Part 6:

System aspects of dependable computers

System design considerations

Fault-tolerance is not the only means for dependability

To achieve given dependability goals the following aspects need consideration (usually in the order given here):

▪ perfection	*low*
▪ maintenance	↓
▪ fault-tolerance	*design complexity*
– systematic fault-tolerance	↓
– application-specific fault-tolerance	*high*

Perfection vs. fault-tolerance

- **Perfection is easier than fault-tolerance:**
  - if it is possible to attain a given dependability goal by means of perfection then use perfection in favor of fault-tolerance
  - perfection leads to conceptual simpler systems
  - lower probability of design faults
  - does not require error detection, damage confinement and assessment, error recovery and fault treatment to tolerate faults
  - steady reliability improvement of hardware components supports perfection
- **But, perfection is limited:**
  - perfection is limited by the dependability of individual components
  - very high dependability goals can only be reached by maintenance or by fault-tolerant systems

Maintenance vs. fault-tolerance

- **Maintenance is easier than fault-tolerance:**
  - if it is possible to attain a given dependability goal (availability) by means of maintenance then use maintenance in favor of fault-tolerance
  - maintenance adds to system complexity, but is still considerable simpler than fault-tolerance
  - lower probability of design faults
  - requires error detection and damage confinement, but no error recovery and fault treatment at system level
  - also trade off between maintenance and reliability (connector vs. solder joint)
- **But, maintenance is limited:**
  - maintenance is limited by the dependability of individual components
  - applicability of maintenance is limited (cf. next slide)

Limitations of maintenance

- maintenance (without fault-tolerance) is only applicable if system down times
  are permitted
  – fail-stop or fail-safe systems allow down times:
    (train signalling, anti-lock braking system, …)
  – fail-operational systems do not allow down times:
    (fly-by-wire, reactor safety system, …)
- only restricted reliability and safety improvements by preventive maintenance
- preventive maintenance is only reasonable if:
  – replacement units have constant or increasing failure rate
  – infant mortality is well controlled and failure rates are sufficiently low

The maintenance procedure

The maintenance procedure consists of the following phases:

- error detection
- call for maintenance
- maintenance personnel arrival
- diagnosis
- supply of spare parts
- replacement of defect components
- system test
- system re-initialization
- resynchronization with environment

System aspects of dependable computers

Aspects of maintenance

- maintenance costs vs. system costs
- error latency period, error propagation and error diagnosis
- maintenance personnel
  (number, education, equipment, location, service hours 0 to 24, ...)
- spare part supply, stock or shipment
- maintainability
  - documentation
  - test plans
  - system structure
  - error messages
  - size and interconnection of replacement units
  - accessibility of replacement units
  - mechanical stability of replacement units

Determining factors for SRU size

The size of the smallest replaceable unit (SRU) is determined by the following factors:

factor (increases)	*SRU size*
qualification of service personnel	decreases
effort for diagnosis	decreases
cost of SRU	increases
spare part costs[1]	increases
maintainability	increases
maintenance duration	decreases

[1]Cost for parts which are used to construct SRU’s

Diagnosis support for maintenance

- diagnosis support is very important
- self diagnosis with meaningful messages
- complete coverage of error domain
- maintenance documentation:
  - symptom $\rightarrow$ cause and affected SRU
  - error symptom/cause matrix indicates for each symptom all possible SRU’s that may cause the symptom
  - sparse matrices indicate good diagnosability
- expert system support for diagnosis
- duration of diagnosis is important for MTTR
- needs to be considered during system design

Fault-tolerance: systematic vs. application-specific

There are two fundamental aproaches to fault-tolerance:

- **Systematic fault-tolerance**
  - replication of components
  - divergence of components is used for fault-detection
  - redundant components are used for continued service
- **Application-specific fault-tolerance**
  - reasonableness checks for fault detection
    (based on model of real world)
  - state estimations for continued service

Application-specific fault-tolerance

- the computer system interacts with some physical process, the behavior of the process is constrained by the law of physics
- these laws are implemented by the computer system to check its state for reasonableness
- for example:
  - the acceleration/deceleration rate of an engine is constrained by the mass and the momentum that affects the axle
  - signal range checks for analog input signals
- reasonableness checks are based on application knowledge
- fail-stop behavior can be implemented based on reasonableness checks

Application-specific fault-tolerance (cont.)

- the laws of physics constraining the process can be used to perform state estimations in case some component has failed
- for example:
  - if the engine temperature sensor fails a simple state estimation could assume a default value
  - a better state estimation can be based on the ambient temperature of the engine, engine load and thermostatical behavior of the engine
  - the speed of a vehicle can be estimated if the engine speed and the transmission ratio is known
- state estimations are based on application knowledge
- fail-operational behavior can be implemented based on reasonableness checks and state estimations

Systematic fault-tolerance

- does not use application knowledge, makes no assumptions on the physical process or controlled object
- uses replicated components instead
- if among a set of replicated components, some—but not all—fail then there will be divergence among replicas
- information on divergence is used for fault detection
- replicas are therefore required to deliver corresponding results in the absence of faults
- The problem of **replica determinism**:
  due to the limited accuracy of any sensor that maps continuous quantities onto computer representable discrete numbers it is impossible to avoid nondeterministic behavior

Systematic fault-tolerance (cont.)

- systematic fault-tolerance requires agreement protocols due to replica nondeterminism
- the agreement protocol has to guarantee that correct replicas return corresponding results
  (the problem of replica determinism is discussed later)
- fail-stop behavior can be implemented by using the information of divergent results
- fail-operational behavior can be implemented by using redundant components

Comparison of fault-tolerance techniques

Systematic fault-tolerance	Application-specific fault-tolerance
• replication of components	• no replication necessary
• divergence among replicas in case of faults	• —
• no reasonableness checks necessary	• reasonableness checks for fault detection
• requires replica determinism	• —
• no application knowledge necessary	• depends on application knowledge
• exact distinction between correct and faulty behavior	• fault detection is limited by a *gray zone*

Comparison of fault-tolerance techniques (cont.)

Systematic fault-tolerance	Application-specific fault-tolerance
• no state estimations necessary	• state estimations for continued service
• independence of application areas	• missing or insufficient reasonableness checks for some application areas
• service quality is independent of whether replicated components are faulty or not	• quality of state estimations is lower than quality delivered during normal operation
• correct system function depends on the number of correct replicas and their failure semantics	• correct system function depends on the severity of faults and on the capability of reasonableness checks and state estimations
• only backward recovery	• forward and backward recovery

Comparison of fault-tolerance techniques (cont.)

Systematic fault-tolerance	Application-specific fault-tolerance
• additional costs for replicated components (if no system inherent replication is available)	• no additional costs for replicated components
• no increase in application complexity	• considerable increase in application complexity
• considerable increase of system level complexity	• no increase of system level complexity
• separation of fault-tolerance and application functionality	• application and fault-tolerance are closely intertwined
• fault-tolerance can be handled transparently to the application	• — // —

Systematic *and* application-specific fault-tolerance

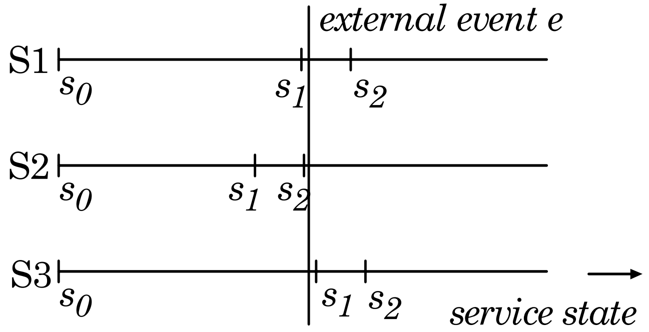
- under practical conditions there will be a compromise between systematic and application-specific fault-tolerance
- usually cost, safety and reliability are the determining factors to choose a proper compromise
- **software complexity** plays an important role:
  - for complex systems software is almost unmanageable without adding fault-tolerance
    (fault containment regions and software robustness)
  - therefore systematic fault-tolerance should be applied in favor of application-specific fault-tolerance to reduce the software complexity
  - systematic fault-tolerance allows to test and to validate the mechanisms independently of the application software (divide and conquer)

The problem of *Replica Determinism*

- For systematic fault-tolerance it is necessary that replicated components show consistent or deterministic behavior in the absence of faults
- If for example two active redundant components are working in parallel, both have to deliver corresponding results at corresponding points in time
- This requirement is fundamental to differentiate between correct and faulty behavior
- At a first glance it seems trivial to fulfill replica determinism since computer systems are assumed to be examples of deterministic behavior, but
- in the following it is shown that computer systems behave *almost* deterministically

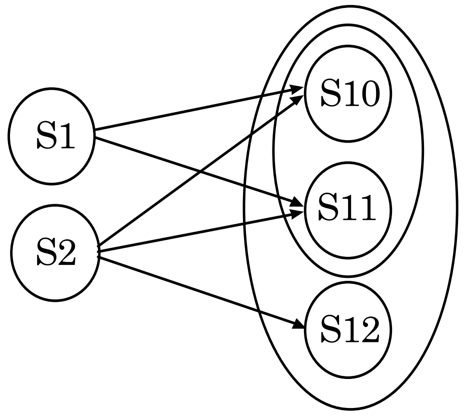
Nondeterministic behavior

- **Inconsistent inputs:**
  If inconsistent input values are presented to the replicas then the results may be inconsistent too.
  - a typical example is the reading of replicated analogue sensors
    read(S1) = 99.99 °C, *read*(S2) = 100.00 °C
- **Inconsistent order:**
  If service requests are presented to replicas in different order then the results will be inconsistent.



Nondeterministic behavior (cont.)

- **Inconsistent membership information:**
  Replicas may fail or leave groups voluntarily or new replicas may join a group.
  If replicas have inconsistent views about group membership it may happen that the results of individual replicas will differ.



Nondeterministic behavior (cont.)

