SFU SIMON FRASER UNIVERSITY
ENGAGING THE WORLD

CMPT 431 Distributed Systems

Fall 2019

Failures & Consensus

https://www.cs.sfu.ca/~keval/teaching/cmpt431/fall19/

Instructor: Keval Vora

Reading

- [DC] Chapter 5.2.11
- [DC] Chapter 14
  - Selected topics

- Slides based on Distributed Systems course by Indranil Gupta @ UIUC

Failure Models

- Model how components in a distributed system may fail
- Distributed algorithms to solve any problem can vary dramatically depending on the failure model

- Fail-stop
  - Properly functioning process stops execution
  - Other processes learn about the failed process
- Crash
  - Properly functioning process stops execution
  - Other processes do not learn about the failed process

Failure Models

- Receive/Send Omission
  - Properly functioning process fails, and it only receives some of the messages that have been sent to it
  - Properly functioning process fails, and it only sends some of the messages it is supposed to send

- Byzantine or malicious failure
  - Process may (mis)behave anyhow, including sending fake messages
  - Most severe since it allows processes to arbitrarily change states

Availability Guarantees

- Why do distributed service vendors always only offer solutions that promise five-9s reliability, seven-9s reliability, but never 100% reliable?

- The fault does not lie with the companies themselves, or the worthlessness of humanity

- The fault lies in the impossibility of consensus

What is common to all of these?

A group of servers attempting to:

- Make sure that all of them receive the same updates in the same order as each other
- Keep their own local lists where they know about each other, and when anyone leaves or fails, everyone is updated simultaneously
- Elect a leader among them, and let everyone in the group know about it
- Ensure mutually exclusive (one process at a time only) access to a critical resource like a file

What is common to all of these?

A group of servers attempting to:

- Make sure that all of them receive the same updates in the same order as each other [Reliable Multicast]
- Keep their own local lists where they know about each other, and when anyone leaves or fails, everyone is updated simultaneously [Membership/Failure Detection]
- Elect a leader among them, and let everyone in the group know about it [Leader Election]
- Ensure mutually exclusive (one process at a time only) access to a critical resource like a file [Mutual Exclusion]

What is common?

- All of these are groups of processes attempting to coordinate with each other and reach agreement on the value of something
  - The ordering of messages
  - The up/down status of a suspected failed process
  - Who the leader is
  - Who has access to the critical resource

- All of these are related to the consensus problem

Consensus Problem

- N processes
- Each process p has
  - Input variable x_p : initially either 0 or 1
  - Output variable y_p : initially b (can be changed only once)

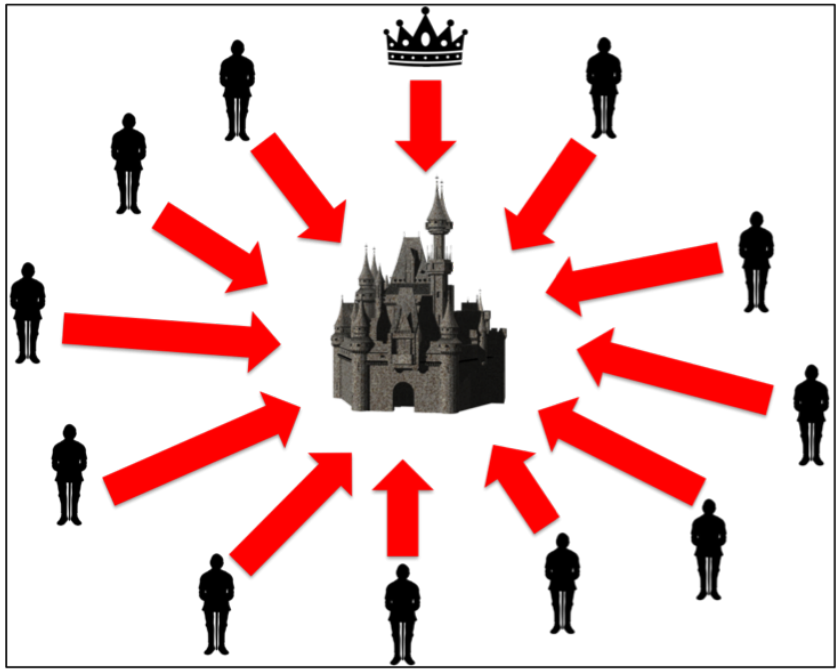
- Consensus problem is to design a protocol so that:
  - Either all processes set their output variables to 0 (all-0's)
  - Or all processes set their output variables to 1 (all-1's)

