COMP90020: Distributed Algorithms

6. Consensus in DS with Byzantine Failures

Hard as Getting Byzantine Generals to Agree on Anything

Miquel Ramirez

THE UNIVERSITY OF MELBOURNE

Semester 1, 2019

Agenda

1. Models of Failure
2. Models of Distributed Systems & Algorithms
3. Consensus: Introduction
4. Biblio & Reading

Agenda

1. Models of Failure
2. Models of Distributed Systems & Algorithms
3. Consensus: Introduction
4. Biblio & Reading

Distributed Systems are Complex Systems

Distributed computing radically different from uniprocessor settings

→ Process execution is interleaved, enabling race conditions

Distributed Systems are Complex Systems

Distributed computing radically different from uniprocessor settings

→ Process execution is interleaved, enabling race conditions

Non–Determinism: Running a distributed system (DS) twice from same initial conditions yields different results.

Distributed Systems are Complex Systems

Distributed computing radically different from uniprocessor settings

→ Process execution is interleaved, enabling race conditions

Non–Determinism: Running a distributed system (DS) twice from same initial conditions yields different results.

Complexity: Number of possible DS configurations exponential on the number of processes.

Distributed Systems are Complex Systems

Distributed computing radically different from uniprocessor settings

→ Process execution is interleaved, enabling race conditions

Non–Determinism: Running a distributed system (DS) twice from same initial conditions yields different results.

Complexity: Number of possible DS configurations exponential on the number of processes.

Partial Knowledge: Processes in DS lack up-to-date knowledge of the global state of the system.

Models of Non-Determinism

Both processes and comms channels can fail to uphold guarantees

- Omission – failing to do something
- Timing – failing to do something in a timely fashion
- Byzantine – processes and channels show arbitrary behaviour

Models of Non-Determinism

Both processes and comms channels can fail to uphold guarantees

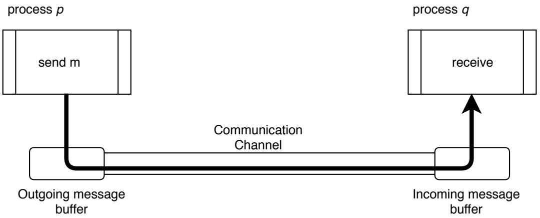
- Omission – failing to do something
- Timing – failing to do something in a timely fashion
- Byzantine – processes and channels show arbitrary behaviour

Failure Models are useful to design robust algorithms for DS

→ Identify special cases which are easier to handle

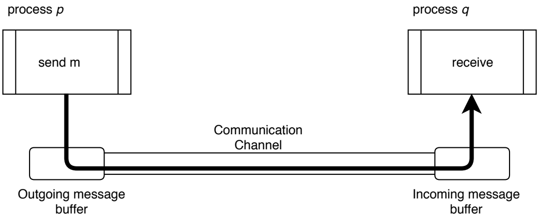
→ Apply divide & conquer to design problem: see next slide

Reliable One–to–One Communications



Strategy: Construct reliable service masking comms channel failures

Reliable One–to–One Communications



Strategy: Construct reliable service masking comms channel failures

- Validity – All outgoing messages eventually delivered
- Integrity – Messages identical to one sent, delivered exactly once

Integrity is crucial and actionable (sequence numbers, digital certificates)

The Plan for the Next Two Lectures

- System Models:
  - → How do we define a DS formally?
- The Consensus, Byzantine Generals and Interactive Consistency problems
  - → How to get DS components to agree on something?
- Feasibility under Byzantine failures
  - → Time to redesign your DS, no algorithm will pick up the slack
- Consensus in Asynchronous systems
  - → What can we do when comms lag masks failures?
- Las Vegas consensus algorithms
  - → Because Monte Carlo is too posh

Agenda

1. Models of Failure
2. Models of Distributed Systems & Algorithms
3. Consensus: Introduction
4. Biblio & Reading

Distributed Systems

A Distributed System consists of

- *Finite* network of N *uniquely identified* processes.

$$\mathcal{P} = \{p_1, p_2, \ldots, p_i, \ldots, p_N\}$$

Distributed Systems

A Distributed System consists of

- *Finite* network of N *uniquely identified* processes.

