TDD25
Distributed Systems

Fault Tolerance

Christoph Kessler

IDA
Linköping University
Sweden

2024

LINKÖPING
UNIVERSITY

Agenda

FAULT TOLERANCE

1. Fault Tolerant Systems
2. Faults and Fault Models
3. Redundancy
4. Time Redundancy and Backward Recovery
5. Hardware Redundancy
6. Software Redundancy
7. Distributed Agreement with Byzantine Faults
8. The Byzantine Generals Problem

Fault Tolerant Systems

- A system **fails** if it behaves in a way which is not consistent with its specification. Such a **failure** is a result of a **fault** in a system component.

- Systems are **fault-tolerant** if they behave in a predictable manner, according to their specification, in the presence of faults
  → **there are no failures in a fault-tolerant system.**

- Several application areas need systems to maintain a correct (predictable) functionality in the presence of faults:
    - banking systems
    - avionics, medical, automotive
    - manufacturing systems

What means **correct functionality in the presence of faults?**

- The answer depends on the application (on the specification of the system):
    - The system stops and does not produce any erroneous (dangerous) result / behaviour.
    - The system stops and restarts after a while without loss of information.
    - The system keeps functioning without any interruption and (possibly) with unchanged performance.

Faults

A fault can be:

- **Hardware fault:** malfunction of a hardware component (processor, communication line, switch, etc.).
- **Software fault:** malfunction due to a software bug.

A fault can be the result of:

1. **Mistakes in specification or design:** such mistakes are at the origin of all software faults and of some of the hardware faults.
2. **Defects in components**: hardware faults can be produced by manufacturing defects or by defects caused as result of deterioration in the course of time.
3. **Operating environment**: hardware faults can be the result of stress produced by adverse environment: temperature, radiation, vibration, etc.

Faults

Fault types according to their **temporal behaviour**:

1. **Permanent fault**:
   the fault remains until it is repaired
   or the affected unit is replaced.
2. **Intermittent fault**:
   the fault vanishes and reappears
   (e.g. caused by a loose wire).
3. **Transient fault**:
   the fault dies away after some time
   (caused by environmental effects).

Faults

Fault types according to their **output behaviour**:

1. **Fail-stop fault** (**omission faults**):
    - Either the processor is executing and produces *correct* values, or it failed and will *never* respond to any request.
        - Working processors can detect the failed processor by a *time-out* mechanism.
2. **Byzantine fault** (**arbitrary faults**):
    - A process can fail and stop, execute slowly, or execute at a normal speed but produce erroneous values and actively try to make the computation fail
        - Any *message* can be corrupted, and correctness has to be decided upon by a group of processors.

- The fail-stop model is the easiest to handle; unfortunately, sometimes it is too simple to cover real situations.
- The Byzantine model is the most general; it is very expensive, in terms of complexity, to implement fault-tolerant algorithms based on this model.

Redundancy

If a system has to be fault-tolerant,
it has to be provided with **spare capacity** → redundancy:

1. **Time redundancy**: the timing of the system is such that if certain tasks have to be rerun and recovery operations have to be performed, system requirements are still fulfilled.
2. **Hardware redundancy**: the system is provided with far more hardware than needed for basic functionality.
3. **Software redundancy**: the system is provided with different software versions:
    - results produced by different versions are compared;
    - when one version fails, another one can take over.
4. **Information redundancy**: data is coded in such a way that a certain number of bit errors can be *detected* and, possibly, *corrected* (using parity coding, checksum codes, cyclic codes).

Backward Recovery

Basic idea: roll back the computation to a previous **checkpoint** and retake from there.

Essential aspects:

- Backward recovery assumes time redundancy!
- The system periodically saves *globally consistent states* of the distributed system, which can serve as *recovery points*.
- When a fault is detected, the system is recovered from the most recent recovery point.

Corrective action:

- Carry on with the same processor and software
  (a *transient fault* is assumed).
- Carry on with a new processor
  (a *permanent hardware fault* is assumed).
- Carry on with the same processor and another software version
  (a *permanent software fault* is assumed).

Forward Recovery

- Backward recovery is based on time redundancy and on the availability of back-up files and saved checkpoints;
  - This is expensive in terms of time.

- Control applications and, in general, real-time systems have very strict timing requirements.
  - Recovery has to be very fast
    and preferably to be continued from the current state.

Forward recovery:
the error is masked without redoing any computations.

  - Forward recovery is based on hardware and, possibly, software redundancy.

Hardware Redundancy

Hardware redundancy: use of additional hardware to compensate for failures:

- **Fault detection, correction, and masking**:

  Multiple hardware units are assigned to the same task in parallel and their results are compared.

  - **Detection**: if one or more (but not all) units are faulty, this shows up as a disagreement in the results.
  - **Correction** and **masking**: if only a minority of the units are faulty, and sufficient units produce the same output, this output can be used to correct and mask the failure.

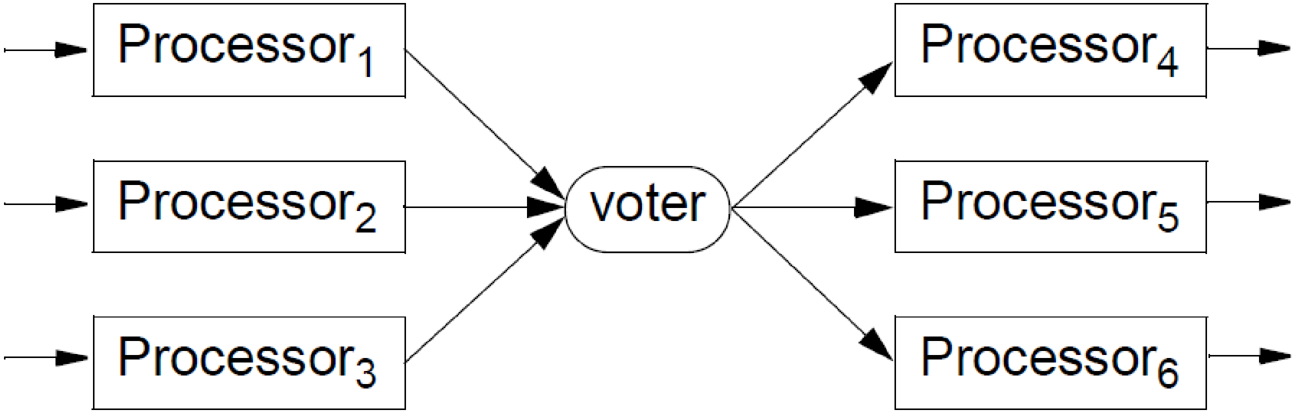
- **Replacement** of malfunctioning units:

  Correction and masking are short-term measures.
  In order to restore the initial performance and degree of fault-tolerance, the faulty unit has to be replaced.

Hardware redundancy is a fundamental technique to provide fault-tolerance in **safety-critical distributed systems**: aerospace applications, automotive applications, medical equipment, some parts of telecommunications equipment, nuclear centres, military equipment, etc.

