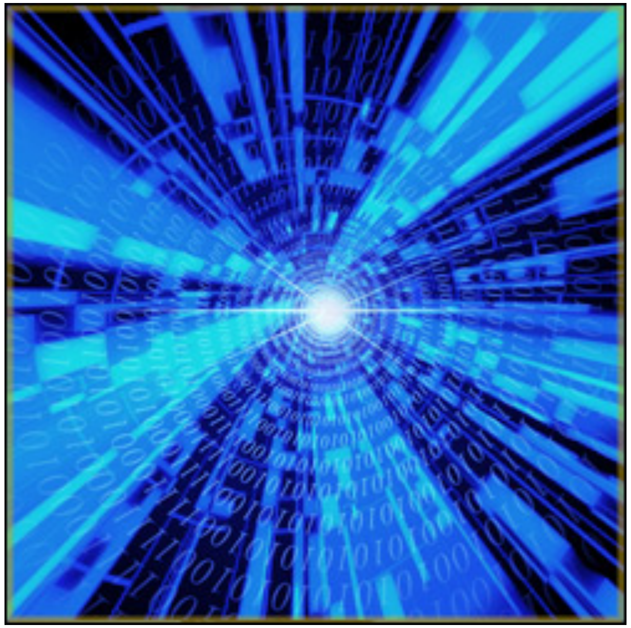
UNIVERSITY OF COPENHAGEN

Reliability: Basic Concepts and Replication

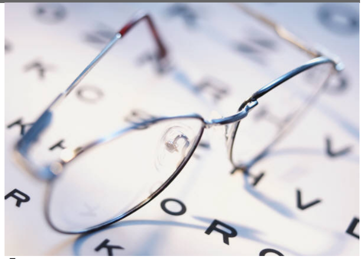
PCSD, Marcos Vaz Salles

Do-it-yourself-recap: Designing with BASE

- BASE
  - Basically-Available: only partitions affected by failure become unavailable, not whole system
  - Soft-State: partition state may be out-of-date, and events may be lost without affecting availability
  - Eventually Consistent: under no further updates and no failures, partitions converge to consistent state



- Why would we want to partition state in systems?
- Why would designing with a BASE methodology lead to systems with higher availability?
- When is it OK to compromise atomicity?



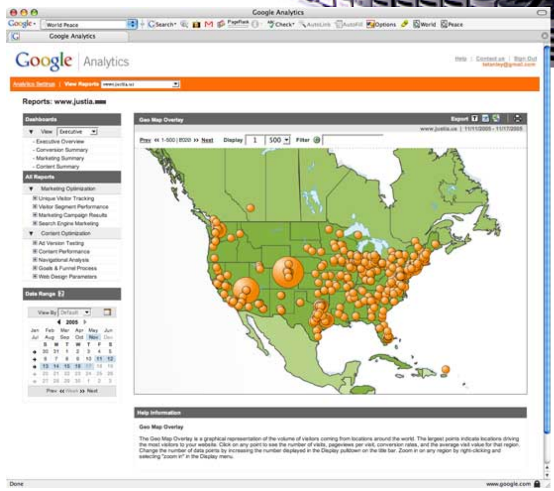
What should we learn today?

- Identify hardware scale of highly-available services common by current standards
- Predict reliability of a component configuration based on metrics such as failure probability, MTTF/MTTR/MTBF, availability/downtime
- Explain how assumptions of reliability models are at odds with observed events
- Explain and apply common fault-tolerance strategies such as error detection, containment, and masking
- Explain techniques for redundancy, such as n-version programming, error coding, duplicated components, replication
- Categorize main variants of replication techniques and implement simple replication protocols

Highly-Available Systems

- Content distribution, web, media
  - E.g., YouTube

- Data Stores
  - E.g., Amazon Dynamo, Google Megastore, F1
- Analytics
  - Deriving value from loosely structured data
  - MapReduce / Hadoop



