Distibuted Algorithms Exam Study Notes

Paul Gribelyuk (pg1312, a5)

May 16, 2013

1 Introduction slides

- Characterizing good models: *accuracy* (how true are the properties compared to real world) vs *tractability* (can be analyzed)
- Modeling costs of an algorithm: state size, complexity, amount of communication
- Modelling elements:
  - *process* is a computational resource
  - *channel* is a communication link between processes; used to transport messages
- Communication Operations:
  - send: sends message m "atomically" from P_i to P_j; invoked by sender
  - receive: invoked by receiver and is a *state change* for receiver
  - broadcast: a combination of send operations to all processes; not atomic and not deterministic
- Problems in Distributed Algorithms: reaching a decision (consensus on the same value or decision about a process being different, i.e. a leader), detecting remote failures, tolerating process or communication failures, measuring progress
- Keywords for consensus:
  - *Agreement*: all processes decided the same value
  - *Termination*: all processes make a decision in finite time
  - *Validity*: the decided value must be one that was proposed
- With a lossy channel, agreement is impossible, even among 2 processes. To prove this, use contradiction, assume it is possible, and notice that the sender of the last message in the communication sequence cannot rely on it being delivered, and thus, his actions cannot depend on the contents of that message. Therefore, that message is not necessary, therefore, the optimal sequence of messages is shorter. Iterative argument shows there is no optimal sequence of messages
- Failure types:
  - *Failstop* A process fails by stopping (halting) and remains stopped. This can be detected externally
  - *Crash* Same as failstop by other processes may not be able to detect the failure
  - *Crash+Link* Needs further explanation

  - *Receive Omission* Process does not receive all messages
  - *Send Omission* Process does not send all messages it wants to
  - *General Omission* Combination of receive and send omissions
  - *Byzantine* Failure by doing random behavior
- Channels:
  - *Reliable*: A message is received if the sending process has not failed at tiem of send and receiving process has not failed at time of arrival
  - *Quasi-reliable*: A message is received if both sending and receiving processes have not failed prior to arrival of message
  - *Unreliable*: There is no guarantee of a sent message being received
- 3 Properties of Failure-free networks:
  - *Liveness* - each process will execute another step eventually
  - *Safety* - messages are received exactly as many times as they are sent
  - *Liveness*(again) - receipt of a message is guaranteed if process will take infinitely many steps
- Robust algorithms assume permanent process failures and use failure detection
- Stabilizing algorithms assume processes will be eventually correct and work with transient failures, i.e. non-permament
- Requirements for decision problems: termination and consistency (all decide the same value for either leader or consensus)
- Reliable Atomic broadcast means all processes receive their messages in correct order as they were sent; equivalent to solving consensus
- Types of distributed networks:
  - Synchronous: there are known upper bounds on computation delays and message delays and processes have synchronized clocks; this allows them to be able to estimate how long each "round" should take and if they don't receive a message from another process, they know for certain that it has failed. Solving election is easy: each process with $PID = i$ waits $i - 1$ rounds and receives messages. If it receives a message before round i, then it decides the leader to be the sender of that message. Otherwise, it broadcasts its own UID in round i.
  - Asynchronous: Unbounded execution time, unbounded message sending/receiving time, unsynchronized clocks. In this case, leader election is hard

2 Synchronous Algorithms slides

We use the synchronous network model of definite upper bounds on processing and communication delays with synchronized clocks. The network model is one of a graph $G = (V, E)$ of verteces and edges.

- $distance(i, j)$ is the shortest path from i to j
- $diameter(G) = max_{i,j} distance(i, j)$

A process is a state machine with following properties:

- *states*
- *start* (initial states)
- *msg* function to generate messages (function from state space to output message)
- *trans* is transition function from state to state (function from state space together with message space to state space)

A channel can hold a message from message space M or null (no message).
Synchronous execution happens in *rounds* with the steps:

1 generate outgoing message using *msg* at each process

2 use *trans* at each process to obtain new state (using current state and incoming message).

