Fault Tolerance in Distributed Systems

Fault Tolerant Distributed Systems

Prof. Nalini Venkatasubramanian
(with some slides modified from Prof. Ghosh, University of Iowa and Indranil Gupta, UIUC)

Fundamentals

- What is fault?
  - A fault is a blemish, weakness, or shortcoming of a particular hardware or software component.
  - Fault, error and failures
- Why fault tolerant?
  - Availability, reliability, dependability, …
- How to provide fault tolerance ?
  - Replication
  - Checkpointing and message logging
  - Hybrid

Reliability

- Reliability is an emerging and critical concern in traditional and new settings
  - Transaction processing, mobile applications, cyberphysical systems
- New enhanced technology makes devices vulnerable to errors due to high complexity and high integration
  - Technology scaling causes problems
    - Exponential increase of soft error rate
  - Mobile/pervasive applications running close to humans
    - E.g Failure of healthcare devices cause serious results
  - Redundancy techniques incur high overheads of power and performance
    - TMR (Triple Modular Redundancy) may exceed 200% overheads without optimization [Nieuwland, 06]
- Challenging to optimize multiple properties (e.g., performance, QoS, and reliability)

Classification of failures

Crash failure

Security failure

Omission failure

Temporal failure

Transient failure

Byzantine failure

Software failure

Environmental perturbations

Crash failures

Crash failure = the process halts. It is *irreversible*.

In synchronous system, it is easy to detect crash failure (using *heartbeat signals* and timeout). But in asynchronous systems, it is never accurate, since it is not possible to distinguish between a process that has crashed, and a process that is running very slowly.

Some failures may be complex and nasty. Fail-stop failure is a *simple abstraction* that *mimics* crash failure when program execution becomes arbitrary. Implementations help detect which processor has failed. If a system cannot tolerate fail-stop failure, then it cannot tolerate crash.

Transient failure

(Hardware) Arbitrary perturbation of the global state. May be induced by power surge, weak batteries, lightning, radio-frequency interferences, cosmic rays etc.

Not *Heisenberg*

(Software) Heisenbugs are a class of temporary internal faults and are intermittent. They are essentially permanent faults whose conditions of activation occur rarely or are not easily reproducible, so they are harder to detect during the testing phase.

Over 99% of bugs in IBM DB2 production code are non-deterministic and transient (Jim Gray)

Temporal failures

Inability to meet deadlines – correct results are generated, but too late to be useful. Very important in real-time systems.

May be caused by poor algorithms, poor design strategy or loss of synchronization among the processor clocks

Byzantine failure

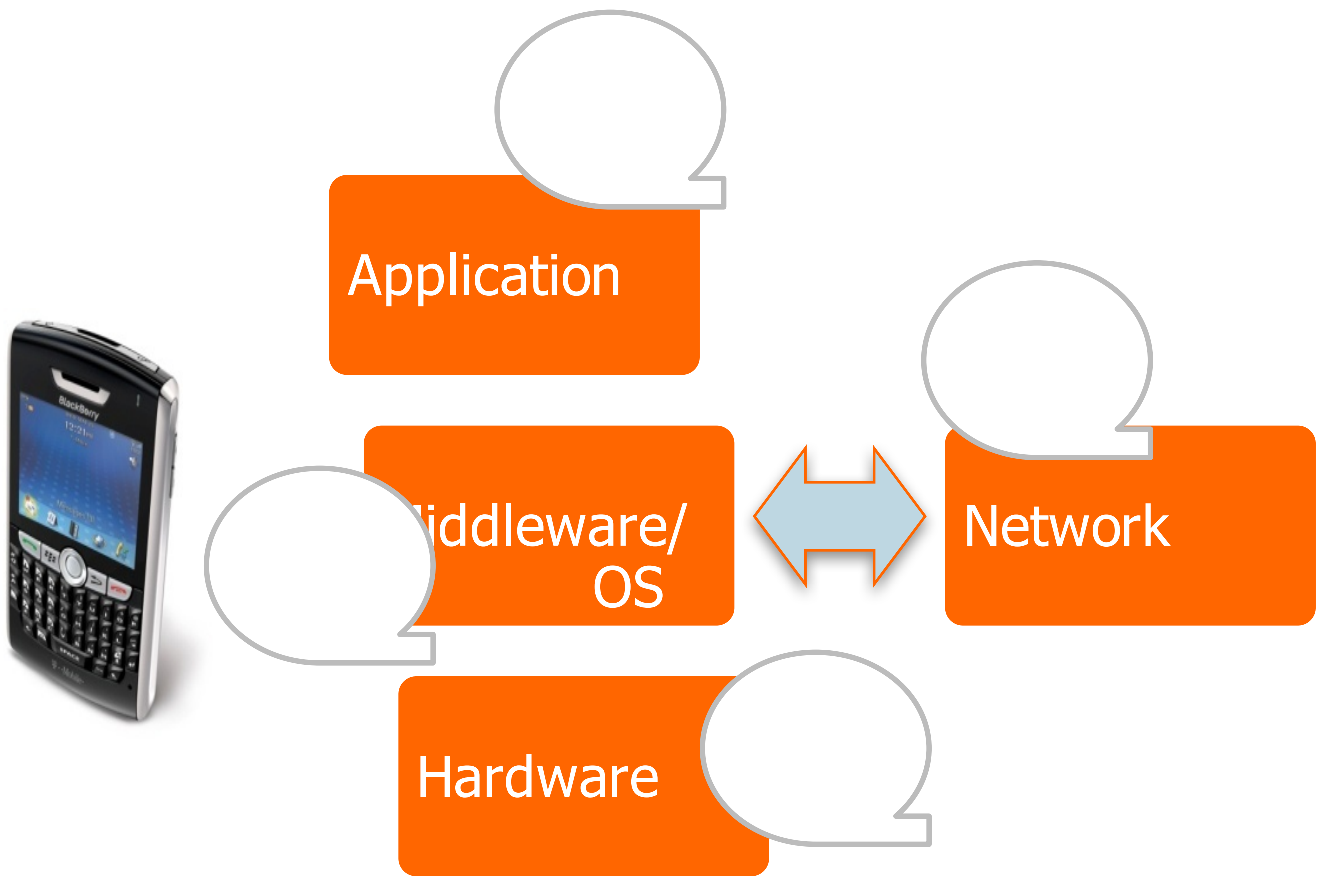
Anything goes! Includes every conceivable form of erroneous behavior. The weakest type of failure

Numerous possible causes. Includes malicious behaviors (like a process executing a different program instead of the specified one) too.

Most difficult kind of failure to deal with.

Errors/Failures across system layers

- Faults or Errors can cause Failures



Hardware Errors and Error Control Schemes

Failures	Causes	Metrics	Traditional Approaches
Soft Errors, Hard Failures, System Crash	External Radiations, Thermal Effects, Power Loss, Poor Design, Aging	FIT, MTTF, MTBF	Spatial Redundancy (TMR, Duplex, RAID-1 etc.) and Data Redundancy (EDC, ECC, RAID-5, etc.)

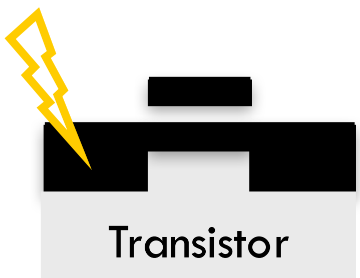
- Hardware failures are increasing as technology scales
  - (e.g.) SER increases by up to 1000 times [Mastipuram, 04]
- Redundancy techniques are expensive
  - (e.g.) ECC-based protection in caches can incur 95% performance penalty [Li, 05]

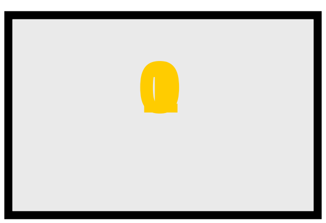
•FIT: Failures in Time (10^9 hours)
•MTTF: Mean Time To Failure
•MTBF: Mean Time b/w Failures
•TMR: Triple Modular Redundancy
•EDC: Error Detection Codes
•ECC: Error Correction Codes
•RAID: Redundant Array of Inexpensive Drives

Soft Errors (Transient Faults)

- SER increases exponentially as technology scales
- Integration, voltage scaling, altitude, latitude

