TECHNISCHE UNIVERSITÄT WIEN

TTTech

Dependable Computer Systems

Part 6c: System Aspects

Contents

- Consensus
- Interactive Consistency Algorithms
- Broadcast Properties and Algorithms
- Checkpointing
- Stable Storage
- Diagnosis
- Fault-Tolerant Software

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.

Consensus (cont.)

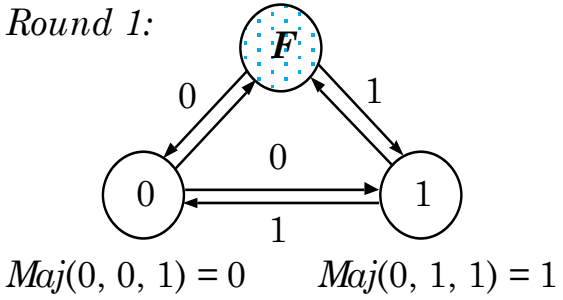
- The consensus problem under the assumption of byzantine failures was first defined in 1980 in the context of the SIFT project which was aimed at building a computer system with ultra-high dependability. Other names are
  - byzantine agreement or byzantine general problem
  - interactive consistency

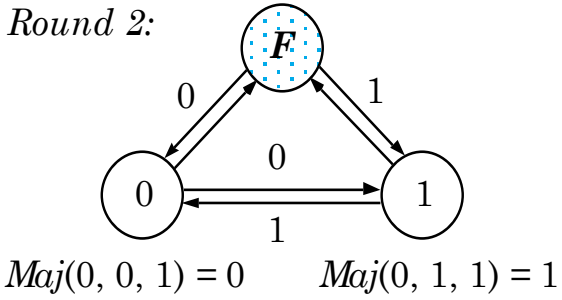
Impossibility of deterministic consensus in asynch. systems

- 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

Byzantine Failure Behaviour

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



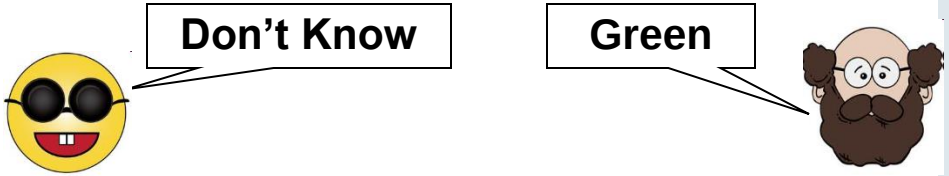


- Situation:

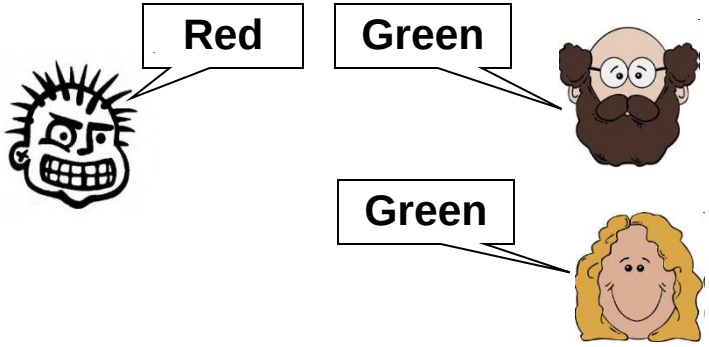
  What is the color of the house?

No Failure

Fail-Silence Failure



Fail-Consistent Failure



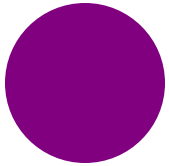
Situation:

What is the color of the house?

Static Situation – one Truth

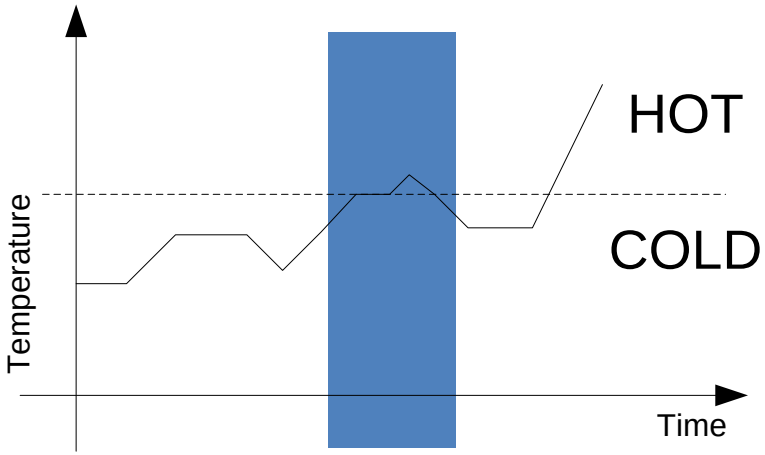
Situation:

