Final Review
Grading Update

• HW5 and MP2 graded
• This leaves:
  • HW6 (6%) + final (33%) for 3-credit
  • HW6 (0~4%) + MP3 (7% + 3% bonus) + final (25%) for 4-credit
• Participation grade (1%)
  • Will be used to bump people to next grade
  • Based on your activity in CampusWire
• Reminder:
  • CR/NC means course does not affect GPA, but grade < C- means no credit
  • If you keep a letter grade, anything >= D- is a pass
Final Logistics

Timed Exam: Wed, May 13, 8–11 a.m.
- Format similar to midterm 2
  - Exam released to you at 8 a.m.
  - Upload answer sheet to Gradescope by 11 a.m.
  - Open book, but individual work
  - Zoom room + Google doc for clarifications
- Topic coverage: 50% from MT1+MT2, 50% new material
- Exam structure similar to MT1/MT2
  - Some multiple choice
  - Some short answer / synthesis
  - ~50% longer
Topic coverage (Part III, 50% of final)

- DHTs
- RPCs
- Distributed transactions
  - Concurrency, isolation, and deadlocks
  - Atomicity and 2PC
- Combining 2PC and Paxos; Spanner and linearizability
- Cloud computing and MapReduce
- Distributed data stores and Cassandra
Topic coverage (Parts I and II, 50% of final)

• System model and Failures
• Failure Detection
• Clock Synchronization
• Event ordering and Logical Timestamps
• Global Snapshot
• Multicast
• Mutual Exclusion

• Leader election
• Consensus
  • Formulation
  • Synchronous consensus
• Paxos
• FLP Theorem
• Bitcoin
• Raft
Distributed Hash Tables / Chord

**Goal**: horizontal scaling to millions of peers and data items

- Consistent hashing of keys, node IDs into a large \(2^{128}–2^{256}\) key space
- Neighbor table: successors and fingers
- Finger-based routing:
  - \(O(\log N)\) neighbors and \(O(\log N)\) lookup steps
- Resilient structure to departing nodes / incorrect fingers
  - Track multiple successors and replicated keys at succ’s/pred’s
- Stabilization to restore DHT structure
RPC / RMI

**Goal**: Execute functions / procedures / methods on remote system, and implement *remote objects*

- Interface definition languages ()
- External data representation (JSON, protobuf, etc.)
  - Marshaling / unmarshaling arguments
- Dealing with failures (at least once, at most once, idempotence)
- Remote objects
  - Proxy objects (client)
  - Dispatcher and skeleton (server)
  - Remote reference module
Transactions & Isolation/Concurrency

**Goal**: support high-level updates to data while maintaining useful properties

- ACID properties (Atomicity, Consistency, Isolation, Durability)
- Isolation
  - Lost update problem
  - Inconsistent retrieval problem
  - Serial equivalence
  - Conflicts (R/W, W/W)
  - Conflicts follow tx ordering === serial equivalence
Concurrency Techniques

**Goal:** achieve isolation / serial equivalence of transactions

- Two-phase locking: acquire locks during tx, release during commit/abort
  - Exclusive or shared locks
  - Lock promotion
- Timestamp ordering: ensure operations consistent with tx timestamp
  - Don’t read data from newer tx’s
  - Don’t overwrite data from newer tx’s
  - Abort (or skip writes) if order not satisfied
- Optimistic vs. pessimistic concurrency
Deadlocks

**Goal**: Avoid or detect deadlocks

- **Deadlock requirements**:
  - no preemption
  - hold and wait
  - circular wait

- **Deadlock detection**
  - timeout
  - centralized
  - edge chasing

- **Deadlock avoidance**: lock ordering
  - E.g., wait-die / wound-wait
  - E.g., dependency lists in timestamped concurrency

- **Deadlock resolution**
  - Abort newer tx
Two-phase Commit

**Goal:** ensure atomic commit across distributed participants

- Problem: need *unanimous* agreement to commit
  - not majority like Paxos / Raft
- First phase: precommit
  - any participant who agrees to commit must be able to proceed
- Second phase: commit/abort
- Crash-recover semantics
  - Use participant or coordinator log after recovery
  - Sacrifice availability
Distributed, replicated txs