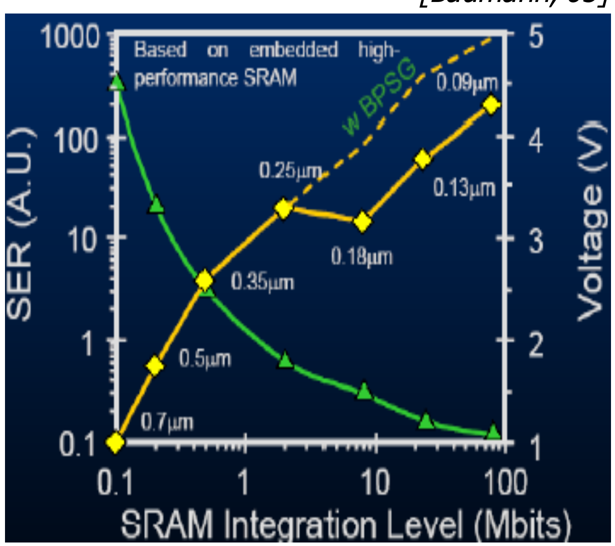
- Caches are most hit due to:
  - Larger portion in processors (more than 50%)
  - No masking effects (e.g., logical masking)

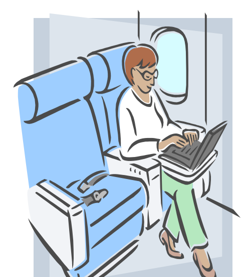




Bit Flip

[Baumann, 05]

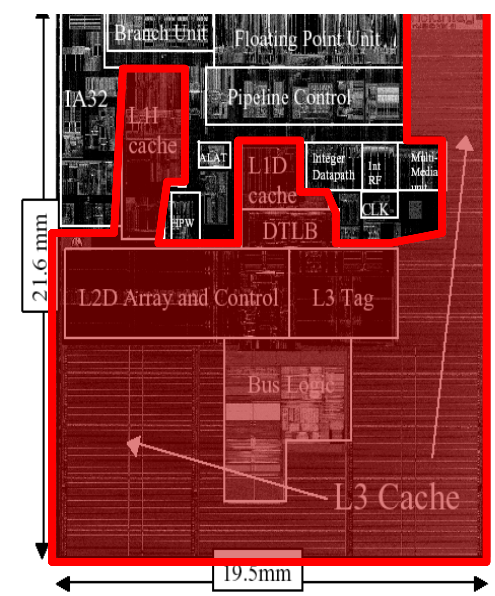




5 hours MTTF

1 month MTTF

Intel Itanium II Processor



•MTTF: Mean time To Failure

Soft errors

	SER (FIT)	MTTF	Reason
1 Mbit @ 0.13 µm	1000	104 years	
64 MB @ 0.13 µm	64x8x1000	81 days	High Integration
128 MB @ 65 nm	2x1000x64x8x1000	1 hour	Technology scaling and Twice Integration
A system @ 65 nm	2x2x1000x64x8x1000	30 minutes	Memory takes up 50% of soft errors in a system
A system with voltage scaling @ 65 nm	100x2x2x1000x64x8x1000	18 seconds	Exponential relationship b/w SER & Supply Voltage
A system with voltage scaling @ flight (35,000 ft) @ 65 nm	800x100x2x2x1000x64x8x1000 FIT	0.02 seconds	High Intensity of Neutron Flux at flight (high altitude)

Soft Error Rate (SER) – FIT (Failures in Time) = number of errors in 10^9 hours

Software Errors and Error Control Schemes

Failures	Causes	Metrics	Traditional Approaches
Wrong outputs, Infinite loops, Crash	Incomplete Specification, Poor software design, Bugs, Unhandled Exception	Number of Bugs/Klines, QoS, MTTF, MTBF	Spatial Redundancy (N-version Programming, etc.), Temporal Redundancy (Checkpoints and Backward Recovery, etc.)

- Software errors become dominant as system's complexity increases
  - (e.g.) Several bugs per kilo lines
- Hard to debug, and redundancy techniques are expensive
  - (e.g.) Backward recovery with checkpoints is inappropriate for real-time applications

•QoS: Quality of Service

Software failures

Coding error or human error

On September 23, 1999, NASA lost the $125 million Mars orbiter spacecraft because one engineering team used metric units while another used English units leading to a navigation fiasco, causing it to burn in the atmosphere.

Design flaws or inaccurate modeling

Mars pathfinder mission landed flawlessly on the Martial surface on July 4, 1997. However, later its communication failed due to a design flaw in the real-time embedded software kernel VxWorks. The problem was later diagnosed to be caused due to *priority inversion*, when a medium priority task could preempt a high priority one.

Software failures

Memory leak

Processes fail to entirely free up the physical memory that has been allocated to them. This effectively reduces the size of the available physical memory over time. When this becomes smaller than the minimum memory needed to support an application, it crashes.

Incomplete specification (example Y2K)

Year = 99 (1999 or 2099)?

Many failures (like crash, omission etc) can be caused by software bugs too.

Network Errors and Error Control Schemes

Failures	Causes	Metrics	Traditional Approaches
Data Losses, Deadline Misses, Node (Link) Failure, System Down	Network Congestion, Noise/Interference, Malicious Attacks	Packet Loss Rate, Deadline Miss Rate, SNR, MTTF, MTBF, MTTR	Resource Reservation, Data Redundancy (CRC, etc.), Temporal Redundancy (Retransmission, etc.), Spatial Redundancy (Replicated Nodes, MIMO, etc.)

- Omission Errors – lost/dropped messages
- Network is unreliable (especially, wireless networks)
  - Buffer overflow, Collisions at the MAC layer, Receiver out of range
- Joint approaches across OSI layers have been investigated for minimal costs [Vuran, 06][Schaar, 07]

•SNR: Signal to Noise Ratio
•MTTR: Mean Time To Recovery
•CRC: Cyclic Redundancy Check
•MIMO: Multiple-In Multiple-Out

Classifying fault-tolerance

Masking tolerance.
Application runs as it is. The failure does not have a visible impact.
All properties (both liveness & safety) continue to hold.

Non-masking tolerance.
Safety property is *temporarily affected*, but not liveness.

Example 1. Clocks lose synchronization, but recover soon thereafter.
Example 2. Multiple processes temporarily enter their critical sections, but thereafter, the normal behavior is restored.

Classifying fault-tolerance

Fail-safe tolerance

Given safety predicate is preserved, but liveness may be affected

Example. Due to failure, no process can enter its critical section for an indefinite period. In a traffic crossing, failure changes the traffic in both directions to red.

Graceful degradation

Application continues, but in a "degraded" mode. Much depends on what kind of degradation is acceptable.

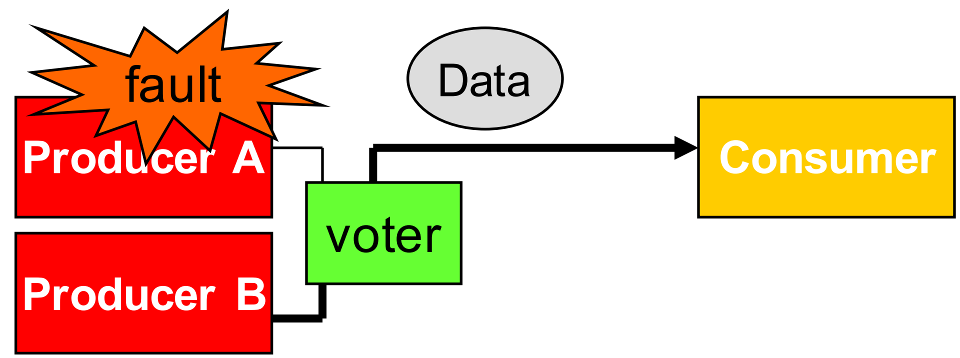
Example. Consider message-based mutual exclusion. Processes will enter their critical sections, but not in timestamp order.

Conventional Approaches

- Build redundancy into hardware/software
  - Modular Redundancy, N-Version ProgrammingConventional TRM (Triple Modular Redundancy) can incur 200% overheads without optimization.
  - Replication of tasks and processes may result in overprovisioning
  - Error Control Coding
- Checkpointing and rollbacks
  - Usually accomplished through logging (e.g. messages)
  - Backward Recovery with Checkpoints cannot guarantee the completion time of a task.
- Hybrid
  - Recovery Blocks

