05_01_Fault Tolerance and Consensus

1. Faults and Failures

1.1. Fault Classification

1.1.1. Permanent Faults

for example:

- processors halting

- malfunctioning sensor giving erroneous result

will cause:

- crash failures
- Byzantine Failures

1.1.2. Transient Faults

for example:

- part of memory momentarily corrupted caused by power glitch
- transmission errors

1.2. Permanent Faults

Dealing with permanent faults has often **been modeled as** the need for **reaching agreement or consensus** among a set of processors some of which may exhibit faults

1.2.1. Crash Failures (stopping failure)

- a processor simply **stops at some point and does not resume at alater time**
- We assume that a processor does not stop in the middle of sending a message
- Stopping failures model the event of a processor going down

1.2.2. malicious or Byzantine failure

Byzantine failures model a component **continuing to operate but exhibiting failures** (such as a sensor giving values with bits inverted).

for example:

- stop for a while then continue
- sending error message

1.3. Transient Failures

Transient faults are faults exhibited by components that **will return to normal operation** after a while. Transient faults cause the state of a system to be incorrect.

- The incorrect (or the absence of a proper) initialization of a distributed system

- The corruption of a part of the **main memory** of a number of processors. We will assume that only the **variables of processes** can be corrupted (including the program counter), but **not the programs they execute**

2. Consensus in Synchronous Systems with Crash Failures

2.1. Target

Every process **starts with a value** from the set of possible initial values, and they have to **reach agreement** which satify:

- **<u>Agreement:</u>** No two processes decide on different values
- **<u>Validity:</u>** If all processes start with the same value, then no process decides on a different value;
- **<u>Termination:</u>** All non-faulty processes decide within finite time

2.2. Simple Flooding Algorithm

<u>Assumption</u>

1. In a synchronous system
2. at most **f** crash failures (Each participants know exactly the upper-bound number of faults in the system)
3. the set of value domain contains a default value (e.g., 0)that processors can resort to in case they do not know what value to use.

<u>Preparation</u>

Every process matins a set W, initially W={v}

<u>Main Process</u>

1. Every process starts with a **value v**
2. for the first f+1 rounds, each node:
    a. broadcast(W) to all other processes
    b. receive(W) from all processes and set W to the union of their current set W and all sets they receive
3. Finally:
    a. if W contains only a single element v, decide(v)
    b. else decide(default)

<u>Understanding</u>

The alg solves a problem: how to achieve an agreement even if some process crashed.

1. Because the algorithm consists of f +1 rounds and there are at most f processor failures, **there is at least one round during which no processor fails**
2. In this round all active processes can obtain a identical sets W and the final active processes is a subset of this moment active processes

<u>Complexity</u>

only consider message complexity:

n nodes:

Optimization

processes only need to know whether at the end |W|=1 or |W|>1

so let processes only broadcast at most two values:

- their initial value
- the first different value they receive (broadcast when it is active until the process is finished)

One basic idea is only the set W of the final active process matters, that means:

- if a process receives a different value and immediately crashed, it does not matter
- an throughout active process will eventually broadcast its value/different value received to other active processes

