Tasks — IIS

Solving a task in the iterated immediate snapshot model

Topological representation
Topological representation: one shot Immediate snapshot

$p_2$ sees \{p_2\}

$p_1$ sees \{p_1, p_2\}

$p_3$ sees \{p_2, p_3\}

$p_3$ sees \{p_1, p_2, p_3\}

$p_2$ sees \{p_1, p_2\}

$p_2$ sees \{p_2, p_3\}

$p_3$ sees \{p_2, p_3\}

$p_1$ sees \{p_1\}

$p_3$ sees \{p_3\}
Iterated standard chromatic subdivision (ISDS)
ISDS: one round of IIS
two rounds of IIS
to solve task: IIS is equivalent to read-write
Consensus generalized: k-Set Agreement

A process *proposes* an *input* value in $V$ ($|V| \geq k+1$) and tries to *decide* on an *output* value in $V$

- **k-Agreement**: At most $k$ distinct values are decided
- **Validity**: Every decided value is a proposed value
- **Termination**: No process takes infinitely many steps without deciding
  (Every *correct* process decides)

1-set agreement = consensus
(N-1)-set agreement = set agreement
Impossibility of Set Agreement

**Theorem 2** No *wait-free* algorithm solves set agreement in read-write model

- Starting with N values, there is no way to drop one (decide on at most N-1)

- Implies the impossibility of wait-free k-set agreement for all k≤N-1
Solving Set Agreement in IIS

• Each $p_i$ proposes its id $i$
• There is an IIS round $r$, such that each process (that reaches round $r$) outputs some id
• In each run, the set of output ids is a subset of participants of size $\leq N-1$

Implies Sperner coloring of the subdivided simplex $\text{IIS}_r$
« Chromatic » Subdivision

Simplex A

B Subdivision de A

From IIS
Sperner Coloring

“Corners” have distinct colors

Edge vertexes have corner colors

Every vertex has face boundary colors

Sperner Lemma: In any Sperner coloring, at least one $n$-simplex has all $n+1$ colors

Assume there is a wait-free protocol solving set agreement

after $r$ rounds IIS we get a subdivision of $S^{n-1}$ each node is coloured by its decision, we get a Sperner coloring $\Rightarrow$ N different values are decided
Sperner’s coloring of a subdivided simplex

IIS:
Each vertex of the original simplex $S^{N-1}$ (initial state) is colored with a process id

Sperner’s coloring:
Each vertex $v$ in the subdivision $D$ is colored with a distinct color:
- If the vertex belongs to a face $S$ of $S^{N-1}$, then $d$ is in $\text{colors}(S)$
Sperner’s Lemma

Sperner’s coloring of any subdivision $D$ of $S^{N-1}$ has a simplex with all $N$ colors

IIS and decision on set agreement agreement defines a Sperner’s coloring. Then at least one execution with $N$ decided values!

**Corollary:** wait-free set agreement is impossible in IIS (and, thus, in read-write)
In at least one IIS run, N values are decided.

Here: suppose processes decide in one round.
Thus

✶ No algorithm can wait-free solve (N-1)-set agreement in IIS
  • otherwise Sperner’s coloring of some $IIS_r$ would have no fully colored simplexes (but there is at least one!)

✶ No algorithm can wait-free solve set agreement in AS (or any read-write model):
  • otherwise we can (non-blocking) simulate it in IIS and thus find a wait-free algorithm in IIS

✶ We cannot tolerate N-1 failures: can we tolerate less?
  • E.g., can we solve consensus (1-set agreement) 1-resiliently?
So...

- No algorithm can wait-free ((N-1)—resiliently) solves consensus (proved!)

- We cannot tolerate N-1 failures: can we tolerate less?
  - can we solve consensus 1-resiliently?
1-resilient consensus?

What if we have 1000000 processes and one of them can crash?

