# 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
- control systems
- manufacturing systems

☞ What means correct functionality in the presence of faults?

The answer depends on the particular application (on the specification of the system):
- The system stops and doesn't 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:

1. Hardware fault: malfunction of a hardware component (processor, communication line, switch, etc.).
2. 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 (cont'd)

☞ Fault types according to their temporal behavior:

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 (cont'd)

☞ Fault types according to their output behaviour:

1. Fail-stop fault: either the processor is executing and can participate with correct values, or it has failed and will never respond to any request (see omission faults, Fö 2/3, slide 20). Working processors can detect the failed processor by a *time-out mechanism*.

2. Slowdown fault: it differs from the fail-stop model in the sense that a processor might fail and stop or it might execute slowly for a while ⇒ there is no time-out mechanism to make sure that a processor has failed; it might be incorrectly labelled as failed and we can be in trouble when it comes back (take care it doesn't come back unexpectedly).

3. Byzantine fault: 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 corrupt and has to be decided upon by a group of processors (see arbitrary faults, Fö 2/3, slide 21).

- 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.





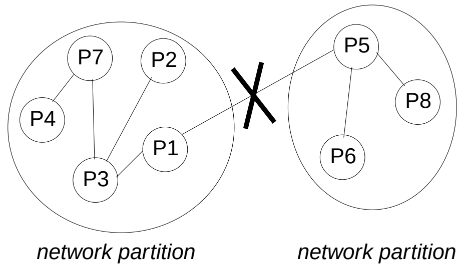
## Faults (cont'd)

☞ A fault type specifically related to the communication media in a distributed system:

- Partition Fault

  Two processes, which need to interact, are unable to communicate with each other because there exists no direct or indirect link between them ⇒ the processes belong to different *network partitions*.

  Partition faults can be due to:
  - broken communication wire
  - congested communication link.




A possible very dangerous consequence:
- Processes in one network partition could believe that there are no other working processes in the system.





## 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 are coded in such a way that a certain number of bit errors can be detected and, possibly, corrected (parity coding, checksum codes, cyclic codes).





## Time Redundancy and Backward Recovery

☞ The basic idea with *backward recovery* is to roll back the computation to a previous checkpoint and to continue from there.

☞ Essential aspects:
1. Save *consistent* states of the distributed system, which can serve as recovery points. Maintain replicated copies of data.
2. Recover the system from a recent recovery point and take the needed corrective action.

- Creating globally coherent checkpoints for a distributed systems is, in general, performed based on strategies similar to those discussed in Fö 5 for *Global States* and *Global State Recording*.

- For managing coherent replicas of data (files) see Fö 8.

- 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).

## Time Redundancy and Backward Recovery (cont'd)

Recovery in transaction-based systems

Transaction-based systems have particular features related to recovery:

☞ A *transaction* is a sequence of operations (that virtually forms a single step), transforming data from one consistent state to another.

Transactions are applied to *recoverable data* and their main characteristic is *atomicity*:

- *All-or-nothing* semantics: a transaction either completes successfully and the effects of all of its operations are recorded in the data items, or it fails and then has no effect at all.
  - *Failure atomicity*: the effects are atomic even when the server fails.
  - *Durability*: after a transaction has completed successfully all its effects are saved in permanent storage (this data survives when the server process crashes).
- Isolation: The intermediate effects of a transaction are not visible to any other transaction.





## Time Redundancy and Backward Recovery (cont'd)

☞ *Transaction processing implicitly means recoverability*:

- When a server fails, the changes due to all completed transactions must be available in permanent storage ⇒ the server can recover with data available according to *all-or-nothing* semantics.

☞ *Two-phase commitment*, *concurrency control*, and *recovery system* are the key aspects for implementing transaction processing in distributed systems.

See data-base course!





## Forward Recovery

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

☞ The basic fault model behind transaction processing and backward recovery is the fail-stop model

☞ 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. For such applications, which often are safety critical, the fail-stop model is not realistic.