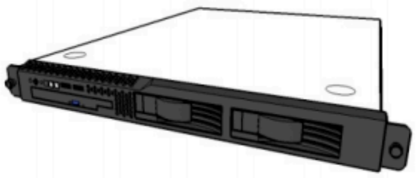
- Archival Systems
  - E.g., Oceanstore

Throughput Most Important Metric

Scale is the name of the game

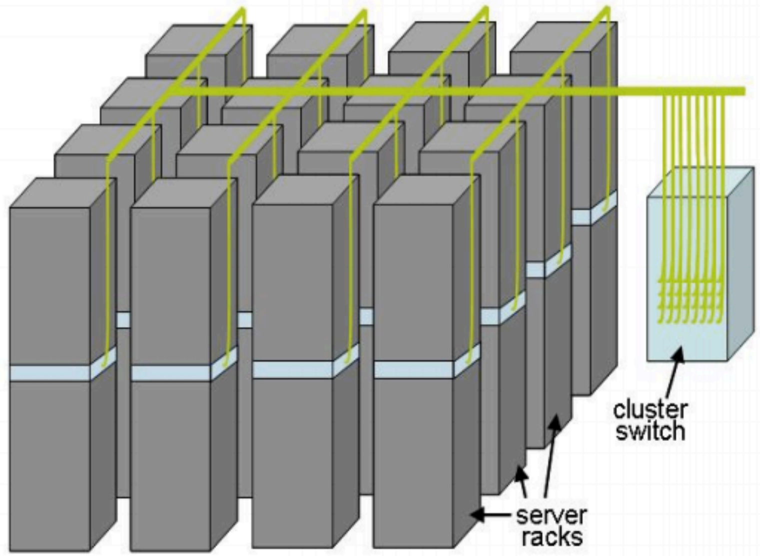
- Volumes of information
  - TBs – PBs of raw data
- Variety of schemas and formats
  - Tens – hundreds of schemas
- Large data centers
  - Tens of thousands of machines

The Machinery



Servers
- CPUs
- DRAM
- Disks

Racks
- 40-80 servers
- Ethernet switch



Clusters

Source: Dean

So, everything works?

- Assume a computer has probability of failure p
- If system needs N computers to work, what is probability of system working?

- Probability of one component working: $1-p$
- Probability of all components working: $(1-p)^N$
  - Assuming failures are independent!
  - Correlated failures are the reality – and make it even worse

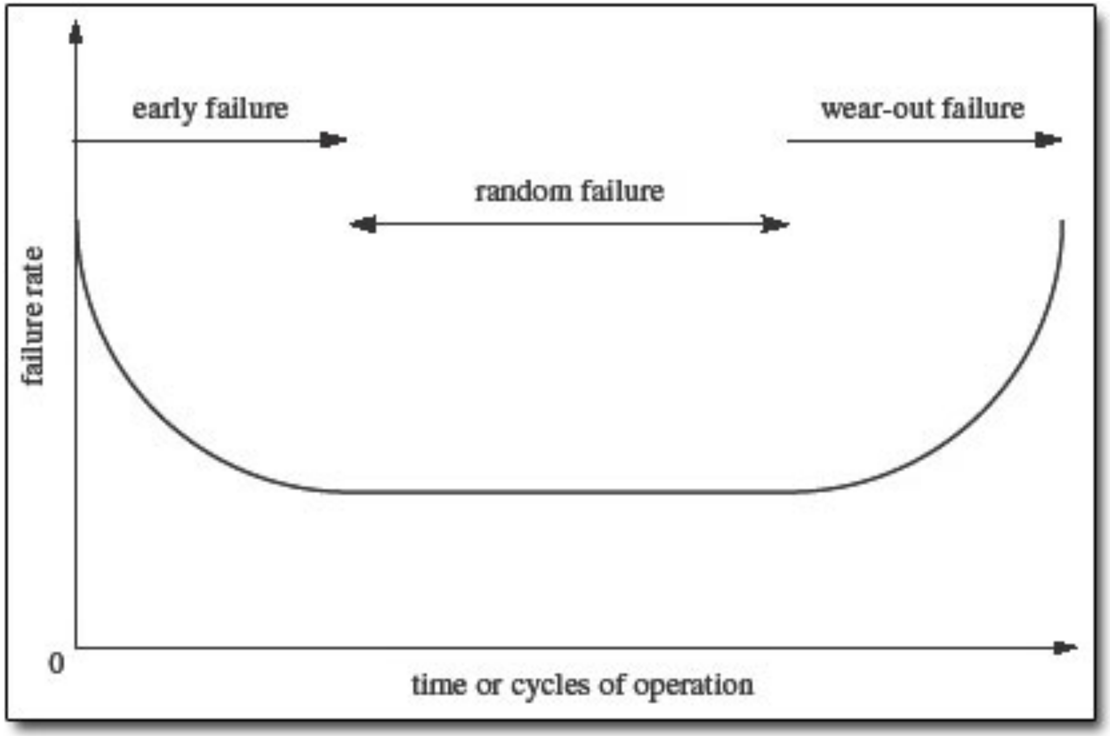
Reliability Measures

- Mean Time to Failure (MTTF)
- Mean Time to Repair (MTTR)
- Mean Time Between Failures (MTBF)
- MTBF = MTTF + MTTR
- Availability = MTTF / MTBF
- Downtime = (1 – Availability) = MTTR / MTBF

- Consider N = 10,000 and for one computer, MTBF= 30 years

How often do you estimate to see a computer failing in this scenario?

