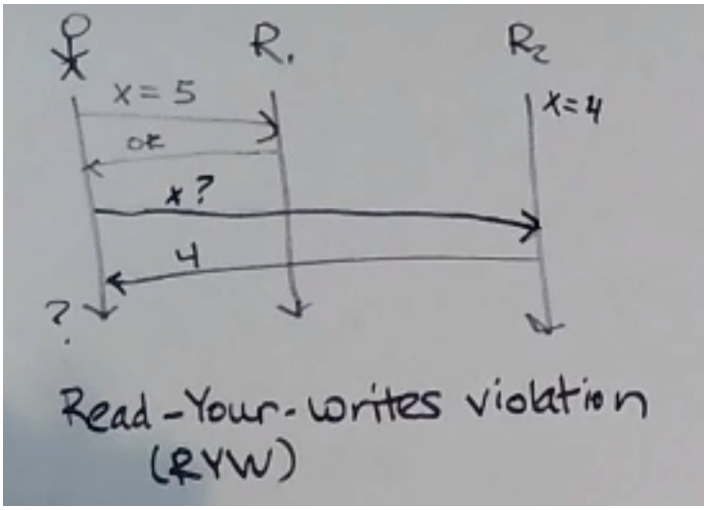
`determinism`

On every run, the same outcome is achieved.

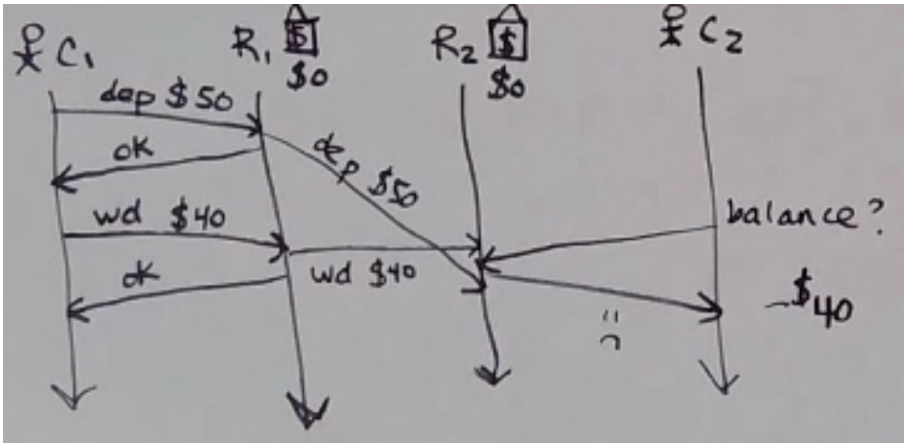
6.4 Bad Things

What happens if a client *can* tell that data is replicated (i.e. the replication is not strongly consistent)?

Read-Your-Writes Violation: A client's written is not immediately returned on a subsequent read.



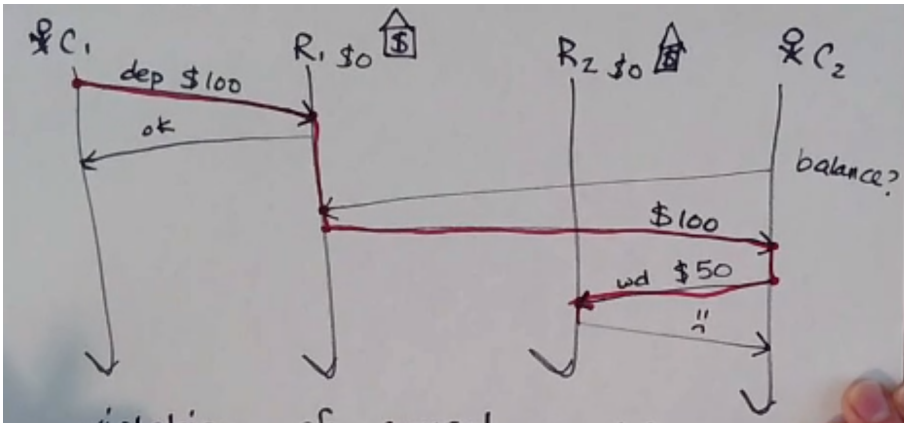
FIFO Consistency Violation



`fifo consistency`

Writes done by a single process are seen by all processes in the order they were issued

Causal Consistency Violation



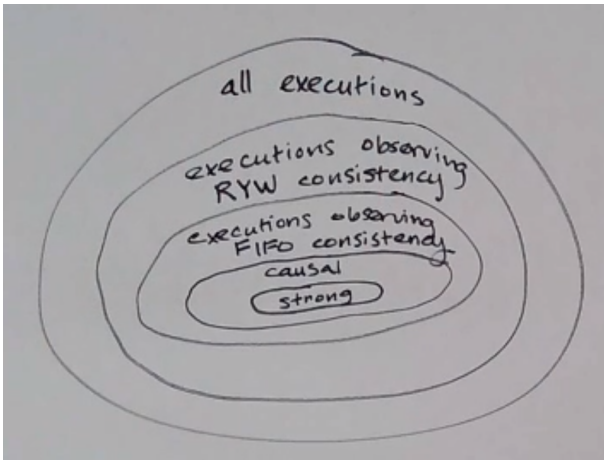
`causal consistency`

Writes that are related by happens-before (i.e. potentially causally related) must be seen in the same causal order by all processes

6.5 Consistency

Actually, we can define different **consistency models**:

(aside: a *model* is the set of assumptions you keep in mind when building a system)



But maintaining stronger consistency requires more work, which means more latency and just being harder! Remember, replication/consistency usually involves duplicating messages too, so more bandwidth too

6.6 Coordination

Going back to our strongly consistent replication protocols (PB/CR) - both of these need some kind of coordinator process to know which process is the primary/head/tail/etc.

6.6.1 Chain Replication

CR uses the fail-stop fault model (i.e. crashes can occur and be detected by the environment), and requires that not all processes crash. There are some ways to implement this (like heartbeating), but sometimes you'll have a false positive.

- If the head process crashes, the coordinator makes the next process in line the new head
- If the tail process crashes, the coordinator makes the preceding process the new tail
- If a middle processes crashes, it just gets skipped over (although the clients do not have to be notified)

Additionally, when a failure happens, there has to be some handling of writes that are partway through the chain when the failure happened - out of the scope of this class though (van Renesse & Schneider, 2004).

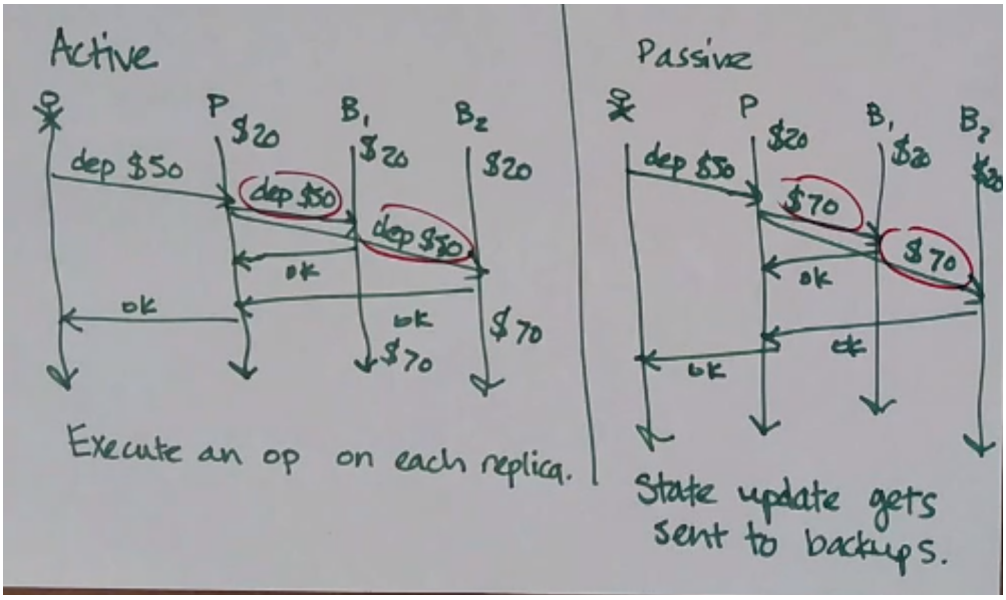
Important: What if the coordinator fails?!?!?! Do we have to replicate the coordinator?

(Next: Consensus)

6.7 Active v. Passive Replication

- Active replication: execute an operate on each replica (aka state machine replication)
- Passive replication: state gets sent to backups

Example (primary-backup replication):



You might choose one over the other for the following reasons:

- the updated state might be large (use active)
- an operation might be expensive to do on each replica (use passive)
- the operation might depend on local process state (use passive)

CHAPTER

SEVEN

CONSENSUS

Consensus is *hard*.

When do you need it? When you have a bunch of processes, and…

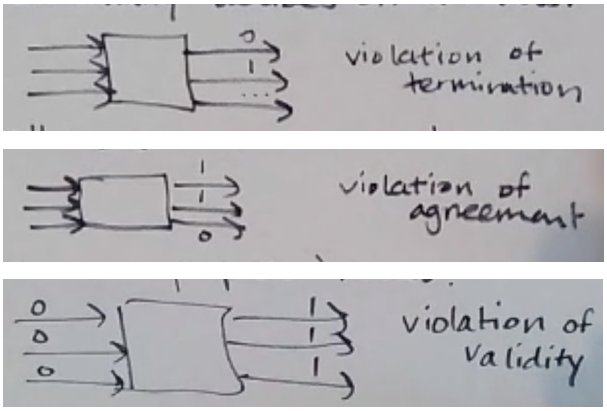
- they need to deliver the same messages, in the same order (**totally-ordered/atomic broadcast**)
- they need to each know what other processes exist, and keep those lists up to date (**group membership problem, failure detection**)
- one of them needs to play a particular specific role and the others need to agree who that is (**leader election**)
- they need to be able to take turns accessing a resource that only one can access at a time (**mutual exclusion problem**)
- they're participating in a transaction and need to agree on a commit/abort decision (**distributed transaction commit**)

We can view consensus as a kind of box, with multiple inputs going in and coming out; the inputs might differ, but in a correct system, they agree coming out.

7.1 Properties

Consensus algorithms try to satisfy the following properties:

- **termination**: each correct process eventually decides on a value (whatever it is)
- **agreement**: all correct processes decide on the *same* value
- **validity** (aka integrity, nontriviality): the agreed-upon value must be one of the proposed values



Note: In the asynchronous network + crash fault model, no algorithm actually satisfies all 3 of these (Fischer, Lynch, Paterson, 1983)! In this model, we have to compromise, and usually it's termination that we compromise on.

7.2 Paxos

Leslie Lamport, 1998

Each process takes on some of 3 roles:

- **proposer** - proposes values
- **acceptor** - contribute to choosing from among the proposed values
- **learner** - learns the agreed-upon value

A process could take on multiple roles, but we usually examine the case where each process only takes on one. Each process that plays any role is called a *Paxos node*.

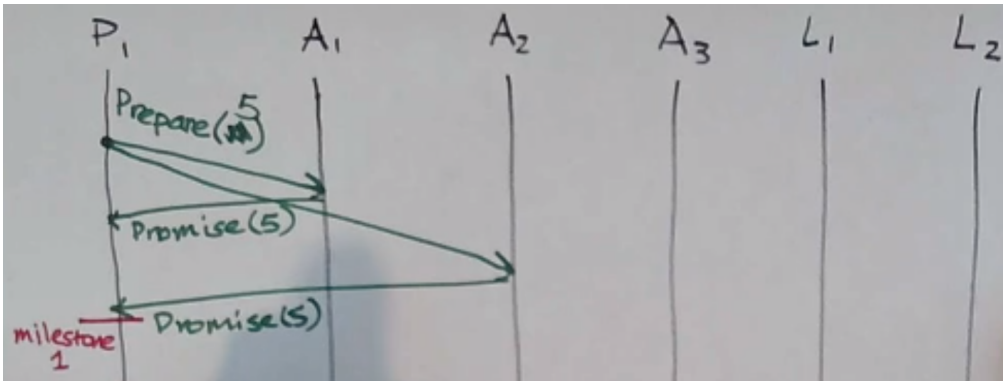
Paxos nodes must:

- persist data
- know how many nodes is a majority of acceptors

7.2.1 How it Works

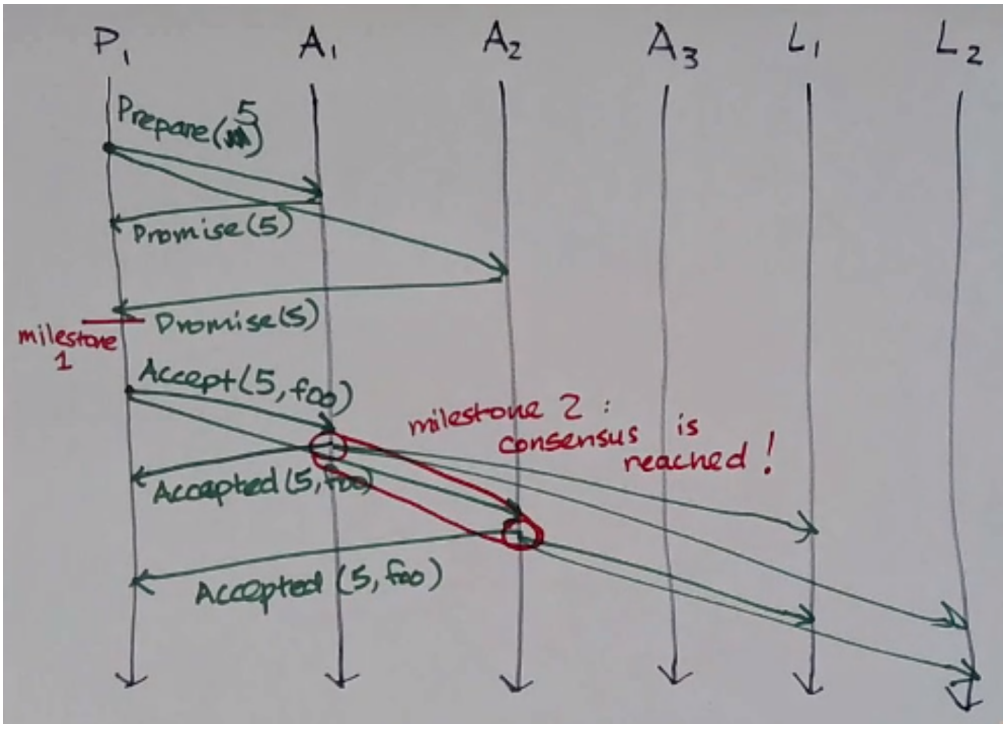
Phase 1

- **proposer: sends a prepare message with a unique proposal number to a majority of acceptors, `Prepare(n)`**
  - the proposal number *n* must be unique, and higher than any proposal number that *this* proposer has used before
- acceptor: upon receiving a `Prepare(n)` message, check: "did I previously promise to ignore requests with this proposal number?"
  - if so, ignore the message
  - **otherwise, promise to ignore requests with proposal number < *n*, and reply with `Promise(n)` (*)**
    - `Promise(n)`: "I will ignore any request with a proposal number < n"
    - when a majority reply with a promise, we reach a milestone: it is impossible to get a majority to promise anything lower than *n*