- **Nondeterministic program constructs:**
  Besides intentional nondeterminism, like random number generators, some programming languages have nondeterministic program constructs for communication and synchronization (Ada, OCCAM, and FTCC).

```
task server is
   entry service_1();
   ...
   entry service_n();
end server;
```

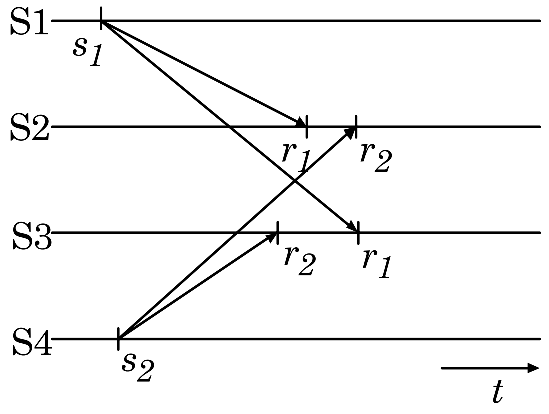
```
task body server is
begin
   select
      accept service_1() do
         action_1();
      end;
   ...
   or
      accept service_n() do
         action_n();
      end;
   end select;
end server;
```

Nondeterministic behavior (cont.)

- **Local information:**
  If decisions with a replica are based on local knowledge (information which is not available to other replicas) then the replicas will return different results.
  - system or CPU load
  - local time
- **Timeouts:**
  Due to minimal processing speed differences or due to slight clock drifts it may happen that some replicas locally decide to timeout while others do not.
- **Dynamic scheduling decisions:**
  Dynamic scheduling decides in which order a series of service requests are executed on one or more processors. This may cause inconsistent order due to:
  - non-identical sets of service requests
  - minimal processing speed differences

Nondeterministic behavior (cont.)

- **Message transmission delays:**
  Variabilities in the message transmission delays can lead to different message arrival orders at different servers (for point-to-point communication topologies or topologies with routing).



Nondeterministic behavior (cont.)

- **The consistent comparison problem:**
  - computers can only represent finite sets of numbers
  - it is therefore impossible to represent the real numbers exactly, they are rather approximated by equivalency classes
  - if the results of arithmetic calculations are very close to the border of equivalency classes, different implementations can return diverging results
  - different implementations are caused by: N-version programming, different hardware, different floating point libraries, different compilers
  - for example the calculation of $(a - b)^2$ with floating point representation with a mantissa of 4 decimal digits and rounding where $a = 100$ and $b = 0.005$ gives different result for mathematical equivalent formulas.

$$(a - b)^2 = 1.000\ 10^4$$

$$(a - b)^2 = a^2 - 2ab + b^2 = 9.999\ 10^3$$

Fundamental limitations to replication

- **The *real world abstraction limitation*:**
  - dependable computer systems usually interface with continuos real-world quantities:

quantity	*SI-unit*
distance	meter [*m*]
mass	kilogram [*kg*]
time	second [*s*]
electrical current	ampere [*A*]
thermodynamic temperature	degree kelvin [*K*]
gramme-molecule	mol [*mol*]
luminous intensity	candela [*cd*]

  - these continuos quantities have to be abstracted (or represented) by finite sets of discrete numbers
  - due to the finite accuracy of any interface device, different discrete representations will be selected by different replicas

Fundamental limitations to replication (cont.)

- **The *impossibility of exact agreement*:**
  - due to the real world abstraction limitation it is impossible to avoid the introduction of replica non-determinism at the interface level
  - but it is also impossible to avoid the once introduced replica nondeterminism by agreement protocols completely
  - exact agreement would require ideal simultaneous actions, but in the best case actions can be only simultaneous within a time interval Δ
- **Intention and missing coordination:**
  - replica nondeterminism can be introduced intentionally
  - or unintentionally by omitting some necessary coordinating actions

Replica control

Due to these fundamental limitations to replication it is **necessary** to enforce replica determinism which is called *replica control*.

Def.: *Replica Determinism*

Correct replicas show *correspondence* of service outputs and/or service states under the assumption that all servers within a group start in the same initial state, executing *corresponding* service requests within a *given time interval*.

- this generic definition covers a broad range of systems
- *correspondence* and within a *given time interval* needs to be defined according to the application semantics

Synchronous vs. asynchronous systems

- **Synchronous processors:**
  A processor is said to be *synchronous* if it makes at least one processing step during Δ real-time steps (or if some other computer in the system makes *s* processing steps).
- **Bounded communication:**
  The communication delay is said to be *bounded* if any message sent will arrive at its destination within Φ real-time steps (if no failure occurs).
- **Synchronous system:**
  A system is said to be synchronous if its processors are synchronous **and** the communication delay is bounded. Real-time systems are per definition synchronous.
- **Asynchronous systems:**
  A system is said to be asynchronous if either its processors are asynchronous **or** the communication delay is unbounded.

Groups, resiliency and replication level

- Replicated entities such as processors are called **groups**.
- The number of replicas in a group is called **replication level**
- A group is said to be ***n*-resilient** if up to *n* processor failures can be tolerated

Group vs. hierarchical failure masking

- **Group failure masking:**
  The group output is a function of the individual group members output (e.g. a majority vote, a consensus decision). Thus failures of group members are hidden from the service user.
- **Hierarchical failure masking:**
  The processors within a group come up with diverging results and the faults are resolved by the service user one hierarchical level higher.

Internal vs. external replica control

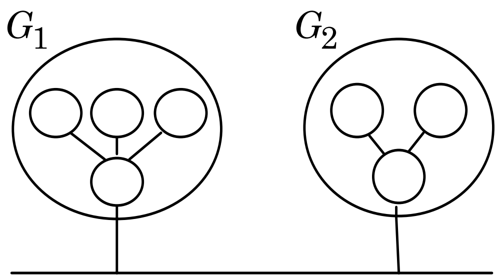
- **Internal replica control:**
  - avoid nondeterministic program constructs, uncoordinated timeouts, dynamic scheduling decisions, diverse program implementations, local information, and uncoordinated time services
  - can only be enforced partially due to the fundamental limitations to replication
- **External replica control:**
  - control nondeterminism of sensor inputs
  - avoid nondeterminism introduced by the communication service
  - control nondeterminism introduced by the program execution on the replicated processors by exchanging information

The basic services for replicated fault-tolerant systems

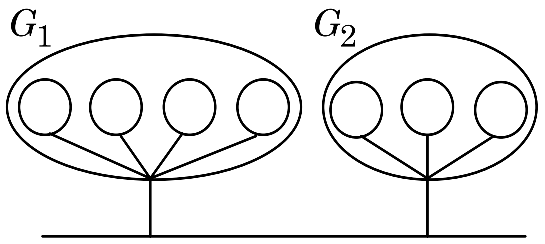
- **Membership:**
  Every non-faulty processor within a group has timely and consistent information on the set of functioning processors which constitute the group.
- **Agreement:**
  Every non-faulty processor in a group receives the same service requests within a given time interval.
- **Order:**
  Explicit service requests as well as implicit service requests, which are intro-duced by the passage of time, are processed by non-faulty processors of a group in the same order.

Central vs. distributed replica control

- **Strictly central replica control:**
  - – there is one distinguished processor within a group called *leader* or *central processor*
  - – the leader takes all nondeterministic decisions
  - – the remaining processors in the group, called *followers*, take over the leaders decisions



- **Strictly distributed replica control:**
  - – there is no leader role, each processor in the group performs exactly the same way
  - – to guarantee replica determinism the group members have to carry out a consensus protocol on nondeterministic decisions



The rationale for communication services

The following arguments motivate the close interdependence of fault-tolerant computer systems, communication and replica control:

- fault-tolerant systems are built on the assumption that individual components fail independently
- this assumptions requires the physical and electrical isolation of components at the hardware level
- these properties are best fulfilled by a distributed computer system where nodes are communicating through message passing but have no shared resources except for the communication media
- furthermore it has to be guaranteed that faulty nodes are not able to disturb the communication of correct nodes and that faulty nodes are not allowed to contaminate the system

Basic communication services for fault-tolerance

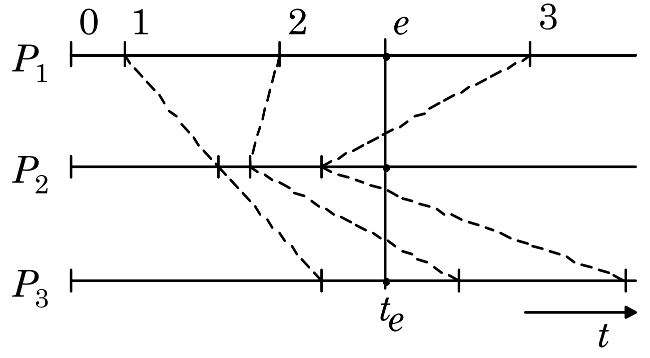
- **Distributed replica control:**
  Any (partially) distributed replica control strategy requires a communication service assuring consensus.
- **Consensus:**
  Each processor starts a protocol with its local input value, which is sent to all other processors in the group, fulfilling the following properties:
  - **Consistency:**
    All correct processors agree on the same value and all decisions are final.
  - **Non-triviality:**
    The agreed-upon input value must have been some processors input (or is a function of the individual input values).
  - **Termination:**
    Each correct processor decides on a value within a finite time interval.

Basic communication services for fault-tolerance
(cont.)

- **Central replica control:**
  Strictly central replica control requires a communication service assuring reliable broad- or multicast.
- **Reliable broadcast:**
  A distinguished processor, called the transmitter, sends its local service request to all other processors in the group, fulfilling the following properties:
  - **Consistency:**
    All correct processors agree on the same value and all decisions are final.
  - **Non-triviality:**
    If the transmitter is non faulty, all correct processors agree on the input value sent by the transmitter.
  - **Termination:**
    Each correct processor decides on a value within a finite time interval .