Assumptions and the Bathtub Curve



Source: The Open University

Reliability & Availability

- Things will crash. Deal with it!
  - Assume you could start with super reliable servers (MTBF of 30 years)
  - Build computing system with 10 thousand of those
  - **Watch one fail per day**

- Fault-tolerant software is inevitable

- Typical yearly flakiness metrics
  - 1-5% of your disk drives will die
  - Servers will crash at least twice (2-4% failure rate)

Source: Dean

The Joys of Real Hardware

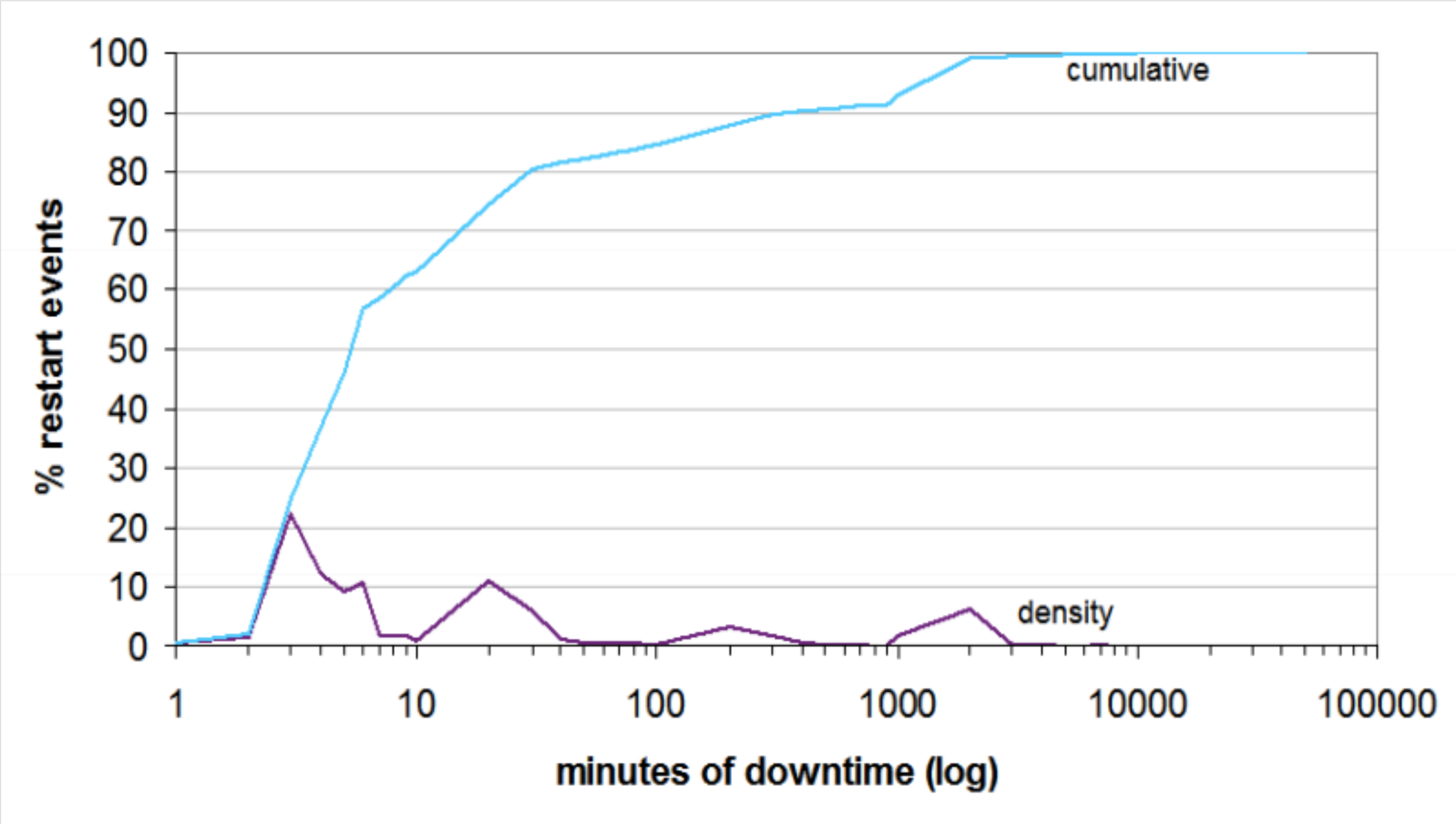
Typical first year for a new cluster:

- ~0.5 overheating (power down most machines in <5 mins, ~1-2 days to recover)
- ~1 PDU failure (~500-1000 machines suddenly disappear, ~6 hours to come back)
- ~1 rack-move (plenty of warning, ~500-1000 machines powered down, ~6 hours)
- ~1 network rewiring (rolling ~5% of machines down over 2-day span)
- ~20 rack failures (40-80 machines instantly disappear, 1-6 hours to get back)
- ~5 racks go wonky (40-80 machines see 50% packetloss)
- ~8 network maintenances (4 might cause ~30-minute random connectivity losses)
- ~12 router reloads (takes out DNS and external vips for a couple minutes)
- ~3 router failures (have to immediately pull traffic for an hour)
- ~dozens of minor 30-second blips for dns
- ~1000 individual machine failures
- ~thousands of hard drive failures
- slow disks, bad memory, misconfigured machines, flaky machines, etc.

Long distance links: wild dogs, sharks, dead horses, drunken hunters, etc.

Source: Dean

Understanding Downtime Behavior Matters



Source: Dean

Faults, Errors, and Failures

- Fault
  - Defect that has potential to cause problems
- Error
  - Wrong result caused by an active fault
- Failure
  - Unhandled error that causes interface to break its contract

Fault Tolerance

- Error detection
  - Use limited redundancy to verify correctness
  - Example: detect damaged frames in link layer
  - **Fail fast:** report error at interface

- Error containment
  - Limiting propagation of errors
  - Example: enforced modularity
  - **Fail stop:** immediately stop to prevent propagation
  - **Fail safe**: transform wrong values into conservative "acceptable" values, but limiting operation
  - **Fail soft**: continue with only a subset of functionality

Fault Tolerance

- Error masking
  - Ensure correct operation despite errors
  - Example: reliable transmission, process pairs

