Distributed Systems: Consensus Algorithms

Alistair Rendell

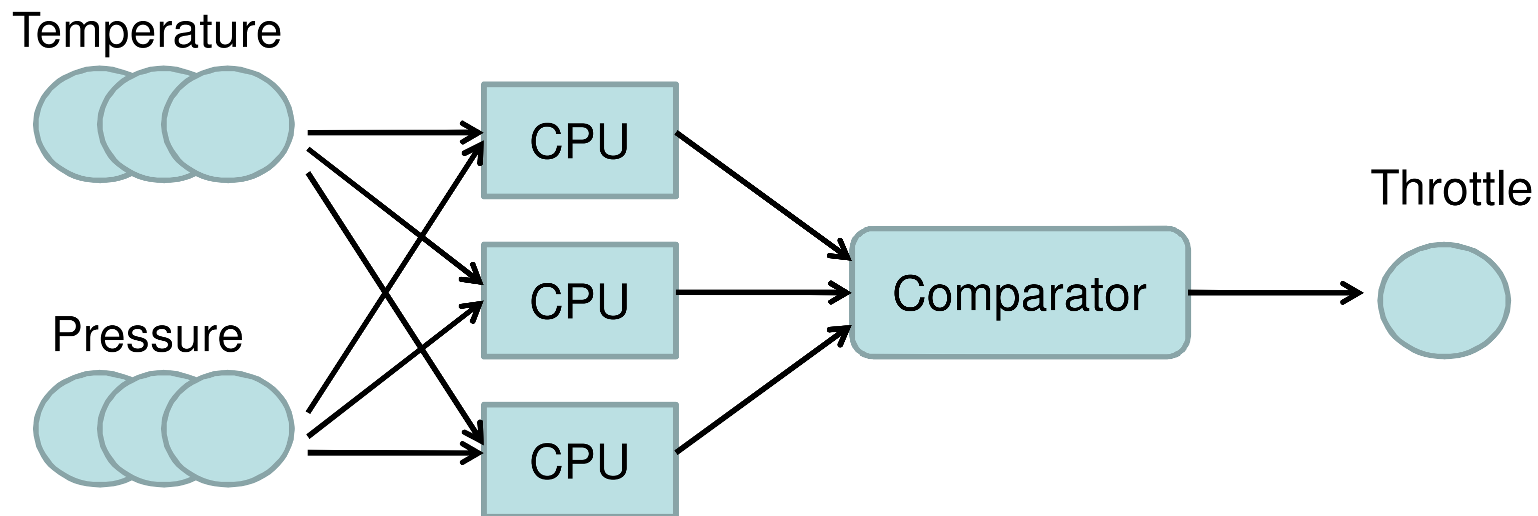
©2009 School Computer Science, Australian National University

Motivation

- A primary objective of distributed systems is to improve reliability
- Two important properties
  - Fail-safe: if one or more failures do not cause damage to the system
  - Fault-tolerant: system continues to fulfill its requirements even if there are on or more failures
- A distributed system need not be fail-safe or fault-tolerant

©2009 School Computer Science, Australian National University

Example Reliable System



- Input sensors replicated as are CPUs
- Results compared and algorithm, such as majority voting used to determine outcome
- Complexity
  - input sensors may not give identical results
  - input sensors may not fail gracefully, but produce spurious data
  - Same software on CPUs implies not tolerant to software bugs, different software implies slightly different output

©2009 School Computer Science, Australian National University

Distinguish Two Cases

- Crash Failures
  - a node/component in our system simply stops sending results/messages
  - Simple Paxos algorithm
- Byzantine Failures
  - a node/component in our system can send arbitrary messages, not just those required by algorithm, and nodes may collude to divert the protocol
  - Byzantine Paxos

©2009 School Computer Science, Australian National University

Simple Paxos

- Get a set of processors to reach consensus on a single proposed value
- Safety requirement
  - has to be a proposed value
  - only one value chosen
  - no processes learn of value until it has actually been chosen
- Don't care what value is chosen, provided it satisfies the above

©2009 School Computer Science, Australian National University

Model

- Processes communicate by sending messages
- Messages are asynchronous
  - arbitrary time to deliver
  - might be duplicated
  - might be lost
- Processes are fail-stop, but
  - can be restarted if they fail
  - can remember some information if restarted after failure

©2009 School Computer Science, Australian National University

Roles

- Processes are one of:
  - Proposers: proposing a value to be chosen
  - Acceptors: decide which value to choose
  - Learners: informed of what value was chosen after event (we will ignore here)
- Processes can play multiple roles

©2009 School Computer Science, Australian National University

An Easy Solution

- Have single acceptor
  - Proposers send to single acceptor
  - Acceptor decides
- But this is not distributed at all!
  - if the acceptor dies nothing will work!

