Distributed Systems

CS425/ECE428

April 2 2021

Instructor: Radhika Mittal

Midterm 2 on Monday, April 5, 7-8:50pm

- Same format at Midterm 1.
- Revise the instructions shared on CampusWire.
- Syllabus: Everything covered beyond the syllabus of Midterm1 upto and including Raft.

Disclaimer for our agenda today

- Quick reminder of the relevant concepts we covered in class, that are included in second midterm.

- Not meant to be an exhaustive review!

- Go over the slides for each class.
    - Refer to lecture videos, textbook, and readings to fill in gaps in understanding.

Topics for second midterm

- Mutual Exclusion
- Leader Election
- Consensus
  - Synchronous Consensus
  - Asynchronous Consensus: Paxos, Raft

Topics for second midterm

- Mutual Exclusion
- Leader Election
- Consensus
  - Synchronous Consensus
  - Asynchronous Consensus: Paxos, Raft

Problem Statement for mutual exclusion

- ***Critical Section* Problem:**
  - Piece of code (at all processes) for which we need to ensure there is at most one process executing it at any point of time.

- Each process can call three functions
  - enter() to enter the critical section (CS)
  - AccessResource() to run the critical section code
  - exit() to exit the critical section

Mutual Exclusion Requirements

- Need to guarantee 3 properties:
  - Safety (essential):
    - At most one process executes in CS (Critical Section) at any time.
  - Liveness (essential):
    - Every request for a CS is granted eventually.
  - Ordering (desirable):
    - Requests are granted in the order they were made.

Analyzing Performance

- **Bandwidth:** the total number of messages sent in each *enter* and *exit* operation.
- **Client delay:** the delay incurred by a process at each enter and exit operation (when *no* other process is in CS, or waiting)
    - *We will focus on the client delay for the enter operation.*
- **Synchronization delay:** the time interval between one process exiting the critical section and the next process entering it (when there is *only one* process waiting).

Mutual exclusion in distributed systems

- Classical algorithms for mutual exclusion in distributed systems.
  - Central server algorithm
  - Ring-based algorithm
  - Ricart-Agrawala Algorithm
  - Maekawa Algorithm

Central server based

- A client process:
  - sends request to the central server when it wants to enter CS.
  - enters CS only after receiving a token from the server.
  - releases the token back to the server upon exiting CS.
- Server grants token to only one process at a time.

- Does it guarantee safety, liveness, and ordering?

- What is its bandwidth usage, client delay, and synchronization delay?

Ring based

- A single token moves around a logical ring of processes.
- A process holds the token while executing CS, and releases it when done.
    - It simply forwards the token if it does not want to enter CS.

- Does it guarantee safety, liveness, and ordering?

- What is its bandwidth usage, client delay, and synchronization delay?

Ricart-Agrawala Algorithm

- Send request to all processes and wait for reply from all.
- A process always replies back to a request, except when:
  - It is currently executing CS (in HELD state)
  - It wants to enter CS (in WANTED state) and deserves to enter it sooner.
    - The Lamport timestamp of its own request is smaller than the Lamport timestamp of the received request.
    - Use process ID to break ties.

- Does it guarantee safety, liveness, and ordering?
- What is its bandwidth usage, client delay, and synchronization delay?

Maekawa Algorithm

- Each process has a voting set consisting of a subset of processes.
- Intersection of voting set of any two processes must be non-zero.
- Send request to all processes in the voting set and wait for reply from all of them.
- A process replies back to a request only if it has not replied to (or voted for) a request from another process.

- Does it guarantee safety, liveness, and ordering?
- What is its bandwidth usage, client delay, and synchronization delay?