$$\mathcal{P} = \{p_1, p_2, \ldots, p_i, \ldots, p_N\}$$

- Processes are connected by E channels,
- only one channel between any two processes,

$$E = \{(p_i, p_j) \mid i \neq j, \}$$

Distributed Systems

A Distributed System consists of

- *Finite* network of N *uniquely identified* processes.

$$\mathcal{P} = \{p_1, p_2, \ldots, p_i, \ldots, p_N\}$$

- Processes are connected by E channels,
- only one channel between any two processes,

$$E = \{(p_i, p_j) \mid i \neq j,\}$$

- network diameter D is *distance* between any two processes.

$$D = \max|\pi(i,j)|,\ \pi(i,j) = (q_1, \ldots q_k, \ldots, q_m),\ (q_k, q_{k+1}) \in E$$
$$q_k \in \mathcal{P}, q_1 = p_i,\ q_m = p_j$$

Distributed Systems

A Distributed System consists of

- *Finite* network of N *uniquely identified* processes.

$$\mathcal{P} = \{p_1, p_2, \ldots, p_i, \ldots, p_N\}$$

- Processes are connected by E channels,
- only one channel between any two processes,

$$E = \{(p_i, p_j) \ | \ i \neq j, \}$$

- network diameter D is *distance* between any two processes.

$$D = \max|\pi(i,j)|, \ \pi(i,j) = (q_1, \ldots q_k, \ldots, q_m), \ (q_k, q_{k+1}) \in E$$
$$q_k \in \mathcal{P}, q_1 = p_i, \ q_m = p_j$$

- Channels are reliable, processes may fail

Distributed Systems

A Distributed System consists of

- *Finite* network of N *uniquely identified* processes.

$$\mathcal{P} = \{p_1, p_2, \ldots, p_i, \ldots, p_N\}$$

- Processes are connected by E channels,
- only one channel between any two processes,

$$E = \{(p_i, p_j) \ \mid \ i \neq j, \}$$

- network diameter D is *distance* between any two processes.

$$D = \max|\pi(i,j)|,\ \pi(i,j) = (q_1, \ldots q_k, \ldots, q_m),\ (q_k, q_{k+1}) \in E$$
$$q_k \in \mathcal{P}, q_1 = p_i,\ q_m = p_j$$

- Channels are reliable, processes may fail

We will assume network of procs is fully connected (D finite)

Distributed Systems in Motion

Distributed algorithms (DA)

- Steer changes in global states of controlled DS,

Distributed Systems in Motion

Distributed algorithms (DA)

- Steer changes in global states of controlled DS,
- these follow from events generated by the execution of the DA,

Distributed Systems in Motion

Distributed algorithms (DA)

- Steer changes in global states of controlled DS,
- these follow from events generated by the execution of the DA,
- and aim at ensuring certain conditions hold for DS global states,

Distributed Systems in Motion

Distributed algorithms (DA)

- Steer changes in global states of controlled DS,
- these follow from events generated by the execution of the DA,
- and aim at ensuring certain conditions hold for DS global states,
- for every global state reached (always), or at least one (eventually).

Illustrative Question #1

Process	x	y	z	w
p_1	$\top$	$\bot$	$\top$	$\bot$
p_2	$\bot$	$\bot$	$\top$	$\bot$
p_3	$\top$	$\bot$	$\top$	$\top$
p_4	$\top$	$\top$	$\top$	$\bot$

Question!

How many global states are possible for the DS above?

(A): 4

(B): 8

(C): 64

(D): 16

Illustrative Question #1

Process	x	y	z	w
p_1	$\top$	$\bot$	$\top$	$\bot$
p_2	$\bot$	$\bot$	$\top$	$\bot$
p_3	$\top$	$\bot$	$\top$	$\top$
p_4	$\top$	$\top$	$\top$	$\bot$

Question!

How many global states are possible for the DS above?

(A): 4

(B): 8

(C): 64

(D): 16

$\rightarrow$ (64): We have 4 procs, each proc has 4 binary local vars, 4×2^4.

