Lecture 10: Fault Tolerance

Fault Tolerant Concurrent Computing

- The main principles of fault tolerant programming are:
  - *Fault Detection* - Knowing that a fault exists
  - *Fault Recovery* - having atomic instructions that can be rolled back in the event of a failure being detected.
- System's viewpoint it is quite possible that the fault is in the program that is attempting the recovery.
- Attempting to recover from a non-existent fault can be as disastrous as a fault occurring.

Fault Tolerant Concurrent Computing (cont'd)

- Have seen replication used for tasks to allow a program to recover from a fault causing a task to abruptly terminate.
- The same principle is also used at the system level to build fault tolerant systems.
- Critical systems are replicated, and system action is based on a majority vote of the replicated sub systems.
- This redundancy allows the system to successfully continue operating when several sub systems develop faults.

Types of Failures in Concurrent Systems

- Initially dead processes (*benign fault*)
  - A subset of the processes never even start
- Crash model (*benign fault*)
  - Process executes as per its local algorithm until a certain point where it stops indefinitely
  - Never restarts
- Byzantine behaviour (*malign fault*)
  - Algorithm may execute any possible local algorithm
  - May arbitrarily send/receive messages

A Hierarchy of Failure Types

- *Dead process*
  - This is a special case of crashed process
  - Case when the crashed process crashes before it starts executing
- *Crashed process*
  - This is a special case of Byzantine process
  - Case when the Byzantine process crashes, and then keeps staying in that state for all future transitions

Types of Fault Tolerance Algorithms

- *Robust algorithms*
  - Correct processes should continue behaving thus, despite failures.
  - These algorithms tolerate/mask failures with *replication* & *voting.*
  - Never wait for all processes as processes could fail.
  - Usually deal with permanent faults.
  - Usually tolerate: $N/2$ *benign*, $N/3$ *malign* failures for N processes.
  - Study of robust algorithms centres around *decision* problems
- (*Self*-)*Stabilizing algorithms*
  - Processes could fail, but eventually become correct.
  - System can start in any state (possibly temporally faulty), but should eventually resume correct behaviour.
  - This *eventually* is known as the *stabilization period*

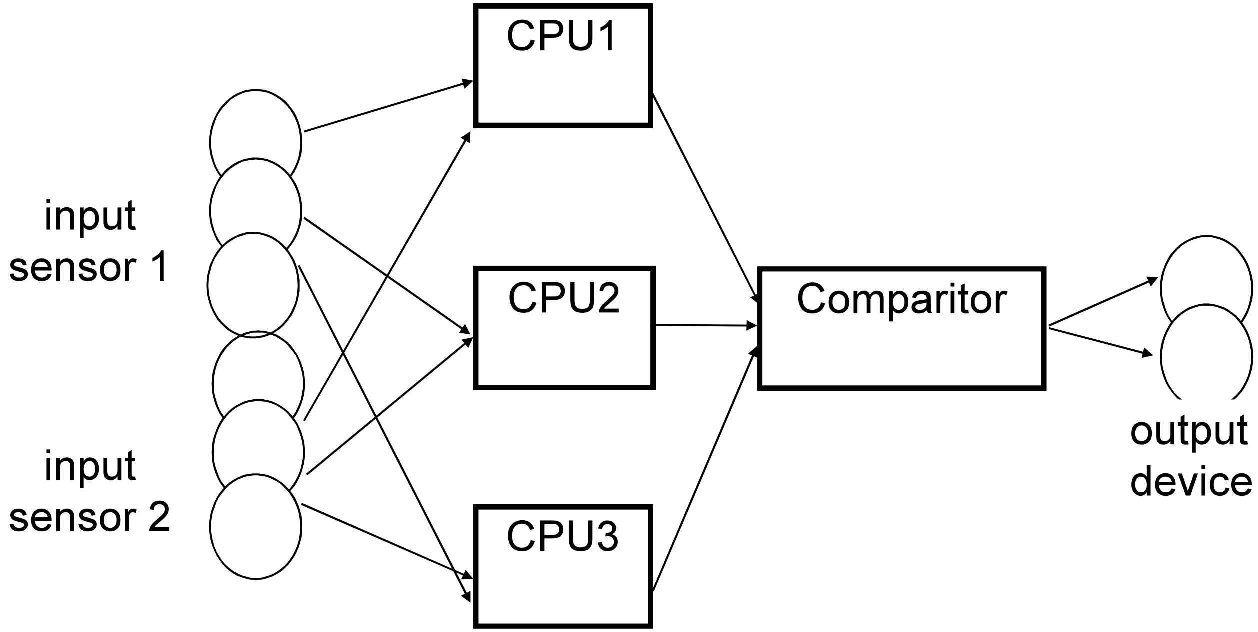
Types of Fault Tolerance Algorithms (cont'd)