3. Byzantine Model Preliminary

3.1. Conditions for a solution for Byzantine

3.1.1. Necessary and sufficient condition

$$f < n/3$$
$$(n-f)/2 > f$$

3.1.2. minimal number of rounds in a determinisitic solultion

$$f+1$$

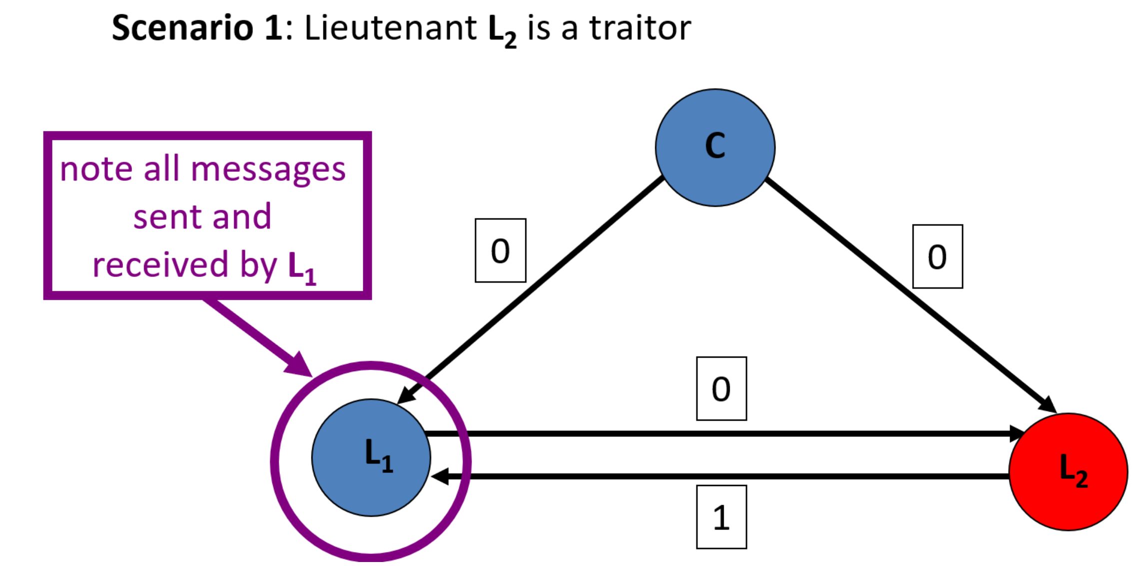
3.2. Impossibility results

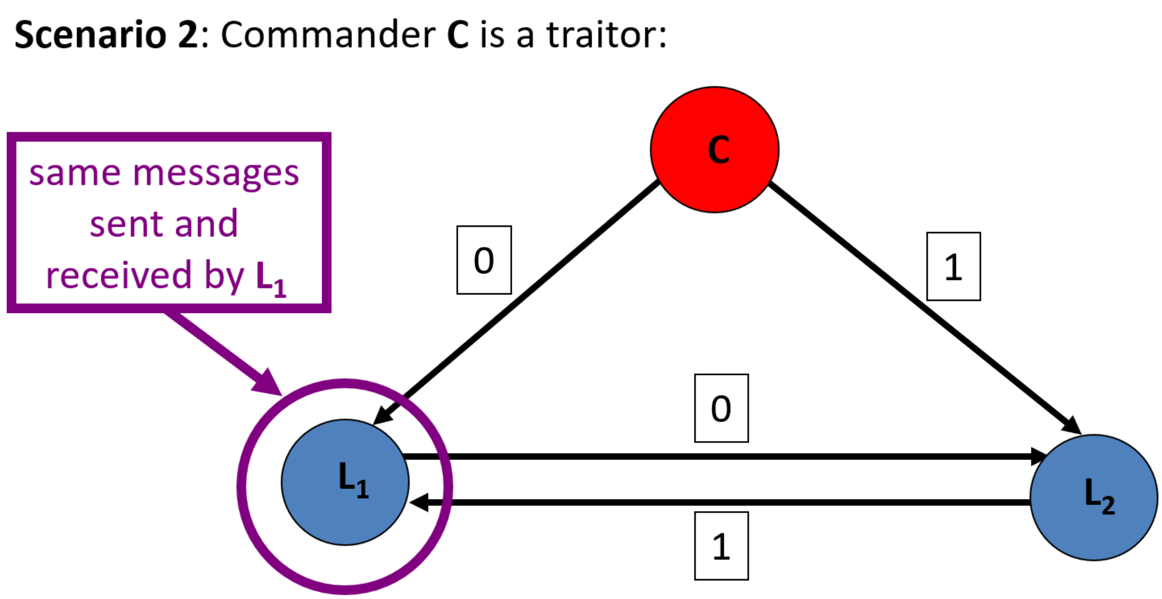
For some consensus problems no solutions exist.

- in purely asynchronous system: there is no possibility to overcome even a single processor failure
- some interesting examples: in a synchronous case of three generals, one of which is a traitor

  In this two situation, there is no difference between the messages L1 received, **so L1 is not able to judge the loyalty extent of others according to the message received**, so he can just obey the commander's order.