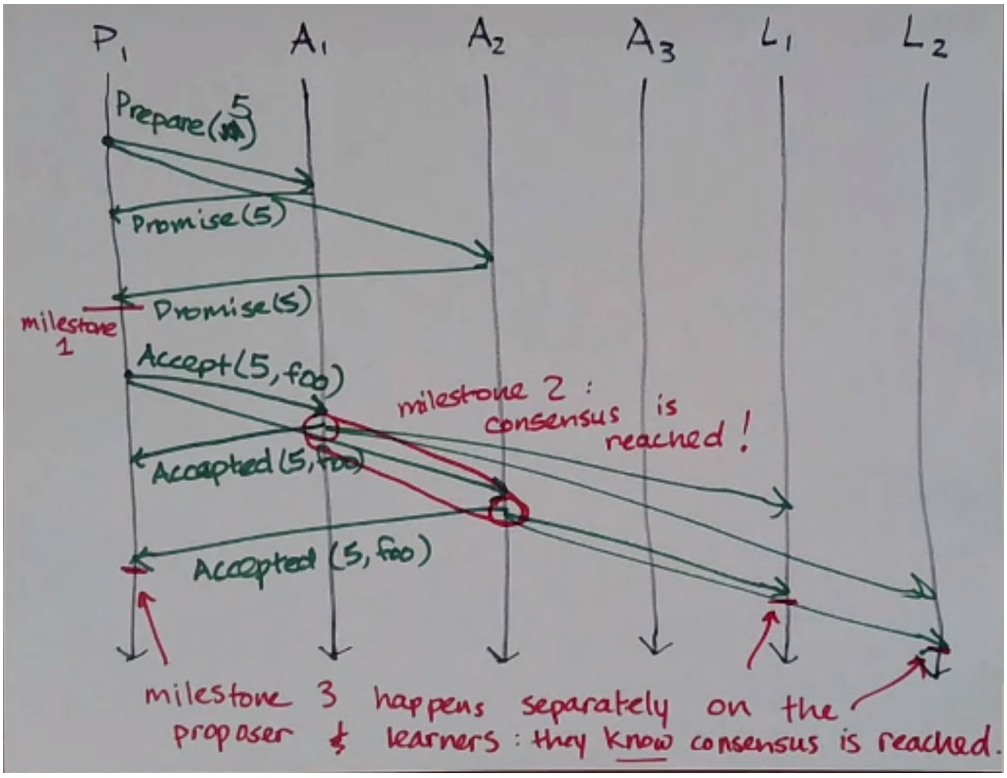


Phase 2 - the proposer has received `Promise(n)` from a majority of acceptors (for some *n*)

- **proposer: send an `Accept(n, val)` message to at least a majority of acceptors, where:**
  - *n* is the proposal number that was promised
  - *val* is the actual value it wants to propose (**)
- **acceptor: upon receiving an `Accept(n, val)` message, check if it previously promised to ignore requests with *n***
  - if so, ignore the message
  - **otherwise, reply with `Accepted(n, val)`, and also sends `Accepted(n, val)` to all *learners***
    - when the majority of acceptors have sent an `Accepted(n, val)` message for a given *n*, we reach a milestone: we have consensus on *val* (but no one knows)



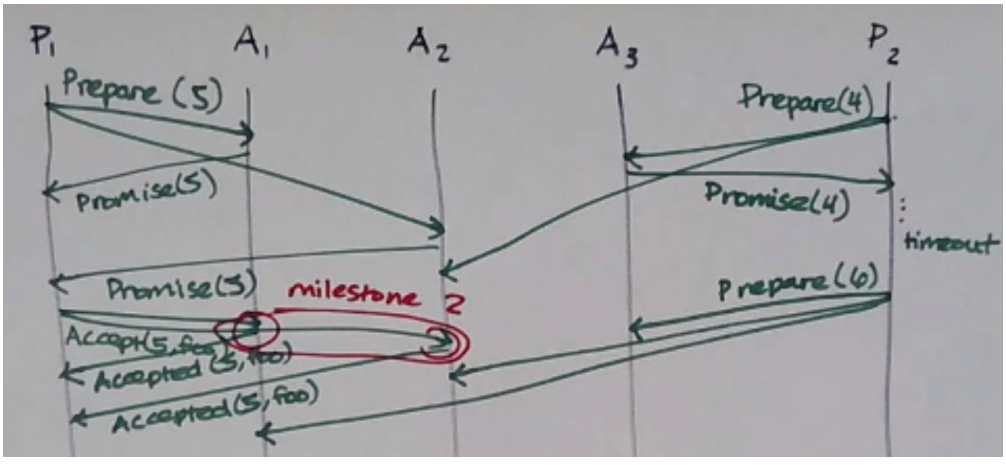
When each participant receives `Accepted` from a majority of acceptors, then they know consensus is reached (this happens separately on the proposer and learners)



7.2.2 Getting Weird

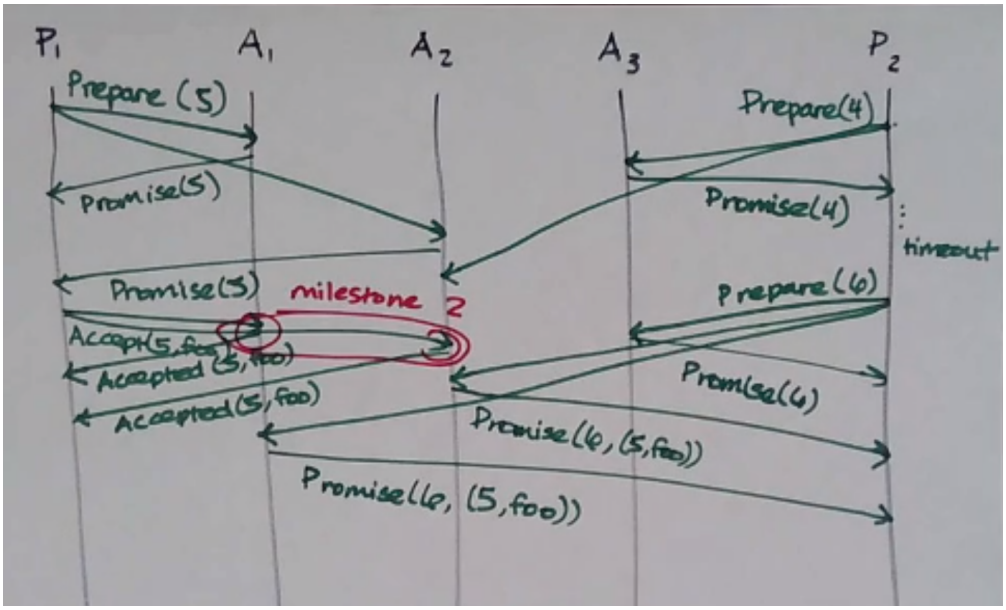
Consider the following execution:

- P1 sends `Prepare(5)`
- **P2 sends `Prepare(4)`**
  - it doesn't receive ``Promise``s in time, so it tries again with a higher proposal number
- P1 sends `Accept(5, foo)` and the acceptors send `Accepted(5, foo)` at the same time P2 sends `Prepare(6)`
- oh no!



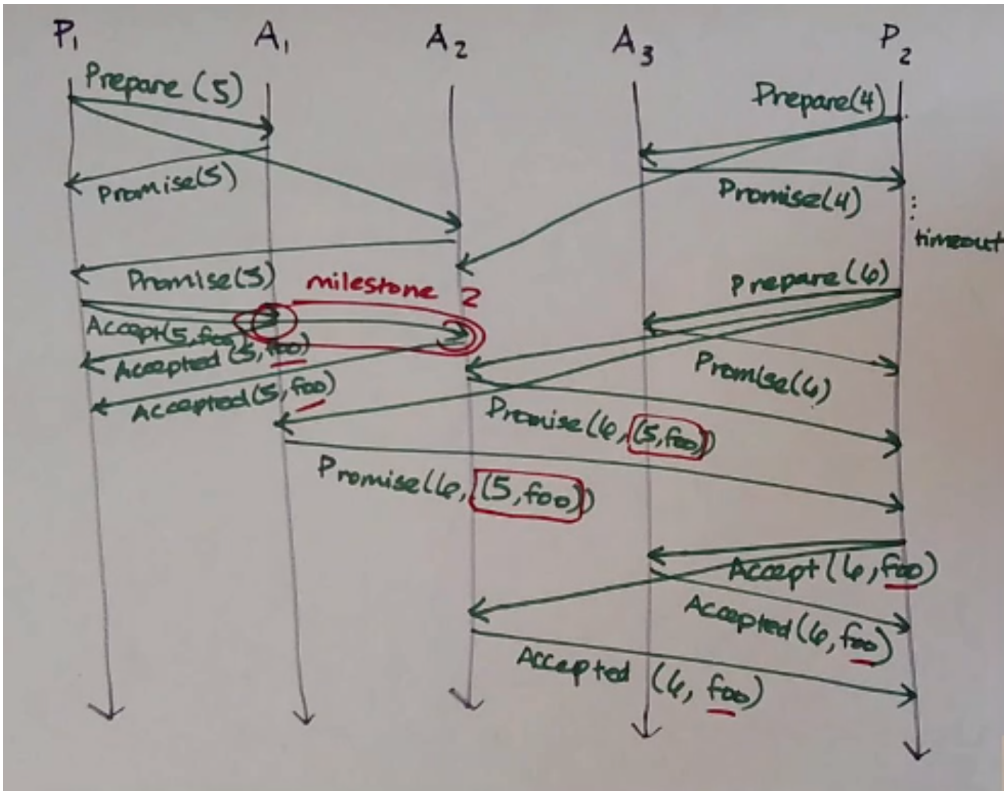
So we have to change what happens in phase 1:

- acceptor: upon receiving a `Prepare(n)` message, check: "did I previously promise to ignore requests with this proposal number?"
  - if so, ignore the message
  - **otherwise, check if it has previously accepted anything**
    - if so, reply with `Promise(n, (n_prev, val_prev))`, where `n_prev` is the highest previously-accepted proposal number and `val_prev` is the previously accepted proposed value
    - otherwise, reply with `Promise(n)`



Now, in phase 2, the proposer has to do something different having received `Promise(n)` *or* `Promise(n, (n_prev, val_prev))` from a majority of acceptors:

- **proposer: send an `Accept(n, val)` message to at least a majority of acceptors, where:**
  - *n* is the proposal number that was promised
  - ***val* is chosen as follows:**
    - the `val_prev` corresponding to the highest `n_prev`
    - or the value it wants, if no `n_prev` info was received



7.2.3 Final Algorithm

Phase 1

- **proposer: sends a prepare message with a unique proposal number to a majority of acceptors, `Prepare(n)`**
  - the proposal number *n* must be unique, and higher than any proposal number that *this* proposer has used before
- acceptor: upon receiving a `Prepare(n)` message, check: "did I previously promise to ignore requests with this proposal number?"
  - if so, ignore the message
  - **otherwise, check if it has previously accepted anything**
    - if so, reply with `Promise(n, (n_prev, val_prev))`, where `n_prev` is the highest previously-accepted proposal number and `val_prev` is the previously accepted proposed value
    - otherwise, reply with `Promise(n)`

Phase 2 - the proposer has received `Promise(n)` or `Promise(n, (n_prev, val_prev))` from a majority of acceptors (for some *n*)

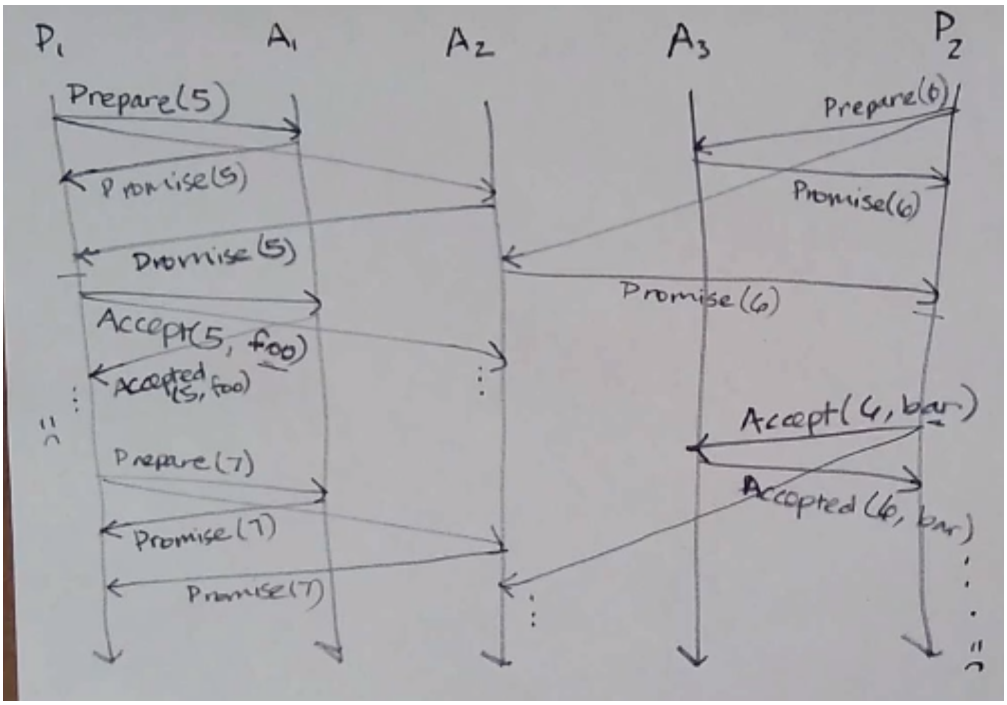
- **proposer: send an `Accept(n, val)` message to at least a majority of acceptors, where:**
  - *n* is the proposal number that was promised
  - ***val* is chosen as follows:**
    - the `val_prev` corresponding to the highest `n_prev`
    - or the value it wants, if no `n_prev` info was received

- **acceptor: upon receiving an `Accept(n, val)` message, check if it previously promised to ignore requests with *n***
  - if so, ignore the message
  - **otherwise, reply with `Accepted(n, val)`, and also sends `Accepted(n, val)` to all *learners***
    - when the majority of acceptors have sent an `Accepted(n, val)` message for a given *n*, we reach a milestone: we have consensus on *val* (but no one knows)

Paxos satisfies agreement and validity! What might cause it not to terminate…?