Topics for second midterm

- Mutual Exclusion
- Leader Election
- Consensus
  - Synchronous Consensus
  - Asynchronous Consensus: Paxos, Raft

Election Problem

- Goal:
  - Elect one leader only among the non-faulty processes
  - All non-faulty processes agree on who is the leader
- A run of the election algorithm must always guarantee:
  - **Safety**: For all non-faulty processes *p*, *p* has elected:
    - (q: a particular non-faulty process with the *best attribute value*) or Null
  - **Liveness**: For all election runs:
    - election run terminates
    - & for all non-faulty processes *p*: *p*'s elected is not Null
- At the end of the election protocol, the non-faulty process with the best (highest) election attribute value is elected.
  - Common attribute : leader has highest id

Calling for an Election

- Any process can call for an election.
- A process can call for at most one election at a time.
- Multiple processes are allowed to call an election simultaneously.
  - All of them together must yield only a single leader
- The result of an election should not depend on which process calls for it.

Two Classical Election Algorithms

- Ring election algorithm
- Bully algorithm

Key Metrics

- **Bandwidth usage:** Total number of messages sent.

- **Turnaround time:** The number of serialized message transmission times between the initiation and termination of a single run of the algorithm.

Ring-based algorithm

- Attribute circulated around a ring in an "election" message.
- If a process' own attribute is better than received attribute, overwrite the value before forwarding.
- If a process receives back its own attribute, it can declare itself as leader, and circulate the "elected" message.
- When multiple processes simultaneously call for an election?
  - What optimization proposed in Chang and Roberts algorithm reduces the number of messages exchanged?

- *What is bandwidth and turnaround time under different scenarios?*
- *What happens when a process fails?*
- *Can we achieve both safety and liveness in an asynchronous system?*

Bully algorithm

- Each process aware of process ids (attributes) of other processes.
- Send election message only to higher id process.
  - if response received, back off and wait for "coordinator" message.
    - If no "coordinator" message received after a timeout, restart the election.
  - If no response received after a timeout, assume all higher id processes are dead, and send "coordinator" message to all processes.
- If "election" message received from lower id process, send "disagree" and start another election run.

- *What are suitable timeout values?*
- *What is bandwidth and turnaround time under different scenarios?*
- *What happens when a process fails during an election run?*
- *Can we achieve both safety and liveness in an asynchronous system?*

Topics for second midterm

- Mutual Exclusion
- Leader Election
- Consensus
  - Synchronous Consensus
  - Asynchronous Consensus: Paxos, Raft

Basic Consensus Problem

- System of N processes (P_1, P_2, ….., P_n)
- Each process P_i:
  - begins in an *undecided* state.
  - proposes value $\mathbf{v_i}$.
  - at some point during the run of a consensus algorithm, sets a decision variable $\mathbf{d_i}$ and enters the *decided* state.

Required Properties

- **Termination (liveness):** Eventually each process sets its decision variable.

- **Agreement (safety):** The decision value of all correct processes is the same.
  - If P_i and P_j are correct and have entered the *decided* state, then $\mathbf{d_i = d_j}$.

- **Integrity:** If the correct processes all proposed the same value, then any correct process in the decided state has chosen that value.
  - *Safeguard against algorithms that decide on a fixed constant value.*

Synchronous Consensus

- Round-based algorithm
  - Proposed values exchanged over 'synchronized rounds".
  - In round i+1, each process P_k multicasts all new values it received in the previous round i.
- How many rounds needed to tolerate up to 'f' failures?

Asynchronous Consensus

- Can we achieve both safety and liveness for consensus in an asynchronous system?

