Impossibility of Distributed Consensus with one Faulty Process

M.J. Fischer, N.A. Lynch and M.S. Paterson, Journal of the Association for Computing Machinery, Vol. 32 No.2, April 1985, pp. 174-183.

Landmark paper which proves that a distributed system with failures cannot guarantee a 100% probability to reach a consensus
-> this has to be considered in real-world applications
-> explains the transaction commit problem in distributed database systems
-> every large application has its “window of vulnerability”

1. System setup

2. Definitions & vocabulary

3. Proof by contradiction
   I   an asynchronous system might reach an undecided state
   II from there you might reach another undecided state
   -> induction: the system might never reach a decided state

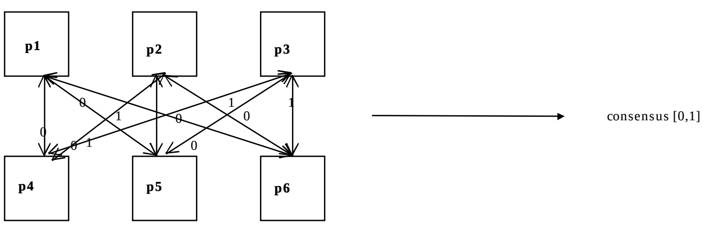
4. Summary

System Overview

Task
- processes shall agree on a binary value [0,1] in finite time
- depending on some system state, this binary value has to change (non-triviality)
- any consensus protocol might be applied

Rules
- the processes are completely asynchronous, running on distributed nodes
--> no assumptions about the relative speeds of the processors or the communication
--> no synchronized clocks and therefore no timeout
--> no method to identify a failed process

--> we cannot distinguish between a very slow or missing message



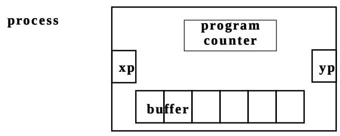
System Setup

"Benign" Setup
- the processors are modeled as automata which communicate by messages
- all messages among the nodes arrive at their target in finite time
- they might be out of order
- they cannot be reversed (e.g. "oops, changed my mind")

States & Decisions
- processes change state depending on arriving messages
- all system configurations can be reached from initial configurations
- the system decision is based on majority vote

System failures
- there might be only one faulty process at a time
- all non-faulty processes receive their messages eventually

--> not all processes have to participate in consensus



Vocabulary

p	== process
xp	== one-bit input register of p [ø, 0, 1]
yp	== one-bit output register of p [ø, 0, 1]
initial value	== [0,1]
internal state	== values in **xp** + **yp** + program counter + internal storage
initial state	== (**xp** = ?) + (**yp** = ø) + program counter + internal storage
decision state	== ((**yp** = 0) | (**yp** = 1))
transition function	== **xp** -> **yp** // deterministic
P	== consensus protocol of system with **N** processes (**N**>=2) + transition functions of all proceses + internal states of all xp
message	== (**p**,**m**) // **p**=destination process, **m**=[0,1]

message system	== single message buffer of not delivered messages
	+ operation send (**p**, **m**) // send message **m** to **p**
	+ operation receive (**p**) // read **m**, then del **m** from buffer

Vocabulary

C == configuration == internal state of all processes
+ message buffer content

initial configuration == initial state for all **p**
+ message buffer empty

atomic step == takes one configuration to another // deterministic

phase 1: process attempts to receive a message (or null ø)

phase 2: local computation, if a message was received
(internal state + **m** + transition fkt -> new internal state)

phase 3: send finite set of **m** to any number of other processes
in one step (== atomic broadcast)

--> all non faulty processes will receive message at some point in time
--> messages might be out of order

Vocabulary II

e == event == (**p**, **m**) // (**p**,ø) always possible
e (**C**) == **e** can be applied to **C**, yielding a new configuration

s == schedule == (in)finite sequence of events starting with **C**
run == sequence of steps in a schedule

reachable == if **s** is finite
+ and **s** (**C**) is resulting configuration
accessible == **C** reachable from initial configuration

Vocabulary III

v == decision value == process **p** is in decision state with **yp** = **v**

partially correct == a consensus protocol **P** satisfies 2 conditions
I. Every accessible **C** has exactly one **v**
II. For each **v** [0,1] some accessible **C** has decision value **v**

non-faulty == a process is in a run off any length, even infinite
faulty == otherwise (e.g. blocking)
admissible run == at most one process is faulty
+ message to all other non-faulty p are delivered
deciding run == some, not all, processes reach a decision state in that run

totally correct == a consensus protocol **P** is totally correct, if
+ **P** is partially correct
+ every admissible run is decided