Clock synchronization

- **Logical clocks:**
  - all members in a group observe the same events in the same order
  - this applies to process internal events and external events such as service requests and faults
  - for all events e_i and e_j, if $e_i \rightarrow e_j$ then $LC(e_i) < LC(e_j)$ where $\rightarrow$ is the *happened before* relation as defined by Lamport
  - external events need to be reordered according to the internal precedence relation and individual processing speeds



Logical clocks

- The *happend before* relation ($\rightarrow$) defines a *partial order* of events and captures *potential causality*, but excludes external clandestine channels

- **Def.:** The relation $\rightarrow$ on a set of events in a distributed system is the smallest relation satisfying the following three relations:

  (1) If *a* and *b* are events performed by the same process, and *a* is performed before *b* then $a \rightarrow b$.
  (2) If *a* is the event of sending of a message by one process and *b* the receiving of the same message by another process, then $a \rightarrow b$.
  (3) Transitivity: if $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.

- Two distinctive events are said to be *concurrent* if neither $a \rightarrow b$ nor $b \rightarrow a$.

- An algorithm implementing logical clocks:

  (1) Each process P_i increments LC_i between two successive events
  (2) Upon receiving a message *m*, a process P_j sets $LC_j = max(LC_j, T_m)$, where T_m is message *m*'s timestamp.

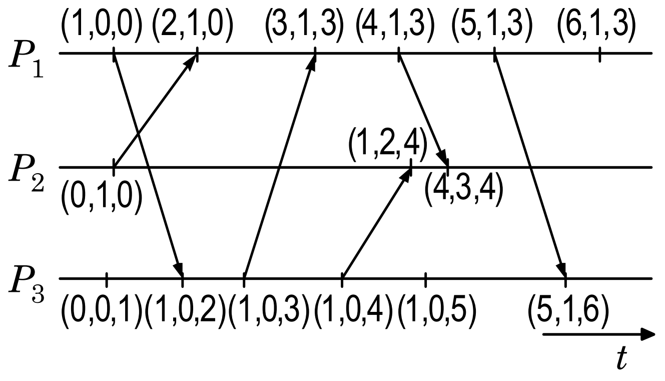
Logical clocks (cont.)

- the algorithm defines no total order since independent processes may use the same timestamp for different events
- a possible solution is to break ties by using a lexicographical process order
- logical clocks have no *gap-detection* property
- **Gap-detection:**
  Given two events *e* and *e'* with clock values $LC(e) < LC(e')$, determine whether some other event *e''* exists such that $LC(e) < LC(e'') < LC(e')$.
- the gap-detection property is necessary for *stability* and a bounded *action delay*, i.e., before an action is taken it has to be guaranteed that no earlier messages are delivered
- stability and action delay are based on *potential causality*, two events *e* and *e'* are potential causal related if $e \rightarrow e'$.

Vector clocks

- vector clocks are an extension of logical clocks which have gap-detection property
- An algorithm implementing vector clocks:

  (1) Each process P_i increments VC_i between two successive events
  (2) Upon receiving a message *m*, a process P_j sets all vector elements VC_j to the maximum of VC_j and T_m, where T_m is message *m*'s vector clock timestamp. Afterwards the element $VC_j[j]$ is incremented.

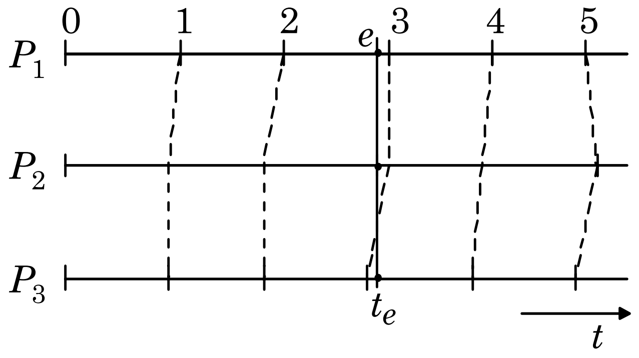


- Potential causality for vector clocks $e \rightarrow e' \equiv VC(e) < VC(e')$

  VC < *VC'* ≡ (*VC* ° *VC'*) ∧ (∀*i*:1 Š *i* Š *n*: *VC*[*i*] Š *VC'*[*i*])

Real-time clocks:

- all processors have access to a central real-time clock or
- all processors have local real-time clocks which are approximately synchronized
- the synchronized clocks define a global time grid where individual clocks are off at most by one tick at any time instant *t*
- the maximum deviation among clocks is called *precision*
- *t*-precedent events (events that are at least *t* real-time steps apart) can be causally related regardless of clandestine channels



Comparing real-time and logical clocks

real-time clocks	logical clocks
synchronous system model	asynchronous system model
higher synchronization overhead	little delays and synchronization overhead if only system internal events are considered
needs to achieve consensus on the systematic clock error of one tick	external events need to be reordered in accordance to logical time
stability within one clock tick	unbounded duration for stability, requires consistent cut or vector clock
potential causality for *t*-precedent external events	potential causality only for closed systems
bounded action delay (total order)	unbounded action delay (no total order)

Replica control strategies

- **Lock-step execution:**
  - processors are executing in synchronous
  - the outputs of processors are compared after each single operation
  - typically implemented at the hardware level with identical processors
- **Advantages:**
  - arbitrary software can be used without modifications for fault-tolerance (important for commercial systems)
- **Disadvantages:**
  - common clock is single point of failure
  - transient faults can affect all processors at the same point in the computation
  - high clock speed limits number and distance of processors
  - restricted failure semantics

Replica control strategies (cont.)

- **Active replication:**
  - all processors in the group are carrying out the same service requests in parallel
  - strictly distributed approach, nondeterministic decisions need to be resolved by means of an agreement protocol
  - the communication media is the only shared resource
- **Advantages:**
  - unrestricted failure semantics
  - no single point of failure
- **Disadvantages:**
  - requires the highest degree of replica control
  - high communication effort for consensus protocols
  - problems with dynamic scheduling decisions and timeouts

Replica control strategies (cont.)

- **Semi-active replication:**
  - – intermediate approach between distributed and centralized
  - – the leader takes all nondeterministic decisions
  - – the followers are executing in parallel until a potential nondeterministic decision point is reached
- **Advantages:**
  - – no need to carry out a consensus protocol
  - – lower complexity of the communication protocol (compared to active replication)
- **Disadvantages:**
  - – restricted failure semantics, the leaders decisions are single points of failures
  - – problems with dynamic scheduling decisions and timeouts

Replica control strategies (cont.)

- **Passive replication:**
  - – only one processor in the group – called *primary* – is active
  - – the other processors in the group are in *standby*
  - – checkpointing to store last correct service state and pending service requests
- **Advantages:**
  - – requires the least processing resources
  - – standby processors can perform additional tasks
  - – highest reliability of all strategies (if assumption coverage = 1)
- **Disadvantages:**
  - – restricted failure semantics (crash or fail-stop)
  - – long resynchronization delay

Failures and replication

- **Centralized replication:**
  - – semi-active and passive replication
  - – the leading processor is required to be fail restrained
  - – byzantine or performance failures of the leader cannot be detected by other processors in the group (“heartbeat” or “I am alive” messages)
  - – to tolerate *t* failures with crash or omission semantics *t* + 1 processors are necessary
  - – the result of any processor (e.g. the fastest) can be used
  - – if no reliable broadcast service is available 2*t* + 1 processors are necessary

Failures and replication (cont.)

- **Distributed replication:**
  - – active replication
  - – no restricted failure semantics of processors
  - – to tolerate *t* crash or omission failures *t* + 1 processors are necessary
  - – to tolerate *t* performance failures 2*t* + 1 processors are necessary
  - – to tolerate *t* byzantine failures 3*t* + 1 processors are necessary
  - – for crash or omission failures it is sufficient to take 1 processor result
  - – for performance or byzantine failure *t* + 1 identical results are required

Failure recovery

- After occurrence of a failure (that is covered by the fault hypothesis) the group has to perform some recovery actions
- **Centralized replication:**
  - failures of followers require no recovery actions
  - if a leader fails a new leader needs to be elected
  - then the new leader has to take over the service of the failed leader
  - typically solved by backward recovery (reexecution from last fault free state)
  - recovery time needs to be considered for real-time services
  - window of vulnerability where new leader cannot decide whether the last output was made successfully or not
  - output devices typically require at least once semantics (state semantics)

Failure recovery (cont.)

- **Distributed replication:**
  - no special recovery actions necessary since all services are executed in parallel
  - no election in case of processor failures
  - output devices have to consider the results of all group members or each group member has its own output device (idempotence)
  - no state semantics for output devices necessary (exactly once semantics possible)

Redundancy preservation

- to guarantee fault-tolerance and to cover the fault hypothesis the replication level has to be kept above a given threshold
- assuming n processors are in a group where f have failed and up to t failures have to be tolerated then one of the following combining conditions needs to be satisfied:

  $n - f > 2t$ for byzantine failures

  $n - f > t$ for performance failures

  $n - f > 0$ for crash or omission failures
- if this combining condition is violated
  – a new processor needs to be added to the group (redundancy preservation)

  – or the service of the group has to be abandoned

System aspects of dependable computers

Redundancy preservation (cont.)

- real-time requirements for redundancy preservation need special consideration
- faults in the reconfiguration service need to be considered
- *f* is therefore the number of failed processors plus the number of correct processors that are configured by a faulty reconfiguration service
- this requires a membership protocol:
  - – detect departures and joins of processors to groups
  - – provide consistent and timely group membership information on a system wide basis
- joins of processors are difficult to handle:
  - – the new processor needs to be synchronized to the service state of the group
  - – but the groups service state is evolving over time
  - – after synchronization it has to be guaranteed that all further service requests are delivered to the new group member as well

Coverage vs. Complexity

- **High assumption coverage implies high complexity**
  - for byzantine faults the assumption coverage is 1
  - byzantine faults require consensus protocols and very complex fault-tolerance mechanisms
  - high probability of faults in the fault-tolerance mechanisms (35% ESS-1)
  - due to the high complexity the system will have a low dependability
- **Low assumption coverage implies low dependability**
  - low assumption coverage implies high possibility of assumption violations
  - in case of assumption violations a fault-tolerant system can fail completely
  - the system will therefore have a low dependability
- for optimal dependability a compromise between the assumption coverage rate and complexity of the fault-tolerance mechanism has to be made

System aspects of dependable computers

Hardware vs. Software

- hardware components are more reliable compared to software components
- very mature technology for hardware process validation
- but: “build it in hardware instead” is no solution at all since the problem of design dependability arises because of the system inherent complexity:
  - very complex systems are realized in software because of theire complexity
  - software is often used to implement radically new systems
- higher flexibility of software is often exploited by very short modification cycles
- Outages in % by fatal faults for the *Tandem* system illustrates the shift from hardware to software:

year	*1985*	*1987*	*1989*
software	34%	39%	62%
hardware	29%	22%	7%
maintenance	19%	13%	5%
operation	9%	12%	15%

Consensus

- Each processor starts a protocol with its local input value, which is sent to all other processors in the group, fulfilling the following properties:

  Consistency: All correct processors agree on the same value and all decisions are final.