⇩

*Forward recovery*: the error is masked without any computations having to be redone.

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





## Hardware Redundancy

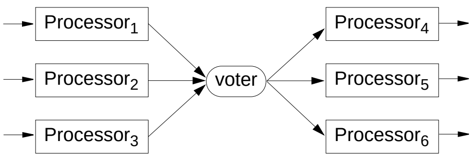
☞ Hardware redundancy is the use of additional hardware to compensate for failures:

- Fault detection, correction, and masking: multiple hardware units are assigned to the same task in parallel and their results compared.
  - Detection: if one or more (but not all) units are faulty, this shows up as a disagreement in the results (even byzantine faults can be detected).
  - Correction and masking: if only a minority of the units are faulty, and a majority of the units produce the same output, the majority result 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* (NMR) 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.

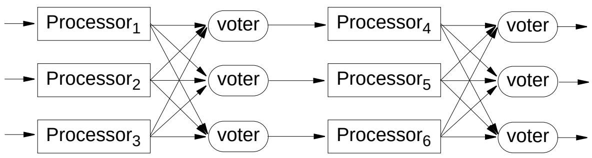
☞ *3-modular redundancy* is the most commonly used.



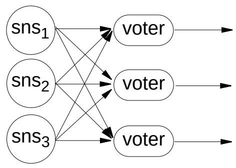


## *N*-Modular Redundancy (cont'd)

☞ The voter itself can fail; the following structure, with redondant voters, is often used:




☞ 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.

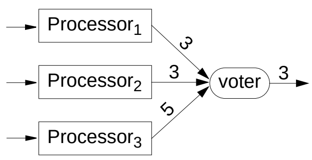
☞ The most common one: *majority voter*

- 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 $card(P_i) \geq \lceil N/2 \rceil$, then $x \in P_i$ is the correct output and the error can be masked.
  - if $card(P_i) < \lceil N/2 \rceil$, then the error cannot be masked (it has only been detected).





## Voters (cont'd)

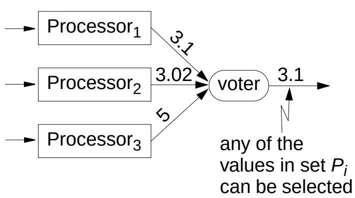



Sometimes we can not use strict equality:

- 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 $x = y$.

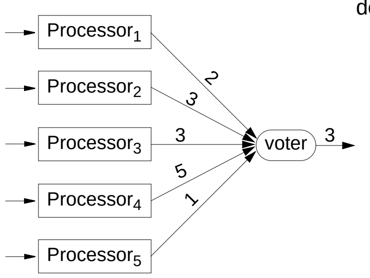


## Voters (cont'd)

Other voting schemes:

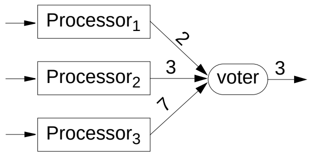
☞ *k-plurality voter*

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




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



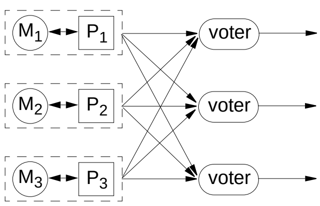


## Processor and Memory Level Redundancy

☞ *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 are handled as a unit and voting is on processor outputs:



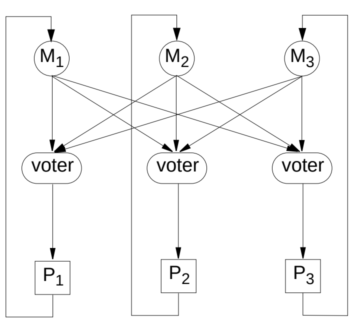





## Processor and Memory Level Redundancy

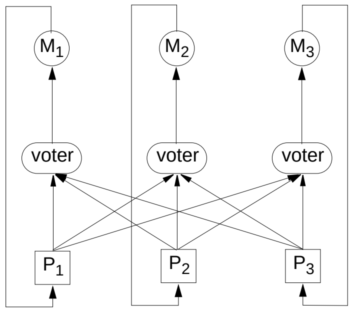
- Processors and memories can be handled as separate modules.

