### CS 268: Lecture 19

**A Quick Survey of Distributed Systems**

*80 slides in 80 minutes!*

#### Agenda

- Introduction and overview
- Data replication and eventual consistency
- Bayou: beyond eventual consistency
- Practical BFT: fault tolerance
- SFS: security

#### To Paraphrase Lincoln....

- Some of you will know all of this
- And all of you will know some of this
- But not all of you will know all of this.....

#### Why Distributed Systems in 268?

- You won't learn it any other place....
- Networking research is drifting more towards DS
- DS research is drifting more towards the Internet

*It has nothing to do with the fact that I'm also teaching DS... ...but for those of you co-enrolled, these slides will look very familiar!*

#### Two Views of Distributed Systems

- **Optimist:** A distributed system is a collection of independent computers that appears to its users as a single coherent system
- **Pessimist:** “You know you have one when the crash of a computer you’ve never heard of stops you from getting any work done.” (Lamport)

#### History

- First, there was the mainframe
- Then there were workstations (PCs)
- Then there was the LAN
- Then wanted collections of PCs to act like a mainframe
- Then built some neat systems and had a vision for future
- But the web changed everything
Why?

- The vision of distributed systems:
  - Enthralling dream
  - Very promising start, theoretically and practically

- But the impact was limited by:
  - Autonomy (fate sharing, policies, cost, etc.)
  - Scaling (some systems couldn’t scale well)

- The Internet (and the web) provided:
  - Extreme autonomy
  - Extreme scale
  - Poor consistency (nobody cared!)

Recurring Theme

- Academics like:
  - Clean abstractions
  - Strong semantics
  - Things that prove they are smart

- Users like:
  - Systems that work (most of the time)
  - Systems that scale
  - Consistency per se isn’t important

- Eric Brewer had the following observations

A Clash of Cultures

- Classic distributed systems: focused on ACID semantics
  - A: Atomic
  - C: Consistent
  - I: Isolated
  - D: Durable

- Modern Internet systems: focused on BASE
  - Basically Available
  - Soft-state (or scalable)
  - Eventually consistent

ACID vs BASE

<table>
<thead>
<tr>
<th>ACID</th>
<th>BASE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strong consistency for transactions</td>
<td>Availability and scaling highest priorities</td>
</tr>
<tr>
<td>Availability less important</td>
<td>Weak consistency</td>
</tr>
<tr>
<td>Pessimistic</td>
<td>Optimistic</td>
</tr>
<tr>
<td>Rigorous analysis</td>
<td>Best effort</td>
</tr>
<tr>
<td>Complex mechanisms</td>
<td>Simple and fast</td>
</tr>
</tbody>
</table>

Why Not ACID+BASE?

- What goals might you want from a shared-date system?
  - C, A, P

- Strong Consistency: all clients see the same view, even in the presence of updates

- High Availability: all clients can find some replica of the data, even in the presence of failures

- Partition-tolerance: the system properties hold even when the system is partitioned

CAP Theorem [Brewer]

- You can only have two out of these three properties

- The choice of which feature to discard determines the nature of your system
Consistency and Availability

- Comment:
  - Providing transactional semantics requires all functioning nodes to be in contact with each other (no partition)

- Examples:
  - Single-site and clustered databases
  - Other cluster-based designs

- Typical Features:
  - Two-phase commit
  - Cache invalidation protocols
  - Classic DS style

Consistency and Partition-Tolerance

- Comment:
  - If one is willing to tolerate system-wide blocking, then can provide consistency even when there are temporary partitions

- Examples:
  - Distributed locking
  - Quorum (majority) protocols

- Typical Features:
  - Pessimistic locking
  - Minority partitions unavailable
  - Also common DS style
    - Voting vs primary copy

Partition-Tolerance and Availability

- Comment:
  - Once consistency is sacrificed, life is easy....

- Examples:
  - DNS
  - Web caches
  - Coda
  - Bayou

- Typical Features:
  - TTLs and lease cache management
  - Optimistic updating with conflict resolution
  - This is the “Internet design style”

Voting with their Clicks

- In terms of large-scale systems, the world has voted with their clicks:
  - Consistency less important than availability and partition-tolerance

Data Replication and Eventual Consistency

- Why replication?
  - Volume of requests
  - Proximity
  - Availability

- Challenge of replication: consistency

Replication
Many Kinds of Consistency

- Strict: updates happen instantly everywhere
- Linearizable: updates happen in timestamp order
- Sequential: all updates occur in the same order everywhere
- Causal: on each replica, updates occur in a causal order
- FIFO: all updates from a single client are applied in order

Focus on Sequential Consistency

- Weakest model of consistency in which data items have to converge to the same value everywhere
- But hard to achieve at scale:
  - Quorums
  - Primary copy
  - TPC
  - ...