  Non-triviality: The agreed-upon input value must have been some processors input (or is a function of the individual input values).

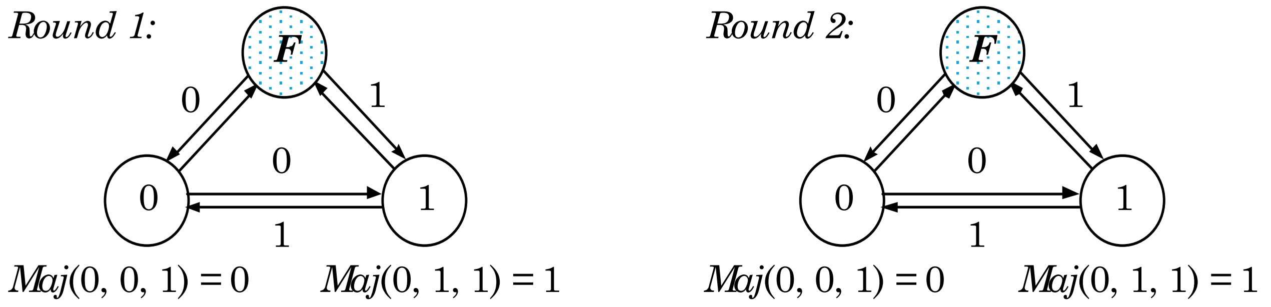
  Termination: Each correct processor decides on a value within a finite time interval.

- The consensus problem under the assumption of byzantine failures was first defined in 1980 in the context of the SIFT project which was aimed at building a computer system with ultra-high dependability. Other names are
  - byzantine agreement or byzantine general problem
  - interactive consistency

Impossibility of deterministic consensus in asynch. systems

- asynchronous systems cannot achieve consensus by a deterministic algorithm in the presence of even one crash failure of a processor
- it is impossible to differentiate between a late response and a processor crash
- by using *coin flips*, probabilistic consensus protocols can achieve consensus in a constant expected number of rounds
- *failure detectors* which suspect late processors to be crashed can also be used to achieve consensus in asynchronous systems

$n \geq 3t + 1$ processors are necessary to tolerate t failures



An interactive consistency algorithm

- assumptions about the message passing system:

  A1: Every message that is sent by a processor is delivered correctly by the message passing system to the receiver
  A2: The receiver of a message knows which node has sent a message
  A3: The absence of messages can be detected

- the recursive algorithm for n □ $3t + 1$:

***ICA*(*t*):**

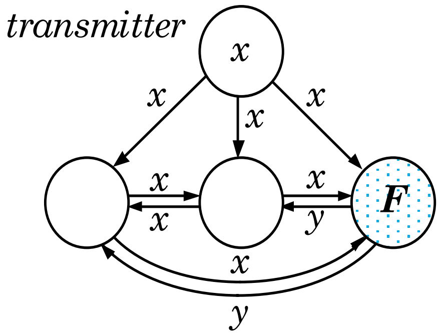
1. The transmitter sends its value to all the other $n - 1$ processors.
2. Let v_i be the value that processor i receives from the transmitter, or else be the default value if it receives no value. Node i acts as the transmitter in algorithm *ICA*($t - 1$) to send the value to each other of the other $n - 2$ receivers.
3. For each processor i, let v_j be the value received by processor j (j ° i). Processor i uses the value *Majority*($v_1, \dots, v_{n-1}$).

***ICA*(0):**

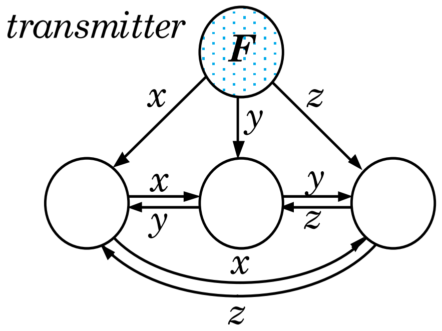
1. The transmitter sends its value to all the other $n - 1$ processors.
2. Each processor uses the value it receives from the transmitter, or uses the default value, if it receives no value.

An example

- there are 4 processors (*n* = 4)
- among them at most one is faulty (*t* = 1)
- the condition *n* ≥ 3*t* + 1 is satisfied
- **Case 1:**
  - one of the receivers is faulty
  - all correct processors decide *x*



- **Case 2:**
  - the transmitter is faulty
  - depending on the majority function all processors decide either *x*, *y* or *z*



Interactive consistency with signed messages

- if a processor sends *x* to some processor it appends its signature, denoted $x : i$
- when some processor receives this message and passes it further then $x : i : j$
- the algorithm for $n \geq t + 1$
- V_i is the set of all received messages which is initially $V_i = 0$

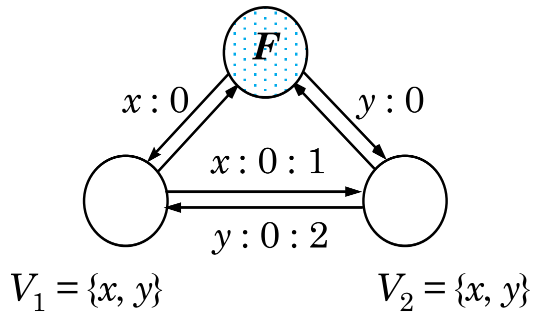
***SM*(*t*):**

1. The transmitter signs its value and sends it to all other nodes
2. $\forall i$:
   (A) If processor *i* receives a message of the form $v : 0$ from the transmitter then (i) it sets $V_i = \{v\}$, and (ii) it sends the message $v : 0 : i$ to every other processor.
   (B) If processor *i* receives a message of the form $v : 0 : j_1 : j_2 : \ldots : j_k$ and *v* is not in V_i, then (i) it adds *v* to V_i, and (ii) if $k < t$ it sends the message $: 0 : j_1 : j_2 : \ldots : j_k : i$ to every other node processor than $j_1, j_2, \ldots , j_k$.
3. $\forall i$: when processor *i* receives no more messages, it considers the final value as *Choice*(V_i).

- The function *Choice*(V_i) selects a default value if $V_i = 0$, it selects *v* if $V_i = \{v\}$ in other cases it could select a median or some other value.

An example

- there are 3 processors (n = 3)
- among them at most two can be faulty (t = 2)
- the condition $n \geq t + 1$ is satisfied
- we again consider the case of the faulty transmitter:



- because of the signed messages it becomes clear that the transmitter is faulty

Complexity of consensus

- *ICA*(t) and *SM*(t) require t + 1 rounds of message exchange
- t + 1 rounds are optimal in the worst case, the lower bound for early stopping algorithms is min(f + 2, t + 1)
- for *ICA*(t) the number of messages is exponential in t, since (n – 1)(n – 2) … (n – t – 1) are required $O(n^t)$, similarly the message complexity for *SM*(t) is exponential
- the lower bound is $O(nt)$, for authentification detectable byzantine failures, performance or omission failures the lower bound is $O(n + t^2)$
- practical experience has shown that the complexity and resource requirements
  of consensus under a byzantine failure assumption are often prohibitive
  (up to 80% overhead for SIFT project)

Reliable broadcast

- A distinguished processor, called the transmitter, sends its local service request to all other processors in the group, fulfilling the following properties:

  Consistency: All correct processors agree on the same value and all decisions are final.

  Non-triviality: If the transmitter is non faulty, all correct processors agree on the input value sent by the transmitter.

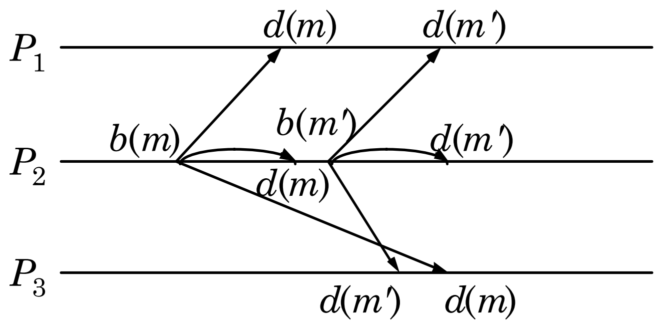
  Termination: Each correct processor decides on a value within a finite time interval.

