Autonomic Computer Systems CS321: Self-Stabilization, logical clocks, distr. snapshot, Byzantine agreement

September 27, 2011

Prof. Dr. Christian Tschudin

Departement Mathematik und Informatik, Universität Basel

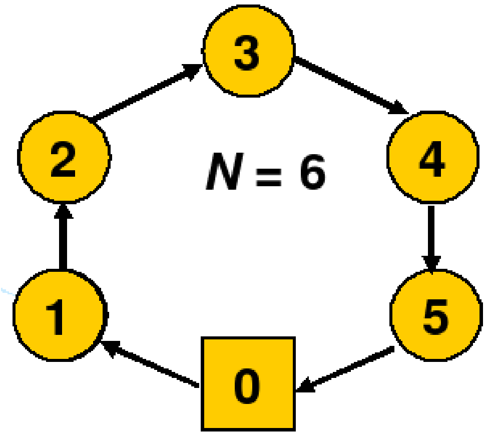
Overview

Selected “Self-” Algorithms (I)

- From last session:
  – Notes on message passing models
  – intro self-stabilization
- **Self-stabilization (example: mutual exclusion)**
- Logical clocks
- Distributed snapshot
- Byzantine agreement

Mutual Exclusion in a Ring

Also known as **Token Ring**:

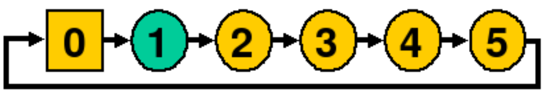


(Thijs Krol, UTwente)

- Anonymous ring (no IDs)
- One special node ’0’, otherwise identical algorithm
- One node is owner (“has token”, “privileged” node)
- Token is passed on clockwise
- Note: no need for all nodes to know who has the token (LE)

Token Ring – Algorithm

N = number of nodes, number m with:
each node i has state $s(i) \in \{0, ...m-1\}$, $m \geq N-1$

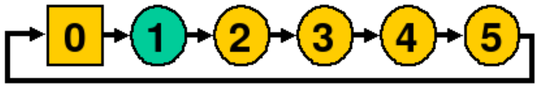


(this shows index, not ID)

```
Node 0: is privileged if           s(0) == s(N-1)
        when releasing token do: s(0) := (s(0) + 1) mod m

Node i: is privileged if           s(i) != s(i-1)
        when releasing token do: s(i) := s(i-1)
```

TR – Algorithm (contd 1: normal)



4	3	3	3	3	3
4	4	3	3	3	3
4	4	4	3	3	3
4	4	4	4	3	3
4	4	4	4	4	3
4	4	4	4	4	4
5	4	4	4	4	4
5	5	4	4	4	4

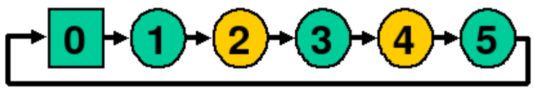
(Thijs Krol, UTwente)

- Example shows initial consistent states: one token circulates.
- Algorithm:

```
Node 0: wins if s(0) == s(N-1)
  Release token:
    s(0) := (s(0) + 1) mod m

Node i: wins if s(i) != s(i-1)
  Release token:
    s(i) := s(i-1)
```


TR – Algorithm (contd 2: convergence)



$\boxed{4}$	3	3	5	5	4
5	3	3	$\boxed{5}$	5	4
5	3	3	3	$\boxed{5}$	4
5	$\boxed{3}$	3	3	3	4
5	5	$\boxed{3}$	3	3	4
5	5	5	3	3	$\boxed{4}$
5	5	5	$\boxed{3}$	3	3
5	5	5	5	$\boxed{3}$	3
5	5	5	5	5	$\boxed{3}$
$\boxed{5}$	5	5	5	5	5
6	$\boxed{5}$	5	5	5	5

- Example shows initial arbitrary states, but only one token survives.
- Algorithm:

```
Node 0: wins if s(0) == s(N-1)
  Release token:
    s(0) := (s(0) + 1) mod m

Node i: wins if s(i) != s(i-1)
  Release token:
    s(i) := s(i-1)
```

Token Ring – Properties

- The algo above is **Self-Stabilizing**
  – arbitrary intial values for $s(i)$
  – yet converges to correct 1-token situation
- Errors can be added, at any time
  – system will regain legal state
- Theorem: There is no Token Ring algorithm
  for an anonymous ring without special node 0.