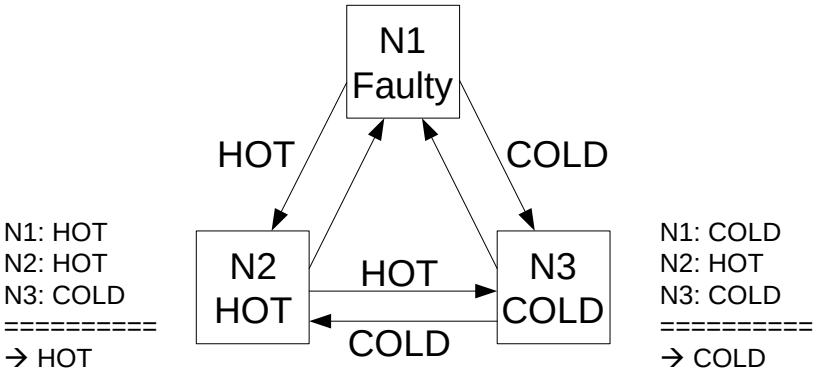
What is the color of the ball ?



Dynamic Situation – >one Truth

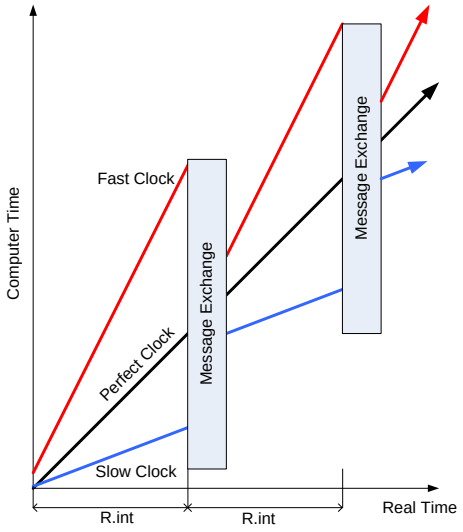
A distributed system that measures the temperature of a vessel shall raise an alarm when the temperature exceeds a certain threshold.
The system shall tolerate the arbitrary failure of one node.
How many nodes are required?
How many messages are required?

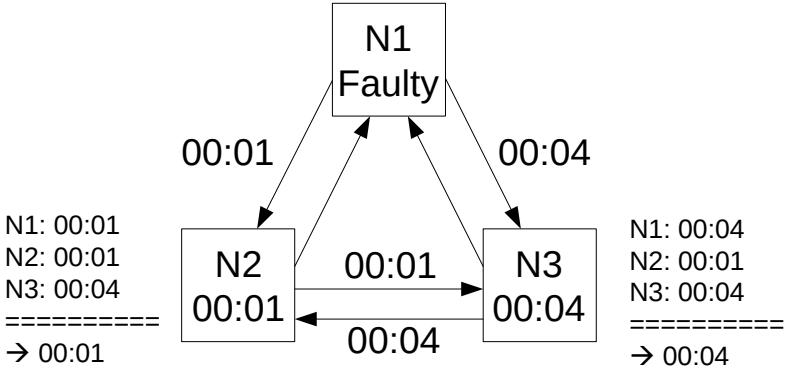




In general, three nodes are insufficient to tolerate the arbitrary failure of a single node.
The two correct nodes are not always able to agree on a value.
A decent body of scientific literature exists that address this problem of dependable systems, in particular dependable communication.

A distributed system in which all nodes are equipped with local clocks, all clocks shall become and remain synchronized.
The system shall tolerate the arbitrary failure of one node.
How many nodes are required?
How many messages are required?





In general, three nodes are insufficient to tolerate the arbitrary failure of a single node.
The two correct nodes are not always able to bring their clocks into close agreement.
A decent body of scientific literature exists that address this problem of fault-tolerant clock synchronization.

Interactive Consistency Algorithms

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.

