Concept and Practice of Fault-Tolerant Distributed Systems[1]

Tamás Bartha

Technical University of Budapest

Contents:

- Hardware failure semantics
- Hardware failure masking
- Architectural issues
- Software failure semantics
- Software failure masking
- Implementation issues

1. Based on F. Cristian, "Understanding Fault-Tolerant Distributed Systems," *Comm. of the ACM*, vol. 34, no. 2, pp. 57–78, February 1991.

Introduction

<u>Distributed system architecture:</u>

- Basic building blocks

  a. **service**: collection of operations, triggered by user inputs or time

  - executions may result in outputs to users and service state changes

  b. service operations can be performed *only* by the related **server**

  - implements a service according to the *service specification*
  - implementation details are hidden from the users

  c. **depends** relation: servers use other services to implement a service

  Server s_i depends on s_j if the correctness $C(s_i)$ of s_i depends on $C(s_j)$.

  - user / client—resource / server relation is *relative* to depends relation
  - abstraction levels can be represented as an acyclic digraph

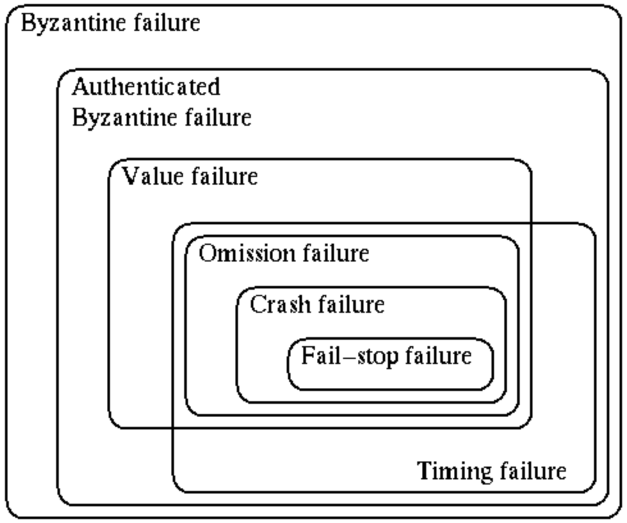
Failure classification / 1.

Possible behavior of a server:

- *Correct*: the delivered service is consistent with its specification
- The service differs from the specification: *failure*
  - **omission failure**: the server does not respond to a request
  - timing failure: the response is functionally correct, but untimely
    - **early timing failure**: response *before* the service time frame
    - **late / performance failure**: response *after* the service time frame
  - response failure: the server responds incorrectly
    - **value failure**: the value of the output is incorrect
    - **state transition failure**: the generated state transition is incorrect

Failure classification / 2.

- crash failure: the server omits responses until it restarts
  - **amnesia** ↔ **partial-amnesia crash**: entire / partial state is lost
  - **pause** ↔ **halting crash**: successful restart / fail to restart
- **Byzantine / arbitrary failure**: unknown, inconsistent, malicious ~
  - authenticated Byzantine failure: altered messages are detected



Hardware failure semantics / 1.

Specification of fault-tolerant systems must be *extended* to include failure behaviors.

- Allowable server behaviors under failure
  - examples of failure semantics:
    - service loses messages, negligible delay → *omission*
    - service loses or delays, does not corrupt → *omission / performance*
    - processor: crash, memory read: omission → *crash / read omission*
    - any failure behavior is allowed → *arbitrary*
  - **design of the server** must ensure that it implements a specified ~

stronger failure semantics

- network with omission / performance failure semantics:
  error-detecting codes to avoid message corruption
- network with omission failure semantics:
  access protocols that assure bounded access delay, message transmission delay
- raw hardware processor with crash failure semantics:
  processor duplication and matching

Hardware failure masking / 1.

Hierarchical failure masking:

A failure behavior can be classified only with respect to a certain server specification, at a certain level of abstraction.

Failed resource		**Server**		**User**	**Provides**
network	2 bit corruption	network layer	2-bit EDC	data link layer	*omission*
local clock	crash	clock server		communication layer	*arbitrary*

- Failure propagation among servers
  - arbitrary ~ in a server: depending servers will likely have arbitrary ~
  - designers prefer to use failure semantics **stronger than** arbitrary
    - crash, omission, or performance failures
    - failure masking: server delivers its service despite lower-level failures
    - exception handling: propagate information across abstraction levels
    - servers with *atomic behavior* simplify fault-tolerant design

Hardware failure masking / 2.