Note on Self-Stabilization

Problem (and first solution) was proposed by Dijkstra in 1973

- Systems with a selfstab algorithm can recover from
  "loss of coordination"
  (due to node crash, memory crash etc)
- SelfStab **does not mask failures** (e.g. reliable transport, fault tolerance), but **recovers** from failure.

Transition to next topic:
one LE algo requires "timestamps". How to obtain?

Overview

Selected “Self-” Algorithms (II)

- Notes on message passing models
- Self-stabilization (example: mutual exclusion)
- **Logical clocks**
- Distributed snapshot
- Byzantine agreement

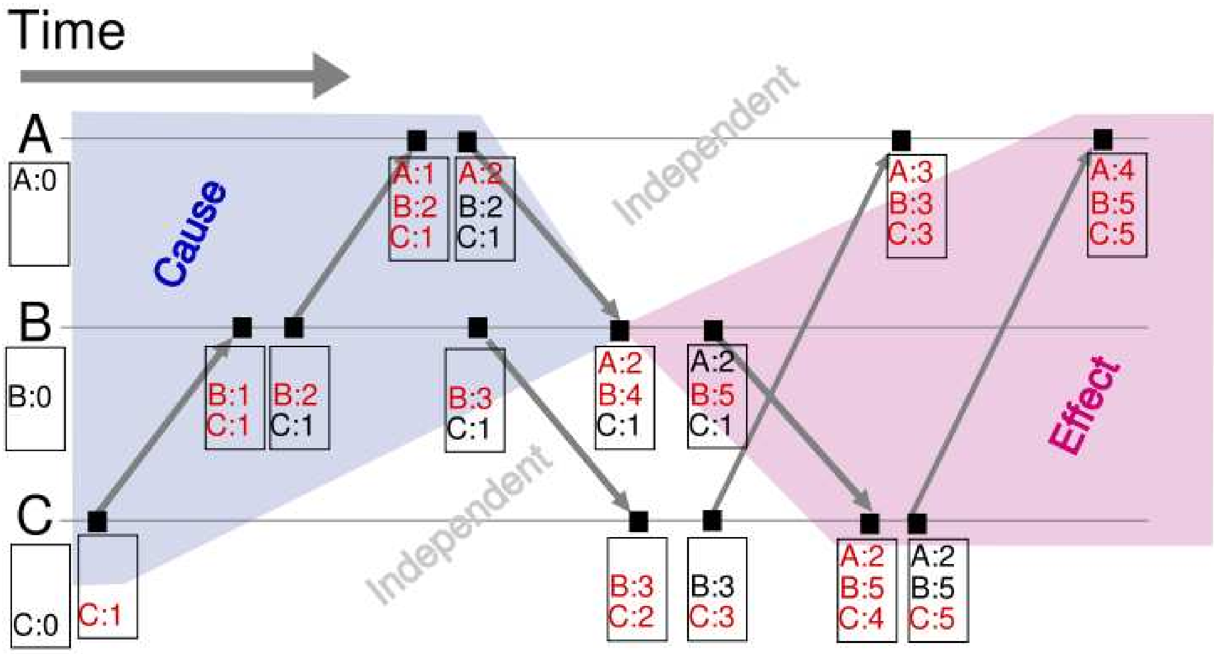
Logical clocks

No global “now”: all message exchange takes some finite time.

- Logical ordering of events:
  – “happens-before” relation
  – obvious inside a node
  – two nodes: send happens before receive
- Each node keeps local clock:
  – counts events (sending, receiving)
  – we also say: messages receive a timestamp at reception

Leads to partial order. Some events a,b are “concurrent” i.e., neither time(a)$<$time(b) nor time(b)$>$time(a)

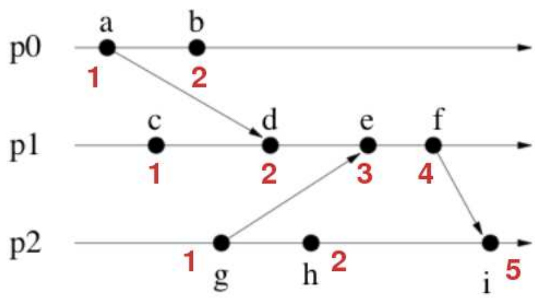
Cause and Effect, Independent or Concurrent



Logical Clocks

L(event): assign a number to an event

- We want: if $a \to b$, then $L(a) < L(b)$
- Logical clocks do this, but $L(a) < L(b)$ does not imply $a \to b$ (integers have total order, but happens-before is partial)



Problem when comparing g and b

Vector Clocks (Mattern, 1989)

- Each node keeps a vector of all logical clocks it learned, vector is timestamp
- On reception, use max of component values
- $<$ defined such that:
  All components equal or less, and then
  at least one component must be different.
- Incomparable values! They mean “concurrent”.
  See “causality” figure.

