Lecture 11
Replication

Primary/Backup

Gifford weighted Quorum Consensus

Demers Epidemic Algorithms

Why replicate?

- Data replication: common technique in distributed systems
- Reliability
  - If one replica is unavailable or crashes, use another
  - Protect against corrupted data
- Performance
  - Scale with size of the distributed system (replicated web servers)
  - Scale in geographically distributed systems (web proxies)
- Key issue: need to maintain *consistency* of replicated data
  - If one copy is modified, others become inconsistent

Challenges: Fault Tolerance

- The goal is to have data available despite failures
- If one site fails others should continue providing service
- ***How many replicas should we have?***
- It depends on:
  - How many faults we want to tolerate
  - The **types** of faults we expect
  - How much we are willing to pay

Challenges: Data Consistency

- We will study systems that use ***data replication***
- It is hard, because data must be kept **consistent**
- Users submit operations against the logical copies of data
- These operations must be translated into operations against one, some, or all physical copies of data

Design Considerations for Replicated Services

- Where to submit updates?
  - A designated server or any server?
- When to propagate updates?
  - Eager or lazy?
- How consistent?
  - strict
  - eventual
- How many replicas to install?

Example of Data Inconsistency

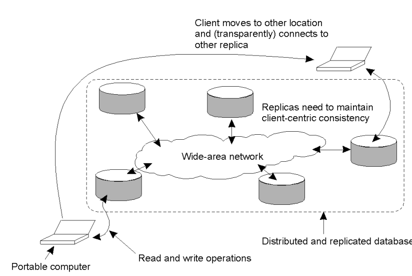
- **Client operations:**
  write(x = 5)
  read (x) // should return 5 on a single-server system
- **On a replicated system:**
  write (x = 5)
  Primary responds to client
  Primary crashed before propagating update to other replicas
  A new primary is selected
  read (x) // may return x ≠ 5, the new primary does not know about the update to x

Strict Consistency

- Any read always returns the result of the most recent write
  - Implicitly assumes the presence of a global clock
  - A write is immediately visible to all processes
    - Difficult to achieve in real systems (network delays can be variable)
- Nearly all existing approaches follow a **ROWA(A)** approach:
  - Read-one-write-all-(available)
  - Update has to be (eventually) executed at all replicas to keep them consistent
  - Read can be performed at any replica

Eventual Consistency

- Assume a replicated database with few updaters and many readers
- Eventual consistency: in absence of updates, all replicas converge towards identical copies
  - Only requirement: an update should eventually propagate to all replicas
  - Cheap to implement
  - Things work fine so
  - What if they don't:

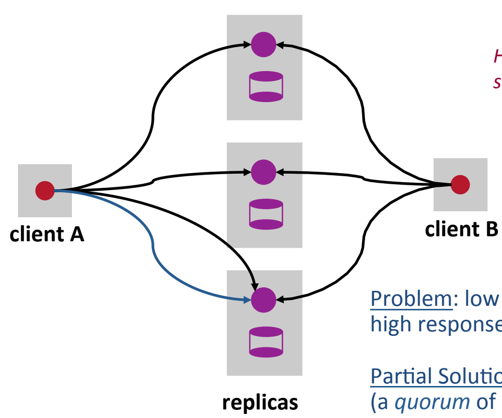


Eventual Consistency

- Many systems: one or few processes perform updates
  - How frequently should these updates be made available to other read-only processes?
- Examples:
  - DNS: single naming authority per domain
  - Only naming authority allowed updates (no write-write conflicts)
  - How should read-write conflicts (consistency) be addressed?
  - NIS: user information database in Unix systems
    - Only sys-admins update database, users only read data
    - Only user updates are changes to password

Synchronous Replication

Basic scheme: connect each client (or *front-end*) with every replica: writes go to all replicas, but client can read from any replica (*read-one-write-all replication*).



How to ensure that each replica sees updates in the "right" order?

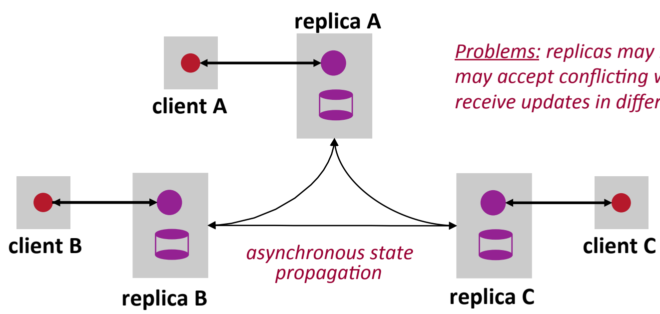
Problem: low concurrency, low availability, and high response times.