- *Robust* (*Self*-)*Stabilizing algorithms*
  – As seen, system can start in any state (possibly temporally faulty), but should eventually resume correct behaviour.
  – BUT during stabilization period, Self-Stabilizing systems do not guarantee <u>any</u> property
  – A (self-)stabilizing algorithm is *robust* if able to quickly start working correctly regardless of initial state, not just mask faults.
  – So, not only is it self-stabilizing but it also guarantees that:
    - After a short time, a basic service is resumed;
    - Basic service maintained until when optimum service resumed.

Decisions in Robust Algorithms

- Robust algorithms typically try to solve some decision problem in which each correct process irreversibly "decides"
- There are 3 requirements for decision problems:
  - *Termination* (All correct processes eventually decide but don't indefinitely wait for all processes to reply)
  - *Consistency* (All correct processes' decisions should be related)
  - *Non-triviality* (Processes should communicate to solve the problem).

Typical Fault Tolerant Architecture



- Have seen that not all sub-systems fail gracefully.
- Instead it continues to operate, generating incorrect data.
- Such problems are called Byzantine Generals problems.
- Diagram above shows how such problems could be handled using a *Comparitor*.

The Byzantine Generals Problem

- This generalises the situation where faulty processes are *actively* traitorous.
- They send messages to others intending to cause a system failure.
- Units of the Byzantine army are preparing to enter a battle.
- A general leads each unit, and all generals communicate with each other by sending messengers.
- These messengers:
  - Do not alter a message once it is given to them.
  - Always make to their destination.
  - Always identify the sender of the message.

The Byzantine Generals Problem (cont'd)

- Generals have pre-arranged a set of alternative actions, such as attack, retreat, or hold a position.

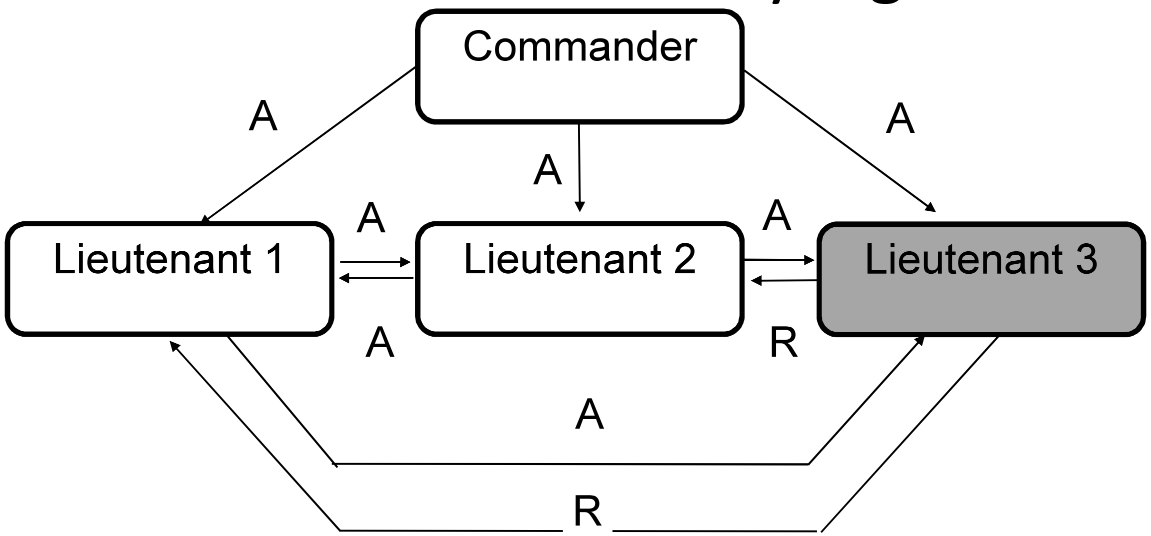
- The goal is to develop an algorithm such that:
  1. All loyal generals take the same decision.
  2. Every loyal general must base his decision on correct information from every other loyal general.