Illustrative Question #2

- $|\mathcal{P}| = 10$,
- each proc $p_i \in \mathcal{P}$ can send 2 messages,
- messages received by proc p_j change local variable x to $\top$ with $\frac{1}{2}$ probability.

Question!

How many executions considering up to 10 time steps are possible for the DS above?

(A): 1,048,576

(B): $\approx\ 3.52^{3082}$

(C): 42

(D): 21

Illustrative Question #2

- $|\mathcal{P}| = 10$,
- each proc $p_i \in \mathcal{P}$ can send 2 messages,
- messages received by proc p_j change local variable x to $\top$ with $\frac{1}{2}$ probability.

Question!

How many executions considering up to 10 time steps are possible for the DS above?

(A): 1,048,576 (B): $\approx\ 3.52^{3082}$

(C): 42 (D): 21

$\rightarrow$ ($\approx\ 3.52^{3082}$): At every step, there are 2^{10}, (1024) possible combinations of messages, and two possible outcomes, so the DS could be in one of $2^{2^{10}}$ states after one round of messages. Over 10 time steps, we get $2^{2^{10^{10}}} \approx\ 3.52^{3082}$ possible reachable states.

Automated Vehicle Platooning

[Youtube] SCANIA's Truck Platooning

DS + DA = Transition Systems

DS under DA captured by transition system $\mathcal{T} = \langle \mathcal{C}, \delta, \mathcal{I}, F \rangle$

- $\mathcal{C}$ is set of configurations (*global states*) γ of DS,
- a transition function $\delta : \mathcal{C} \mapsto \mathcal{C}$, and
- a *set* initial configurations $\mathcal{I} \subseteq \mathcal{C}$,
- and terminal configurations $F \subset \mathcal{C}$, such that $\delta(f) = f$, $f \in F$.

An execution of DA over DS is a sequence

$$h = (\gamma_0, \gamma_1, \gamma_2, \ldots),\ \gamma_0 \in \mathcal{I},\ \gamma_{i+1} = \delta(\gamma_i)$$

Configs γ^* reachable if exists $h = (\gamma_0, \ldots, \gamma_k)$, $\gamma_k = \gamma^*$, where k is finite.

States & Events

Configurations γ made up of the local states of procs and channels.

States & Events

Configurations γ made up of the local states of procs and channels.

Events result in transitions between configurations

States & Events

Configurations γ made up of the local states of procs and channels.

Events result in transitions between configurations

- Synchronous: two events happen at two different processes (p_i, p_j).
- Asynchronous: transitions follow from one, no simultaneous events

States & Events

Configurations γ made up of the local states of procs and channels.

Events result in transitions between configurations

- Synchronous: two events happen at two different processes (p_i, p_j).
- Asynchronous: transitions follow from one, no simultaneous events

Three types of events:

- Internal: reading and writing local variables.
- Send: a message is put through a channel.
- Receive: follows from a Send event from another process.

States & Events

Configurations γ made up of the local states of procs and channels.

Events result in transitions between configurations

- Synchronous: two events happen at two different processes (p_i, p_j).
- Asynchronous: transitions follow from one, no simultaneous events

Three types of events:

- Internal: reading and writing local variables.
- Send: a message is put through a channel.
- Receive: follows from a Send event from another process.

in synchronous DS, Send & Receive happen at the same time.

States & Events

Configurations γ made up of the local states of procs and channels.

Events result in transitions between configurations

- Synchronous: two events happen at two different processes (p_i, p_j).
- Asynchronous: transitions follow from one, no simultaneous events

Three types of events:

- Internal: reading and writing local variables.
- Send: a message is put through a channel.
- Receive: follows from a Send event from another process.

in synchronous DS, Send & Receive happen at the same time.

Causally related events $a \prec b$ assigned times $C(a) < C(b)$ by global clock

Conditions, Assertions and Properties

A condition is logical statement over γ, either true or false

→ A condition P holds on config γ when P is true ($\gamma \models P$).

Conditions, Assertions and Properties

A condition is logical statement over γ, either true or false

$\rightarrow$ A condition P holds on config γ when P is true ($\gamma \models P$).