7.2.4 Non-Termination

Paxos can fail to terminate if you have *duelling proposers*:



In the above execution, a proposer never receives a majority `Accepted` because the other proposer butts in.

So why don't we just always only have one proposer? Theoretically we could just have one process declare itself the leader and tell the others what the accepted value it is… but picking the leader requires consensus in itself!

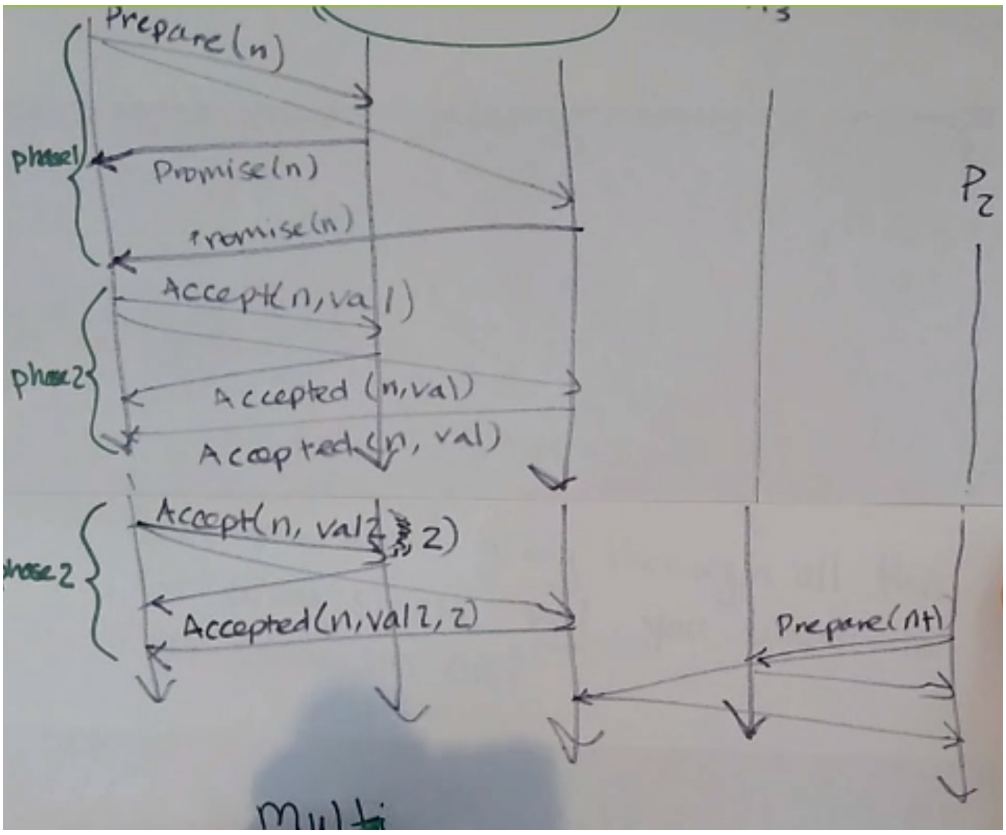
We could, however, choose a different leader election protocol to choose the proposer for a paxos run; and that leader election protocol could have different guarantees (e.g. termination and validity, instead of agreement/validity)

7.3 Multi-Paxos

Paxos is good for gaining consensus on a *single* value - for multiple (e.g. a sequence of values), you have to rerun the whole thing. What if you want to decide on a *sequence* of values?

For example, if you wanted to implement TO delivery (e.g. in a system where 2 messages are sent), you need to agree on two values: what message is sent first, and which is second. In normal Paxos, this takes a lot of messages!

What if we (well, the accepted proposer) just kept sending `Accept` messages with the same proposal number (i.e. kept repeating phase 2)? Turns out, you can keep doing this until your messages start getting ignored (i.e. a higher `Prepare` is received)!



And if a second process butts in, Multi-Paxos pretty much just becomes normal Paxos.

Note: An alternative way to agree on a sequence is *batching* - queuing up multiple values and using normal Paxos to get consensus on a batch at a time.

7.4 Paxos: Fault Tolerance

We can't have just one acceptor, since it can crash.

7.4.1 Acceptors

If we have 3 acceptors, only one can crash and Paxos will still work - you need to hear from both live acceptors.

If you have 5, you can accept 2 crashes. In general, a *minority* of acceptors can crash.

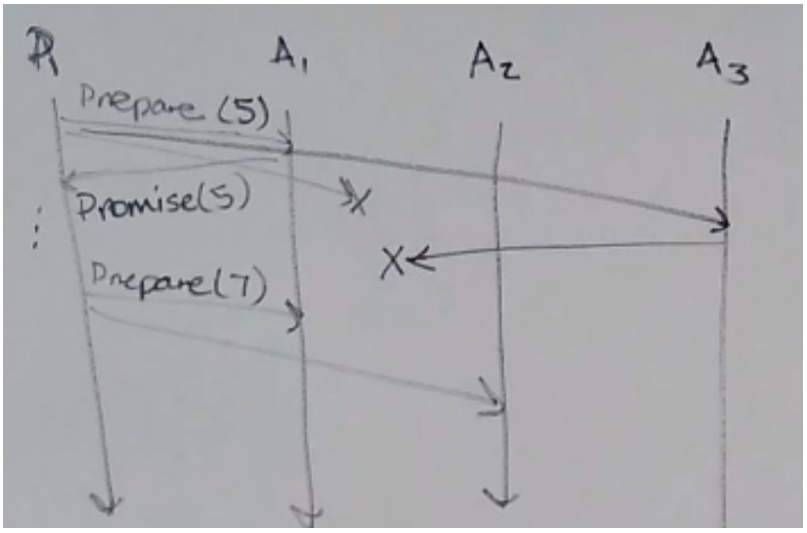
If f is the number of acceptor crashes you want to tolerate, you need $2f + 1$ acceptors.

7.4.2 Proposers

If f is the number of proposer crashes you want to tolerate, you need $f + 1$ proposers.

7.4.3 Omission Faults

Paxos is tolerant to omission faults (given timeouts) - it might not terminate, but that's already not a guarantee, so eh. In this scenario, it's *safe* but not *live* - fail-safe.



7.5 Other Consensus Protocols

- **Raft (Diego Ongaro, John Ousterhout, 2014)**
  - designed to be easier to understand than other protocols
- Zab (Zookeeper Atomic Broadcast; Yahoo Research, late 2000s)
- Viewstamped Replication (Brian Oki, Barbara Liskov, 1998)

All of these are for a sequence of values, like Multi-Paxos.

CHAPTER

EIGHT

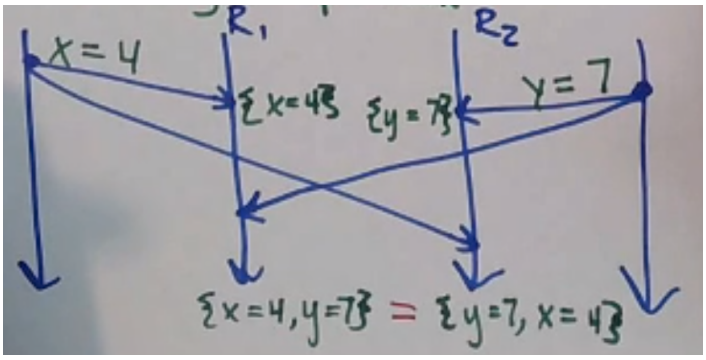
CONSISTENCY

See also: Replication

Recall: *strong consistency* is when a client can't tell that data is replicated.

Strong consistency ultimately relies on consensus, if you also want fault tolerance. For example, we can't enforce TO delivery using just vector clocks, but what if we could make it so that the order they arrived in didn't matter?

One example of a case where order doesn't matter is two clients trying to modify different keys:



This isn't *strong* consistency, though - a client could see the replication in the time between the two requests landing on each replica.

8.1 Eventual Consistency

`eventual consistency`

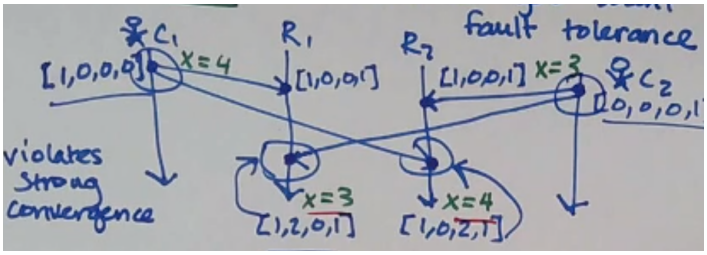
Replicas *eventually agree*, if clients stop submitting updates.

Eventual consistency is a *liveness* property, but usually consistency guarantees are safety properties - it's hard to consider in the same hierarchy as other guarantees. We do have a related safety property, though:

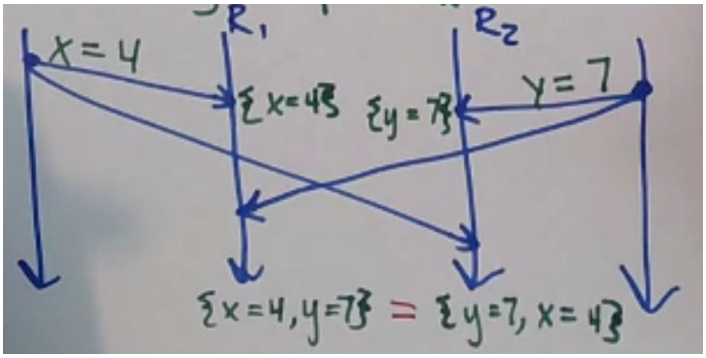
`strong convergence`

Replicas that have delivered the same *set* of updates have equivalent state.

This execution fails strong convergence:



But this execution satisfies it:

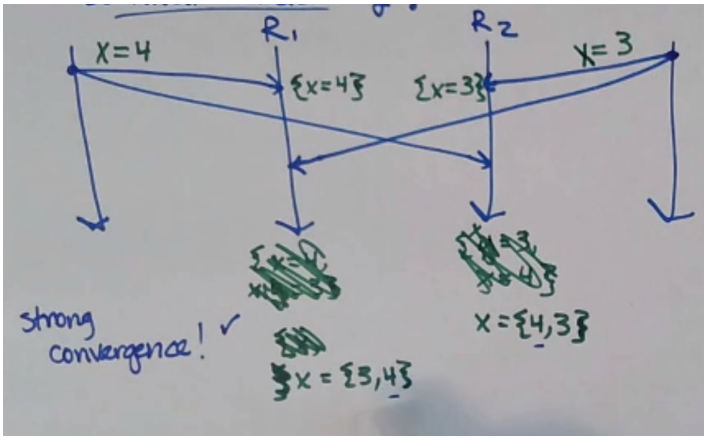


Often, these two properties are combined into *strong eventual consistency*, which is both a safety and liveness property.

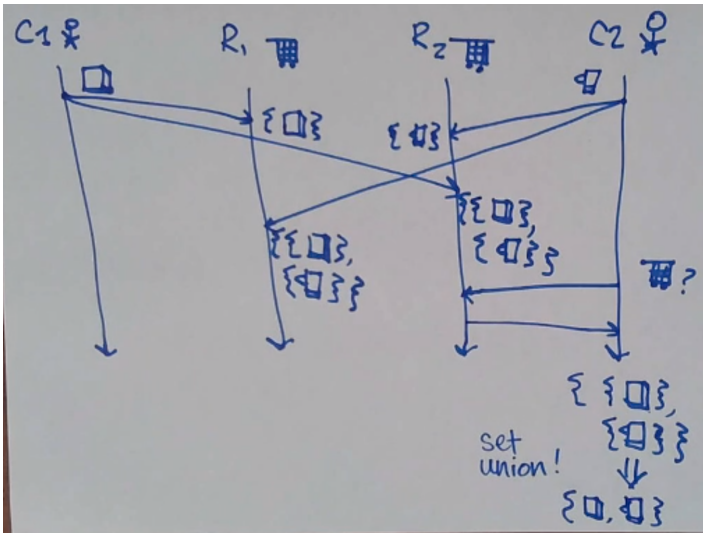
8.1.1 Concurrency

Strong convergence is easy if different clients write to different keys… what if we wanted different clients write to the same key?

