Outline

- Coordination
  - Distributed mutual exclusion
  - Election algorithms
- Multicast
- Message ordering
- Agreement
  - Byzantine generals problem
  - Paxos algorithm
- Summary
**Coordination and Agreement**

- Processes often need to coordinate their actions and/or reach an agreement
  - Which process gets to access a shared resource?
  - Has the master crashed? Elect a new one!
  - Failure detection – how can we know that a node has failed (e.g., crashed)?

**Failure detection**

- Keepalive messages every X time units
  - No message: the process has failed
  - Crashing immediately after sending?
  - Message delays, re-routing, etc.
- Unreliable failure detection
  - Unsuspected vs. Suspected
- Reliable failure detection
  - Unsuspected vs. Failed
  - Synchronous systems

**Distributed mutual exclusion**

- ME1 (safety): at most 1 process may enter the critical section at a time
- ME2 (liveness): requests to enter the critical section eventually succeed
- ME3 (ordering): if a request to enter the critical section happened before another, then access is granted according to that order
Mutex algorithms

- Central server
  - Simple and error-prone!
- Ring-based algorithm
  - Also simple, but not single point of failure
- Ricart and Agrawala
  - Multicast request, access when all have replied (ordered using logical clocks)
- Maekawa's voting algorithm
  - Ask only a subset for access: works if subsets are overlapping

Mutex: Central server

- Send request to server, oldest process in queue gets access, return token when done
- Safety?
  - Yes!
- Liveness?
  - Yes (as long as server does not crash)
- → ordering?
  - No! Why not?

Mutex: Ring-based

- Token is passed around a ring of processes
  - Want access? Wait until token comes, and claim it (then pass the token along)
- Safety? Liveness?
  - Yes (assuming no crashes)
- → ordering?
  - Not even close
Mutex: Ricart and Agrawala

- Each process
  - Has unique process ID
  - Maintains a logical (Lamport) clock
  - Has state $\in \{\text{wanted, held, released}\}$
- Requests are multicasted to group
  - Contain process ID and clock value
- Lowest clock value gets access first
  - Equal values? Check process ID!

Mutex: Ricart and Agrawala

- Have access or want it yourself, and your $<\text{id}, \text{value}>$ is lower than incoming request? Queue the request! Else, reply to it
- Safety? Liveness? $\rightarrow$ ordering?
  - Yes!
- Improve performance
  - If process wants to re-enter critical section, and no new requests have been made, just do it!

Mutex: Maekawa’s voting

- Optimization: only ask subset of processes for entry
  - Works as long as subsets overlap
  - Vote only in one election at a time!
- Safety?
  - Yes!
- Liveness? $\rightarrow$ ordering?
  - No, deadlocks can happen! But, enhance using vector clocks, and we get both!
Comparison of mutex algos

- Study the algorithms yourselves, including their performance
  - This is an excellent question for exams (just sayin’)
- Message loss?
  - None of the algorithms handle this
- Crashing processes?
  - Ring? No! Others? Only if the process is unimportant (not server, in voting set)

Election algorithms

- Ring-based
  - You may be familiar with this one!
- Bully
  - Very specific requirements, not generally applicable

Election requirements

- E1 (safety): A participant has elected = False or elected = P, where P is chosen as the non-crashed process with the highest identifier
- E2 (liveness): All processes participate and eventually set elected to non-False or crash
**Election: Ring-based**

- Processes have unique identifiers
- During election, pass \( \max(\text{own ID}, \text{incoming ID}) \) to next process
  - If a process receives own ID, it must have been highest and may send that it has been elected
- Safety? Liveness?
  - Yes!
- Tolerates no failures (limited use)

**Election: Bully algorithm**

- Requires
  - Synchronous system
  - All processes know of each other
  - Reliable failure detectors
  - Reliable message delivery
- Allows
  - Crashing processes
- Safety? Not if process IDs can be reused!
- Liveness? Yes (if message delivery is reliable)

**Election algorithms**

- Study these algorithms on your own, and their properties
  - Again, excellent exam material...
- Want to read more about non-trivial election algorithms?
  - [http://www.sics.se/~ali/teaching/dalg/l06.ppt](http://www.sics.se/~ali/teaching/dalg/l06.ppt)
Multicast

- Receive vs. Deliver
  - Receive: message has arrived and will be processed
  - Deliver: message is allowed to reach upper layer

- Closed vs. open groups
  - (Dis-)allow external processes to send to group

- Unreliable (basic) multicast
  - Send (unicast) to each other process!
    - What if sender fails halfway through?

Reliable multicast

- Integrity: messages delivered at most once

- Validity: if a correct process multicasts message m, it will eventually deliver m

- Agreement: if a correct process delivers m, then all correct processes will eventually deliver m

Reliable multicast

- Use basic multicast to send to all (including self)
  - When basic multicast delivers, check if message has been received before
    - If it has, do nothing further
    - If not, and sender is not own process:
      - Basic multicast message to others
      - Deliver message to upper layer
Reliable multicast

- Integrity? Validity? Agreement?
  - Yes!
- Insane amounts of traffic?
  - Yes! Every message is sent sizeof(group) to each process!
  - A single message will be sent 100 times if we just have 10 processes

Reliable multicast over IP multicast

- IP multicast requires hardware support
  - IP, however, is unreliable
- Study algorithm on your own
  - Perfect exam material...