- We will focus on error masking next
- Main technique: **Redundancy**

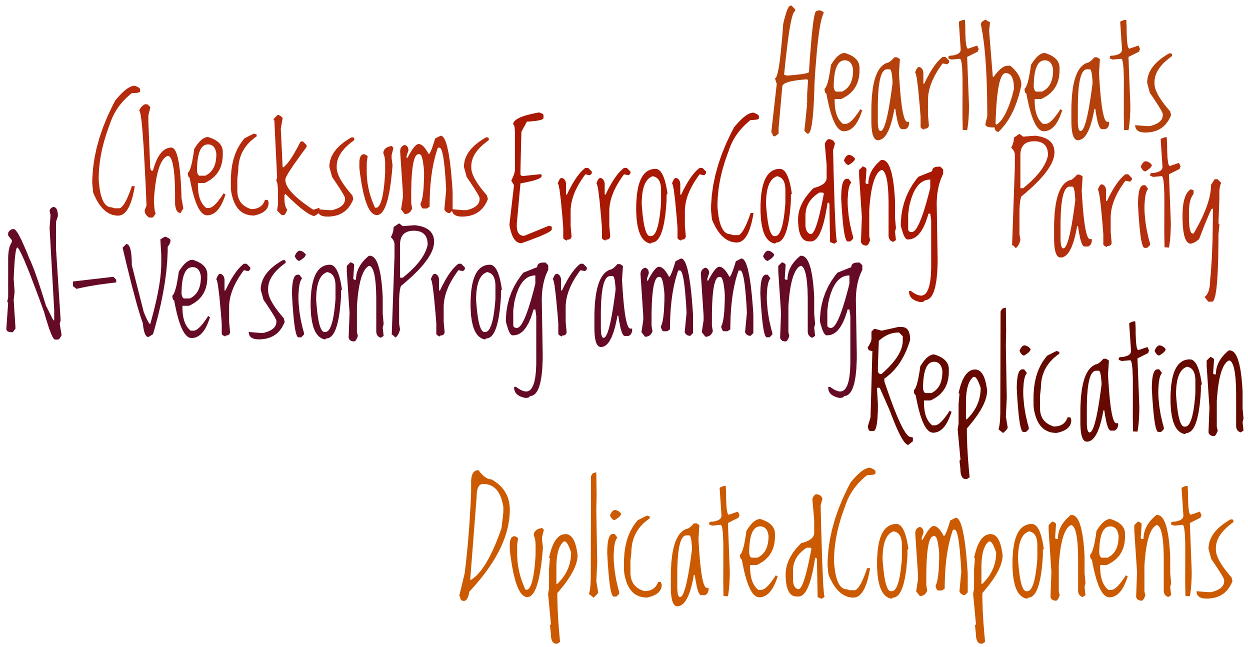
Summary

- Highly-available systems & large-scale infrastructures: design for failure
- Reliability: basic models, calculations & metrics, and reality
- Fault tolerance strategies

Questions so far?

Redundancy

- Applied to hardware, software, data



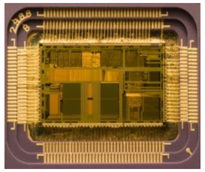
Discussion: How do these redundancy techniques work?

Replication

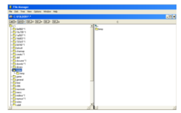
(loop (print (eval (read))))

- **MAKE COPIES!!** ☺
  - State-machine replication
  - Asynchronous replication
    - Primary-Site
    - Peer-to-Peer
  - Synchronous replication
    - Read-Any, Write-All
    - Quorums

Replicated Interpreter



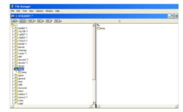
Replicated memory



- Techniques only good enough for a specific **failure model**
  - Nuclear holocaust
  - Component maliciously outputs random gibberish (**Byzantine**)
  - Components **crash** without telling you anything
  - Components are **fail-stop**

Asynchronous Replication



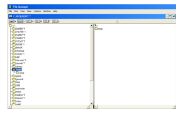


- Allows WRITEs to return before all copies have been changed
  - READs nonetheless look at just one copy
  - Users must be aware of which copy they are reading, and that copies may be out-of-sync for short periods of time.
- Two approaches: Primary Site and Peer-to-Peer replication
  - Difference lies in how many copies are "*updatable*" or "*master copies*".

Source: Ramakrishnan & Gehrke (partial)

Primary Site Replication





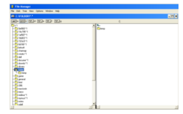
- Exactly one copy is designated the primary or master copy. Replicas at other sites cannot be directly updated
  - The primary copy is published
  - Other sites subscribe to this copy; these are secondary copies

- Main issue: How are changes to the primary copy propagated to the secondary copies?
  - Done in two steps: First, `CAPTURE` changes made at primary; then `APPLY` these changes
  - Many possible implementations for `CAPTURE` and `APPLY`

Source: Ramakrishnan & Gehrke (partial)

Peer-to-Peer Replication





- More than one of the copies of an object can be a master in this approach
  - Changes to a master copy must be propagated to other copies
  - If two master copies are changed in a conflicting manner, this must be resolved. (e.g., Site 1: Joe's age changed to 35; Site 2: to 36)

- Best used when conflicts do not arise
- **Examples**
  - Each master site owns a disjoint fragment of the data
  - Updating rights owned by one master at a time
  - Operations are associative-commutative

Source: Ramakrishnan & Gehrke (partial)