N-Modular Redundancy

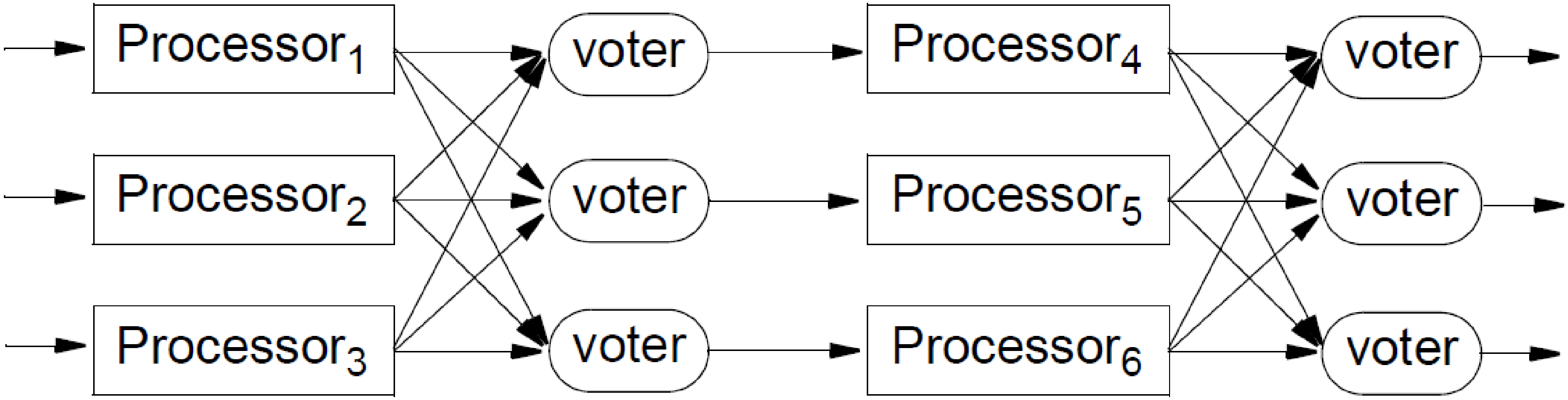
N*-modular redundancy** (N*-MR**) is a scheme for forward error recovery. *N* units are used, instead of one, and a voting scheme is used on their output.



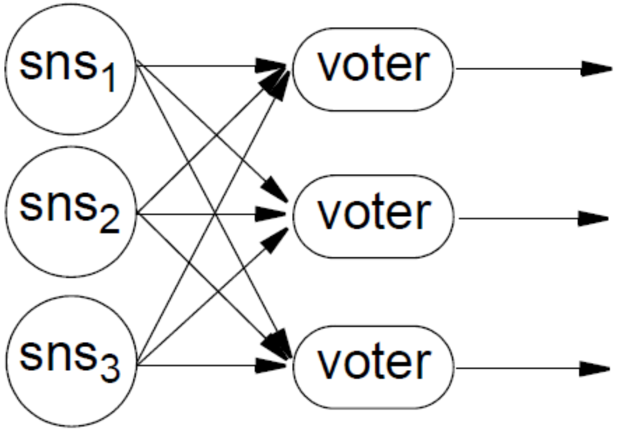
- The same inputs are provided to all participating processors, which are supposed to work synchronously
  - a new set of inputs is provided to all processors simultaneously, and the corresponding set of outputs is compared.
- **3-modular redundancy** is the most commonly used.

N-Modular Redundancy

- The voter itself can fail
  → structure with **redundant voters**:



- Voting on inputs from **sensors**:



Voters

Several approaches for voting are possible.
The goal is to "filter out" the correct value from the set of candidates.

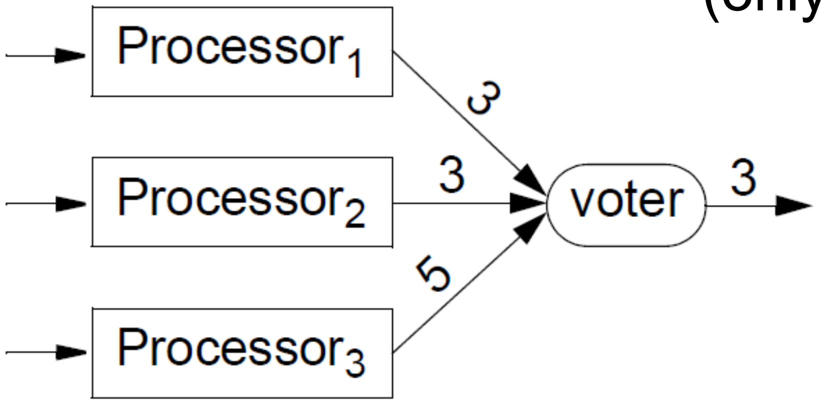
- The most common one: **majority voter**
  - The voter constructs a set of classes of values:

    $P_1, P_2, \ldots, P_n$:

    $x, y \in P_i$ if and only if $x = y$
  - If P_i is the largest set and N is the number of outputs (N is odd):

    if card(P_i) $\geq \lceil N/2 \rceil$ → $x \in P_i$ is correct output;
    the error can be masked.

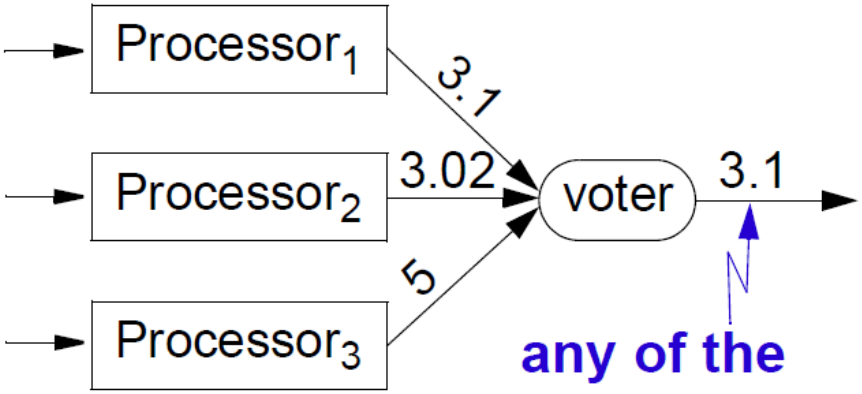
    if card(P_i) $< \lceil N/2 \rceil$ → the error cannot be masked
    (only be detected).



Voters

- Sometimes we can not use strict equality:
  - sensors can provide slightly different values;
  - the same application can be run on different processors, and outputs can be different only because of internal representations used (e.g., floating point).

→ if $|x - y| < \varepsilon$ then we consider $x = y$.