Vocabulary IV

bivalent **C** == **v** element of |**V**| = 2 // no clear outcome
univalent **C** == **v** element of |**V**| = 1 // clear outcome

0-valent (**v** = 0) == **v** always 0 // decided, no change in **v**
1-valent (**v** = 1) == **v** always 1 // decided, no change in **v**

adjacent == 2 initial configurations differ only in one **xp**
neighbours == 2 configurations differ only in one single step

Lemma 1: Commutativity

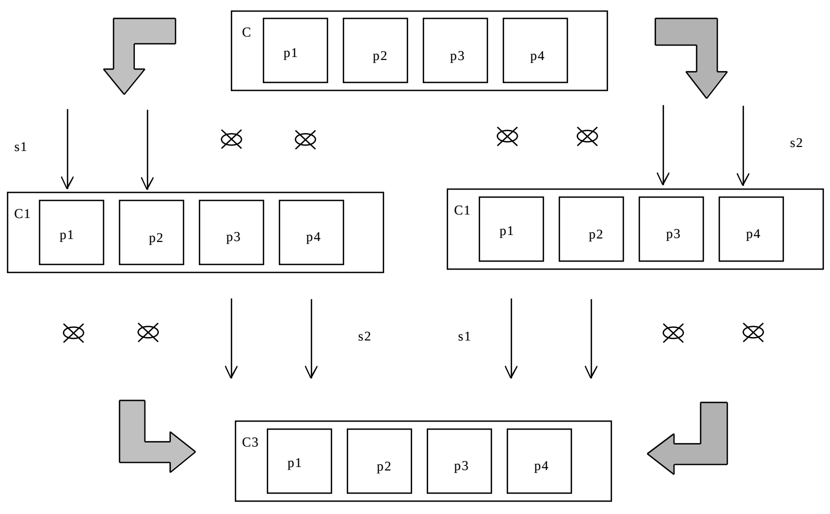
Suppose that from some configuration C, the schedules s1, s2 lead to configurations C1, C2 respectively. If the sets of processes taking steps in s1 and s2, respectively, are disjoint, then s2 can be applied to C1 and s1 can be applied to C2, and both lead to the same configuration C3.

Proof: s1 and s2 do not interact.

- schedules s1 and s2 are a (in-)finite sequence of events, that can be applied to C
- the associated sequence of steps is called a run
- s1 and s2 are fixed, independent if they are applied to C or C1/C2 - given a (p,m) pair, the transition function is deterministic
- time delays are not considered in this system
- according to the system setup each set of processes executes its runs independent

--> after each system-part has executed its independent run, the resulting configuration is the same

Lemma 1: flow graph



Lemma 2: there exist undecided states

P has a bivalent initial configuration

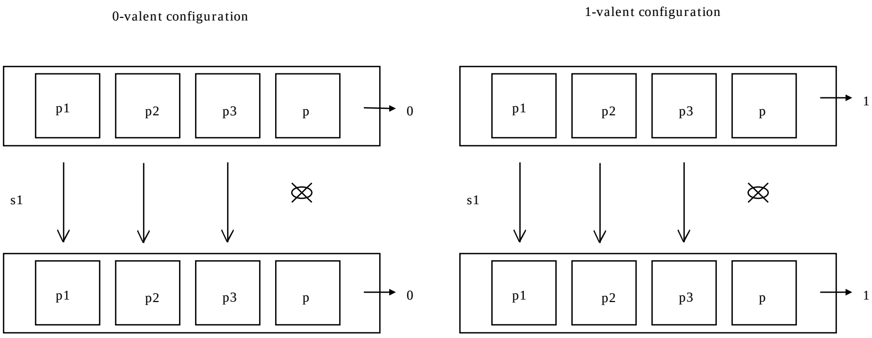
Proof: Assume not.
- P is by definition partially correct
-> therefore P must have both 0-valent and 1-valent initial configurations
-> any two adjacent configurations are joined by a chain of initial conf. (no steps)
-> there must exist a 0-valent initial configuration C0 adjacent to a 1-valent C1

Consider some admissible deciding runs from C, where
- p is the only difference between the adjacent configurations C0 and C1
- p takes no steps (blocks)
-> then **s** can be applied also to C0 and to C1, and reach the same decision value

--> if the decision value is 1, then C0 has to be bivalent :: contradiction.
--> if the decision value is 0, then C1 has to be bivalent :: contradiction.