Eventual consistency

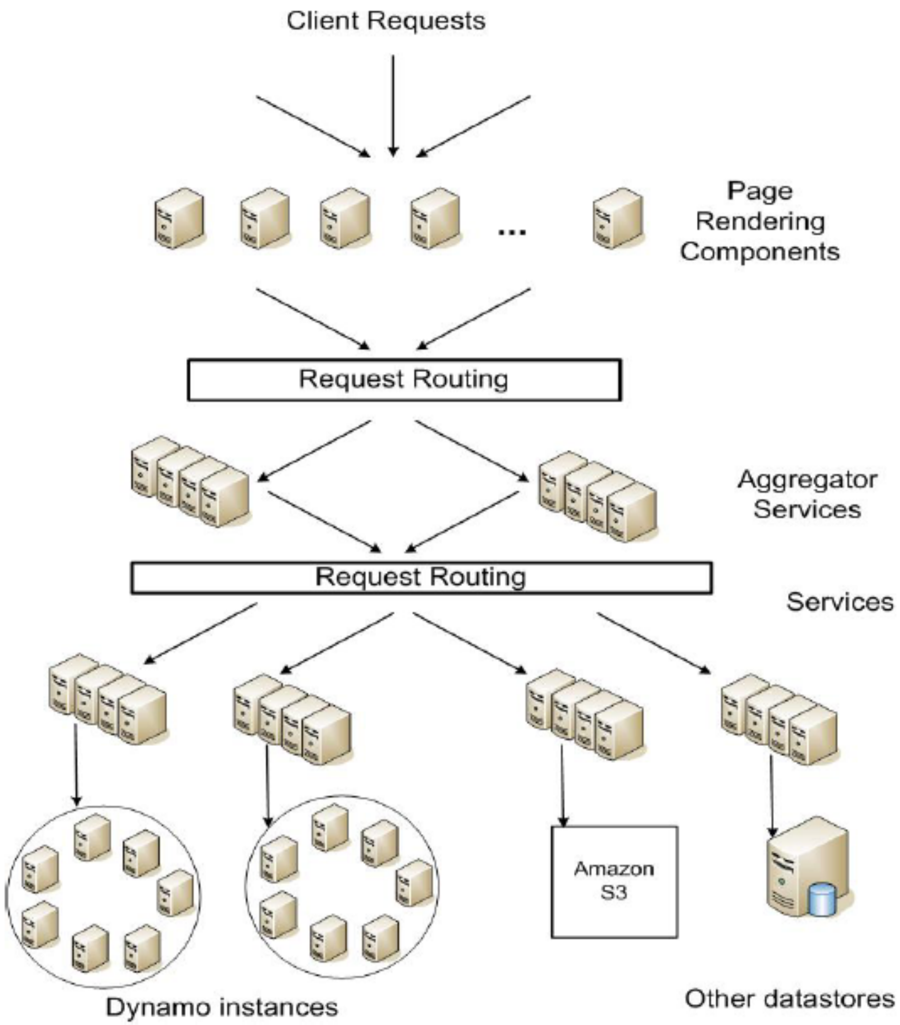
- If no new updates are made to an object, after some inconsistency window closes, all accesses will return the same "last" updated value

- **Prefix property:**
  - If Host 1 has seen write $w_{i,2}$: i th write accepted by host 2
  - Then 1 has all writes w $_{j,2}$ (for j<i) accepted by 2 prior to $w_{i,2}$

- Assumption: write conflicts will be easy to resolve
  - Even easier if whole-"object" updates only