Overview

Selected “Self-” Algorithms (II)

- Notes on message passing models
- Self-stabilization (example: mutual exclusion)
- Logical clocks
- **Distributed snapshot**
- Byzantine agreement

Global State

No global time available (you can't reconstruct *exact* physical time over asynchronous channels), no accurate global state obtainable.

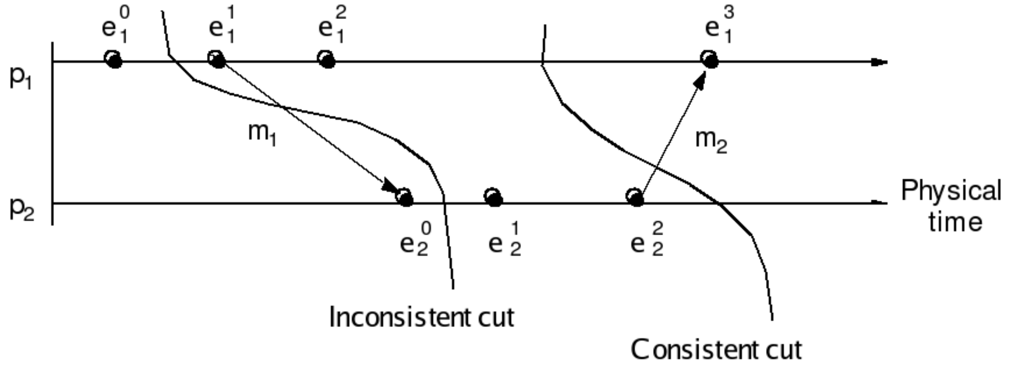
- Weaker form: global state =
  combination of local states of processes and channels
  at some time which could have occurred
- Why we want global state:
  – finding lost token
  – termination of distributed computation
  – garbage collection.

We assume a strongly connected network.

"Consistent State"

Intuitively: **consistent global state** =
"snapshot" that looks to all processes as if it was
taken at the same instant, everywhere.

- Causal relations must be observed, **consistent cut**



Consistent Cut

Global state = must be a consistent cut

Property:

- Either:
  all events before the cut have to “happened-before” the events after the cut,
- Or: the events are unrelated (“concurrent”)
- This can be checked with vector clocks.

The big problem are the messages in the channel, they are part of the state.

Application of cuts

Nodes can crash – how to restart in a safe state?

- Logging:
  Nodes periodically write their current state to disk
- In case of a node crash, after restart, the node knows its latest “cut”
- Other nodes in the system have to roll back
  to a maximally consistent cut.

Other use of cuts: take a (distributed) snapshot! Next slides . . .

Distributed Snapshot

Snapshot algorithms to record a consistent state

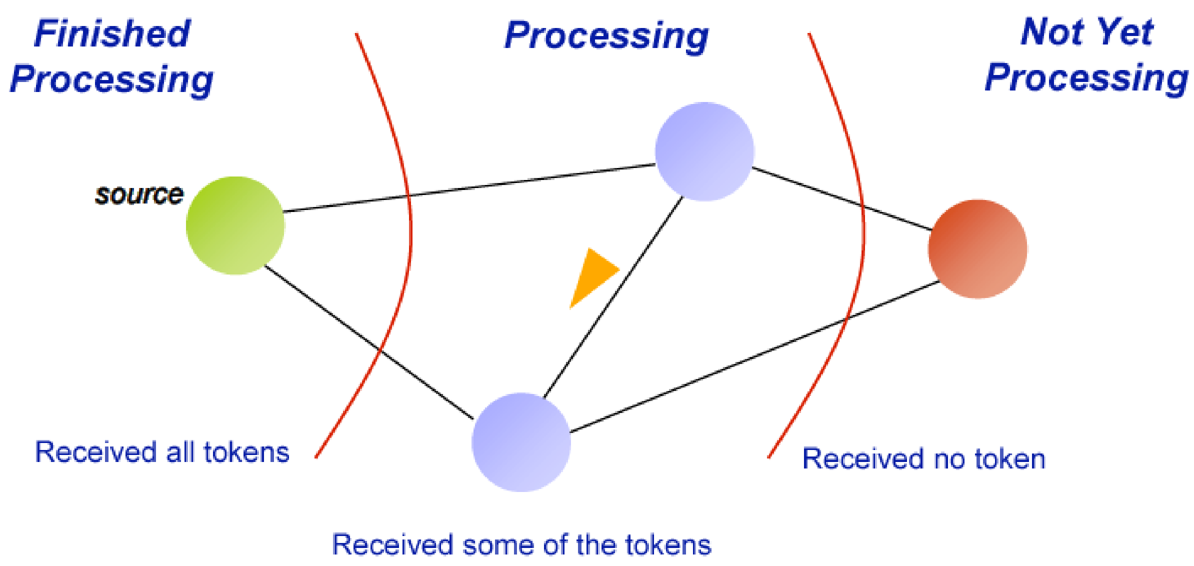
- Snapshots can be used to detect stable states.
  Examples of stable states: lost token, deadlock, termination.
- One well-know algo: Chandy and Lamport
  It creates one (possible) consistent cut.
- Assumptions for Chandy and Lamport:
  – reliable message exchange, FIFO, unidirectional
  – no failures (of nodes, links)
  – strongly connected graph