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 from processor j (j !=i) in step 2. Processor i uses the value *Majority*($v_1, \ldots, v_{n-1}$).

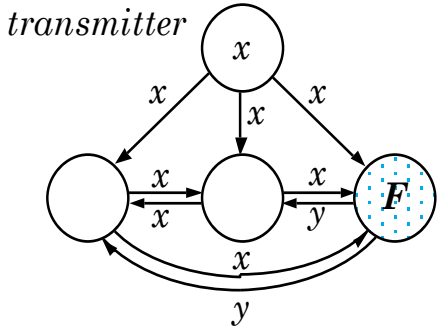
***ICA*(0):**

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.

Example (n=4, t=1)

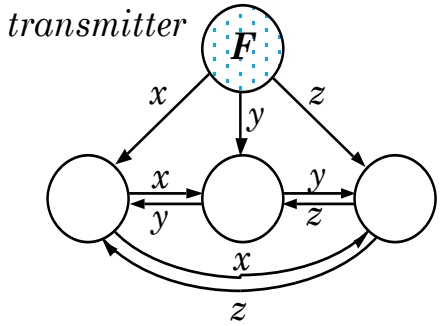
Case 1, one of the receivers is faulty:

- 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 $V_i = 0$
- 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.

Interactive consistency with signed messages (cont.)

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

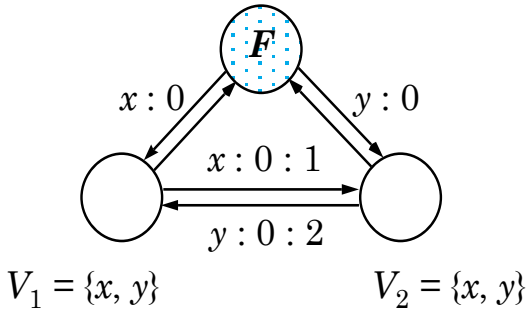
1. The transmitter signs its value and sends it to all other nodes
2. $\forall i$:

   (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.

Example (n=3, t=2)

- we again consider the case of the faulty transmitter:
- because of the signed messages it becomes clear that the transmitter is faulty

