Cristina Nita-Rotaru

7610 : Distributed Systems

Consensus.

Required reading for this topic…

‣ Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson for "Impossibility of Distributed Consensus with One Faulty Process," 1985.

‣ Consensus in the Presence of Partial Synchrony, C. Dwork, Nancy Lynch, L. Stockmeyer, Journal of the Association for Computing Machinery, 1988

‣ L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem ACM Transactions on Programming 1982.

‣ M. Ben-Or. Another advantage of free choice (Extended Abstract): Completely asynchronous agreement protocol. 1983.

Synchronous/asynchronous communication

let Δ be the time it takes to send a message between two processes

▶ <u>Synchronous:</u>

   ▸ Δ is known and fixed

▶ <u>Asynchronous:</u>

   ▸ There is no bound for Δ

Most common failure models

- <u>Fail-stop (crash)</u>: accurately detectable halting failures, once failed the process does not restart

- <u>Omission</u>: just a subset of the messages are delivered

- <u>Network partitioning</u>: subset of processes are not reachable

- <u>Byzantine</u>: arbitrary failures, include both benign and malicious failures; processes can be authenticated or not

Consensus.

1: Consensus in synchronous systems with no failures

Consensus in distributed systems

▸ **Consensus: According to Merriam-Webster dictionary it means general agreement**

▸ **When do we need consensus in distributed systems?**

  ▸ Read-Modify-Write Memory

  ▸ Database commit

  ▸ Transactional file system

  ▸ Totally ordered broadcast

The consensus problem

- Input:
  - Each process has a value, either 1 or 0
- Properties:
  - **Agreement**: all nodes decide on the same value
  - **Validity**: if a process decides on a value, then there was a process that started with that value

Consensus in a synchronous system: No failures

- Failure model:
  - No faults
- Assumptions:
  - Communication is synchronous
- Algorithm - **requires 1 round**:
  - Each process sends its value to all the other processes
  - Each process decides:
    - 1: If all received values including its own are 1
    - 0: otherwise decides 0

  WHY IS THIS ALGORITM CORRECT?
  (i.e. provides agreement and validity)

Consensus in a synchronous system with crash failures: Model

▶ **Failure model:**

    ▶ Any process can crash, once crashed, a process does not recover

    ▶ At most f processes can crash

▶ **Communication model**

    ▶ Communication is synchronous

    ▶ Network is a fully connected graph

Consensus in a synchronous system with crash failures: Properties

▶ Input:

 ▶ 1 or 0 to each process

▶ Properties:

 ▶ **Agreement**: <u>all non-faulty</u> processes decide on the same value

 ▶ **Validity**: if a process decides on a value, then there was a process that started with that value

 ▶ **Termination**: non-faulty processes decide in a finite time

 NOTE: FAULTY PROCESSES MAY DECIDE DIFFERENTLY FROM CORRECT PROCESSES

Consensus in a synchronous system with crash failures: Algorithm

▸ Algorithm tolerates at most f failures, out of n nodes

▸ Each process maintains V the set of values proposed by other processes (initially it contains only its own value)

▸ In every round a process:

  ▸ Sends to all other processes the values from V that it has not sent before

▸ **After f+1 rounds** each process decides on the minimum value in V

$$f < n$$

Consensus in a synchronous system with crash failures: Algorithm

P_i::

var

    V: set of values initially $\{v_i\}$;

for $k := 1$ to $f + 1$ **do**

    send $\{v \in V \mid P_i$ has not already sent $v\}$ to all;

    receive S_j from all processes $P_j, j \neq i$;

    $V := V \cup S_j$;

endfor;

$y := min(V)$;

A closer look…

- ▶ Remember that communication is synchronous (the time it takes to send a message between two process is known and fixed)

- ▶ A process in a round may:

  - ▶ Send messages to any set of processes
  - ▶ Receive messages from any set of processes
  - ▶ Do local processing
  - ▶ Make a decision
  - ▶ Crash

- ▶ If a process p crashes in a round, then any subset of the messages sent by p in this round can be lost

- ▶ If a message was sent in a round it is either received in that round or it will never be received at all

- Key: all processes must be sure that all processes that did not crashed have the same information (so they can make the same decision)

- Processes can not decide in less than f+1 because they can not distinguish between executions in which all alive have the same list, or all alive have different lists.

Example

- Consider that you have 5 processes and 2 processes can crash
- After 1 round p1 has all 5 values, can he decide?
  - No, because he is not sure that every other process has all the information he has, for example, p5 crashed and p1 is the only one having info from p5
- After round 2, can p1 decide?
  - He has all the info.
  - But what about p2? P2 can not be sure that all the other processes received the info from p1, maybe p1 send his list to p2 and then crashed
  - Can he decide now?
- After round 3? Can p2 decide? Yes, assuming that p2 does not have the same list as another process, leads to more than 2 processes have crashed which contradicts our assumption

Sketch proof for the agreement property

▸ Assume by contradiction that two processes decide on different values, this means they had different final set of values, let's say p has a value v that q does not have

▸ How come that p got v and q did not? The only possible case is that a third process s, sent v to p, and crashed before sending v to q. SO in ROUND f +1, process s crashed (1 process)

▸ Because q does not have v it means that any process that may have sent v crashed also in round f

▸ Proceeding in this way, we infer at least one crash in each of the preceding rounds.

▸ STOP! We can have at most f crashes and we obtained that there are f+1 crashes (one in each round) → contradiction.

Variant: Uniform Consensus

▸ Input:

  ▸ 1 or 0 to each process

▸ Properties:

  ▸ Uniform Agreement: all processes (correct or faulty) decide on the same value

  ▸ Validity: if a process decides on a value, then there was a process that started with that value

  ▸ Termination: a non-faulty process decides in a finite time

2: Consensus in Asynchronous Systems

Includes slides from Ali Ghodsi, UC Berkeley

Synchronous/asynchronous communication

let Δ be the time it takes to send a message between two processes

- **<u>Synchronous</u>:**
  - Δ is known and fixed

- **<u>Asynchronous</u>:**
  - There is no bound for Δ

Consensus in Asynchronous Systems

> There is no asynchronous algorithm that achieves agreement on a one-bit value in the presence of **crash faults**. The result is true even if no crash actually occurs!

- ▸ Also known as the FLP result
- ▸ Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson for "Impossibility of Distributed Consensus with One Faulty Process," Journal of the ACM, April 1985, 32(2):374

What happens ...

- In an asynchronous system, a process p_i cannot tell whether a non-responsive process p_j has crashed or it is just slow

- What can p_i do?

  - If it waits, it will block since it might never receive the message from the non-responsive process

  - If it decides, it may find out later that the non-responsive process p_i made a different decision

