Distributed Systems

Failure detection & Leader Election

Rik Sarkar

University of Edinburgh
Fall 2016
Failures

• How do we know that something has failed?
• Let’s see what we mean by failed:

• Models of failure:
  1. Assume no failures
  2. Crash failures: Process may fail/crash
  3. Message failures: Messages may get dropped
  4. Link failures: a communication link stops working
  5. Some combinations of 2,3,4
  6. More complex models can have recovery from failures
  7. Arbitrary failures: computation/communication may be erroneous
Failure detectors

- Detection of a crashed process
  - (not one working erroneously)

- A major challenge in distributed systems

- A failure detector is a process that responds to questions asking whether a given process has failed
  - A failure detector is not necessarily accurate
Failure detectors

• Reliable failure detectors
  – Replies with “working” or “failed”

• Difficulty:
  – Detecting something is working is easier: if they respond to a message, they are working
  – Detecting failure is harder: if they don’t respond to the message, the message may have been lost/delayed, may be the process is busy, etc..

• Unreliable failure detector
  – Replies with “suspected (failed)” or “unsuspected”
  – That is, does not try to give a confirmed answer

• We would ideally like reliable detectors, but unreliable ones (that say give “maybe” answers) could be more realistic
Simple example

• Suppose we know all messages are delivered within $D$ seconds

• Then we can require each process to send a message every $T$ seconds to the failure detectors

• If a failure detector does not get a message from process $p$ in $T+D$ seconds, it marks $p$ as “suspected” or “failed”
Simple example

• Suppose we assume all messages are delivered within D seconds

• Then we can require each process to send a message every T seconds to the failure detectors

• If a failure detector does not get a message from process p in T+D seconds, it marks p as “suspected” or “failed” (depending on type of detector)
Synchronous vs asynchronous

• In a synchronous system there is a bound on message delivery time (and clock drift)

• So this simple method gives a reliable failure detector

• In fact, it is possible to implement this simply as a function:
  – Send a message to process p, wait for $2D + \varepsilon$ time
  – A dedicated detector process is not necessary

• In Asynchronous systems, things are much harder
Simple failure detector

• If we choose T or D too large, then it will take a long time for failure to be detected
• If we select T too small, it increases communication costs and puts too much burden on processes
• If we select D too small, then working processes may get labeled as failed/suspected
Assumptions and real world

• In reality, both synchronous and asynchronous are a too rigid
• Real systems, are fast, but sometimes messages can take a longer than usual
  – But not indefinitely long
• Messages usually get delivered, but sometimes not..
Some more realistic failure detectors

• Have 2 values of D: D1, D2
  – Mark processes as working, suspected, failed

• Use probabilities
  – Instead of synchronous/asynchronous, model delivery time as probability distribution
  – We can learn the probability distribution of message delivery time, and accordingly estimate the probability of failure
Using bayes rule

- \( a = \) probability that a process fails within time \( T \)
- \( b = \) probability a message is not received in \( T + D \)

- So, when we do not receive a message from a process we want to estimate \( P(a | b) \)
  - Probability of \( a \), given that \( b \) has occurred

\[
P(a | b) = \frac{P(b | a)P(a)}{P(b)}
\]

If process has failed, i.e. \( a \) is true, then of course message will not be received! i.e. \( P(b | a) = 1 \). Therefore:

\[
P(a | b) = \frac{P(a)}{P(b)}
\]
Leader of a computation

• Many distributed computations need a coordinating or server process
  – E.g. Central server for mutual exclusion
  – Initiating a distributed computation
  – Computing the sum/max using aggregation tree
• We may need to elect a leader at the start of computation
• We may need to elect a new leader if the current leader of the computation fails
The Distinguished leader

• The leader must have a special property that other nodes do not have

• If all nodes are exactly identical in every way then there is no algorithm to identify one as leader

• Our policy:
  – The node with highest identifier is leader
Node with highest identifier

- If all nodes know the highest identifier (say n), we do not need an election
  - Everyone assumes n is leader
  - n starts operating as leader
- But what if n fails? We cannot assume n-1 is leader, since n-1 may have failed too! Or may be there never was process n-1

- Our policy:
  - The node with highest identifier and still surviving is the leader

- We need an algorithm that finds the working node with highest identifier
Strategy 1: Use aggregation tree

• Suppose node $r$ detects that leader has failed, and initiates leader election

• Node $r$ creates a BFS tree

• Asks for max node id to be computed via aggregation
  – Each node receives id values from children
  – Each node computes max of own id and received values, and forwards to parent

• Needs a tree construction
• If $n$ nodes start election, will need $n$ trees
  – $O(n^2)$ communication
  – $O(n)$ storage per node
Strategy 1: Use aggregation tree

- Suppose node $r$ detects that leader has failed, and initiates leader election

- Node $r$ creates a BFS tree

- Asks for max node id to be computed via aggregation
  - Each node receives id values from children
  - Each node computes max of own id and received values, and forwards to parent

- Needs a tree construction
- If $n$ nodes start election, will need $n$ trees
  - $O(n^2)$ communication
  - $O(n)$ storage per node
Strategy 2: Use a ring

• Suppose the network is a ring
  – We assume that each node has 2 pointers to nodes it knows about:
    • Next
    • Previous
    • (like a circular doubly linked list)
  – The actual network may not be a ring
  – This can be an overlay
Strategy 2: Use a ring

• Basic idea:
  – Suppose 6 starts election
  – Send “6” to 6.next, i.e. 2
  – 2 takes max(2, 6), send to 2.next
  – 8 takes max(8, 6), sends to 8.next
  – etc
Strategy 2: Use a ring

• The value “8” goes around the ring and comes back to 8

• Then 8 knows that “8” is the highest id
  – Since if there was a higher id, that would have stopped 8

• 8 declares itself the leader: sends a message around the ring
Strategy 2: Use a ring

• The problem: What if multiple nodes start leader election at the same time?