The Byzantine General Algorithm for One Traitorous General

- One general, the commander, decides on an initial decision. The remaining generals are called *lieutenants*.
- The algorithm for one traitorous general is:
  1. Commander sends his decision.
  2. Each lieutenant relays the commander’s decision to every other lieutenant.
  3. Upon receiving both the direct message from the commander and the relayed messages from the other lieutenants, the lieutenant decides on an action my majority vote.

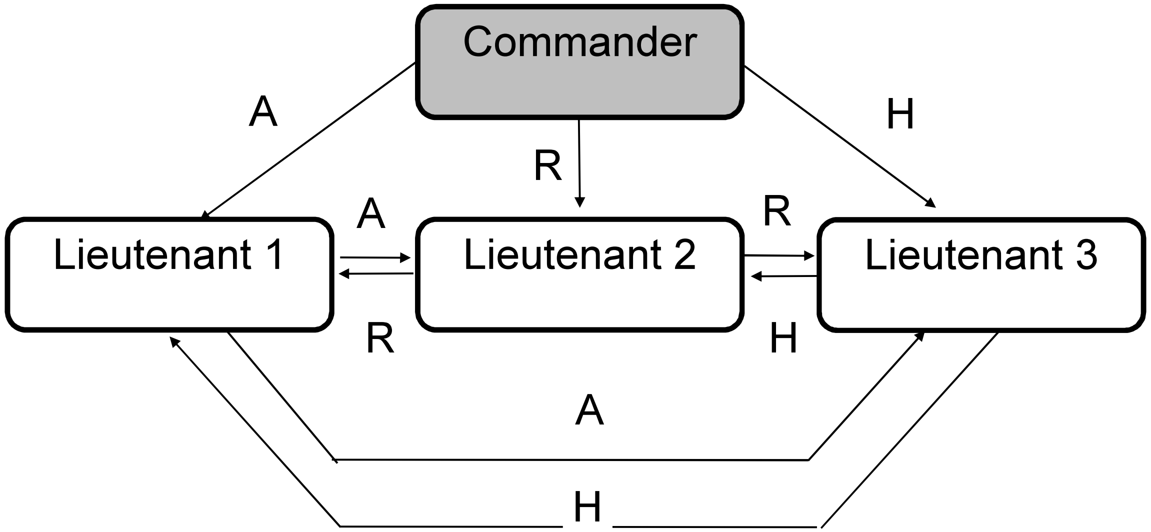
The Byzantine General Algorithm for One Traitorous General

- If a lieutenant traitorous, each loyal lieutenant will receive
  - $(n - 3)$ correct messages from other loyal lieutenants,
  - a correct message from the commander,
  - and an incorrect message from the traitor.
- In order for there to be a majority n must be greater than 3
- There is no known solution for only 3 generals.



The Byzantine General Algorithm cont'd: Traitorous Commander

- For a traitorous commander, it doesn't matter what messages he sends, as all lieutenants are loyal they will relay messages received from the commander.
- Each lieutenant receives the exact same set of messages.
- Since lieutenants all react the same way on the information they receive, they will all make the same decision.