Source: Freedman

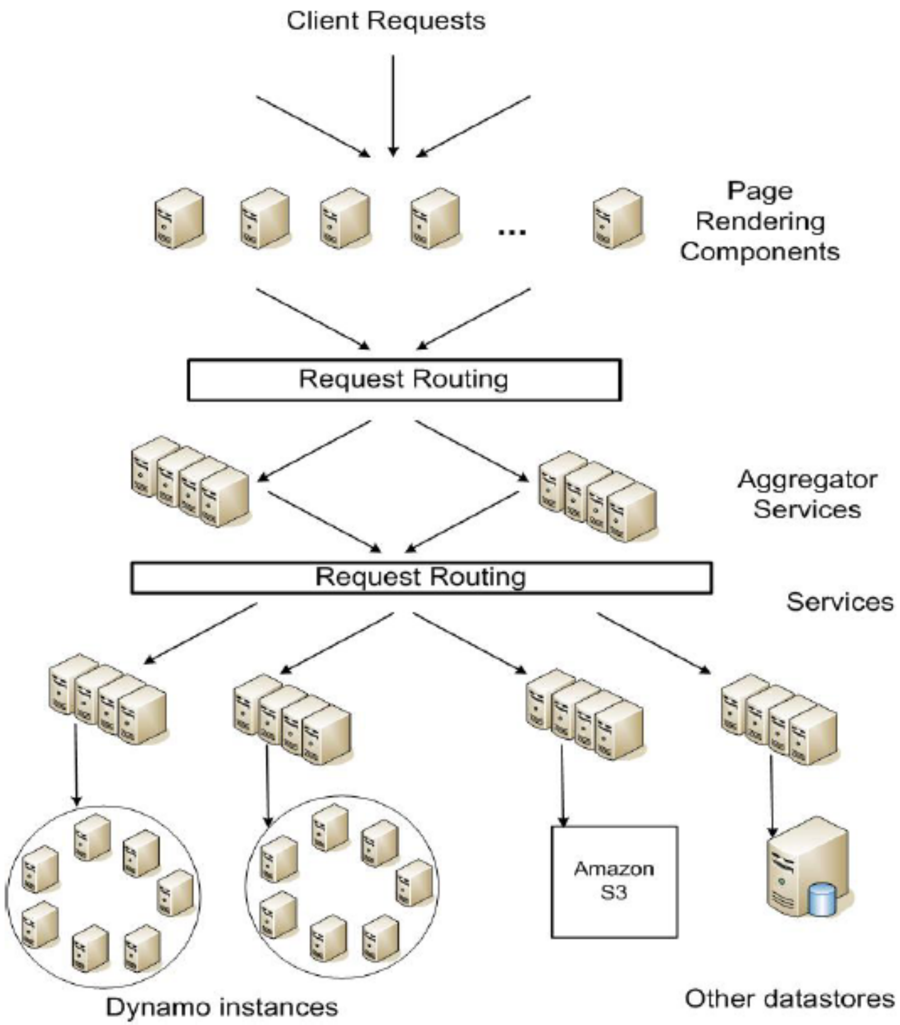
Eventual consistency & Amazon Dynamo



- Distributed, inconsistent state
- Writes only go to some subset of storage nodes
  - By design (for higher throughput)
  - Due to transmission failures
  - Declare write as committed if received by "**quorum**" of nodes
- Reads also go to subset only

Source: Freedman (partial)

Eventual consistency & Amazon Dynamo



- **Anti-entropy / gossiping** fixes inconsistencies
  - **Prefix property** helps nodes know consistency status
  - Use **vector clock** to see which is older
    - Strict precedence: take older version
    - Can't tell: ask application!
  - If automatic, requires some way to handle write conflicts
    - **Application-specific merge()** function
    - Amazon's Dynamo: Users may see multiple concurrent "branches" before app-specific reconciliation kicks in

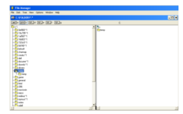
Source: Freedman (partial)

Eventual is not the only choice

- Host of other properties available
  - Beyond our scope!
- **Examples**
  - Strong consistency
  - Weak consistency
  - Causal consistency
  - Read-your-writes consistency
  - Session consistency
  - Monotonic read consistency
  - Monotonic write consistency
- See Werner Vogels' entry http://www.allthingsdistributed.com/2007/12/eventually_consistent.html for informal overview, or a good distributed systems book for algorithms ☺

Synchronous Replication





- Hide replication behind READ/WRITE memory abstraction

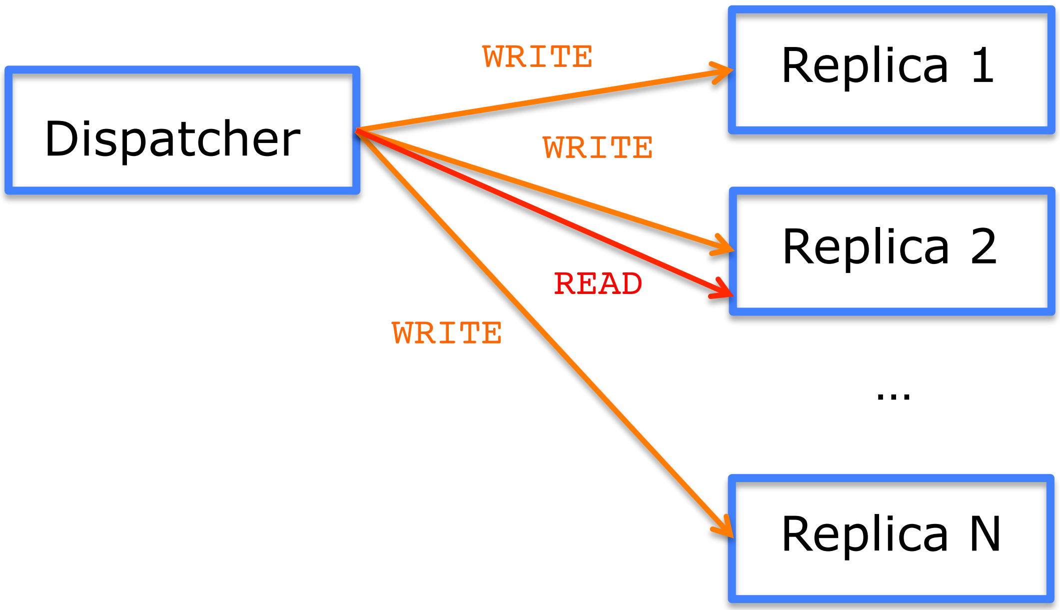
- Program operates against memory

- Memory makes sure READs and WRITEs are **atomic**
  - **All-or-nothing:** either in all correct replicas or none
  - **Before-or-after:** Equivalent to a total order