Execution, configuration, events

- Set of processes p_i,

- State S_i: each process has a state

- Configuration C_t: set of state of each process at some moment in time

- Events: send and deliver messages, events can change the state at a process

- Execution: sequence of configuration and events
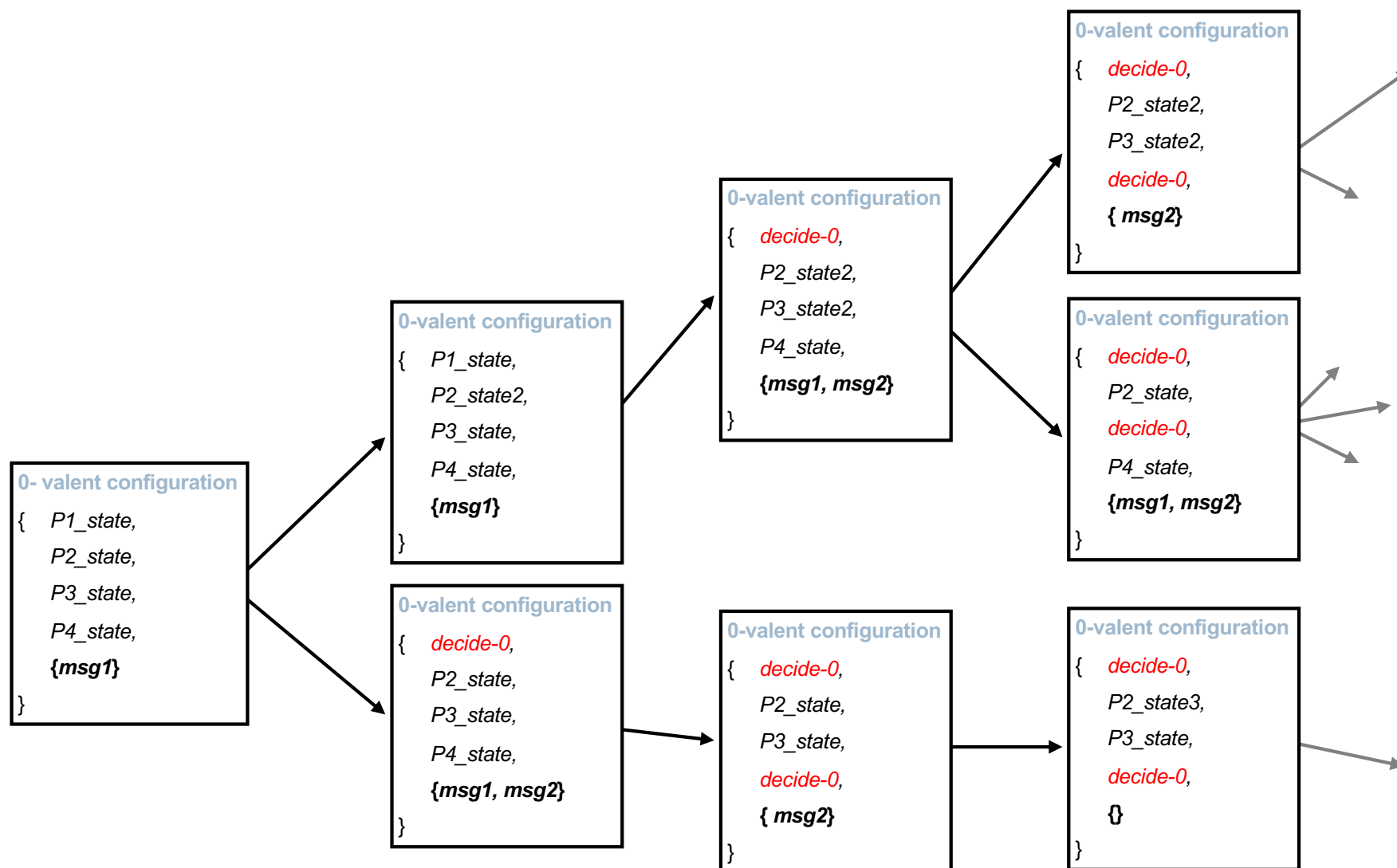
- A process p_i can send a message at most once

The problem

- Input:
  - 1 or 0 to each process

- Properties:
  - Agreement: all non-faulty processes decide on the same value
  - Validity: if a process decides on a value, then there was a process that started with that value
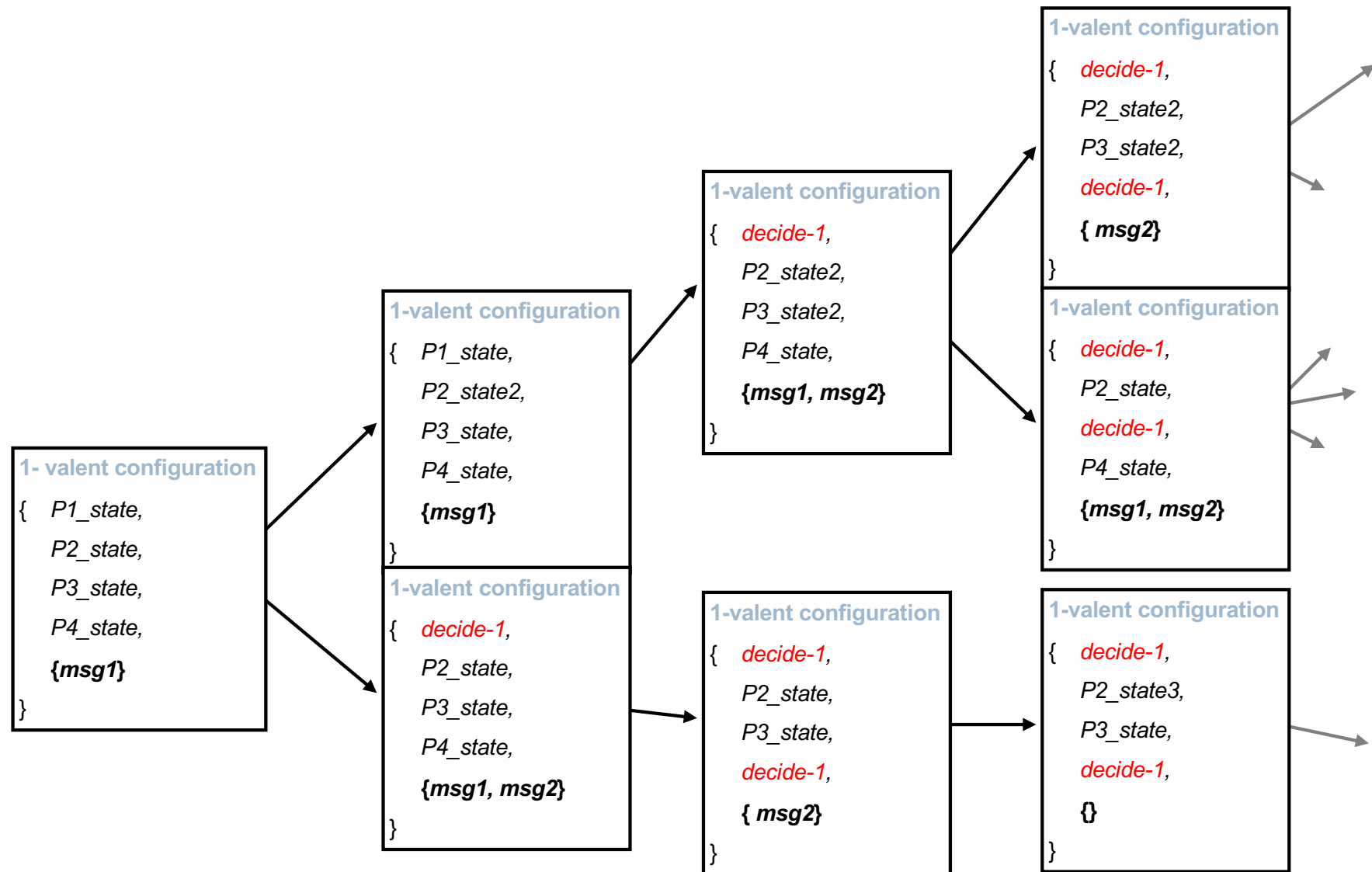 - Termination: A non-faulty process decides in a finite time

Proof Sketch

▸ The main idea of the proof is to construct an execution that does not decide (i.e. <u>does not terminate</u>), showing that the protocol remains forever indecisive

▸ Classify configurations (set of state of each process) as

  ▸ **0-valent** will result in deciding 0 (some process has decided 0 in C or if all configurations accessible from C are 0-valent; no 1-decided configurations are reachable)

  ▸ **1-valent** will result in deciding 1 (some process has decided 1 in C or if all configurations accessible from C are 1-valent; no 0-decided configurations are reachable)