- Reliable broadcast is a building block for the solution of a broad class of problems in fault-tolerant computer systems
- Often there are additional requirements to reliable broadcast protocols (cf. next slides)

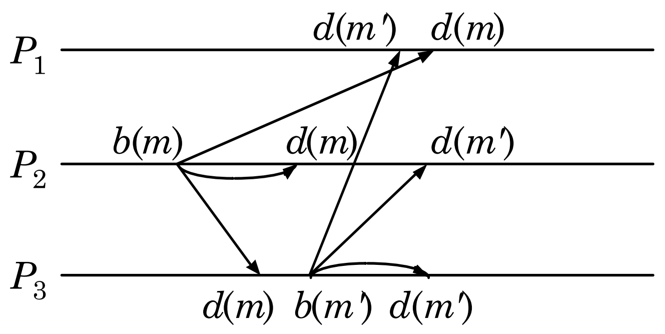
FIFO Broadcast

- FIFO Broadcast = Reliable Broadcast + FIFO order
- **FIFO Order:** If a process broadcasts *m* before it broadcasts *m’*, then no correct process delivers *m’* unless it has previously delivered *m*.

non-FIFO Broadcast



Problem with FIFO Broadcast

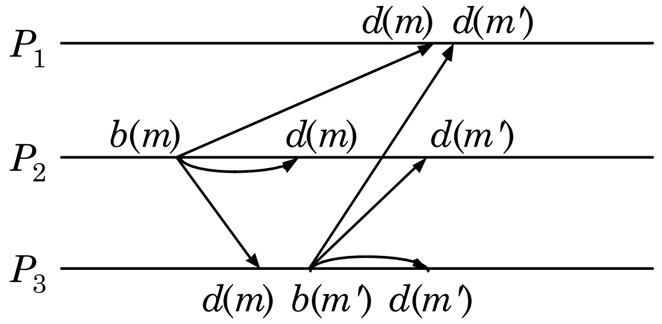


$b(m)$ … broadcast message m $d(m)$ … deliver message m

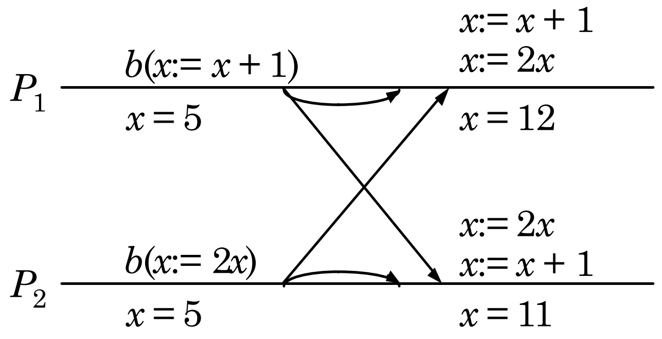
Causal Broadcast

- Causal Broadcast = Reliable Broadcast + Causal order
- **(Potential) Causal Order:** If the broadcast of *m* causally (→) precedes the broadcast *m'*, then no correct process delivers *m'* unless it has previously delivered *m*.

Causal Broadcast



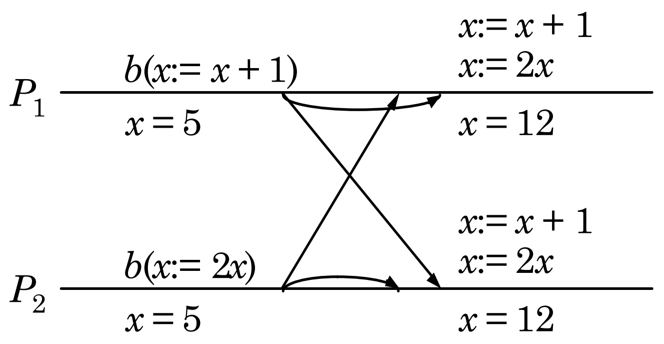
Problem with Causal Broadcast



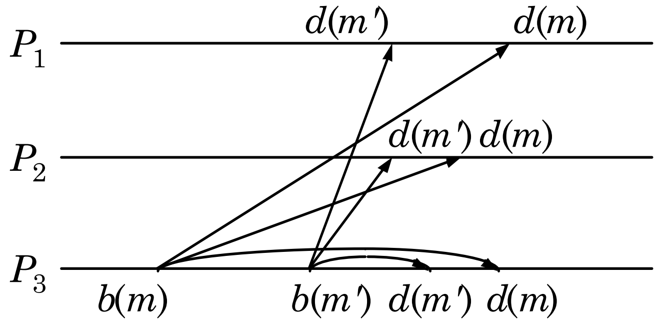
Atomic Broadcast

- Atomic Broadcast = Reliable Broadcast + Total order
- **Total Order:** If correct processes P_1 and P_2 deliver *m* and *m'*, then P_1 delivers *m* before *m'* if and only if P_2 delivers *m* before *m'*.

Atomic Broadcast

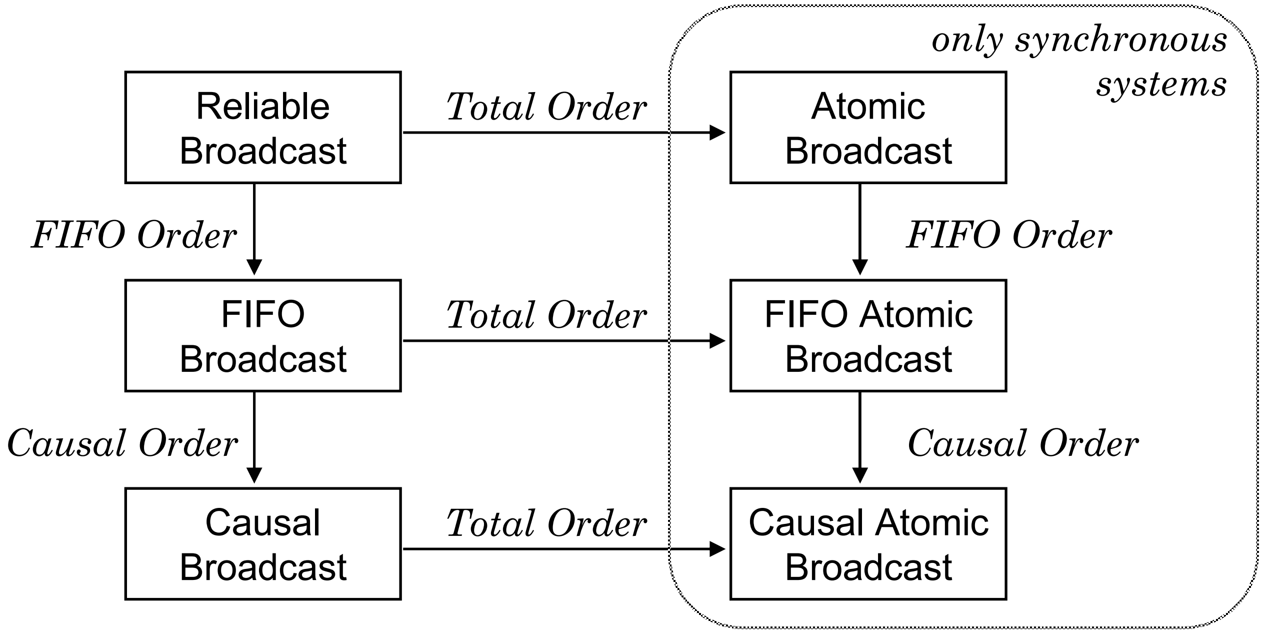


Atomic Broadcast is not FIFO



Extensions of Atomic Broadcast

- FIFO Atomic Broadcast = Reliable Broadcast + FIFO Order + Total Order
- Causal Atomic Broadcast = Reliable Broadcast + Causal Order + Total Order
- Relationships among broadcast protocols:



Reliable broadcast

- **Diffusion algorithm:** To R-broadcast *m*, a process *p* sends *m* to itself. When a process receives *m* for the first time it relays *m* to all its neighbors, and then R-delivers it.

```
broadcast(R, m):
    send(m) to p

deliver(R, m):
   upon receive(m) do
      if p has not previously executed deliver(R, m)
      then
         send(m) to all neighbors
         deliver(R, m)
```

- in synchronous systems the diffusion algorithm may be used as well, but it additionally guarantees real-time timeliness

Atomic broadcast

- **Transformation:** any {Reliable, FIFO, Causal} Broadcast algorithm that satisfies real-time timeliness can be transformed to {Atomic, FIFO Atomic, Causal Atomic} Broadcast.

```
broadcast(A*, m):
    broadcast(R*, m)

deliver(A*, m):
    upon deliver(R*, m) do
        schedule deliver(A*, m) at time TS(m) + Δ
```

- *TS*(*m*) is the timestamp of message *m*
- the maximum delay for message transmission is Δ
- if two messages have the same timestamp then ties can be broken arbitrarily, e.g. by increasing sender id's

FIFO and Causal Broadcast

- **FIFO Transformation:** Reliable broadcast can be transformed to FIFO broadcast by using sequence numbers.
- **Causal Transformation:** All messages that are delivered between the last broadcast and this send operation are “piggy-packed” when sending a message.

```
broadcast(C, $m$):
    broadcast(F, $\langle rcntDlvrs \,||\, m \rangle$)
    $rcntDlvrs$:= $\perp$

deliver(C, –):
    upon deliver(F, $\langle m_1, m_2, \dots m_l \rangle$) do
        for $i$:= 1 .. $l$ do
            if $p$ has not previously executed deliver(C, $m_i$)
            then
                deliver(C, $m_i$)
                $rcntDlvrs$:= $rcntDlvrs \,||\, m_i$
```

- *rcntDlvrs* is the sequence of messages that *p* C-delivered since its previous C-broadcast

Broadcast protocol implementations

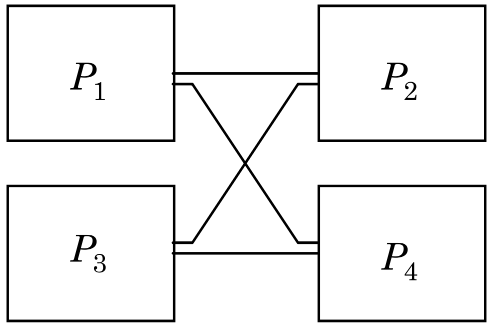
- Asynchronous systems: broadcast protocols are typically based on positive acknowledgment and retransmit (PAR) schemes and on message diffusion
- Synchronous systems: broadcast protocols are clock-driven, the protocol progression is guaranteed by the advance of time

Complexity of broadcast

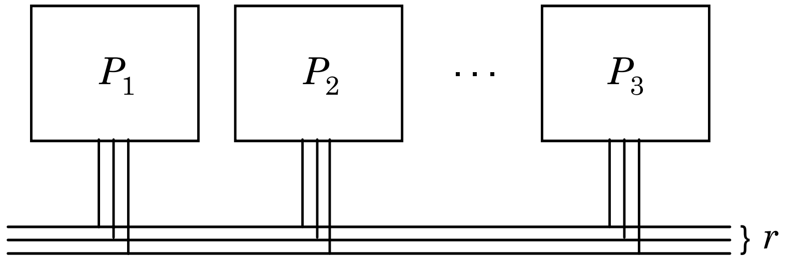
- to tolerate t byzantine failures $n > 3t + 1$ processors are required
- a connectivity $k > 2t$ is required to tolerate t byzantine failures, $k > t$ to tolerate more benign failures
- for synchronous systems at least $\min(f + 2, t + 1)$ rounds are required, where f is the actual number of failures ($t > f$).