©2009 School Computer Science, Australian National University

Multiple Acceptors

- Value chosen once “enough” acceptors have accepted it
- Define enough as simple majority
  - For n acceptors we need >n/2 to accept value
  - Each acceptor can only accept one value at a time

©2009 School Computer Science, Australian National University

Differentiating Proposals

- Proposal have two parts
  - A unique identifier
  - The proposed value
- Unique identifier is like a timestamp, but can be any unique ID from a totally ordered set
- There can be multiple distinct proposals for the same value

©2009 School Computer Science, Australian National University

Simple Paxos Outline

- Prepare
  - a proposer selects a proposal number (N) and sends a prepare message to a quorum of acceptors
- Promise
  - if proposal number (N) is larger than previous proposal,
    - ➢ then acceptor promises not to accept future proposals less than N
    - ➢ returns value and proposal number it last accepted, if any
  - else, reject proposal
- Accept
  - if proposer receives promise from a quorum of acceptors, it now chooses a value
    - ➢ if any acceptor has already chosen a value the proposer must chose the one with the highest proposal number, else it is free to chose any value
  - the proposer sends an accept message to each acceptor with the chosen value
- Accepted
  - each acceptor that receives an accept message for a proposal it has promised, then accepts the value if it has not made any conflicting promises in the meantime

©2009 School Computer Science, Australian National University

Example

- Proposers are p_1 and p_2
- Acceptors are a_1, a_2 and a_3
- p_1 sends prepare for proposal 1 to a_1 and a_2
  - a_1 and a_2 reply to p_1
- p_2 sends prepare for proposal 2 to a_2 and a_3
  - a_2 and a_3 reply to p_2
- p_1 sends accept request to a_1 and a_2 for proposal 1 with value "pepperoni" (p_1 got to select which value to propose)
  - a_1 accepts proposal 1
  - a_2 does not accept proposal 1 (a_2 promised p_2 it wouldn't accept proposals < 2)

©2009 School Computer Science, Australian National University

Example (cont)

- p_2 sends accept request to a_2 and a_3 for proposal 2 with value “mushrooms” (p_2 also got to select which value to propose)
  - a_2 accepts proposal 2
  - a_3 accepts proposal 2
  - $\{a_2, a_3\}$ is a majority of acceptors, so proposal 2 is chosen
  - the chosen value is “mushrooms”
- p_1 sends prepare for proposal 3 to a_1 and a_2
  - a_1 replies, it last accepted proposal 1 for “pepperoni”
  - a_2 replies it last accepted proposal 2 for “mushrooms”
- p_1 sends accept request to a_1 and a_2 for proposal 3 with value “mushrooms” (value must match that from proposal 2)
  - a_1 and a_2 accept proposal 3

©2009 School Computer Science, Australian National University

Byzantine Problems

- A group of Byzantine armies is surrounding an enemy city.
- The generals of the armies have reliable messengers who successfully deliver any message sent from one general to another
- Some generals are traitors endeavoring to bring about the defeat of the Byzantine armies
- Devise an algorithm to ensure all loyal generals come to a consensus on a plan
- The final decision should be almost the same as a majority vote of their initial choices, if the vote is tied they retreat
- (following diagrams taken from Ben-Ari Chapter 12)