  ▸ **Bivalent** has a bivalent outcome, decision is not already predetermined, outcome can be either 0 or ; both 0-decide and 1-decide configurations are reachable
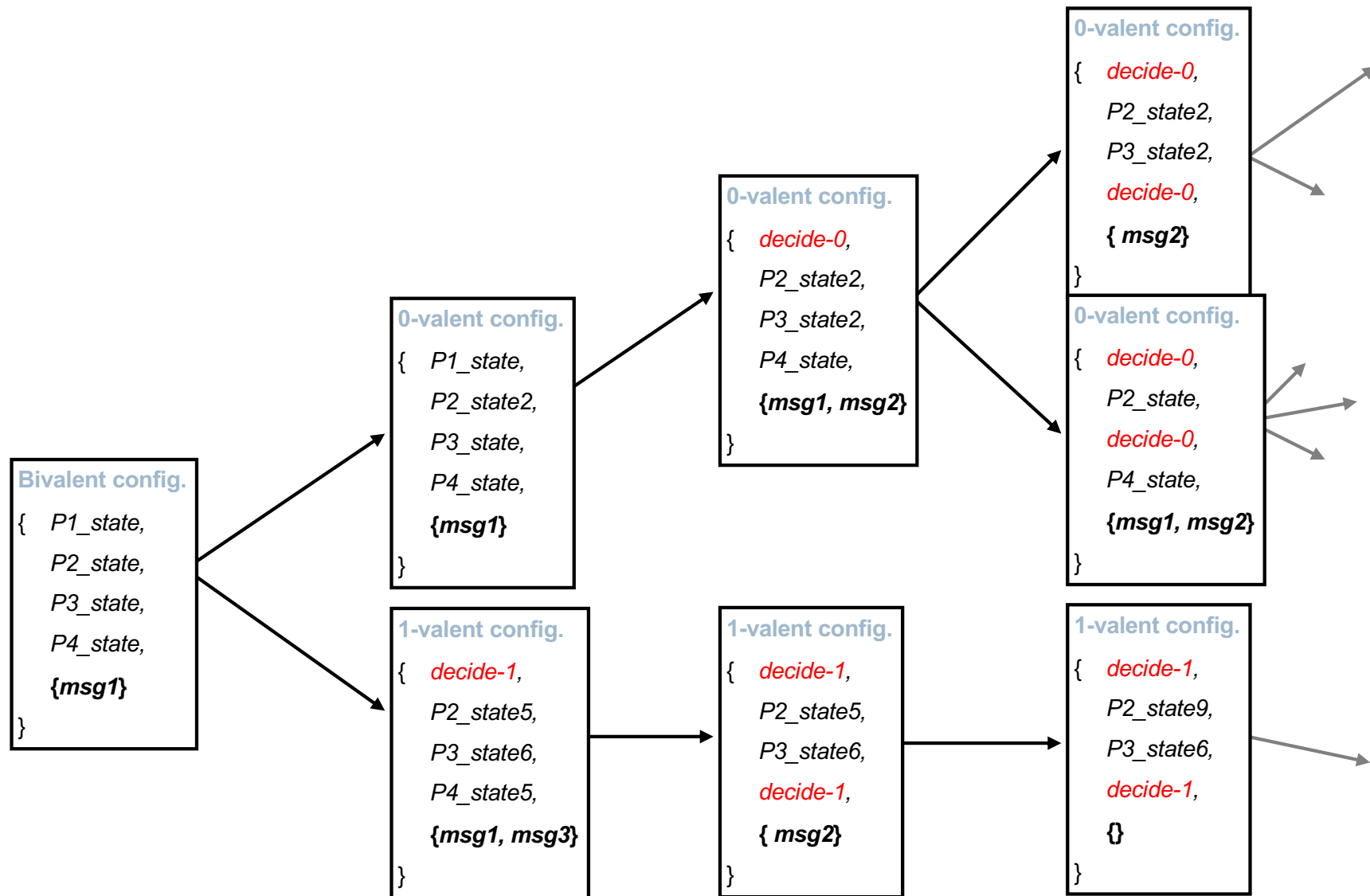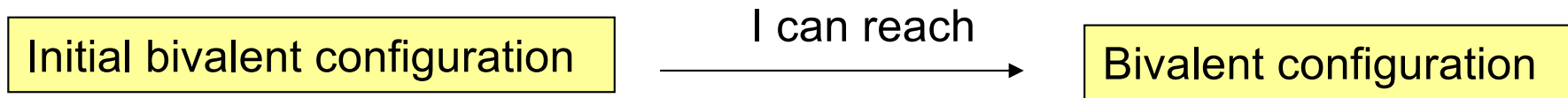
0-Valent Configuration Example

0-valent configuration

{ *decide-0*,
  P2_state2,
  P3_state2,
  decide-0,
  { msg2}
}

0-valent configuration

{ *decide-0*,
  P2_state2,
  P3_state2,
  P4_state,
  {msg1, msg2}
}

0-valent configuration

{ *decide-0*,
  P2_state,
  decide-0,
  P4_state,
  {msg1, msg2}
}

0-valent configuration

{ P1_state,
  P2_state2,
  P3_state,
  P4_state,
  {msg1}
}

0- valent configuration

{ *P1_state,*
  P2_state,
  P3_state,
  P4_state,
  {msg1}
}

0-valent configuration

{ *decide-0*,
  P2_state,
  P3_state,
  P4_state,
  {msg1, msg2}
}

0-valent configuration

{ *decide-0*,
  P2_state,
  P3_state,
  decide-0,
  { msg2}
}

0-valent configuration

{ *decide-0*,
  P2_state3,
  P3_state,
  decide-0,
  {}
}

Ali Ghodsi, alig(at)cs.berkeley.edu

1-Valent Configuration Example

1- valent configuration

{   P1_state,

P2_state,

P3_state,

P4_state,

{msg1}

}

1-valent configuration

{   P1_state,

P2_state2,

P3_state,

P4_state,

{msg1}

}

1-valent configuration

{   *decide-1*,

P2_state,

P3_state,

P4_state,

{msg1, msg2}

}

1-valent configuration

{   *decide-1*,

P2_state2,

P3_state2,

P4_state,

{msg1, msg2}

}

1-valent configuration

{   *decide-1*,

P2_state,

P3_state,

decide-1,

{ msg2}

}

1-valent configuration

{   *decide-1*,

P2_state2,

P3_state2,

decide-1,

{ msg2}

}

1-valent configuration

{   *decide-1*,

P2_state,

decide-1,

P4_state,

{msg1, msg2}

}

1-valent configuration

{   *decide-1*,

P2_state3,

P3_state,

decide-1,

{}

}

Ali Ghodsi, alig(at)cs.berkeley.edu

Bivalent Configuration Example

Bivalent config.
{   P1_state,
    P2_state,
    P3_state,
    P4_state,
    {msg1}
}

0-valent config.
{   P1_state,
    P2_state2,
    P3_state,
    P4_state,
    {msg1}
}

0-valent config.
{   *decide-0*,
    P2_state2,
    P3_state2,
    P4_state,
    {msg1, msg2}
}

0-valent config.
{   *decide-0*,
    P2_state2,
    P3_state2,
    decide-0,
    { msg2}
}

0-valent config.
{   *decide-0*,
    P2_state,
    decide-0,
    P4_state,
    {msg1, msg2}
}

1-valent config.
{ *decide-1*,
    P2_state5,
    P3_state6,
    P4_state5,
    {msg1, msg3}
}

1-valent config.
{ *decide-1*,
    P2_state5,
    P3_state6,
    decide-1,
    { msg2}
}

1-valent config.
{ *decide-1*,
    P2_state9,
    P3_state6,
    decide-1,
    {}
}

Ali Ghodsi, alig(at)cs.berkeley.edu

Proof Sketch (2)

▸ Start with an initially bivalent configuration

▸ Identify an execution that would lead to a univalent state, let's say 0-valent

  ▸ The switch from bivalent to univalent is due to an event e = (p,m) in which some process p receives some message m

▸ We will delay the e event for a while. Delivery of m would make the run univalent but m is delayed (fair-game in an asynchronous system)

▸ Since the protocol is indeed fault-tolerant there must be a run that leads to the other univalent state (1-valent in this case)

▸ Now let m be delivered, this will bring the system back in a bivalent state

▸ Decision can be delayed indefinitely

Proof: More Details

Initial bivalent configuration   I can reach   ⟶   Bivalent configuration

- Lemma 1: There exists an initial configuration that is bivalent.
- Lemma 2: Starting from a bivalent configuration C and an event e = (p, m) applicable to C, consider **C** the set of all configurations reachable from C without applying e and **D** the set of all configurations obtained by applying e to the configurations from **C**, then **D** contains a bivalent configuration.