1) Modular Redundancy

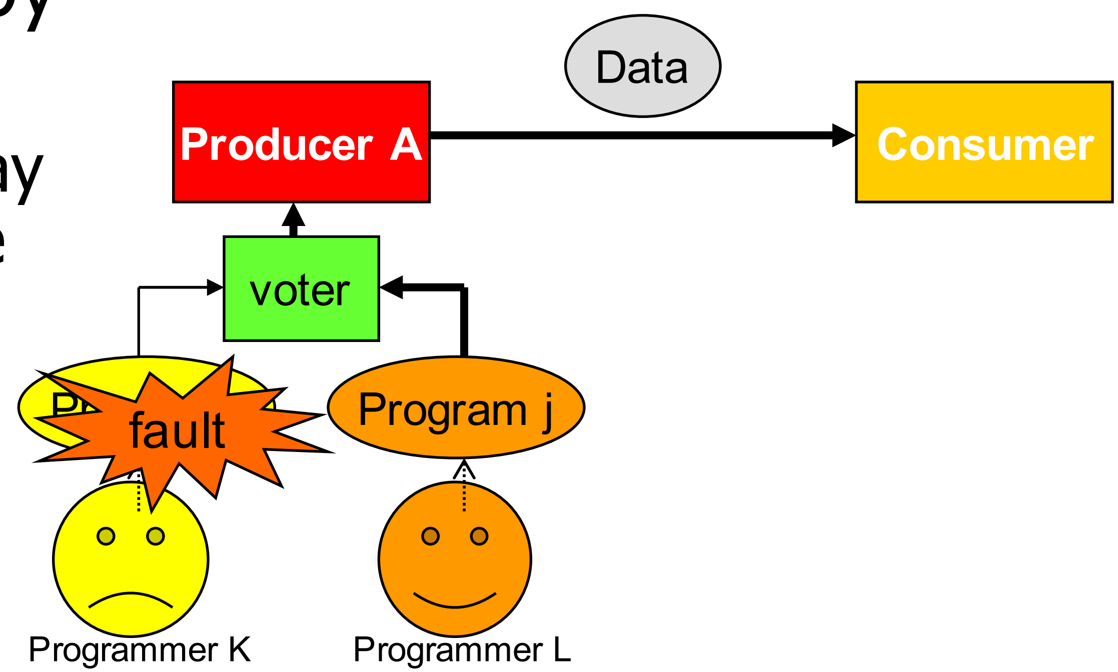
- Modular Redundancy
  - Multiple identical replicas of hardware modules
  - Voter mechanism
    - Compare outputs and select the correct output
  - ➔Tolerate most hardware faults
  - ➔Effective but expensive



2) N-version Programming

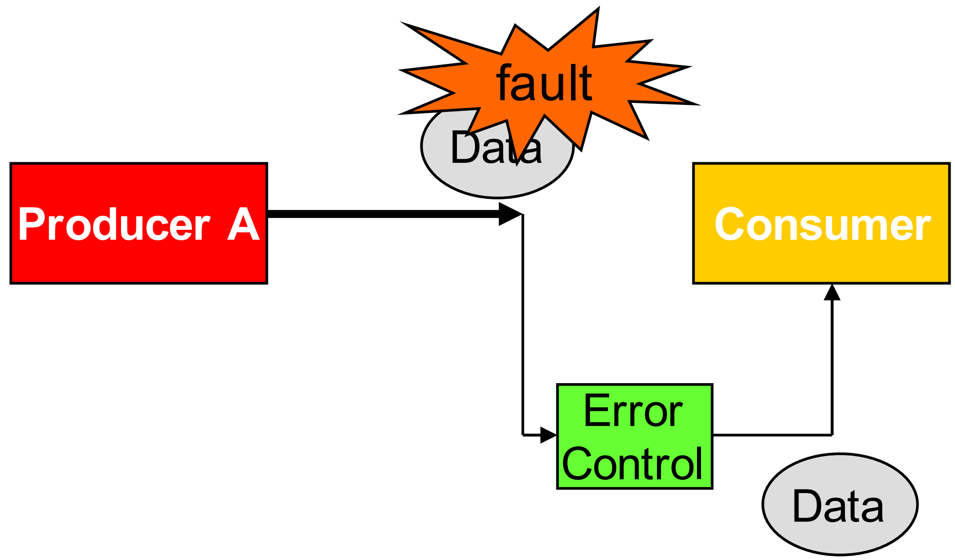
- N-version Programming
  - Different versions by different teams
    - Different versions may not contain the same bugs
  - Voter mechanism

➔Tolerate some software bugs



3) Error-Control Coding

- Error-Control Coding
  - Replication is effective but expensive
  - Error-Detection Coding and Error-Correction Coding
    - (example) Parity Bit, Hamming Code, CRC
  - ➔ Much less redundancy than replication



Concept: Consensus

Reaching Agreement is a fundamental problem in distributed computing

- Mutual Exclusion
  - processes agree on which process can enter the critical section
- Leader Election
  - the processes agree on which is the elected process
- Totally Ordered Multicast
  - the processes agree on the order of message delivery
- Commit or Abort in distributed transactions
- Reaching agreement about which process has failed
- Other examples
  - Air traffic control system: all aircrafts must have the same view
  - Spaceship engine control – action from multiple control processes(“proceed” or “abort”)
  - Two armies should decide consistently to attack or retreat.

Defining Consensus

- N processes
- Each process p has
  - input variable xp : initially either 0 or 1
  1. output variable yp : initially b (b=undecided) – can be changed only once
- Consensus problem: design a protocol so that either
  1. all non-faulty processes set their output variables to 0
  2. Or non-faulty all processes set their output variables to 1
  3. There is at least one initial state that leads to each outcomes 1 and 2 above

Solving Consensus

- No failures – trivial
  - All-to-all broadcast
- With failures
  - Assumption: Processes fail only by *crash-stop*ping
- Synchronous system: bounds on
  - Message delays
  - Max time for each process step

  e.g., multiprocessor (common clock across processors)
- Asynchronous system: no such bounds!

  e.g., The Internet! The Web!

Variant of Consensus Problem

- Consensus Problem (C)
  - Each process propose a value
  - All processes agree on a single value
- Byzantine General Problem (BG)
  - Process fails arbitrarily, byzantine failure
  - Still processes need to agree
- Interactive Consistency (IC)
  - Each process propose its value
  - All processes agree on the vector

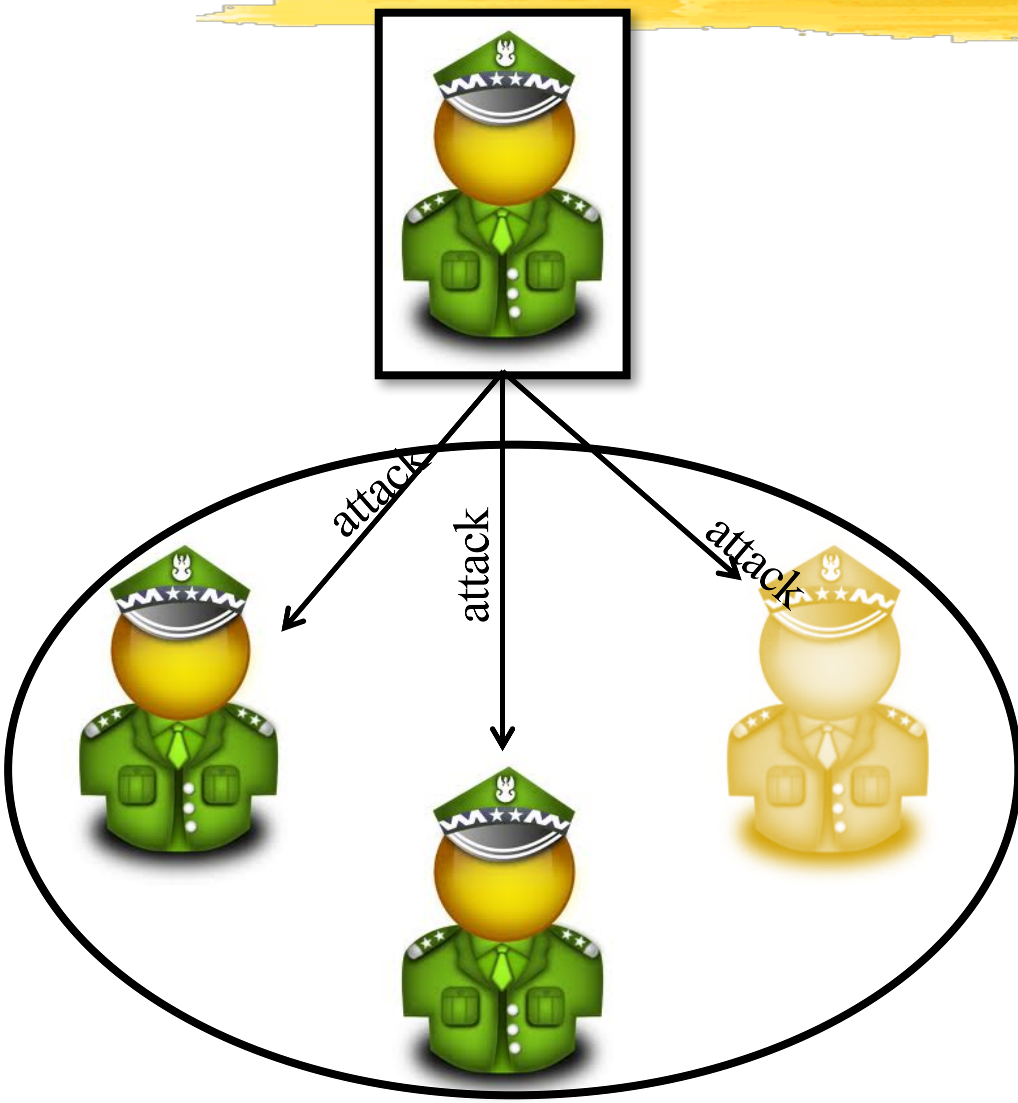
Consensus in Synchronous System

- Possible
  - With one or more faulty processes
- Solution:
  - Basic idea: all processes exchange (multicast) what other processes tell them in several rounds
- Proof: to reach consensus with f failures, the algorithm needs to run in $f + 1$ rounds
  - Basic idea: if *A* and *B* do not agree on value X, then some other process C, did send *X* to *A*, but not to *B*.
  - Requires 1 failure in each round, so f+1 fails. Contradiction!

