Distributed Mutual Exclusion

- Picking up where we left off, we saw three different algorithms for distributed mutual exclusion:
  - Centralized server (PA#2 topic)
  - Token ring
  - Ricart/Agrawala

- Today we pick this back up with Maekawa's voting algorithm

Algorithm 4: Maekawa's Voting Alg.

- Based on the idea that clever partitioning of the processes into "voting sets" allows granting of access to critical section from only a subset of processes.
  - The trick is that every subset has a non-empty intersection with every other.
- We'll walk through this over two slides – it's a bit more complex, but quite clever.

Algorithm 4: Maekawa's Voting Alg.

- Initialization:
  - Everyone sets their state to released.
  - Everyone sets their voted flag to false.

- For a process p to enter the critical section:
  - Sets its state to wanted.
  - Multicast request to all processes in its voting set.
  - Wait until it hears back from everyone in its voting set.
  - Set its state to held when this occurs, and proceed into the CS.

Algorithm 4: Maekawa's Voting Alg.

- On receipt of message from p on q.
  - If q is in a held state or has voted
    - Queue the request from p, don't reply.
  - Otherwise
    - Reply to P
    - Set voted flag to true.

- When p exits CS:
  - Set state to released
  - Multicast release to all members of voting set

- On receipt of release from p on q.
  - If queued requests exist
    - Pop the queue, send a reply to it.
    - Set voted to true.
  - Otherwise
    - Set voted to false.

Algorithm 4: Maekawa's Voting Alg.

- The benefit of this algorithm is that Maekawa showed the optimal size of the voting sets is related to sqrt(N).
- So the number of messages is reduced to O(sqrt(N)), which is preferable to the other O(N) or worse solutions.
- Delay again related to round trip time.

Algorithm 4: Maekawa's Voting Alg.

- This algorithm has a potential problem.
  - Deadlock!
- V[1] = {p1, p2}. V[2] = {p2,p3}. V[3] = {p3,p1}
- Three processes concurrently ask for entry into the CS.
- Algorithm can be fixed though.
  - Queueing events in a "happened before" order. See book for reference to paper that talks about this.

Elections

- **Election**: Picking some process to play a special role ("server" or "master") from a set of peers.

- We saw an instance of this previously in the Berkeley time algorithm.
  - One machine is tagged as the server, and it coordinates synchronizing a set of peers.
  - If that machine goes away, the set of peers can elect a new machine to play server.

Elections

- Elected process is unique.
  - If two processes call an election at the same time, only one process will end up being the winner.
- The result of an election is the process with the largest identifier.
  - Identifier is some arbitrary, but likely useful piece of information, like 1/load.

Election requirements

- Safety
  - Each process will either have an undefined elected peer, or all will have the same non-crashed process at the end of the election with the largest identifier.

- Liveness
  - All processes will eventually determine who was elected, or crash.

Algorithm 1: Ring election

- All processes start as non-participants.
- One process calls the election.
  - Marks itself as a participant.
  - Sends an election message clockwise with its identifier attached.
- Receivers compare the content of the message with their own identifier.
- If it is greater
  - Forward the message
- Otherwise
  - If not already a participant
    - Substitute local identifier and pass on
  - Otherwise
    - Don't forward the message.
- Either way, mark self as a participant.

Algorithm 1: Ring election

- If the identifier of the received message is the same as the receiver, then receiver knows it was greatest.
  - Mark self as non-participant.
  - Send elected message with it's identity.

- Receive elected message
  - Set self as non-participant.
  - Store elected identifier.
  - Forward it if it wasn't the originator of the elected message.

Algorithm 2: Bully algorithm

- The bully algorithm allows for crashes.
  - The ring algorithm doesn't tolerate these very well since everyone only knew their neighbors.
- Synchronous model assumed, with timeouts to detect failures.
- The basic idea is that instead of the neighbor-only model of the ring, every process knows the processes above it assuming everyone has a unique identifier with a well defined order (<).

Algorithm 2: Bully algorithm

- A process who wants to start an election sends an election message to all processes above it.
  - If it receives an answer from at least one, it is not the coordinator.
- A process that receives the election message answers and attempts to contact those above it.
- This proceeds until a process receives no answers.
  - Either due to timeouts and process failures, or it being the maximum.
- This allows a single process to decide that it is coordinator.
  - Of course, we have to deal with slow processes. What happens if a process doesn't fail but just goes slow in answering?
  - Safety violation.

Multicast

- Multicast is a generic group communication operation.
  - IP multicast is simply one instance of it.
- The basic primitives are:
  - `Multicast(m,g)` where m is a message, and g is a group of participating processes.
  - `Deliver(m)` delivers the message on the process that received it.
- Messages in a multicast system are annotated with the sender and group that they are intended for.

Basic multicast

- This is the easiest multicast scheme.
- For all processes in the group, the sender sends a point-to-point message.
- Delivery is achieved with the basic receive operation.

- Unlike IP multicast, we assume the communication is reliable.
  - One can use IP multicast to implement this if a layer is placed over the UDP messages to do retries, duplicate handling, and order enforcement.
- Not efficient in its basic form. Bottleneck on the sender side.