Three types of conditions (or *properties*)

- Safety: P holds in each reachable config γ
- Liveness: P holds for some γ_i and then in each γ_j, $j > i$
- Invariant: P always holds
  1. $\gamma \models P$, for all $\gamma \in \mathcal{I}$
  2. if $\gamma' = \delta(\gamma)$ and $\gamma \models P$, then $\gamma' \models P$.

Conditions, Assertions and Properties

A condition is logical statement over γ, either true or false

$\rightarrow$ A condition P holds on config γ when P is true ($\gamma \models P$).

Three types of conditions (or *properties*)

- Safety: P holds in each reachable config γ
- Liveness: P holds for some γ_i and then in each γ_j, $j > i$
- Invariant: P always holds
  1. $\gamma \models P$, for all $\gamma \in \mathcal{I}$
  2. if $\gamma' = \delta(\gamma)$ and $\gamma \models P$, then $\gamma' \models P$.

A DA guarantees safety or liveness iff above true for every possible h.

Invariant are satisfied by a DA, then safety is guaranteed too by DA.

Transition System + Condition = Problem

To sum up:

- DA's control the evolution through time of DS
- Transition systems $\mathcal{T}$ describe behaviour of DS under DA control
- Requirements on behaviour formalised as logical conditions
  - $\rightarrow$ Safety: "something bad will never happen"
  - $\rightarrow$ Liveness: "something good will eventually happen"
  - $\rightarrow$ Invariant: "safety from every beginning to every end"

Point to Take Home

We *formulate* the problems DA's solve as the combination of transition systems and conditions .

Agenda

1. Models of Failure
2. Models of Distributed Systems & Algorithms
3. Consensus: Introduction
4. Biblio & Reading

Why Consensus Matters?

Leading truck wants to go straight

Consensus DA guarantee trucks working correctly will follow leading truck

What could go wrong?

Question!

What kind of issues could compromise the DS above?

(A): "Commander" human minder asleep at wheel, NN reads incorrectly road sign.

(B): "Commander" truck Google Maps app flip–flops between routes.

(C): Unit #1 network interface crash.

(D): Unit #2 on board computer rans out of mem due to mem leak

What could go wrong?

Question!

What kind of issues could compromise the DS above?

- (A): "Commander" human minder asleep at wheel, NN reads incorrectly road sign.
- (B): "Commander" truck Google Maps app flip–flops between routes.
- (C): Unit #1 network interface crash.
- (D): Unit #2 on board computer rans out of mem due to mem leak

→ (All of them): These are all examples of "Byzantine" failures.

The Consensus Problem (restricted to Crash Failures)

DS Specification:

- $\mathcal{P} = \{p_1, \ldots, p_N\}$, $E = \{(p_i, p_j), (p_j, p_i) \ \mid \ i \neq j\}$
- Comms reliable, procs subject to Byzantine (crash) failures

The Consensus Problem (restricted to Crash Failures)

DS Specification:

- $\mathcal{P} = \{p_1, \ldots, p_N\}$, $E = \{(p_i, p_j), (p_j, p_i) \mid i \neq j\}$
- Comms reliable, procs subject to Byzantine (crash) failures

Local variables for each p_i:

- *Proposed* value $v(p_i) \in D$, (v_i for short) and received values, V_i^r and V_i^{r-1}
- *Decision* variable $d(p_i) \in D \cup \{\perp\}$, ($d_i$ for short)
- v_i is constant, d_i initially set to $\perp$

The Consensus Problem (restricted to Crash Failures)

DS Specification:

- $\mathcal{P} = \{p_1, \ldots, p_N\}$, $E = \{(p_i, p_j), (p_j, p_i) \mid i \neq j\}$
- Comms reliable, procs subject to Byzantine (crash) failures

Local variables for each p_i:

- *Proposed* value $v(p_i) \in D$, (v_i for short) and received values, V_i^r and V_i^{r-1}
- *Decision* variable $d(p_i) \in D \cup \{\perp\}$, ($d_i$ for short)
- v_i is constant, d_i initially set to $\perp$

DA Design Problem

Find DA that guarantees the following for every execution h

1. Termination: eventually every correct p_i sets d_i to $x \neq \perp$.
2. Agreement: for every correct (p_i, p_j), eventually $d_i = d_j$.
3. Validity: if $v_i = x$ for every correct p_i then $d_i = x$.