Group failure masking:

- Group management mechanisms
  - service by a **group** of redundant, physically independent servers
    - mask member failures
    - make the group functionally indistinguishable from a single server
    - depend on member failure semantics
  - **group output** is a function of individual member outputs:
    - output by fastest member, distinguished member, majority voting
    - group masking any *k* concurrents member failures: *k*-fault tolerant

Members		**Function**	**Tolerates**	**Provides**
k members	crash / performance	fastest member	k - 1 members	crash / performance
k members	crash / performance	primary / backup	k - 1 members	crash / performance
$2k + 1$ members	arbitrary	majority voting	k members	arbitrary

Hardware failure semantics / 2.

Choosing a failure semantics:

- A server having stronger failure semantics is
  - more expensive
  - cheaper to handle its failure at higher abstraction levels
    - in practice hierarchical and group masking is often combined
    - balance the amount of failure detection, recovery, and masking
    - obtain the best overall cost / performance / dependability
- Server specification = functional + stochastic requirements
  - stochastic requirements for a server r:
    - s_r > P(standard behavior is observed at run-time)
    - c_r < P(failure different from specified failure semantics is observed)

Hardware failure semantics / 3.

- Checking user specifications
  - validate s_r, c_r by stochastic modeling / simulation / testing
  - example: single-fault tolerant storage server
    - S-read after S-write returns the value written
    - S-write always succeeds
    - S-read failures are omission failures
      - S-read might return 10^{10} unique values different from the written
    - evaluation results:

	Requirements	Elementary servers	Duplication, matching	Triplication, voting
s_r	10^{-9}	10^{-9}	10^{-18}	10^{-18}
c_r	10^{-18}	10^{-19}	10^{-19}	10^{-48}

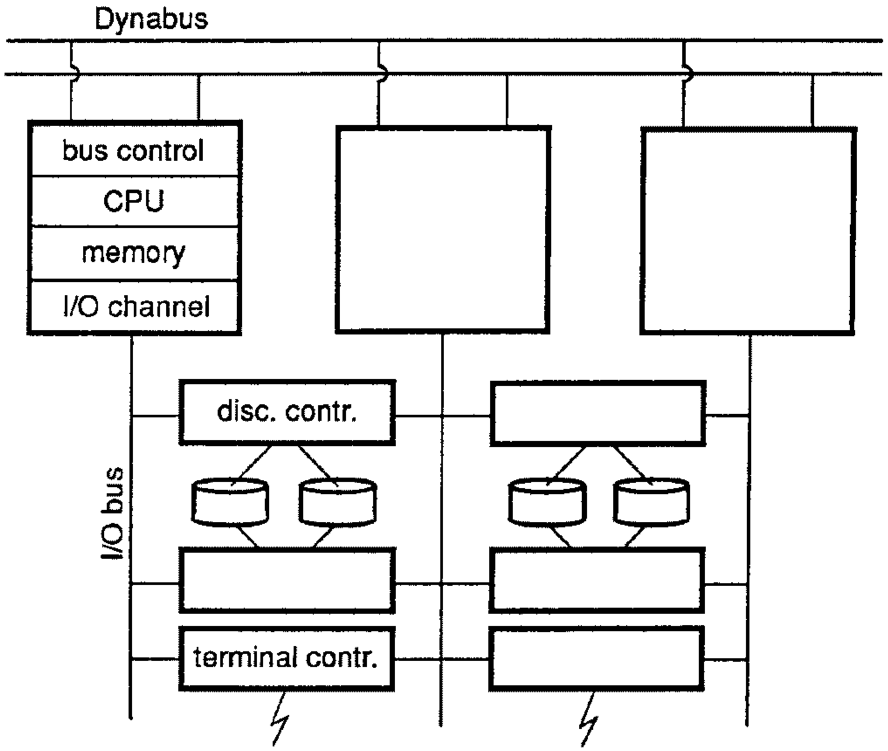
Hardware architectural issues

An OS uses hardware resources as CPUs, memory, I / O and communication controllers.

