Distributed Systems

Failure detectors

Rik Sarkar
James Cheney

University of Edinburgh
Spring 2014

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

Ref: CDK

- 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 hev 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 + ε 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 extimate 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

$$P(a \mid b) = \frac{P(b \mid a)P(a)}{P(b)}$$

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

$$P(a \mid b) = \frac{P(a)}{P(b)}$$

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

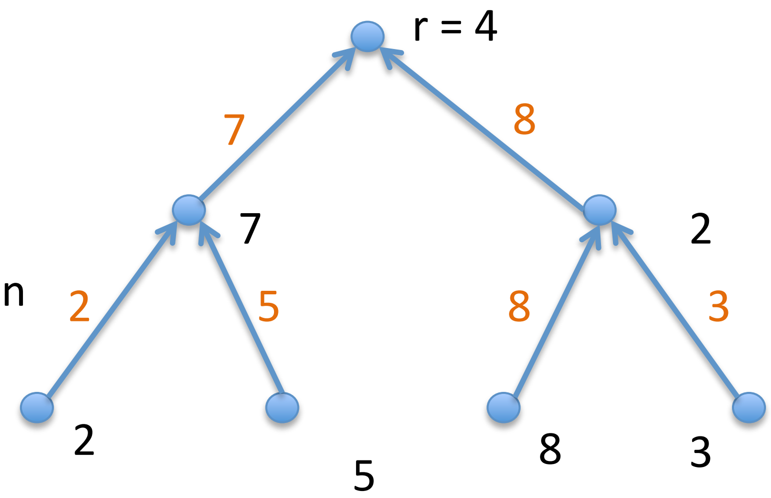
- If all nodes know the highest identifier (say n), we do not need an election
  – Everyone assumes n is leader
  – n starts operating as leader
- But what if n fails? We cannot assume n-1 is leader, since n-1 may have failed too! Or may be there never was process n-1

- 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