Frequently asked questions from the previous class survey

- How are vector clocks actually used?
- Causally ordered multicasting
- When would a stream be stateful? Why would we actually need it?
Topics covered in this lecture

- Types of replicas
- Replicated write protocols
- Eventually Consistent
Types of Replicas

- Permanent Replicas
  - Initial set of replicas that comprise data store
    - Usually a small set
  - Files stored across servers at a single location
    - Request forwarded using round-robin strategy
  - Files copied to mirror sites
    - Geographically dispersed
Server initiated replicas

- Copies that exist to enhance performance
- Created at the initiative of the owner of data store

Server initiated replicas: Example

- Web server in NYC
  - Can handle dissemination loads effectively
- **Bursts** of traffic over 2-3 days may come in
  - From some specific location (or set of locations)
- Install **temporary replicas** in regions where requests originate
Server initiated replicas:
Issues in dynamic replications

- Replication takes place to **reduce load** at server
- *Specific* files on server migrated/replicated to servers in **proximity** of requesting clients

Dynamic replication: Migrating/replicating files

- Each server tracks **access counts** per file
  - And also **who** initiates accesses
- Given a client C
  - Each server can determine which of the servers is closest to C
Counting access requests from clients: C1 and C2 share closest server P

- Accesses from C1, C2 for file F at server Q are registered as if they are from P
  - \( count_Q(P, F) \)

Replication threshold: \( rep(S, F) \) for file F at server S

- Indicates number of requests for file is high
- Might be worth replicating it
Deletion thresholds

- When requests for file $F$ at server $S$ drops below deletion threshold, $\text{del}(S, F)$
  - File $F$ removed from $S$
- Number of replicas reduce
- Higher loads at the other servers
- Ensure at least one copy of file continues to exist

More on replication and deletion thresholds

- $\text{rep}(S, F)$ always chosen to be higher than the $\text{del}(S, F)$
- If a number of requests lie between deletion and replication threshold
  - File can only be migrated
  - Number of replicas for file should be the same
Reevaluating the placement of files at a server Q

- Check **access count** for each file
- If number of accesses < $del(Q, F)$?
  - File deleted unless it is the last copy
- For some server $P$, if $\text{count}_Q(P, F)$ is more than $\frac{1}{2}$ of requests for $F$ at $Q$?
  - Server $P$ is requested to **take over** copy of $F$
  - **Migration**

Migration/replication of a file may not always succeed

- Server $P$ might already be heavily overloaded
- $Q$ will then attempt to replicate $F$ **elsewhere**
  - Number of access > $\text{rep}(Q, F)$
- If $\text{count}_Q(R, F)$ exceeds a certain fraction of all requests for $F$ at $Q$
  - Try to replicate at $R$
Client initiated replicas: Client cache

- Temporarily store data that was just requested
  - Could be on client’s machine or nearby machine
- Used to improve access times
- Data kept in cache for a limited time
  - Avoid stale data problem
  - Make room for other data
- To improve cache hits; cache may be shared between clients

Replicated Write Protocols
Replicated write protocols

- Write operations are carried out at multiple replicas
  - Not just 1 (or primary)

- Active Replication
  - Operation forwarded to all replicas

- Quorum-based
  - Based on majority voting

Active Replication

- Operation is sent to each replica

- Must be carried out in same order everywhere
  - Lamport’s clocks
  - Use of a central coordinator: Sequencer
    - Could start to resemble primary-based protocols
Quorum-based protocols:
Clients must request and acquire permissions

- From **multiple** servers
- **Before** reading and writing replicated data items

Quorum-based protocols:
Distributed File System example \{Write\}

- File is replicated on \(N\) servers
- To update a file
  - Client must contact at least \((N/2 + 1)\) servers
  - Majority
  - Get them to agree to do the update
- Upon agreement
  - File is changed and version number incremented
Quorum-based protocols:
Reading a replicated file

- Client must contact at least \((N/2 + 1)\) servers
  - Ask them for version numbers of file
- If version numbers agree ... most recent version
- With \(N=5\), and
  - Clients see 3 responses with version-8
  - Then getting 2 responses with version-9?
    - Impossible, because update to version-9 needs 3 to agree

Quorum-based protocols:
When there are \(N\) replicas

- Read quorum \(N_R\)
- To modify a file, write-quorum \(N_W\)
- \(N_R + N_W > N\)
  - Prevent read-write conflict
- \(N_W > N/2\)
  - Prevent write-write conflict
**Quorum-based protocols:**

**Example 1**

\[ N_R = 3 \quad N_W = 10 \]

\[ N_R = 7 \quad N_W = 6 \]

*Read Quorum:*  
*Write Quorum:*

**Write-write conflict**

Concurrent writes to \{A, B, C, E, F, G\} and \{D, H, I, J, K, L\} will be accepted.