Is Sequential Consistency Overkill?

- Sequential consistency requires that at each stage in time, the operations at a replica occur in the same order as at every other replica
- Ordering of writes causes the scaling problems!
- Why insist on such a strict order?

Eventual Consistency

- If all updating stops then eventually all replicas will converge to the identical values

Implementing Eventual Consistency

Can be implemented with two steps:

1. All writes eventually propagate to all replicas
2. Writes, when they arrive, are written to a log and applied in the same order at all replicas
   - Easily done with timestamps and “undo-ing” optimistic writes

Update Propagation

- Rumor or epidemic stage:
  - Attempt to spread an update quickly
  - Willing to tolerate incompletely coverage in return for reduced traffic overhead
- Correcting omissions:
  - Making sure that replicas that weren’t updated during the rumor stage get the update
Rumor Spreading: Push

- When a server P has just been updated for data item x, it contacts some other server Q at random and tells Q about the update.
- If Q doesn’t have the update, then it (after some time interval) contacts another server and repeats the process.
- If Q already has the update, then P decides, with some probability, to stop spreading the update.

Performance of Push Scheme

- Not everyone will hear!
  - Let S be fraction of servers not hearing rumors
  - Let M be number of updates propagated per server
- \( S = \exp(-M) \)
- Note that M depends on the probability of continuing to push rumor.
- Note that S(M) is independent of algorithm to stop spreading.

Pull Schemes

- Periodically, each server Q contacts a random server P and asks for any recent updates.
- P uses the same algorithm as before in deciding when to stop telling rumor.
- Performance: better (next slide), but requires contact even when no updates.

Variety of Pull Schemes

- When to stop telling rumor: (conjectures)
  - Counter: \( S = \exp(-M) \)
  - Min-counter: \( S = \exp(-2M) \) (best you can do!)
- Controlling who you talk to next
  - Can do better
- Knowing N:
  - Can choose parameters so that \( S << 1/N \)
- Spatial dependence

Finishing Up

- There will be some sites that don’t know after the initial rumor spreading stage.
- How do we make sure everyone knows?

Anti-Entropy

- Every so often, two servers compare complete datasets.
- Use various techniques to make this cheap.
- If any data item is discovered to not have been fully replicated, it is considered a new rumor and spread again.
### We Don’t Want Lazarus!

- Consider server P that does offline
- While offline, data item x is deleted
- When server P comes back online, what happens?

### Death Certificates

- Deleted data is replaced by a death certificate
- That certificate is kept by all servers for some time T that is assumed to be much longer than required for all updates to propagate completely
- But every death certificate is kept by at least one server forever

---

### Bayou

**Bayou**

- Eventual consistency: strongest scalable consistency model
- But not strong enough for mobile clients
  - Accessing different replicas can lead to strange results
- Bayou was designed to move beyond eventual consistency
  - One step beyond CODA
  - Session guarantees
  - Fine-grained conflict detection and application-specific resolution

### Why Should You Care about Bayou?

- Subset incorporated into next-generation WinFS
- Done by my friends at PARC

### Bayou System Assumptions

- Variable degrees of connectivity:
  - Connected, disconnected, and weakly connected
- Variable end-node capabilities:
  - Workstations, laptops, PDAs, etc.
- Availability crucial
**Resulting Design Choices**

- **Variable connectivity** ⇒ Flexible update propagation
  - Incremental progress, pairwise communication (anti-entropy)

- **Variable end-nodes** ⇒ Flexible notion of clients and servers
  - Some nodes keep state (servers), some don’t (clients)
  - Laptops could have both, PDAs probably just clients

- **Availability crucial** ⇒ Must allow disconnected operation
  - Conflicts inevitable
  - Use application-specific conflict detection and resolution

**Components of Design**

- Update propagation
- Conflict detection
- Conflict resolution
- Session guarantees

**Updates**

- Client sends update to a server
- Identified by a triple:
  - <Commit-stamp, Time-stamp, Server-ID of accepting server>
- Updates are either committed or tentative
  - Commit-stamps increase monotonically
  - Tentative updates have commit-stamp = inf
- Primary server does all commits:
  - It sets the commit-stamp
  - Commit-stamp different from time-stamp