a) voting at read from memory

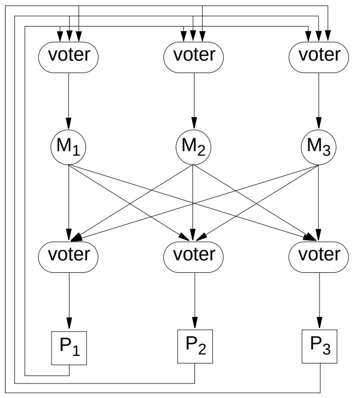


### Processor and Memory Level Redundancy (cont'd)

b) voting at write to memory




c) voting at read and write








## Software Redundancy

☞ There are several aspects which make software 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).

⇩

1. No software faults are produced by manufacturing, aging, stress, or environment.
2. Different copies of identical software always produce the same behavior 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.

☞ N *different versions* of the software are needed in order to provide redundancy.
Two possible approaches:

1. All *N* versions are running in parallel and voting is performed on the output.
2. Only one version is running; if it fails, another version is taking over after recovery.





### Software Redundancy (cont'd)

☞ The *N* versions of the software must be diverse ⇒ the probability that they fail on the same input has to be sufficiently small.

☞ It is very 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.





## Distributed Agreement with Byzantine Faults

☞ Very often it is the case that distributed processes have to come to an agreement.
For example, they have to agree on e certain value, with which each of them has to continue operation.

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

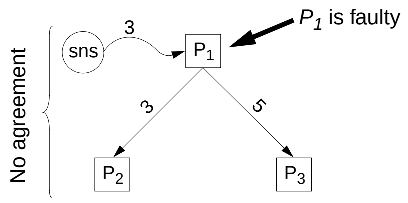
☞ Remember (slide 17): with a simple *voting scheme*, 2*k*+1 components are needed to achieve *k* fault tolerance in the case of byzantine faults ⇒ 3 processors are sufficient to mask the fault of one of them.

- *However, this is not the case for agreement* !

## Distributed Agreement with Byzantine Faults (cont'd)

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.

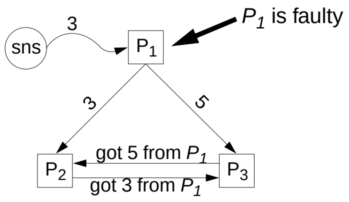



Maybe, by letting $P_2$ and $P_3$ communicate, they could get out of the trouble:

$P_2$ doesn't 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







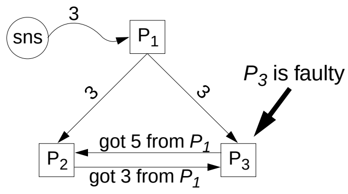

## Distributed Agreement with Byzantine Faults (cont'd)

The same if $P_3$ is faulty:

$P_2$ doesn't know if $P_1$ or $P_3$ is the faulty one, thus it cannot handle the contradicting inputs.

⇩

No agreement



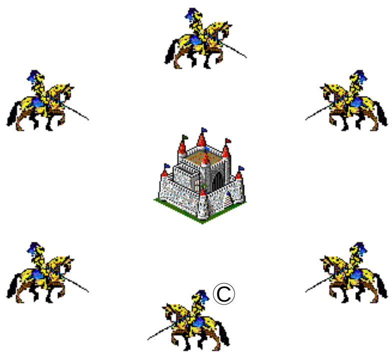

☞ With three processors we cannot achieve agreement, if one of them is faulty (with byzantine behaviour)!