NO

We present a direct proof now
(an indirect proof by reduction to the wait-free impossibility also exists)
Impossibility of 1-resilient consensus [FLP85, LA87]

Theorem 2 No 1-resilient (assuming that one process might fail) algorithm solves consensus in read-write

Proof
By contradiction, suppose that an algorithm A solves 1-resilient binary consensus among $p_0, \ldots, p_{N-1}$
A configuration (or state) = state of each process + state of shared memory

Initial configuration = one input value per process + initial state of processes and shared memory

A step = atomic step (read or write on shared memory) of a process

A run of A is a sequence of steps applied to the initial state

A schedule is a (infinite) sequence of process ids \(i_1, i_2, \ldots\)

A run of A can be seen as an initial input configuration (one input per process) and a schedule (or an initial sequence of a schedule)

Every correct (taking sufficiently many steps) process decides!

(t-resiliency means that in every run at most \(t\) processes take a finite number of steps)
Proof: valence

Let $R$ be a finite run

- We say that $R$ is \textit{v-valent} (for $v$ in $\{0,1\}$) if $v$ is decided in every infinite extension of $R$

- We say that $R$ is \textit{bivalent} if $R$ is neither 0-valent nor 1-valent
  (there exists a 0-valent extension of $R$ and a 1-valent extension of $R$)

- We say $R$ is \textit{univalent} if $R$ is not bivalent
Proof: valence claims

Claim 1 Every finite run is 0-valent, or 1-valent, or bivalent. (by Termination)

Claim 2 Any run in which some process decides $v$ is $v$-valent (by Agreement)

Corollary 1: No process can decide in a bivalent run (by Agreement).
Bivalent input

Claim 3 There exists a bivalent input configuration (empty run)

Proof
Suppose not:
Consider sequence of input configurations $C_0, \ldots, C_N$:
in $C_i$: $p_0, \ldots, p_{i-1}$ propose 1, and $p_i, \ldots, p_N$ propose 0

* All $C_i$'s are univalent (hypothesis)
* $C_0$ is 0-valent (by Validity)
* $C_N$ is 1-valent (by Validity)
Bivalent input

There exists $i$ in $\{0, \ldots, N-1\}$ such that $C_i$ is 0-valent and $C_{i+1}$ is 1-valent! $C_i$ and $C_{i+1}$ differ only in the input value of $p_i$ (it proposes 0 in $C_i$ and 1 in $C_{i+1}$).

Consider a run $R$ starting from $C_i$ in which $p_i$ takes no steps (crashes initially): eventually all other processes decide 0.

Consider $R'$ exactly like $R$ except that it starts from $C_{i+1}$, $R$ and $R'$ are indistinguishable ($p_i$ takes no steps).

- Thus, every process decides 0 in $R'$ --- contradiction ($C_{i+1}$ is 1-valent).
Critical run

**Claim 4** There exists a finite run $R$ and two processes $p_i$ and $p_j$ such that $R.i$ is 0-valent and $R.j.i$ is 1-valent (or vice versa)

(R is called **critical**)

**Proof of Claim 4:** By construction,
First take the bivalent empty run $C$
(by Claim 3 it exists)

Then construct an ever-extending fair (giving each process enough steps) critical run
Proof of Claim 4: critical run

repeat forever
  take the next process p_i (in round-robin fashion)
  if for some R’, an extension of R, R’.i is bivalent then R:=R’.i
  else stop

* If never stops – ever extending (infinite) bivalent runs in which every process is correct (takes infinitely many steps) – contradiction with termination

* If stops – (suppose R.i is 0-valent) – R is bivalent then consider a 1-valent R’ extension of R
  • There is a critical configuration between R and R’
Proof (contd.): the next steps in R

(Let $p_0=i$ and $p_1=j$)

Four cases, depending on the next steps of $p_0$ and $p_1$ in R

* $p_0$ and $p_1$ are about to access different objects (registers) in R
* $p_1$ reads $X$ and $p_0$ reads $X$
* $p_0$ writes in $X$ and $p_1$ reads $X$
Proof (contd.): cases and contradiction