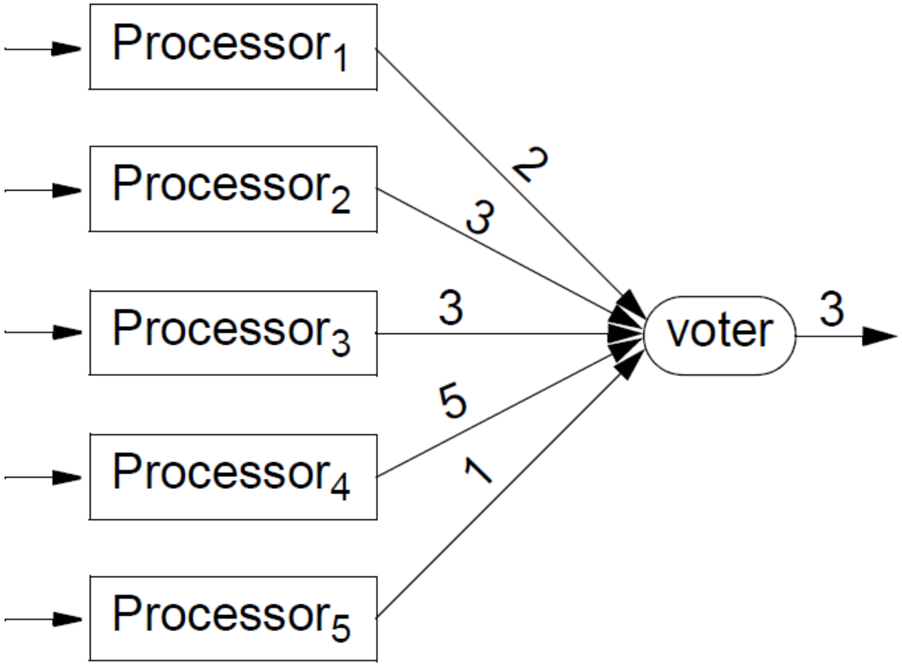


any of the values in set P_i can be selected

Voters

Other voting schemes:

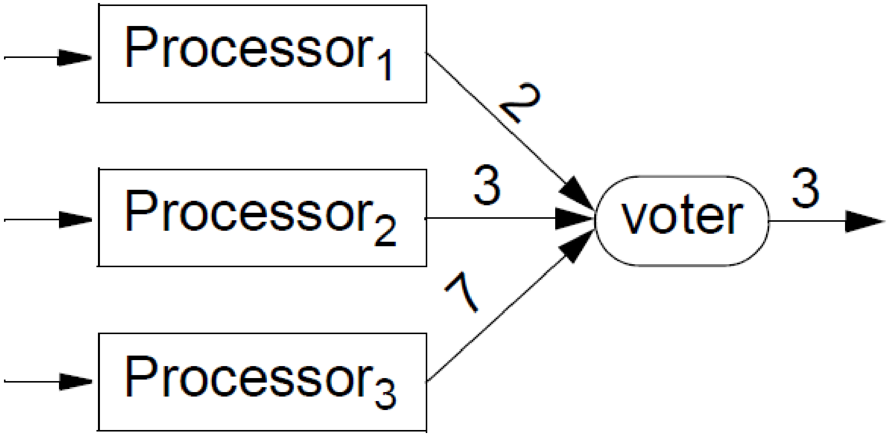
- ***k*-plurality voter**
  - Similar to majority voting:
    the largest set needs not contain more than $N/2$ elements,
    it is sufficient that card(P_i) = k, k selected by the designer



Voters

Other voting schemes:

- **Median voter**
  - The median value is selected.



k-Fault-Tolerant Systems

A system is ***k*-fault-tolerant** if it can survive faults in *k* components and still meet its specifications.

- How many components do we need in order to achieve *k*-fault-tolerance with voting?
  - With **fail-stop faults**:

    having k+1 components is enough to provide *k*-fault-tolerance:
    - if *k* stop, the answer from the one left can be used.
  - With **Byzantine faults**, components continue to work and send out erroneous or random replies:

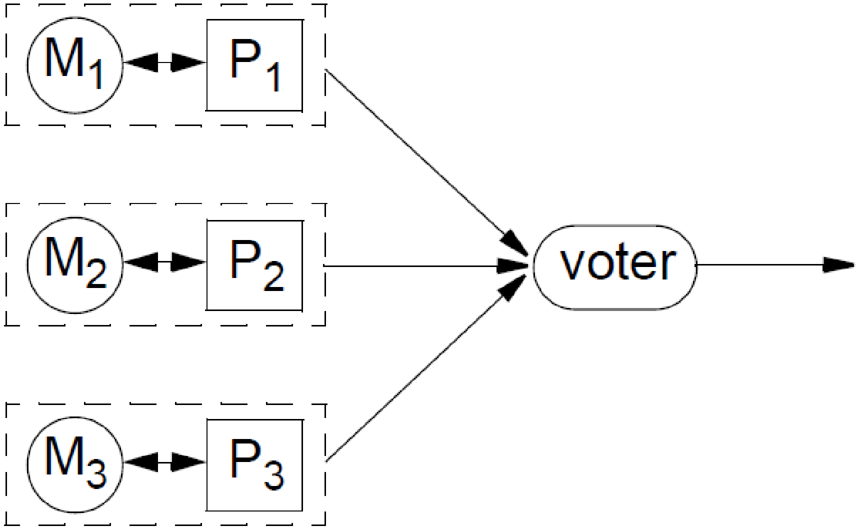
    2k+1 components are needed to achieve *k*-fault-tolerance
    - a majority of k+1 correct components can outvote *k* components producing faulty results.

Processor and Memory Level Redundancy

- *N*-modular redundancy can be applied at any level:
  gates, sensors, registers, ALUs, processors, memories, boards.
- If applied at a lower level,
  time and cost overhead can be high:
  - voting takes time
  - number of additional components (voters, connections) becomes high.

Processor and Memory Level Redundancy

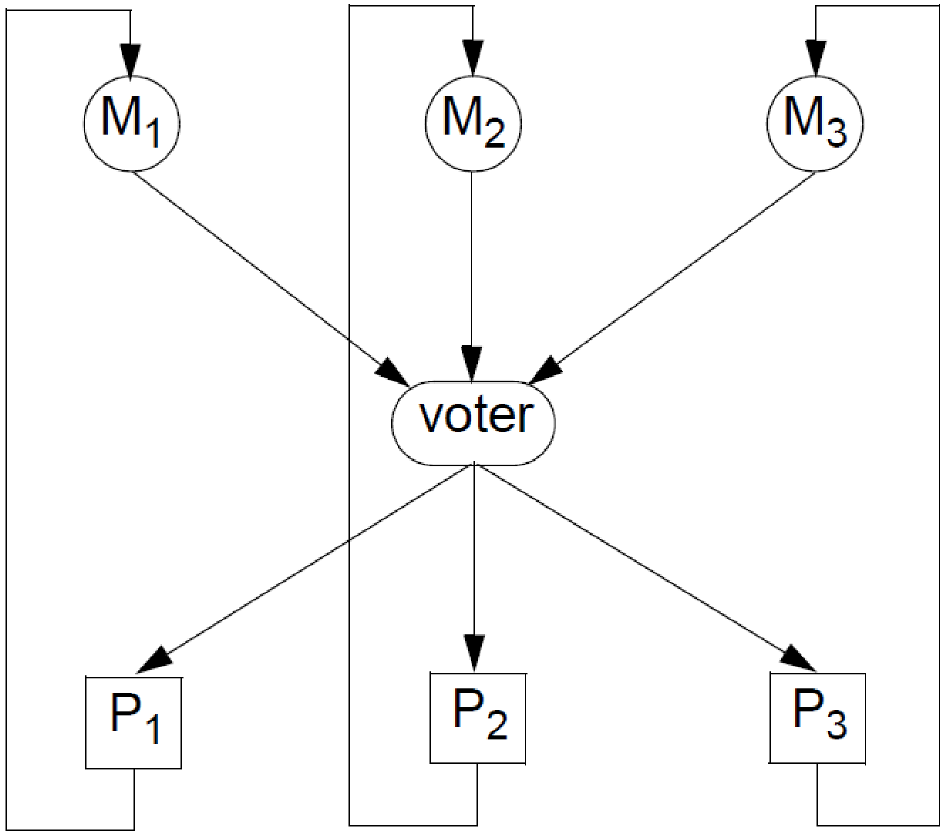
- Processor and memory are handled as a unit; voting is on processor outputs:



Processor and Memory Level Redundancy

Processors and memories can be handled as separate modules.

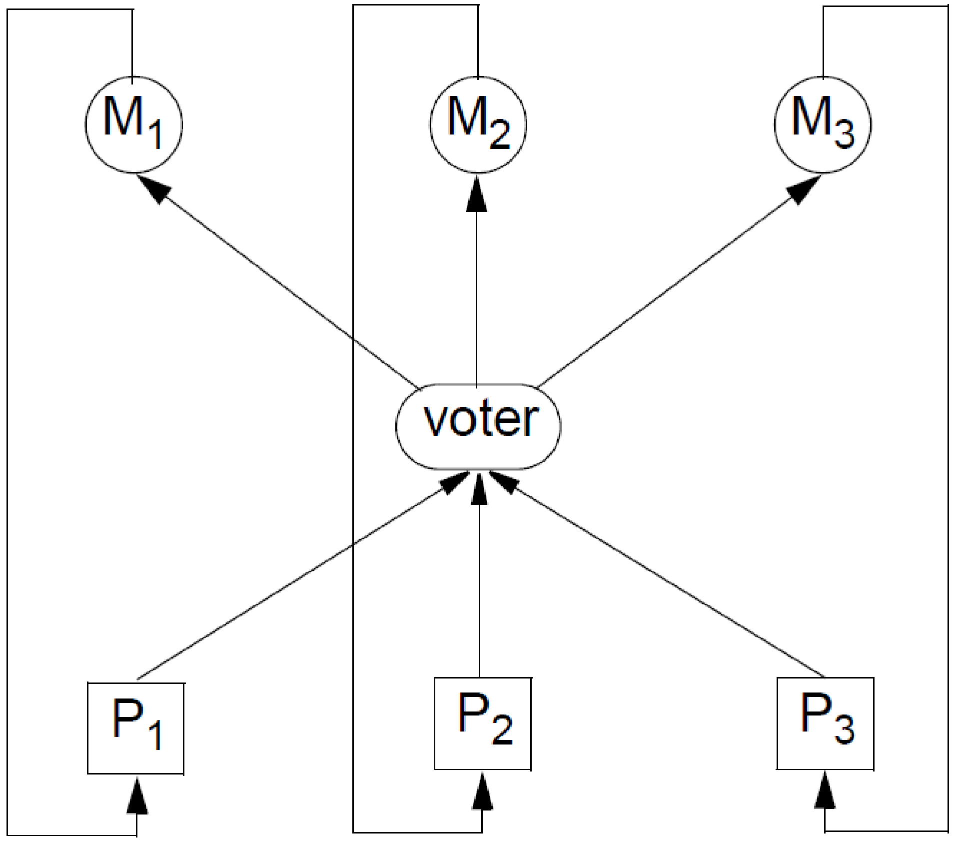
(a) voting at read from memory



Processor and Memory Level Redundancy

Processors and memories can be handled as separate modules.

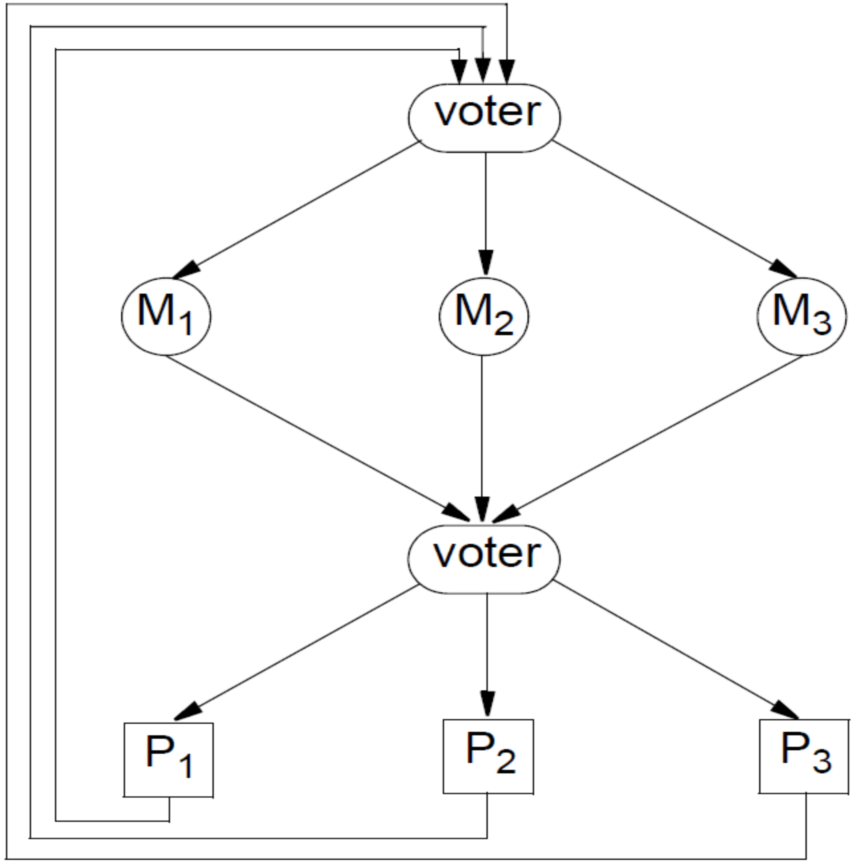
(b) voting at write to memory



Processor and Memory Level Redundancy

Processors and memories can be handled as separate modules.

(c) voting at read <u>and</u> write



Software Redundancy

Software is very different from hardware
in the context of redundancy:

- A software fault is always caused by a mistake in specification or by a bug (a design error).
- Software faults are not produced by manufacturing, aging, stress, or environment.
- Different copies of identical software always produce the same behaviour for identical inputs
- Replicating the same software *N* times, and letting it run on *N* processors, does not provide any *software* redundancy:
if there is a software bug, it will be produced by all *N* copies.