Message ordering

- FIFO
- Total
- Causal
- Hybrids such as Causal-Total
**FIFO ordering**

- Intuition: messages from a process should be delivered in the order in which they were sent
- Solution: let sending process number the messages and hold back those that have been received out of order

**Total ordering**

- Intuition: messages from all processes should get a groupwide ordering number, so all processes can deliver in a single order!
  - Mental pitfall: the order itself does not have to make any sense, as long as all processes abide by it!

**Implementing total ordering**

- Sequencer
  - Simple
  - Central server (= single point of failure)
- ISIS-algorithm
  - Not as simple
  - Distributed
  - Study on your own!
**Total ordering via sequencer**

- Sequencer is logically external to the group
- Messages are sent to all members and the Sequencer
  - Initially, have no "ordering" number
- Sequencer maps message identifiers to ordering numbers
  - Multicasts mapping to group
    - Once a message has an ordering number, it can be delivered according to that number

**Total ordering contd.**

- Note, again, that the ordering is completely up to the sequencer
  - It could collect all messages for half an hour and then assign numbers according to how many "a"s there are in the message
  - While annoying to use, this is still a total order, and all processes will have to follow it!

**Causal ordering**

- Intuition: captures causal (cause and effect) relationships via happened-before ordering
- Using vector clocks, ensures that replies are delivered after the message that they are replying to
- Example: USENET replication
Hybrid orderings

- Causal order is not unique
  - Concurrent messages
- ...neither is FIFO
  - FIFO only guarantees per process, not inter-process
- Total order only guarantees a unique order
  - Combine with others to get stronger delivery semantics!

Message ordering

- You will need to get familiar with these orderings during the assignment
  - Read the book carefully
    - Avoid mistakes
    - Adhere to definitions
    - Get a few very nice hints...

Agreement

- Consensus
  - Any process can suggest a value, all have to reach consensus on which is correct
- Byzantine generals problem
  - One process can suggest a value, others have to agree on what that value was
Byzantine Generals problem

- Commander
  - Issues order to attack or retreat
- Lieutenants
  - Decide what to do
  - Coordinate their actions by repeating the order from the commander to each other
- One or more processes (commander or lieutenants) may be “treacherous” and report wrong/inconsistent values!

Byzantine Generals problem

- Synchronous system!
  - Impossible with three processes
    - See Figure 12.19 in the book, © Pearson Education 2005
  - Possible with \( N \geq 3f + 1 \) processes, where \( f \) is amount of treacherous ones
    - See Figure 12.20 in the book, © Pearson Education 2005

Byzantine Generals problem

- Asynchronous system – bad!
  - No timing constraints
- Fischer's impossibility result
  - Even with just one crashing process, we cannot reach consensus in an asynchronous system
  - This also applies to total ordering of messages and reliable multicast...
  - Still, we manage to do quite well in practice – how can that be?
Impossibility work-arounds

- Mask the faults
  - Use persistent storage and allow process restarts
- Use failure detectors
  - No reliable detectors: but good enough (agree that process is crashed if too long time has passed)
  - Eventually weak failure detector
- Randomization
  - Great paper (which is required reading) on the literature page!

Paxos algorithm

- Properties:
  - Non-triviality: only proposed values can be learned
  - Consistency: at most one value can be learned (two learners cannot learn different values)
  - Liveness (C,L): if a value C has been proposed, then eventually learner L will learn some value
- Asynchronous

Paxos algorithm contd.

- Client
  - Initiates with a request, expects response
- Acceptor
  - Collected into Quorums (groups)
    - Messages must be sent to a whole quorum
    - Messages are ignored unless they come from the whole quorum
- Proposer
  - Handles request on behalf of Client
- Learner
  - Acts if Acceptors agree on a value
- Leader
  - Distinguished Proposer, handles protocol
Paxos Quorums

- A quorum contains a majority of available Acceptors
- Several subsets of Acceptors such that any pair of Quorums always have some overlap

Paxos deployment

- Each Client request starts a new instance
- Each instance has two successful phases per round
  - If a phase is unsuccessful, start a new round
- Output from an instance is a single agreed-upon value

Paxos phases

- Prepare (1a)
  - Leader proposes number N
- Promise (1b)
  - If N is greater than any previous proposal, Acceptors in some quorum promise to reject proposals less than N
  - Acceptors respond with the last (highest) value they have accepted
- Accept (2a)
  - Leader chooses value from response set, sends value to some quorum of Acceptors
- Accepted (2b)
  - If Acceptors have not promised to do otherwise (promised to another concurrent proposal > N), they accept the value to Learners and the Leader
Paxos further reading

- Leslie Lamport Paxos Made Simple
  ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001) 51-58.
  - PDF here
  - Well worth your time...

Summary

- Failure detection
  - Hard to get right in async. systems
- Mutual exclusion
- Election algorithms
  - Seems like a simple problem, but non-trivial solutions are non-trivial
- Multicast, reliable and unreliable
- Message ordering
  - Very important for the assignment

Summary

- Byzantine generals problem
- Fisher's impossibility result
  - Avoiding it by various techniques
- Paxos algorithm
Next lecture

- Replication
  - Group communication
  - Fault-tolerant services
    - Active and Passive replication