  Same thing happen in L2, so L1 has 0 in case 2, L2 has 1 in case 2, does not allow *agreement*





3.3. Target

- **Agreement:** All non-faulty nodes agree on the same value
- **Validity:** if the source is non-faulty, then all non-faulty nodes agree on the initial value of the source
- **Termination:** all processes decide within finite time

So:

- if the source is faulty, the non-faulty processes can agree on any value, but they have to agree on the same value

- what value a faulty process decides is irrelevant

4. Consensus in Synchronous Systems with Byzantine Failures

4.1. Assumption for Three Algorithms

1. Each participants know exactly the upper-bound number of faults in the system
2. Sometimes, we call broken node as **non-faulty nodes**

4.2. Authentication

- **without authentication:** means that a non-initial-proposer node can forge message
- **with authentication:** means that a non-initial-proposer node cannot forge message

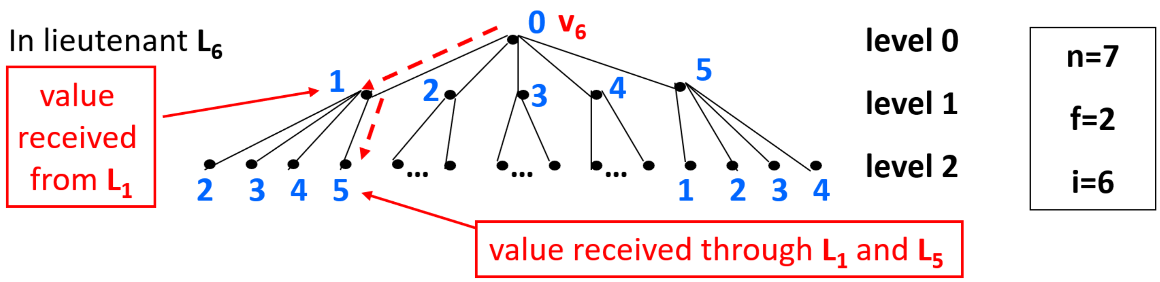
4.3. Lamport-Pease-Shostak algorithm without authentication in synchronous systems

Assumption

- Nodes **know how many faulty nodes (f) (upper-bound)** are in the network

Preparation(Structure)

- **Messages should carry**
  - value
  - f-related parameter
  - the nodes passed (follow sequence)
- **labelled tree for decision**

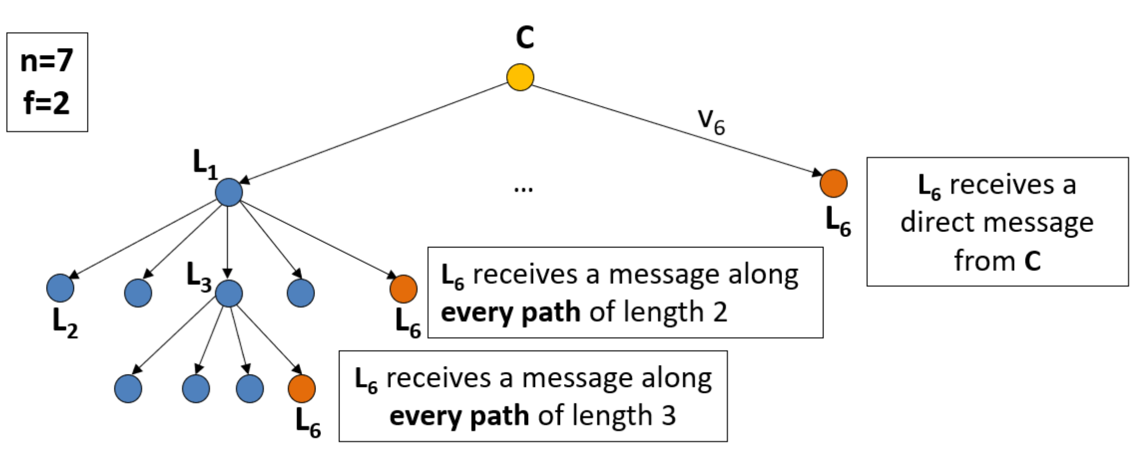


- Label the nodes of the tree in $\mathbf{L_i}$ with **additional labels**:
  - **level 0**: $\mathbf{v_i}$ (value received from the commander)
  - **level 1**: the value that $\mathbf{L_j}$ told $\mathbf{L_i}$ that the commander told him
  - label of any node: the value that was passed to $\mathbf{L_i}$ from the commander **through the chain of lieutenants on the path** from the root to the node

