CSCI 5673
Distributed Systems

Lecture Set Four

Asynchrony, Failure Models and Distributed Consensus

Lecture Notes by
Shivakant Mishra
Computer Science, CU Boulder
Last update: February 14, 2017

Synchrony and Failure Models

- Design of a distributed system is dependent on two important *system assumptions*
  - Synchrony Model
  - Failure model

Synchrony Model

- Synchrony model refers to the assumptions about latencies in distributed systems
- Synchronous systems
  - There is a *known* bound on scheduling and communication delays
- Asynchronous systems
  - There is no *known* bound on communication or scheduling delays
- Timed asynchronous systems
  - There is no *known* bound on communication or scheduling delays, but a statistical model exists
    - e.g. 99% latencies are less than 2 msec

Failure Models

- A failure model describes the behavior of a component when it fails
  - Fail stop failure
    - The component stops responding when it fails
    - No incorrect state transition
    - Detectable
  - Crash failure
    - The component stops responding when it fails
    - No incorrect state transition
    - Most common type of failure

- Performance failure
  - (Correct) response is produced either too early or late.
- Omission failure
  - The component omits to produce a response to one or more inputs.
- Byzantine failure
  - Arbitrary failure
  - Incorrect state transition, coordinated failures, malicious intent, …

Subset relationship among failure models

There are several other failure models proposed in literature

Failure Models

Subset relationship among failure models

Failure Models

There are several other failure models proposed in literature

Distributed Systems

Synchrony Model

- Generally, a distributed system designed for a synchronous environment is simpler
- Most non-critical distributed systems are designed for asynchronous environment

Failure model

- Generally, a system designed to tolerate fail stop failures is simpler than a system designed to tolerate crash failure, …
- Most non-critical systems are designed to tolerate crash failures

Consensus Problem

- Fundamental problem in distributed systems
  - Mutual exclusion, replication consistency, message ordering, distributed locking, leader election are all instances of distributed agreement
- Problem
  - M processes: P: p_1, …, p_m
  - Each process p_i stores a value v_i
  - A subset F of P are faulty
  - Goal: Each process p_i calculates a consensus value a_i.

- Solution requirements
  - Agreement: For every pair of non-faulty process p_i and p_j, $a_i = a_j$. This is the consensus value.
  - Validity: The consensus value is a function of the initial value v_i of all non-faulty processes
  - Termination: All non-faulty processes eventually calculate a consensus value

Consensus algorithm depends on system model and failure model

Distributed Agreement: Synchronous Systems

- Termination requirement: Agreement should be achieved with in *r* rounds of message exchanges for some fixed *r*.

Byzantine Generals Problem

- Reference: Lamport, Shostak and Pease. The Byzantine Generals Problem. ACM TOPLAS, 4(3), 1982.
- Turkish sultan led an invasion into Byzantium empire
- Byzantium emperor has several armies, each led by a general
- Byzantine generals can survive (win or retreat and escape safely) if their action (attack/retreat) is coordinated
- Generals do not talk with one another directly
  - They send messengers
- Goal: Devise an algorithm that enables Byzantine generals to win or escape safely.

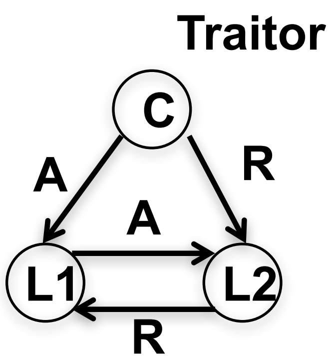
Byzantine Generals Problem

- Turkish sultan led an invasion into Byzantium empire
- Byzantium emperor has several armies, each led by a general
- Byzantine generals can win (or retreat and escape safely) if their action (attack/retreat) is coordinated
- Generals do not talk with one another directly
  - They send messengers
- Goal: Devise an algorithm that enables Byzantine generals to win or escape safely.
- Problem:
  - *Some generals may be on Turkish payroll*
  - These treacherous generals will try to deceive (e.g. lie to) the loyal generals to prevent a coordinated action