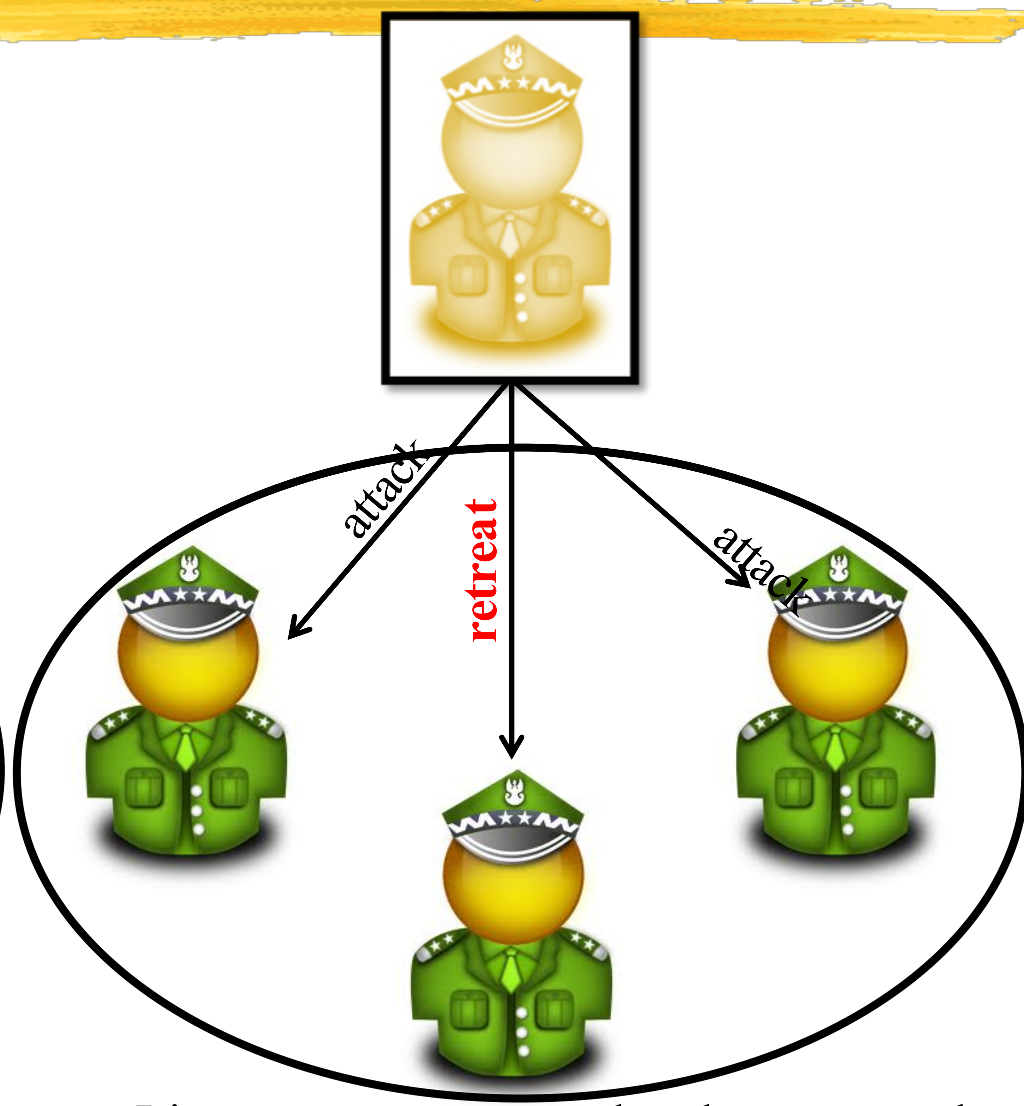
Asynchronous Consensus

- Messages have arbitrary delay, processes arbitrarily slow
- Impossible to achieve!
  - a slow process indistinguishable from a crashed process
- **Theorem:** In a purely asynchronous distributed system, the consensus problem is impossible to solve if even a single process crashes
- Result due to Fischer, Lynch, Patterson (commonly known as FLP 85).

Byzantine General Problem



Lieutenants agree on what the commander says



Lieutenants agree on what the commander says

Byzantine General Problem

The Byzantine generals problem • In the informal statement of the *Byzantine generals problem* [Lamport *et al.* 1982], three or more generals are to agree to attack or to retreat. One, the commander, issues the order. The others, lieutenants to the commander, must decide whether to attack or retreat. But one or more of the generals may be 'treacherous' – that is, faulty. If the commander is treacherous, he proposes attacking to one general and retreating to another. If a lieutenant is treacherous, he tells one of his peers that the commander told him to attack and another that they are to retreat.

The Byzantine generals problem differs from consensus in that a distinguished process supplies a value that the others are to agree upon, instead of each of them proposing a value. The requirements are:

Termination: Eventually each correct process sets its decision variable.

Agreement: The decision value of all correct processes is the same: if p_i and p_j are correct and have entered the *decided* state, then $d_i = d_j$ $(i, j = 1, 2, \ldots, N)$.

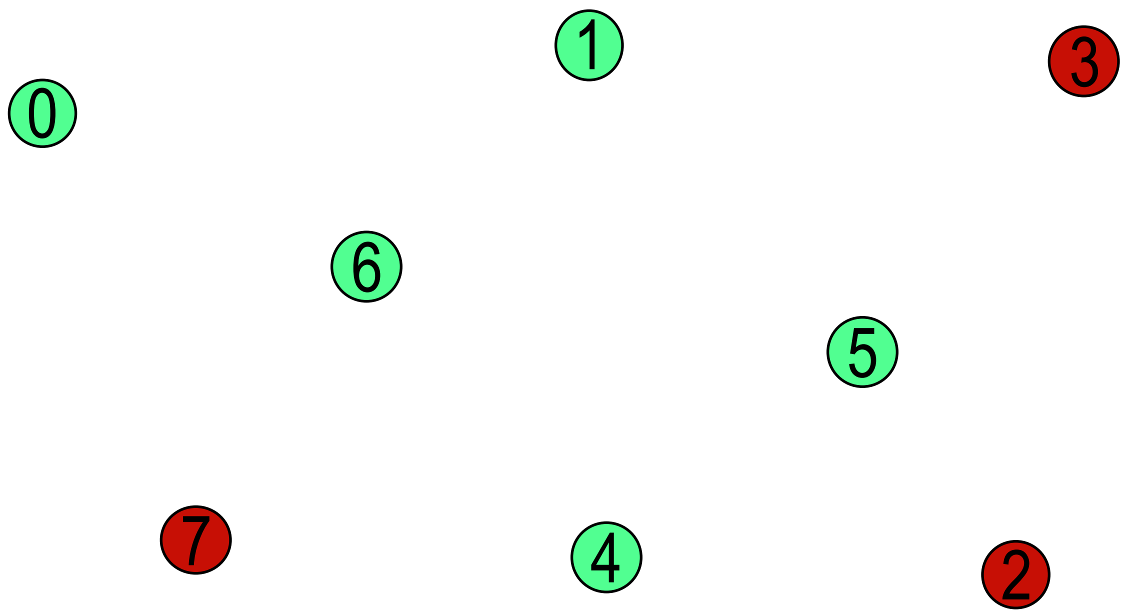
Integrity: If the commander is correct, then all correct processes decide on the value that the commander proposed.

Failure detection

The design of fault-tolerant algorithms will be simple if processes can detect failures.

- In synchronous systems with bounded delay channels, crash failures can definitely be detected using timeouts.
- In asynchronous distributed systems, the detection of crash failures is imperfect.
- ***Completeness*** – Every crashed process is suspected
- ***Accuracy*** – No correct process is suspected.

Example



0 suspects {1,2,3,7} to have failed. Does this satisfy completeness?
Does this satisfy accuracy?

Classification of completeness

- **Strong completeness.** Every crashed process is eventually suspected by *every* correct process, and remains a suspect thereafter.

- **Weak completeness.** Every crashed process is eventually suspected by *at least one* correct process, and remains a suspect thereafter.

Note that we don't care what mechanism is used for suspecting a process.

Classification of accuracy

- **Strong accuracy.** No correct process is ever suspected.

- **Weak accuracy.** There is at least one correct process that is never suspected.