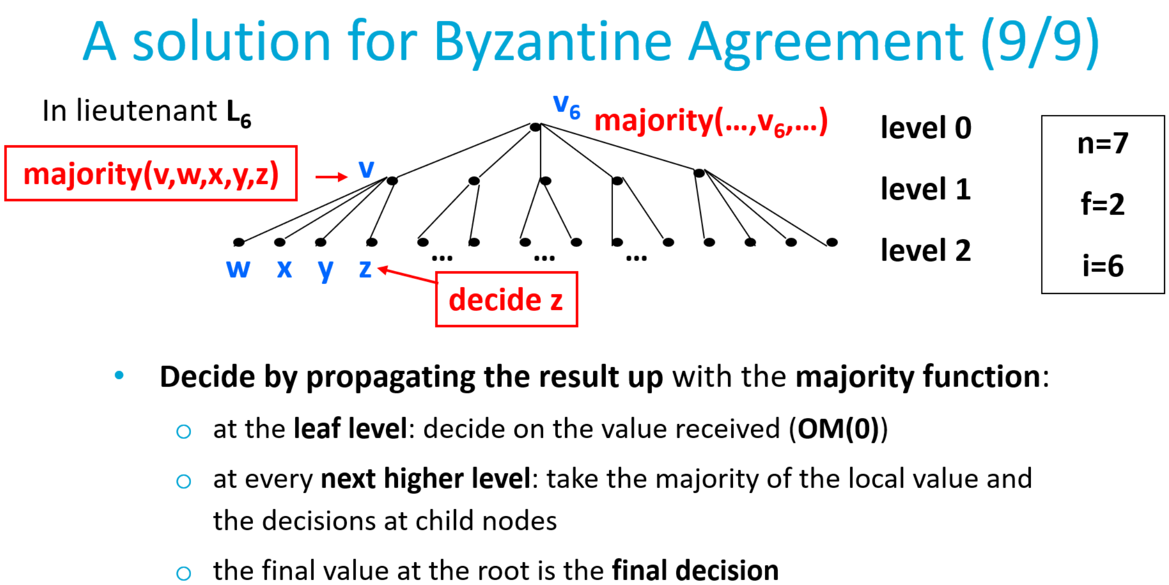
  - each path is the path of a certain message
  - **label of edges are a value (the value that La told told Li that Lb told La that Lc told Lb that .....)**

<u>Process</u>

- When **f==0 OM(0):** the commander sends his value to the lieutenants, who simply decide on this value
- When **f≠0 OM(f)**:
  - the commander broadcasts its initial value
  - let v_i be the value received from the commander by lieutenant L_i or the **default** if no value is received
  - doing recursive steps
- recursive steps:
  - L_i executes OM(f-1), acting as the commander and send his value to other lieutenants （lieutenants that not appear in a OM(f) message）
  - The recursive steps is as a tree, in each sub-tree, the k-th layer (start from 0) has n-k nodes



- make decision "recursively",consider the decision tree:



- from bottom to top
- decide **final value**=majority(v_commander, v of layer 1)
- each node **in the each layer: value**=majority(v_node,v_childnode)
- if there is not a majority value, it partly means the commander is non-faulty, so all faulty node decides on an default value

Implementation

Implementation:

I. Code executed by the commander in `OM(f)`

broadcast `(v)`

II. Code executed by the lieutenants in `OM(0)`.

receive(v)

`order` $\leftarrow$ `v`

III. Code executed by lieutenant L_i in `OM(f)`

if receive `(v)` **then**

  $\mathtt{v_i} \leftarrow \mathtt{v}$

else

  $\mathtt{v_i} \leftarrow \mathtt{default}$

$\mathtt{OM}(\mathtt{f}-1, \mathtt{v_i}, \mathtt{i}, 1, \ldots, \mathtt{i}-1, \mathtt{i}+1, \ldots, \mathtt{n}-1)$

$\mathtt{v'_j} \leftarrow \mathtt{order\ \ in}\ L_i\ \mathtt{of}\ \mathtt{OM}(\mathtt{f}-1, \mathtt{v_j}, \mathtt{j}, 1, \ldots, \mathtt{j}-1, \mathtt{j}+1, \ldots, \mathtt{n}-1)$