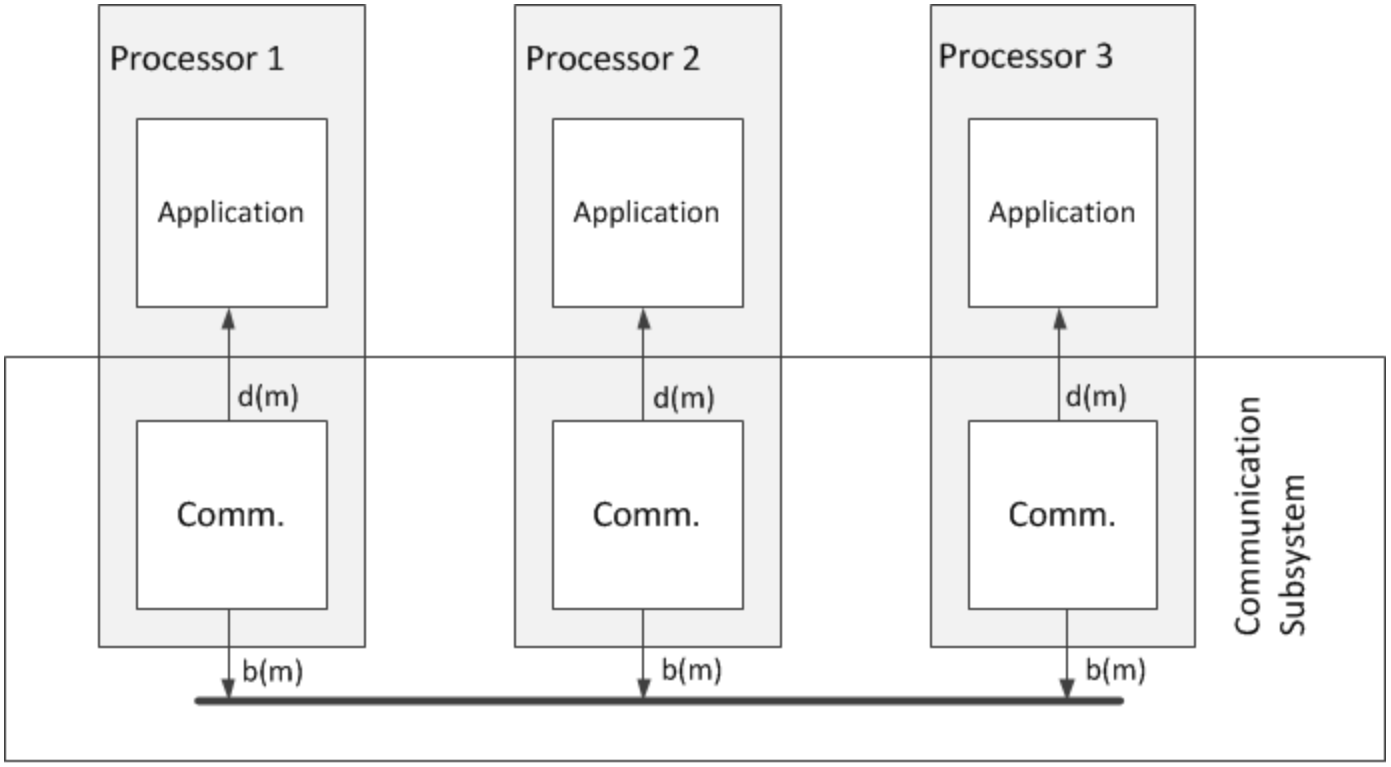


Complexity of consensus

- *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 $(n - 1)(n - 2) \ldots (n - t - 1)$ 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)

Broadcast Algorithms

Terminology and Concepts



d(m) ... deliver message m
b(m) ... broadcast message m

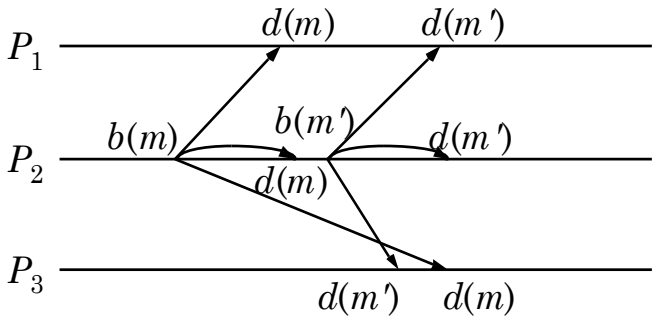
Reliable broadcast

- 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)