- Replaceable hardware units
  - physical units of failure, replacement and growth of hardware
    - goals: install or remove units without service disruption, or
    - at least nice failure semantics (*crash*, *omission*, *performance*)
    - *field* ↔ *customer* replaceable unit: replacement requires / does not require the assistance of a field engineer
- Example systems
  - coarse granularity architecture:
    - Tandem
    - DEC VAX Cluster
    - IBM MVS / XRF
  - fine granularity architecture:
    - Stratus
    - Sequoia

Tandem / 1.

Tandem processor architecture:

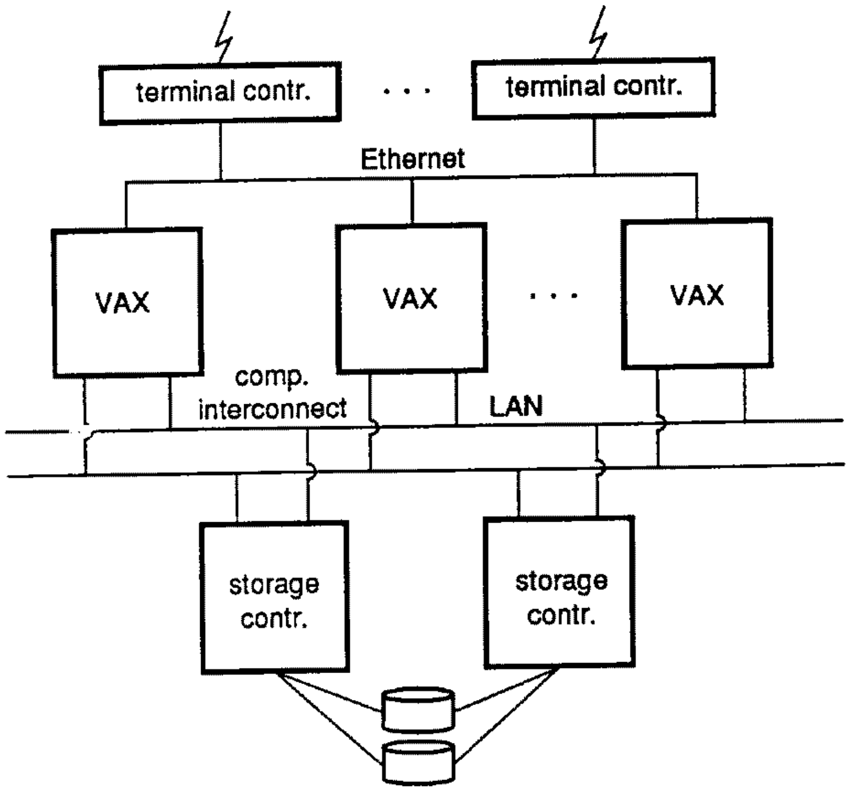


Tandem / 2.

- Tandem hardware architecture
  - replaceable hardware units: CPU, memory, bus and I / O controllers
    - field ~ are the disk, tape, communication and terminal controllers
  - single-fault tolerant architecture:
    i. any terminal controller $\rightarrow$ physical resources $\geq$ **two disjoint paths**
    ii. group servers into failure-masking **server pair groups**
- Operating system
    - implements pair management algorithms
    - decides which resources play an active / backup role
    - hierarchical and group masking to mask hardware resource failures
    - disk, bus, or bus controller failures are masked from applications
  - CPU / memory failure: the backup processor is promoted to primary
  - user-level servers are not paired $\rightarrow$ their failure is visible to users

DEC VAX Cluster / 1.

VAX Cluster processor architecture:



DEC VAX Cluster / 2.