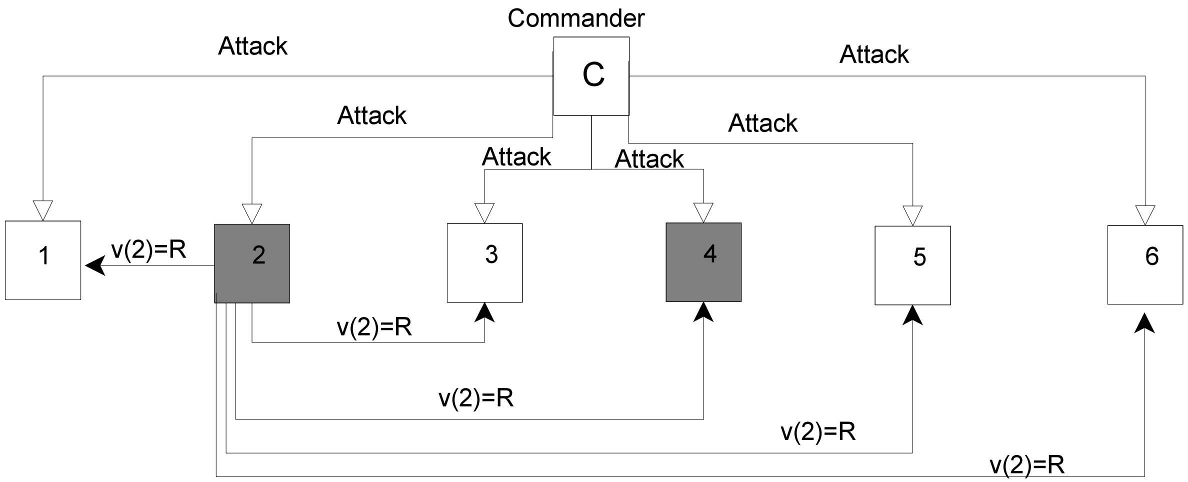


Byzantine General for Two Traitorous Generals

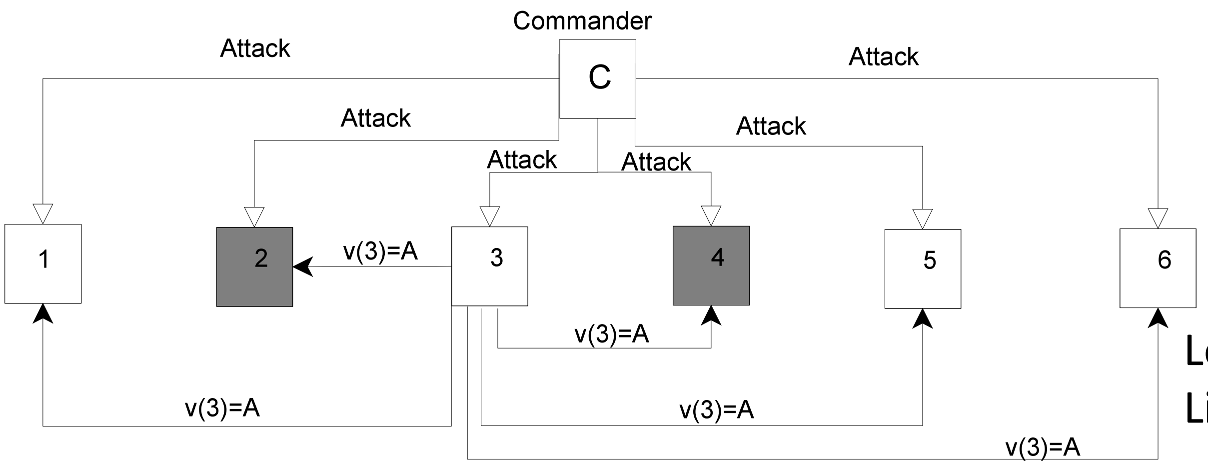
1. Commander sends his decision to each of the $n - 1$ lieutenants.
   - This is called the *level-2 message*.
2. Each lieutenant i sends the *level-2* message to each of the $n - 2$ other lieutenants. This is a *level-1* message $v(i)$.
3. Each lieutenant k sends each of the $n - 3$ *level-1* messages $v(i)$ to the other $n - 2$ lieutenants. This is a *level-0* message $v(i, k)$.
4. Eventually each lieutenant i receives $n - 2$ messages from lieutenant k; one *level-1* message $v(k)$ & $n - 3$ *level-0* messages $v(j, k)$ $j \neq k$.
   - Using majority vote (ie $n - 2$ odd) lieutenant i can determine a value for lieutenant j.
5. Using the $n - 2$ values from the other lieutenants and the *level 2* message from the commander (commander could be traitor, after all), lieutenant i can use majority voting to determine his action.

In general an algorithm exists if less than a third of generals are traitorous.

Byzantine General for Two Traitorous Generals (cont'd)



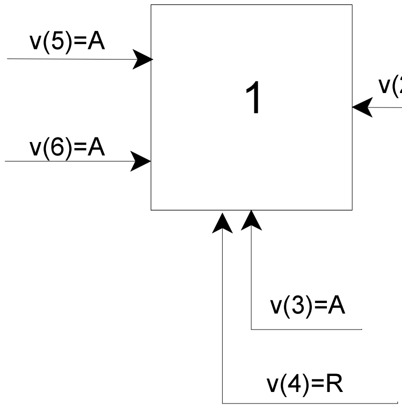
Level 1 , Level 2 Msges Sent by Traitor and Commander (resp):



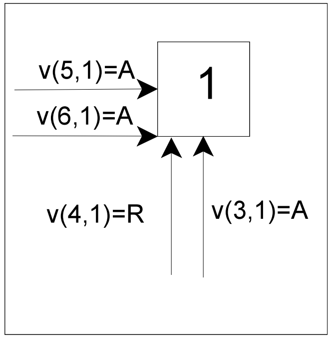
Level 1, Level 2 Msges Sent by Loyal Lieutenant and Commander (resp):