- Theorem: There is always a run of events in an asynchronous distributed system such that the group of processes never reach consensus

Lemma 1:Proof Sketch

> Lemma 1:
> There exists an initial configuration that is bivalent

Proof by contradiction

▶ Let's assume that there is no bivalent initial configuration; then all configurations are 0-valent or 1-valent.

▶ List all initial configurations (list the 1 bits on the left).

▶ Consider a 0-valent initial configuration C_0 adjacent to a 1-valent configuration C_1: they differ only in the value corresponding to some process p.

Lemma 1 (cont.)

Lemma 1: There exists an initial configuration that is bivalent

- Let this process p crash.
- Note that now that p is crashed both C_0 and C_1 will lead to the same final configuration because with the exception of internal state of p they were identical (the only difference was determined by p).
  - Assume decision reached is 1: C_1 was 1-valent, then C_0 must have been bivalent (we assumed it was 0-valent and that is impossible)
  - Assume decision reached is 0: C_0 was already 0-valent, then C_1 must have been bivalent (we assumed it 1-valent, impossible)
- Thus, there exists an initial configuration that is bivalent.

Commutativity Lemma

Lemma

- Let S_1 and S_2 be schedules (sequences of events) applicable to some configuration C, and suppose that the set of processes taking steps in S_1 is disjoint from the set of processes taking steps in S_2 .
- Then, $S_1; S_2$ and $S_2; S_1$ are both sequences applicable to C, and they lead to the same configuration.

Lemma 2

▸ Consider an event e = (p, m) that can be applied to a bivalent initial configuration C

▸ **C** the set of all configurations reachable from C without applying e

▸ **D** the set of all configurations obtained by applying e to a configuration in C

▸ We want to show that **D** contains a bivalent configuration.

Lemma 2 (cont.)

▸ Lemma 2: ……We want to show that **D** contains a bivalent configuration

Proof by contradiction

▸ Assume that there is no bivalent configuration in **D**

▸ However there are adjacent configurations C_0 and C_1 in **C** such that $C_1 = C_0$ followed by event $e' = (p', m')$
      WHY (remember initial configuration is bivalent)

▸ Then denote

  ▸ D_0 be C_0 followed by $e=(p,m)$
  ▸ D_1 be C_1 followed by $e=(p,m)$

▸ Since there are no bivalent configurations in **D**, let's assume that D_0 is 0-valent and D_1 is 1-valent

Consensus

Lemma 2 (cont.)

▸ Case 1: p and p' are different.

▸ If we apply e' to D_0 we obtain D_1 since e and e' are disjoint..

▸ CONTRADICTION, any successor of a 0-valent configuration must be 0-valent.

$p \neq p'$
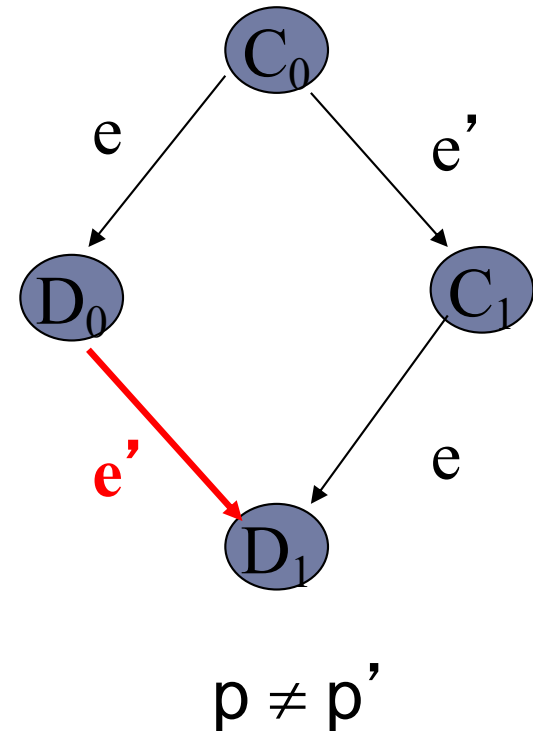
- Case 2: Process p is the same process as p'
- S is a run that reaches a decision, consider A that configuration.
- We obtain that A is bivalent, contradiction!
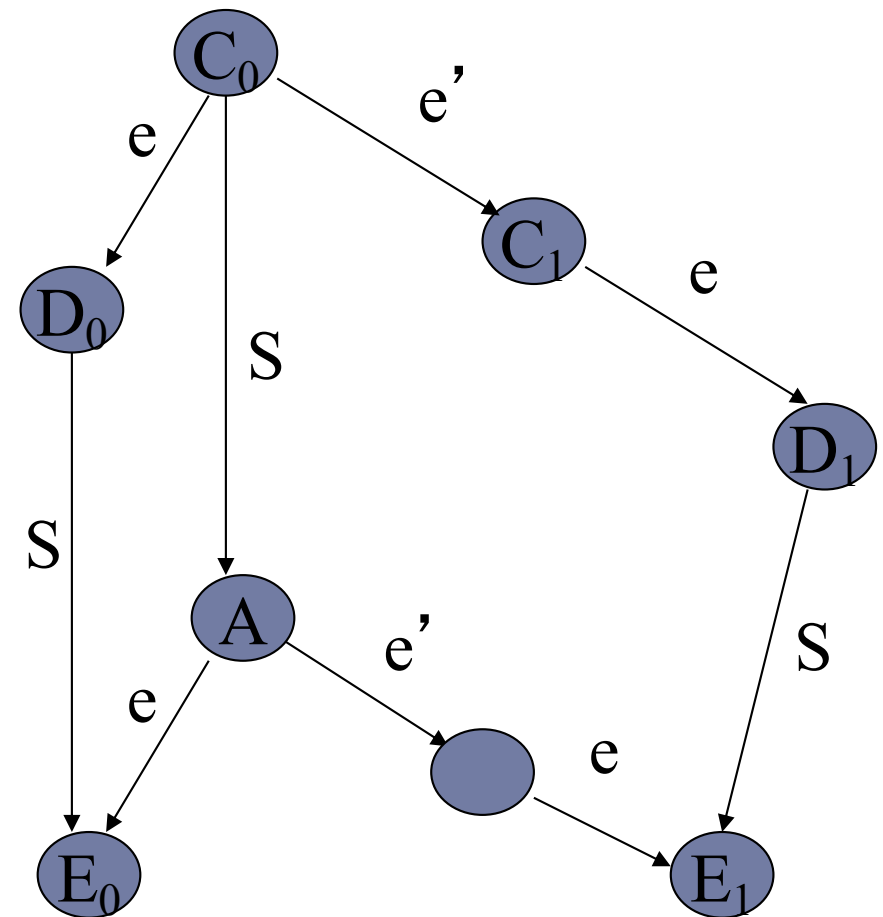
FLP impact on distributed systems design

▸ **FLP proves that any fault-tolerant algorithm solving consensus has runs that never terminate**

  ▸ These runs are extremely unlikely ("probability zero")

  ▸ Yet they imply that we can't find a totally correct solution

  ▸ And so "consensus is impossible" ("not always possible")

▸ **A distributed system trying to agree on something in which process p plays a key role will not terminate if p crashes**

So what can we do?

▸ Alternative? Synchronous models? BUT REAL, PRACTICAL SYSTEMS ARE NOT SYNCHRONUS !!!