- Memory replicates data for fault tolerance

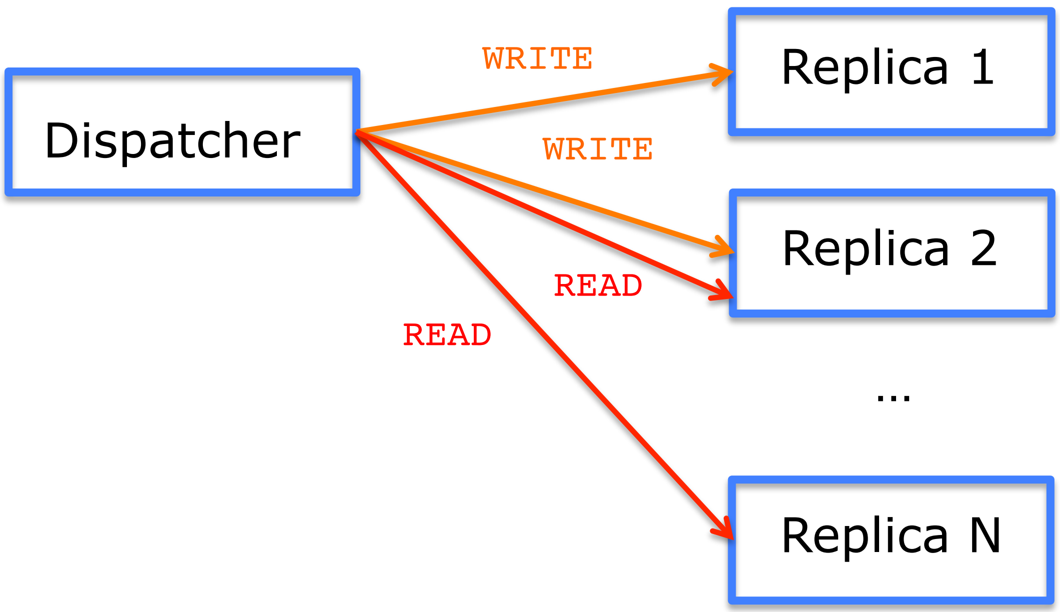
Synchronous Replication

- Read Any, Write-All
  - For now assume we have a centralized Dispatcher → state-machine replication algorithms drop that assumption!
- `WRITES` synchronously sent everywhere
- But `READS` can be answered by any replica



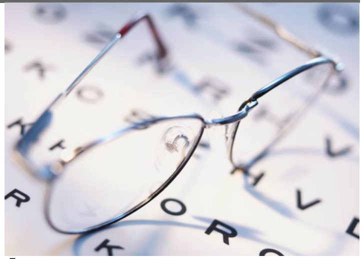
Synchronous Replication

- Quorums
  - Read Quorum (Q_r) / Write Quorum (Q_w)
  - $Q_r + Q_w > N_{replicas}$
- Reads or writes only succeed if same response is given by respective quorum
  - Read any, Write all case is $Q_w = N_{replicas}$, $Q_r = 1$



Summary

- Many techniques for redundancy
- Replication widely used technique in practice
- Many flavors
  - State-machine replication
  - Asynchronous replication
    - Primary-Site
    - Peer-to-Peer
  - Synchronous replication
    - Read-Any, Write-All
    - Quorums
  - Tons of combinations of flavors possible!
    - E.g., primary-site + synchronous
  - Tons of variations in implementation according to failure model!
    - E.g., fail-stop, crash, Byzantine



What should we learn today?

- Identify hardware scale of highly-available services common by current standards
- Predict reliability of a component configuration based on metrics such as failure probability, MTTF/ MTTR/MTBF, availability/downtime
- Explain how assumptions of reliability models are at odds with observed events
- Explain and apply common fault-tolerance strategies such as error detection, containment, and masking
- Explain techniques for redundancy, such as n-version programming, error coding, duplicated components, replication
- Categorize main variants of replication techniques and implement simple replication protocols