Chandy and Lamport

Idea: "drain" all messages pending in the channels,
put a marker before and after these messages.

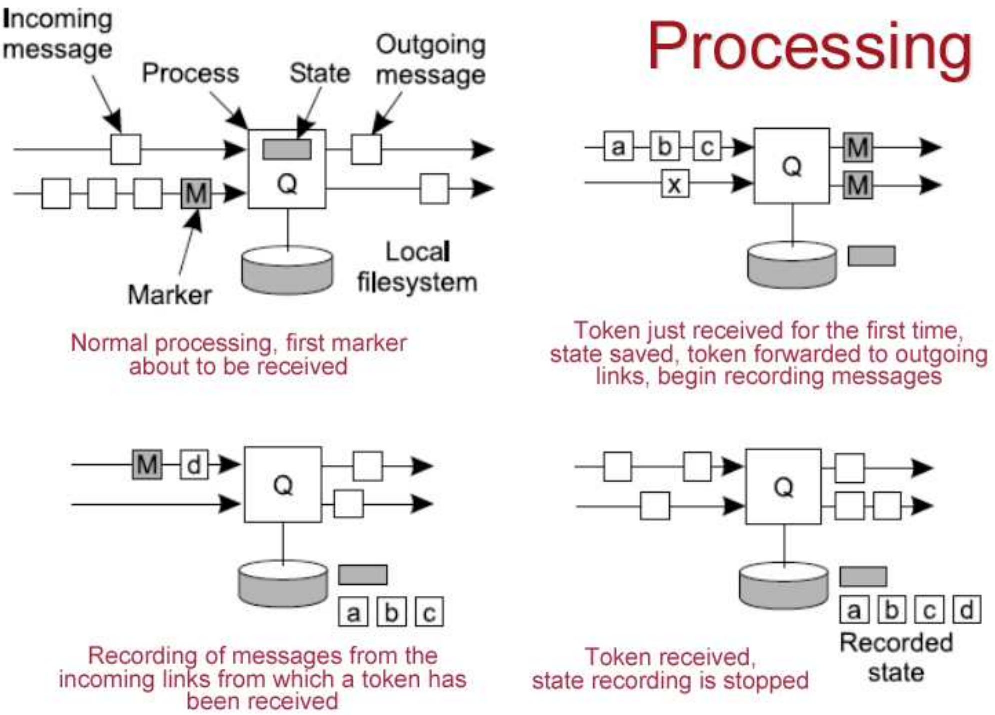
- Initiate snapshot by sending a marker M
- When receiving M
  – stop processing, record own state
  – send out marker on all links
  – record all subsequently received messages
    (except where M arrived)
  – until a next marker is received on each link.

Chandy and Lamport: Intuition



"Token wave" which flushes the channels (two frontiers)

Chandy and Lamport: Example at one Node



Overview

Context: consensus finding (=distributed agreement)

- Notes on message passing models
- Self-stabilization (example: mutual exclusion)
- Logical clocks
- Distributed snapshot
- **Byzantine agreement**
  – two general's problem
  – byzantine failures
  – byzantine problem, (impossibility) results
  – the oral message protocol

Byzantine Agreement and related topics

Presentation approach:

- Failure modes (including "byzantine failures")
- Problematic consensus in case of message loss ("two generals problem")
- Problematic consenus in a group, with reliable messages (byzantine agreement problem, $3f + 1$ requirement)
- Impossibility result for asynchronous settings (Fischer, Lynch and Peterson, 1982)
- Discussion: ways out of this problem

Failure Modes

Depending on the assumptions of failure, different algorithms (or impossibility results) will ensue.