Consensus Problem

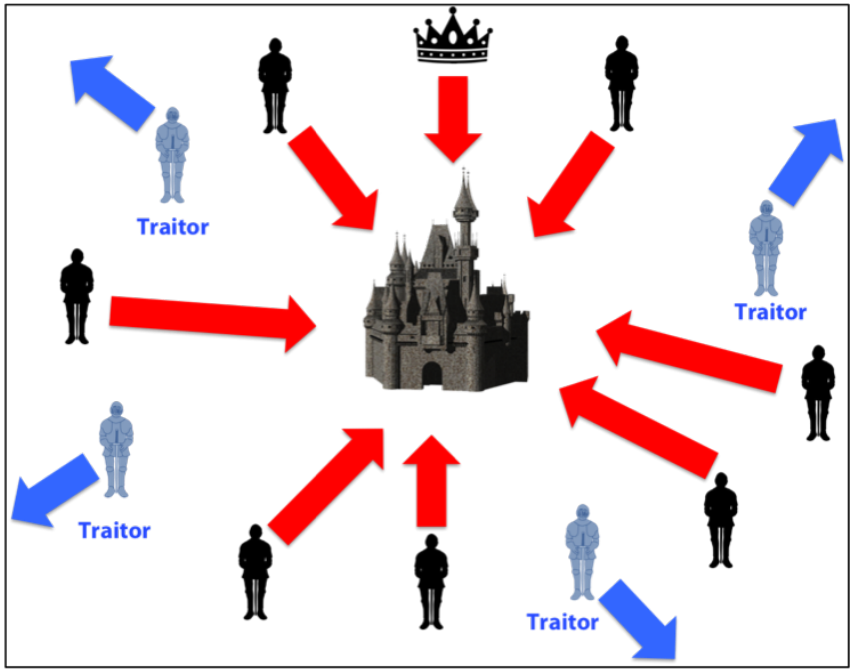
- Every process contributes a value
- Goal is to have all processes decide same (some) value
  - Decision once made can't be changed

- Constraints
  - Agreement: all non-faulty processes must agree on the same (single) value
  - Validity: only proposed values can be chosen
    (if all non-faulty processes proposed the same value v, then any non-faulty process must decide v)
  - Termination: each non-faulty process must eventually decide on a value

Byzantine Generals Problem



Coordinated Attack Leading to Victory



Uncoordinated Attack Leading to Defeat

PBFT: https://en.bitcoinwiki.org/wiki/PBFT

Why is Consensus Important?

- Many problems in distributed systems are equivalent to (or harder than) consensus!
  - Perfect failure detection
  - Leader election (select exactly one leader, and every alive process knows about it)
  - Agreement

- So consensus is a very important problem, and solving it would be really useful!
- How to solve consensus?

Models of Distributed Systems

- Synchronous Distributed System
  - Each message is received within bounded time
  - Drift of each process' local clock has a known bound
  - Each step in a process takes lb < time < ub
- Asynchronous Distributed System
  - No bounds on process execution
  - The drift rate of a clock is arbitrary
  - No bounds on message transmission delays

A protocol for an asynchronous system will also work for a synchronous system (but not vice-versa)

Consensus: Possible or Not

- In the synchronous system model
  - Consensus is solvable!

- In the asynchronous system model
  - Consensus is impossible to solve!
  - Whatever protocol/algorithm you suggest, there is always a worst-case possible execution (with failures and message delays) that prevents the system from reaching consensus
  - Powerful result (check out FLP proof)
  - Subsequently, safe or probabilistic solutions have become quite popular to consensus or related problems

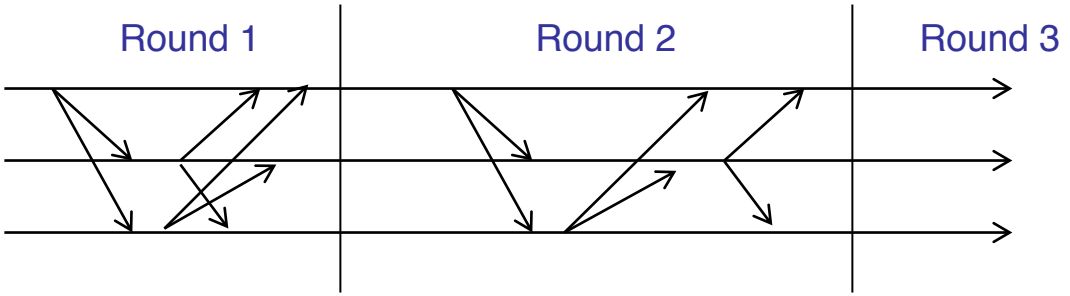
Consensus in Synchronous Systems

- Synchronous system has bounds on
  - Message delays
  - Clock drift rates
  - Max time for each process step

- Processes can fail by stopping (fail-stop or crash failures)

Consensus in Synchronous Systems

- For a system with at most f processes crashing
  - All processes are synchronized and operate in “rounds” of time
  - The algorithm proceeds in f+1 rounds (with timeout), using reliable communication to all members
  - $Values^r_i$: the set of proposed values known to p_i at the beginning of round r.