* $p_0$ and $p_1$ are about to access different objects in $R$
  - $R.0.1$ and $R.1.0$ are indistinguishable
Proof (contd.): cases and contradiction

* $p_0$ and $p_1$ are about to read the same object $X$

$R.0.1$ and $R.1.0$ are indistinguishable
Proof (contd.): cases and contradiction

\* \( p_0 \) is about to write to X
  
  • Extensions of R.0 and R.1.0 are indistinguishable for all except \( p_1 \) (assuming \( p_1 \) takes no more steps)
Proof (contd.): cases and contradiction

ispiel: $p_0$ is about to read to $X$
• Extensions of R.0.1 and R.1.0 are indistinguishable for all but $p_0$ (assuming $p_0$ takes no more steps)
Thus

- No critical run exists
- A contradiction with **Claim 4**

→ 1-resilient consensus is impossible in read-write
and message passing?

message passing:

• send/receive messages with asynchronous communication (each message sent by a process is eventually received by a correct process)
• process may crash (stop its execution)
• process $p$ is correct if it doesn’t crash (makes an infinity of steps)
• $t$-resiliency at most $t$ processes may crash
Message Passing versus Shared memory

simulating message passing: n*n registers
The channel between p and q is simulated by one register written by p read by q
Message Passing versus Shared memory

simulating shared memory: with a majority of correct processes atomic registers may be implemented in asynchronous message passing models.

(Message-Passing and shared memory models are « equivalent » -with a majority of correct processes)
Simulating shared registers in message passing

Simulation of a single-writer single-reader register

• assume a majority of correct processes
SWSR atomique

For the writer
To write(v)
  seq:=seq+1
  send(W,v,seq) to all
  wait until receiving \([n/2] + 1\) messages (ACK,seq)

For the reader
To read()
  send(R) to all
  wait until receiving \([n/2] + 1\) messages (V,v,s) such that \(s \geq seq\)
  let S be the max of the sequence number of received messages
  seq:=S
  let val such that (V,val,S) has been received
  return(val)

For all processes
  when (W,v,s) is received from the writer
    If \(s > seq\) then val:=v; s:=seq
    send(ACK,s) to the writer

  When (R) is received from the reader
  send (V,val,seq) to the reader
A majority of correct processes is needed

partition argument:

• if $n \leq 2t$ then we can partition the set of processes in two set $S_1$ and $S_2$ such that $|S_1| \geq t$ and $|S_2| \geq t$.

  • Run $A_1$: all processes in $S_1$ are correct and all processes in $S_2$ are initially dead, $p_0$ invokes a write(1), at some time $t_0$ the write terminates
  • Run $A_2$: all processes in $S_2$ are correct and all processes in $S_1$ are initially dead, $p_1$ $S_2$ invokes a read(), at time $t_0 + 1$ the read terminates at time $t_1$
  • Run $B$: « merge » of $A_1$ and $A_2$ but no process crash. write(1) terminates before the read() and the read return 0

contradiction
Consensus in message passing

Same definition as in shared memory:

- Decision algorithm with:
  - Agreement (no two correct processes decide on different values)
  - Termination (all correct processes terminate)
  - Validity (the decision is an initial value).

There is no 1-resilient consensus algorithm in message passing.