Software Redundancy

- *N* different versions of software are needed in order to provide redundancy.
- Two possible approaches:
  1. All *N* versions are running in parallel; voting is done on the output.
  2. One version is running;
     if it fails, another one takes over after recovery.
- The *N* versions of the software must be **diverse**
  - the probability that they all fail on the same input has to be sufficiently small.
- It is difficult to produce sufficiently diverse versions for the same software:
  - Let independent teams, with no contact between them, generate software for the same application.
  - Use different programming languages.
  - Use different tools like, for example, compilers.
  - Use different (numerical) algorithms.
  - Start from differently formulated specifications

  → Expensive and not always possible

Distributed Agreement with Byzantine Faults

Very often, distributed processes have to come to an **agreement**.

For example, they have to agree on a certain value, with which each of them has to continue operation.

- What if some of the processors are faulty and exhibit Byzantine faults?
- How many *correct* processors are needed in order to achieve *k*-fault-tolerance?

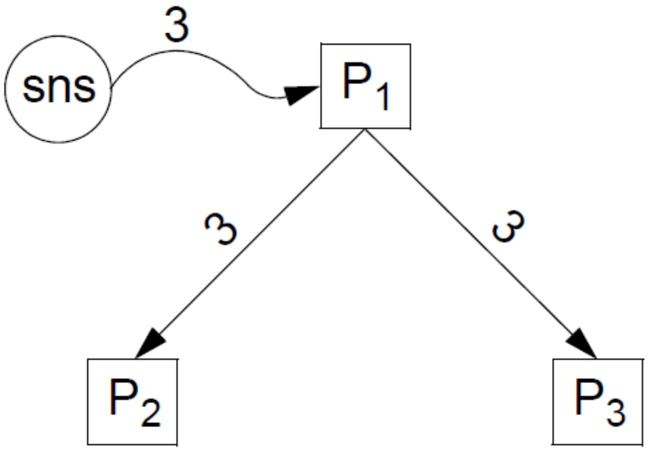
Remember:

- With a simple **voting** scheme, 2*k*+1 components are needed to achieve *k*-fault-tolerance in the case of Byzantine faults

  → 3 processors are sufficient to mask the fault of one of them.

However, this is not the case for agreement!

Distributed Agreement with Byzantine Faults

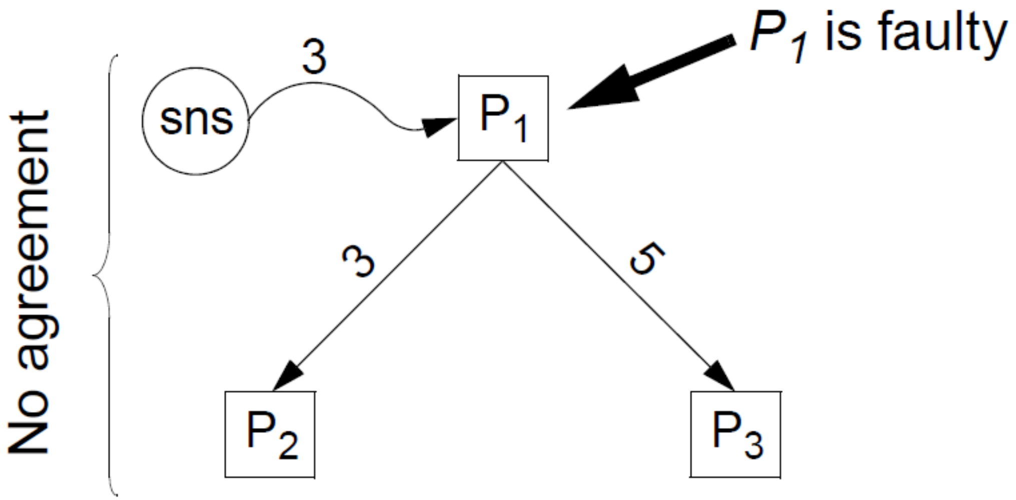


Example

- P_1 receives a value from the sensor, and the processors have to continue operation with that value;

  in order to achieve fault tolerance, they have to **agree** on the value to continue with:
  - **this should be the value received by P_1 from the sensor, if P_1 is not faulty**;
  - **if P_1 is faulty, all non-faulty processors should use the same value to continue with**.

Distributed Agreement with Byzantine Faults

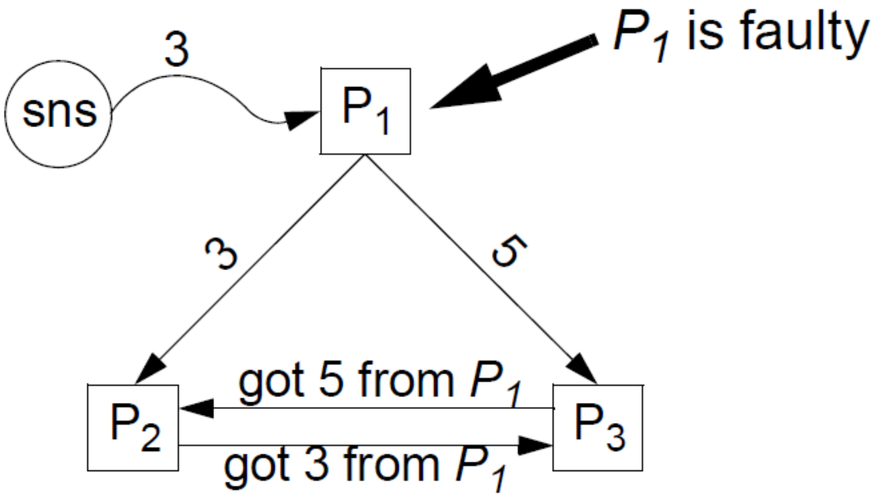


Example

- P_1 receives a value from the sensor, and the processors have to continue operation with that value;

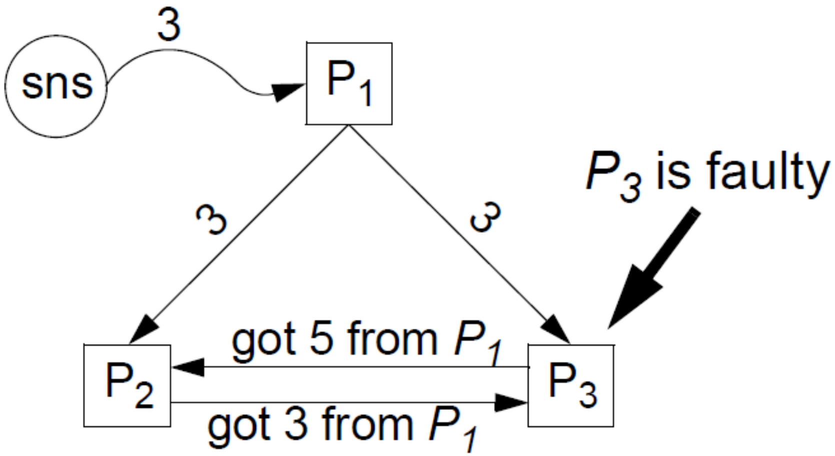
  in order to achieve fault tolerance, they have to **agree** on the value to continue with:
  - **this should be the value received by P_1 from the sensor, if P_1 is not faulty**;
  - **if P_1 is faulty, all non-faulty processors should use the same value to continue with**.