**Update Log**

- Update log in order:
  - Committed updates (in commit-stamp order)
  - Tentative updates (in time-stamp order)
- Can truncate committed updates, and only keep db state
- Clients can request two views: (or other app-specific views)
  - Committed view
  - Tentative view

**Bayou System Organization**

**Anti-Entropy Exchange**

- Each server keeps a version vector:
  - R.V[X] is the latest timestamp from server X that server R has seen
- When two servers connect, exchanging the version vectors allows them to identify the missing updates
- These updates are exchanged in the order of the logs, so that if the connection is dropped the crucial monotonicity property still holds
  - If a server X has an update accepted by server Y, server X has all previous updates accepted by that server
Requirements for Eventual Consistency

- Universal propagation: anti-entropy
- Globally agreed ordering: commit-stamps
- Determinism: writes do not involve information not contained in the log (no time-of-day, process-ID, etc.)

Example with Three Servers

All Servers Write Independently

<table>
<thead>
<tr>
<th>P</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="none" alt="inf,1,P&gt;" /></td>
<td><img src="none" alt="inf,2,A&gt;" /></td>
<td><img src="none" alt="inf,1,B&gt;" /></td>
</tr>
<tr>
<td><img src="none" alt="inf,4,P&gt;" /></td>
<td><img src="none" alt="inf,3,A&gt;" /></td>
<td><img src="none" alt="inf,5,B&gt;" /></td>
</tr>
<tr>
<td><img src="none" alt="inf,8,P&gt;" /></td>
<td><img src="none" alt="inf,10,A&gt;" /></td>
<td><img src="none" alt="inf,9,B&gt;" /></td>
</tr>
<tr>
<td><img src="none" alt="8,0,0&gt;" /></td>
<td><img src="none" alt="0,10,0&gt;" /></td>
<td><img src="none" alt="0,0,9&gt;" /></td>
</tr>
</tbody>
</table>

P and A Do Anti-Entropy Exchange

<table>
<thead>
<tr>
<th>P</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="none" alt="inf,1,P&gt;" /></td>
<td><img src="none" alt="inf,2,A&gt;" /></td>
<td><img src="none" alt="inf,1,B&gt;" /></td>
</tr>
<tr>
<td><img src="none" alt="inf,2,A&gt;" /></td>
<td><img src="none" alt="inf,3,A&gt;" /></td>
<td><img src="none" alt="inf,5,B&gt;" /></td>
</tr>
<tr>
<td><img src="none" alt="inf,3,A&gt;" /></td>
<td><img src="none" alt="inf,4,P&gt;" /></td>
<td><img src="none" alt="inf,9,B&gt;" /></td>
</tr>
<tr>
<td><img src="none" alt="inf,4,P&gt;" /></td>
<td><img src="none" alt="inf,8,P&gt;" /></td>
<td><img src="none" alt="inf,10,A&gt;" /></td>
</tr>
<tr>
<td><img src="none" alt="inf,10,A&gt;" /></td>
<td><img src="none" alt="inf,1,B&gt;" /></td>
<td><img src="none" alt="inf,10,A&gt;" /></td>
</tr>
<tr>
<td><img src="none" alt="8,10,0&gt;" /></td>
<td><img src="none" alt="8,10,0&gt;" /></td>
<td><img src="none" alt="0,0,9&gt;" /></td>
</tr>
</tbody>
</table>

P Commits Some Early Writes

<table>
<thead>
<tr>
<th>P</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="none" alt="&lt;1,1,P&gt;" /></td>
<td><img src="none" alt="&lt;inf,1,P&gt;" /></td>
<td><img src="none" alt="&lt;inf,1,B&gt;" /></td>
</tr>
<tr>
<td><img src="none" alt="&lt;2,2,A&gt;" /></td>
<td><img src="none" alt="&lt;inf,2,A&gt;" /></td>
<td><img src="none" alt="&lt;inf,5,B&gt;" /></td>
</tr>
<tr>
<td><img src="none" alt="&lt;3,3,A&gt;" /></td>
<td><img src="none" alt="&lt;inf,3,A&gt;" /></td>
<td><img src="none" alt="&lt;inf,9,B&gt;" /></td>
</tr>
<tr>
<td><img src="none" alt="&lt;inf,4,P&gt;" /></td>
<td><img src="none" alt="&lt;inf,4,P&gt;" /></td>
<td><img src="none" alt="&lt;inf,10,A&gt;" /></td>
</tr>
<tr>
<td><img src="none" alt="&lt;inf,8,P&gt;" /></td>
<td><img src="none" alt="&lt;inf,8,P&gt;" /></td>
<td><img src="none" alt="0,0,9&gt;" /></td>
</tr>
<tr>
<td><img src="none" alt="&lt;inf,10,A&gt;" /></td>
<td><img src="none" alt="&lt;inf,10,A&gt;" /></td>
<td><img src="none" alt="&lt;inf,10,A&gt;" /></td>
</tr>
<tr>
<td><img src="none" alt="8,10,0&gt;" /></td>
<td><img src="none" alt="8,10,0&gt;" /></td>
<td><img src="none" alt="8,10,0&gt;" /></td>
</tr>
</tbody>
</table>