- Shared memory enables to simulate message passing
Reliable Broadcast

primitives $\text{Rbcast}$ $\text{Rdeliver}$

- **Agreement:** If correct process $p$ $\text{Rdeliver} m$ then every correct process $\text{Rdeliver} m$

- **Validity:** If correct process $p$ $\text{Rbcast} m$ then $p$ $\text{Rdeliver} m$

- **Integrity:** If $p$ $\text{Rdeliver} m$ then there is a process $q$ that has $\text{Rbcast} m$
Algorithm for process $p$:

To execute $\text{Rbcast}(m)$
  send $(m)$ to $p$

$\text{Rdeliver}(m)$ occurs when
  \begin{verbatim}
  upon receive$(m)$ do
    if has not previously executed $\text{Rdeliver}(m)$
    then
      send $(m)$ to all
      $\text{Rdeliver}(m)$
  \end{verbatim}
ABcast

primitives ABcast ABdeliver:
  • RBcast properties:
  • Total order: If p and q ABdeliver m and m' then if p ABdelivers m before m' then q ABdelivers m before m'

ABCast is « universal »: (very informal) (active replication)
  • state machine replication:
    - any sequential state machine A
    - t+1 processes simulate A
    - each request is made by atomic broadcast
    - then we get a t-resilient implementation of A
Active and passive replication

Active Replication

- Client
- Process
  - State
  - Server

Passive Replication

- Client
- Process
  - State
  - Server

Active Replication (SM)

- Client
  - Sends op to all replicas
- Replica
  - Requests get ordered
- Execution
- All reply to client

Passive Replication

- Client
  - Sends op to primary
- Primary
  - Primary forwards changes
  - Primary replies to client
- Backup
  - Only primary executes
Algorithm for process $p$:

**Initialization:**

$R_{Delivered} := \emptyset$

$A_{Delivered} := \emptyset$

To execute $Abcast(m)$

$R_{broadcast}(m)$

$A_{deliver}(\_)$ occurs when

upon $R_{deliver}(m)$ do

$R_{Delivered} := R_{Delivered} \cup \{m\}$

do forever

$A_{Undelivered} := R_{Delivered} \setminus A_{Delivered}$

if $A_{Undelivered} \neq \emptyset$ then

$k := k + 1$

propose($k, A_{Undelivered}$)

wait for decide($k, msgSet$)

$batch(k) := msgSet \setminus A_{Delivered}$

A-deliver all messages in $batch(k)$ in some deterministic order

$A_{Delivered} := A_{Delivered} \cup batch(k)$
Then

Consensus (and reliable broadcast) enable Atomic broadcast

Atomic Broadcast enable consensus Consensus?

Atomic broadcast and consensus are equivalent in message passing with crash failure

Then Consensus is « universal » in message passing with crash failure
Consensus... circumvent Impossibility
Circumventing consensus impossibility

« Strengthening » the model

- communication primitive: using more « powerful » objects than read-write register
- synchrony assumptions: round model, partial synchrony

Relaxing properties

- safety (agreement with probability 1, epsilon agreement),
- liveness (safe agreement, obstruction free, termination with probability 1)
Leader election **oracle** Ω

At every process and each time Ω outputs a process identifier in leader\_i.

Eventually, the same correct process is output at every correct process

- There is a correct process q and a time after which all correct processes p\_i have leader\_i=q

**Ω** failure detector

(May be done in a system where after some time, there exists k, there exists a correct process that takes at least one step each k steps of every process: limited asynchrony)
Shared variables

A infinite array of atomic snapshot object init ⊥
decide init ⊥

Upon propose(v) by process pi

prop:=v
for r:=0 to ∞
    wait until leader=pi or decide ≠ ⊥
    A[2r].update(prop)
    U:= A[2r].snapshot()
    if all values non ⊥ in U are equal to prop
        then report:=prop
        else report:=?
    A[2r+1].update(report)
    V:= A[2r+1].snapshot()
Let L be the non ⊥ in V
if |L| = 1 then
    if L = {1} then decide:=1
    if L = {0} then decide:=0
else
    if L = {1, ?} then prop:=1
    if L = {0, ?} then prop:=0
if decide ≠ ⊥ then return decide
Strengthening synchrony assumptions:

- Binary consensus
- Leader election (failure detector $\Omega$)
  - Each process have a local variable leader
  - There is a correct process $q$ and a time after which all correct processes $p_i$ have $\text{leader}_i = q$
- May be done in a system where after some time, there exists $k$, there exists a correct process that takes at least one step each $k$ steps of every process
Strengthening synchrony assumptions:

- Synchronous system, partially synchronous system
- **Safety**: Agreement+ Validity
- **Liveness**: Termination
Failure Detector Chandra & Toueg

- PODC 91 - J. ACM 96
- Distributed oracles that give hints on the failure pattern (e.g. set of suspected processes)
- A FD is basically defined by:
  - a completeness property: actual detection of failure
  - an accuracy property: restrict the mistake that a FD can make
Failures pattern

- failures and failure pattern:
  - at time $t$: $p$ is faulty (dead, crashed: $p$ stops making steps) or is alive ($p$ eventually makes steps)
  - *failure pattern* (schéma de pannes): $F(t)$ is the set of faulty processes at time $t$; $(F(t) \subseteq F(t+1))$
    
    $p$ is faulty in $F$: $\exists \ t \ p \in F(t)$
    
    $p$ is correct in $F$: $p$ is not faulty: $\forall t \ p \notin F(t)$
another schedule for the same failure pattern △
Failure detector[CT96]

- **failure detector**: oracle that may be invoked locally by processes depending only on failure pattern (not on the schedule!). At each step a process may invoke its failure detector module and gets an answer.

- depending only on the failure pattern: failure detector $\mathcal{X}$ is defined for each failure pattern $F$ by « histories » $H$

  - $H$ history of $\mathcal{X}$ for the failure pattern $F$: $H(p,t)$ is the output of $\mathcal{X}$ for process $p$ at time $t$ (if $p$ invokes failure detector $\mathcal{X}$ at time $t$, $p$ gets the answer $H(p,t)$)
Failure detectors

H(p,t) the output for process p at time t for history H

e**example:** H(p,t) a list of suspected processes

these histories may ensure properties:

- **completeness:** after some time H(p,t) contains forever all faulty processes
- **accuracy:**
  - **strong:** no process is suspected before it crashes
  - **eventual strong:** after some time strong accuracy
  - **weak:** at least one correct process is never suspected
  - **eventual weak:** after some time weak accuracy
Failure detectors

Examples of failure detectors with lists of suspected processes:

- $\mathcal{P}$ perfect FD: completeness + strong accuracy
- $\Diamond \mathcal{P}$: completeness + eventual strong accuracy
- $\Diamond \mathcal{S}$: completeness + eventual weak accuracy
- $\Omega$: outputs at each process one id of process (the expected leader):
  eventually $\Omega(p,t)$ is forever the same correct process for all processes (leader election)

$\mathcal{P}$ enables the consensus (synchronous rounds)

$\Diamond \mathcal{P}$, $\Diamond \mathcal{S}$, $\Omega$ enable the consensus (in shared memory or with a majority of correct processes in message passing)
Comparing failure detectors

Remarks:

• environment: set of failure pattern e.g. less than m faulty processes (here implicitly any number of faulty processes)

• shared memory versus message passing: depending on environments, solvability is not (exactly) the same for message passing and shared memory
Comparing failure detectors

- Problem $P$ is solvable with FD $\mathcal{X}$ iff there is a distributed algorithm using FD $\mathcal{X}$ that solves $P$

- Implementing a FD $\mathcal{X}$ may be (with more formal definitions) considered as a problem

$P \leq \mathcal{X}$

$\mathcal{X} \leq \mathcal{Y}$
\( x \preceq y \)

- Algorithm \( A \) using FD \( Y \)
  - Transforming \( Y \) in \( X \)

- Algorithm \( A \) using FD \( X \)

\( P \preceq X \)

- Algorithm \( A \) using FD \( X \)

\( P \preceq Y \)

- Algorithm \( A' \) using FD \( Y \)
Comparing failure detectors

$\mathcal{A} \preceq \mathcal{B}$ ($\mathcal{A}$ is weaker than $\mathcal{B}$) : there is a (distributed) algorithm with failure detector $\mathcal{B}$ that implements $\mathcal{A}$.

