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

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: no or infrequent write-write conflicts
  - Things work fine so long as user accesses same replica
  - What if they don’t:
Client-centric Consistency Models

- Assume read operations by a single process $P$ at two different local copies of the same data store
  - Four different consistency semantics
- **Monotonic reads**
  - Once read, subsequent reads on that data items return same or more recent values
- **Monotonic writes**
  - A write must be propagated to all replicas before a successive write by the same process
  - Resembles FIFO consistency (writes from same process are processed in same order)
- **Read your writes**: read(x) always returns write(x) by that process
- **Writes follow reads**: write(x) following read(x) will take place on same or more recent version of x

Epidemic Protocols

- Used in Bayou system from Xerox PARC
- Bayou: weakly connected replicas
  - Useful in mobile computing (mobile laptops)
  - Useful in wide area distributed databases (weak connectivity)
- 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
Spreading an Epidemic

• Anti-entropy
  – Server $P$ picks a server $Q$ at random and exchanges updates
  – Three possibilities: only push, only pull, both push and pull
  – Claim: A pure push-based approach does not help spread updates quickly (Why?)
    • Pull or initial push with pull work better

• Rumor mongering (aka gossiping)
  – Upon receiving an update, $P$ tries to push to $Q$
  – If $Q$ already received the update, stop spreading with prob $1/k$
  – Analogous to “hot” gossip items => stop spreading if “cold”
  – Does not guarantee that all replicas receive updates
    • Chances of staying susceptible: $s = e^{(k+1)(1-s)}$

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
Implementation Issues

- Two techniques to implement consistency models
  - Primary-based protocols
    - Assume a primary replica for each data item
    - Primary responsible for coordinating all writes
  - Replicated write protocols
    - No primary is assumed for a data item
    - Writes can take place at any replica

Remote-Write Protocols

- Traditionally used in client-server systems
Remote-Write Protocols (2)

- Primary-backup protocol
  - Allow local reads, sent writes to primary
  - Block on write until all replicas are notified
  - Implements sequential consistency

Local-Write Protocols (1)

- Primary-based local-write protocol in which a single copy is migrated between processes.
  - Limitation: need to track the primary for each data item
Local-Write Protocols (2)

- Primary-backup protocol in which the primary migrates to the process wanting to perform an update.

Replicated-write Protocols

- Relax the assumption of one primary
  - No primary, any replica is allowed to update
  - Consistency is more complex to achieve

- Quorum-based protocols
  - Use voting to request/acquire permissions from replicas
  - Consider a file replicated on \( N \) servers
    - \( N_R + N_W > N \)  \( N_W > N/2 \)
  - Update: contact at least \( (N/2+1) \) servers and get them to agree to do update (associate version number with file)
  - Read: contact majority of servers and obtain version number
    - If majority of servers agree on a version number, read
Gifford’s Quorum-Based Protocol

- Three examples of the voting algorithm:
  a) A correct choice of read and write set
  b) A choice that may lead to write-write conflicts
  c) A correct choice, known as ROWA (read one, write all)

Replica Management

- Replica server placement
  - Web: geographically skewed request patterns
  - Where to place a proxy?
    - K-clusters algorithm
- Permanent replicas versus temporary
  - Mirroring: all replicas mirror the same content
  - Proxy server: on demand replication
- Server-initiated versus client-initiated
Content Distribution

• Will come back to this in Chap 12

• CDN: network of proxy servers
• Caching:
  – update versus invalidate
  – Push versus pull-based approaches
  – Stateful versus stateless
• Web caching: what semantics to provide?

Final Thoughts

• Replication and caching improve performance in distributed systems
• Consistency of replicated data is crucial
• Many consistency semantics (models) possible
  – Need to pick appropriate model depending on the application
  – Example: web caching: weak consistency is OK since humans are tolerant to stale information (can reload browser)
  – Implementation overheads and complexity grows if stronger guarantees are desired
Fault Tolerance

• Single machine systems
  – Failures are all or nothing
    • OS crash, disk failures
• Distributed systems: multiple independent nodes
  – Partial failures are also possible (some nodes fail)
• Question: Can we automatically recover from partial failures?
  – Important issue since probability of failure grows with number of independent components (nodes) in the systems
  – Prob(failure) = Prob(Any one component fails) = 1 - P(no failure)

A Perspective

• Computing systems are not very reliable
  – OS crashes frequently (Windows), buggy software, unreliable hardware, software/hardware incompatibilities
  – Until recently: computer users were “tech savvy”
    • Could depend on users to reboot, troubleshoot problems
  – Growing popularity of Internet/World Wide Web
    • “Novice” users
    • Need to build more reliable/dependable systems
  – Example: what is your TV (or car) broke down every day?
    • Users don’t want to “restart” TV or fix it (by opening it up)
• Need to make computing systems more reliable
  – Important for online banking, e-commerce, online trading, webmail…
Basic Concepts

• Need to build *dependable* systems
• Requirements for dependable systems
  – Availability: system should be available for use at any given time
    • 99.999% availability (five 9s) => very small down times
  – Reliability: system should run continuously without failure
  – Safety: temporary failures should not result in a catastrophic
    • Example: computing systems controlling an airplane, nuclear reactor
  – Maintainability: a failed system should be easy to repair

Basic Concepts (contd)

• Fault tolerance: system should provide services despite faults
  – Transient faults
  – Intermittent faults
  – Permanent faults
Failure Models

<table>
<thead>
<tr>
<th>Type of failure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crash failure</td>
<td>A server halts, but is working correctly until it halts</td>
</tr>
<tr>
<td>Omission failure</td>
<td></td>
</tr>
<tr>
<td>Receive omission</td>
<td>A server fails to respond to incoming requests</td>
</tr>
<tr>
<td>Send omission</td>
<td>A server fails to receive incoming messages</td>
</tr>
<tr>
<td>Send omission</td>
<td>A server fails to send messages</td>
</tr>
<tr>
<td>Timing failure</td>
<td>A server's response lies outside the specified time interval</td>
</tr>
<tr>
<td>Response failure</td>
<td></td>
</tr>
<tr>
<td>Value failure</td>
<td>The server's response is incorrect</td>
</tr>
<tr>
<td>State transition failure</td>
<td>The value of the response is wrong</td>
</tr>
<tr>
<td>State transition failure</td>
<td>The server deviates from the correct flow of control</td>
</tr>
<tr>
<td>Arbitrary failure</td>
<td>A server may produce arbitrary responses at arbitrary times</td>
</tr>
</tbody>
</table>

- Different types of failures.

Failure Masking by Redundancy

- Triple modular redundancy.