Let's make our replicas save all concurrent updates in a set! But what does the client do?



Generally, it's up to the client (*application-specific conflict resolution*). In some cases, the client can be smart! For example, if your state is something like a shopping cart, you can merge the sets!



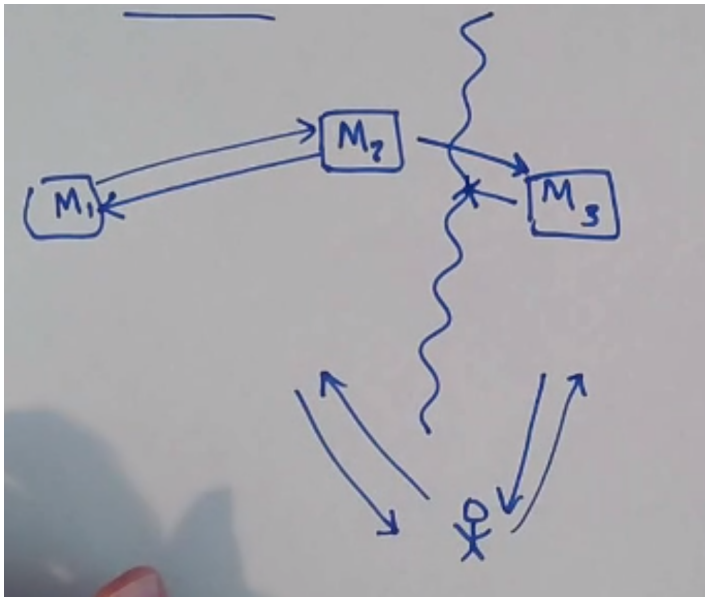
We'll see more like this in the Dynamo paper.

8.2 Misc

Terminology used in the Dynamo paper.

network partition

A network partition is a failure state where some part of a network cannot communicate with another part of the network:

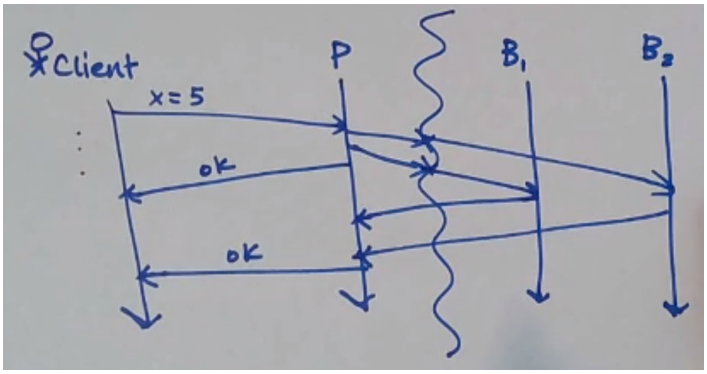


availability

perfect availability: "every request receives a response" - a liveness property

(usually there's some more to this, like "within X time" or "99th percentile", but this is a fine start)

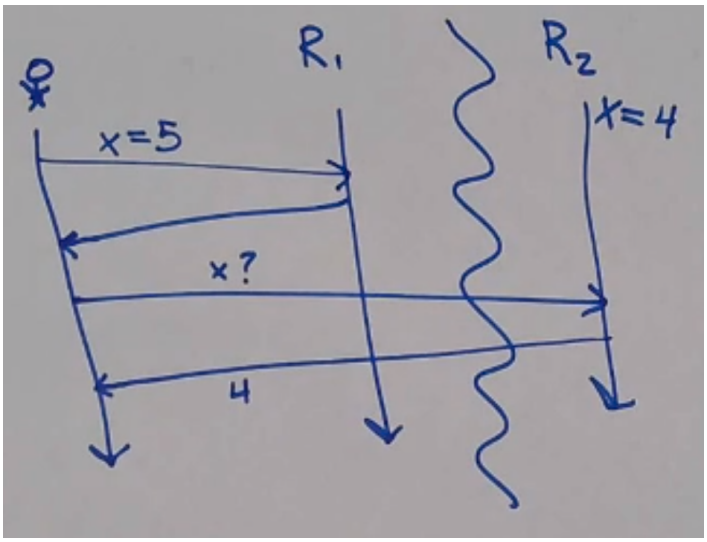
Consider the following: what if you have a network partition that prevents the primary from communicating with backups - when should the primary ACK the client?



- We could wait for the partition to heal before ACKing, but then the client could be left waiting a long time. This gives you consistency, but less availability.
- We could ACK immediately and do the replication when the partition eventually heals, but then the system is inconsistent for some time. This gives you availability, but at the cost of consistency.

Primary-Backup/CR prioritizes consistency over availability; Dynamo et al. choose availability.

If the client can talk to replicas on both sides of the partitions, though, this is bad - the first replica should probably choose not to ACK the write at all!



This tradeoff is called **CAP**:

CAP

- Consistency
- Availability
- Partition tolerance

It's impossible to have perfect consistency and availability, because of network partitions.

Note: If you happen to be reading these notes front-to-back, you should go take a look at Dynamo now.

8.3 Quorum Consistency

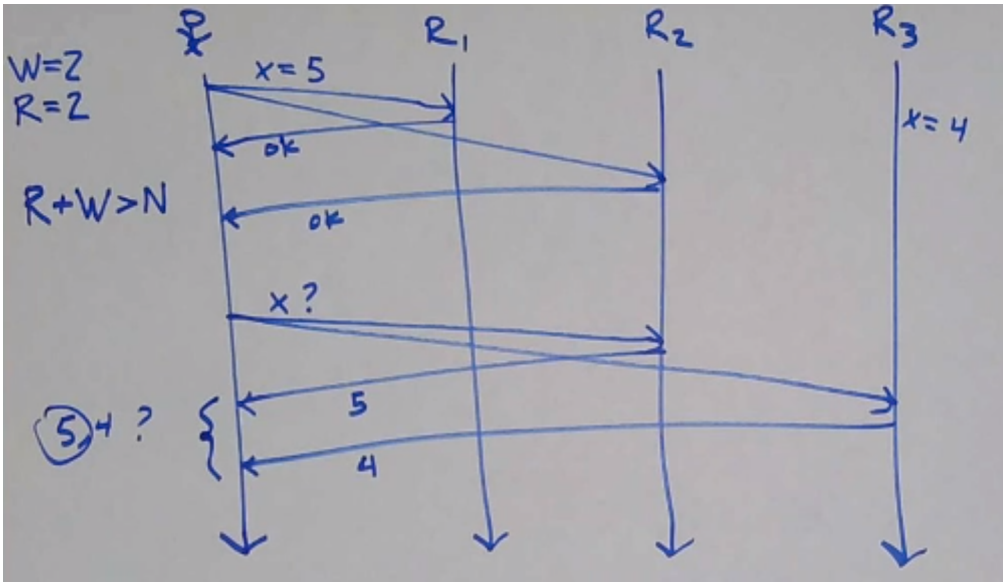
How many replicas should a client talk to? *Quorum systems* let you configure this.

- N: number of replicas
- W: "write quorum" - how many replicas have to acknowledge a write operation to be considered complete
- R: "read quorum" - how many replicas have to have to acknowledge (i.e. respond to) a read operation

Obviously, W <= N and R <= N.

Consider N = 3, W = 3, R = 1 (Read-One-Write-All: ROWA). This doesn't necessarilygive you strong consistency because replicas might deliver writes from different clients in different orders. There's other problems too, like a network partition or a failed replica blocking all writes! (Also, it's just slow to write.)

The Dynamo paper suggests, for N = 3, R = 2 and W = 2. This is so read and write quorums overlap:



In general, if R + W > N, read quorums will intersect with write quorums.

Some database systems (e.g. Cassandra) let you configure these constants. For example, if you wanted super fast writes but less guarantee on consistency, you might set W = 1.

CHAPTER

NINE

DYNAMO

This section of notes discusses concepts found in the Dynamo paper, at http://composition.al/CSE138-2021-03/readings/dynamo.pdf.

9.1 Concepts

- **availability**
  - "every request will receive a response" (quickly? 99.9% of requests?)
- **network partitions**
  - some machine(s) can't talk to other(s)
  - temporary and unintentional
- **eventual consistency**
  - a liveness property
  - "replicas eventually agree if updates stop arriving"
  - Doug Terry, Bayou, 1995
- **application-specific conflict resolution**
  - the client can resolve conflicting states (e.g. merging shopping carts)
  - dynamo: deleted items can reappear in a cart (bug)
  - dynamo: if client has no specific conflict resolution, last write wins (LWW)

Consider the problem of adding items to a shopping cart: writes commute, which implies strong convergence (replicas that deliver the same set of requests have the same state).

9.2 Disagreements

Dealing with replicas that disagree: the first step is to find out that some replicas disagree.

Dynamo has 2 concepts to deal with disagreements: anti-entropy and gossip.

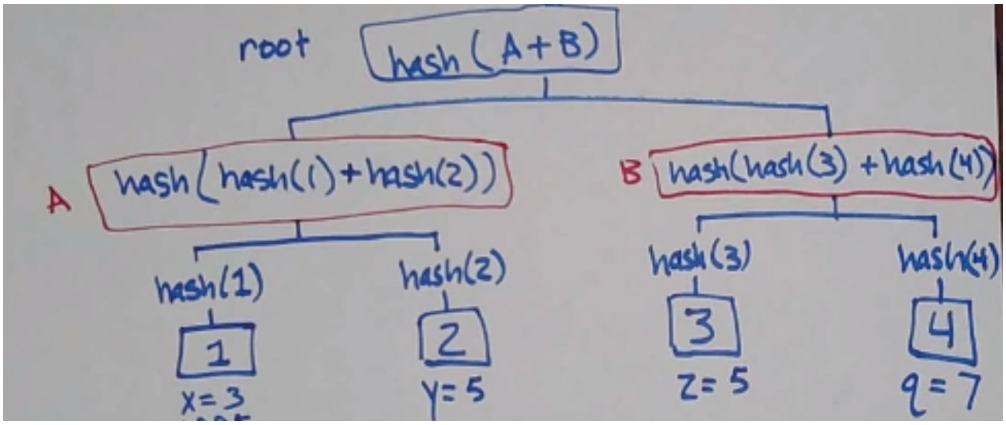
`anti-entropy`
: resolving conflicts in application state (a KVS)

`gossip`
: resolving conflicts in view state / group membership (what nodes are alive?)

Note: In general, anti-entropy and gossip are synonymous, but they have differing meanings in Dynamo.

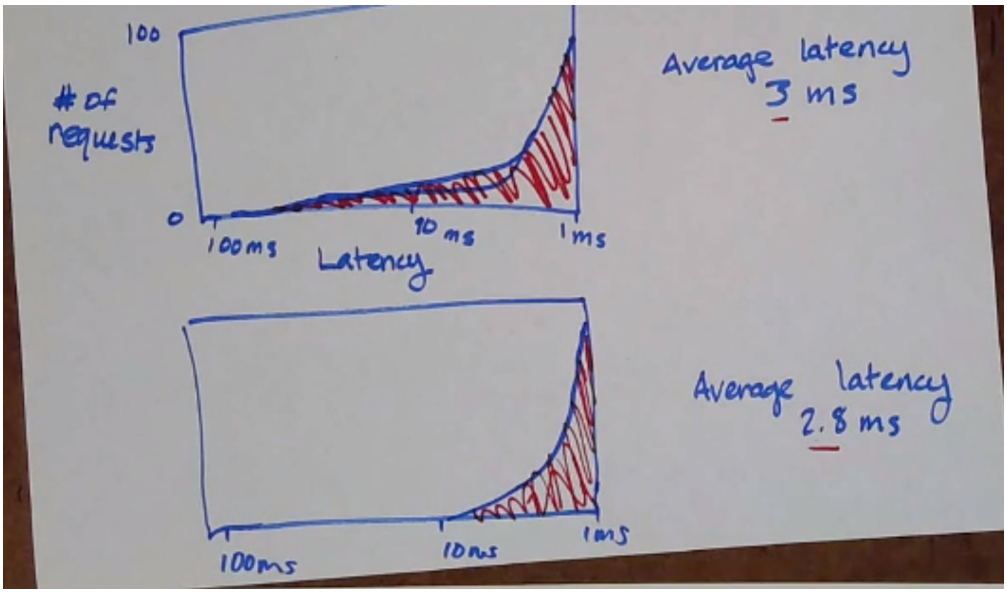
9.3 Merkle Trees

In Dynamo, KVS states are *large*, but the view state is generally small. To compare KVS states without sending the entire state, Dynamo uses *Merkle trees* (hash trees) - each leaf is a hash of a KVS pair, and each parent is a hash of its children. Two replicas can then compare their root node - if it's equal, all leaves must be equal; otherwise, compare the direct children and so on.

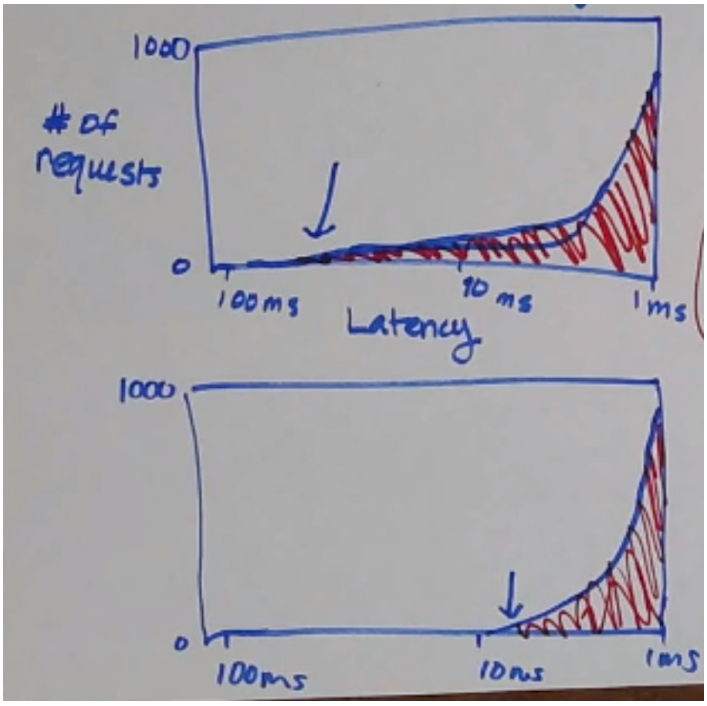


9.4 Tail Latency

Recall: latency = time between the start and end of a single action



But the top one is worse than the bottom! Tail latency examines the latency of the worst requests (e.g. the 99.9th percentile).