P and B Do Anti-Entropy Exchange

<table>
<thead>
<tr>
<th>P</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="none" alt="&lt;1,1,P&gt;" /></td>
<td><img src="none" alt="&lt;inf,1,P&gt;" /></td>
<td><img src="none" alt="&lt;inf,1,B&gt;" /></td>
</tr>
<tr>
<td><img src="none" alt="&lt;2,2,A&gt;" /></td>
<td><img src="none" alt="&lt;inf,2,A&gt;" /></td>
<td><img src="none" alt="&lt;inf,5,B&gt;" /></td>
</tr>
<tr>
<td><img src="none" alt="&lt;3,3,A&gt;" /></td>
<td><img src="none" alt="&lt;inf,3,A&gt;" /></td>
<td><img src="none" alt="&lt;inf,9,B&gt;" /></td>
</tr>
<tr>
<td><img src="none" alt="&lt;inf,4,P&gt;" /></td>
<td><img src="none" alt="&lt;inf,4,P&gt;" /></td>
<td><img src="none" alt="&lt;inf,10,A&gt;" /></td>
</tr>
<tr>
<td><img src="none" alt="&lt;inf,5,B&gt;" /></td>
<td><img src="none" alt="&lt;inf,8,P&gt;" /></td>
<td><img src="none" alt="&lt;inf,9,B&gt;" /></td>
</tr>
<tr>
<td><img src="none" alt="&lt;inf,8,P&gt;" /></td>
<td><img src="none" alt="&lt;inf,10,A&gt;" /></td>
<td><img src="none" alt="inf,10,A&gt;" /></td>
</tr>
<tr>
<td><img src="none" alt="inf,9,B&gt;" /></td>
<td><img src="none" alt="inf,10,A&gt;" /></td>
<td><img src="none" alt="inf,10,A&gt;" /></td>
</tr>
<tr>
<td><img src="none" alt="8,10,0&gt;" /></td>
<td><img src="none" alt="8,10,0&gt;" /></td>
<td><img src="none" alt="8,10,0&gt;" /></td>
</tr>
</tbody>
</table>
P Commits More Writes

<table>
<thead>
<tr>
<th>P</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;1,1,P&gt;</td>
<td>&lt;1,1,P&gt;</td>
</tr>
<tr>
<td>&lt;2,2,A&gt;</td>
<td>&lt;2,2,A&gt;</td>
</tr>
<tr>
<td>&lt;3,3,A&gt;</td>
<td>&lt;3,3,A&gt;</td>
</tr>
<tr>
<td>&lt;inf,1,B&gt;</td>
<td>&lt;4,1,B&gt;</td>
</tr>
<tr>
<td>&lt;inf,4,P&gt;</td>
<td>&lt;inf,4,P&gt;</td>
</tr>
<tr>
<td>&lt;inf,5,B&gt;</td>
<td>&lt;6,5,B&gt;</td>
</tr>
<tr>
<td>&lt;inf,8,P&gt;</td>
<td>&lt;7,8,P&gt;</td>
</tr>
<tr>
<td>&lt;inf,9,B&gt;</td>
<td>&lt;inf,10,A&gt;</td>
</tr>
<tr>
<td>[8,10,9]</td>
<td>[8,10,9]</td>
</tr>
</tbody>
</table>

Bayou Writes

- Identifier (commit-stamp, time-stamp, server-ID)
- Nominal value
- Write dependencies
- Merge procedure

Conflict Detection

- Write specifies the data the write depends on:
  - Set X=8 if Y=5 and Z=3
  - Set Cal(11:00-12:00)=dentist if Cal(11:00-12:00) is null
- These write dependencies are crucial in eliminating unnecessary conflicts
  - If file-level detection was used, all updates would conflict with each other

Conflict Resolution

- Specified by merge procedure (mergeproc)
- When conflict is detected, mergeproc is called
  - Move appointments to open spot on calendar
  - Move meetings to open room