• We need to adapt algorithm slightly so that it can work whenever a leader is needed, and works for multiple leader
Strategy 2: Use a ring (Algorithm by chang and roberts)

- Every node has a default state: *non-participant*
- Starting node sets state to *participant* and sends *election* message with id to next
Strategy 2: Use a ring
(Algorithm by chang and roberts)

• If node p receives election message m

• If p is non-participant:
  – send max(m.id, p.id) to p.next
  – Set state to participant

• If p is participant:
  – If m.id > p.id:
    • Send m.id to p.next
  – If m.id < p.id:
    • do nothing
Strategy 2: Use a ring
(Algorithm by chang and roberts)

• If node p receives *election* message m with m.id = p.id

• P declares itself leader
  – Sets p.leader = p.id
  – Sends *leader* message with p.id to p.next
  – Any other node q receiving the leader message
    • Sets q.leader = p.id
    • Forwards leader message to q.next
Strategy 2: Use a ring
(Algorithm by chang and roberts)

• Works in an asynchronous system
• Assuming nothing fails while the algorithm is executing

• Message complexity $O(n^2)$
  – When does this occur?
  – (hint: all nodes start election, and many messages traverse a long distance)

• What is the time complexity?
• What is the storage complexity?
Strategy 3: Use a ring – smartly (Hirschberg Sinclair)

• Assume all nodes want to know the leader
• k-neighborhood of node p
  – The set of all nodes within distance k of p

• How does p send a message to distance k?
  – Message has a “time to live variable”
  – Each node decrements m.ttl on receiving
  – If m.ttl=0, don’t forward any more
Strategy 3: Use a ring – smartly (Hirschberg Sinclair)

• Basic idea:
  – Check growing regions around yourself for someone with larger id
Strategy 3: Use a ring – smartly (Hirschberg Sinclair)

• Algorithm operates in phases
• In phase 0, node p sends election message m to both p.next and p.previous with:
  – m.id = p.id and ttl = 1

• Suppose q receives this message
  – Sets m.ttl=0
  – If q.id > m.id:
    • Do nothing
  – If q.id < m.id:
    • Return message to p
Strategy 3: Use a ring – smartly  
(Hirschberg Sinclair)

• Algorithm operates in phases
• In phase 0, node p sends election message m to both p.next and p.previous with:
  – m.id = p.id and ttl = 1

• Suppose q receives this message
  – Sets m.ttl=0
  – If q.id > m.id:
    • Do nothing
  – If q.id < m.id:
    • Return message to p

• If p gets back both message, it decides itself leader of its 1-neighborhood, and proceeds to next phase
Strategy 3: Use a ring – smartly
(Hirschberg Sinclair)

- If \( p \) is in phase \( i \), node \( p \) sends election message \( m \) to \( p\.next \) and \( p\.previous \) with:
  - \( m\.id = p\.id \), and \( m\.ttl = 2^i \)

- A node \( q \) on receiving the message (from next/previous)
  - If \( m\.ttl=0 \): forward suitably to previous/next
  - Sets \( m\.ttl=m\.ttl-1 \)
  - If \( q\.id > m\.id \):
    - Do nothing
  - Else:
    - If \( m\.ttl = 0 \): return to sending process
    - Else forward to suitably to previous/next

- If \( p \) gets both message back, it is the leader of its \( 2^i \) neighborhood, and proceeds to phase \( i+1 \)
Strategy 3: Use a ring – smartly (Hirschberg Sinclair)

- When $2^i \geq n/2$
  - Only 1 process survives: Leader

- Number of phases: $O(\log n)$

- What is the message complexity?
Strategy 3: Use a ring – smartly 
(Hirschberg Sinclair)

In phase i

• At most one node initiates message in any sequence of $2^{i-1}$ nodes

• So, $n/2^{i-1}$ candidates
  – Each sends 2 messages, going at most $2^i$ distance, and returning: $2*2*2^i$ messages

• $O(n)$ messages in phase i

There are $O(\log n)$ phases

• Total of $O(n \log n)$ messages
Strategy 3: Use a ring – smartly (Hirschberg Sinclair)

- Assume synchronous operation
- Assume nodes do not fail during algorithm run

- What is time complexity?
- What is storage complexity?
Strategy 4: Bully Algorithm

Ref: CDK

• Assume:
  – Each node knows the id of all nodes in the system (some may have failed)
  – Synchronous operation

• Node p decides to initiate election
• p sends election message to all nodes with id > p.id
• If p does not hear “I am alive message” from any node, p broadcasts a message declaring itself as leader
• Any working node q that receives election message from p, replies with own id and “I am alive” message
  – And starts an election (unless it is already in the process of an election)
• Any node q that hears a lower id node being declared leader, starts a new election
Strategy 4: Bully Algorithm

• Assume:
  – Each node knows the id of all nodes in the system (some may have failed)
  – Synchronous operation

• Works even when processes fail
• Works when (some) message deliveries fail.

• What are the storage and message complexities?