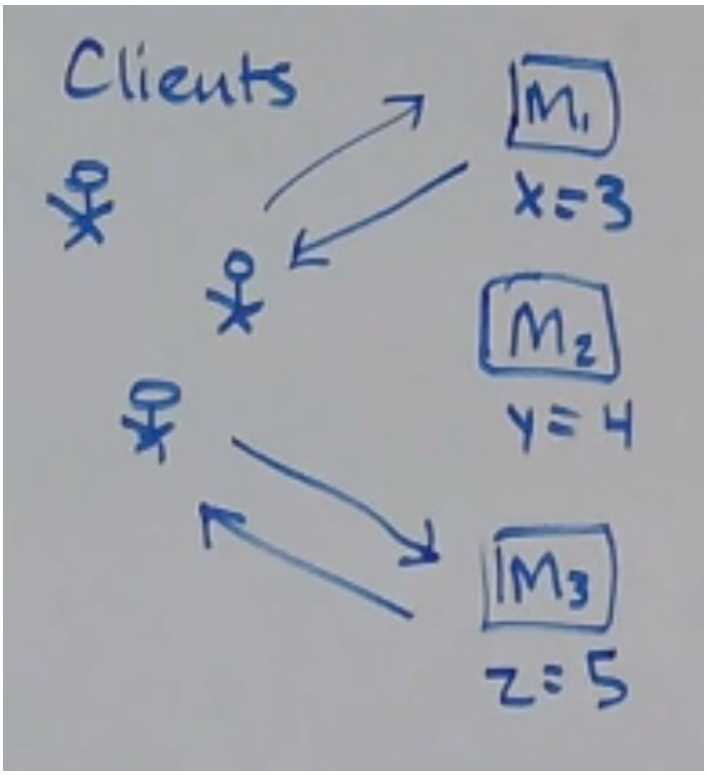
Dynamo has very good tail latency.

9.5 Sharding

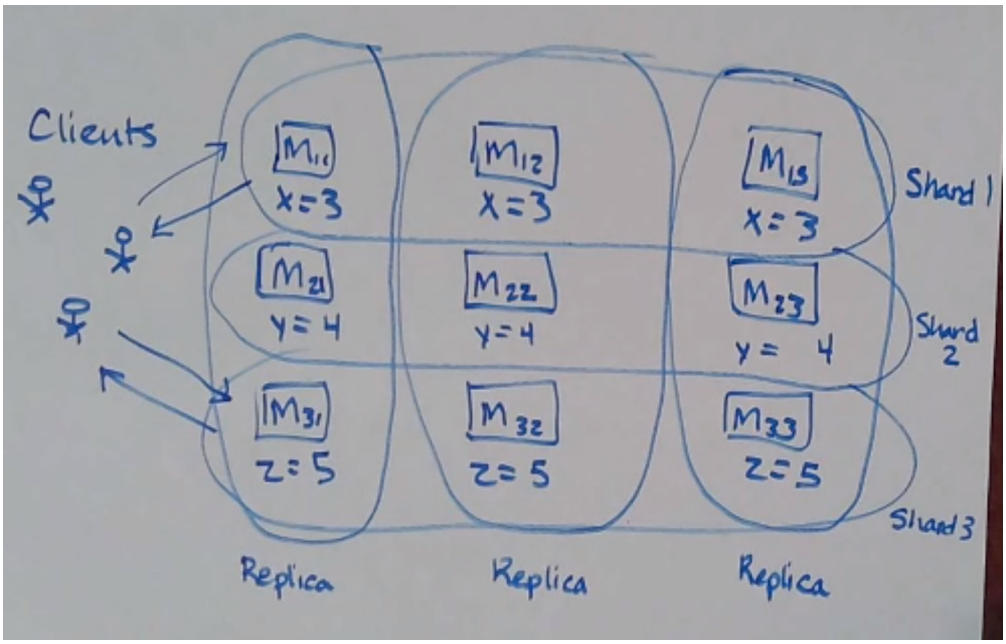
What's wrong with the approach where *everyone* stores *all* the data?

- what if you have more data than fits on one machine?
- if everyone stores all the data, consistency is more expensive to maintain

Let's store different partitions of data on different machines instead!



But now you lose fault tolerance… so let's replicate it some!



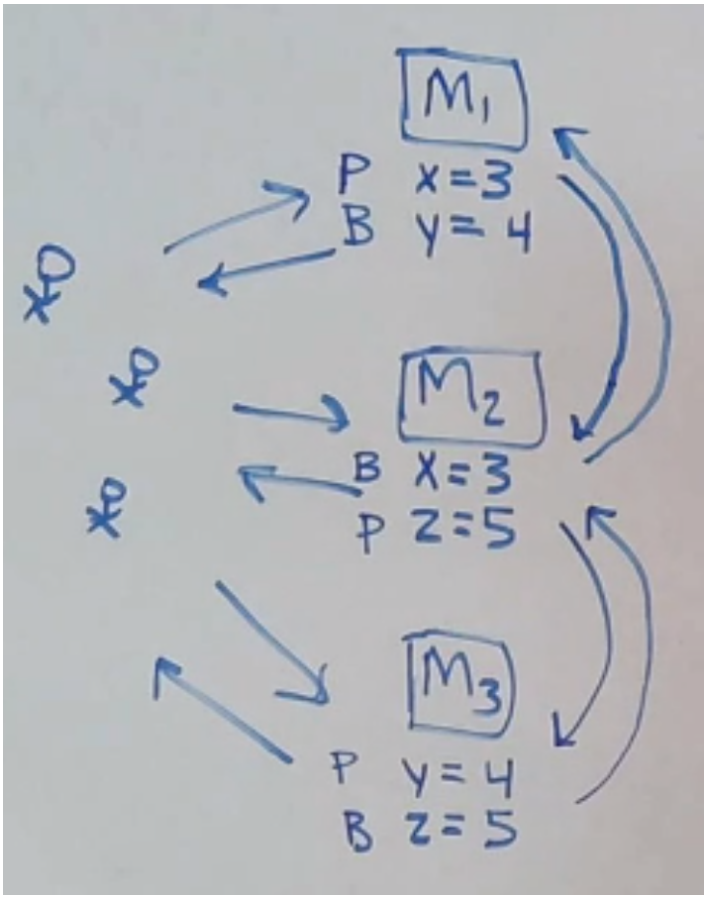
This is called *data partitioning* or *sharding*. Sharding gives you:

- increased capacity
- increased throughput

Note: Choice of replication is usually orthogonal to how you choose to shard your data; in the notes below, we focus

only on sharding and not replication.

Let's consider a different sharding setup: where each machine acts as a primary for some part of the data, and the backup for some other set of data!



Now you have more than one shard per node.

9.5.1 How to Shard

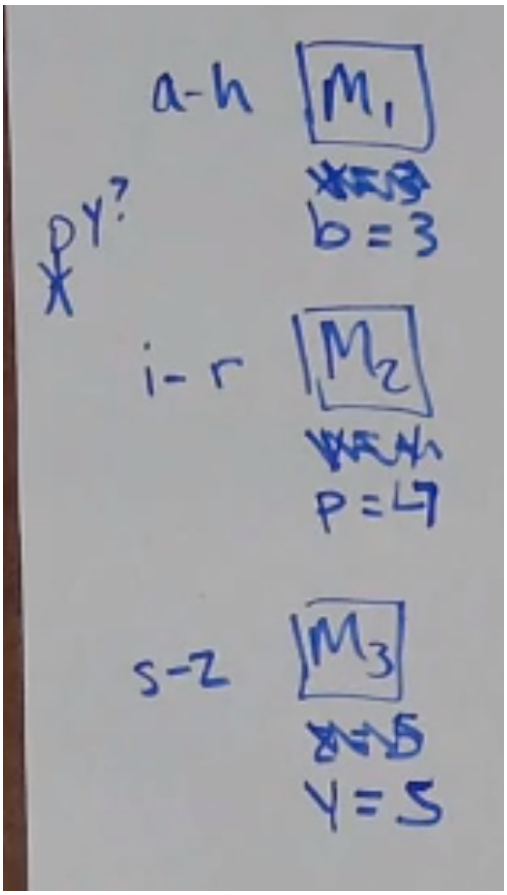
How do we choose how to split up our data among the shards?

What makes a sharding strategy good?

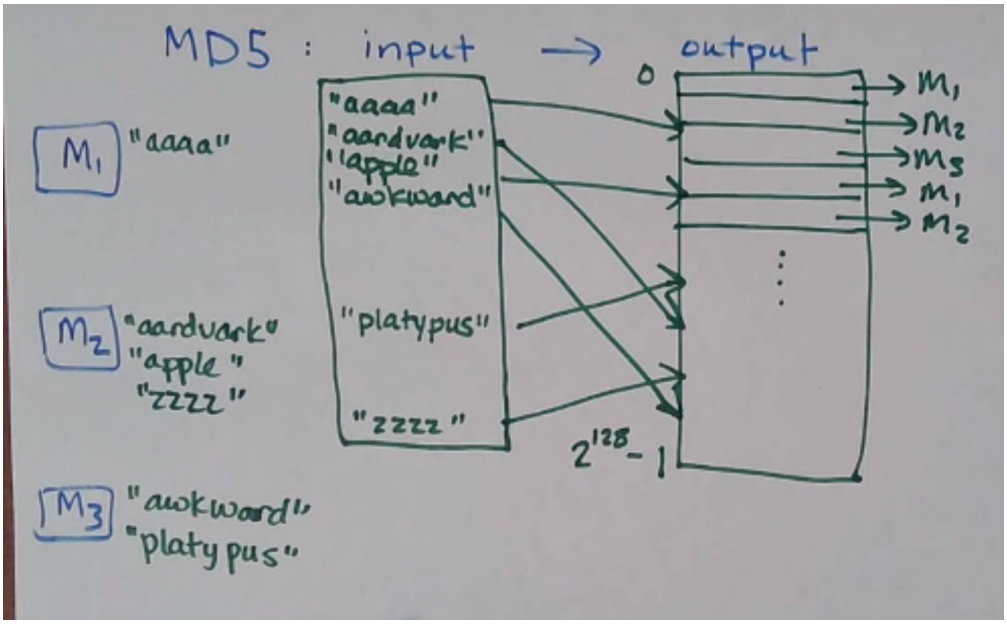
Consider the following strategies:

- **randomly assigning data to nodes**
  - data is evenly distributed
  - it's hard to find where a specific piece of data is
- **assign all data to one node**
  - easy to find where a specific piece of data is
  - data is very unevenly distributed
- **partition by key range**

- easy to find where a specific piece of data is
- data can be unevenly distributed if keys are not evenly distributed
- we can make it more uniform by *hashing* the keys (and modulo num of shards)



key range partitioning



key hashing

But partitioning based on hash(key) % N causes problems… what if you add (or remove) a machine? Now everything is off, and you have to rebalance the whole keyspace across the new nodes! It might not just be moving data to the new machine - you have to move data between the old machines, too.

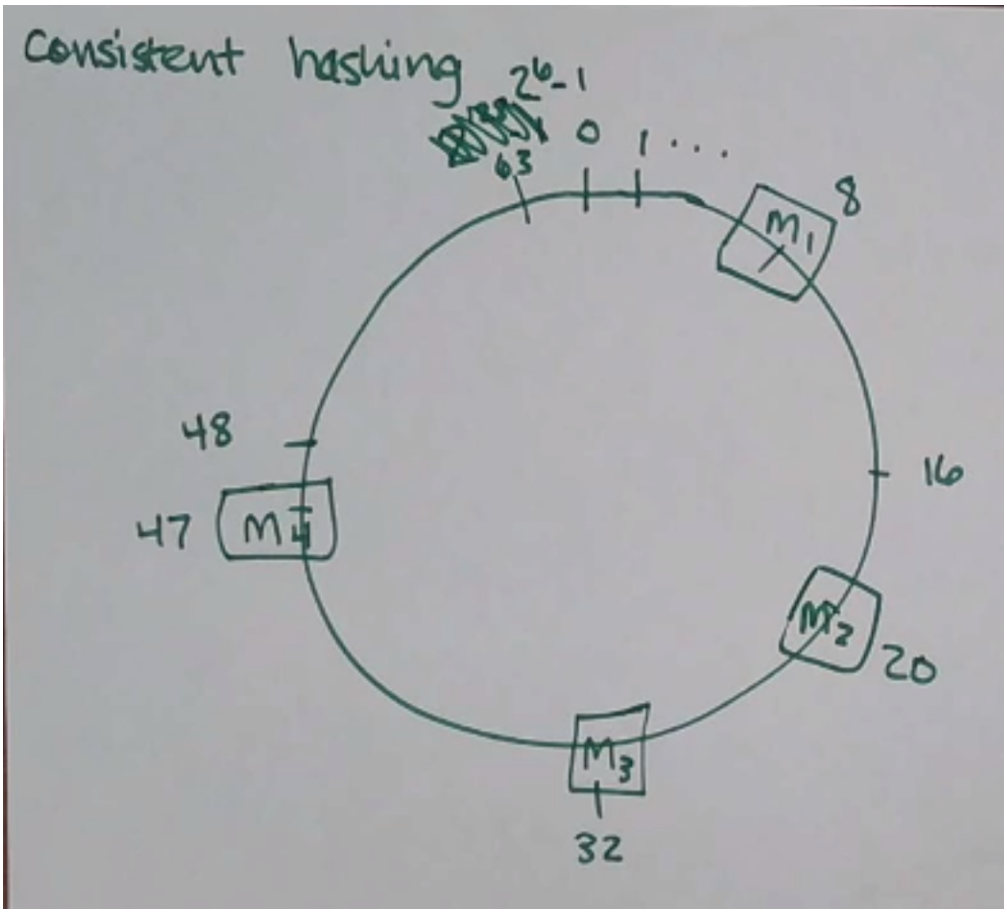
Drawback: too much data movement when N changes.