Eventual accuracy

A failure detector is *eventually strongly accurate*, if there exists a time **T** after which no correct process is suspected.

(*Before that time, a correct process be added to and removed from the list of suspects any number of times*)

A failure detector is *eventually weakly accurate*, if there exists a time **T** after which at least one process is no more suspected.

Classifying failure detectors

Perfect P. (Strongly) Complete and strongly accurate

Strong S. (Strongly) Complete and weakly accurate

Eventually perfect ◊P.

(Strongly) Complete and eventually strongly accurate

Eventually strong ◊S

(Strongly) Complete and eventually weakly accurate

Other classes are feasible: W (weak completeness) and weak accuracy) and **◊W**

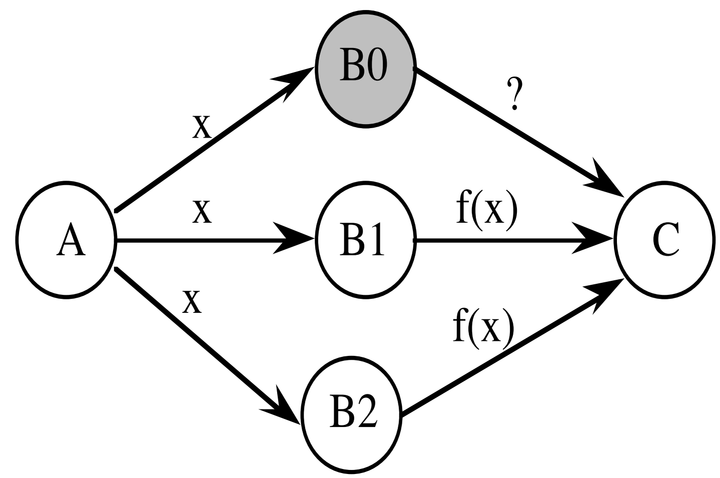
Detection of crash failures

Failure can be detected using heartbeat messages (periodic "I am alive" broadcast) and timeout

- if processors speed has a known lower bound
- channel delays have a known upper bound.

Tolerating crash failures

Triple modular redundancy (TMR) for masking any single failure. *N-modular redundancy masks up to m failures, when N = 2m +1.*



Take a vote

What if the voting unit fails?

Detection of omission failures

For **FIFO** channels: Use sequence numbers with messages.

(1, 2, 3, 5, 6 …) $\Rightarrow$ message 4 is missing

Non-FIFO bounded delay channels - use timeout

What about non-FIFO channels for which the upper bound of the delay is not known?

Use *unbounded sequence numbers* and acknowledgments.

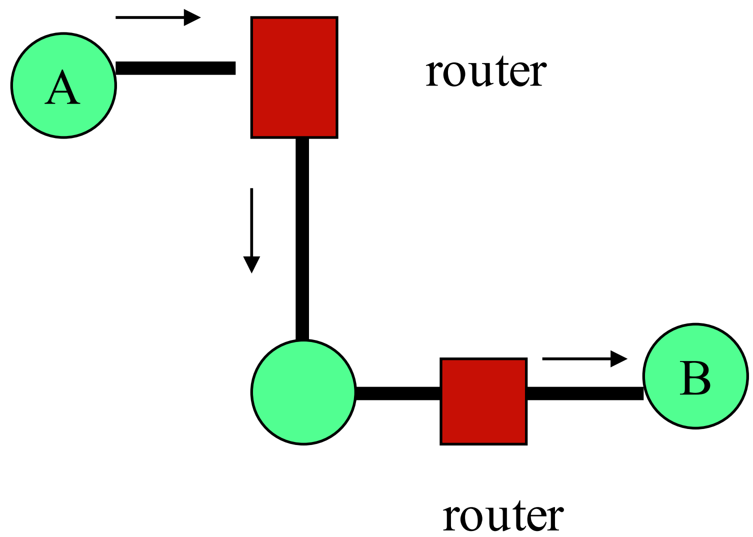
But acknowledgments may be lost too!

Let us look how a real protocol deals with omission ….

Tolerating omission failures

A central issue in networking

Routers may drop messages, but reliable end-to-end transmission is an important requirement. If the sender does not receive an ack within a time period, it retransmits (it may so happen that the was not lost, so a duplicate is generated). This implies, the communication must tolerate **Loss, Duplication, and Re-ordering** of messages



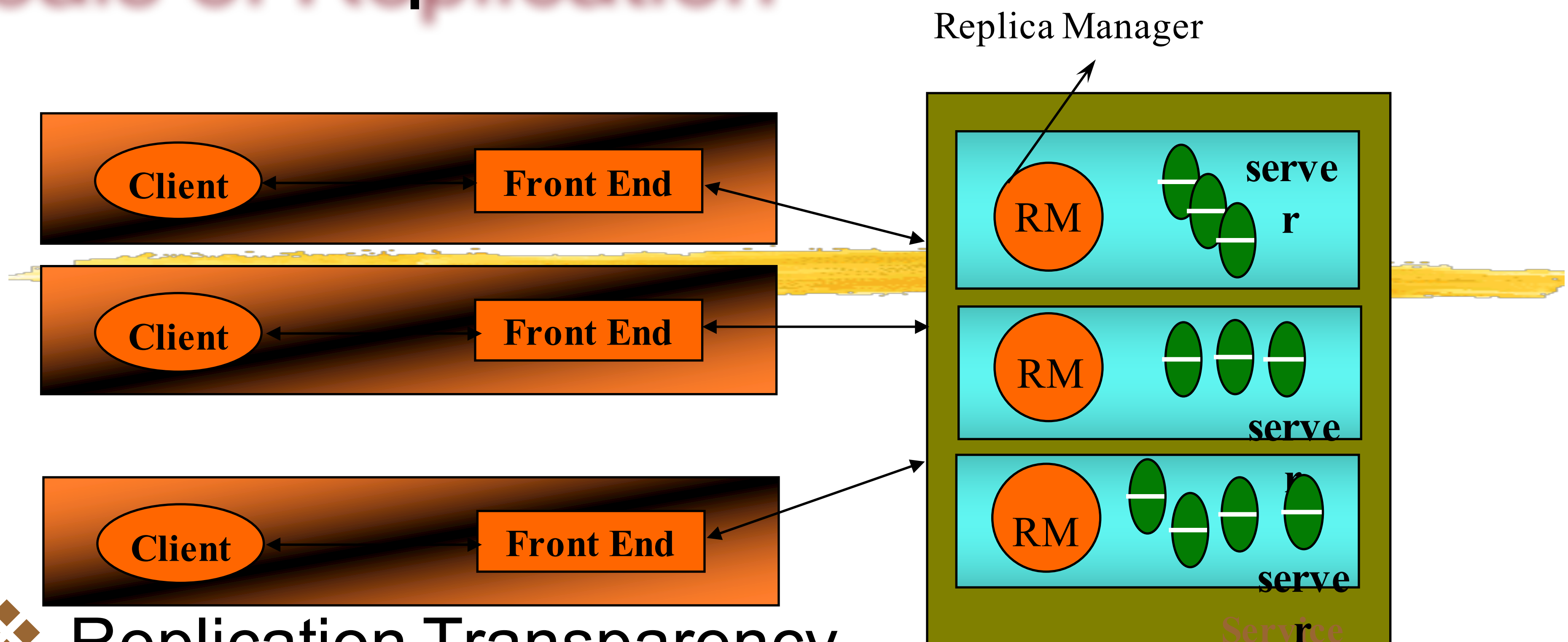
Replication

- Enhances a service by replicating data
  - Increased Availability
    - Of service. When servers fail or when the network is partitioned.
  - Fault Tolerance
    - Under the fail-stop model, if up to *f* of *f+1* servers crash, at least one is alive.
  - Load Balancing
    - One approach: Multiple server IPs can be assigned to the same name in DNS, which returns answers round-robin.

P: probability that one server fails $= 1 - P =$ availability of service.
e.g. $P = 5\%$ => service is available 95% of the time.

P^n: probability that n servers fail $= 1 - P^n =$ availability of service.
e.g. $P = 5\%$, $n = 3$ => service available 99.875% of the time

Goals of Replication



- **Replication Transparency**

  User/client need not know that multiple physical copies of data exist.