**Quorum-based protocols:**

**Example 2**

\[ N_R = 1 \quad N_W = 12 \]

*Read Quorum:*  
*Write Quorum:*
EVENTUALLY CONSISTENT

Werner Vogels: Eventually Consistent.

Amazon systems use replication techniques ubiquitously

- Predictable performance
- Availability
Replication helps with these goals, but ...

- Not necessarily transparent
  - Under a number of *conditions*, *consequences* of using replication techniques come to the fore
    - Network partitions
    - Node failures

Ideal world

- One consistency model
- When an update is made all observers see that update
Distribution transparency

- To the user of the system, it *appears* as if there is only one system
  - Instead of a number of collaborating systems

- Approach taken in such systems?
  - Better to fail the complete system rather than break this transparency

In the mid-90s these practices were revisited

- Larger internet systems

- For the first time, *availability* was being considered the most important property
Brewer’s CAP Conjecture (and later on ... Theorem)

Brewer’s CAP Theorem

- By Eric Brewer in 2000
- Three properties of shared-data systems
  1. Data consistency
  2. System availability
  3. Tolerance to network partitions
- There are limits to your choices of what can be achieved at a given time
Brewer’s CAP: Consequences

- In large-scale distributed systems, network partitions are common
- So, consistency and availability cannot be achieved at the same time

What is the trade-off? [1/2]

- If your application requires consistency?
  - And some replicas are disconnected from the other replicas due to a network problem …
  - Then some replicas cannot process requests while they are disconnected:
    - They must either wait until the network problem is fixed, or return an error
    - Either way, they become unavailable
What is the trade-off?  [2/2]

- If your application does not require consistency?
  - Then each replica can process requests independently
    - Even if it is disconnected from other replicas
  - The application can remain available in the face of a network problem, but its behavior is not consistent

- Thus, applications that don’t require consistency can be more tolerant of network problems

Characterizing CAP correctly  [1/3]

- CAP is sometimes presented as Consistency, Availability, Partition tolerance: pick 2 out of 3
  - Unfortunately, putting it this way is misleading

- Because network partitions are a kind of fault, they aren’t something about which you have a choice:
  - They will happen whether you like it or not
Characterizing CAP correctly [2/3]

- At times when the network (and system) is working correctly, a system can provide both consistency and total availability.
- When a network fault occurs, you have to choose between consistency OR total availability.

Characterizing CAP correctly [3/3]

- A better way of phrasing CAP would be Either Consistent or Available when Partitioned.
- A more reliable network needs to make this choice less often, but at some point the choice is inevitable!
CAP: Two choices on what to drop

- Relax consistency
  - To allow system to be available under partitionable conditions

- Make consistency a priority
  - And the system will be unavailable under certain conditions

The choices requires the developer to be aware of what is being offered by system

- If consistency is emphasized?
  - Developer must account for system unavailability
  - If a write fails?
    - Plan on what will be done with the data that must be written

- If availability is emphasized?
  - System may always accept writes but ...
    - Under certain conditions a read will not reflect the results of a recently completed write
The C in ACID is a different kind of consistency
{Atomicity, Consistency, Isolation and Durability}

- When a transaction is finished, the database is in a consistent state
- For e.g., when money is transferred between two accounts?
  - The total money in the two accounts should not change
- This kind of consistency is the responsibility of the developer writing the transaction
  - Database assists via managing integrity constraints

The “I” in ACID

- **Isolation**
  - Ensures *concurrent execution* of transactions results in a final system state similar to what would be achieved if transactions were executed serially
Consistency: Two ways to look at this

- Client-side
  - How do clients observe updates?