How do we move as little data as possible to achieve an even split? It turns out that the minimum movement possible is $\frac{K}{N}$, where K is the number of keys and N is the number of machines.

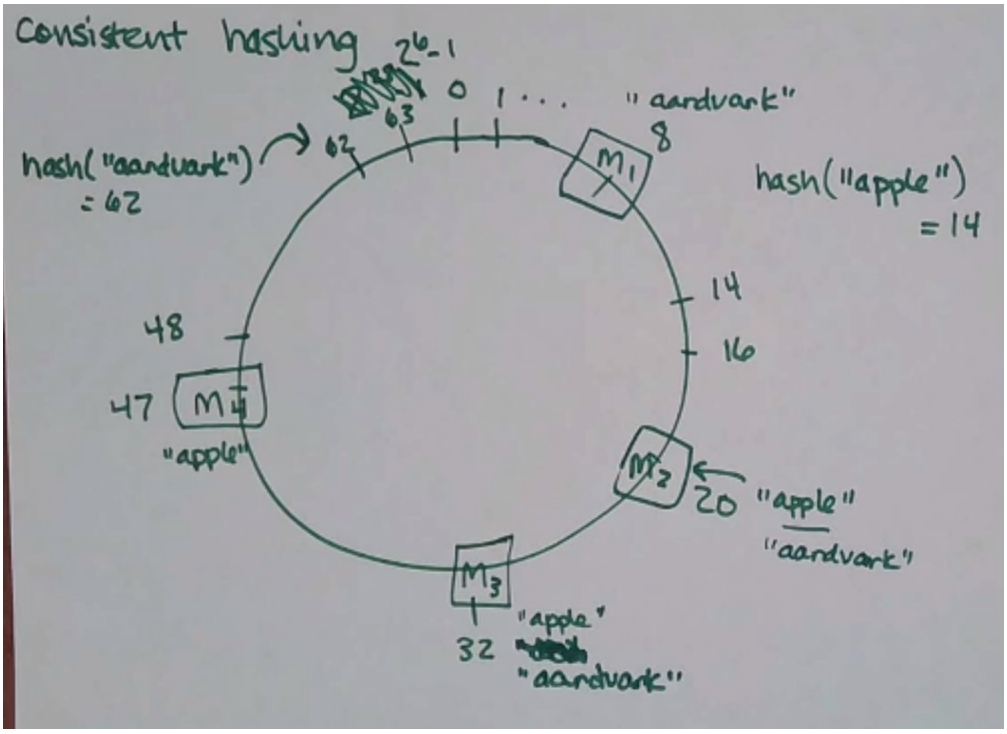
We use a mechanism called *consistent hashing* to get this minimum movement!

9.6 Consistent Hashing

Think of the hash space as a ring, and each node has a position on the ring:



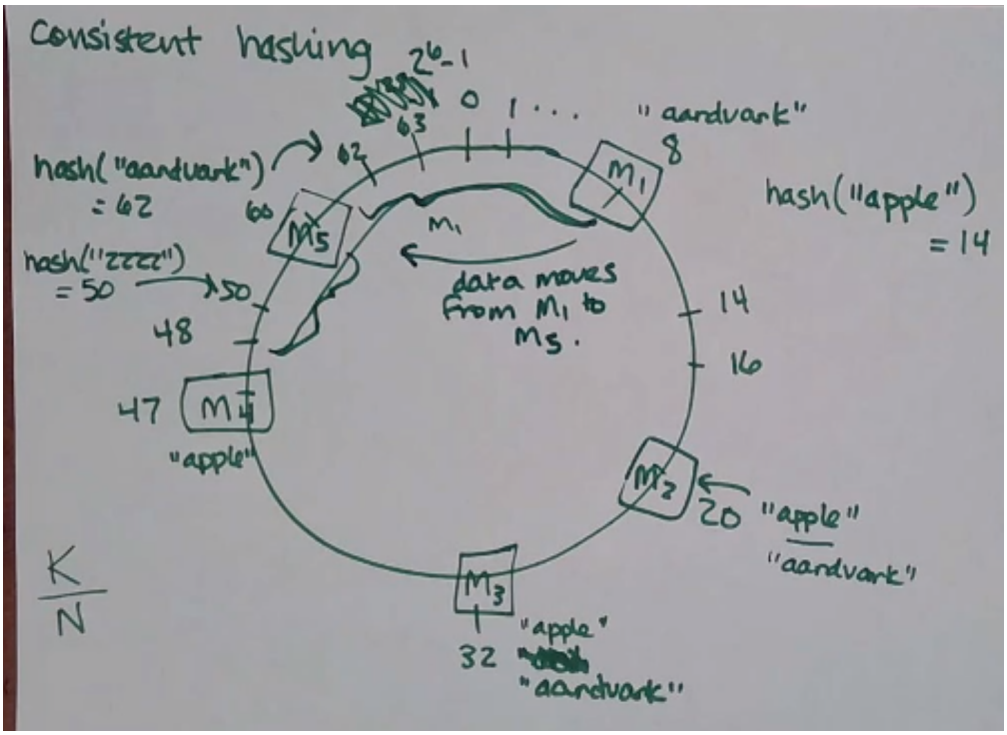
Now, we want to add a data point. We hash the key and it gives us a point on the ring - we then walk clockwise to find the next server on the ring. The key lives on that server! Additionally, to do some replication, the key lives on the next couple servers along the ring, too:



(in Dynamo it's a little more complicated, each key has a preference list that is greater than the replication factor)

9.6.1 Adding Nodes

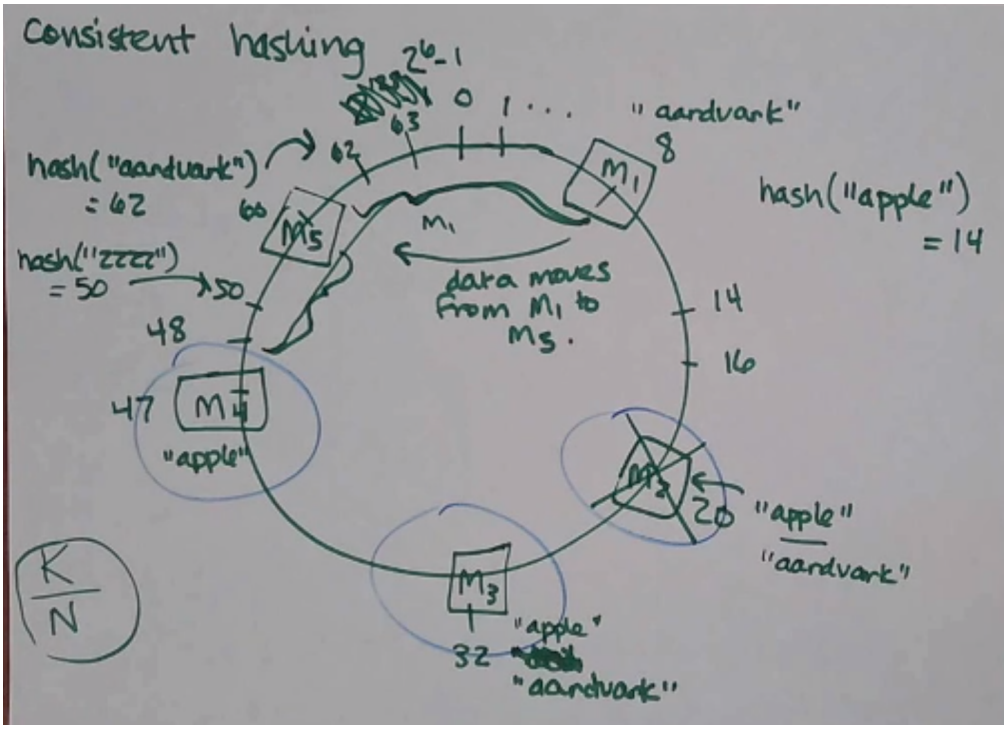
Now, when we add a new node, we only have to move the keys along the ring in the range between the previous node and the new node (e.g. adding M5 below moves the keys between 48-60 to M5 from M1).



On average, you only have to move $\frac{K}{N}$ pieces of data, and only 2 nodes (the added one and the one after it) are affected!

9.6.2 Crashing Nodes

What if a node crashes? For example, if M2 crashes, M3 becomes responsible for M2's range, and M3 becomes responsible for any keys that hash to 9-20 as well:



But other nodes are unaffected! And in Dynamo, since we replicate keys past a key's home node, M3 just picks up all the keys left homeless by M2's death!

9.6.3 Oh No

What could possibly go wrong?

- cascading crashes: a machine doesn't have the capacity to pick up the slack caused by another crashing and crashes too
- **the nodes aren't actually evenly spaced**
  - dynamo fixes this by hashing to multiple places on the ring ("virtual nodes"), instead of just one
  - virtual nodes also lets you add more load to machines with more capacity (give it more vnodes)
  - but using virtual nodes creates a bunch of little places along the ring that get affected when a physical node crashes (which may be a good or a bad thing - more nodes have to take action, but less chance of cascading crash)
  - you also have to be careful that not all your replication to other virtual nodes end up on the same physical node

CHAPTER

TEN

HETEROGENEOUS DISTRIBUTED SYSTEMS

guest lecture, Cyrus Hall

- Heterogeneous systems are the natural outcome of complex products
- Such products have diverse features, making composing them inherently difficult
- Products aimed at humans have more diverse concerns and different priorities

10.1 First-Order Distributed Systems

Classical distributed systems

- Multiple processes working toward some common goal
- Distributed, meaning, don't share a memory space
- Have some notion of time (eg. vector clocks)
- Acceptance decisions (what to do with messages, are decisions accepted)
- Trouble scaling to WAN

more extreme distributed systems

- Often never actually consistent
- Local decisions are often greedy
- Example is BGP (border gateway protocol) routing in the internet
- there is a singular purpose to the collection of processes
- no common memory space
- all processes run the same algorithm (some may run different parts)
- ex. Paxos, Raft, Dynamo, BGP, TCP, bittorrent, etc

10.2 Heterogeneous

Heterogeneous distributed systems are composed of more than one first-order system!

- Often made up of distributed and non-distributed systems
- behaviour determined by the behaviour of the involved 1st order components, and how they're composed
- system composition is primarily a human task
- interaction between systems leads to complex failure modes

Systems are diverse, so it might be hard to compose them together:

- different purpose/goals
- frameworks: languages, APIs, protocols, service discovery framework, etc
- network location
- configuration

Differences in these properties lead to differences in behaviour. Failure, error states, "correctness", etc are all expressed differently!

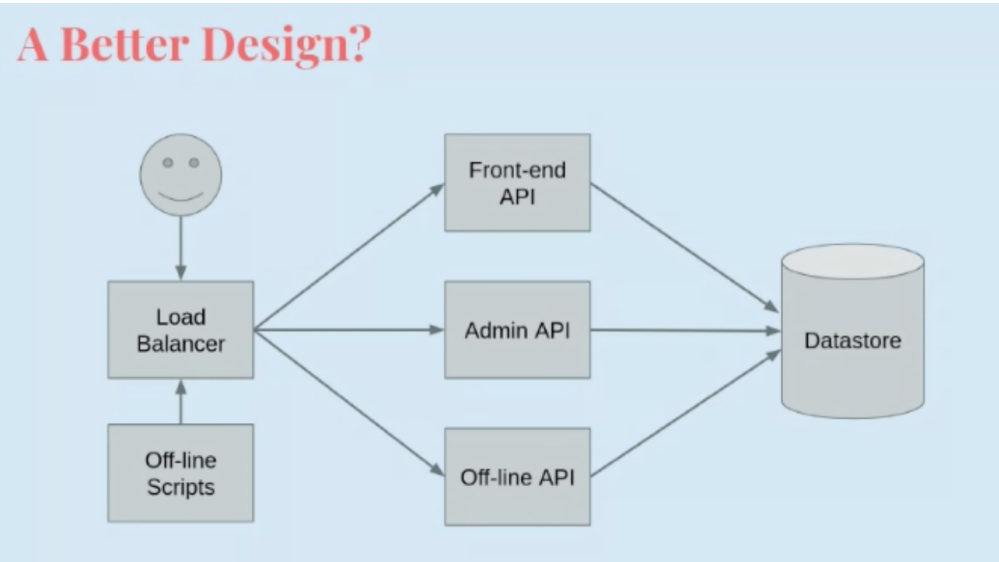
10.3 Monolith

Lots of web services are monolithic - everything is all shoved into one service!

The service wants to provide multiple interfaces with different requirements ("multi-API syndrome"), though, for example:

- frontend: on-line tasks, low latency required
- background aggregarion: off-line tasks, high latency is fine
- admin: who even knows lol

So why don't we split those out into 3 different APIs?



10.4 Problems

This still has problems, though:

In the backend:

- datastore is a single point of failure for all APIs
- variable request patterns: mix of latency sensitivities, spiky traffic patterns
- **resource management is distributed**
  - each instance of each API is managing its request rate to the datastore
  - **maybe manage request load from each API in datastore layer?**
    - won't scale as number of services grow
    - an API later can still overwhelm the central datastore

In the frontend:

- load balancer is a single point of failure for all APIs
- **more issues with variable request patterns**
  - users can deny off-line scripts from running
  - more importantly, off-line scripts can deny users
- loadbalancer can ratelimit requests, but has fundamental limits (DDoS vuln)
- **privacy and security concerns**
  - is the load balancer pure or does it cache and replay results?
  - results for an offline system available to users?