☞ The *Byzantine Generals Problem* is used as a model to study agreement with byzantine faults





## The Byzantine Generals Problem




Picture by courtesy Minas Lamprou and Ioannis Psarakis

The story

- The byzantine army is preparing for a battle.
- A number of *generals* must coordinate among themselves through (reliable) messengers on weather to attack or retreat.
- A *commanding general* will make the decision whether or not to attack.
- **Any of the generals, including the commander, may be traitorous, in that they might send messages to *attack* to some generals and messages to *retreat* to others.**





## The Byzantine Generals Problem (cont'd)

☞ The problem in the story:

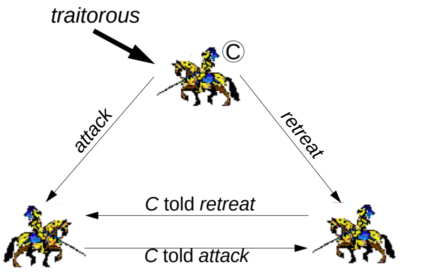
- The loyal generals have all to agree to attack, or all to retreat.
- If the commanding general is loyal, all loyal generals must agree with the decision that he made.

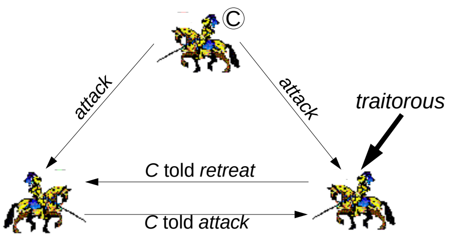
☞ The problem in real life (see slide 24):

- 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 (cont'd)

Let's see the case with three Generals (two Generals + the Commander): **No agreement is possible if one of three generals is traitorous**.



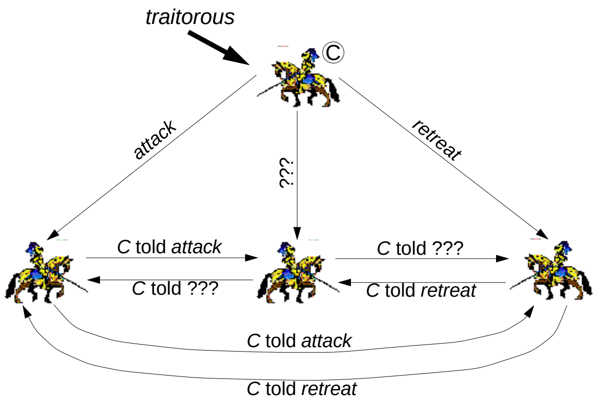








## The Byzantine Generals Problem (cont'd)

The case with four generals (three + the Commander):








## The Byzantine Generals Problem (cont'd)

Messages received at *Gen. left*: *attack*, ???, *retreat*.
Messages received at *Gen. middle*: ???, *attack*, *retreat.*
Messages received at *Gen. right*: *retreat*, ???, *attack.*

☞ The generals take their decision 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*.

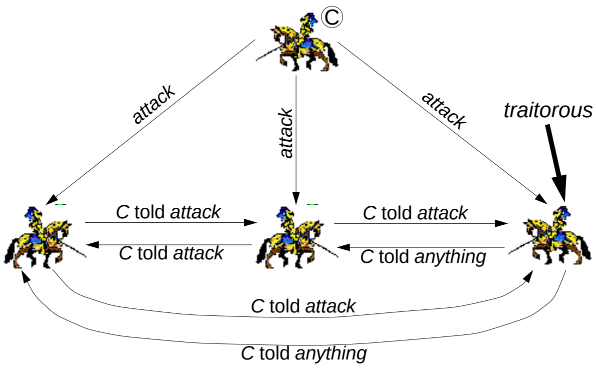
⇩

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





## The Byzantine Generals Problem (cont'd)




Messages received at *Gen. left*: *attack*. *attack*, *anything*.
Messages received at *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*), regardless the messages spread by the traitorous general.