Byzantine General for Two Traitorous Generals (cont'd)

Level 1 Messages Received by 1

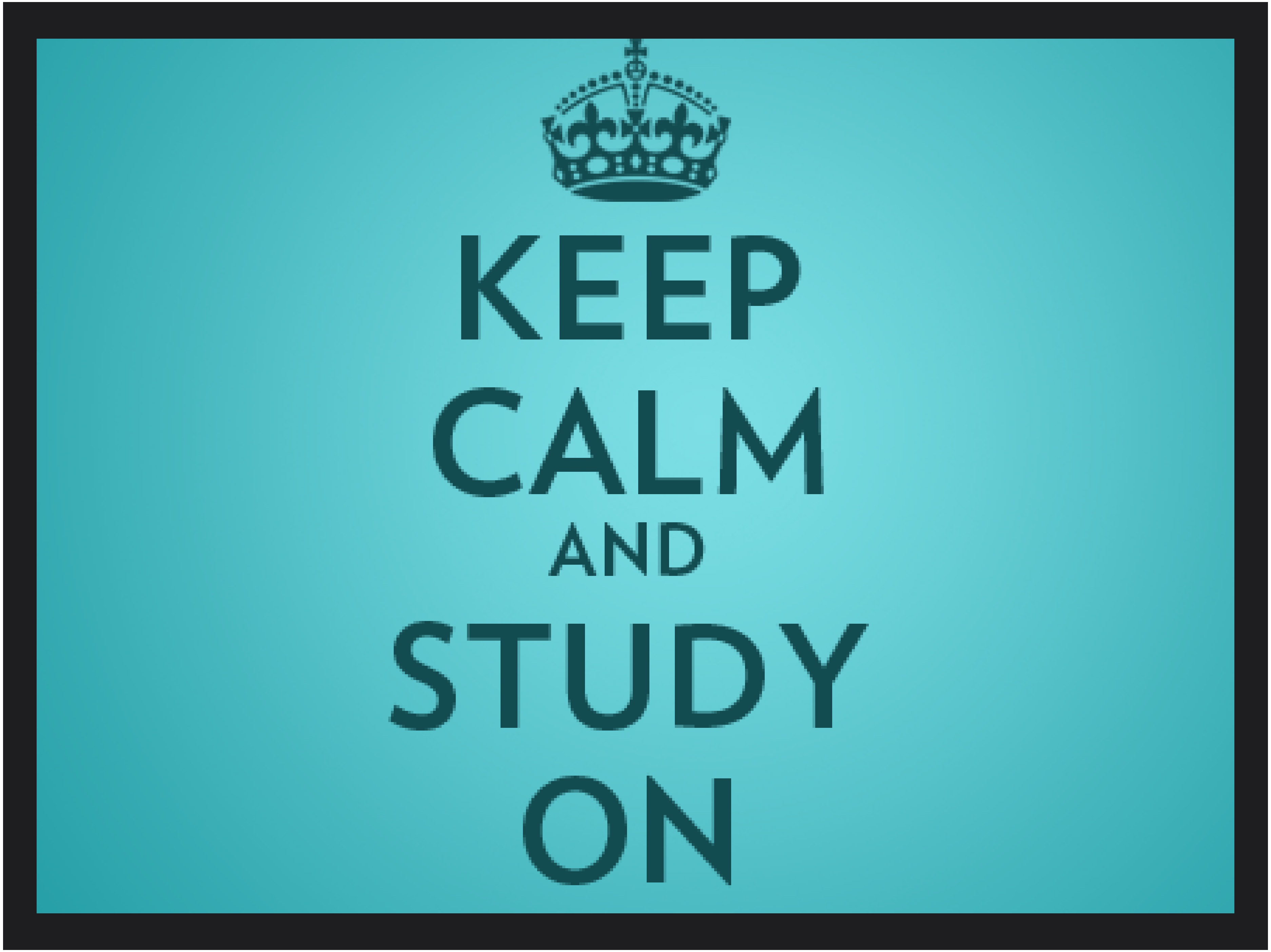


Level 0 Messages Sent by 1 to 2



What messages does (e.g.) Lieutenant 3 get?:

Level/From	1	2*	-	4*	5	6
Level 0	RARA	RAAA	-	RAAA	RARA	RARA
Level 1	A	R	-	R	A	A
Majority Decision on others	A	A		A	A	A
Level 2	A					


KEEP
CALM
AND
STUDY
ON