TLDR: availability and resiliency are the main concerns

`availability`
: The percentage of the time a system responds

`resilience`
: The ability of a system to remain available (even in the face of spiky events)

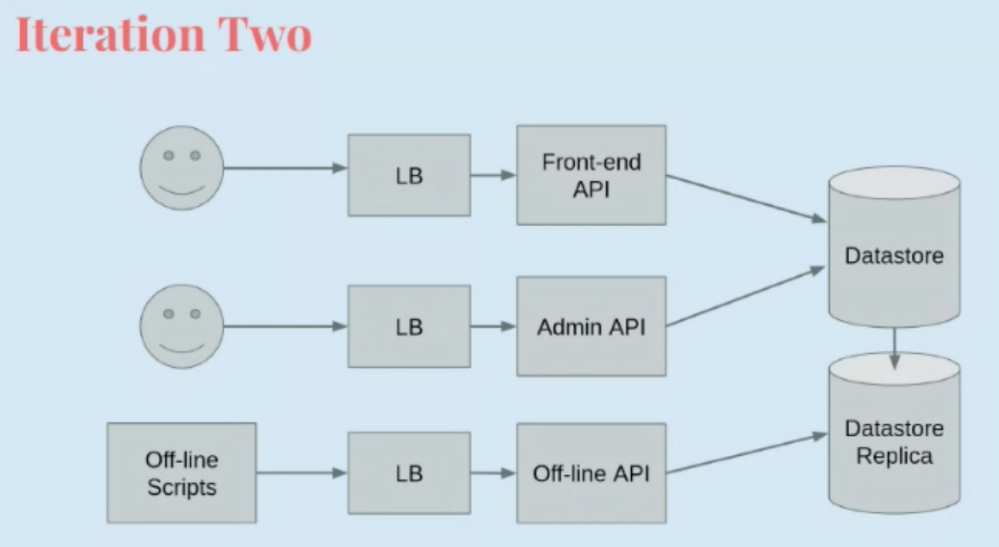
Resiliency is the product of many factors, the key elements being:

- failure isolation
- redundancy and ease of scalability
- load management

Note: Availability may *not* mean a service is properly functioning in all respects - this has concerning implications for failure detection!

10.5 Iteration 2

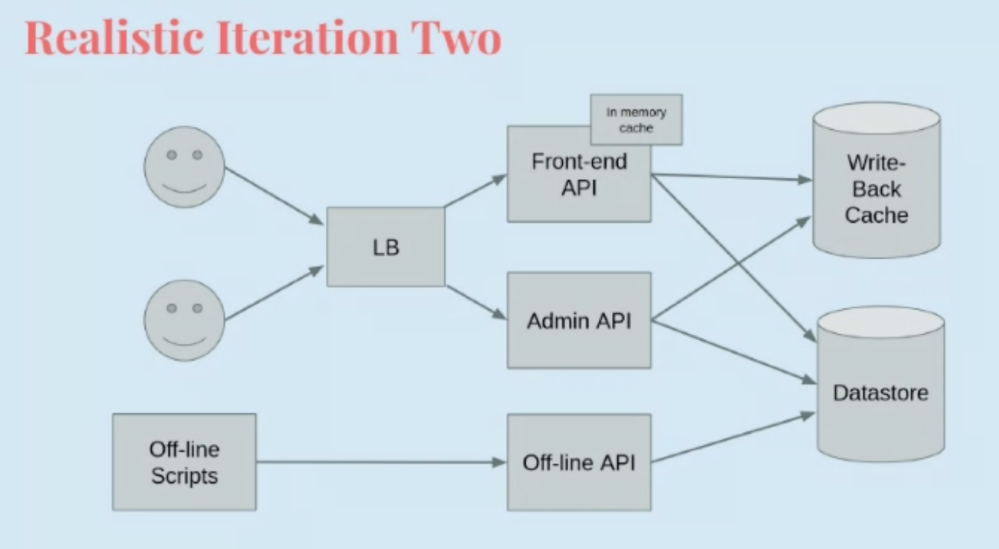
Let's iterate again:



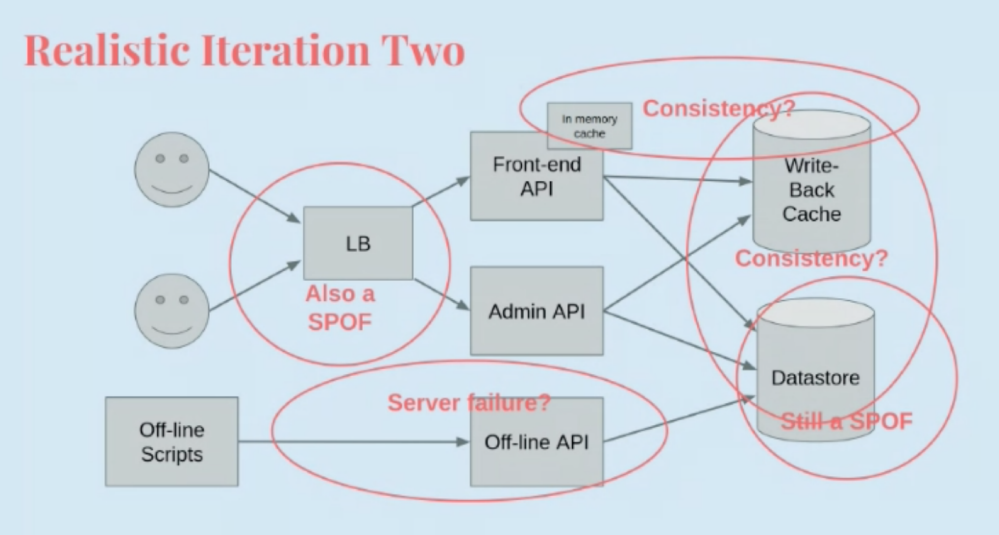
We're starting to see a *recursive service pattern*: as a product feature becomes important, the desire to isolate it grows. Eventually new features and products are just built as a new service.

This iteration looks pretty good, but the real world is not so clean!

- engineering is happening during huge growth
- ops and downtime takes up a ton of engineers' time
- product changes flying thick and fast, sales selling things that don't exist
- etc



But this design brings with it a whole host of interesting new problems:



Specifically, in addition to the SPOFs, cache consistency is really hard to keep in this model! Even just the different instances of the frontend API might have different data in the cache :(

10.6 Business

Heterogeneous systems are, in part, the result of *business* decisions. As engineers, we need to make estimates of risk and impact known; a healthy balance must be reached between eng, product, and business needs. Heterogeneous system engineering is partly about finding a sense of zen.

10.7 Ideals + Rules of Thumb

- Accept failure
- Embrace error
- If possible, be stateless
- Isolate!!!!!!

Ideally, each system and service should:

- address a single concern
- loosely couple with other systems
- be easy to maintain (and test!)
- be independently and repeatedly deployable
- have clear documentation of its behaviour and semantics

Some rules of thumb:

- **retries are dangerous (in heterogeneous systems)**
  - certain operations might not be idempotent (e.g. `INCR`)
  - a web server that tries to charge a credit card to display an order to the frontend fails (might be ok to retry IF the credit card processor guarantees that not only did it fail, it didn't already charge)

- **make stateless requests**
  - it's hard not to retry if a transaction is completed in multiple requests or it's not idempotent
  - use idempotent requests whereever possible (e.g. `SET` vs `INCR`)
- **propagate errors early**
  - if some downstream service gets an error, make the error the original requester's problem
  - lets us know the system is struggling
  - often the original request is no longer important - user has moved on or never cared
  - even better with active load control (e.g. queueing - is your dependency healthy?)
- **retries can be a necessary evil**
  - but have backups!
- **ratelimit traffic!**
  - each system has a max sustainable load peak
  - you should ratelimit to X% of the peak sustainable rate (like, 95-98%)
  - use a benchmarker to calculate this rate

10.8 Conclusion

- complex business processes and lifecycles lead to heterogeneous systems
- holding back the business is not the right tradeoff
- we can use isolation and loose coupling (and other) to mitigate complexity
- systems design is a social process

Contact: cyrusphall (at) gmail

CHAPTER

ELEVEN

MAPREDUCE

This section of notes discusses concepts found in the MapReduce paper, at http://composition.al/CSE138-2021-03/readings/mapreduce.pdf

11.1 Online v. Offline Systems

`online systems`

AKA services - e.g. KVSes, web servers, databases, caches, etc.

These services wait for requests to arrive from clients, and then handle them quickly. Low latency and availability are often prioritized.

`offline systems`

AKA batch processing systems - e.g. MapReduce

These services are focused around processing a lot of data, not necessarily for a client. High throughput is often prioritized.

Note: There is a class of system somewhere between on- and offline systems, which have to process a lot of data but also have to be responsive to the data as you get it - this class is pretty recent, and is casually called "streaming systems". They might be used in systems like Google Docs or Twitch.

How do you shard data that might be related? Partitioning by key, for example, might have one key that references another in a different shard. (e.g. in SQL)

So why MapReduce? The same underlying data might need different representations (e.g. for a data analyst) that would be really slow to generate online, but the clients don't want all their requests to be slow. Instead, we can generate the different representations offline and store a copy for our clients when they need it.

`raw data`

the authoritative version of the data - the format new data is stored in

`derived data`

the result of taking existing data (either raw or derived) and processing it somehow

One example of derived data is inverted indexes (e.g. word -> docs, where forward is doc -> words) - used in search engines to support fast lookup by word.

We could create an inverted index from a forward index using something like this:

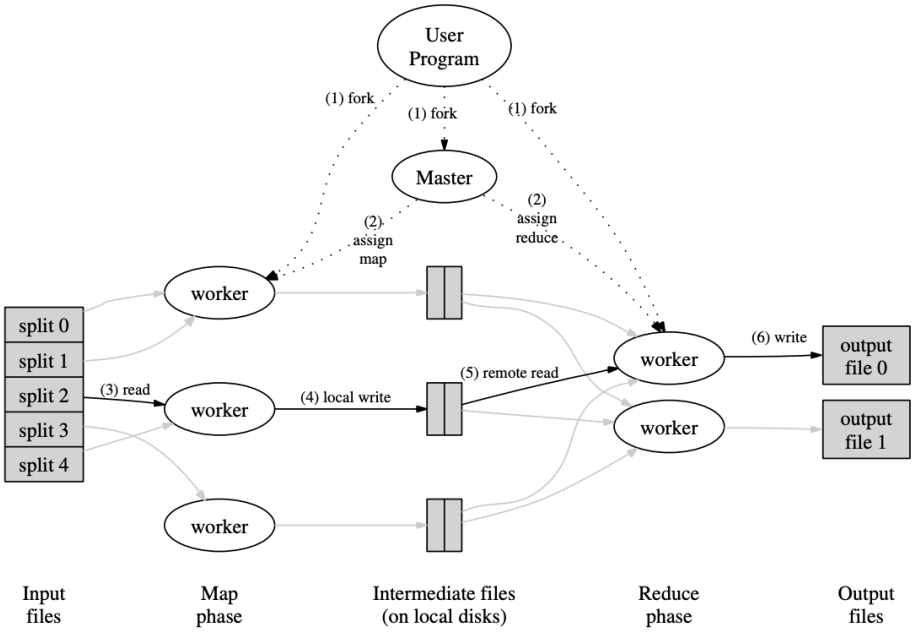
```
# step 1: map
>>> for doc in documents:
...     for word in doc:
...         yield (word, doc)

(the, Doc1)
(quick, Doc1)
# ...
(the, Doc2)
(dog, Doc2)

# step 2: reduce
>>> ind = {}
>>> for word, doc in pairs:
...     ind[word].append(doc)

{
    the: [Doc1, Doc2],
    quick: [Doc1],
    dog: [Doc2]
}
```

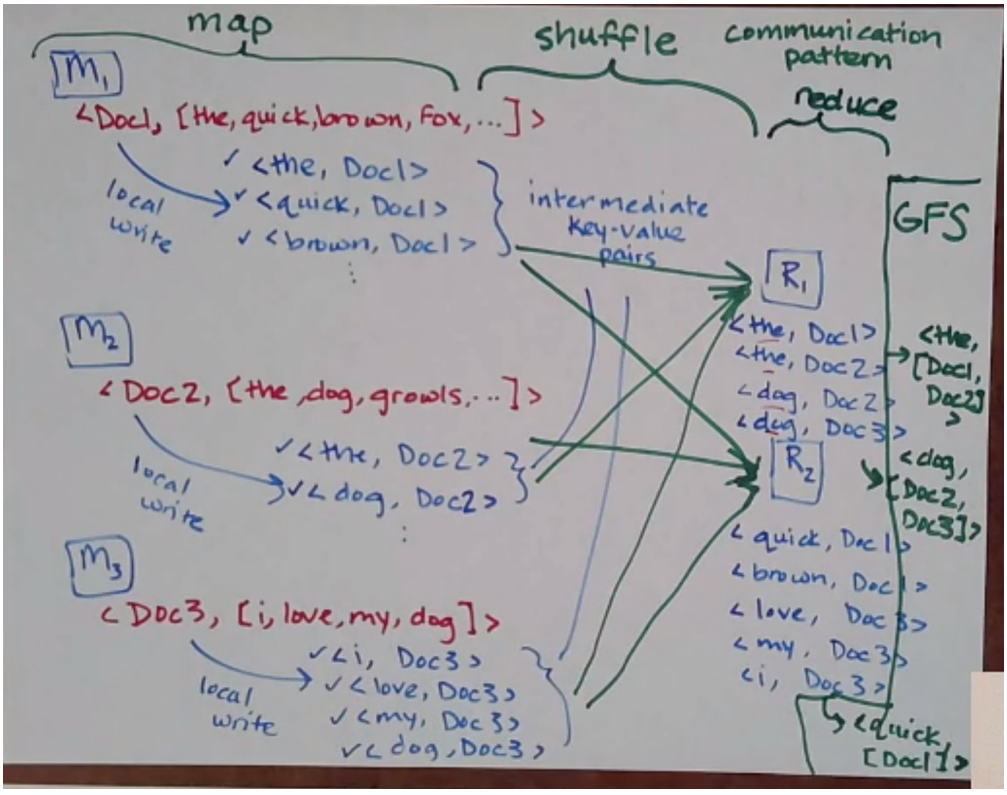
Now, this algorithm isn't particularly complicated on its own. But how do we run this at a *large* scale?