Partial Solution: Allow writes to any *N* replicas (a *quorum* of size *N*). To be safe, reads must also request data from a quorum of replicas.

Asynchronous Replication

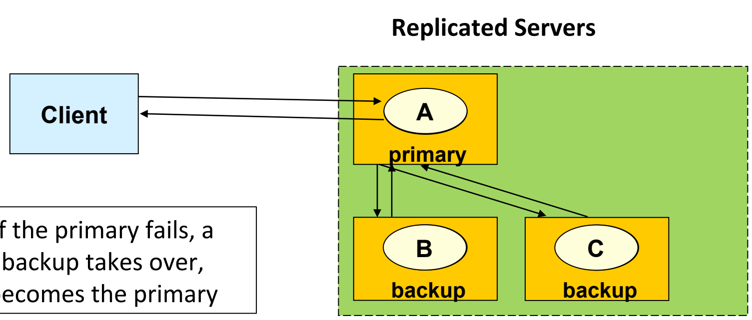
<u>Idea</u>: build available/scalable information services with *read-any-write-any* replication and a weak consistency model.

- no denial of service during transient network partitions
- supports massive replication without massive overhead
- "ideal for the Internet and mobile computing" [Golding92]



<u>Problems:</u> replicas may be out of date, may accept conflicting writes, and may receive updates in different orders.

Primary-Backup Replication (PB)



If the primary fails, a backup takes over, becomes the primary

Also known as passive replication

Where to Submit Updates?

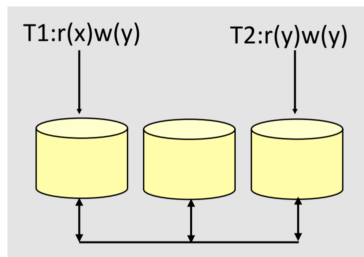
- Primary Copy
  - Choose one replica of data item to be the **primary copy**.
    - Site containing the replica is called the **primary site** for that data item
    - Different data items can have different primary sites
  - When a transaction needs to lock a data item *Q*, it requests a lock at the primary site of *Q*.
    - Implicitly gets lock on all replicas of the data item
  - Benefit
    - Concurrency control for replicated data handled similarly to unreplicated data - simple implementation.
  - Drawback
    - If the primary site of *Q* fails, *Q* is inaccessible even though other sites containing a replica may be accessible.

PB Replication with Eager Updates

1. The client sends the request to the primary
2. There is no initial coordination
3. The primary executes the request
4. The primary coordinates with the other replicas by sending the update information to the backups
5. The primary (or another replica) sends the answer to the client

Where to Submit Updates

- Update Everywhere:
  - Both read and write operations can be submitted to any server
  - This server takes care of the execution of the operation and the propagation of updates to the other copies



Majority Protocol (Cont.)

- In case of replicated data
  - If *Q* is replicated at n sites, then a request message must be sent to more than half of the n sites in which *Q* is stored.
  - The transaction does not operate on *Q* until it has obtained a lock on a majority of the replicas of *Q*.
  - When writing the data item, transaction performs writes on *all* replicas.
- Benefit
  - Can be used even when some sites are unavailable
    - details on how handle writes in the presence of site failure later
- Drawback
  - Need to talk to half of replicas for all read/write operations

Quorum Consensus

- Goal: prevent partitions from from producing inconsistent results.
- Quorum: subgroup of replicas whose size gives it the right to carry out operations.
- Quorum consensus replication:
  - Update will propagate successfully to a subgroup of replicas.
  - Other replicas will have outdated copies but will be updated off-line.

Quorum Consensus Protocol

- A generalization of both majority and biased protocols
- Each site is assigned a weight.
  - Let S be the total of all site weights
- Choose two values read quorum Q_r and write quorum Q_w
  - Such that $Q_r + Q_w > S$ and $2 * Q_w > S$
  - Quorums can be chosen (and S computed) separately for each item
- Each read must lock enough replicas that the sum of the site weights is >= Q_r
- Each write must lock enough replicas that the sum of the site weights is >= Q_w
  - Any two write quorums must share a member
- For now we assume all replicas are written
  - Extensions to allow some sites to be unavailable described later

Weighted Voting [Gifford] 1

- Every copy assigned a number of votes (weight assigned to a particular replica).
- Read: Must obtain *R* votes to read from any up-to-date copy.
- Write: Must obtain write quorum of *W* before performing update.