Communication failures resulting in message omission are modeled as null messages in channels.
A synchronous execution is an infinite sequence:

$$C_0, M_1, N_1, M_2, N_2, C_2, \ldots$$

where:

- C_r is state at round r
- M_r is messages sent at round r
- N_r is messages received at round r

Identical processes arranged in a ring network cannot solve leader election (proof by contradiction: if exists and execution sequence leading to election, then all processes reach the decision they are the leader at the same time), therefore we need UID to distinguish them (Lynch p.27)

2.1 Leader Election In a Ring

LCR algorithm

- unidirectional communication
- processes don't know size of ring
- each process has unique UID

Algorithm:

- Each process forwards UID to neighbor
- Each process discards received UID if it's less that their own
- Each process forwards UID if received UID is greater than their own
- If process receives their own UID, then they're the leader

Proving correctness:

1 The *snd* function of a process sending values to process r hops away from max-UID-valued process is u_{max} for any r. Thus, if $r = n - 1$, then i_{max} receives its own UID and declares itself the leader

2 No other process sees their own UID because i_{max} acts as a barrier to forwarding messages (since it has the highest UID).

3 Algorithm terminates by the leader sending a "halt" message around.

Analysis:

- Time Complexity: $2 \cdot n$ if counting n rounds to send "halt" message around
- Communication Complexity: $\frac{n(n-1)}{2} + n \approx O(n^2)$

2.2 Leader Election in General Network

FloodMax solves leader election on any connected graph by asuming processes know upper bound on diameter (D_{max}) of network. Algorithm:

- Each process maintains maximum UID seen
- Each process broadcasts maximum on all available channels
- If after D_{max} rounds, a process has same maximum seen as its own UID, then it is the leader.

So, each process keeps track of its own UID, the max seen UID, the stat ($\{leader, unknown, follower\}$), and number of rounds. *msg* function places max seen on all output channels so long as number of rounds less than D_{max}. *trans* function updates number of rounds by 1, receives messages from all input channels, updates max seen and makes decision about whether it should update its stat variable (initially set to unknown).
Analysis:

- Need to prove correctness
- Time complexity is D_{max} rounds
- Communication complexity is $D_{max} \cdot |E|$ sice $|E|$ messages get sent in each round.

Another approach: *FloodMaxOpt* cuts down on message complexity by only forwarding the max seen if it has been updated by incoming values. Need to keep another boolean value to keep track of that.

2.3 Breadth-first search in General Network

SyncBFS
Setup: processes unaware of size of network and laid out in general graph.
Problem is to find the *spanning tree* starting from a specific node i_0

- Each process receives probe messages on incoming channels.
- If that process is unmarked, it marks itself and declares one of the senders of the probe messages its parent
- If a process marked itself in the previous round, then it sends a probe on all outgoing channels

Analysis:

- Time complexity: number of rounds

- Communication Complexity: $O(|E|)$

To account for processes knowing their children (as well as parents), allow for back-channel replies from processes to which probe messages are sent ("child" or "not child"). Initially, the reply will be "not child", but when a marked process gets "child" replies from all its outgoing neighbors, it replies "child" to its parent. Thus when i_0 gets a reply from all outgoing channels, algorithm terminates.
Spanning tree can be used to perform efficient broadcast, or compute distance to each node.

3 Atomic Commitment (AC) slides

Distributed transactions are responsible for accessing data at multiple (distributed) sites and bringing state of the whole system from one consistent state to another. How to do this in the presence of failures of individual nodes? *Partial failures* can lead to inconsistency. Definitions:

- *Agreement*: all processes decide the same thing (even failed ones!, Lynch p. 183)
- *Termination*: all correct processes make a decision in finite time (this is strong termination Lynch p. 184)
- *Abort Validity*: If one process wants to abort commit, then that has to be the decision
- *Commit Validity*: If all processes are non-faulty, and they all vote "commit", only then is commit possible

Many similarities to the consensus (C) problem: only difference is that we have added restrictions on the binary outcomes: propose(commit) and propose(abort).
Two-Phase Commit:

1 All processes send the coordinator their vote.

2 Coordinator decides whether abort or commit and sends that back to all processes

Axioms for deciding coordinator:

AX1 At most one participant becomes a coordinator

AX2 Wihtout failures, exactly one participant becomes coordinator

AX3 No process assumes coordinator role after some constant Δ_c time from start of transaction (why is this needed?)

From Lynch p. 184, Two-Phase Commit solves weak termination (blocking termination) since we assume no failures. If coordinator fails and has "abort" while all others have "commit", then it is not clear how decision can be made in two-phase-commit. Desired properties of an atomic commitment protocol:

AC1 All deciding participants reach same conclusion

AC2 Any paricipant deciding "commit" imples all participants voted "YES"

AC3 If all votes are "YES" and no failures then we have a "commit"

AC4 Each participant decides 1 or 0 times.