Session Guarantees

- When client move around and connects to different replicas, strange things can happen
  - Updates you just made are missing
  - Database goes back in time
  - Etc.
- Design choice:
  - Insist on stricter consistency
  - Enforce some “session” guarantees
- SGs ensured by client, not by distribution mechanism

Read Your Writes

- Every read in a session should see all previous writes in that session
Monotonic Reads and Writes

- A later read should never be missing an update present in an earlier read
- Same for writes

Writes Follow Reads

- If a write W followed a read R at a server X, then at all other servers
  - If W is in Y's database then any writes relevant to R are also there

Supporting Session Guarantees

- Responsibility of “session manager”, not servers!
- Two sets:
  - Read-set: set of writes that are relevant to session reads
  - Write-set: set of writes performed in session
- Causal ordering of writes
  - Use Lamport clocks

Practical Byzantine Fault Tolerance

*Only a high-level summary*

The Problem

- Ensure correct operation of a state machine in the face of arbitrary failures
- Limitations:
  - no more than f failures, where n>3f
  - messages can't be indefinitely delayed

Basic Approach

- Client sends request to primary
- Primary multicasts request to all backups
- Replicas execute request and send reply to client
- Client waits for f+1 replies that agree

*Challenge: make sure replicas see requests in order*
### Algorithm Components

- Normal case operation
- View changes
- Garbage collection
- Recovery

### Normal Case

- When primary receives request, it starts 3-phase protocol
  - **pre-prepare**: accepts request only if valid
  - **prepare**: multicasts prepare message and, if 2f prepare messages from other replicas agree, multicasts commit message
  - **commit**: commit if 2f+1 agree on commit

### View Changes

- Changes primary
- Required when primary malfunctioning

### Communication Optimizations

- Send only one full reply: rest send digests
- Optimistic execution: execute prepared requests
- Read-only operations: multicast from client, and executed in current state

### Most Surprising Result

- Very little performance loss!

### Secure File System (SFS)
Secure File System (SFS)

- Developed by David Mazieres while at MIT (now NYU)
- Key question: how do I know I’m accessing the server I think I’m accessing?
- All the fancy distributed systems performance work is irrelevant if I’m not getting the data I wanted
- Several current stories about why I believe I’m accessing the server I want to access

Trust DNS and Network

- Someone I trust hands me server name: www.foocom.com
- Verisign runs root servers for .com, directs me to DNS server for foo.com
- I trust that packets sent to/from DNS and to/from server are indeed going to the intended destinations

Trust Certificate Authority

- Server produces certificate (from, for example, Verisign) that attests that the server is who it says it is.
- Disadvantages:
  - Verisign can screw up (which it has)
  - Hard for some sites to get meaningful Verisign certificate

Use Public Keys

- Can demand proof that server has private key associated with public key
- But how can I know that public key is associated with the server I want?

Secure File System (SFS)

- Basic problem in normal operation is that the pathname (given to me by someone I trust) is disconnected from the public key (which will prove that I’m talking to the owner of the key).
- In SFS, tie the two together. The pathname given to me automatically certifies the public key!

Self-Certifying Path Name

- LOC: DNS or IP address of server, which has public key K
- HID: Hash(LOC,K)
- Pathname: local pathname on server
### SFS Key Point

- Whatever directed me to the server initially also provided me with enough information to verify their key
- This design separates the issue of who I trust (my decision) from how I act on that trust (the SFS design)
- Can still use Verisign or other trusted parties to hand out pathnames, or could get them from any other source

### SUNDR

- Developed by David Mazieres
- SFS allows you to trust nothing but your server
  - But what happens if you don’t even trust that?
  - Why is this a problem?
- P2P designs: my files on someone else’s machine
- Corrupted servers: sourceforge hacked
  - Apache, Debian,Gnome, etc.

### Traditional File System Model

- Client sends read and write requests to server
- Server responds to those requests
- Client/Server channel is secure, so attackers can’t modify requests/responses
- But no way for clients to know if server is returning correct data
- What if server isn’t trustworthy?

### Byzantine Fault Tolerance

- Can only protect against a limited number of corrupt servers

### SUNDR Model V1

- Clients send digitally signed requests to server
- Server returns log of these requests
  - Server doesn’t compute anything
  - Server doesn’t know any keys
- Problem: server can drop some updates from log, or reorder them

### SUNDR Model V2

- Have clients sign log, not just their own request
- Only bad thing a server can do is a fork attack:
  - Keep two separate copies, and only show one to client 1 and the other to client 2
- This is hopelessly inefficient, but various tricks can solve the efficiency problem