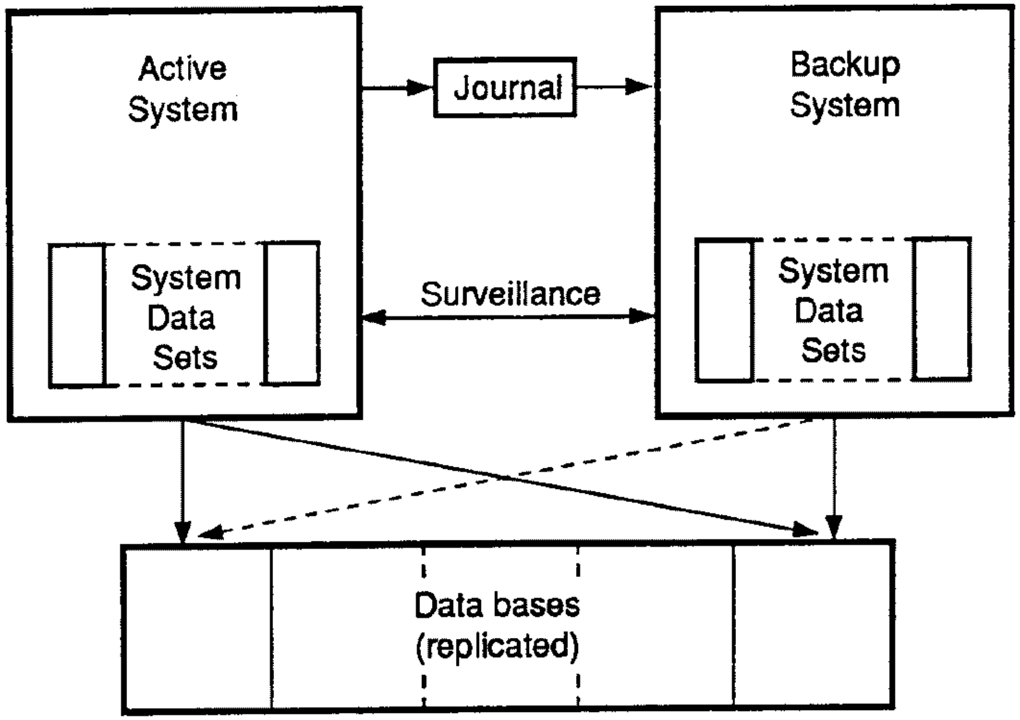
- VAX Cluster hardware architecture:
  - replaceable hardware units:

entire VAXs (CPU, storage, LAN)	entire storage controllers (μP, device drivers, LAN)	entire terminal controllers

  - local are networks:
    - high-speed custom-designed Computer Interconnect
    - low-speed Ethernet
  - two disjoint paths from terminal controllers to any physical resource
  - hardware resource failures are masked by hierarchical and group masking
  - if a VAX fails:
    - all software servers running on the failed VAX will also fail
    - they have to be *explicitly* restarted on another VAX
    - higher-level application server failures are visible to users

IBM MVS / XRF / 1.

IBM XRF architecture:



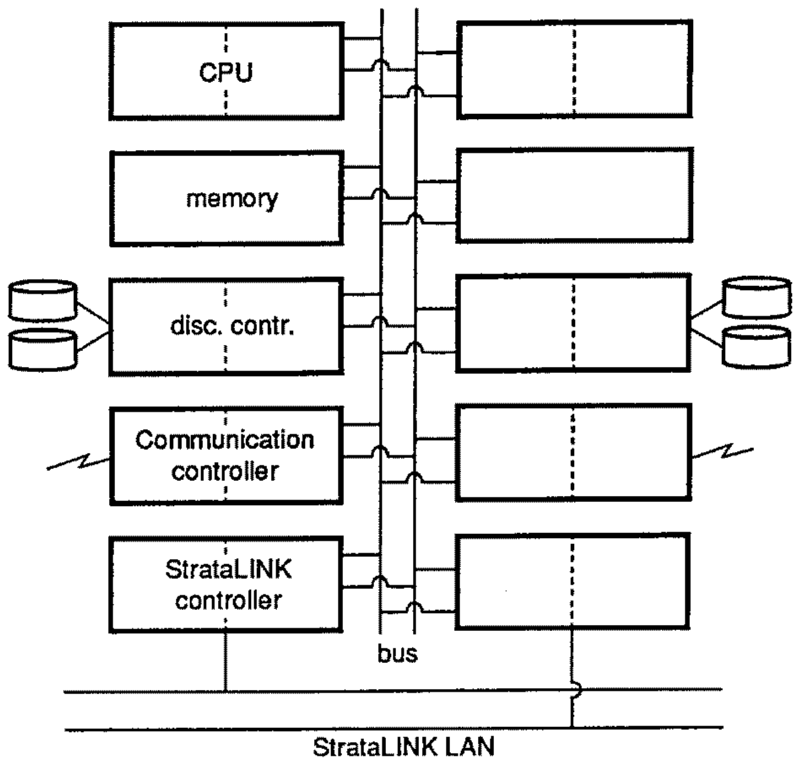
IBM MVS / XRF / 2.