©2009 School Computer Science, Australian National University

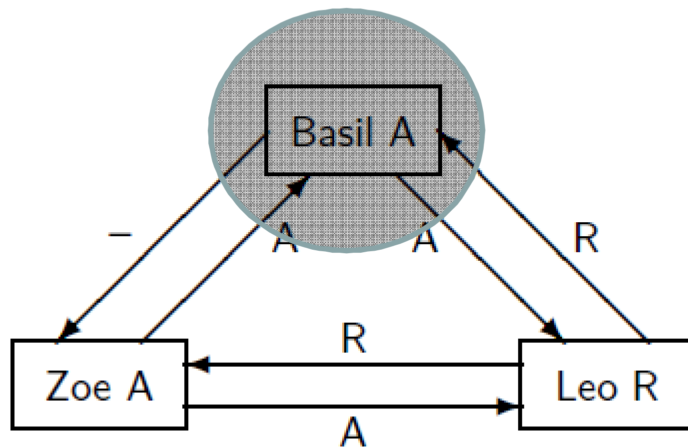
A One-Round Algorithm

Algorithm 12.1: Consensus - one-round algorithm	
	planType finalPlan planType array[generals] plan
p1:	plan[myID] ← chooseAttackOrRetreat
p2:	for all *other* generals G
p3:	send(G, myID, plan[myID])
p4:	for all *other* generals G
p5:	receive(G, plan[G])
p6:	finalPlan ← majority(plan)

©2009 School Computer Science, Australian National University

Three General Case

- Two loyal generals, Zoe and Leo, and a third traitor (Basil)
- Basil and Zoe chose to attach, while Leo chooses to retreat
- Basil crashes after sending message to Leo that he has chosen to attack



Leo	
general	plan
Basil	A
Leo	R
Zoe	A
majority	A

Zoe	
general	plans
Basil	–
Leo	R
Zoe	A
majority	R

Fact that traitor basil crashes causes two loyal generals to fail to come to an agreement

©2009 School Computer Science, Australian National University

The Byzantine Generals Algorithm

- The problem with the one-round approach is that we are not using the fact that certain generals are loyal
  - Leo needs to somehow attribute more weight to the plan received by loyal general Zoe than traitor Basil
- The Byzantine Generals algorithm achieves this by using multiple rounds
  - in subsequent rounds each general relays previous information received
  - loyal generals, by definition correctly relay all information received
  - traitors relay whatever they want!
- In general case the total number of generals must be at least 3t+1, where t is the number of traitors and there are t rounds

©2009 School Computer Science, Australian National University

2-Round Byzantine Algorithm

```
Algorithm 12.2: Consensus - Byzantine Generals algorithm

          planType finalPlan
          planType array[generals] plan, majorityPlan
          planType array[generals, generals] reportedPlan

p1:  plan[myID] ← chooseAttackOrRetreat
p2:  for all other generals G                          // First round
p3:      send(G, myID, plan[myID])
p4:  for all other generals G
p5:      receive(G, plan[G])
p6:  for all other generals G                          // Second round
p7:      for all other  generals G' except G
p8:          send(G', myID, G, plan[G])
p9:  for all other generals G
p10:     for all other generals G' except G
p11:         receive(G, G', reportedPlan[G, G'])
p12: for all other generals G                          // First vote
p13:     majorityPlan[G] ← majority(plan[G] ∪ reportedPlan[*, G])
p14: majorityPlan[myID] ← plan[myID]                   // Second vote
p15: finalPlan ← majority(majorityPlan)
```

Byzantine Failures with 4 Generals

- Zoe is traitor
- Partial data structure at Basil, showing only loyal generals
- Basil has two correct votes for John and Leo regardless of what Zoe does

Basil					
general	plan	reported by			majority
		John	Leo	Zoe	
Basil	A				A
John	**A**		**A**	?	A
Leo	R	R		?	R
Zoe	?	?	?		?
majority					?

©2009 School Computer Science, Australian National University

Adding Zoe's Plan

- Zoe sends arbitrary messages, but these are accurately relayed by loyal generals
- Decision is 2-1 in favour of R for Zoe
- Traitor cannot cause loyal generals to fail to come to a consensus
  - eg if Zoe sent A instead of R to Basil, final decision would be to A

Basil					
general	plans	reported by			majority
		John	Leo	Zoe	
Basil	A				A
John	A		A	?	A
Leo	R	R		?	R
Zoe	**R**	**A**	**R**		R
					R

©2009 School Computer Science, Australian National University