Dolev–Strong–Attiya–Welch Algorithm for Consensus

DSAW Consensus for process p_i

Initialization

$$V_i^1 \leftarrow \{v_i\},\ V_i^0 \leftarrow \emptyset$$

In round $1 \leq r \leq |\mathcal{F}| + 1$

1. **B-multicast**$(V_i^r \setminus V_i^{r-1})$
2. $V_i^{r+1} \leftarrow V_i^r$

* On **B-deliver**(V_j) from some p_j

   a. $V_i^{r+1} \leftarrow V_i^{r+1} \cup V_j$

After $|\mathcal{F}| + 1$ rounds

$$d_i \leftarrow \min V_i^{|\mathcal{F}|+1}$$

Assumptions:

- comms are synchronous,
- $\mathcal{F} \subset \mathcal{P}$ set of faulty procs,
- $f = |\mathcal{F}|$
- failures are crashes

Notes:

- Reentrant
- Round duration based on timer

Correctness of DSAW

Termination

- Guaranteed by synchronous communication

Correctness of DSAW

Termination

- Guaranteed by synchronous communication

Agreement & Integrity (Proof Sketch)

- Let γ_l, $l = f + 1$, be cfg with $d_i \neq d_j$ for procs p_i, p_j,
- this can happen iff in γ_{l-1}, a proc p_k sent v_k to p_i and *crashed*, before being able to send to p_j,
- if p_k had v, but p_j did not receive it, then in γ_{l-2} some other proc p_m send v to p_k and crashed,
- easy to see path from γ_0 to γ_l requires $f + 1$ crashes,
- which violates assumption that at most f procs crash.

Correctness of DSAW

Termination

- Guaranteed by synchronous communication

Agreement & Integrity (Proof Sketch)

- Let γ_l, $l = f + 1$, be cfg with $d_i \neq d_j$ for procs p_i, p_j,
- this can happen iff in γ_{l-1}, a proc p_k sent v_k to p_i and *crashed*, before being able to send to p_j,
- if p_k had v, but p_j did not receive it, then in γ_{l-2} some other proc p_m send v to p_k and crashed,
- easy to see path from γ_0 to γ_l requires $f + 1$ crashes,
- which violates assumption that at most f procs crash.

Lower bound for Synchronous Systems

Consensus will require $f + 1$ rounds of message exchanges for any kind of Byzantine failure.

Consensus from RTO-Multicast (Chandra & Toueg)

Consensus *equivalent to* reliable, totally ordered *multicast*.

Consensus from RTO-Multicast (Chandra & Toueg)

Consensus *equivalent to* reliable, totally ordered *multicast*.

How it works?

- All processes p_i form up a group g
- Every p_i makes a call to **RTO-multicast**(v_i, g)
- p_i sets d_i to m_i, first value coming via **RTO-delivers**$()$

Consensus from RTO-Multicast (Chandra & Toueg)

Consensus *equivalent to* reliable, totally ordered *multicast*.

How it works?

- All processes p_i form up a group g
- Every p_i makes a call to **RTO-multicast**(v_i,g)
- p_i sets d_i to m_i, first value coming via **RTO-delivers**$()$

Why it works?

- Termination guaranteed by *reliability* of **RTO-multicast**
- Agreement and Validity guaranteed by **RTO-deliver**
  - Delivery is totally ordered and reliable

Consensus from RTO-Multicast (Chandra & Toueg)

Consensus *equivalent to* reliable, totally ordered *multicast*.

How it works?

- All processes p_i form up a group g
- Every p_i makes a call to **RTO-multicast**(v_i,g)
- p_i sets d_i to m_i, first value coming via **RTO-delivers**()

Why it works?