Consensus in Synchronous Systems

Initially $Values^0_i = \{\}$; $Values^1_i = \{v_i\}$

for round = 1 to $f+1$ do

  multicast ($Values^r_i - Values^{r-1}_i$) // send only changed values

  $Values^{r+1}_i \leftarrow Values^r_i$

  for each V_j received

    $Values^{r+1}_i = Values^{r+1}_i \cup V_j$

  end

end

d_i = minimum($Values^{f+2}_i$)

How to prove correctness?

Why does the Algorithm work?

- After *f+1* rounds, all non-faulty processes would have received the same set of Values
- Proof by contradiction: assume that two non-faulty processes, say p_i and p_j, differ in their final set of values (i.e., after *f+1* rounds)
- Assume that p_i possesses a value *v* that p_j does not possess
  - p_i must have received *v* in the very last round
    - Else, p_i would have sent *v* to p_j in that last round
  - So, in the last round: a third process, p_k, must have sent *v* to p_i, but then crashed before sending *v* to p_j
  - Similarly, a fourth process sending *v* in the last-but-one round must have crashed; otherwise, both p_k and p_j should have received *v*
  - Proceeding in this way, we can infer at least one (unique) crash in each of the preceding rounds
  - This means a total of *f+1* crashes, while we have assumed at most *f* crashes can occur
  - Contradiction!

Consensus in Asynchronous Systems

- Consensus impossible to solve in asynchronous systems

- Key to the FLP Proof
  - It is impossible to distinguish a failed process from one that is just very very (very) slow
  - Hence the rest of the alive processes may stay ambivalent (forever) when it comes to deciding

- Paxos algorithm

Paxos

- Most popular “consensus-solving” algorithm
- Does not solve consensus problem (which is impossible!)
- But provides safety & eventual liveness

- A lot of systems use it
  - Zookeeper (Yahoo!), Google Chubby, and many other companies

- Invented by Leslie Lamport

Paxos

- Paxos provides safety and eventual liveness
  - Safety: Consensus is not violated
  - Eventual Liveness: If things go well sometime in the future (messages, failures, etc.), there is a good chance consensus will be reached. But there is no guarantee.

- FLP result still applies: Paxos is not guaranteed to reach Consensus (ever, or within any bounded time)
- In practice, consensus is reached fairly quickly

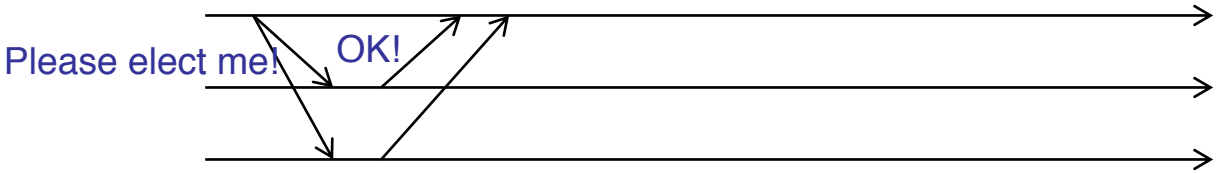
Paxos

- Paxos has rounds; each round has a unique ballot id
- Rounds are asynchronous
  - Time synchronization not required
  - If you're in round j and hear a message from round j+1, abort everything and move over to round j+1
  - Use timeouts (to move over rounds); may be pessimistic
- Each round is broken into phases (which are also asynchronous)
  - Phase 1: A leader is elected (Election)
  - Phase 2: Leader proposes a value, processes ack (Bill)
  - Phase 3: Leader multicasts final value (Law)

Paxos: Phase 1 - Election

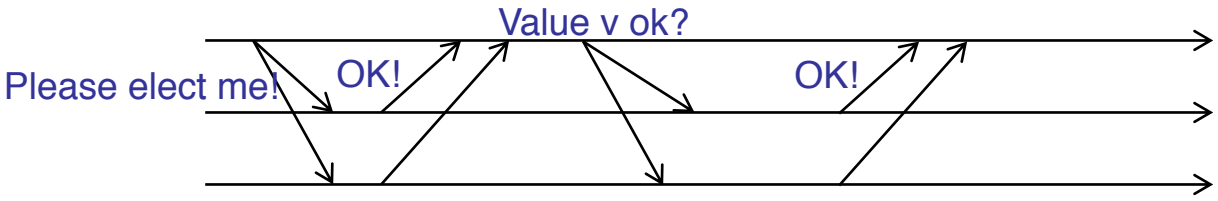
- Potential leader chooses a unique ballot id, higher than seen anything so far
- Sends election message to all processes
- Processes wait, then respond once to highest ballot id
  - If the potential leader sees a higher ballot id, it can't be a leader
  - Processes also log received ballot ID on disk
- If a process has in a previous round decided on a value v', it includes v' in its response
- If majority (i.e., quorum) respond OK then you are the leader
  - If no one has majority, start new round

(If things go right) A round cannot have two leaders (why?)