Distributed Agreement with Byzantine Faults



Example

- Maybe, by letting P_2 and P_3 communicate, they could get out of the trouble?
  - P_2 does not know if P_1 or P_3 is the faulty one, thus it cannot handle the contradicting inputs.
  - The same for P_3.
  - → No agreement
- The same if P_3 is faulty:
  - P_2 does not know if P_1 or P_3 is the faulty one, thus it cannot handle the contradicting inputs
  - → No agreement



Distributed Agreement with Byzantine Faults

- With three processors we cannot achieve agreement, if one of them is faulty (with Byzantine behaviour)!
- The **Byzantine Generals Problem** is used as a **model** to study agreement with Byzantine faults

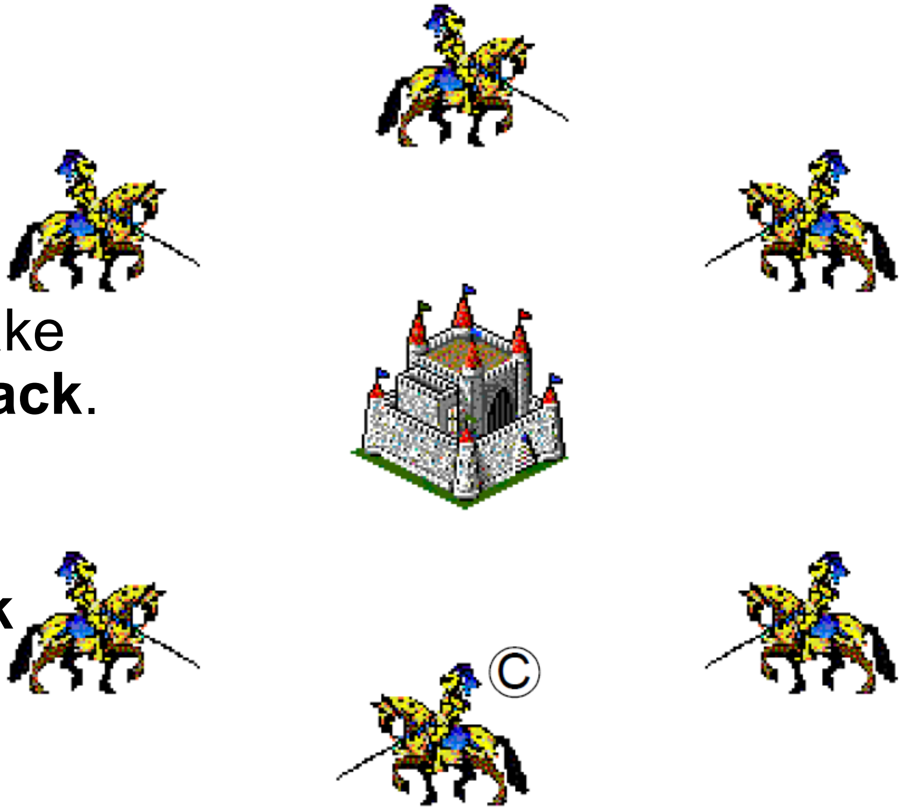
The Byzantine Generals Problem

The Byzantine army is preparing for a battle.

A number of **generals** must coordinate among themselves through (reliable) messengers on whether to attack or retreat.

A **commanding general** (C) will make the **decision whether or not to attack**.

Any of the generals, including the commander, may be **traitorous**: they might send messages to **attack** to some generals and messages to **retreat** to others.



The Byzantine Generals Problem

The problem in the story:

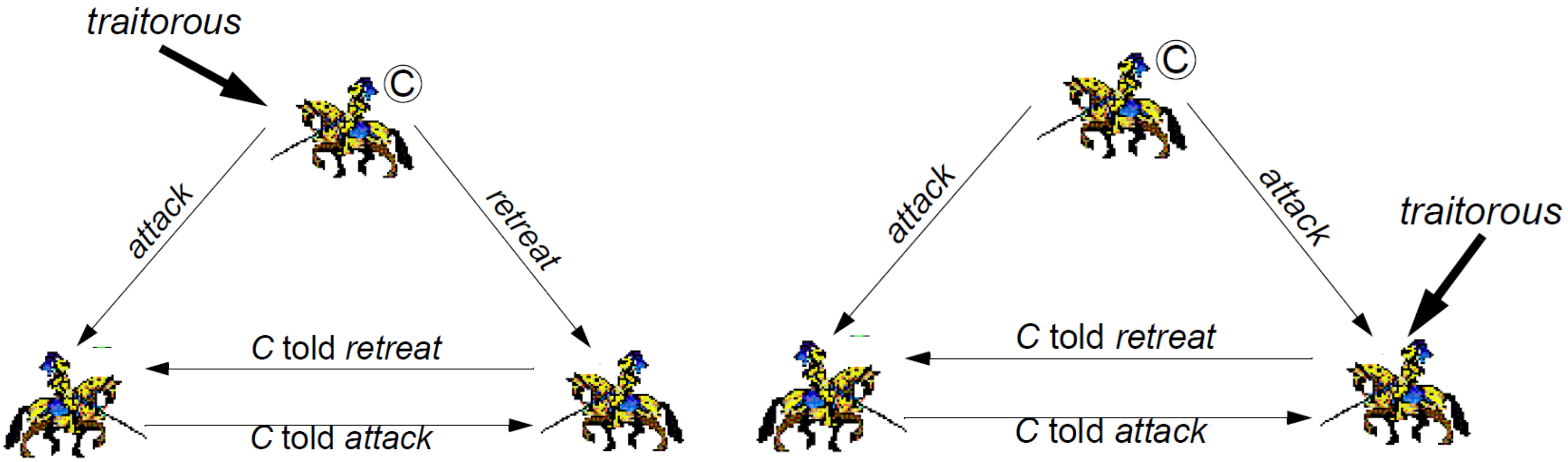
- The loyal generals have all to agree to attack, or all to retreat.
- If the commanding general is loyal, all loyal generals must agree with the decision that he made.

The problem in real life:

- All non-faulty processors must use the same input value.
- If the input unit (P_1) is not faulty,
  all non-faulty processors must use the value it provides.