- **Fail-stop**:
  – either a processor works correctly and participates
  – or it has failed and will never respond again.
  Moreover: the others can reliably detect failed processors.
- **Slowdown**: when some processors execute slowy (or fail), the others cannot know for sure.
- **Byzantine**: every action might be corrupt (see next slide)

Fail-stop is often assumed, but is not realistic (e.g. msg delay)

The Byzantine Failure Mode

Byzantine failures include everything:
– lost messages
– crashing nodes
– faulty implementation
– **malicious implementation, even collusion!**

In the following we make things simple:
– synchronous execution model
– reliable message exchange
– full connectivity and sender authentication
That is: “only” the processors behave byzantine, it’s problem enough

Prelude: “Two Generals’ Problem”

What if message passing is not reliable? Big trouble!

- Two generals have to coordinate an attack (“we attack at 4am”)
  – each must be absolutely sure that the other agrees
  – in case of doubt: none will attack
- Because of message loss, sender is uncertain
  – even with confirmation, this persists
  (the other thinks: did he receive my confirmation?)
- Result: impossible to get common agreement, for sure.

In practice: accept state with some probability
(after N rounds we are “confident”, instead of sure)

Byzantine Agreement Problem

The story:

- Romans still rule over Constantinople, but Ottoman battalions want to launch an attack.
- Only a coordinated attack will be successful, a coordinated retreat is also an option.
- The general decides to attack (or retreat):
  all lieutenants receive the order,
  lieutenants will exchange messages to verify the outcome.
- Some lieutenants (incl the general) are traitors, and pass on lies.

Is there a protocol for (successfully) coordination?

Also known as: **consensus algorithms, interactive consistency algorithms**

Byzantine Agreement Problem, CS version

- Input: Each process starts with a bit
- Goal: run a protocol such that
  – all processes output the same bit
  – the output bit must match at least one of the initial bits
- Q: how many faulty processes can you tolerate?

The problem was proposed in 1980
by Lamport, Shostak and Pease.

Byzantine Agreement: Desired Properties

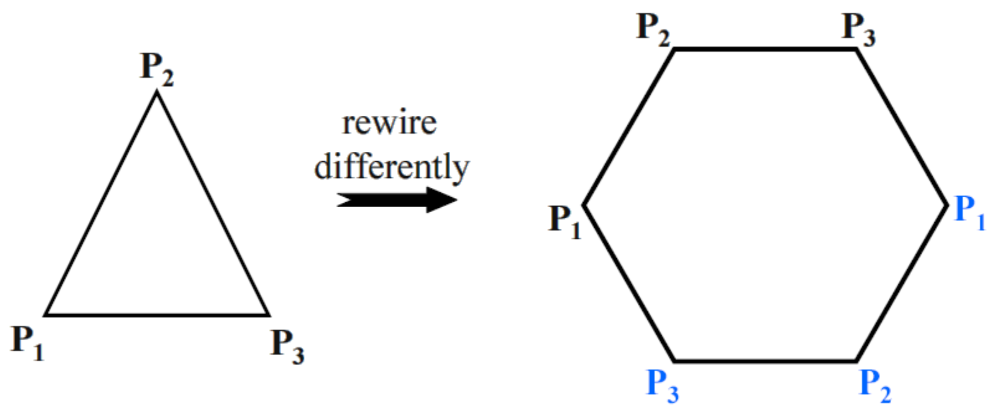
- Non-trivial protocol:
  if all lieutenants have same input b, the loyal lieutenants' agreement must be b.
- Core property:
  All loyal lieutenants agree on the same value
  (which must not be original b, but it must be the same for all)
- Protocol must terminate.