- XRF hardware architecture
  - replaceable hardware units: entire high-end IBM processors
  - provides continuous IMS database service by using group masking
    - two IMS servers on two distinct processors, point-to-point network
    - *primary server*: interprets user requests, up-to-date application state
    - *backup server*: delayed knowledge reading a log made by primary
    - certain hardware and operating system failures can be masked
- MVS operating system
  - processor with several CPUs:
    i. reading a CPU register: parity exception $\rightarrow$ automatic CPU retry
       - successful masking: failure data log, operator notification
       - failed CPU retry: report crash CPU failure by interrupt to MVS

    ii. re-execute the program on an alternate CPU

Stratus / 1.

Stratus processor architecture:



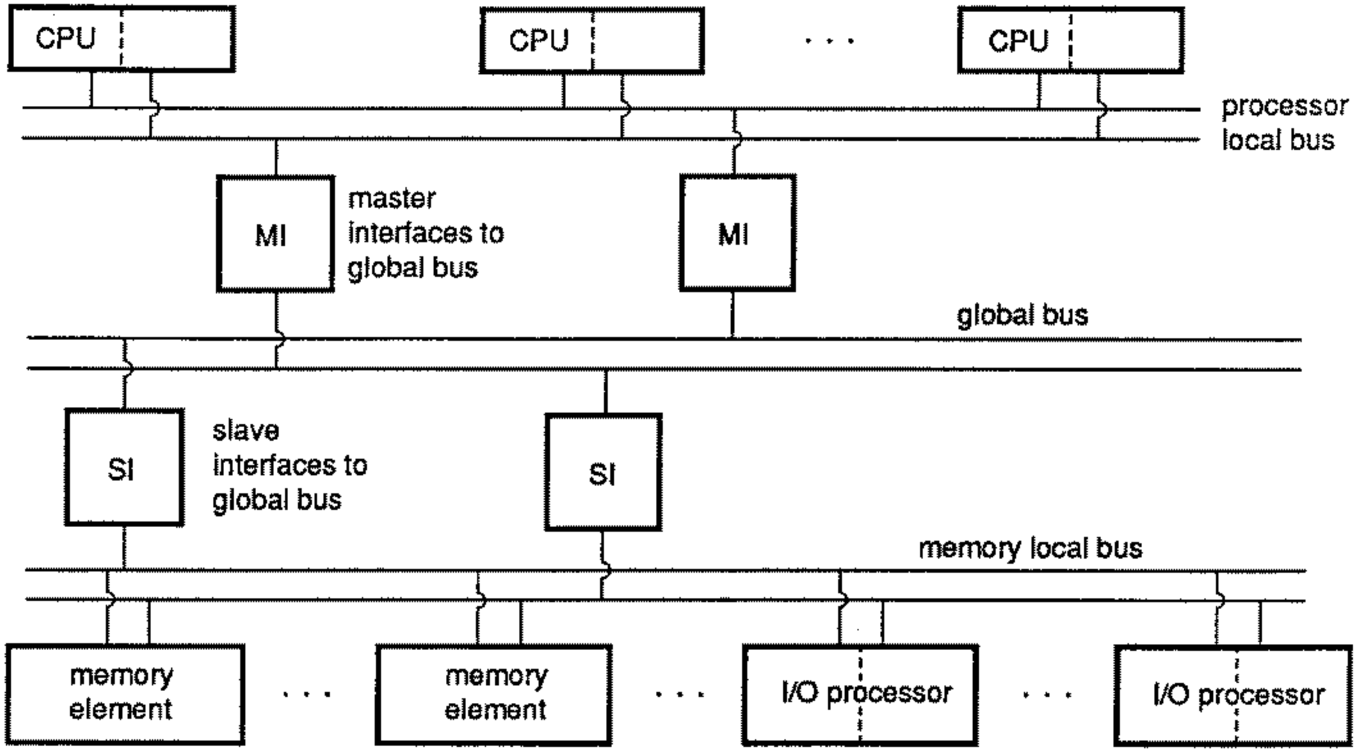
Stratus / 2.