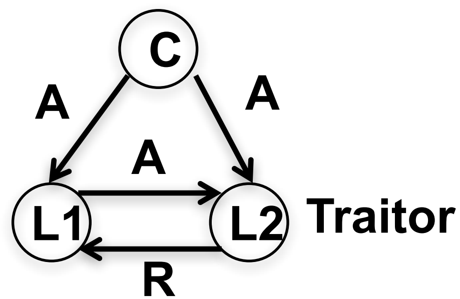
Byzantine Generals Problem

- Assumptions:
  - Synchronous, reliable channels
  - Messages cannot be altered
  - Receivers can authenticate the senders

- Result 1: Suppose that there are three generals and one of them is a traitor
  - Byzantine generals problem cannot be solved



L1: 1 attack; 1 retreat



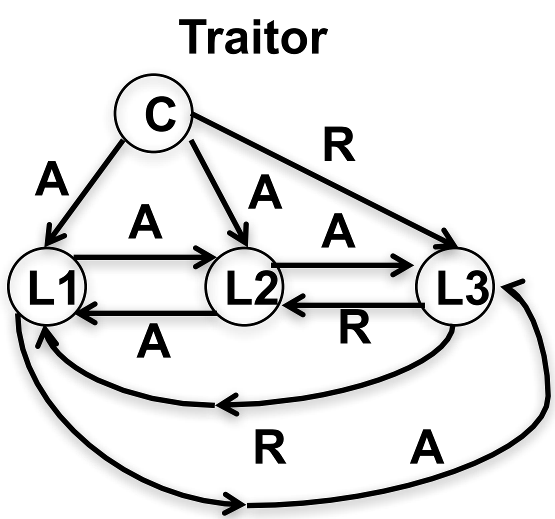
L1: 1 attack; 1 retreat

L1 cannot distinguish between the two scenarios

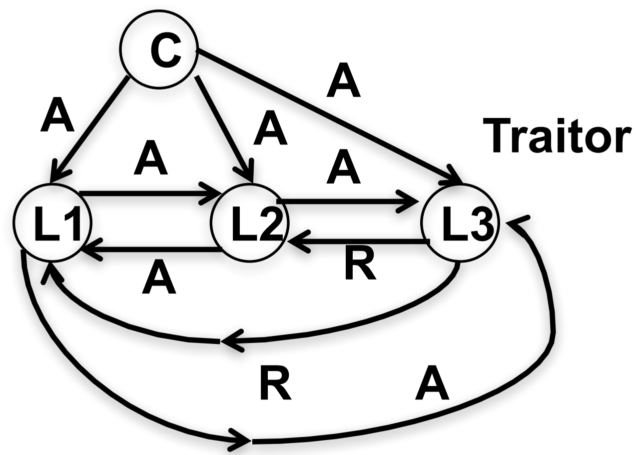
- Result 2: Suppose that there are four generals and at most one of them is a traitor
  - Byzantine generals problem can be solved

Algorithm

1. C sends order to everyone
2. Ls exchange the orders received with one another
3. Ls decide on an action based on majority



L1, L2, L3: 2 Attacks, 1 Retreat



L1, L2: 2 Attacks, 1 Retreat

- General result: Suppose that there are *n* generals and at most *f* of them are traitors
  - Byzantine generals problem can be solved, if $n > 3f$

Consensus

- Scenario 1:
  - Synchronous system
  - No process or message omission failures
- One-round algorithm:
  - Each process sends its value to all the processes.
  - If all values a process has (including its own) are 1 then decide 1. Otherwise decide 0.

<Figure>

Consensus

- Scenario 2:
  - Synchronous system
  - No process failure
  - Message omission failure: Any number of messages may be lost.
- Solution: ???
- Theorem: There is no algorithm that solves the agreement problem for even two processes

Hint: What if all messages are lost?

Network partition

Consensus

- Scenario 3:
  - Synchronous system
  - No messages are lost (No message omission failure).
  - Up to f processes may fail (fail stop failure)
- Algorithm: ???