Byzantine Agreement: Impossibility (1)

Theorem: **There is no protocol for byzantine agreement among 3 nodes if at least one node fails.**

- Applied to story:
  with three lieutenants, already one traitor lieutenant will prevent successfull agreement
- Proof preparation:
  – we build a 6-node network of reliable processes
  – on this we **emulate** (malicious) failures
  – ie, some processes output (deliberate) wrong answers

Byzantine Agreement: Impossibility (2)

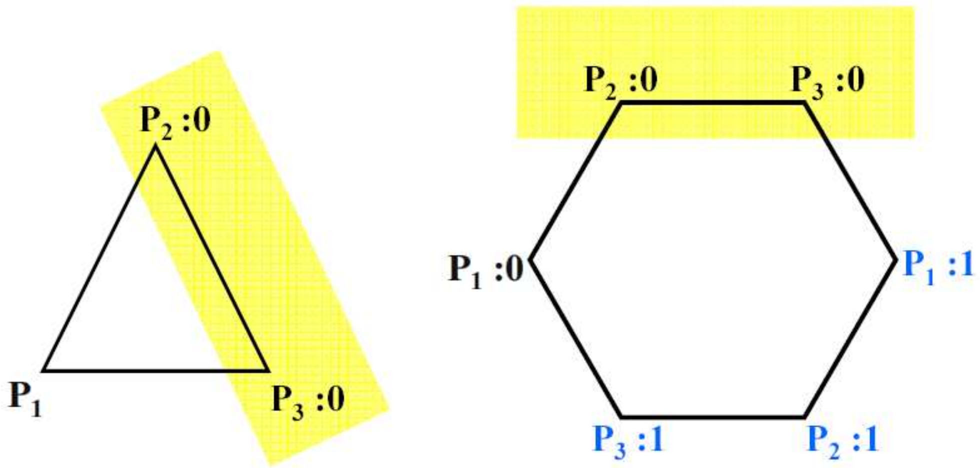


(Figs from G. Candea)

Broadcast (left) is represented as point-to-point (right),
the blue processes P run identical code as P.

Assumption: ∃ byz. agr. protocol for three nodes and one failure

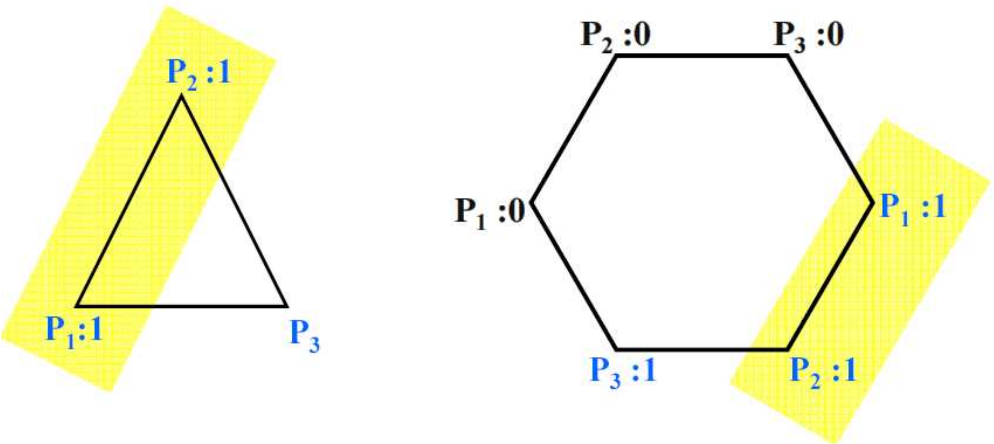
Byzantine Agreement: Impossibility (3)



Given: P2 and P3 have initial input 0
What happens if P1 is faulty? E.g. P1 sends different values!

P2 and P3 nevertheless agree on 0 (assumption!)

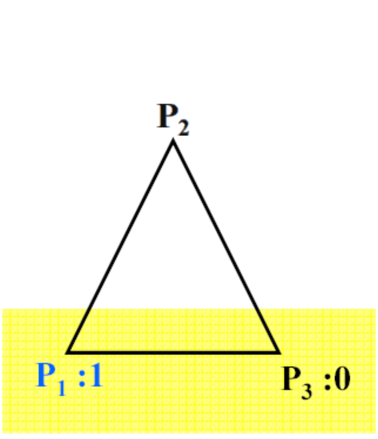
Byzantine Agreement: Impossibility (4)

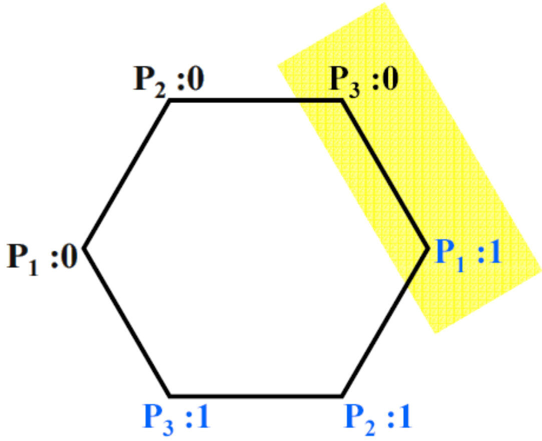


Given: P1 and P2 have initial input 1
What happens if P3 is faulty? E.g. P3 sends different values!