▸ Use randomization, probabilistic guarantees

▸ Process groups: sacrifice liveness under the assumption that retransmissions will eventually be received from good participants, the protocol eventually terminates

▸ Avoid consensus, use quorum systems

Partial Synchrony

let Δ be the time it takes to send a message

- ## Synchronous:
  - Δ is known and fixed

- ## Asynchronous:
  - There is no bound for Δ

- ## Partially synchronous:
  - Version 1: There is a bound for Δ but it it not known
  - Version 2: Bound for Δ is known and holds after a time T

Consensus: Summary so Far

▶ Considered only benign failures

▶ In synchronous systems we have a f+1 rounds consensus algorithm that can tolerate f failures, f < n

▶ In asynchronous systems

  ▶ We can not solve consensus

  ▶ We can order events and determine consistent snapshots

3: Byzantine Agreement

Byzantine Failures

▸ Model arbitrary failures

▸ Model insider attacks

▸ Name comes from a paper that introduced the problem "Byzantine Generals Problem"

▸ A process having Byzantine behavior can:

  ▸ Crash

  ▸ Delay messages

  ▸ Refuse to forward messages

  ▸ Two-face behavior

  ▸ Lie

Byzantine Generals Problem
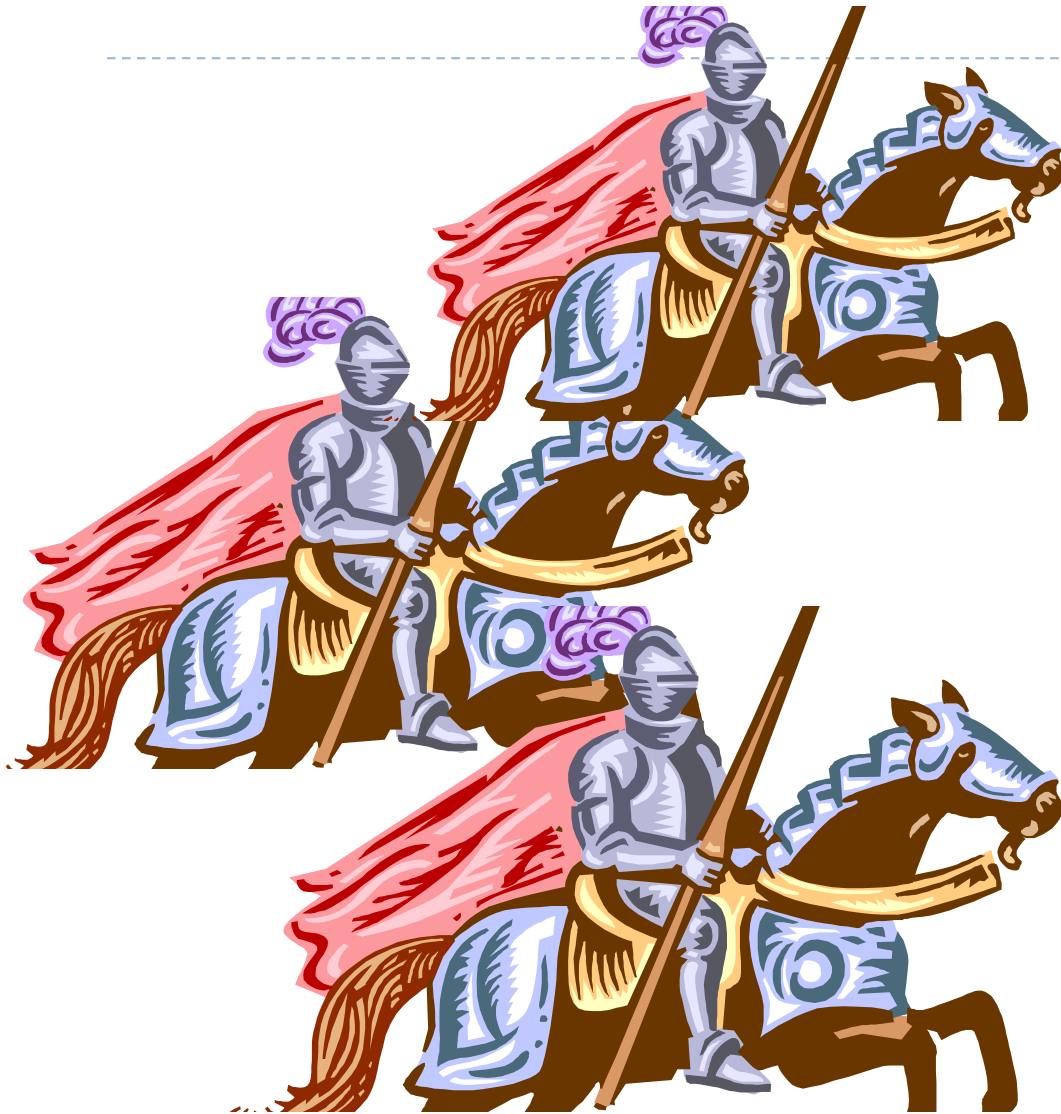
- Several Byzantine generals are laying siege to an enemy city
- They can only communicate by messenger
- They have to agree on a common strategy, *attack* or *retreat*.
- Some general may be traitors (their identity is not known)

Do you see the connection with the consensus problem?

Goals

▸ A: All loyal generals decide on same plan of action

▸ B: A small number of traitors cannot cause a bad plan to be adopted.

▸ B is difficult to formalize

Basic Ideas

▸ To satisfy condition A

  ▸ Have all the general use the same method of combining information to come up with a plan

▸ To satisfy condition B

  ▸ Use a robust method (median function of some sort)

▸ Why not just use the median?

  ▸ Traitors lie and not everyone may have the same information!

Formal Problem Definition

▸ Assume there is one commanding general and his subordinate lieutenant generals.

▸ Commanding general need not be loyal.

▸ IC1: All loyal lieutenants obey the same order.

▸ IC2: If the commanding general is loyal, then every loyal lieutenant obeys the order he sends.

3 Generals: Impossibility Result

▸ **Assume communication is with oral, easily changed messages**

▸ **Oral Messages**

  ▸ Models messages with no integrity, no authentication

  ▸ Contents are under the control of the sender

  ▸ Traitor can do anything to the message

▸ **There is no solution in the case of 3 generals, one traitor makes the protocol fail!**

Scenario A

IC2: If the commanding general is loyal, then every loyal lieutenant obeys the order he sends.

Attack!

Attack!

General said "Retreat!"