- **Replication Consistency**

  Data is consistent on all of the replicas (or is converging towards becoming consistent)

Replication Management

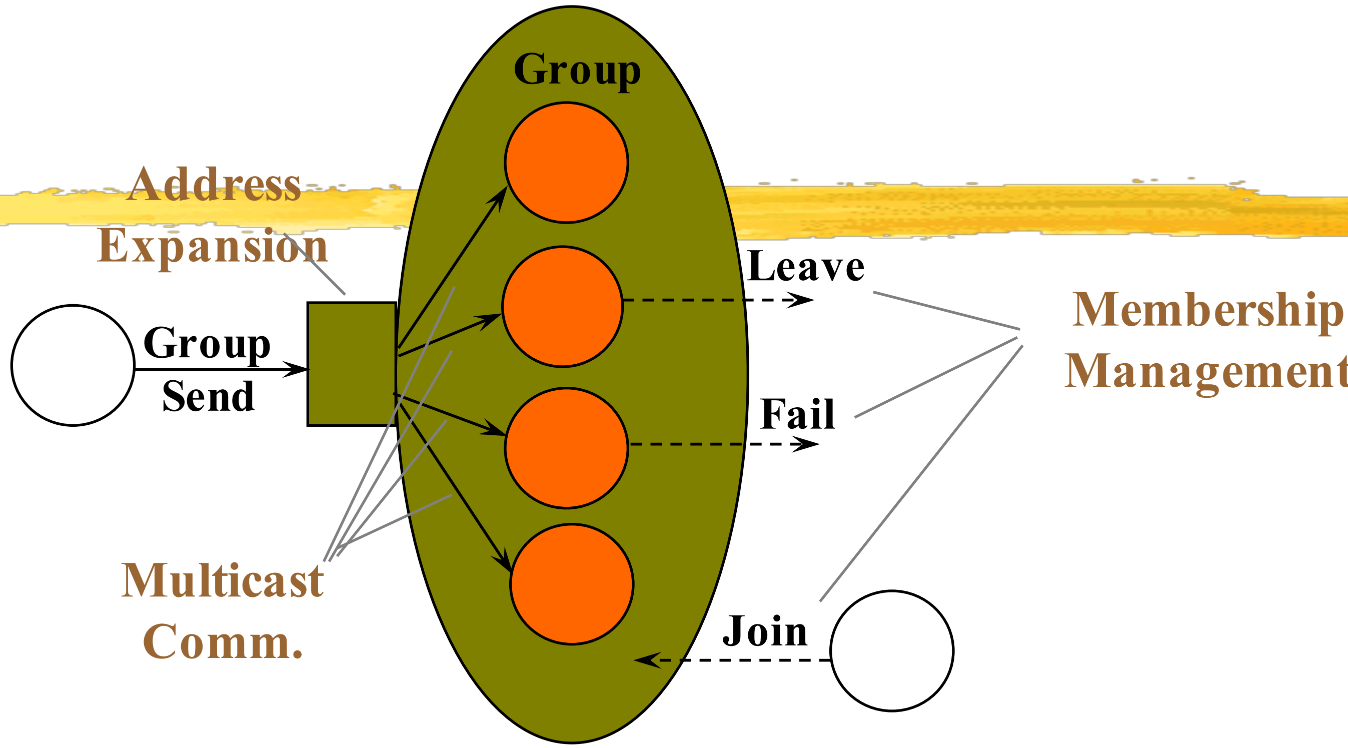
- Request Communication
  - Requests can be made to a single RM or to multiple RMs
- Coordination: The RMs decide
  - whether the request is to be applied
  - the order of requests
    - FIFO ordering: If a FE issues *r* then *r'*, then any correct RM handles *r* and then *r'*.
    - Causal ordering: If the issue of *r* "happened before" the issue of *r'*, then any correct RM handles *r* and then *r'*.
    - Total ordering: If a correct RM handles *r* and then *r'*, then any correct RM handles *r* and then *r'*.
- Execution: The RMs execute the request (often they do this tentatively).

Replication Management

- Agreement: The RMs attempt to reach consensus on the effect of the request.
  - E.g., Two phase commit through a coordinator
  - If this succeeds, effect of request is made permanent
- Response
  - One or more RMs responds to the front end.
  - The first response to arrive is good enough because all the RMs will return the same answer.
  - Thus each RM is a replicated state machine

    "Multiple copies of the same State Machine begun in the Start state, and receiving the same Inputs in the same order will arrive at the same State having generated the same Outputs." [Wikipedia, Schneider 90]

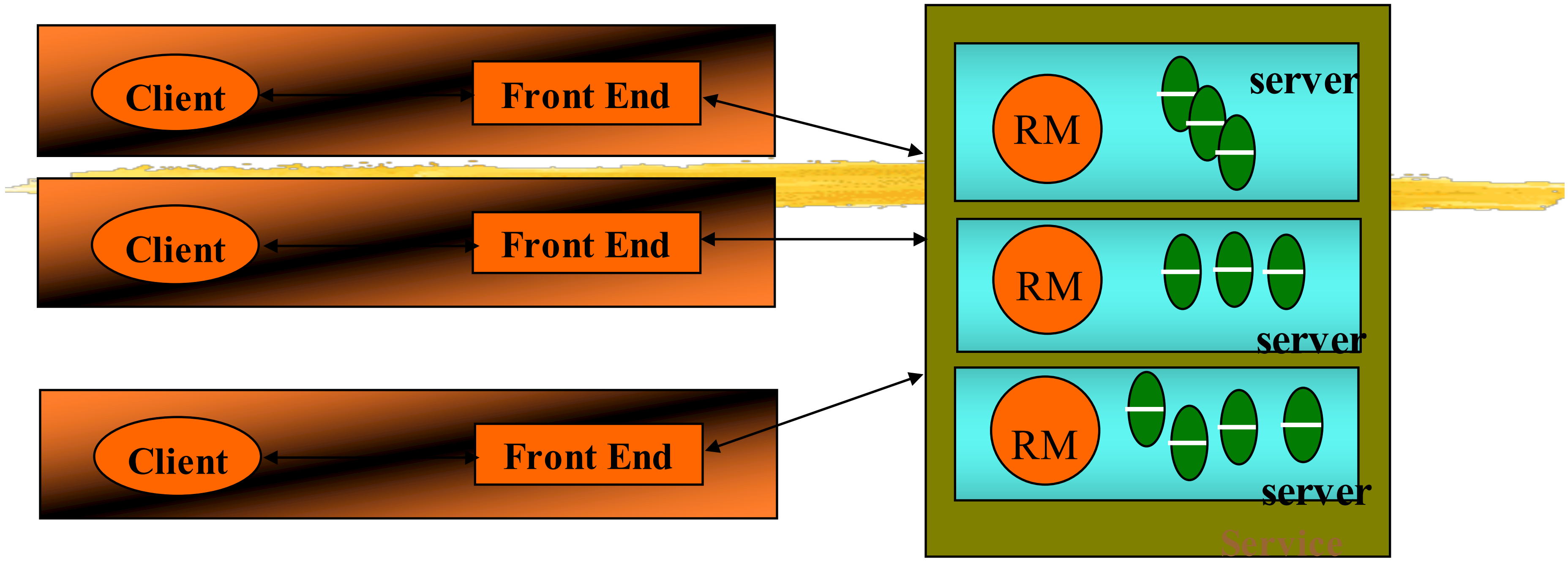
Group Communication: A building block



- "Member" = process (e.g., an RM)
- Static Groups: group membership is pre-defined
- Dynamic Groups: Members may join and leave, as necessary

EXTRA SLIDES

Replication using GC



Need consistent updates to all copies of an object

•Linearizability

•Sequential Consistency

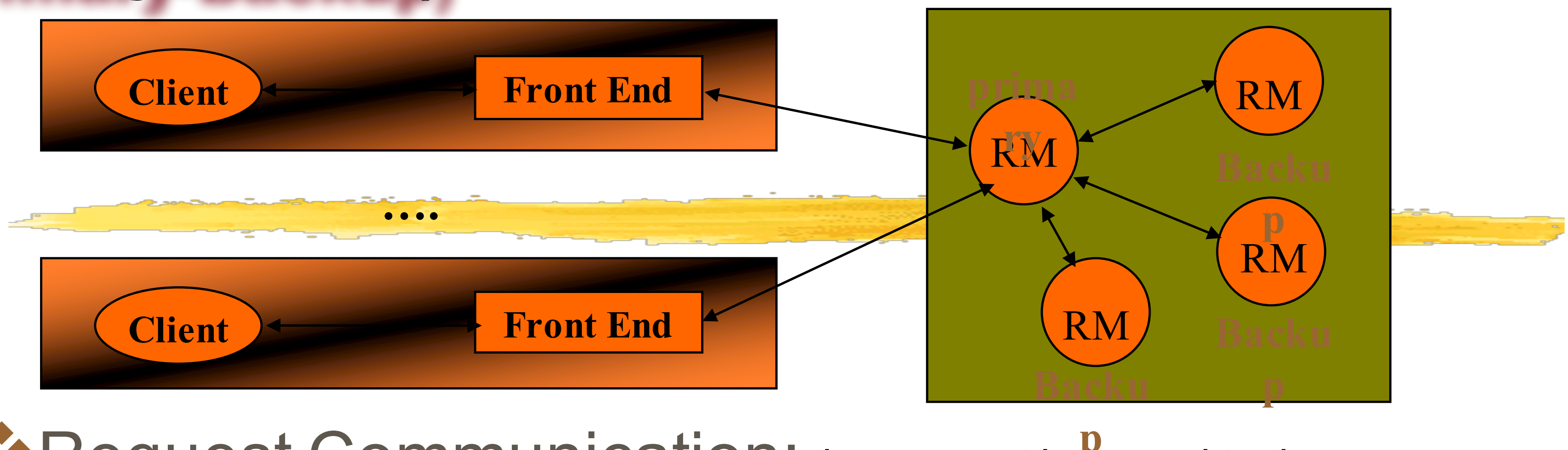
Linearizability

- Let the sequence of read and update operations that client *i* performs in some execution be o_{i1}, o_{i2},….
  - "Program order" for the client
- A replicated shared object service is linearizable if for any execution (real), there is some interleaving of operations (virtual) issued by all clients that:
  - meets the specification of a single correct copy of objects
  - is consistent with the *real times* at which each operation occurred during the execution
- Main goal: any client will see (at any point of time) a copy of the object that is correct and consistent