--> as we cannot tell if a process has died or is just slow, we have to assume all processes participate in the consensus
--> even one faulty process will delay the algorithm is delayed

Lemma 2: flow graph



Lemma 3: any bivalent configuration might lead to another bivalent configuration

Let C be a bivalent configuration of P, and let e =(p,m) be an event that is applicable to C. Let CS be the set of configurations reachable from C without applying e, and let
DS = e(CS) = {e(E)| E element of CS and e is applicable to E}.
Then, DS contains a bivalent configuration.

Proof:
- since e is applicable to C, then by definition of CS and the fact that messages can be delayed arbitrarily, e is applicable to every E elem CS.

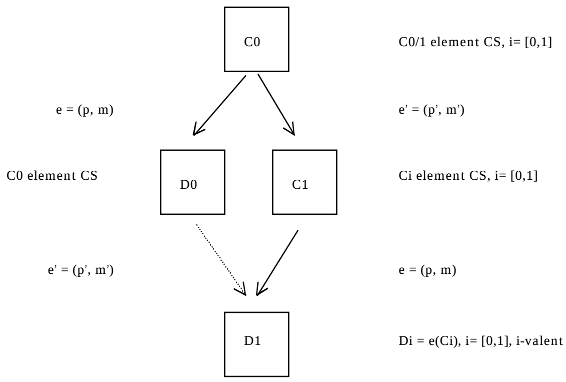
Assume that DS contains no bivalent configurations
- Ei is an i-valent configuration reachable from C, i=[0,1]
- if Ei elem CS, then let Fi = e(Ei) elem DS
- otherwise, e was applied in reaching Ei, so that there exists Fi elem DS from which Ei is reachable

--> in either case, Fi is i-valent (since Fi is elem DS, which shall contain no biv. C)
--> one of Ei and Fi is reachable from the other
--> since Fi elem DS, i =[0,1], DS contains both 0-valent and 1-valent configurations

Lemma 3: flow graph

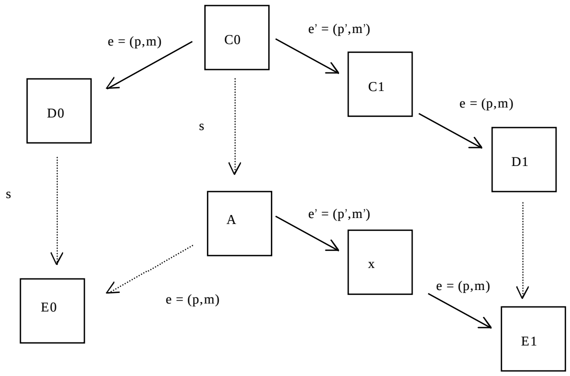
continue:
-> there exist neighbours C0, C1 elem CS such that Di = e(Ci) is i-valent, i=[0,1]
-> C1 = e' (C0), where e' = (p', m') (e' some other e) ; then

Case 1: (p' != p), then D1 = e' (D0) by Lemma 1 --> contradiction since D0 is 0-valent



Lemma 3: flow graph II

continue: Case 2: if (p' = p), then
let there be a deciding run from C0 in finite steps in which p blocks;
let s be the corresponding schedule and A = s (C0)
-> (Lemma 1) **s** is applicable to Di, and it leads to an i-valent configuration Ei=s(Di)
-> (Lemma 1) e (A) = E0 and e(e' (A))
--> A is bivalent --> contradiction --> DS contains a bivalent configuration



Main Result: asynchronous system are not fault-tolerant

No consensus protocol is totally correct in spite of one fault

Proof:
- any deciding run from a bivalent initial C must go to a univalent C
- it is always possible to avoid a decisive step

Setup:
- all processes check for their messages in sequence
- let C0 be a bivalent initial configuration (Lemma 2 ensures that it exists)
- let C be a later, bivalent configuration
- p is next process to check for a message, m is the message received

--> there is a configuration C' reachable from C by a schedule with e being the last event
--> we have reached again a bivalent configuration, without performing non-permissible steps
--> no decision is ever reached

--> P is not totally correct

Summary

Lemma 2: there exist initial states for which the final decision is undecided

Lemma 3: starting at any undecided state can lead to another undecided state

Main Theorem: it is possible to construct an asynchronous system, that, starting out from an undecided state, will stay for ever undecided. Even with only 1 faulty process, a consensus on a binary value cannot be decided in finite time

- even if we consider only fair runs (all processes receive their messages, etc.), blocking processes could halt any asynchronous system

- fault tolerance in asynchronous systems requires making assumptions about the system or about the kinds of faults which can be handled

- in real systems this is usually done by
1) assuming an upper bound in communication and processor speed,
2) considering a process faultly if it doesn't respond within a certain time