Algorithm:

1 only invoker executes a send to ask all processes to find out if they willing/able to commit

2 each process sets their vote variable accordingly

3 they initiate atomicCommitment call

3 In atomicCommitment, only coordinator sends a VOTE-REQUEST to all and waits a maximum of 2∂ for replies

4 All processes receive VOTE-REQUEST and send their vote variable tagged as VOTE

5 If process voted NO, then it sets own decision to ABORT

5 Coordinator decides the outcome and broadcasts it to all processes

6 If timeout is reached, coordinator broadcasts ABORT

An aside about broadcast protocols:

B1 If correct process broadcasts m, then all correct processes eventually deliver(m)

B2 Any message is delivered at most once and only if it was initially broadcast

B3 Exists an upper bound on delivery delay, Δ_b, so no process deliver(m) after that time

Proof that Two-Phase-Commit works:

- To prove AC2, need to look at all paces where a process decides "commit". This happens only if deliver(commit) happened. This message was broadcast by coordinator, and it must have been a "YES".
- To prove AC3, straightforward by following code and assuming that all participants voted "YES" and no failures
- To prove AC4, note that the protocol only allows each participant to make at most one decision
- To prove AC1, suppose that is not the case. Only places for a process to decide "ABORT" are when it voted "NO", when it received "ABORT" decision, and upon timeout. Since other process decided "COMMIT", means coordinator received "YES" votes from everyone, therefore, only way for a process to abort when everyone else committed is if it received an abort message from coordinator. Since coordinator sent only one message, this is contradiction.

This is a blocking protocol since it's possible that no decision is made in some executions (i.e. when coordinator dies). To get nonblocking, need to include assumptions about atomic commit and about broadcast:

AC5 Every correct participant eventually decides

B4 If any process deliver(m), then all correct processes eventually deliver(m)

4 Byzantine Consensus slides

- Ånalogy to failed processes is the loyal generals problem
- Commanding general sends order to other generals (retreat or attack)
- Loyal generals obey orders, traitors cannot be guaranteed to follow orders

Interactive consistency:

IC1 Correct processes receive same message

IC2 If senders is correct, then message sent = message received

Impossibility result: If m processes are traitors, then we need $3m+1$ total processes to gain consensus. Communication assumptions:

A1 Every sent message is delivered correctly

A2 Receiver of message knows identity of sender

A3 Absence of message is detectable

Lamport's solution was as follows:

- $U(n,m)$ for n generals and m traitors
- exists a v_{def} as a global default if no messages arrive from traitorous general
- exists function $majority(v_1,\ldots,v_{n-1})$ to calculate most prevalent vote (v_{def} if tie)
- Algorithm:
  - General G sends v to all lieutenants L_i
  - Each lieutenant L_i sends v_i (value received) to $n-2$ other lieutenants using this algorithm recursively and assuming $m-1$ traitors
  - Each lieutenant collects values v_j received from other lieutenants and calculates $majority(v_j)$ for all j.
- A traitorous lieutenant will broadcast conflicting values, but they will disappear after $m+1$ rounds because they will be drowned out by correct values from $n-m$ loyal lieutenants.
- A traitorous commander will send conflicting values, which will lead to v_{def} being selected by all.
- Message complexity: $O(n^{m+1})$ since each application of $U(n,m)$ causes $n-1$ messages
- Time Complexity: $m+1$ rounds (fundamental to all algorithms in the presence of m failures)
- Need to review the other algorithm presented in notes

5 Asynchronous Replication slides

- Single point of failure problem: In Two-Phase-Commit this is the coordinator (?)
- Challenge with replication is *consistency*, how to make sure all replicas have same data
- In 2PC, no processes are allowed to fail, all votes must be counted, and no progress under failure if there is no recovery
- We model a process as a state machine, data updates are transitions
- Each replica has same state update so is consistent
- If we nominate *primary*, then all operations (state transitions) get sent to primary
- If primary fails, last op it attempted did not complete. Do leader election for next unique primary.

6 Failure Detectors slides

7 Asynchronous Algorithms

- Algorithm 1: process wishing to enter critical region sets a local variable and checks the other process' local lock variable. If the other process has the same lock variable set, it doesn't enter the region. Else, it enters, executs and unsets the local lock variable
- Algorithm 2: For some fairness to hold, we need a shared variable, "turn". The process "i" sets a local flag variable, then stores "i" in "turn". While either the other process has its flag set or turn has value "i", the process blocks. Otherwise, process "i" enters critical region, executes, exits, and unsets local flag variable.
- the second algorithm allows for two modes: when another process is competing and for when it is not
- we can extend to multiple processes by allowing for multiple iterations to happen before reaching critical region

8 a

-