Broadcast topologies

- up till now only point-to-pint topologies have been considered:



- the close resemblance between broadcast communication and broadcast topologies, however, allows a substantial reduction of the communication complexity



Broadcast topologies (cont.)

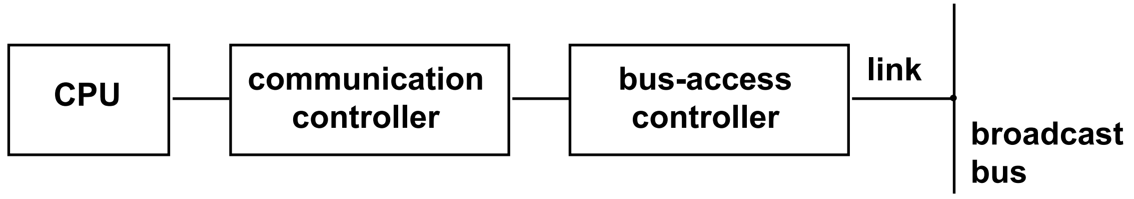
- communication channels for Ethernet, CAN, TTP, ... have broadcast properties
- **Broadcast degree *b*:** A channel has broadcast degree *b* if it echoes a message to at least *b* receivers or none of them.

Complexity revisited

- for byzantine failures 2 rounds are sufficient if $b > t + n/2$ (instead of $t + 1$)
- for omission failures 2 rounds are sufficient if $b > t$ and $t - b + 3$ rounds for $2 \leq b < t$
- if $b = n$ then 1 round is sufficient
- for space and time redundancy messages can be transmitted over *r* redundant channels *f* times (e.g. TTP)

Communication topologies and independence assumption

- **Point-to-point topology:**
  - – fault-tolerance requires that individual components fail independently
  - – a failure of one link has no effect on other links
  - – but, high cost and complexity: $n(n-1)$ links and transmitters for n nodes
- **Broadcast topology:**
  - – failures in the value domain can be handled easily by checksums and CRC's
  - – failures in the time domain can be common mode failures (*babbling idiot*)
  - – to prevent common mode failures by babbling idiots, the communication pattern needs to be highly regular (e.g. fixed time-slices)



Backward or rollback recovery

- systematic fault-tolerance is often based on backward recovery to recover a consistent state
- in distributed systems a state is said to be *consistent* if it *could* exist in an execution of the system
- **Recovery line:** A set of recovery points form a consistent state–called recovery line–if they satisfies the following conditions:

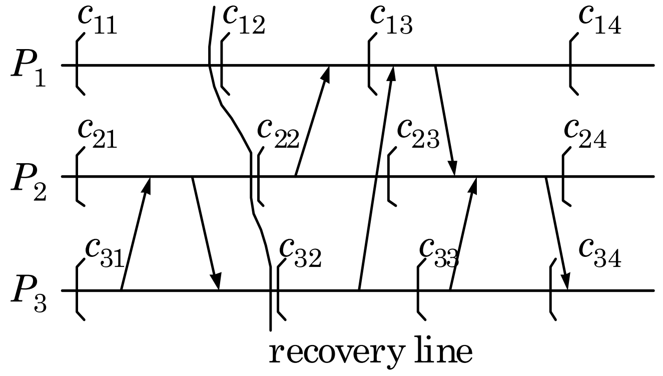
  (1) the set contains exactly one recovery point for each process

  (2) No **orphan** messages: There is no receive event for a message *m* before process P_i's recovery point which has not been sent before process P_j's recovery point.

  (3) No **lost** messages: There is no sending event for a message *m* before process P_i's recovery point which has not been received before process P_j's recovery point.

The domino effect

- the consistency requirement for recovery lines can cause a flurry of rollbacks to recovery points in the past



- to avoid the domino effect:
  – coordination among individual processors for checkpoint establishment
  – restricted communication between processors

A distributed checkpointing and rollback algorithm

- this protocol allows lost messages
- there are two kinds of checkpoints:
  - **permanent:** they cannot be undone
  - **tentative:** they can be undone or changed to permanent
- the checkpointing algorithm works in two phases:

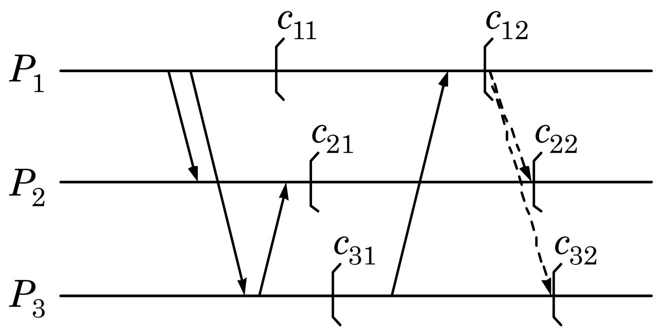
  (1) An **initiator** process P_i takes a tentative checkpoint and requests all processes to take tentative checkpoints. Receiving processes can decide whether to take a tentative checkpoint or not and send their decision to the initiator. There is no other communication until phase 2 is over.

  (2) If the initiator process P_i learns that all tentative checkpoints have been taken then it reverts its checkpoint to permanent and requests others do the same.

- this protocol ensures that no orphan messages are in the recorded state (processes are not allowed to send messages between phase 1 and 2)

Minimizing state recording during checkpointing

- it is not always necessary to record the state of a processor during checkpointing:



- the set $\{c_{12}, c_{21}, c_{32}\}$ is also a consistent set, hence it is not necessary for P_2 to take checkpoint c_{22}, but the set $\{c_{12}, c_{21}, c_{31}\}$ would be inconsistent
- each process assigns monotonically increasing numbers to the messages it sends:
  last_recd$_i$(*j*)   last message number that *i* received from *j* after *i* took a checkpoint
  first_sent$_i$(*j*)   first message number that *i* sent to *j* after *i* took a checkpoint
- if P_i requests P_j to take a tentative checkpoint it adds *last_recd*$_i$(*j*) to the message
- P_j takes a checkpoint only if *last_recd*$_i$(*j*) ≥ *first_send*$_j$(*i*)

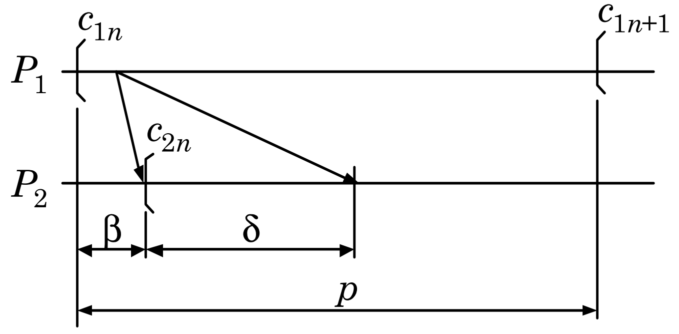
Minimizing state recording during checkpointing
(cont.)

- furthermore if P_i has not received a message from P_j since its last checkpoint then there is no need for P_j to establish a new checkpoint if P_i establishes one
- to make use of this P_i maintains a set $ckpt\_cohort_i$ which contains the processes from which received messages since its last checkpoint

```
upon receipt from i "take tentative checkpoint" || $last\_recd_i(j)$ do
    if $willing\_to\_ckpt_j$ and ($last\_recd_i(j) \geq first\_send_j(i)$ ) then
        take tentative checkpoint
        for all r in $ckpt\_cohort_i$ do
            send to r "take tentative checkpoint" || $last\_recd_j(r)$
        for all r in $ckpt\_cohort_i$ await($willing\_to\_ckpt$)
        if any r in $ckpt\_cohort_i$ and ($willing\_to\_ckpt_r$ = "no") then
            $willing\_to\_ckpt_j$:= "no"
    send to r $willing\_to\_ckpt_j$
upon receipt from i m:= "make checkpoint permanent" or
                       m:="undo tentative checkpoint"
    execute command in m
    for all r in $ckpt\_cohort_i$ send to r m
```

Synchronous checkpointing

- based on synchronized clocks check points are established with a fixed period *p* by all processes, where β is the clock synchronization precision and δ temporal uncertainty of message transmission



- if a message is sent during [*T* – β – δ, *T*] it will be received before *T* + β + δ
- to achieve a consistent state two possibilities exists:
  – prohibit message sending during interval β after checkpoint establishment
  – establish checkpoint earlier, at *kp* – β – δ and log messages during the critical instant

Stable storage

- stable storage is an important building block for many operations in fault-tolerant systems (fail-stop systems, dependable transaction processing, …)
- there are two operations which should work correctly despite of faults (as covered by the fault hypothesis):

  procedure writeStableStorage(*address*, *data*)

  procedure readStableStorage(*address*) **returns** (*status*, *data*)

- many failures can be handled by coding (CRC's) but other types cannot be handled by this technique:

  Transient failures: The disk behaves unpredictably for a short period of time.

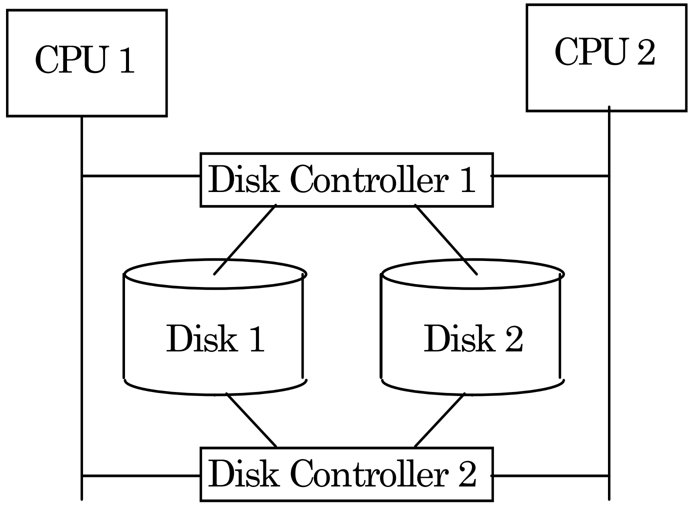
  Bad sector: A page becomes corrupted, and the data stored cannot be read.