The Byzantine Generals Problem

The case with **three generals**:



traitorous

C

attack

retreat

C told *retreat*

C told *attack*

C

attack

attack

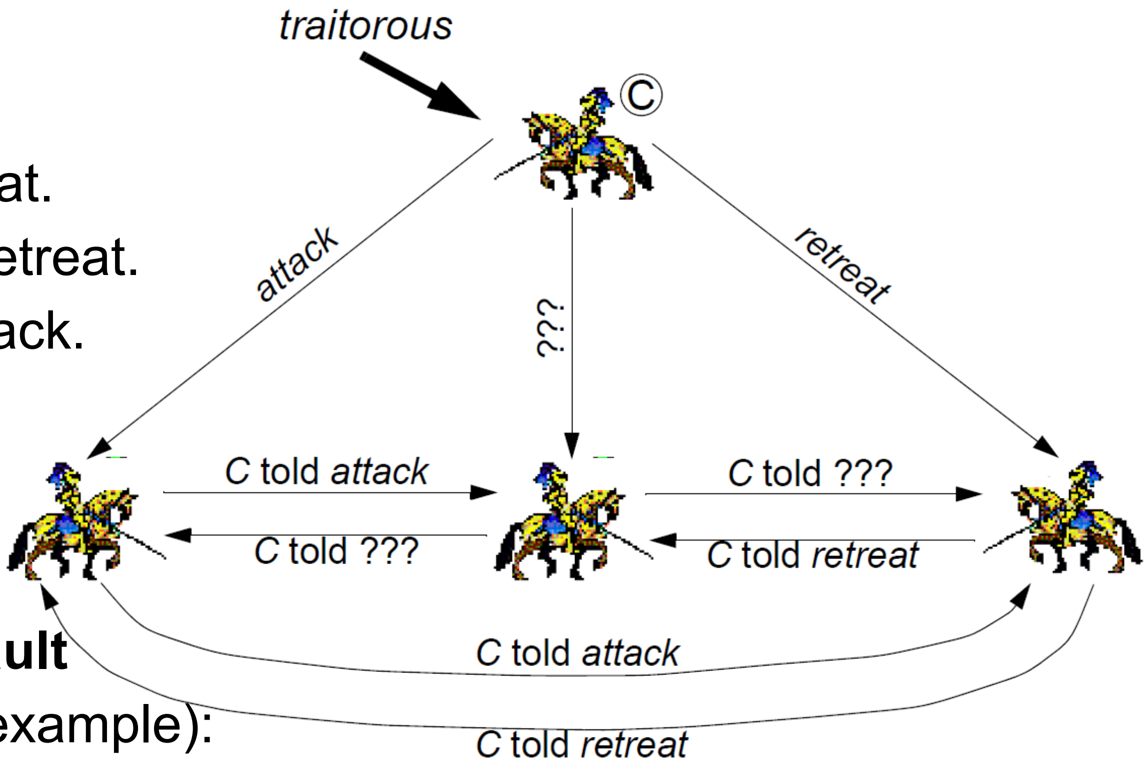
traitorous

C told *retreat*

C told *attack*

No agreement is possible
if one of three generals is traitorous

The Byzantine Generals Problem



The case with **four generals**:

- Gen. left: attack, ???, retreat.
- Gen. middle: ???, attack, retreat.
- Gen. right: retreat, ???, attack.

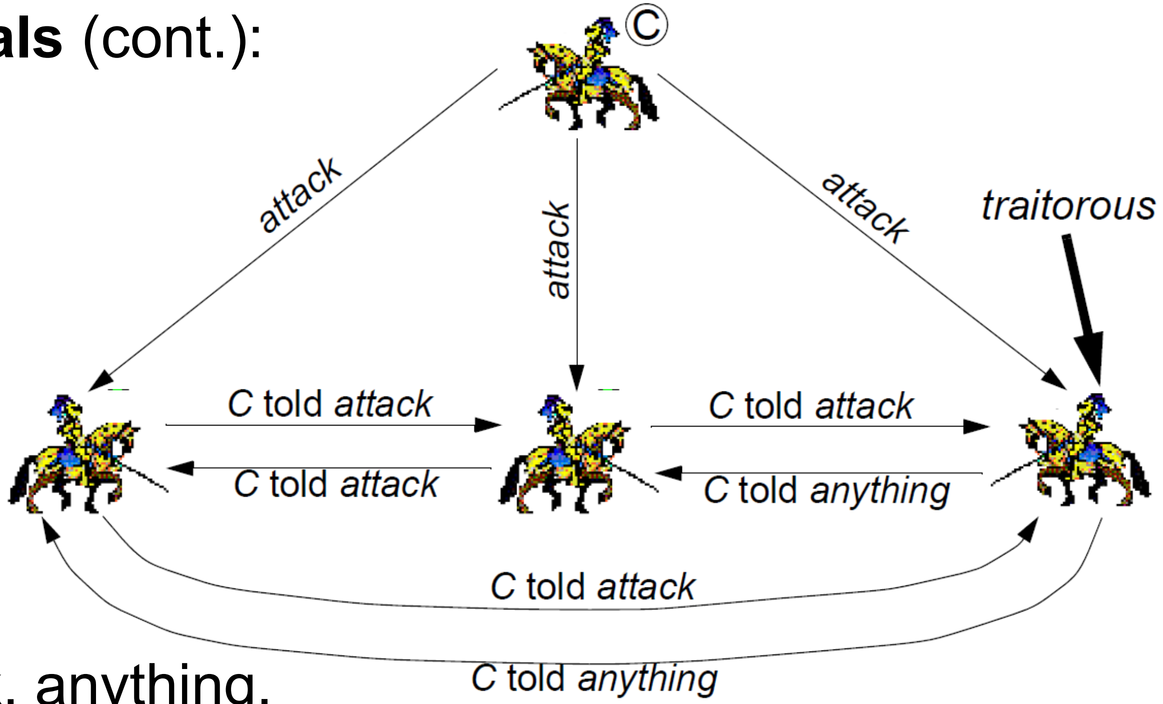
→ The generals decide by **majority voting** on their input; if no majority exists, a **default** value is used (retreat, for example):

- If ??? = attack → all three decide on attack.
- If ??? = retreat → all three decide on retreat.
- If ??? = dummy → all three decide on retreat.

The three loyal generals have reached agreement, despite the traitorous commander.

The Byzantine Generals Problem

The case with **four generals** (cont.):



- Gen. left: attack, attack, anything.
- Gen. middle: attack, attack, anything.

By majority vote on the input messages,
the two loyal generals have agreed on the message
proposed by the loyal commander (attack).