$\mathtt{order} \leftarrow \mathtt{majority}(\mathtt{v'_1}, \ldots, \mathtt{v'_{i-1}}, \mathtt{v_i}, \mathtt{v'_{i+1}}, , \ldots, \mathtt{v'_{n-1}})$

Understanding

Complexity(easy to understand from the tree)

- Number of executions (nodes in each layer)
  - OM(f) 1 time
  - OM(f-1) (n-1) times
  - OM(k) (n-1)(n-2)...(n-f+k) times
- Number of messages: n^{f+1} in total (edges in each layer)
  - OM(f): n-1
  - OM(f-1): (n-1)(n-2)
  - OM(k): (n-1)(n-2)...(n-(f-k))(n-(f-k+1))
  - OM(0): (n-1)(n-2)...(n-(f+1))

4.3. Lamport-Pease-Shostak algorithm with authentication in synchronous systems

Assumption

- Nodes **know how many faulty nodes (f) (upper-bound)** are in the network
- Every message carries a **signature**

- The signature of a loyal general **cannot be forged**
- Alteration of the contents of a signed message can be detected
- Every (loyal) general can verify the signature of any other (loyal) general

Preparation(structure)

- Message
    - value
    - sequence of general: from (and signed by) the commander, and subsequently signed and sent by lieutenants L_{i1}, L_{i2}
- Every lieutenant maintains **a set of orders V**
- In this algorithm, we can define different function on V for deciding, but all nodes should use the same function(e.g. majority, minimum)
- lieutenants **wait until** they have received a message for all possible strings of signatures of **length f+1** (with signature 0 as the first), “do long enough”

Implemenation

```
Implementation:
I. Code executed by the commander
broadcast([v,0])

II. Lieutenant L_i:
V ← ∅
do long enough
    upon receipt of [v,0,s] do
        if (v ∉ V) then
            V ← V ∪ {v}
            if (len(s) < f) then
                for j=1 to i-1,i+1 to n-1, not in s do
                    send([v,0,s,i]) to L_j
order ← choice(V)
```

$s \rightarrow$ sequence of general

$len(s) \rightarrow$ length of s

Understanding

4.3. The Srikanth-Toueg algorithm for consensus with authenticated broadcast in synchronous systems with a completely connected network (no exam material)

5. Randomized Solution

5.1. Background

To introduce a randeom element in disrributed systems is useful, that is allow one or more processes to **flip a coin** once in a while to make progress towards a solution.

It may has two results:

- achieve **more efficient solutions** than deterministic solutions
- achieve a solution for problems for which **no deterministic solution exists**

By using randomization, something has to be **sacrificed**

5.2. Randomized Agreementwith Crash Failures (Ben-Or's)

Ben-Or's randomized algorithm for consensus in **synchronous and asynchronous systems** with crash failures

Assumption

- the number of traitors f satisfies $f < \frac{n}{2}$

Preparation(Structure)

- Messages has a type
  - N for notification phase
  - P for proposal phase
  - D for decision phase

Process

- Each Process **starts with a binary input value v**
- Proceeds in rounds (indexed by r), each round has 3 phase
  - notification phase: broadcast message **(N;r,v)**
  - proposal phase: await **n-f (N;r,*)** from others, if:
    - more than **n/2** of these messages has value 0 or 1, then broadcast **(P;r,v)**

        - else broadcast **(P;r,?)**
    - decision phase
        - if **decided**, then stop
        - else await **n-f** message **(P;r,*)**
            - if more than **1** message **(P;r,w)** with value 0 or 1, own value v ⟵ w
                - if more than **f** messages **(P;r,w)**, decide on w, decided
            - else make own value v a **random value** 0 or 1
            - increase the r value, start a new round

When a process expects messages from all processors, it is **no use waiting** for more than n − f messages. When not enough processors support a possible decision, a process **starts the next round with a new, random value v.**