  Controller failure: The disk controller fails.

  Disk failure: The entire disk becomes unreadable.

Disk shadowing

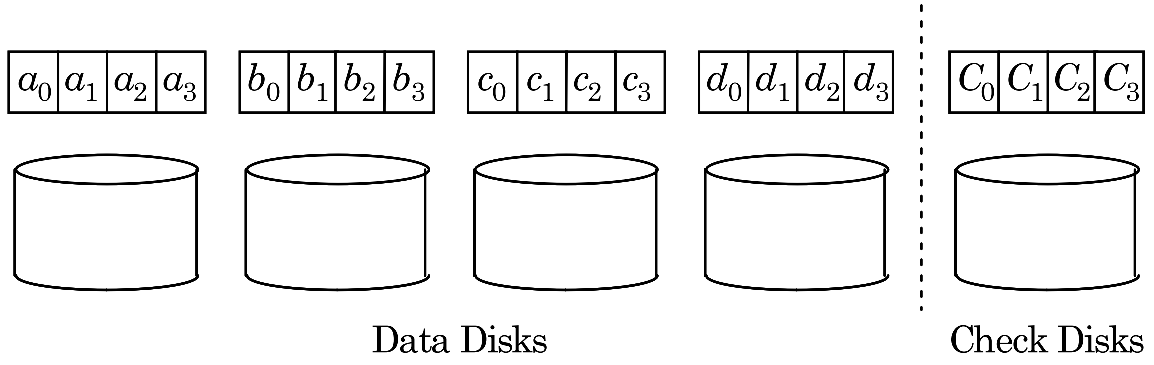
- a set of identical disk images is maintained on separate disks
- in case of two disks this technique is called *disk mirroring*
- for performance and availability reasons the disks should be "dual-ported" (e.g. Tandem system)



$$MTTF_{mirror} = \frac{MTTF}{2}\frac{MTTF}{MTTR}$$

Redundant Array of Inexpensive Disks (RAID)

- data is spread over multiple disks by “bit-interleave” (individual bits of a data word are stored on different disks)
- in the following example single bit failures can be tolerated since a parity bit is stored on a check disk and disks are assumed to detect single bit failures



- RAID’s provide high reliability and I/O throughput (parallel read/write)

$$MTTF_{RAID} = \frac{MTTF}{G+C} \frac{MTTF/(G+C-1)}{MTTR}$$

G .. data disks *C* .. check disks

Fail stop processors

- the visible effects of the failure of a fail stop processor are:

  FS1: It stops executing

  FS2: The internal state of the processor and the volatile storage connected to the processor are lost; the state of the stable storage is unaffected.

  FS3: Any processor can detect the failure of a fail stop processor.

- real processors do not have such a simple well defined semantics
- typically fail stop processors are implemented by a group of regular processors
- ***k*-fail-stop processor:**
  A processor is said to be *k*-fail-stop if it can tolerate up to *k* component (processor) failures while preserving its fail-stop property.

Implementing fail stop processors with stable storage

- **Assumptions:**
  - the stable storage is reliable
  - *k* + 1 *normal* processors
  - communication is reliable
  - message origin can be authenticated (point-to-point or cryptographic check)
  - synchronous system model (synchronized clocks, bounded communication)
- **Implementation:** requests to the stable storage are only granted if *k* + 1 requests are received within a time interval δ:

```
S-process:
    R:= bag of received requests with proper timestamp
    if |R| = k + 1 ∧ all requests are identical ∧ all requests are from different processors
                ∧ ¬failed then
        if request is write then writeStableStorage
        elseif request is read then readStableStorage and send result to all processors
    else // k-fail stop processor has failed
        writeStableStorage failed
```

Implementing fail stop processors without stable storage

- **Assumptions:**
  - – the storage processor are not reliable and can fail byzantine
  - – k + 1 p-processors (program processors)
  - – 2k + 1 s-processors (storage processors)
  - – each s-process has a copy of the stable storage
  - – communication is reliable
  - – message origin can be authenticated (point-to-point or cryptographic check)
  - – synchronous system model
- **Implementation:**
  - – requests to the stable storage subsystem are only granted if k + 1 requests are received within a time interval δ
  - – failures of individual storage processors are masked by byzantine agreement (under the assumption of authentification detectable failures)

Implementing fail stop processors without stable storage

```
p-processWrite:
    initiate byzantine agreement with all s-processors on the write request

p-processRead:
    broadcast the read request to all s-processors
    wait for at least $k + 1$ identical results from the s-processors (majority vote)

s-process:
    M:= bag of received requests with proper timestamp
    if request is read then
        send requested value to all p-processors whose request is in M
    elseif request is write then
        if $|M| = k + 1 \wedge$ all requests are identical $\wedge$ all requests are from different processors
                $\wedge \neg$failed then
            write the value
        else // k-fail stop processor has failed
            write failed
            send message “halt” to all p-processors
```

Fault diagnosis in distributed systems

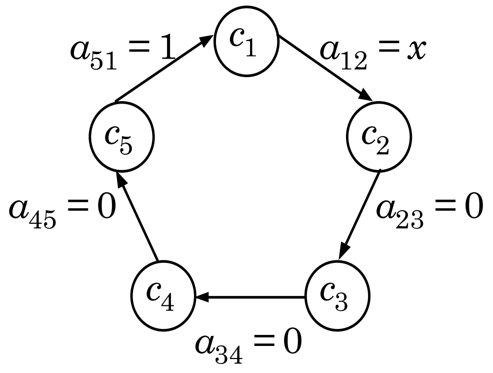
- **Problem:** Each non-faulty component has to detect the failure of other components in a finite time.
- while it is the goal to identify all failed components there are theoretical upper bounds on the number of failed components that can be identified
- **PMC model:**
  - a system S is composed out of n components $C = \{c_1, c_2, \dots , c_n\}$
  - components are either correct or faulty as a whole, they are the lowest level of abstraction that is considered
  - each component is powerful enough to test other components
  - *tests* involve application of stimuli and the observation of responses, tests are assumed to be *complete* and *correct*
  - correct components always report the status of the tested components correctly
  - faulty components can return incorrect results of the tests conducted by them (byzantine failure assumption)

Syndromes

- each component belonging to *C* is assigned a particular subset of *C* to test (no component tests itself)
- the complete set of test assignments is called **connection assignments**, it is represented by a graph $G = (C, E)$
  - each node in *C* represents a component
  - each edge represents a test such that (c_i, c_j) iff c_i tests c_j.
  - each edge is assigned an outcome a_{ij},
    $a_{ij} = 0$ if c_i is correct and c_j is correct
    $a_{ij} = 1$ if c_i is correct and c_j is faulty
    $a_{ij} = x$ if c_i is faulty
- the set of all test outcomes is call the **syndrome** of *S*

An example system

- the system consists of five components, it is assumed that c_1 is faulty



- the syndrome for this system is a 5 bit vector $(a_{12}, a_{23}, a_{34}, a_{45}, a_{51}) = (x, 0, 0, 0, 1)$
- ***t*-diagnosable:** A system is *t* fault diagnoseable if, given a syndrome, all faulty units in *S* can be identified, provided that the number of faulty units does not exceed *t*.
- a system *S* with *n* components is *t*-diagnoseable if $n \geq 2t + 1$ and each component tests at least *t* others, no two units test each other

Central diagnosis algorithms

- A simple algorithm is to take an arbitrary component and suspect it to be either correct or faulty. Based on this guess, and the test results of other components are labeled, if a contradiction occurs, the algorithm backtracks. Complexity $O(n^3)$
- the best known algorithm has a complexity of $O(n^{2.5})$

The adaptive DSD algorithm

- an adaptive distributed system-level diagnose algorithm that is round based
- it stabilizes within n rounds and has no bound on t, provided the communication is reliable
- each component holds an array $TESTED\_UP_i$
- $TESTED\_UP_i[k] = j$: component i has received information from a correct component saying that k has tested j to be fault free

The adaptive DSD algorithm (cont.)

- each component executes the following algorithm periodically

```
$t$:= $i$
**repeat**
   $t$:= ($t$ + 1) mod $n$
   request $t$ to forward $TESTED\_UP_t$ to $i$
**until** ($i$ test $t$ as “fault free”)
$TESTED\_UP_i[i]$:= $t$
**for** $j$:= 1 to $n$ – 1
    **if** $i$ ° $t$
      $TESTED\_UP_i[j]$:= $TESTED\_UP_t[j]$
```

- the algorithm stops if the first fault free component is found
- this component is marked as fault free in $TESTED\_UP_i[i]$
- the information of $TESTED\_UP_t$ is copied to $TESTED\_UP_i$ which forwards the diagnostic information in reverse order through the system

The adaptive DSD algorithm (cont.)