Weighted Voting 2

- W > 1/2 total votes, R+W > total votes.
- Ensures non-null intersection between every read quorum and write quorum.
- Read quorum guaranteed to have current copy.
- Freshness is determined by version numbers.

- QUESTION: What if rules above not hold?
  - not consistent, but still available

Weighted Voting 3

- On read:
  - Try to find enough copies, ie, total votes no less than R. Not all copies need to be current.
  - Since it overlaps with write quorum, at least one copy is current.
- On write:
  - Try to find set of up-to-date replicas whose votes no less than W.
  - If no sufficient quorum, current copies replace old ones, then update.

Gifford's quorum consensus examples (1979)

		Example 1	*Example 2*	*Example 3*
Latency (milliseconds)	Replica 1	75	75	75
	Replica 2	65	100	750
	Replica 3	65	750	750
Voting configuration	Replica 1	1	2	1
	Replica 2	0	1	1
	Replica 3	0	1	1
Quorum sizes	*R*	1	2	1
	W	1	3	3

Derived performance of file suite:

Read	Latency	65	75	75
	Blocking probability	0.01	0.0002	0.000001
Write	Latency	75	100	750
	Blocking probability	0.01	0.0101	0.03

Derived performance
latency
blocking probability - probability that a quorum cannot be obtained, assuming probability of 0.01 that any single RM is unavailable

22

•

Example Explained

Example 1 is configured for a file with high read to write ratio with several weak representatives and a single RM.
Replication is used for performance, not reliability.
The RM can be accessed in 75 ms and the two clients can access their weak representatives in 65 ms, resulting in lower latency and less network traffic

Example 2 is configured for a file with a moderate read to write ratio which is accessed mainly from one local network. Local RM has 2 votes and remote RMs 1 vote each.
Reads can be done at the local RM, but *writes* must access one local RM and one remote RM. If the local RM fails only *reads* are allowed

Example 3 is configured for a file with a very high read to write ratio.
Reads can be done at any RM and the probability of the file being unavailable is small. But *writes* must access all RMs.

When to Propagate Updates?

- Eager:
  - Within the boundaries of the transaction for replicated databases
  - Before response is sent to client for non-transactional services
- Lazy:
  - After the commit of the transaction for replicated databases
  - After the response is sent to client for non-transactional services

– QUESTION: How spread updates?

Direct Mail

- Each update is immediately sent from its entry site to all other sites.
- When a node receives an update, it checks the timestamp of update with local timestamp. Newer updates win
  - Timely – updates are sent immediately
  - Efficiency – reasonable. Number of messages proportional to number of updates and average hop count
  - Problems:
    - Nodes do not know about all replicas
    - Mail is not reliable delivery mechanism

Epidemic Protocols

- Based on theory of epidemics *(spreading infectious diseases)*
  - Upon an update, try to "infect" other replicas as quickly as possible
  - Pair-wise exchange of updates (*like pair-wise spreading of a disease)*
  - Terminology:
    - Infective store: store with an update it is willing to spread
    - Susceptible store: store that is not yet updated
- Many algorithms possible to spread updates
- Use randomness to get probabilistic guarantees
  - Exchange reliability guarantees for better scalability
    - May increase time to converge
    - Decreases complexity/coordination
    - Can improve fault tolerance/performance: fixed amount of load per cycle rather than continuously retrying
  - The achievement of strong reliability) in practical distributed systems requires expensive mechanisms to detect missing messages and initiate retransmissions. Due to the overhead of message loss detection and reparation, protocols offering such strong guarantees do not scale over a couple of hundred processes

Anti-entropy

- Entropy - amount of entropy is a measure of the disorder, or randomness, of a system. (from thermodynamics – Encyclopedia Britannica)
- Updates available in few sites – high entropy. Anti-entropy tries to restore order back into the system
- Every site regularly chooses another side at random and exchanges database contents with it and resolves any different between the two

Feb 7, 2001 CSCI {4,6}900: Ubiquitous Computing 27

Anti-entropy

- Differences are resolved using:
  - Push: infective -> susceptible
  - Pull: susceptible -> infective
  - Push-Pull: depending on the time stamps, updates are either pushed or pulled
- Common case: Pull or push-pull preferred
- Reliable, but high overhead because have to "diff" the databases