P1 and P2 nevertheless agree on 1 (assumption!)

Byzantine Agreement: Impossibility (5)



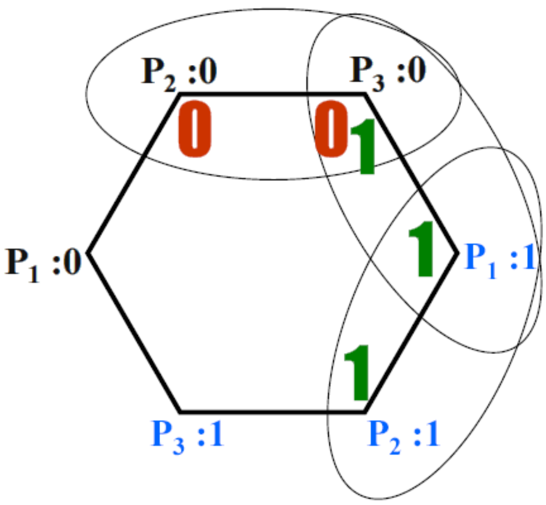


Given: P1 and P3 have initial input 1 and 0, respect.
What happens if P2 is faulty? E.g. P2 sends different values!

P1 and P3 nevertheless agree on some value (assumption!)

Byzantine Agreement: Impossibility (6)

Trying to harmonize the different views results in contradiction:



- P1 and P2 agreed on 1
- P1 and P3 agreed on some value; because P1 above went for 1, this must be 1 also for P3
- However: P3 already agreed on 0 (with P2)
- contradiction.

Byzantine Agreement: Impossibility (6)

Another argument, graphical this time:

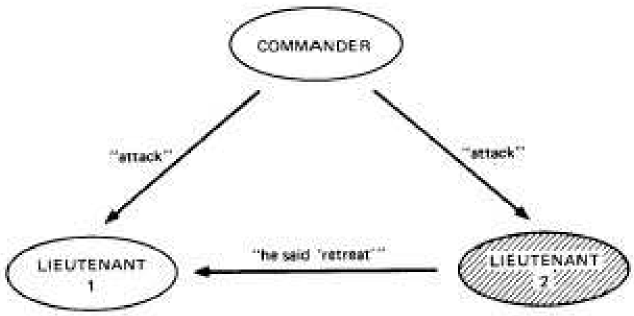


Fig. 1. Lieutenant 2 a traitor.

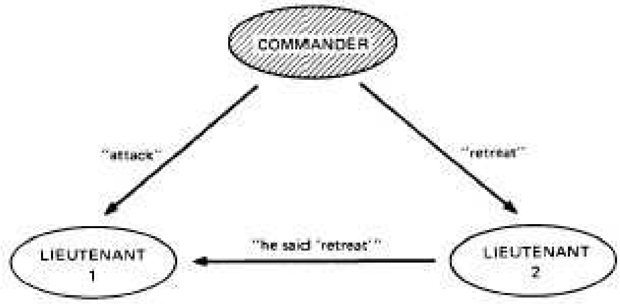


Fig. 2. The commander a traitor.

L1 does not know whom to obey: L1 cannot simply obey the commander because among the lieutenant(s), no consensus possible.

Byzantine Agreement, Theory Results

- Theory result:
  Protocols exist, if we have 3f+1 nodes for f failures
  (see subsequent slides on OM)
- **But:** Result is different in asynchronous model:
  No protocols at all! A single faulty node is the end!
  The "celebrated result" of Fischer, Lynch and Peterson in 1982)

In 2007, more than 1200 follow-up publications . . .

Oral Message Protocol (1)

Proposed by Lamport et al.

- Idea of OM:
  repeat majority vote recursively in smaller and smaller groups
- (another idea would be:
  lieutenants not only exchange their bits, but also what they heared from whom)

Also called an interactive consistency protocol.