**Goal**: combine distributed tx’s (for horizontal scaling) with replication (for availability / durability)

- Run distributed transactions w/ 2PL, 2PC
- Replace each participant / coordinator with replica group
  - Use Paxos / Raft for replica consistency
- Expensive!
Spanner

**Goal:** make previous approach more efficient

- Maintain versioned / timestamped database
  - A log of each previous value of each object with tx timestamp
  - Enables read at a past time
- Transaction timestamp is *global time* that occurred during tx commit phase
  - Needs time with known bound on error
  - Commit wait to ensure property (can overlap with consensus protocols)
  - External consistency
- Reads can be lock free
What is a cloud?

- Cloud = Lots of storage + compute cycles nearby

- Cloud services provide:
  - managed clusters for distributed computing.
  - managed distributed datastores.
Must deal with immense complexity!

- Fault-tolerance and failure-handling
- Replication and consensus
- Cluster scheduling

- How would a cloud user deal with such complexity?
  - **Powerful abstractions and frameworks**
  - Provide **easy-to-use** API to users.
  - Deal with the complexity of distributed computing under the hood.
MapReduce Architecture

- *MapReduce programming abstraction:*
  - Easy to program distributed computing tasks.

- MapReduce programming abstraction offered by multiple open-source *application frameworks:*
  - Handle creation of “map” and “reduce” tasks.
  - *e.g. Hadoop: one of the earliest map-reduce frameworks.*
  - *e.g. Spark: easier API and performance optimizations.*

- Application frameworks use *resource managers.*
  - Deal with the hassle of distributed cluster management.
MapReduce Overview

- Input: a set of key/value pairs
- User supplies two functions:
  - $\text{map}(k,v) \rightarrow \text{list}(k, v_1)$
  - $\text{reduce}(k_1, \text{list}(v_1)) \rightarrow v_2$
- $(k_1, v_1)$ is an intermediate key/value pair.
- Output is the set of $(k_1, v_2)$ pairs.
MapReduce Execution

- **Map tasks**
  - Blocks from DFS
  - Servers
  - Shuffle (group by key and partition)
  - Barrier between map and reduce phases.

- **Reduce tasks**
  - A
  - B
  - C

- **Output files into DFS**
  - I
  - II
  - III

Resource Manager (assigns map and reduce tasks to servers)
Distributed Datastores

- **NoSQL datastores:**
  - Similar to databased (RDBMS), but,
    - lack schema and structure.
    - simplified API; might not support ‘joins’.
    - typically do support ACID semantics.
Distributed Datastores

• NoSQL datastores:
  • Similar to databased (RDBMS), but,
    • lack schema and structure.
    • simplified API; might not support ‘joins’.
Design Requirements

- High performance, low cost, and scalability.
- Avoid single-point of failure
  - Replication across multiple nodes.
- Consistency: reads return latest written value by any client (all nodes see same data at any time).
  - Different from the C of ACID properties for transaction semantics!
- Availability: every request received by a non-failing node in the system must result in a response (quickly).
  - Follows from requirement for high performance.
- Partition-tolerance: the system continues to work in spite of network partitions.
CAP Theorem

- **Consistency**: reads return latest written value by any client (all nodes see same data at any time).
- **Availability**: every request received by a non-failing node in the system must result in a response (quickly).
- **Partition-tolerance**: the system continues to work in spite of network partitions.
- In a distributed system you can only guarantee at most 2 out of the above 3 properties.
  - Proposed by Eric Brewer (UC Berkeley)
  - Subsequently proved by Gilbert and Lynch (NUS and MIT)
CAP Tradeoff

- Starting point for NoSQL Revolution
- A distributed storage system can achieve at most two of C, A, and P.
- Partition-tolerance important for distributed datastores:
  - choose between consistency and availability