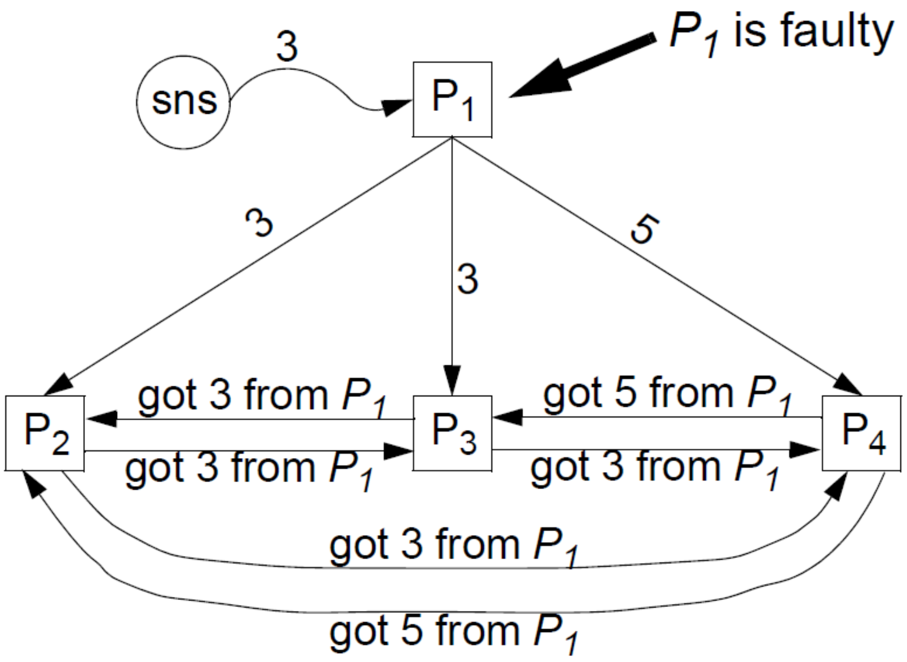
The Byzantine Generals Problem

The conclusion in general:

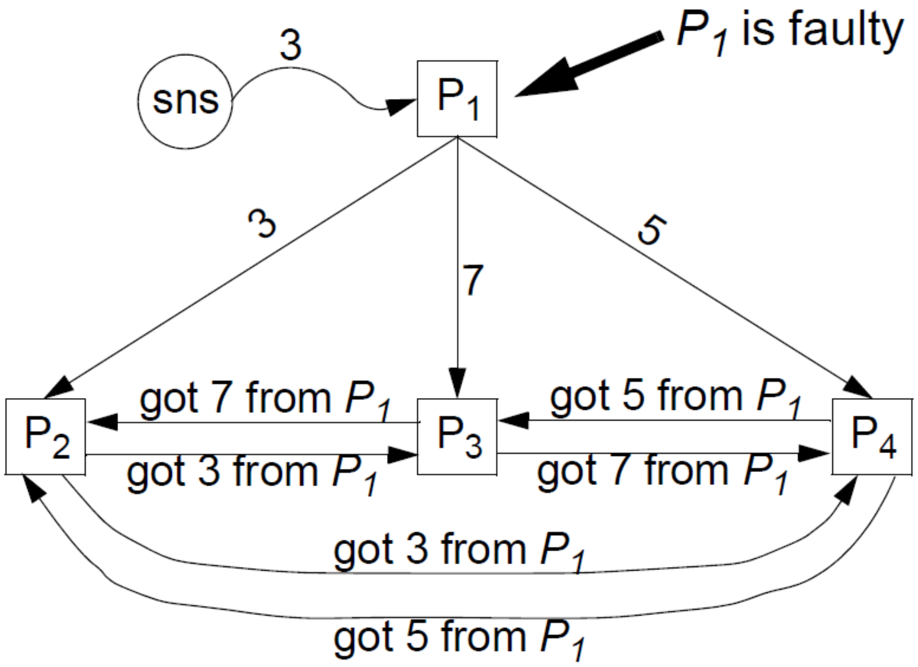
- To reach agreement with *k* traitorous generals requires a total of at least 3*k* + 1 generals.
- **We need 3*k* + 1 processors to achieve *k*-fault-tolerance for agreement with Byzantine faults.**
  - → To mask one faulty processor: total of 4 processors;
  - → To mask two faulty processors: total of 7 processors;
  - → To mask three faulty processors: total of 10 processors;
  - → ...

The Byzantine Generals Problem

Let us come back to our real-life example, this time with **four processors**:



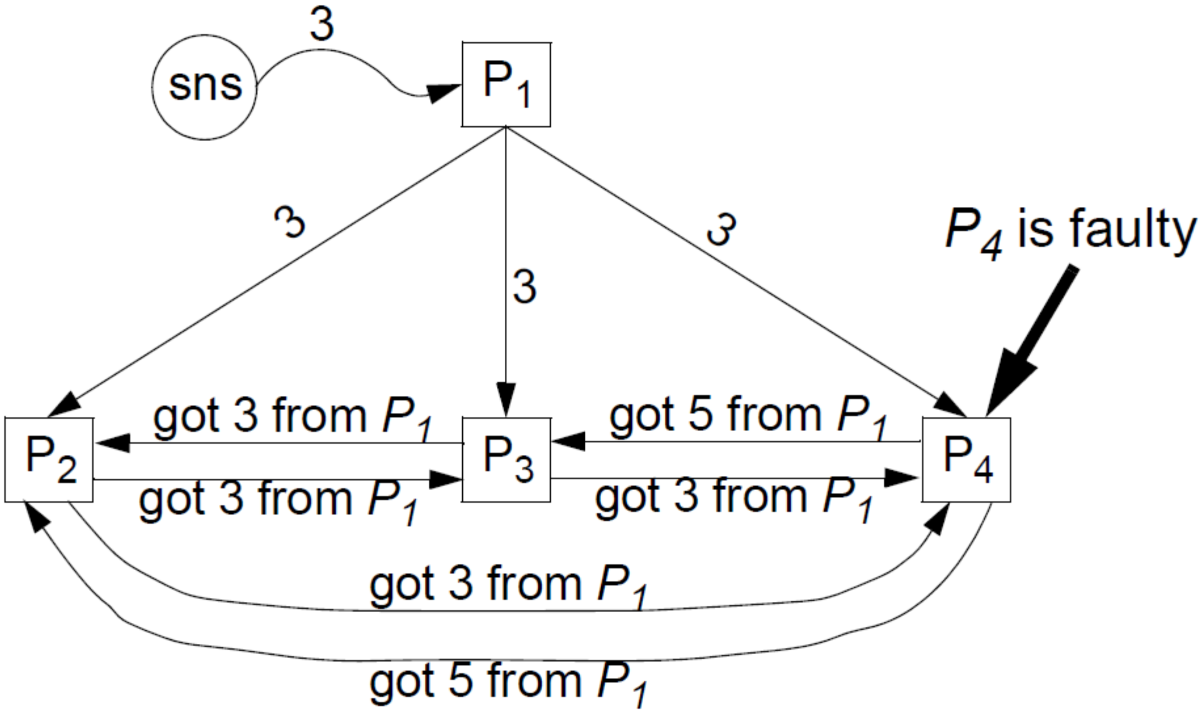
P_2, P_3, and P_4 will reach agreement on value 3, despite the faulty input unit P_1.



P_2, P_3, and P_4 will reach agreement on the default value, e.g. 0 (used when no majority exists), despite the faulty input unit P_1.

The Byzantine Generals Problem

Let us come back to our real-life example, this time with **four processors**:



The two non-faulty processors P_2 and P_3 agree on value 3, which is the value produced by the non-faulty input unit P_1.

Acknowledgments

- Most of the slide contents is based on a previous version by Petru Eles, IDA, Linköping University.