Sequential Consistency

- The real-time requirement of linearizability is hard, if not impossible, to achieve in real systems
- A less strict criterion is sequential consistency: A replicated shared object service is sequentially consistent if for any execution (real), there is some interleaving of clients' operations (virtual) that:
  - meets the specification of a single correct copy of objects
  - is consistent with the program order in which each individual client executes those operations.
- This approach does not require absolute time or total order. Only that for each client the order in the sequence be consistent with that client's program order (~ FIFO).
- Linearilizability implies sequential consistency. Not vice-versa!
- Challenge with guaranteeing seq. cons.?
  - Ensuring that all replicas of an object are consistent.

Passive Replication
(Primary-Backup)

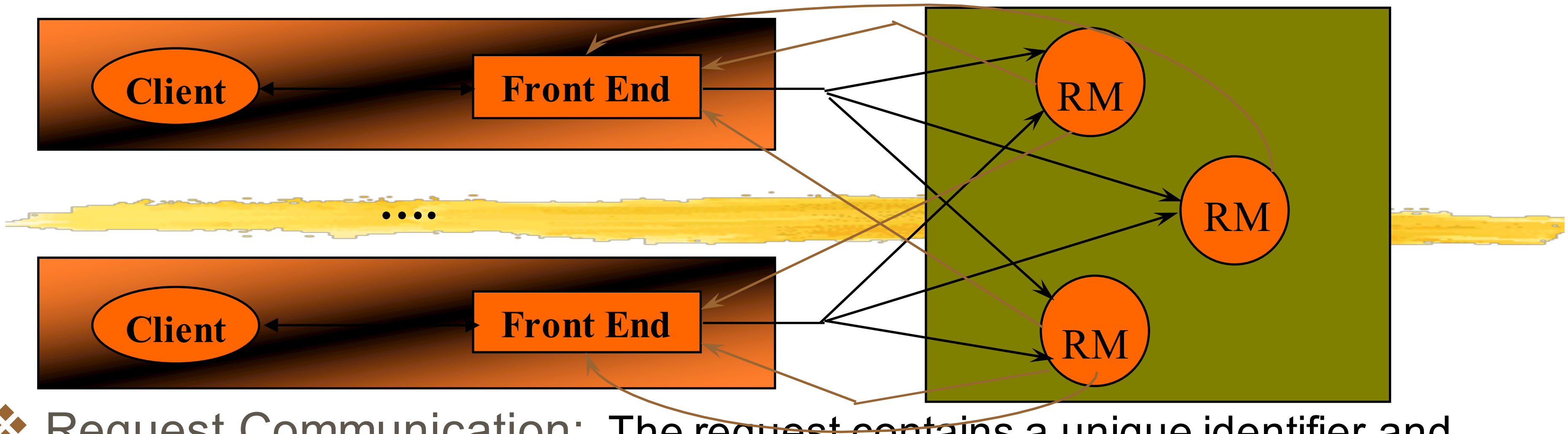


- **Request Communication:** the request is issued to the primary RM and carries a unique request id.
- **Coordination:** Primary takes requests atomically, in order, checks id (resends response if not new id.)
- **Execution:** Primary executes & stores the response
- **Agreement:** If update, primary sends updated state/result, req-id and response to all backup RMs (1-phase commit enough).
- **Response:** primary sends result to the front end

Fault Tolerance in Passive Replication

- The system implements linearizability, since the primary sequences operations in order.
- If the primary fails, a backup becomes primary by leader election, and the replica managers that survive agree on which operations had been performed at the point when the new primary takes over.
  - The above requirement can be met if the replica managers (primary and backups) are organized as a group and if the primary uses view-synchronous group communication to send updates to backups.
- Thus the system remains linearizable in spite of crashes

Active Replication



- Request Communication: The request contains a unique identifier and is multicast to all by a reliable totally-ordered multicast.
- Coordination: Group communication ensures that requests are delivered to each RM in the same order (but may be at different physical times!).
- Execution: Each replica executes the request. (Correct replicas return same result since they are running the same program, i.e., they are *replicated protocols* or *replicated state machines*)
- Agreement: No agreement phase is needed, because of multicast delivery semantics of requests
- Response: Each replica sends response directly to FE

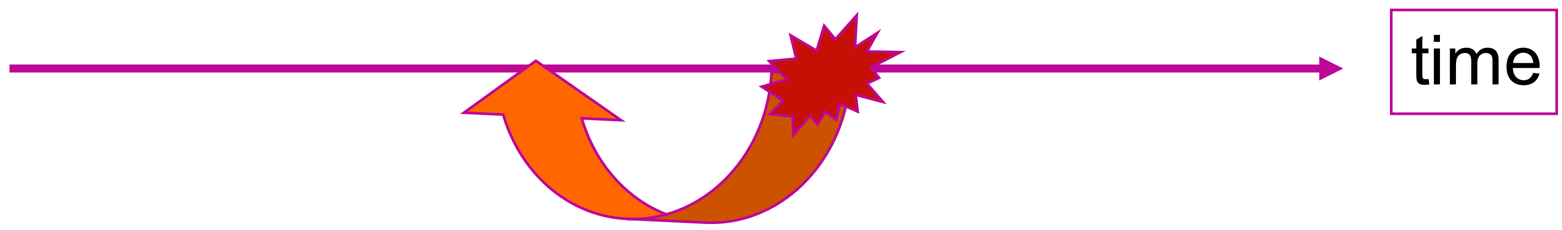
FT via Active Replication

- RMs work as replicated state machines, playing equivalent roles. That is, each responds to a given series of requests in the same way. One way of achieving this is by running the same program code at all RMs (but only one way – why?).
- If any RM crashes, state is maintained by other correct RMs.
- This system implements sequential consistency
  - The total order ensures that all correct replica managers process the same set of requests in the same order.
  - Each front end's requests are served in FIFO order (because the front end awaits a response before making the next request).
- So, requests are FIFO-total ordered.
- Caveat (Out of band): If clients are multi-threaded and communicate with one another while waiting for responses from the service, we may need to incorporate causal-total ordering.

Backward vs. forward error recovery

Backward error recovery

When safety property is violated, the computation **rolls back** and resumes from a previous correct state.



rollback

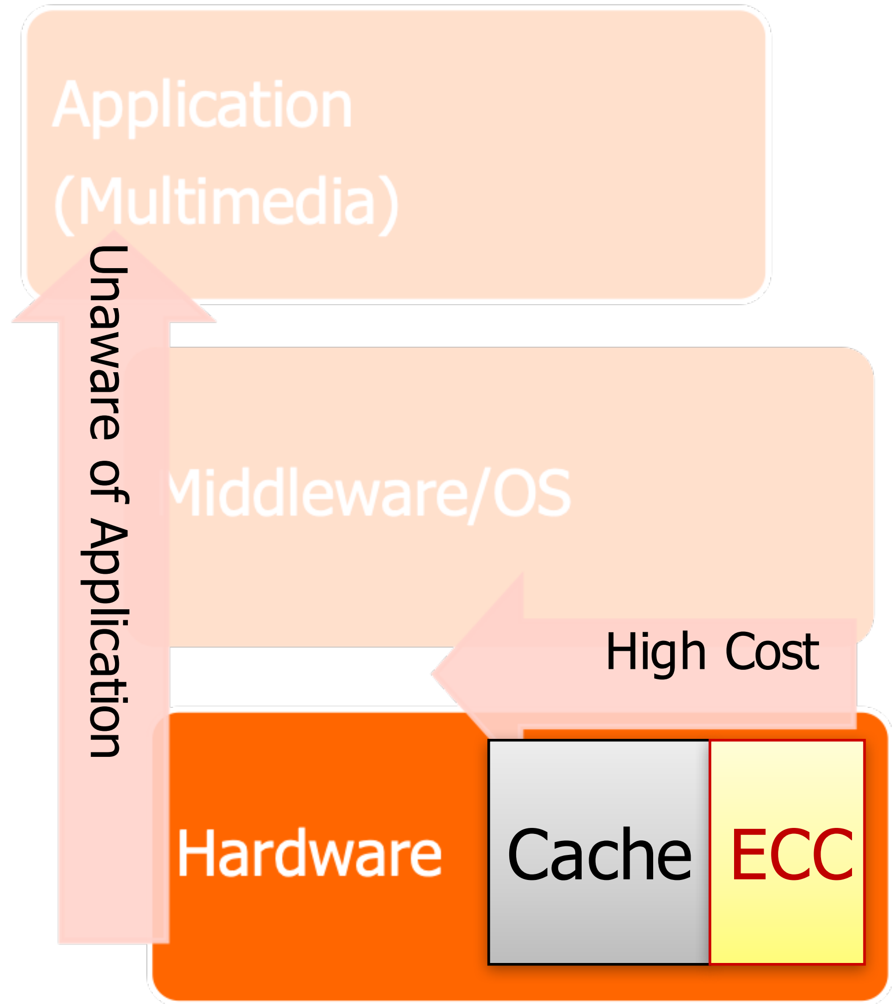
Forward error recovery

Computation does not care about getting the history right, but moves on, as long as eventually the safety property is restored.