- Stratus hardware architecture
  - systematic use of duplication and matching:
    - CPU, I / O and communication controllers: crash failure semantics
    - elementary HW servers are customer replaceable, own power supply
  - group masking at hardware level:

    (pairing elementary servers with crash / omission failure semantics)
    - CPUs: pair of identical microprocessors executing in lockstep
      output from any / functioning member
    - memories: read omission failure semantics
      2 bit detection / 1 bit correction modified Hamming ECC
      single-fault tolerant memory service with read omission ~
  - communication: StrataBUS, StrataLINK (dual)
  - dual hardware and operating system failures are visible to users

Sequoia / 1.

Sequoia multiprocessor architecture:



Sequoia / 2.

- Sequoia hardware architecture
  - customer replaceable units:
    - CPU and I / O controllers: paired μPs executing in lockstep
    - memory servers: error detecting / correcting codes
  - fault-tolerant communication: dual system bus
    - processor segment ↔ global segment ↔ memory segment
    - *master* and *slave interfaces* provide electrical isolation
- Operating system: hierarchical fault masking
  - atomic changes: shadow updating of state changes
    - two copies of the state on different memory units + pointer
  - masking of memory read omission failures
    - **writable** memory pages: duplicated on different memory units
    - **read-only** and **executable** memory pages: stored on disk

Hardware implementation issues / 1.

- Typical assumptions

CPU, I / O and communication controllers	crash failure semantics
memory elements	read omission failure semantics
disks	seek performance, read / write omission failure semantics
communication buses, lines	omission or omission / performance failure semantics

- Designers use hierarchical and group masking techniques
  - storage duplexing to mask loss of data:
    - replicated on two memory or disk servers with read omission ~
  - virtual circuits to mask omission or performance failure semantics:
    - communication failures are masked using time-outs, sequence numbers, positive / negative acknowledgment schemes, and retries (e.g. TCP)
  - strong hardware failure semantics → OS services will have strong ~

Hardware implementation issues / 2.

- Failure detection
  a. **EDC**: buses, communication lines, memories, disks
  b. **lockstep duplication**: CPU, I / O, and communication controllers
- Duplication and matching (master–checker)
  – advantages:
    - better crash failure semantics properties
    - no detection delay, faster servers
    - lower complexity (no EDC circuitry) → increased reliability
    - reduced design and testing costs, rapid redesign, simple upgrade
    - can be built using cheap, commercial, non fault-tolerant, fast μPs
    - improved software quality growth
  – disadvantages:
    - double processor cost

Software architectural issues

Software servers are analogous to hardware replaceable units.

- Software servers
  - basic units of failure, replacement and growth of software
    - goals: remove or install servers without service disruption, or
    - at least nice failure semantics (*crash*, *omission*, *performance*)
    - recovery using simple methods: “log in again”, “wait and try again”
- Failure semantics
  - persistent state: **atomic transaction**
    - a sequence of operations on persistent data
    - must change a consistent state into another or not change it at all
  - not persistent state: **amnesia / partial amnesia** crash
    - low-level I / O and communication controllers
    - must properly re-initialize themselves after a failure

Software failure semantics

- Assumptions on the programs
  - **totally** correct: behaves as specified if the used services do not fail
  - **partially** correct: may fail even if the used services do not fail
    - more easily achievable than total correctness
    - maintains its crash / performance failure semantics if a crash, omission or performance failure occurs in the lower-level servers
    - practically relevant: commercial software systems behave this way
- Software failure semantics implementation
  - pursuit of simplicity and clarity during design
  - hierarchical design methods using information hiding
  - rigorous design specification and verification techniques
  - systematic identification, detection, and handling of exception occurrences
  - use of modern inspection and testing methods

Software failure masking / 1.