Understanding

Implementation

Implementation:

```
r ← 1
decided ← false
  do forever
    broadcast (N;r,v)
    await n − f messages of the form (N;r,*)
    if (> n/2 messages (N;r,w) received with w=0 or 1) then
      broadcast (P;r,w)
    else broadcast (P;r,?)
    if decided then STOP
    else await n − f messages of the form (P;r,*)
    if (> 1 messages (P;r,w) received with w=0 or 1) then
      v ← w
      if (> f messages (P;r,w)) then
        decide w
        decided ← true
    else v ← random(0,1)
    r ← r + 1
```

5.2. Randomized Byzantine Agreement (Ben-Or's)

Ben-Or’s randomized algorithm for consensus in **synchronous and asynchronous systems** with Byzantine failures

5.3.1. Assumption

- the number of traitors f is $n > 5f$

5.3.3. Algorithm

<u>Preparation(Structure)</u>

- Messages has a type
  - N for notification phase
  - P for proposal phase
  - D for decision phase

<u>Process</u>

- Each Process **starts with a binary input value v**
- Proceeds in rounds (indexed by r), each round has 3 phase
  - notification phase: broadcast message **(N;r,v)**
  - proposal phase: await **n-f (N;r,*)** from others, if:
    - more than **(n+f)/2** of these messages has value 0 or 1, then broadcast **(P;r,v)**
    - else broadcast **(P;r,?)**
  - decision phase
    - if **decided**, then stop
    - else await **n-f** message **(P;r,*)**
      - if more than **f** message **(P;r,w)** with value 0 or 1, own value v $\leftarrow$ w
        - if more than **3f** messages **(P;r,w)**, decide on w, decided
      - else make own value v a **random value** 0 or 1
      - increase the r value, start a new round

  When a process expects messages from all processors, it is **no use waiting** for more than n − f messages. When not enough processors support a possible decision, a process **starts the next round with a new, random value v.**

<u>Implementation</u>

```
r ← 1
decided ← false
    do forever
        broadcast (N;r,v)
        await n − f messages of the form (N;r,*)
        if (> (n + f)/2 messages (N;r,w) received with w=0 or 1) then
            broadcast (P;r,w)
        else broadcast (P;r,?)
        if decided then STOP
        else await n − f messages of the form (P,r,*)
        if (> f messages (P;r,w) received with w=0 or 1) then
            v ← w
                if (> 3f messages (P;r,w)) then
                    decide w
                    decided ← true
        else v ← random(0,1)
        r ← r + 1
```

Correctness Probability:

if **n> 5f**,Algorithm 5.13 guarantees Agreement and Validity, and terminates **with probability 1**

Complexity:

Expected number of rounds is of order 2^n (in fact better)

5.3.3. Mathematical Lemmas about correctness

Lemma 1: If a correct process proposes v in round r,then no other correct process proposes 1-v in round r

"No simultaneous contradicting proposal by correct processes"

Proof:

1. If a process propose the value v, the process must have received more than (n+f)/2 messages (N;r,v)
2. In the (n+f)/2 messages, more than (n+f)/2-f=(n-f)/2 are from correct processes, and this is already a majority of the correct processes.
3. So all correct process rececives less thant (n+f)/2 1-v, so 1-v cannot be proposed

Lemma 2: If at the beginning of round r all correct processes have the same value v,then they all decide v in round r

"when all correct processes have the same value, immediate decision"

Proof:

1. each correct process receives at least **n-f notification messages**, at least **n-2f** of which are from correct processes
2. beacuse **n>5f, n-2f>(n+f)/2**
3. so all correct process propose v
4. so each correct process receives at least n-2f proposal messages from correct process
5. beacuse **n>5f, n-2f>3f>f**, so propoosal from traitor will not be accepted, and all correct process all will decide on v

This Lemma guarantee validity

<u>Lemma 3:</u> **If a correct process decides v in round r,then all correct processes decide v in round r+1**

"Decision of any correct process immediately followed by others"

Proof

1. if a process decides v in round r, it must have received more than 3f proposals of v,
2. in these proposals, m of which are from correct processors and m>2f (at most f from incorrect with v)
3. so every other correct processor receives at least m-f>f proposal for v (because at most f messages from correct processes will not be received)
4. so they start with v from next round

5. then use Lemma2

This Lemma guarantee agreement

5.4. Remark

randomization is used only if there is not enough initial support for any decision anyway