Consensus

- Scenario 3:
  - Synchronous system
  - Up to *f* processes may fail (fail stop failure)
  - No messages are lost (No message omission failure).
- Algorithm
  - Each process maintains a vector containing a value for each process {0,1, *u*}. *u* = undefined.
  - One round:
    - Send your vector to all processes.
    - Update local vector according to the received vectors (in case local vector has a "u", and any of received vectors contain "0" or "1").
  - After *f+1* rounds decide according to the local vector. If you have a majority of 1 in the vector then decide 1, otherwise decide 0.

- Scenario 3:

<u>Homework:</u> Construct an example to show that the processes may not reach a consensus with only *f* rounds, for *n* processes $n>f$, for any *n* and *f*.

Consensus: Asynchronous Systems

- Scenario 1:
  - Asynchronous system
  - No process or message omission failures
- Algorithm: ???
- Algorithm: *similar to the synchronous case*
  - Each process sends its value to all the processes
  - After receiving a value from all processes, if all values a process has (including its own) are 1 then decide 1. Otherwise decide 0

Distributed Agreement: Asynchronous Systems

- Asynchronous systems
  - There is no *known* bound on communication delays
- Scenario 1:
  - Asynchronous system
  - No process or message omission failures
- Algorithm: *similar to the synchronous case*
  - Each process sends its value to all the processes
  - After receiving a value from all processes, if all values a process has (including its own) are 1 then decide 1. Otherwise decide 0

Question: How is this algorithm different from the synchronous case?

Distributed Agreement: Asynchronous Systems

- Scenario 2:
  - Asynchronous system
  - No process failure
  - Message omission failure: Any number of messages may be lost
- Algorithm: ???
- Algorithm:
  - There is no algorithm that solves the agreement problem for even two processes
    - Note that the scenario 2 of synchronous case is a special case of this scenario.

Distributed Agreement: Asynchronous Systems

- Scenario 3:
  - Asynchronous system
  - Up to f processes may fail (fail stop failure)
  - No messages are lost (No message omission failure).
- Algorithm: ???

FLP Result

- In an asynchronous distributed system, no algorithm can guarantee to reach a consensus between participating processes if one or more of them can fail by stopping
- Reference: M.J. Fischer, N. Lynch and M.S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. JACM, 32(2), 1985.
- One of the most important result in distributed systems
- Definitively placed an upper bound on what it is possible to achieve with distributed processes in an asynchronous setting

FLP result: Consequences

- It is impossible to distinguish between a process failure and a communication failure in an asynchronous distributed system
- It is impossible to reliably detect the failure of a process in an asynchronous distributed system

Question: Scenario 3 is a very common scenario in practice. So, what can we do in light of the FLP result?

FLP Result: What does ``Impossibility'' mean?

- In formal proofs, an algorithm is totally correct, if
  - It computes the right thing
  - And it ``always'' terminates
- FLP proves that any algorithm that solves consensus in an asynchronous distributed system in the presence of process failures has runs that never terminate
  - These runs are extremely unlikely
- Consensus is <u>impossible</u> thus means consensus is <u>not always possible</u>

Adapted from Ken Birman's Consensus, Impossibility Results and Paxos

Consensus: In Light of the FLP Result

- Probabilistic algorithms for consensus
  - Allow algorithms to not always guarantee consensus
- Allow algorithms to not always terminate
- Failure detectors
- Group membership protocols

Safety and Liveness Properties

- Safety property: Nothing bad will ever happen
  - e.g. No two process will ever acquire a write lock for the same item at the same time
  - e.g. No two messages will ever be delivered in different order in different processes using a total order protocol
  - e.g. at no point, two or more new tokens will be generated
- Liveness property: Something good will eventually happen
  - e.g. a process requesting to acquire a lock will eventually acquire it
  - e.g. a message multicast using a total order atomic multicast protocol will eventually be delivered by all processes
  - e.g. a new token will eventually be generated after a token loss

- Distributed services built for an asynchronous environment where components may suffer crash failures can satisfy safety property but not liveness property
  - i.e. there will be some scenarios under which these services will not be able to provide an output; these scenarios are generally extremely unlikely