Lieutenant 1
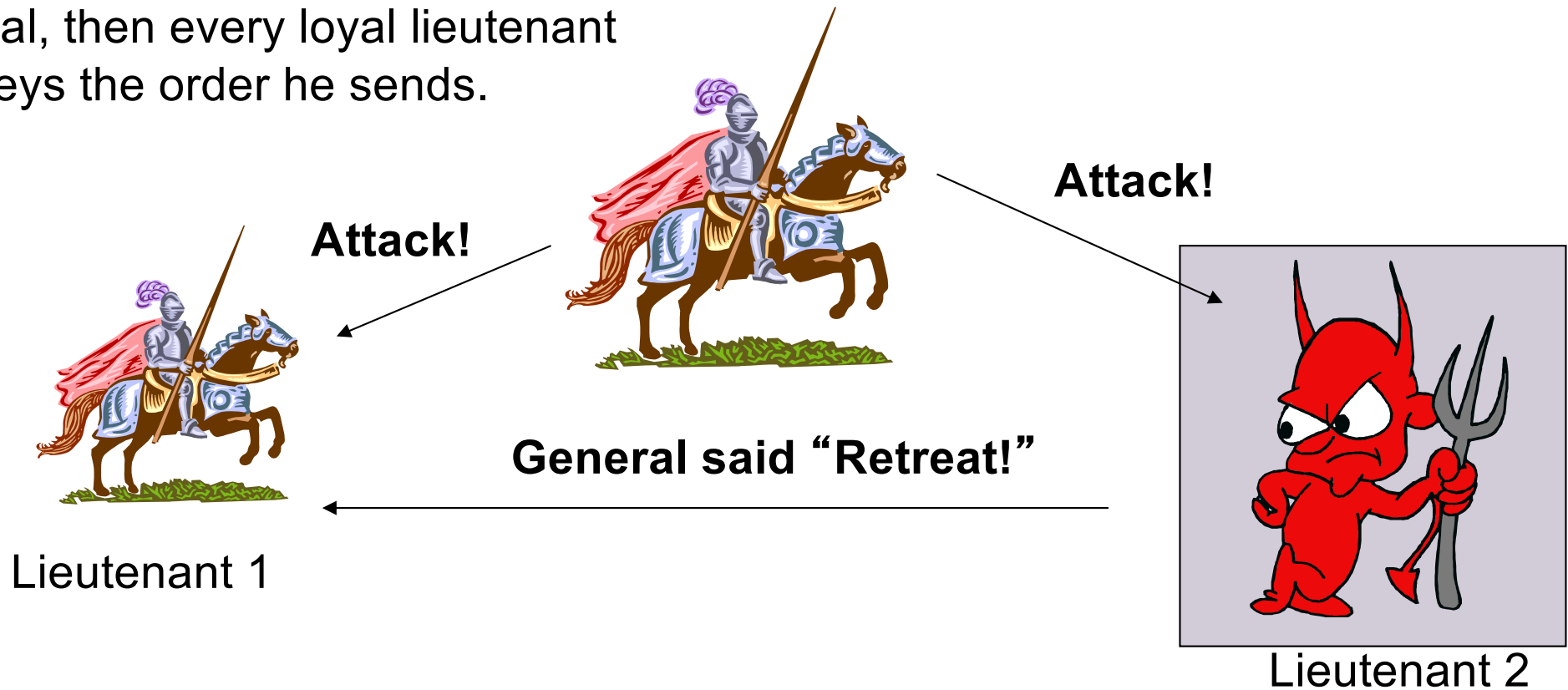
Lieutenant 2

To satisfy IC2, Lieutenant 1 must obey the order from the commander

Scenario B

IC1: All loyal lieutenants obey the same order

Attack!

Retreat!

General said "Retreat!"

General said "Attack!"

Lieutenant 1
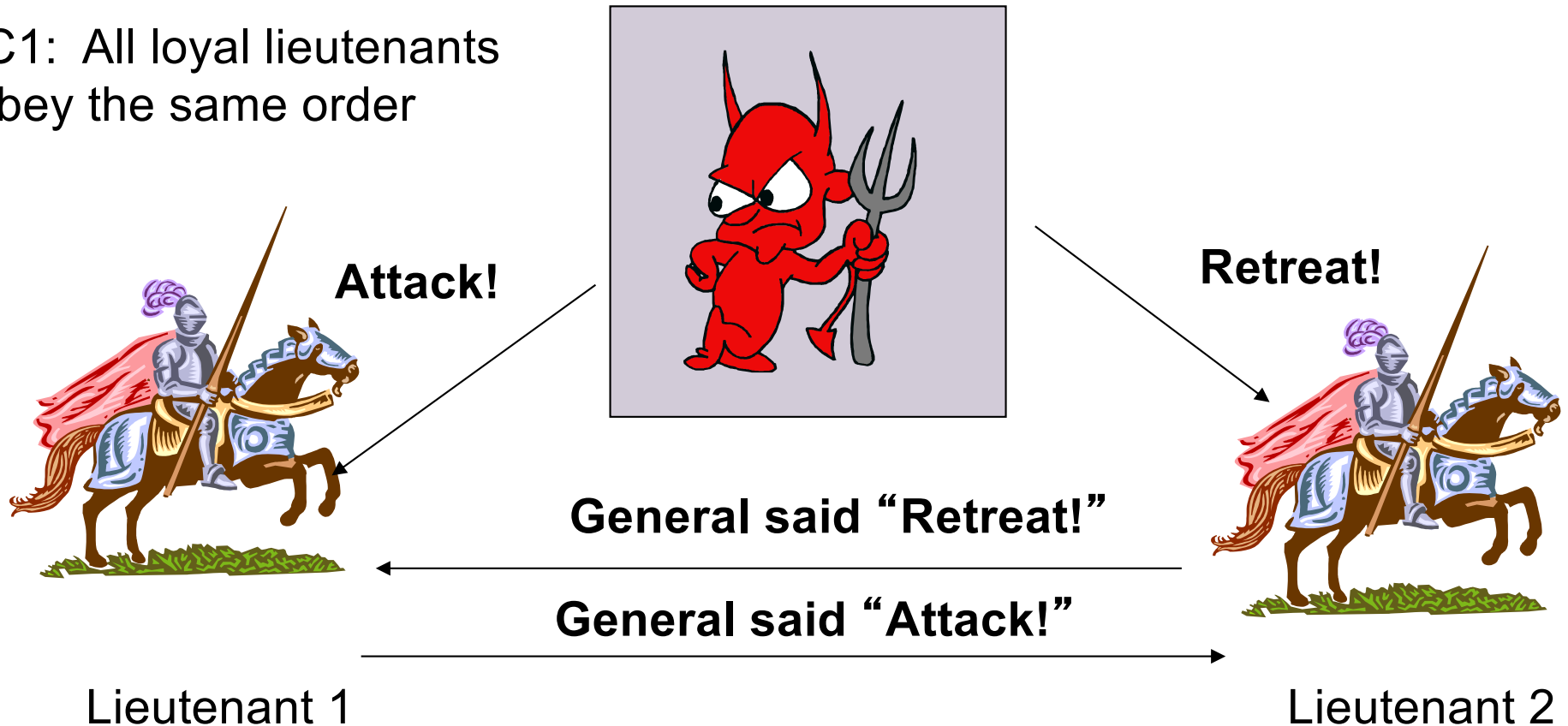
Lieutenant 2

Lieutenant 1 can not distinguish between scenario A and scenario B

Generalizing the Impossibility Result

▸ There is no solution for fewer than **3m+1** generals in the presence of m traitors

Commander and m-1 lieutenants

m lieutenants   m lieutenants

Proof Sketch

▸ By contradiction: assuming that there is a correct solution in 3m case with m traitors, we construct a solution for the case with 3 generals

▸ We call the 3m case the Romanians Generals.

▸ Mapping: each Byzantine general will simulate some of the Romanian generals:

  ▸ Byzantine commander: simulates Romanian commander + m-1 Romanians

  ▸ Each of the 2 Byzantine lieutenant: simulates at most m Romanians

  ▸ Since only one Byzantine can be a traitor, at most m Romanians can be traitors

IC1 and IC2 for Romanians, imply the same properties for the Byzantine

Consensus

Solution with Oral Messages

▸ For 3m+1 generals, solution tolerates m traitors.

▸ Oral messages – the sending of content is entirely under the control of sender.

  ▸ A1 – Each message that is sent is delivered correctly.

  ▸ A2 – The receiver of a message knows who sent it.

  ▸ A3 – The absence of a message can be detected.

▸ What do we get from the assumptions?

  ▸ Traitors cannot interfere with communication as third party.

  ▸ Traitors cannot send fake messages

  ▸ Traitors cannot interfere by being silent.

▸ Default order to "retreat" for silent traitor.

Consensus

Oral Message Algorithm

▸ **Recursive algorithm**

  ▸ Each general (a.k.a. lieutenant) forwards on received values to all other lieutenant

  ▸ The commander sends his value to every lieutenant

  ▸ For each lieutenant i, broadcast the values to all other lieutenants who have not had the value.

  ▸ Take the majority function of the received values.