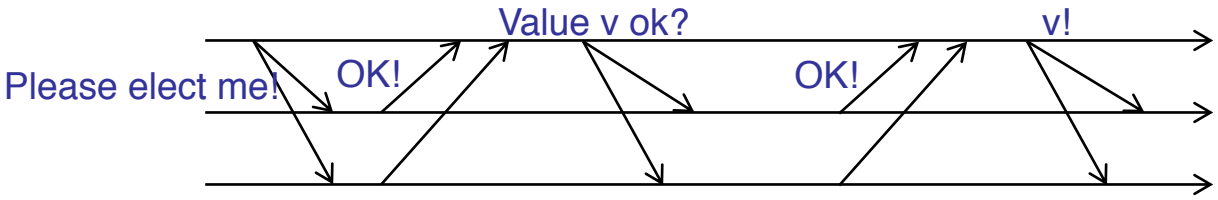
Paxos: Phase 2 – Proposal (Bill)

- Leader sends proposed value v to all
  - use v = v’ if some process already decided in a previous round and sent you its decided value v’
- Recipients log on disk and respond OK



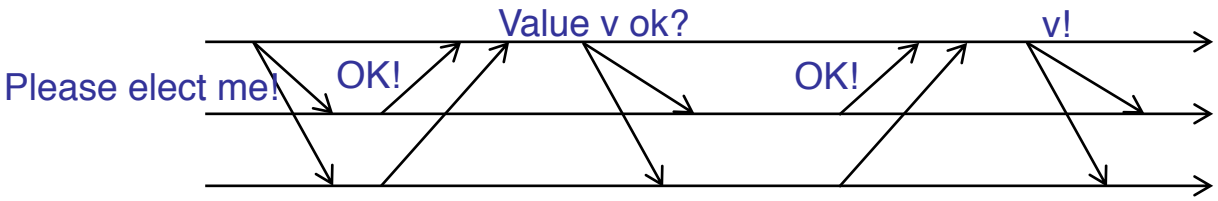
Paxos: Phase 3 - Decision (Law)

- If leader hears a majority of OKs, it lets everyone know of the decision
- Recipients receive decision, log it on disk



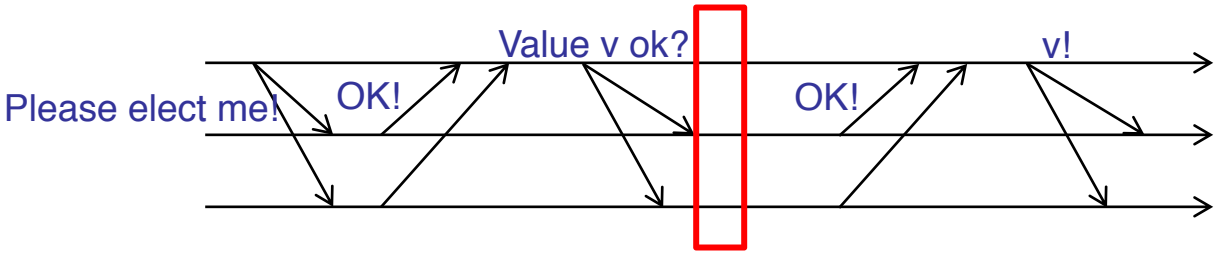
Paxos

- Where is the point of no return?
- That is, when is consensus reached in the system?



Paxos

- If/when a majority of processes hear proposed value and accept it (i.e., are about to/have respond(ed) with an OK!)
- Processes may not know it yet, but a decision has been made for the group
  - Even leader does not know it yet
- What if leader fails after that?
  - Keep having rounds until some round completes



Paxos: Safety

- If some round has a majority (i.e., quorum) hearing proposed value v' and accepting it (middle of Phase 2), then subsequently at each round, either:
  1. the round chooses v' as decision; or
  2. the round fails

- Key behind proof
  - Potential leader waits for majority of OKs in Phase 1
  - Success requires a majority, and any two majority sets intersect

Paxos: What could go wrong?

- Processes can fail
  - Majority does not include it
  - When process restarts, it uses its log to retrieve a past decision (if any) and past-seen ballot ids (tries to know of past decisions)
- Leader can fail
  - Start another round
- Messages can get dropped
  - If too flaky, just start another round
- Anyone can start a round any time
- Protocol may never end
  - If things go well sometime in the future, consensus is reached

Summary

- Consensus is a very important problem
- Consensus is possible to solve in a synchronous system where message delays and processing delays are bounded
- Consensus is impossible to solve in an asynchronous system where these delays are unbounded

- Paxos protocol: widely used implementation of a safe, eventually-live consensus protocol for asynchronous systems