Reliable multicast

- In basic multicast, we can see the following occur:
  - Sender starts its sequence of sends.
  - Part of the way through, it dies.
  - Some set of group members never get the message.

- Reliable multicast adds what is known as an "*Agreement property*".
  - If any member of the group delivers m, then all of the correct (not failed) processes in the group will eventually deliver m.

- We can achieve this on top of basic multicast.

Reliable multicast

- Start: Everyone initializes their received set to empty.
- The originator of the message uses basic multicast to send to the entire group, including itself.

- On the basic deliver occurring on a process q:
  - If the message is not in the received set on q:
    - Add it to the set
    - If q wasn't the originator of the message, basic multicast it to the group.
    - Successfully deliver the message reliably.

Reliable multicast

- Good property: As we can see, processes can die, but if any of them successfully delivered it, then everyone will eventually see it.

- Bad property: The message gets sent by every member of the group to everyone else!
  - Not really efficient.

Ordered multicast

- Ordering requirements on when messages are delivered.
  - **FIFO ordering**: If a correct process says multicast(g,m) and then multicast(g,m'), then every correct process that delivers m' will deliver m before m'.
  - **Causal ordering**: if multicast(g,m) happens before multicast (g,m') then any correct process that delivers m' will deliver m before m'.
    - Note that the happens before relation holds on the group, not just a process.
  - **Total ordering**: If a correct process delivers m before m', then any other correct process that delivers m' will deliver m before m'.

Consequences

- Causal ordering implies FIFO ordering (due to their happens-before relation in a single process).
- FIFO and causal orderings are only partial orders.
- For time reasons, we won't go into these algorithms today.
  - When you read the book though, you should see that use of logical clocks (either Lamport or Vector) makes it possible to negotiate the necessary ordering by attaching stamps to messages as they are moved around and ordered.

Consensus

- Goal: A set of processes agree upon a value after one or more propose one.
- Consensus is hard. In fact, situations can exist in which consensus cannot be achieved.
  - These situations are actually not rare or unexpected.
- Assume a set of processes communicating by messages.
- **Here is the wrinkle**: Consensus must be reachable in the presence of faults.
  - Assume some number of failures.

Consensus problems

- Every process starts undecided.
- Processes propose a value.
- Communication occurs, and a decision value is reached.
- Processes reach the decided state when they receive this final decision message.

Properties

- **Termination**: Eventually every correct process sets its decision variable.
- **Agreement**: All correct processes in the decided state will have the same decision variable value.
- **Integrity**: If a value is agreed upon, then the value was proposed by some process.
- **Validity**: If all processes propose the same value, then every correct process chooses that value.

Simple case

- No failures.
- Everyone multicasts proposed value to everyone else.
- Everyone collects these until all processes see all proposals.
- Some majority function is evaluated to determine the winner, or some special value if no majority winner exists.

Byzantine generals problem

- Proposed by Lamport.
- Three or more generals are to agree to attack or retreat.
- One general, the commander, issues the order.
- The others, are to decide to attack or retreat.
- The complication: one or more generals can be treacherous – faulty.
  - E.g.: Tells one general to attack, another to retreat.
- Differs from "pure" consensus because one special process initiates the orders.

Interactive consistency

- Like consensus, except a vector of values per process is agreed upon.
- E.g.: Let each of a set of processes obtain the same information about their respective states.

Relationships

- These are all related:
  - IC from BG: Run BG N times, once with each process acting as commander. In other words, one BG run per vector entry.
  - C from IC: Run IC, and then compute the majority function on each element of the vector on all processes.
  - BG from C: Commander sends proposed value out and to itself. All processes then run consensus.

- In systems with potential crashes, consensus is equivalent to totally ordered multicast.
  - All processes multicast their value.
  - Each process chooses the first value that it delivers.

Synchronous consensus

- Assume f failures tolerated.
- Everyone initializes a values array v[1] to their value, and creates an empty prior-step array, v[0] = {}.
- For f+1 rounds:
  - B-Multicast the values that are in the current values are in v[i] and not in v[i-1].
  - Set v[i+1] to v[i]
  - Accumulate up received messages from others, union received value set with v[i+1].
  - Increment i.
- After f+1 rounds, assign the decided upon value on each process to the minimum of v[f+1].

Byzantine generals problem

- What is the main idea behind this problem?
- How to detect a system that is faulty.
  - Either by malfunction of lying.
- Proofs that the book refers to for impossibility of detection for N<=3f, where f is the number of failures and N is the number of participants.

- **Example**: Three generals (X,Y,Z). Commander X tells Y to attack, Z to retreat. Y tells Z attack, Z tells Y retreat. Y and Z can't tell if the commander was lying, or if the other general is lying.
  - With 4 you can tell who was lying assuming only one liar.
- Easier to figure out if signatures used.
  - Z can forward message from X to Y, and Y can tell if Z is honest if the signature is preserved and that X told Z something different than it told Y.

Asynchronous consensus

- Asynchronous systems have no known bounds on times.
  - So, a very long message wait period can look like a failure.
- Methods exist to work around this, through fault detection, fault masking, etc…
- In any case, consensus in an arbitrary asynchronous system is impossible.