- Termination guaranteed by *reliability* of **RTO-multicast**
- Agreement and Validity guaranteed by **RTO-deliver**
    - Delivery is totally ordered and reliable

Chandra & Toueg (1996) showed how to obtain RTO multicast from consensus

The Byzantine Generals Problem

DS Specification:

- $\mathcal{P} = \{p_1, \ldots, p_N\}$, $E = \{(p_i, p_j), (p_j, p_i) \mid i \neq j\}$
- There is a leading process $p^* \in \mathcal{P}$ ("the general")
- Comms reliable, procs subject to Byzantine (anything goes) failures

The Byzantine Generals Problem

DS Specification:

- $\mathcal{P} = \{p_1, \ldots, p_N\}$, $E = \{(p_i, p_j), (p_j, p_i) \mid i \neq j\}$
- There is a leading process $p^* \in \mathcal{P}$ ("the general")
- Comms reliable, procs subject to Byzantine (anything goes) failures

Local variables for each p_i:

- *Proposed* value $v(p^*) \in D$, (v^* for short), v_i^j received values
- *Decision* variable $d(p_i) \in D \cup \{\perp\}$, $p_i \neq p^*$, (d_i for short)
- v^* is constant, d_i initially set to $\perp$

The Byzantine Generals Problem

DS Specification:

- $\mathcal{P} = \{p_1, \ldots, p_N\}$, $E = \{(p_i, p_j), (p_j, p_i) \mid i \neq j\}$
- There is a leading process $p^* \in \mathcal{P}$ ("the general")
- Comms reliable, procs subject to Byzantine (anything goes) failures

Local variables for each p_i:

- *Proposed* value $v(p^*) \in D$, (v^* for short), v_i^j received values
- *Decision* variable $d(p_i) \in D \cup \{\perp\}$, $p_i \neq p^*$, (d_i for short)
- v^* is constant, d_i initially set to $\perp$

DA Design Problem

Find DA that guarantees the following for every execution h

1. Termination: eventually every correct p_i sets d_i to v^*.
2. Agreement: for every correct (p_i, p_j), $p_i \neq p^*$, $p_j \neq p^*$, eventually $d_i = d_j = v^*$.
3. Validity: if p^* correct, then every correct p_i, d_i eventually set to v^*.

Lamport-Shostak-Pease's Algorithm for $N \geq 4$, $f < N/3$

Process p^*

In round 1

B-multicast(v^*)

In round 2

Do Nothing

Process p_i

Initialization

$v_i \leftarrow \perp$

In round 1

* On **B-deliver**(v^*) from p^*

$v_i \leftarrow v^*$

In round 2

1. **send**(v_i, p_j) for $p_j \neq p^*$

* On **receive**(v^j) from p_j

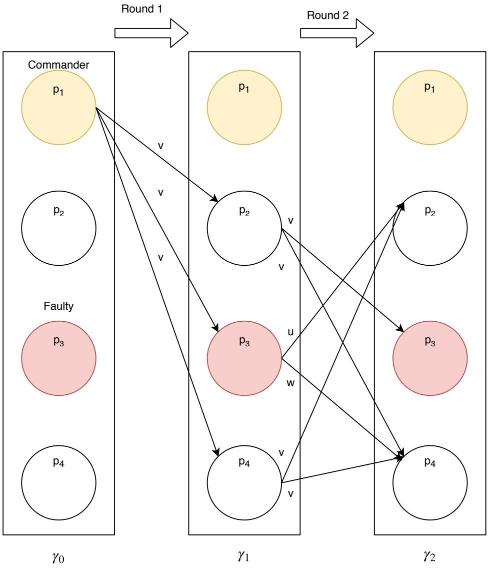
$v_i^j \leftarrow v^j$

2. $d_i = \text{majority}(v_i^1, \ldots, v_i^N)$

$\rightarrow \text{majority}(v_1, v_2, ..., v_n) = \text{argmax}_{v_j} \sum_{v_i} I_{v_j = v_i}$

Example: $\text{majority}(1, 1, 3, 4, 4, 3, 5, 1, \perp) = 1$, $\text{majority}(1, 2, 1, 2, 1, 2) = \perp$.