The problems at the software architecture level are similar, solutions can be different.

- Hierarchical failure masking
  - mask crash, omission, and performance HW / SW server failures
  - group masking is needed to tolerate processor service failures
- Group failure masking
  - can mask a *minority* of arbitrary server failures in application groups
  - prerequisite: *multiple* host processors, *replication* of the resources
  - open problems:
    - how should group members maintain consistency of local states?
      - member failures, member joins, communication failures
    - how should group members communicate?
    - how is the required number of members maintained?
      - operating system, server, and communication failures

Software failure masking / 2.

- Group synchronization (continued)
  - the use of software server groups is not so widespread, because
    1. the (above) open problems are difficult to solve
    2. hardware failures have a longer repair time $\rightarrow$ manufacturers attack these types of failures first to reduce downtime
    3. there is an additional complexity related to group management
    4. it is not *a priori* clear that server groups can provide effective tolerance to residual software design faults
  - recent Tandem report:

0.7%	double server failures
99.3%	single member failures

    - faults are independent $\rightarrow$ servers crash independently
    - the effect residual software faults accumulates at different rates

Implementation issues: synchronization / 1.

Server group ~ policy: degree of local state synchronization among servers.

<u>Close synchronization (masking / active redundancy):</u>

Members execute all requests in parallel, same sequence of transitions.

- group output: depends on the member failure semantics
  - arbitrary failures: majority voting
  - crash, omission, or performance failures: any member output
- high communication overhead
  - **c-ranking** → output by the highest functioning c-ranked member
  - higher output delays when the highest c-ranked member fails:

detect failure of the	reach agreement about a new
highest c-ranked member	c-ranking among surviving members

Implementation issues: synchronization / 2.

Loose synchronization (dynamic / standby redundancy):

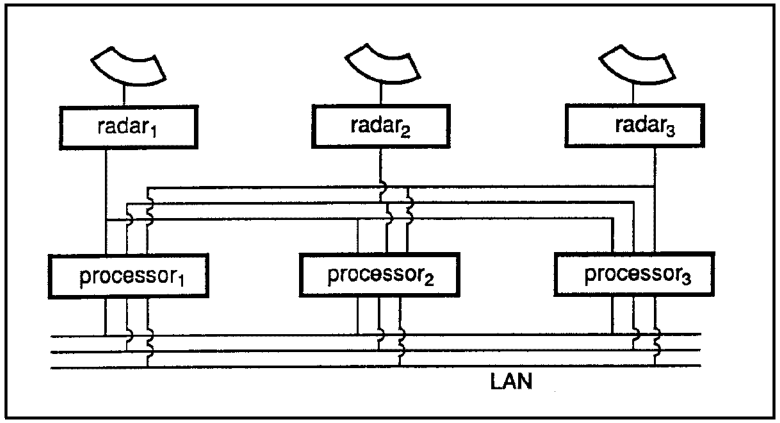
Highest *s-ranked* member processes requests, records current service state.

- **s-ranking**: how closely are different members to the current state
  - application of all operations since start-up
- backup servers: log service requests (checkpoints from primary)
  - if the primary fails → next highest s-ranked member
  - primary / backup: full / restricted use of replicated service resources
- group output: output by the primary member
  - crash, or crash / performance failure semantics
  - higher output delays when the primary member fails:

detect failure of the	reach agreement about	absorb the primary-
primary member	the new primary member	backup processing lag

Implementation issues: members

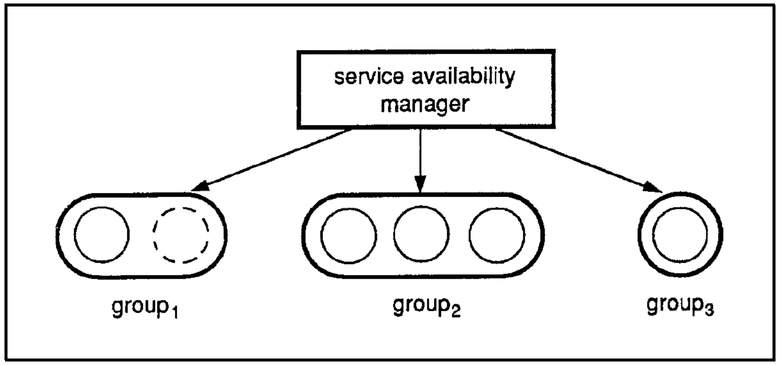
- Replication policy: prescribes the # of members in a group
  - more servers:
    - greater availability and processing capacity
    - higher communication and synchronization costs
  - stochastic modeling and simulation to find an optimum
    - individual server failure rates
    - service request arrival rates
    - server failure semantics
    - message communication costs
  - degree of necessary software server redundancy influences host processor redundancy



Implementation issues: availability / 1.