- $\mathcal{B}$ provides at least as much information about failures than $\mathcal{A}$
- (intuitively: less information about failures in $\mathcal{A}$ than in $\mathcal{B}$)

$\mathcal{A} \prec \mathcal{B}$ ($\mathcal{A}$ is strictly weaker than $\mathcal{B}$) if $\mathcal{A} \preceq \mathcal{B}$ and not $\mathcal{B} \preceq \mathcal{A}$

if $\mathcal{A} \preceq \mathcal{B}$ and $\mathcal{B} \preceq \mathcal{C}$ then $\mathcal{A} \preceq \mathcal{C}$ and (with more formal definitions)

$\mathcal{A} \preceq \mathcal{A}$

- $\preceq$ is a pre-order.
Weakest failure detector

As $\preceq$ is a pre-order, failure detectors can be compared and given some set $S$ of failure detectors $M$ is the weakest for $S$ iff for all $X \in S$ $M \preceq X$

(from any $X$, $X$ contains at least as much information about failures than $M$)

given a « problem » $P$, failure detector $W \mathcal{K}$ is a weakest failure detector for $P$ iff

1. $W \mathcal{K}$ can be used to solve $P$

2. for all $X$ that can be used to solve $P$: $W \mathcal{K} \preceq X$

$W \mathcal{K}$ encapsulates the minimum information about failures to solve problem $P$
Comparing problems with failure detectors

Given a «problem» $P$ (solvable by a FD) there is a weakest failure detector to solve $P$ [JT2008]

from the hierarchy of FD to hierarchy of problems:

- Given problems $P$ and $Q$: $P \preceq_{FD} Q$ iff $Q$ solvable with some FD $\mathcal{X}$ then $P$ is also solvable with $\mathcal{X}$

- $P \preceq_{FD} Q$ iff the WFD($P$) $\preceq$ WFD($Q$)
Example P and $\mathcal{S}$

- **P**: perfect (each crashed process will be suspected, no correct process is suspected)
- **$\mathcal{S}$**: strong (each crashed process will be suspected, at least one correct process is never suspected)
Example: ◇ 𝓢 and Ω

- Ω: outputs at each process one id of process (the expected leader):
  eventually Ω(p,t) is forever the same correct process for all processes (leader election)

- ◇ 𝓢: completeness + eventual weak accuracy

Ω ≥ ⟨⟩S: clear
⟨⟩S ≥ Ω:
  each process p:V[p]: array of size n
  (code of p)
  each k steps:
  Q=query the failure detector ⟨⟩S
  for all processes q :if q in Q then V[p][q] is incremented
  compute V[p][q] = max V[x][q]
  output of Ω = q such that V[p][q] is the smallest element of V[p][*]
Majority and $\Sigma$

With a majority of correct prowess:

Output = $\Pi$ (set of all processes)

repeat forever

send (ARE_YOU_ALIVE, $r$) to all

wait until receive (I_AM_ALIVE, $r$) from a majority $\Sigma = \{q | \text{a message (I_AM_ALIVE, } r\text{) received from } q\}$

$r := r + 1$

when receive (ARE_YOU_ALIVE, $r$) from $q$

send (I_AM_ALIVE, $r$) to $q$

remark: if a majority of processes are correct then the « wait » always terminate (and all correct processes progress)

completeness:
let $t$ s.t. after time $t$ no new process crashes and all messages from crashed processes are received;
p receives only messages from processes that are alive

intersection: two majorities intersect!

with a majority of correct processes $\Sigma$ is implementable
We have to prove:

1. With $\Sigma$ it is possible to implement registers.

2. Consider an implementation of registers with a failure detector $\mathcal{D}$, we have to prove that $\Sigma \preceq \mathcal{D}$ (we can extract $\Sigma$ from $\mathcal{D}$)

$\Sigma$ weakest FD for registers
Σ is the weakest failure detector to implement registers in message passing
Weakest failure detector to solve consensus

- Failure detector $\Omega$: outputs at each correct process a leader (process id) such that eventually all correct processes agree forever in the same leader and this leader is a correct process.