- Server-side
  - How do updates flow through the system?
  - What guarantees can systems give with respect to updates?
Client-side consistency

1. Consider a storage system
2. Process A that writes and reads from the storage system
3. Process B and C are independent of A
   - Write and read from the storage system too

Client-side consistency

1. How and when do observers (A, B, and C) see updates made to a data object?
2. **Strong consistency:**
   - After update completes, any subsequent access by (A, B, or C) will return updated value
3. **Weak consistency:**
   - No guarantee that subsequent accesses will return updated value
   - Number of conditions to be met before value is returned
The inconsistency window

- **Period** between
  - The *update*
  - And
  - When any observer will *always see* the updated value

Eventual consistency

- A form of *weak consistency*
- Storage system guarantees that if no new updates are made to the object?
  - *Eventually* all accesses will return last updated value
- If no failures occur, size of the inconsistency window is determined by:
  - Communication delays, system load, and number of replicas
Eventual consistency variations

- Causal consistency
- Read-your-writes consistency
- Session consistency
  - As long as session exists, system guarantees read-your-writes consistency
  - Guarantees do not overlap sessions
- Monotonic read consistency
- Monotonic write consistency

RDBMS implement replication in different modes

- **Synchronous**
  - Replica update is part of the transaction

- **Asynchronous**
  - Updates arrive at the backup in a delayed manner
    - Log shipping
  - If primary fails before the logs were shipped?
    - Reading from promoted backup will produce old, inconsistent values
Other RDBMS approaches to improve speed

- RDBMSs have also started to provide ability to read from backup
  - Classic case of eventual consistency

- Size of the inconsistency window in such a setting?
  - Periodicity of the log shipping

SERVER SIDE CONSISTENCY
Server-side consistency

- Based on how updates flow through the system
- \( N \): Number of nodes that store replicas of data
- \( W \): Number of replicas that need to acknowledge receipt of update before it completes
- \( R \): Number of replicas that are contacted when data object is accessed through read operation

\[ W + R > N? \]

- The write-set and read-set overlap
  - Possible to guarantee strong consistency

- Primary-backup RDBMS
  - With synchronous replication
    - \( N=2, W=2 \) and \( R=1 \)
    - Client always reads a consistent answer
  - With asynchronous replication
    - \( N=2, W=1 \) and \( R=1 \)
    - Consistency cannot be guaranteed
In distributed storage systems the number of replicas is higher than two

- Systems that focus on fault tolerance use $N=3$
  - With $W=2$ and $R=2$

- Systems that serve very high read loads
  - Replicate data beyond what is needed for fault tolerance
  - $N$ can 10s to 100s of nodes
  - $R$ will be set to 1
    - A single read will return the result
  - For consistency $W=N$ for updates
    - Decreases the probability of write succeeding

For systems concerned about fault tolerance but not consistency

- $W=1$
  - Minimal durability

- Rely on lazy (epidemic) techniques to update other replicas
Configuring values of N, R and W

- Depends on the **common case**
- **Performance path** that needs to be optimized
- If $R=1$ and $N=W$?
  - We optimize for the read case
- If $W=1$ and $R=N$?
  - We optimize for a very fast write
  - Durability is not guaranteed
  - If $W < \frac{(N+1)}{2}$ there is a possibility of conflicting writes when the write-sets do not overlap

Weak/eventual consistency

- Also arises when $W + R \leq N$
  - Possibility that the read and write set will not overlap
- If it’s deliberate and not based on failure cases?
  - Hardly makes sense to set $R$ to anything but 1
Weak/eventual consistency:
Two common cases where R=1

- Massive replication for read scaling
- When data access is more complicated
  - In simple <key, value> systems easy to compare versions to determine latest written value
  - When set of objects are returned, reasoning gets more complicated

When partitions occur

- Some nodes cannot reach a set of other nodes
- With a classic majority quorum approach
  - Partition that has W nodes of the replica set continues to take updates
  - The other partition becomes unavailable
For some applications unavailability of partitions is unacceptable

- Important that clients, that reach a partition, can progress
- Merge operation is executed when partition heals
- Amazon shopping-cart?
  - Write-always system
  - Customer can continue to put items in the cart even when original cart lives on other partitions

The contents of this slide-set are based on the following references