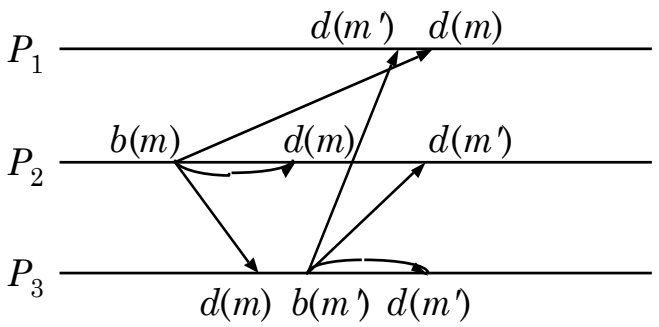
FIFO Broadcast

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

non-FIFO Broadcast



Problem with FIFO Broadcast

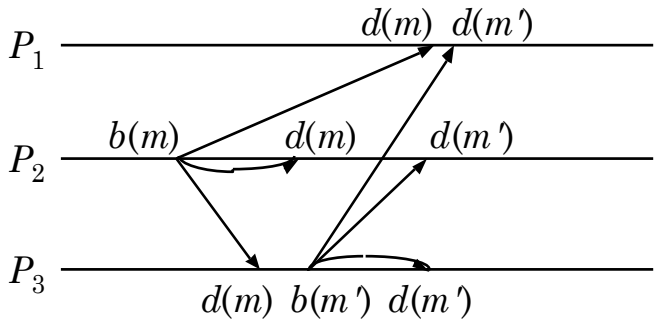


$b(m)$ … broadcast message m $d(m)$ … deliver message m

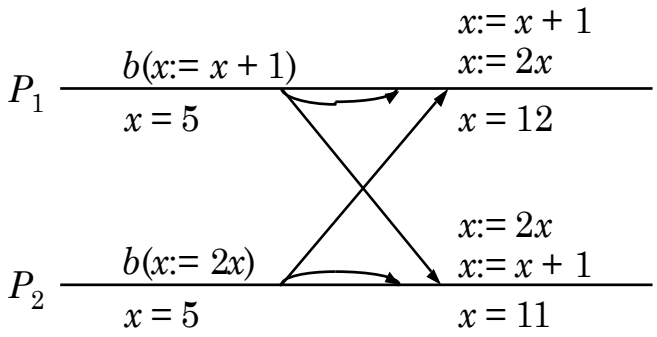
Causal Broadcast

- 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



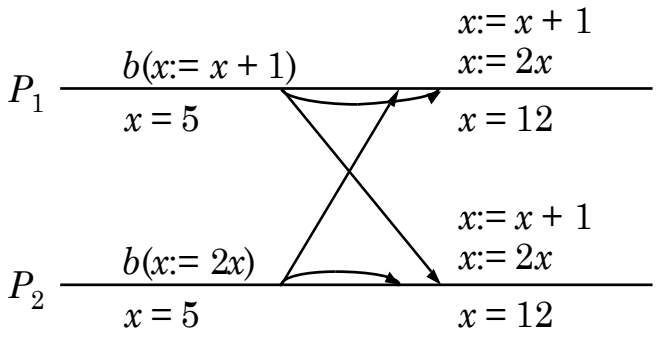
Problem with Causal Broadcast



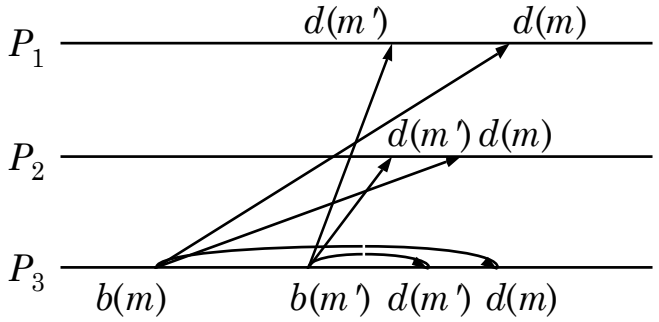
Atomic Broadcast

- 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

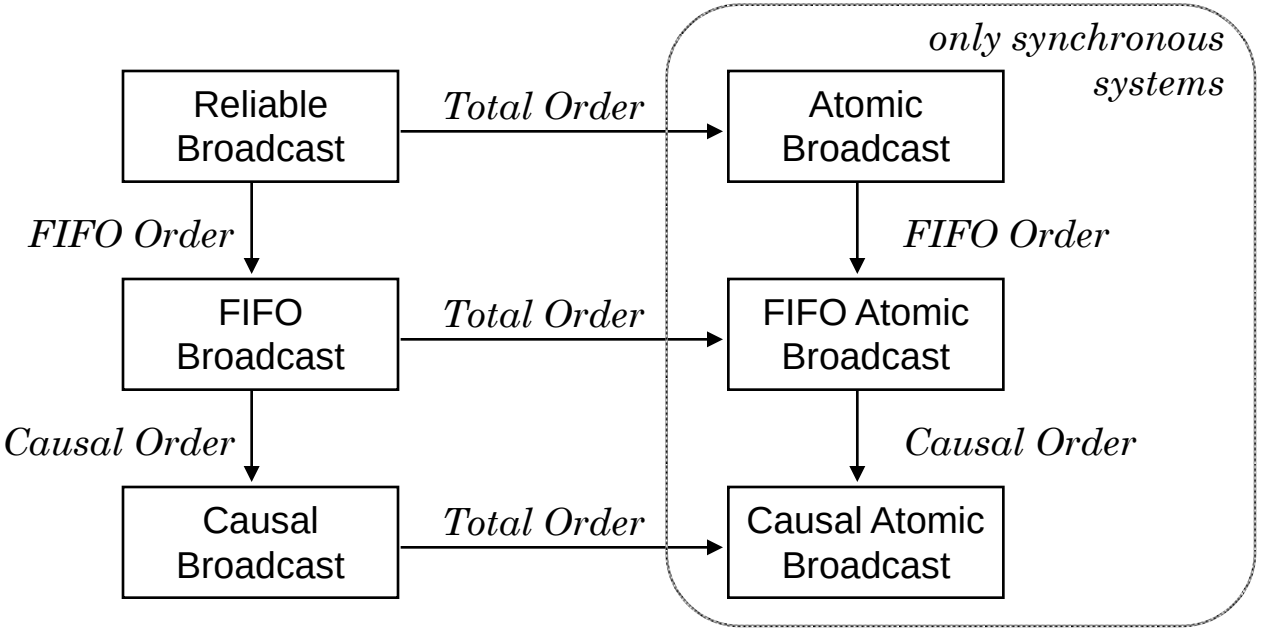


Extensions of Atomic Broadcast

FIFO Atomic Broadcast = Reliable Broadcast + FIFO Order + Total Order

Causal Atomic Broadcast = Reliable Broadcast + Causal Order+ Total Order

Relationships among broadcast protocols:



Reliable broadcast protocol

- **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 Protocols

- **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, ⟨*rcntDlvrs* || *m*⟩)
    rcntDlvrs:= ⊥
  - **deliver**(C, –):
    upon deliver(F, ⟨m_1, m_2, … m_l⟩) 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

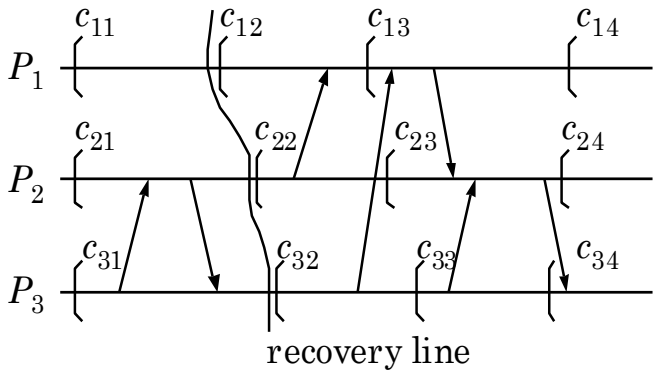
Checkpointing

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:

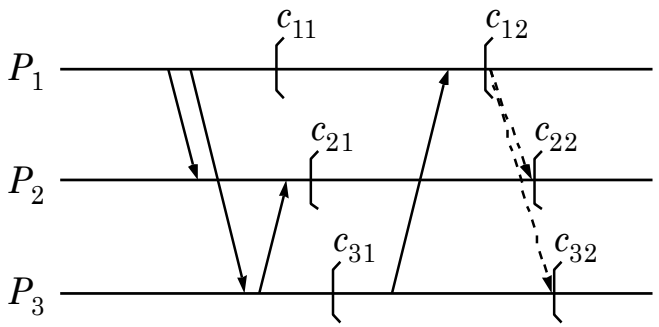
  (1) An **initiator** process P_i takes a tentative checkpoint and requests all processes to take tentative checkpoints. Receiving processes can decide whether to take a tentative checkpoint or not and send their decision to the initiator. There is no other communication until phase 2 is over.

  (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)

A distributed checkpointing and rollback algorithm (cont.)

- 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 $last\_recd_i(j) \geq first\_send_j(i)$

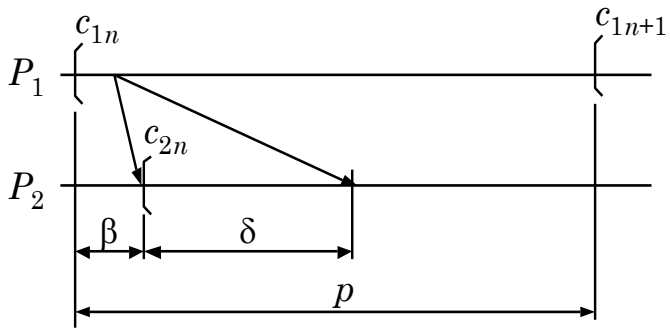
A distributed checkpointing and rollback algorithm (cont.)