- if a component wants to diagnose the system state it executes the following algorithm:

```
**for** $j$:= 1 to $n$ $STATE_i[j]$:= “faulty”
$t$:= $i$
**repeat**
   $STATE_i[t]$:= “fault-free”
   $t$:= $TESTED\_UP_i[t]$
**until** $t = i$
```

- the diagnosis algorithm constructs a cycle that contains all correct components
- if the length of the cycle is *l* then after *l* rounds all vectors *TESTED_UP* will be updated
- since the cycle is constructed by ascending component indices, the repeat loop in the algorithm collects all correct components and updates *STATE* accordingly

Fault-tolerance by self-stabilization

- **Self-Stabilization:** A distributed system *S* is self-stabilizing with respect to some global predicate *P* if it satisfies the following two properties:
  (1) **Closure:** *P* is closed under the execution of *S*. That is, once *P* is established in *S*, it cannot be falsified.
  (2) **Convergence:** Starting from an arbitrary global state, *S* is guaranteed to reach a global state satisfying *P* within a finite number of state transitions.
- self-stabilizing systems need not be initialized and they can recover from transient failures (adaptive DSD is self-stabilizing)
- self-stabilization is a different approach to fault-tolerance, it is not based on countering the effects of failures but concentrating on the ability to reach a consistent state
- problems are how to achieve a global property with local actions and local knowledge, lack of theory on how to design self-stabilizing algorithms and how to guarantee timeliness

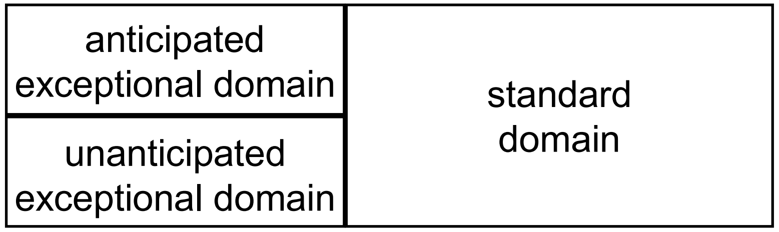
Fault tolerant software

- to tolerate software faults the system must be capable to tolerate design faults
- in contrast, for hardware it is typically assumed that the design is correct and that components fail
- software requires **design diversity**
- **But:** especially for software, perfection is much easier and better understood than fault-tolerance

System aspects of dependable computers

Exception handling

- to detect erroneous states of software modules the exception mechanism can be used (software and hardware mechanisms for detection of exceptional states)
- a procedure (method) has to satisfy a *pre condition* before delivering its intended service which has to satisfy *post conditions* afterwards
- the state domain for a procedure can be subdivided:


anticipated exceptional domain
unanticipated exceptional domain
standard domain

- an *exception mechanism* is a set of language constructs which allows to express how the standard continuation of module is replaced when an exception is raised
- exception handlers allow the designer to specify recovery actions (forward or backward recovery)

Recovery blocks

- a method to apply diverse designs to provide design fault-tolerance
- can also handle (transient) hardware faults
- based on an **acceptance test**—which detects erroneous states—different modules are tried until an acceptable state is reached

```
ensure      acceptance_test
by          primary_module
else by     alternate_module_1
else by     alternate_module_2
   :
else by     alternate_module_n
else error
```

- acceptance tests are application-specific, they have only limited error detection coverage
- before trying a module a recovery point has to be established, backward recovery to try next module

Recovery blocks (cont.)

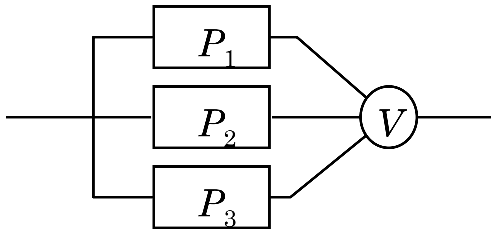
- mixture of systematic and application-specific fault-tolerance:
  - systematic method to apply *n* diverse modules by rollback recovery
  - acceptance test is application specific
- recovery blocks can be nested such that a module itself is a recovery block
- can also be supported with the exception mechanism (e.g. standard exception handler for unidentified exceptions can be used)
- modeling of recovery block with primary and one alternative is equivalent to passive redundant system (acceptance test $\cong$ switch)
- can be used to implement **graceful degradation**, when different modules provide different levels of service
- **Problems:**
  - how to development acceptance tests
  - quality of acceptance test
  - delay for backward recovery in real-time systems

Distributed recovery blocks

- for uniform treatment of software and hardware failures
- the primary module is executed on the primary processor, the alternate is executed on a backup processor
- both processors use duplicated acceptance test
- if the primary module fails, a message is sent to the backup and the backup then forwards its results
- combination of software and hardware diversity

N-Version Programming

- *n* non-identical replicated software modules are applied
- instead of an acceptance test a voter takes a *m* out of *n* or majority decision



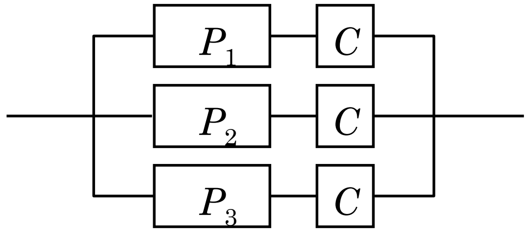
- majority voting can tolerate (*n* – 1)/2 failures of modules
- modeling of *n*-version programming is equivalent to active redundant systems with voting
- *driver* program to invoke different modules (different processes for module execution), wait for results and voting
- require more resources than recovery blocks but less temporal uncertainty (response time of slowest module)

N-Version Programming (cont.)

- *n*-version programming is approach to systematic fault-tolerance:
  - there is no application specific acceptance test necessary
  - exact voting on every bit is systematic
- **But:** problem of replica nondeterminism:
  - the real-world abstraction limitation is no problem (all modules get exactly the same inputs from driver program)
  - consistent comparison problem: diverse implementations, different compilers, differences in floating point arithmetic, multiple correct solutions (*n* roots of *n*th order equation), …
- **Problems:**
  - there is **no** systematic solution for the consistent comparison problem
  - either very detailed specification with many agreement points (limits diversity)
  - or approximate voting to consider nondeterminism (application-specific)

N self-checking programming

- *n* versions are executed in parallel (similar to *N*-version programming)
- each module is self-checking, an acceptance test is used (similar to recovery blocks)
- mixture of application specific and systematic fault-tolerance
- requires no backward recovery and no voting



Deadline mechanism

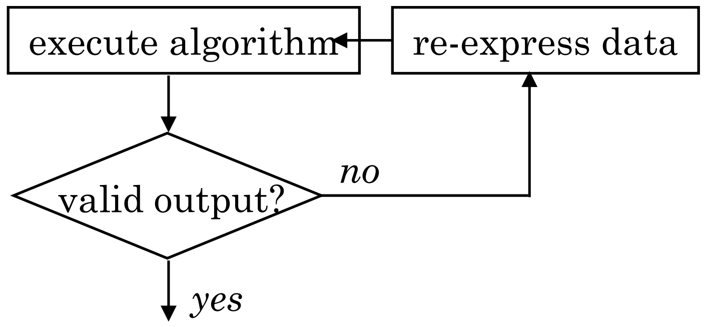
- based on recovery blocks, but deadline instead of acceptance test
- used to avoid timing failures in real-time systems

```
service   name
within  response-period
by          primary_module
else by   alternate_module
```

- it is assumed that an upper execution bound for the alternate is known
- for the primary it is assumed that the execution is timely in most cases
- if the primary does not finish within the slack time (*response-period* – *execution bound for alternate*) then the primary is aborted and the alternate is used

Data diversity

- it is assumed that software fails on some “special” inputs
- if the inputs are changed slightly then the same software may work correctly
- *data re-expression* is necessary to generate different but logically equivalent data sets (application specific)
  – for real variables the value may be changed slightly
  – coordinate transformation to new origin



- cheaper alternative to diverse software

Independence assumption

- software fault-tolerance is based on the assumption that diverse designed models fail independently
  - different programmer teams
  - different programming language and tools, …
- empirical studies have shown that diverse designed software **does not** fail independently (co-dependent failures)
  - 27 program versions have been written by two universities
  - failure probabilities for 1 10^{-6} test cases with 351 2-version systems and 2925 3-version systems were calculated
- for the average 3-version system the failure probability improved by a factor 19, compared to the average single version
- if the independence assumption would hold, the failure probability should have decreased by at least three orders of magnitude

Problems due to co-dependence

- only modest increase for very high effort
- development costs are main costs for software
- replica nondeterminism or application-specific methodology
- handling problems for multiple version systems
  - project management
  - configuration control
  - versioning
  - modifications and updates

Conclusion

- fault-tolerance is not the only means for dependability: perfection, maintenance
- compared to fault-tolerance, perfection and maintenance have lower design complexity
- maintenance aspects: costs, error latency, error propagation, maintainability, smallest replaceable units (SRU), diagnosis
- two fundamental approaches to fault-tolerance:
- **Systematic fault-tolerance**
  - replication of components
  - divergence of components is used for fault-detection
  - redundant components are used for continued service
- **Application-specific fault-tolerance**
  - reasonableness checks for fault detection
  - state estimations for continued service

Conclusion (cont.)

- systematic fault-tolerance:
  - **lower design complexity**, separation of fault-tolerance and application
  - requires replicated components (cost)
  - enforcement of replica determinism
- application specific fault-tolerance:
  - requires no replicated components (low cost)
  - high design complexity, fault-tolerance and application aspects intertwined
  - problems with reasonable checks and state estimations
- **Replica determinism:**
  Non-faulty replicated components are required to behave correspondingly
- computer systems show nondeterministic behavior if no countermeasures are taken: (inconsistent inputs, inconsistent order, timeouts, dynamic scheduling decisions, …)

Conclusion (cont.)

- **fundamental limitations to replication:**
  - the real world abstraction limitation
  - the impossibility of exact agreement
  - intention and missing coordination
- **synchronous systems:**
  - synchronous processors (bounded minimum speed for processing steps) and
  - bounded communication (bounded transmission delay)
- **asynchronous systems:**
  - asynchronous processors and/or unbounded communication
- **replica control:**
  - membership
  - agreement
  - order

Conclusion (cont.)

- central vs. distributed replica control
- basic communication services: consensus, reliable broadcast
- synchronization: logical vs. real-time clocks
- replica control strategies:
  - lock-step execution
  - active replication
  - semi-active replication
  - passive replication
- failure recovery
- redundancy preservation
- coverage vs. complexity
- hardware vs. software

Conclusion (cont.)

- consensus algorithm and impossibility result for asynchronous systems
- reliable broadcast: FIFO, causal, atomic
- broadcast topology and broadcast property
- backward or rollback recovery: recovery points, recovery lines, domino effect
- asynchronous and synchronous checkpointing strategies
- stable storage: disk shadowing, redundant array of inexpensive disks (RAID)
- implementation of fail-stop processors
- fault diagnosis in distributed systems
- fault-tolerance by self-stabilization

System aspects of dependable computers

Conclusion (cont.)

- Fault-tolerance mechanisms for software
  - – exception handling
  - – recovery blocks (distributed recovery blocks)
  - – *n*-version programming
  - – *n*-version self-checking programming
  - – deadline mechanism
  - – data diversity
- software fault-tolerance is based on the assumption of independent design and coding failures, but empirical data has shown that failures are co-related