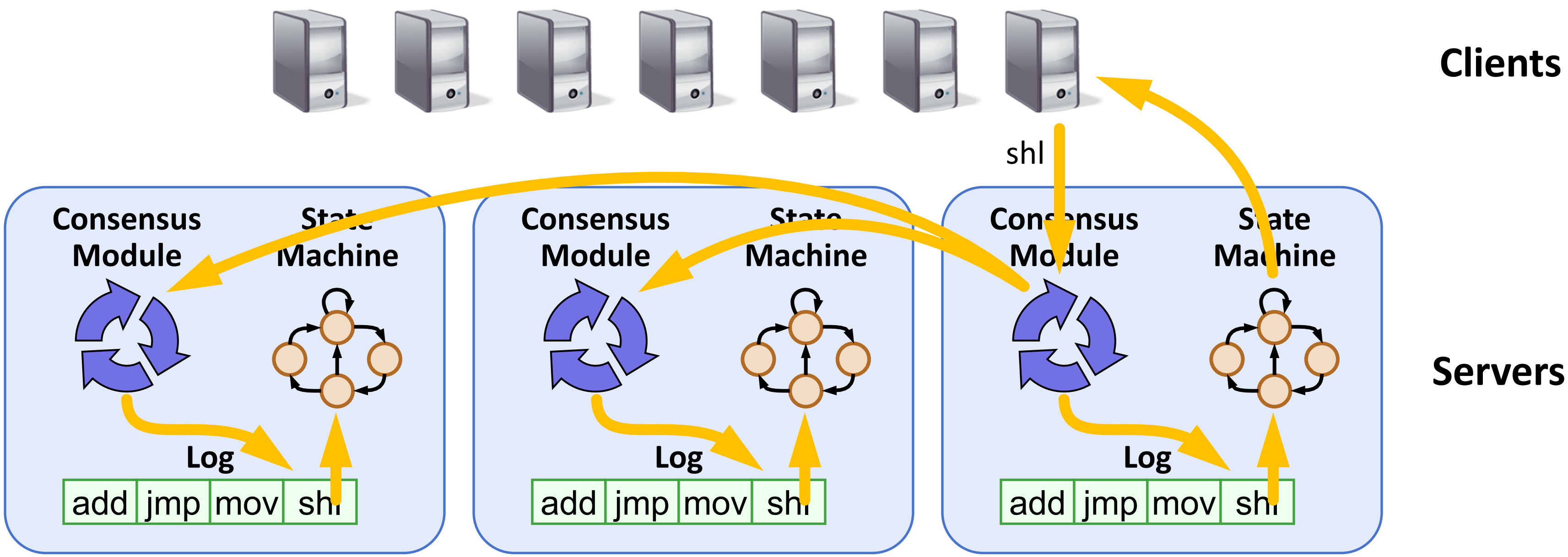
- Algorithms for asynchronous consensus.
  - Paxos, Raft

- What guarantees do they provide?

Paxos

- Three roles: proposer, acceptor, learner.
- Two phases:
  - Phase 1: *prepare* request and response.
    - When will an acceptor respond?
  - Phase 2: *accept* request (if applicable)
    - When will an accept request be sent?
    - What will be the proposed value?
- When is a value implicitly decided?
- How is the value shared with the learners?
- What is required to guarantee safety?

Replicated Log Consensus



- Replicated log => replicated state machine
  - All servers execute same commands in same order
- Consensus module ensures proper log replication

Raft

- Algorithm for log consensus. Designed for simplicity.

- What are the guarantees provided by Raft and how?
- How is leader elected?
    - Under what conditions will a process refuse to grant vote?
- What happens when a leader fails or gets disconnected?
- How are log entries appended?
- What leads to missing / extra entries in a server's log?
- When can log entries be overwritten?
- When can log entries be committed?

Notes on Model and Assumptions

- In a ring-based algorithm, ids of other processes and number of processes are not known.
- In Bully algorithm, all process ids (and attributes) are known, but a process may not know which processes have failed.
- In Paxos and Raft, total number of processes are known.
    - failed processes taken into account when counting for majority acceptor responses in Paxos.
    - failed processes taken into account when counting votes in Raft.
    - failed processed may come back up in Paxos and Raft: will remember the required state.

Topics for second midterm

- Mutual Exclusion
- Leader Election
- Consensus
  - Synchronous Consensus
  - Asynchronous Consensus: Paxos, Raft

Good luck!