In the following:
Algorithm OM(0) for last round, OM(m) for previous rounds

Oral Message Protocol (2)

OM(0) for terminating

- Commander sends his value to every lieutenant
- Each lieutenant uses the value received from commander

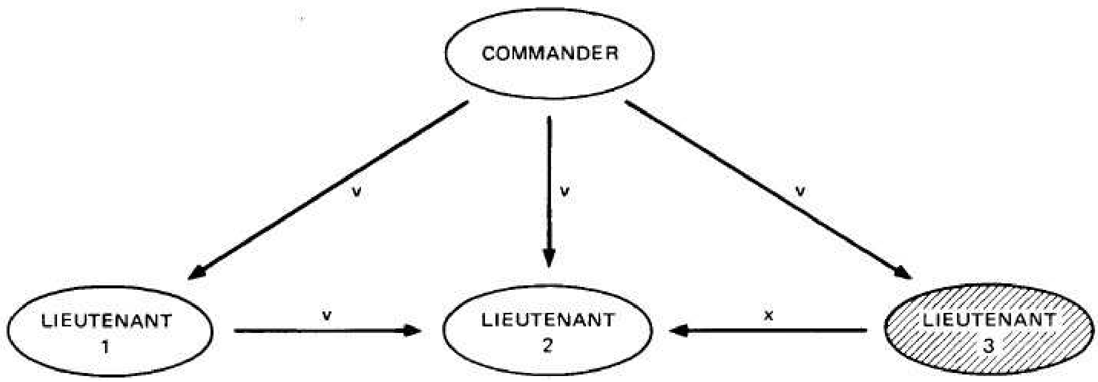
Note: OM(k) means "this protocol tolerates up to k traitors"

Oral Message Protocol (3)

OM(m), $m > 0$

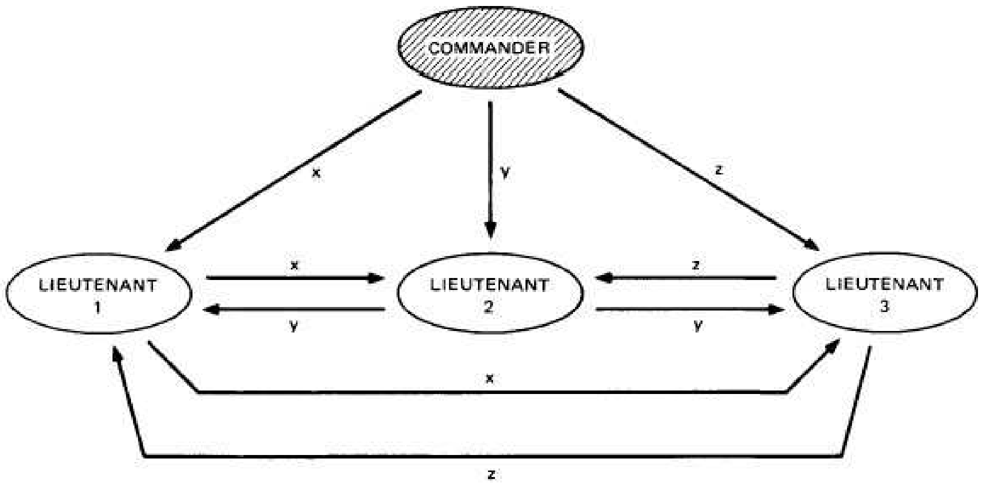
- Commander sends his value to every lieutenant (v_i)
- Each lieutenant acts as a commander for OM(m-1): sends v_i to the other $n-2$ lieutenants
- Lieutentant i receives v_j, and for $j <> i$, computes majority: this becomes his value for OM(m)

Example 1: One traitor lieutenant



OM(1): This time, a majority vote among the lieutenants works, as the loyal ones outnumber the malicious lieutenant.

Example 2: The commander is a traitor



OM(1) works even if the commander is not loyal.

Message complexity of the OM algorithm

Overall: $O(n^m)$ invocations (!),
where n is the number of processes, m number of faulty processes

- OM(m) invokes OM(m-1) $n-1$ times
- each OM(m-1) invokes OM(m-2) $n-2$ times
- etc

Protocol variations (ways out of impossibility result)

Approach: change some assumptions, or the rules/tricks

randomized instead of deterministic algorithm, or private channels, or signed messages, or failure detectors (instead of proceeding despite failures).

- Ben-Or (1983): exponential (in time) asynchronous byzantine agreement, works for $f < n/5$
- still open how much this can be improved