Feb 7, 2001 CSCI {4,6}900: Ubiquitous Computing 28

Anti-Entropy

Assume that

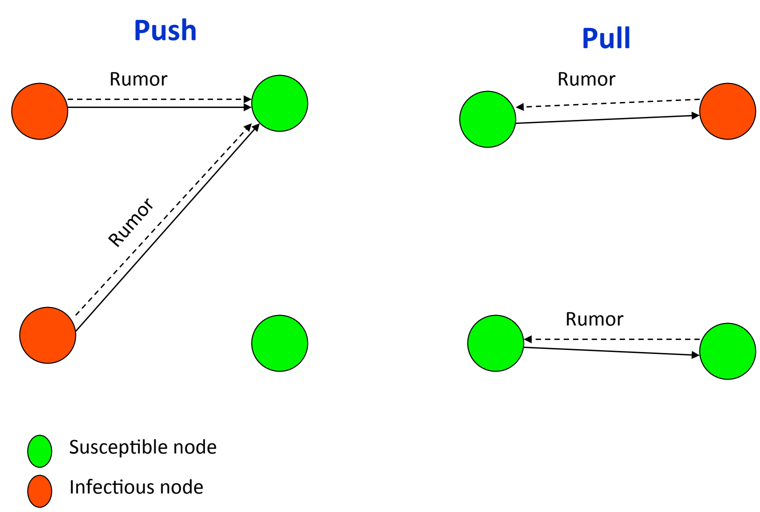
- Site s' is chosen uniformly at random from the set S
- Each site executes the anti-entropy algorithm once per period

It can be proved that

- An update will eventually infect the entire population
- Starting from a single affected site, this can be achieved in time proportional to the log of the population size

29

Anti-entropy – Push and Pull



CS598IG Epidemics 02/09

Anti-Entropy

At each site s periodically execute:

For some s' ∈ S

ResolveDifference[s, s']

Three ways to execute ResolveDifference:

Push

If s.Valueof.t > s'.Valueof.t

s'.ValueOf ← s.ValueOf

Pull

If s.Valueof.t < s'.Valueof.t

s'.ValueOf ← s.ValueOf

Push-Pull

s.Valueof.t > s'.Valueof.t ⇒ s'.ValueOf ← s.ValueOf

s.Valueof.t < s'.Valueof.t ⇒ s.ValueOf ← s'.ValueOf

31

Pull > Push

p_i – Probability that a node is susceptible after the i[th] round

$$p_{i+1} = p_i^2$$ Pull

$$p_{i+1} = p_i \left(1 - \frac{1}{n}\right)^{n(1-p_i)}$$ Push

- For push, suscep = prob suscep * prob no infected site contacted it
- Pull converges faster than push, thus providing better delay
  - Prob still not have update = prob not have in round i * prob of contacting someone who didn't have it
  - Easier for a susceptible node to find an infectious node near the end than vice versa

CS598IG Epidemics 02/09

Anti-entropy: Optimizations

- Maintain checksum, compare databases if checksums unequal
- Maintain recent update lists for time *T*, exchange lists first
- Maintain inverted index of database by timestamp; exchange information in reverse timestamp order, incrementally re-compute checksums

Problems with Anti Entropy

- Can still take a while to converge
  - Many rounds after an update is introduced
- Requires constant traffic to disseminate updates

Complex Epidemic terminology

- Site holding an update it is willing to share "infective"
- Site that has not received an update "susceptible"
- Site that has received an update but not willing to share it "removed"
  - Anti-entropy: sites are always susceptible or infective

Complex Epidemics: Rumor Spreading

There are n individuals initially inactive (susceptible)

We plant a rumor with one person who becomes active (infective), phoning other people at random and sharing the rumor

Every person bearing the rumor also becomes active and likewise shares the rumor

When an active individual makes an unnecessary phone call (the recipient already knows the rumor), then with *probability 1/k* the active individual loses interest in sharing the rumor (becomes removed)

We would like to know:

- How fast the system converges to an inactive state (no one is infective)
- The percentage of people that know the rumor when the inactive state is reached

Rumor mongering

- Sites are initially "ignorant"
- When site receives new information, it becomes a "hot rumor"
  – Periodically chooses another site at random and ensures that the other site has seen the update
  – When a site has tried to share a hot rumor with too many sites that have already seen it, the site stops treating the rumor as hot and retains the update without propagating it further
  – 1/k probability : k=1, 20% and k=2, 6% will miss updates
  – There is a chance that an update will not reach all sites (backup anti-entropy process)

Methods for spreading updates:

Rumor cycles can be more frequent that anti-entropy cycles, because they require fewer resources at each site, but there is a chance that an update will not reach all sites

Criteria to characterize epidemics

Residue

The value of s when i is zero, that is, the remaining susceptible when the epidemic finishes

Traffic

m = Total update traffic / Number of sites

Delay

Average delay (t_{avg}) is the difference between the time of the initial injection of an update and the arrival of the update at a given site averaged over all sites

The delay until (t_{last}) the reception by the last site that will receive the update during an epidemic

Variants of Epidemic Algorithms

- Blind vs. Feedback
  - Feedback: Loss of interest with probability *1/k* only when recipient already knows the rumor
- Counter vs. Coin
  - Counter: Lose interest completely after *k* unnecessary contacts
  - Coin: Lose interest with probability *1/k* for every unnecessary contact
- Push vs. Pull

Simple variations of rumor spreading

Push vs. Pull

Pull converges faster

If there are numerous independent updates, a pull request is likely to find a source with a non-empty rumor list

If the database is quiescent, the push phase ceases to introduce traffic overhead, while the pull continues to inject useless requests for updates

Counter, feedback and pull work better

41

Minimization

Use a push and pull together, if both sites know the update, only the site with the smaller counter is incremented

Connection Limit

A site can be the recipient of more than one push in a cycle, while for pull, a site can service an unlimited number of requests

With limit, only one contact per cycle

Push gets better: if the limit kicks in, the site still gets the update (acts lIke "OR")

Pull gets worst: if limit kicks in, site does not get update at all

42

Removing Data

- Deletion of data items is hard in epidemic protocols
- Example: server deletes data item *x*
  - No state information is preserved
    - Can't distinguish between a deleted copy and no copy!
- Solution: death certificates
  - Treat deletes as updates and spread a death certificate
    - Mark copy as deleted but don't delete
    - Need an eventual clean up
      - Clean up dormant death certificates

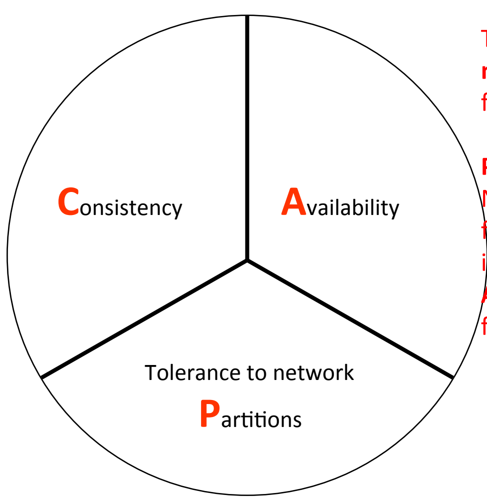
Deletion and Death Certificates

- Absence of item does not spread; On the contrary, it can get resurrected!
- Use of death certificates (DCs) – when a node receives a DC, old copy of data is deleted
- How long to maintain a DC?
- Use Chandy and Lamport snapshot algorithm to ensure all nodes have received
- Simpler strategy – hold DC for fixed amount of time

Complex Epidemics

- Optimizations of simple epidemic algorithms (anti-entropy, rumor mongering)
- "Complex" epidemics have simple implementations!
- Example: Each infectious node loses its ability to "infect" with a probability of *1/k* in each cycle

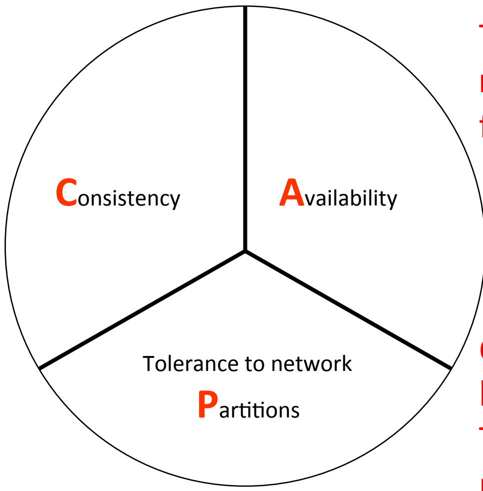
The CAP Theorem



Theorem: You can have **at most two** of these invariants for any shared-data system

Partition tolerance:
No failures less than total network failure cause the system to respond incorrectly
Availability: respond within a time frame