  ▸ To distinguish between messages from different "rounds", index them using the lieutenant's number i

▸ **Intuition:**

  ▸ Generals might have contradictory data

  ▸ In each round, each participant sends out "witness" messages: here's what I saw in round i

More Details…

- **Algorithm OM(0)**
  - Commander sends his value to every lieutenant.
  - Each lieutenant (L) uses the value received from commander, or RETREAT if no value is received.
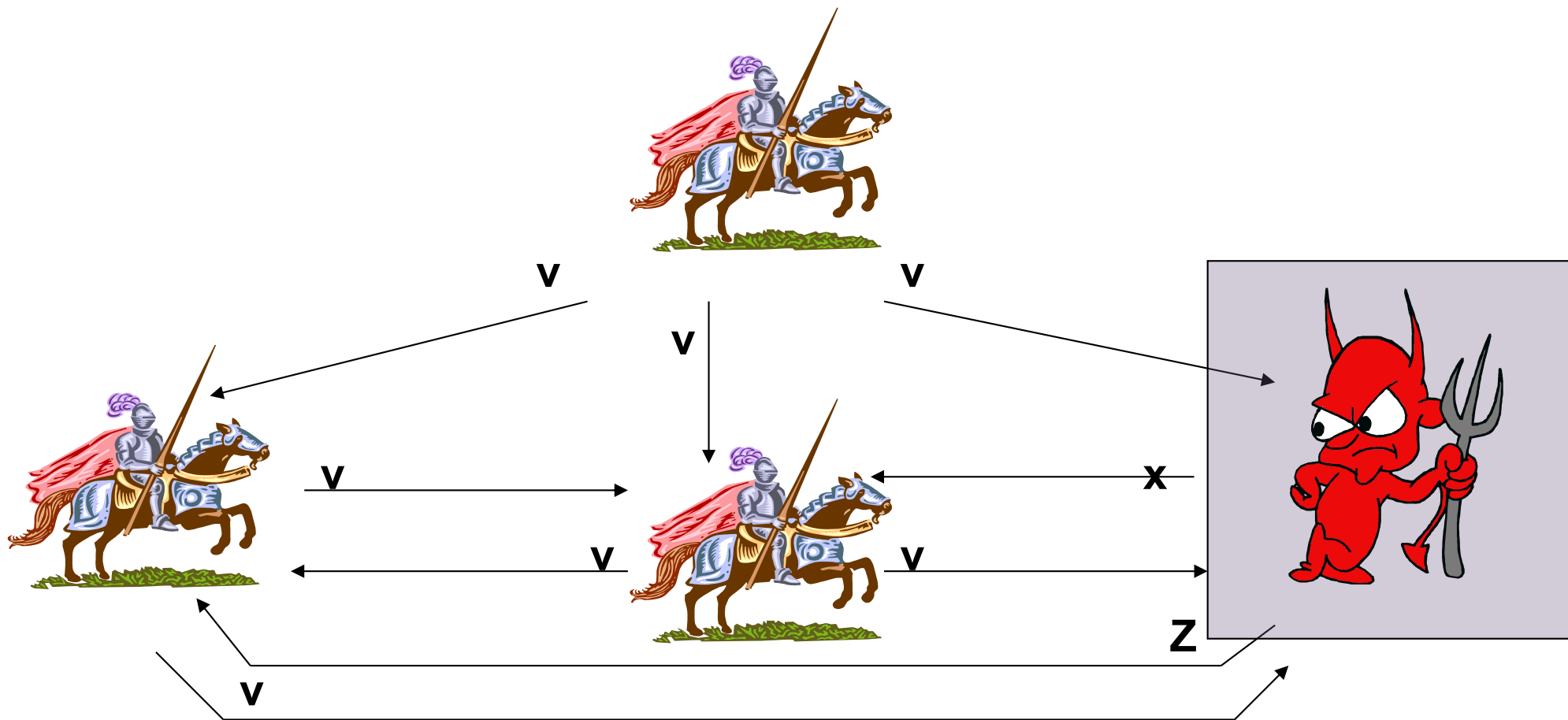
- **Algorithm OM(m), m>0**
  - Commander sends his value to every Lieutenant (vi)
  - Each Lieutenant acts as commander for OM(m-1) and sends vi to the other n-2 lieutenants (or RETREAT)
  - For each i, and each j<>i, let vj be the value lieutenant i receives from lieutenant j in step (2) using OM(m-1). Lieutenant i uses the value majority (v1, …, vn-1).

Consensus

Example with m=1 and n=4
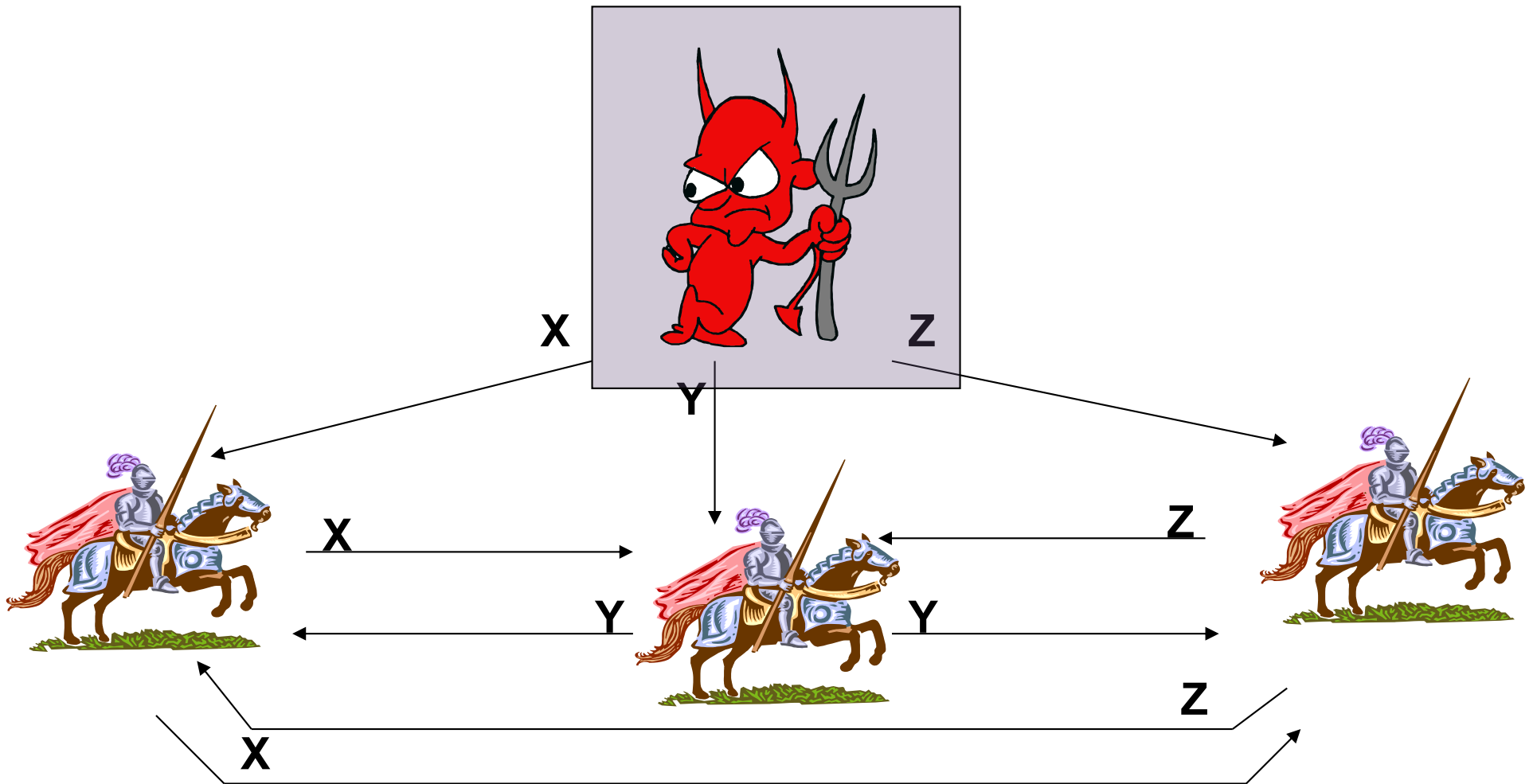
Each lieutenant obtains v1 = x, v2 = y, v3 = z, which all result in the same value when the majority is taken

Example with m=1 and n=4

IC1 and IC2 are met.