Sample Execution



$v = majority(v, u, v)$

$v = majority(v, v, w)$

Notes on LPS algorithm

Implication of synchronous comms:

- if **send**(v_i, p_j) fails (times out), p_j will set v_j^i to $\perp$,

Notes on LPS algorithm

Implication of synchronous comms:

- if **send**(v_i, p_j) fails (times out), p_j will set v_j^i to $\perp$,

When less than $N/3$ processes are faulty,

- every correct process p_i receives $(2N/3) - 1$ replicas of v^*,
- $\mathrm{majority}$ will filter out messages from faulty procs

Notes on LPS algorithm

Implication of synchronous comms:

- if **send**(v_i, p_j) fails (times out), p_j will set v_j^i to $\perp$,

When less than $N/3$ processes are faulty,

- every correct process p_i receives $(2N/3) - 1$ replicas of v^*,
- $\mathrm{majority}$ will filter out messages from faulty procs

When commander proc p^* fails and all procs correct,

- correct procs p_i will reach consensus (to something),

Notes on LPS algorithm

Implication of synchronous comms:

- if **send**(v_i, p_j) fails (times out), p_j will set v_j^i to $\perp$,

When less than $N/3$ processes are faulty,

- every correct process p_i receives $(2N/3) - 1$ replicas of v^*,
- $\mathrm{majority}$ will filter out messages from faulty procs

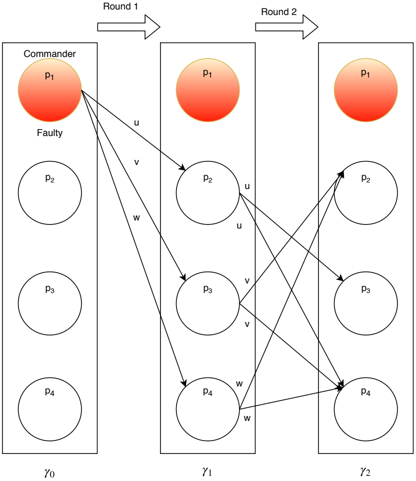
When commander proc p^* fails and all procs correct,

- correct procs p_i will reach consensus (to something),

If p^* failures are fair, sends values equally often

- if all correct, procs p_i will set d_i to $\perp$

Self–Diagnosing Commander is Faulty

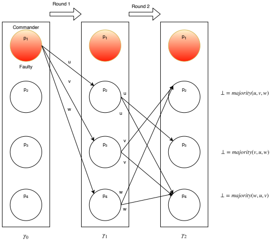


$\perp = majority(u, v, w)$

$\perp = majority(v, u, w)$

$\perp = majority(w, u, v)$

Question: "Unfair" Byzantine failures



Question!

Commander faulty, but sends v to $p4$ rather than w. What are the values of d_i for p_2, p_3 and p_4?

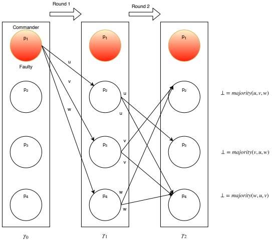
(A): $d_2 = d_3 = d_4 = \perp$

(B): $d_2 = u, d_3 = v, d_4 = w$

(C): $d_2 = v,\ d_3 = u,\ d_4 = v$

(D): $d_2 = d_3 = d_4 = v$

Question: "Unfair" Byzantine failures



Question!

Commander faulty, but sends v to $p4$ rather than w. What are the values of d_i for p_2, p_3 and p_4?

(A): $d_2 = d_3 = d_4 = \perp$

(B): $d_2 = u, d_3 = v, d_4 = w$

(C): $d_2 = v,\ d_3 = u,\ d_4 = v$

(D): $d_2 = d_3 = d_4 = v$

$\rightarrow$ (D): Note that it is quite easy for a hacker taking over p_1 to "poison the well" for the subordinate processes.

Agenda

1. Models of Failure
2. Models of Distributed Systems & Algorithms
3. Consensus: Introduction
4. Biblio & Reading

Further Reading

Coulouris et al. *Distributed Systems: Concepts & Design*

- Chapter 2, Section 2.4.2
- Chapter 15, Section 15.5

Wan Fokkink's *Distributed Algorithms: An Intuitive Approach*

- Chapter 2 - Introduction & Preliminaries
- Chapter 13 - Byzantine Failures

Jeremy Kun *A Programmer's Introduction to Mathematics*

- Chapter 4 - Section 4.1 - *Sets, Functions and their -Jections*
- Chapter 4 - Section 4.3 - *Proof by Induction and Contradiction*