## The Byzantine Generals Problem (cont'd)

The general result

To reach agreement, in the sense introduced on slide 27, with $k$ traitorous Generals requires a total of at least $3k + 1$ Generals.

⇩

**You need $3k + 1$ processors to achieve $k$ fault tolerance for agreement with byzantine faults**.

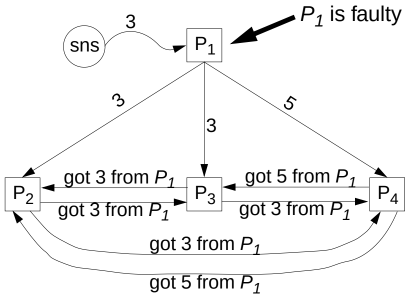
- To mask one faulty processor: total of 4 processors;
- To mask two faulty processor: total of 7 processors;
- To mask three faulty processor: total of 10 processors;
- \- - - - - - - - - - - - - - - - - - - - - - - -



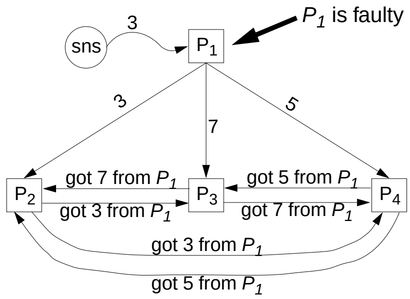


## The Byzantine Generals Problem (cont'd)

☞ Let's come back to our real-life example (slide 24), this time with four processors:




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

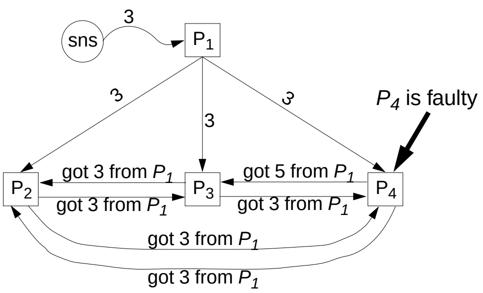



- $P_2$, $P_3$, and $P_4$ will reach agreement on the default value, e.g. 0 (used when no majority exists), despite the faulty input unit $P_1$.





## The Byzantine Generals Problem (cont'd)




☞ 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 $P_1$.





## <u>Summary</u>

- Several application areas need fault-tolerant systems. Such systems behave in a predictable manner, according to their specification, in the presence of faults.
- Faults can be hardware or software faults. Depending on their temporal behavior, faults can be permanent, intermittent, or transient.
- The fault model which is the easiest to handle is *fail-stop*; according to this model, faulty processors simply stop functioning. For real-life applications sometimes a more general fault model has to be considered. The *byzantine fault* model captures the behavior of processors which produce erroneous values and actively try to make computations fail.
- The basic concept for fault-tolerance is redundancy: time redundancy, hardware redundancy, software redundancy, and information redundancy.
- Backward recovery achieves fault-tolerance by rolling back the computation to a previous checkpoint and continuing from there.
- Backward recovery is typically used in transaction-based systems.

## Summary (cont'd)

- Several applications, mainly those with strong timing constraints, have to rely on forward recovery. In this case errors are masked without any computation having to be redone.
- Forward recovery is based on hardware and/or software redundancy.
- *N*-Modular redundancy is the basic architecture for forward recovery. It is based on the availability of several components which are working in parallel so that voting can be performed on their outputs.
- A system is *k* fault tolerant if it can survive faults in *k* components and still meet its specifications.
- Software redundancy is a particularly difficult and yet unsolved problem. The main difficulty is to produce different versions of the same software, so that they don't fail on the same inputs.
- The problem of reliable distributed agreement in a system with byzantine faults has been described as the *Byzantine generals problem*.
- 3*k* + 1 processors are needed in order to achieve distributed agreement in the presence of *k* processors with byzantine faults.