11.2 MapReduce

In MapReduce, many of the algs were conceptually simple - the hard part was running it on a huge amount of data in a distributed fashion. What if our forward index is too large to fit on one machine? What if the inverted index is?

It turns out for this problem, we can easily parallelize by distributing docs to different machines! Each machine can generate a set of *intermediate key-value pairs* from input key-value pairs and then save them locally without having to do any computation - then reducer machines can grab some set of intermediate key-value pairs (e.g. grouped by `hash(key) % R`) to solve them and save them to some file system (e.g. GFS).





Note that a lot of the work is done in the transfer of data between mappers and reducers, where reducers are reading from multiple mappers - this communication pattern is called the *shuffle* and is expensive!

This MapReduce pattern is not specific to just this problem - you can use this same framework for a whole bunch of tasks while writing very little code (the map and reduce functions)!

Note: MapReduce has an open-source clone called Hadoop, and a GFS-equivalent called HDFS.

Other tasks that MapReduce can do include:

- inverted index
- grep (regex match)
- sort
- word count
- etc.

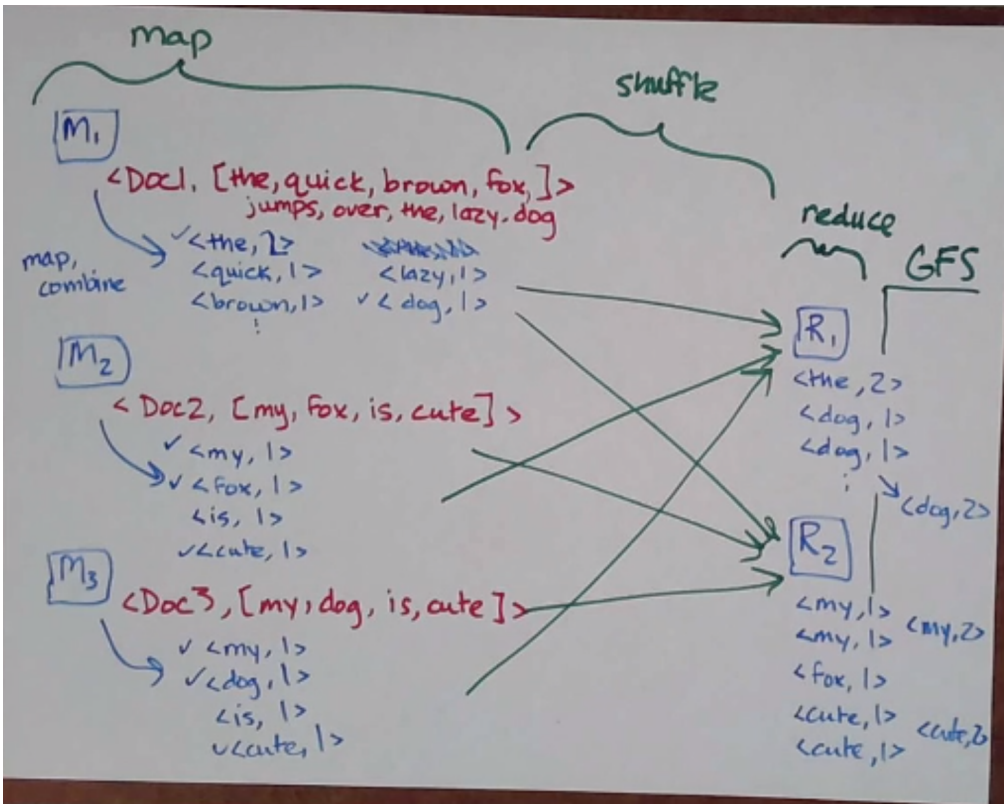
TLDR: MapReduce has 3 primary phases:

1. Map: generate a set of intermediate KV pairs from input KV pairs
2. Shuffle: intermediate KV pairs are sent to reduce workers according to some data partitioning scheme (e.g., but not necessarily, hash(key))
3. Reduce: takes a key and all of its values (drawn from intermediate KV pairs) and generates some output values

11.3 Word Count

Another example: word count - this introduces the concept of a *combiner* function, where each mapper does a little bit of the reducing so less data has to be shuffled (less bandwidth use). This is ok to do when an operation (e.g. addition) is associative!

Note: An example of a non-associative operation might be some average function.



11.4 What Could Go Wrong

MapReduce uses `hash(key) % R` to assign reduce jobs to reduce workers, even though hash-mod has problems in data replication - it's fine here because you know the data amount that needs processing, so you can set *R* up front (it's an offline system).

This differs to an online system, which might need to scale replicas in response to traffic patterns.

If a worker crashes, each task (map, reduce on a subset of data) is deterministic, so a master node can just reassign the task to a new worker. (They just redo the task since the intermediate result is stored locally instead of on, say, GFS - less overhead that way.)

11.5 MapReduce @ Google

MapReduce has limited degrees of freedom - often, you'll need to chain together multiple MR jobs to get the output you want. This involves writing to a filesystem between each job - Google uses Flume, which allows MR chaining without having to hit GFS each time.

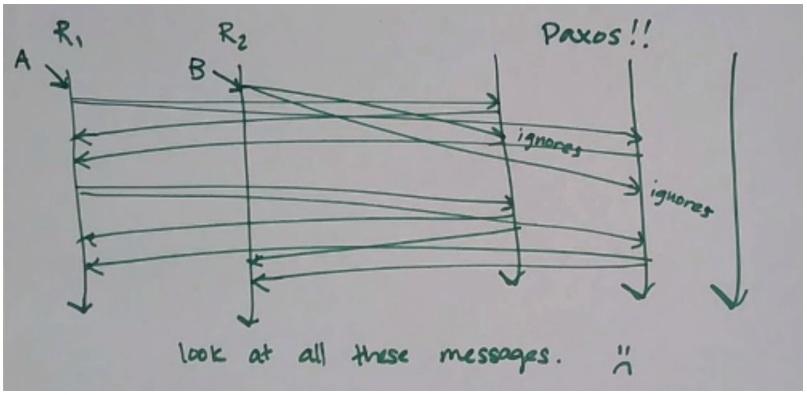
CHAPTER

TWELVE

MATH

12.1 The Cost of Consensus

Consensus takes up a *lot* of messages.



This is because the consensus algorithms we study try to approach strong consistency, regardless of whatever the content of the messages we're sending is. But in many cases, we don't need full strong consistency - just strong *convergence*.

12.2 Strong Convergence

`strong convergence`
: Replicas that have delivered the same set of updates have equivalent state.

So what's the math behind strong convergence? Recall partial orders:

`partially ordered set (poset)`
: A set S, together with a binary relation $\sqsubseteq$ that relates elements of S to each other, where $\sqsubseteq$ has the following properties:

  1. reflexivity: $\forall a \in S, a \sqsubseteq a$
  2. antisymmetry: $\forall a, b \in S, a \sqsubseteq b \wedge b \sqsubseteq a \implies a = b$
  3. transitivity: $\forall a, b, c \in S, a \sqsubseteq b, b \sqsubseteq c \implies a \sqsubseteq c$

An example of a poset is the set of all subsets of some other set, along with the subset relationship.

E.g. given the set $\{A, B, C\}$, the poset $S = \{\{\}, \{A\}, \{B\}, \{C\}, \{A, B\}, \{B, C\}, \{A, C\}, \{A, B, C\}\}$.

12.3 Upper Bounds

Ex. How many elements of S are at least "as big as" (e.g. greater than by subset relation) both of $\{A\}, \{B\}$? There are two: $\{A, B\}, \{A, B, C\}$. These are called the *upper bounds* of $\{A\}$ and $\{B\}$.

upper bound

Given a partially ordered set $(S, \sqsubseteq)$, an upper bound of $a, b \in S$ is an element $u \in S$ such that $a \sqsubseteq u$ and $b \sqsubseteq u$.

A particular pair of elements may have many upper bounds, but the interesting one is usually the smallest one.

least upper bound

An upper bound *u* of $a, b \in S$ is the *least upper bound* (aka lub, join) if $u \sqsubseteq v$ for each upper bound *v* of *a* and *b*.

Note: In our example above, every possible pair of elements has a least upper bound.

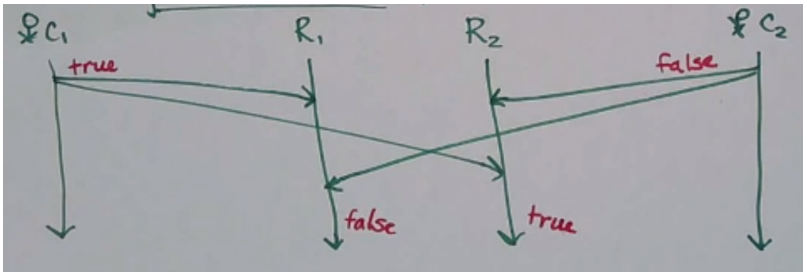
---

join-semilattice

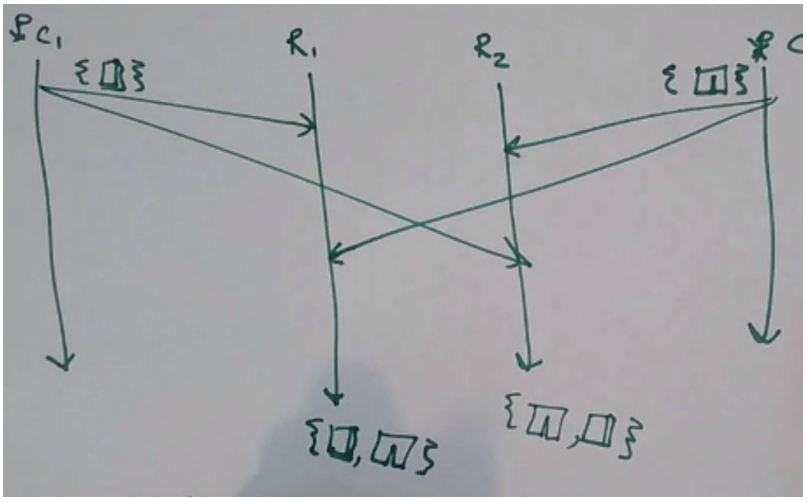
A partially ordered set in which every two elements have a least upper bound is called a *join-semilattice*.

12.4 Examples

Consider a register which can take one of three states $(true, false, empty)$, where $empty \leq true$ and $empty \leq false$. If two clients set the register to different non-empty values simultaneously, there is no upper bound, so you'd have to use some sort of consensus to resolve the conflict.

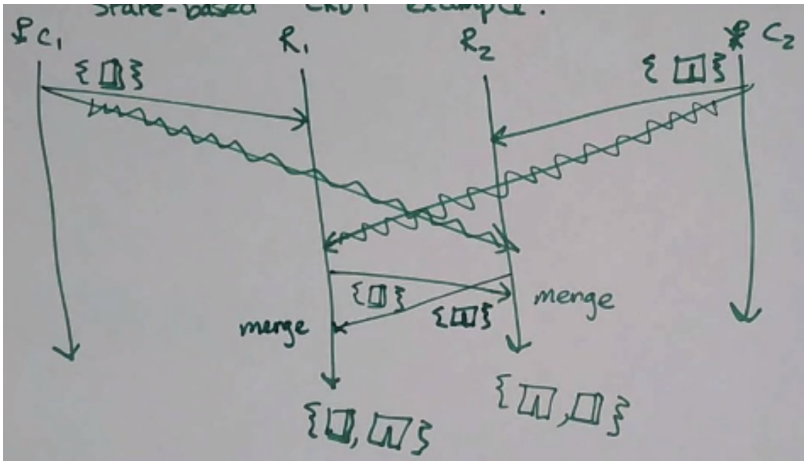


However, if the elements are members of a joined semilattice, there is a natural way to do conflict resolution.



Read more: Conflict-Free Replicated Data Types (the above is an example of a state-based CRDT).

Another example may be if each client only communicates with one replica - it's up to the replicas to share state and resolve conflicts using lubs.



The states that replicas can take on are elements of a joined semilattice, whenever a conflict comes up, you can resolve it by taking the least upper bound.

Note: This gets harder to handle when you have to deal with removing things from a set, not just adding them! One solution is to track the set of all items that have been deleted ("tombstone sets"), but this takes space…!

CHAPTER

THIRTEEN

ACKNOWLEDGEMENT

These notes are based on Lindsey Kuiper's class on distributed systems (CSE 138) at UCSC. Lecture recordings can be found here.

CHAPTER

FOURTEEN

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

A

`availability` (*built-in variable*), 53, 69

B

`broadcast` (*built-in variable*), 16

C

`CAP` (*built-in variable*), 54
`channel` (*built-in variable*), 20
`consistent` (*built-in variable*), 19

D

`deliverability` (*built-in variable*), 16
`determinism` (*built-in variable*), 36

G

`gossip` (*built-in variable*), 57

L

`latency` (*built-in variable*), 35

M

`multicast` (*built-in variable*), 16

P

`protocol` (*built-in variable*), 13

R

`resilience` (*built-in variable*), 69

S

`state` (*built-in variable*), 6

T

`throughput` (*built-in variable*), 34

U

`unicast` (*built-in variable*), 16