The CAP Theorem

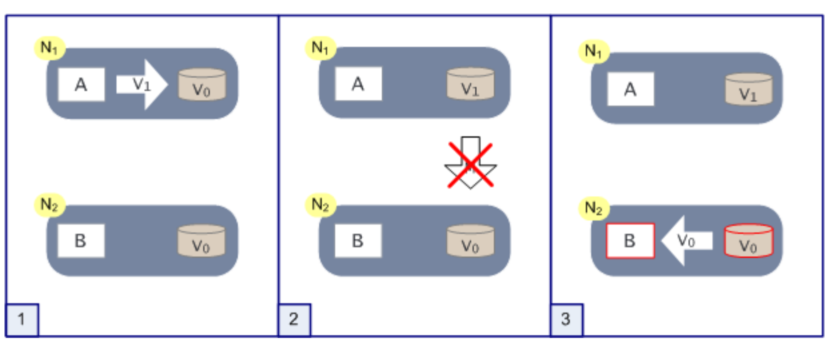


Theorem: You can have **at most two** of these invariants for any shared-data system

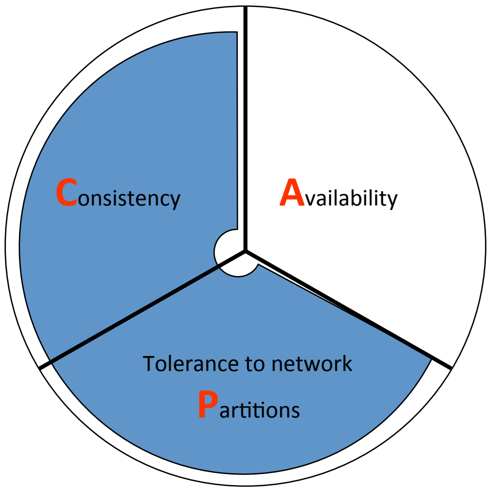
Corollary: consistency boundary must choose A or P
Think: what happens in a partition case? A or C?

Why?

- Consider cases:
  - Primary/backup replication
  - Eager replication with quorums
  - Lazy replication (e.g. epidemic)



Forfeit Availability



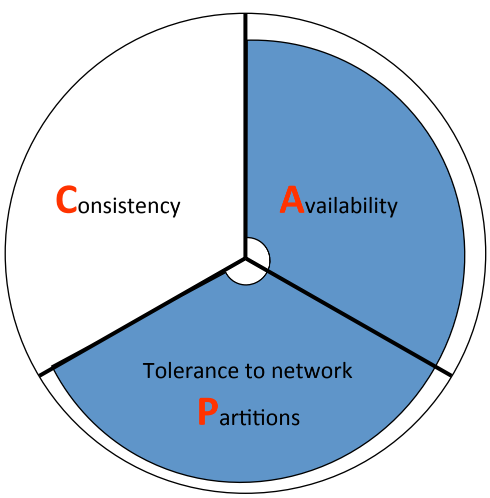
Examples

- Distributed locking
  - If lock held on other side, no availability
- Majority protocols
  - If don't have majority, cannot access

Traits

- Pessimistic locking
- Make minority partitions unavailable

Forfeit Consistency



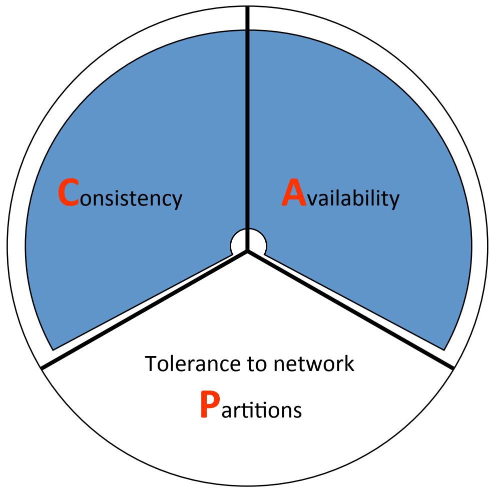
Examples

- Grapevine
- Web caching
- NFS

Traits

- Update-anywhere
- Lazy replication
- Conflict resolution
  - Allow inconsistency but detect
- Eventual consistency

Forfeit Partitions



Examples

- Single-site databases
  - Partition causes failures

Traits

- 2-phase commit
- Primary/backup
- What happens on partition?
  - Can either can inconsistency (both sides can access and see out of date data) or unavailability (

Beating CAP

- What can you do?
  - Have knobs to tune C,A,P
    - E.g. quorum sizes for replication
  - Have normal & failure modes
    - Consistent normally, but in partition fall back
    - E.g. normally direct mail/eager, but fall back to anti-entropy