For all histories $H$ of $\Omega$:

$$\exists \text{leader } \in \text{Correct}(F) \exists t \forall t' > t \forall p \in \text{Correct}(F) H(p, t') = \text{leader}$$

Main result: in shared memory) $\Omega$ is the weakest failure detector to solve the consensus [CHT16]

To get consensus we need a leader!
Solving problem P (practical point of view)

- Given some problem P that is not solvable in asynchronous models
  - From a problem P, find the weakest failure detector \( \text{WFD}(P) \) for P
  - Find the « best » model in which \( \text{WFD}(P) \) can be implemented + a « good » implementation of \( \text{WFD}(P) \)
  - With the « good » implementation of \( \text{WFD}(P) \) and a (good) algorithm using FD: \( \text{WFD}(P) \) solves P
Implementing failure detectors

- Implementing \( \Omega \):
  - « good » partially synchronous model: e.g. in message passing
    - link \( p \) to \( q \) is *eventually timely*: there exists \( \delta \), there exists a time after which all messages from \( p \) to \( q \) are received in less than \( \delta \)
    - if there is a (correct) process \( p \) such that all links from \( p \) are eventually timely then \( \Omega \) may be implemented (source)
  - (then in such systems consensus is possible)
implementing $\Omega$

process $p$:
- sends regularly message(alive) to all
- when no (alive) message from $q$ since $\text{timeout}_p[q]$
  - $\text{timeout}_p[q] = \text{timeout}_p[q] + 1$
  - $\text{counter}_p[q] = \text{counter}_p[q] + 1$
  - reset timer for alive message from $q$
  - send to all ($\text{counter}_p$)

- when receives (counter)
  - $\text{counter}_p = \max(\text{counter}_p, \text{counter})$
  - send to all ($\text{counter}_p$)

$\text{leader}_p = \min\{r | \text{counter}_p(r) = \min \{\text{counter}_p(x) | x \in \mathcal{P}\}\}$

Consider a source $s$:
- at some time for all (alive) processes $\text{timeout}_p[s]$ is greater than communication delay from $s$
- then $\text{counter}_p[s]$ stops increasing eventually
- $\text{counter}_p[s] = \text{counter}_q[s]$

$q$ is not a source $\text{counter}_p[q]$ increases forever for all $p$
Byzantine failures
• A correct process is a process that follow its code

• A byzantine process might fail by exhibiting arbitrary behavior ( e.g. send bogus message )

• ( but it can’t pretend to be another process)( or no message)
• With at most $t$ crash failures, in synchronous system we can solve consensus

• This is not true with Byzantine failure

• First define « byzantine agreement » : the previous definition of validity « the decided value is a value proposed by some process » doesn’t work
Byzantine agreement

- **Agreement**: No two correct processes decide on different values.

- **Validity**: If all correct processes propose the same value $v$ then $v$ is the only possible decision for correct processes.

- **Termination**: All correct processes eventually decide.
Impossibility result

- In synchronous system

- Theorem: If $n \leq 3t$ then it is impossible to solve byzantine agreement

- Proof $n=3 \ t=1$
Decide

Proposed value:
Attack or No
\[ A_1 \] Attack

\[ A_2 \] Attack

\[ A_3 \] Attack

No \[ A_1 \]

No \[ A_2 \]

No \[ A_3 \]
Attack

$A_1$

No

$A_2$

No

$A_3$

No

$A_1$

No

$A_2$

No

83
• The proof may be easily extends to any value of $n$
\[ n \geq 3t \]
\[ n \geq 3t \]