- 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

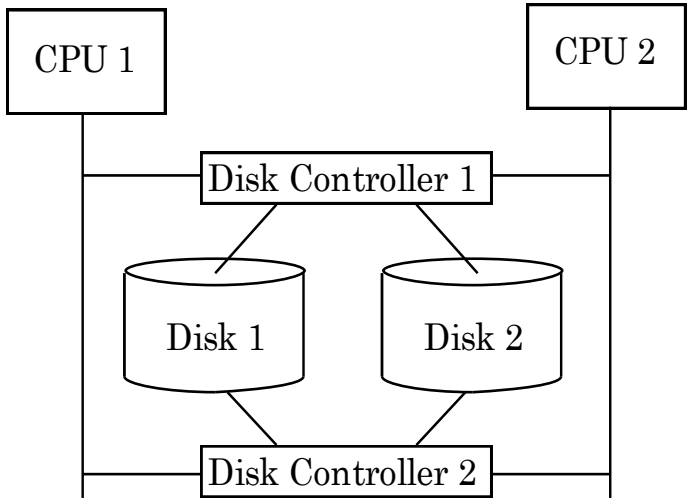
Stable Storage

- 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

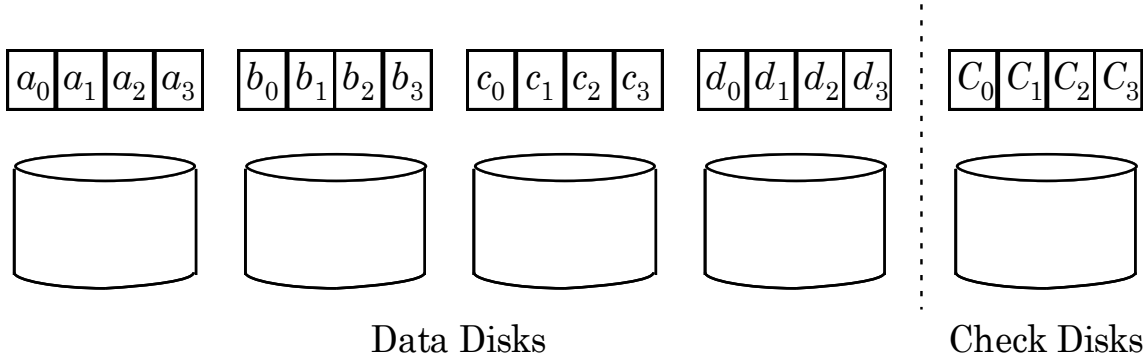
- a set of identical disk images is maintained on separate disks
- in case of two disks this technique is called *disk mirroring*
- for performance and availability reasons the disks should be "dual-ported" (e.g. Tandem system)

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



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

Diagnosis

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

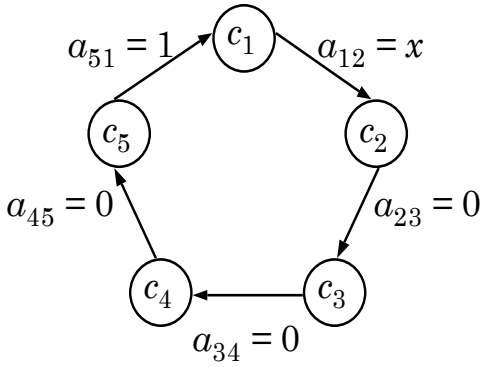
- **PMC model:**
  - a system S is composed out of n components $C = \{c_1, c_2, \ldots, c_n\}$
  - components are either correct or faulty as a whole, they are the lowest level of abstraction that is considered
  - each component is powerful enough to test other components
  - *tests* involve application 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 (x is in {0|1})
- 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) (x is in {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 *i* 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

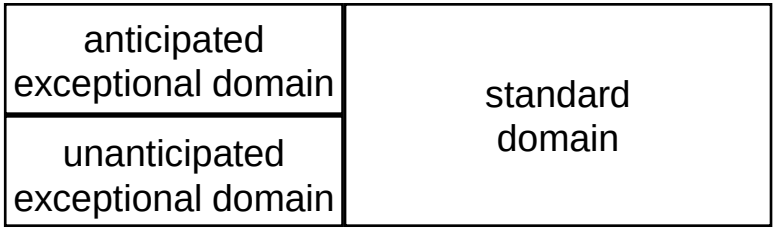
Fault-Tolerant Software

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

Exception handling

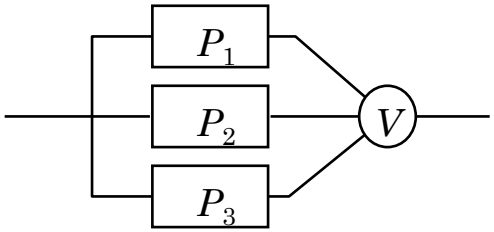
- to detect erroneous states of software modules the exception mechanism can be used (software and hardware mechanisms for detection of exceptional states)
- a procedure (method) has to satisfy a *pre condition* before delivering its intended service which has to satisfy *post conditions* afterwards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



- 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)

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