Proof Sketch

▸ For any m, algorithm OM(m) satisfies conditions IC1 (All loyal lieutenants obey the same order) and IC2 (If the commanding general is loyal, then every loyal lieutenant obeys the order he sends) if there are more than 3m generals and at most m traitors.

▸ Induction on m proves true in all cases.

Written Messages Solution

▶ Written messages (messages are digitally signed)

    ▶ A loyal general's signature cannot be forged or changed and anyone can verify authenticity

▶ Three general solutions now exist!

▶ Works for any n >= m+2 (1 non-faulty commander and 1 loyal lieutenant)

Signed Message Algorithm

▸ General sends a signed order to all his lieutenants

▸ Each lieutenant signs and forwards on the message he received to all the other lieutenants until every message has been signed by everyone else.

▸ Each lieutenant keeps track of the properly signed orders he has received

  ▸ Possible orders are attack, retreat, attack & retreat

▸ Use choice method to have everyone choose same value

Why Does It Work?

▸ Every loyal lieutenant eventually has the same set of signed messages, resulting in the same choice

▸ If commander is loyal, then all loyal lieutenants will have correct messages

▸ If the commander is a traitor, lieutenants receive conflicting messages and but still end up choosing the same choice (retreat).

Revisit the Requirements

▸ **Absence can be detected**

  ▸ Timeout mechanisms are needed

  ▸ Synchronous Communication

▸ **Network is connected (different requirements if the network is not a complete graph)**

▸ **A signature cannot be forged or changed and anyone can verify authenticity**

  ▸ Message signed by i= (M, S(M))

  ▸ Crypto and modular arithmetic

Termination

▸ How many rounds do you need?

▸ For the Byzantine generals algorithm described before you need f+1

Consensus: Summary

▸ **In synchronous systems with benign failures we have a f+1 rounds consensus algorithm that can tolerate f failures, f < n**

▸ **In asynchronous systems**

  ▸ We can not solve consensus

  ▸ We can order events and determine consistent snapshots

▸ **Byzantine failures**

  ▸ No solution for fewer than 3m+1 generals in the presence of m traitors

  ▸ Oral message and written (signed) messages solutions exist

4: Randomized Agreement.

Based on slides by James Aspnes

Consensus in Asynchronous Systems

▶ There is no asynchronous algorithm that achieves agreement on a one-bit value in the presence of crash faults. The result is true even if no crash actually occurs!

▶ Also known as the FLP result

▶ Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson for "Impossibility of Distributed Consensus with One Faulty Process," Journal of the ACM, April 1985, 32(2):374

So what can we do?

▶ Alternative? Synchronous models? BUT REAL, PRACTICAL SYSTEMS ARE NOT SYNCHRONOUS !!!

▶ **Use randomization: probabilistic guarantees for termination**

▶ Use process groups: sacrifice liveness under the assumption that retransmissions will eventually be received from good participants, the protocol eventually terminates

▶ Use quorum systems: avoid consensus among all participants

Randomized Algorithms

▸ Algorithm that uses randomness by adding a coin-flip to the distributed model.

▸ Adversary

  ▸ A function from partial executions to operations.

  ▸ Chooses which operation happens next.

  ▸ Simulates the executing environment.

Types of Adversaries

▸ **Strong adversary**

  ▸ Adversary can see the entire history of execution: outcomes of coin flips, internal states of processes, contents of messages.

▸ **Weak adversary**

  ▸ Adversary chooses for each state, which process executes next, etc.

Randomized Agreement

- Addition of coin-flip to distributed model.

- Agreement: all non-faulty processes agree on the same value

  - Validity: if a process decides on a value, then there was a process that started with that value

- Termination – for all adversaries, every non-faulty process terminates with probability 1

Coin Models

▸ Global reliable coin: not always possible

▸ Local coin: each process tosses a coin independently, works well when the total number of processes is relatively large to the number of faulty ones

Ben-Or's Consensus Protocol

▸ **First protocol to achieve consensus with probabilistic termination in a model with a strong adversary (1983)**

  ▸ Tolerates $t < n/2$ crash failures.

  ▸ Requires exponential expected time to converge in the worst case.

▸ **Strong adversary**

  ▸ Can see the entire history of execution: outcomes of coin flips, internal states of processes, contents of messages.

  ▸ BUT, every message sent to a correct process must eventually be received and the final schedule may have at most t crashed processes.

Ben-Or: Coin Toss

▸ Each process tosses a coin independently

▸ Uniform distribution, the coin outputs 0 or 1 each with probability ½

▸ Used by a process to pick a new local value when a majority was not found

Ben-Or's Consensus Protocol

▶ Operates in rounds, each round has two phases.

▶ Suggestion phase – each process transmits its value, and waits to hear from other processes.

▶ Decision phase – if majority found, take its value. Otherwise, flip a coin to decide a new local value.

Purpose of rounds

▸ If some process decides v then by the next round all the other operating processes will decide the same value v.

Main Ideas

- Exchange initial values and if enough processes detected the majority, decide.
  - Wait for only n-t messages to avoid blocking.

- If a process knows that someone detected majority, switch to the majority's value.

- Terminates, because eventually, the majority of processes will flip coins correctly.

- Algorithm does not wait for all processes, because they might be dead.

- Remember this is asynchronous execution model : no assumptions about the relative speed or about the delay in delivering a message

Ben-Or's Consensus Protocol

Input: Boolean initial consensus value

Output: Boolean final consensus value

Data: Boolean preference, integer round

begin

      preference := input

      round := 1

  while true do

    Body of while statement

  end

end

Body of while Statement

send (1, round, preference) to all processes

wait to receive n – t (1, round, *) messages

if received more than n / 2 (1, round, v) messages with same v

then send (2, round, v, ratify) to all processes

else send (2, round, ?) to all processes

end

n> 2t

wait to receive n – t (2, round, *) messages

If received a (2, round, v, ratify) message

then preference = v

  if received more than t (2, round, v, ratify) messages

  then output = v

  end

else preference = CoinFlip()

end

round = round + 1

Halting

▶ Once a correct process decides a value, it will keep deciding the same value in all subsequent phases.

▶ Easy to modify the algorithm so that every process decides at most once, and halts at most one round after deciding.

Agreement

▸ At most one value can receive majority in the first stage of a round.

▸ If some process sees t + 1 (2, r, v, ratify), then every process sees at least one (2, r, v, ratify) message.

▸ If every process sees a (2, r, v, ratify) message, every process votes for v in the first stage of r + 1 and decides v in second stage of r + 1 (if it hasn't decided before).

Validity

▸ If all processes vote for their common value v in round 1, then all processes send (2, v, 1, ratify) and decide on the second stage of round 1.

▸ Only preferences of one of the processes is sent in the first round.

Termination

▶ **If no process sees the majority value:**

  ▶ Processes will flip coins, and start everything again.

  ▶ Eventually a majority among the non-faulty processes flips the same random value.

  ▶ The non-faulty processes will read the majority value.

  ▶ The non-faulty processes will propagate ratify messages, containing the majority value.

  ▶ Non-faulty process will receive the ratify messages, and the protocol finishes.

Worst Case Probability

▸ Termination: The probability of disagreement for an infinite time is 0. (It is equal to the probability that every turn there will be one 1 and one 0 forever).

▸ Complexity: The chance of n coins to be all 1 (assuming a fair coin) is 2-n. Hence, the expected time of the protocol to converge is O(2n).

▸ This worse case happens if there are (n-1)/2 faulty processes.

▸ If faulty processors are O(√n) then number of rounds to reach consensus is constant

Summary

▸ Agreement is a fundamental problem in distributed systems

▸ Realistic models include failures and asynchronous communication

▸ No solution in asynchronous systems

▸ Solutions exist for synchronous communication for both stop failures and byzantine failures

▸ Randomized algorithms solutions exist in asynchronous communication