Group availability policy: collection of synchronization and replication policies for a service implemented by a server group.

5. Group member mechanisms, detect failures, handle joins
   - substantial code duplication, lack of modularity
6. Alternative approach
   - group members: application-specific get-state and checkpoint
   - service availability management:
     i. detect server failures
     ii. coordinate ranking promotions
     iii. enforce replication policies
   - implemented by a group of service availability managers



Implementation issues: availability / 2.

- Server availability managers

Components	Events	State transitions
service s	is enabled	If $r(s) \le \lvert h(s) \cap members \rvert$ then start servers on $r(s)$ processors, else on all processors. Add s to S, every p to P, and update mappings *server* and *service*.
	is disabled	Remove s from S and shut down all servers for s. Remove every p from P, update mappings *server*s and *service*.
server p implementing s	fails	If there is a free host in $h(s) \cap members$ start another server for s.
	is started	Update P, and mappings *servers* and *service*.
processor m hosting p	fails	Interpret failure of m as the failure of every p running on m. Remove m from the set *members*.
	is started	Start servers for all services that can be hosted by m and number of servers $\le r(s)$. Update variables P, *members*, *servers*, and *service*.

  - Global variables S, P, *members*, *service*, *servers* must be replicated
    - all processors must see the same sequence of state updates

Implementation issues: agreement / 1.

Random communication delays and crash, omission, and performance failures can cause messages broadcast for updating state replicas to be lost or received out of order.

- Agreement protocols
  - **processor membership**: sequence of processor joins and failures
  - **atomic broadcast**: order of broadcast messages
- Requirements
  - *safety* properties: invariants, must be true at all points in real-time
    - membership ensures processors always agree on group membership
    - atomic broadcast ensures processors receive messages in same order
  - *timeliness* properties: upper bound on information propagation
    - membership requires to detect member failures in bounded time
    - atomic broadcast requires to deliver messages in bounded time

Implementation issues: agreement / 2.

- Requirements (continued)
  - synchronous protocols: strong timeliness properties
    - propagation time is bounded even if failures and member joins occur
    - propagation speed $\leftarrow$ message delay d, clock deviation ε
    - *transient partitioning*: the protocol violates its safety requirements
  - asynchronous protocols: strong safety properties
    - one can build ~ that never violate their safety requirements, but

    i. they do not guarantee an *a priori* known upper bound

    ii. membership computation and broadcast can interfere $\rightarrow$ aborted

    iii. a quorum must be formed before any work can be done

Implementation issues: agreement / 3.

- Comparison of synchronous and asynchronous protocols

<table>
<tr><th></th><th colspan="2">Synchronous</th><th>Asynchronous</th></tr>
<tr><td rowspan="3">Requires</td><td>Knowledge of maximum system load, real-time operating system, massive communication hardware redundancy.</td><td>a time-out delay d’,
a synchronous approach based on it provides sufficient recovery speed</td><td></td></tr>
<tr><td rowspan="2">Upper bound d allowing a satisfactory recovery speed.</td><td colspan="2">Upper bound d is not small enough to allow a satisfactory recovery speed → a time-out delay $d' < d$</td></tr>
<tr><td>cost of compensating for inconsistent actions during transient partitions < cost of missing recovery deadlines</td><td>cost of compensating for inconsistent actions during transient partitions > cost of missing recovery deadlines</td></tr>
<tr><td>Results</td><td>Strong timeliness properties, autonomous operation (at least one fault-free processor)</td><td>Strong timeliness properties, no quorums</td><td>weak timeliness properties, additional tasks for quorum maintenance</td></tr>
</table>