True for self-stabilizing systems.

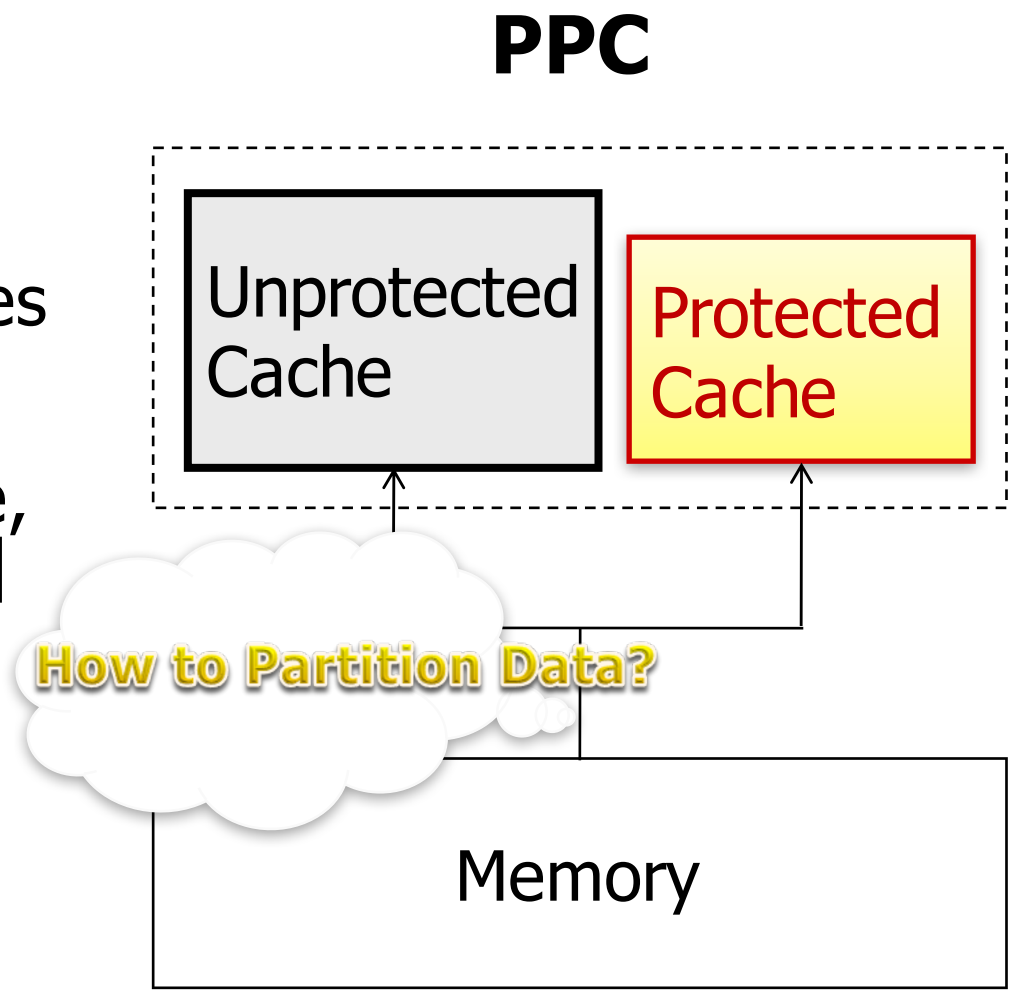
Conventional Protection for Caches

- Cache is the most hit by soft errors
- Conventional Protected Caches
  - Unaware of fault tolerance at applications
  - Implement a redundancy technique such as ECC to protect all data for every access
    - **Overkill for multimedia applications**
  - ECC (e.g., a Hamming Code) incurs high performance penalty by up to 95%, power overhead by up to 22%, and area cost by up to 25%

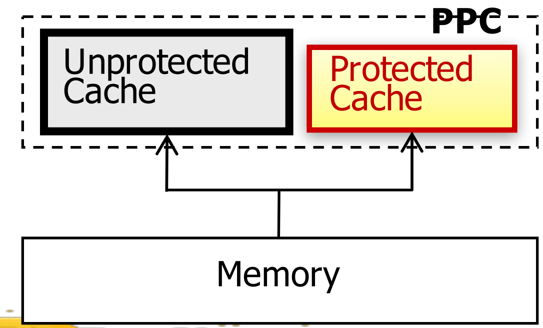


PPC (Partially Protected Caches)

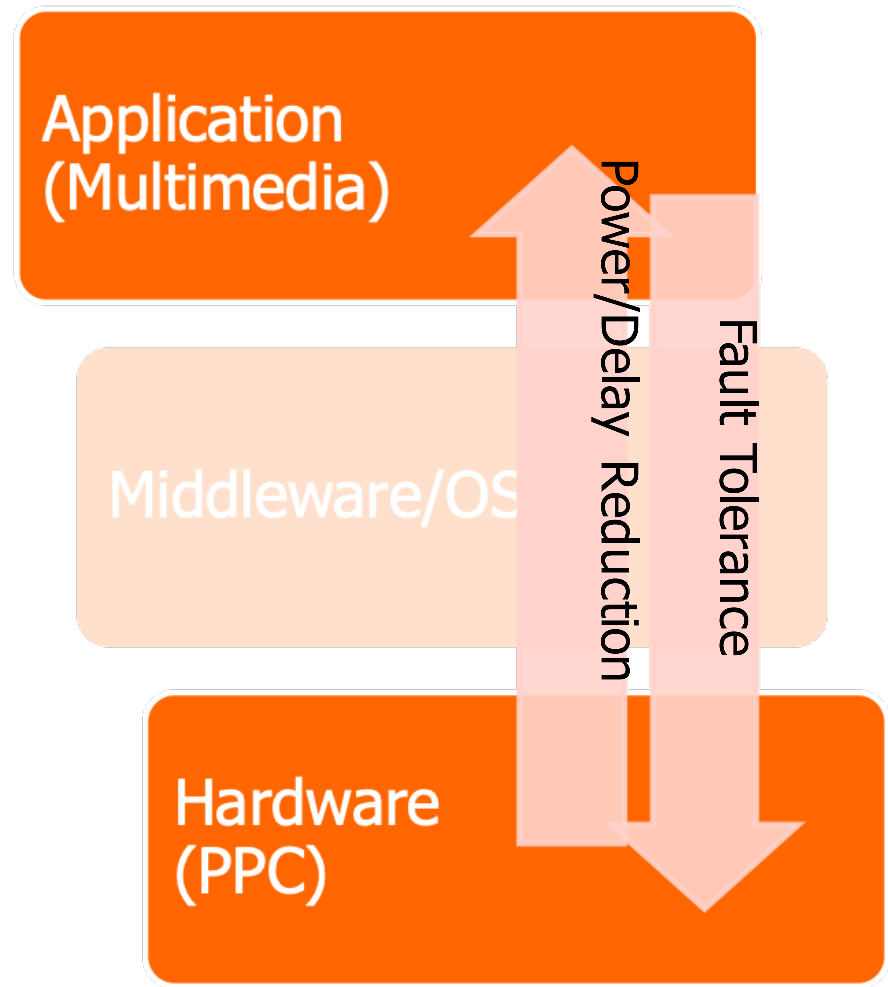
- Observation
  - Not all data are equally failure critical
    - Multimedia data vs. control variables
- Propose PPC architectures to provide an unequal protection for mobile multimedia systems [Lee, CASES06][Lee, TVLSI08]
  - **Unprotected cache** and **Protected cache** at the same level of memory hierarchy
  - Protected cache is typically smaller to keep power and delay the same as or less than those of Unprotected cache



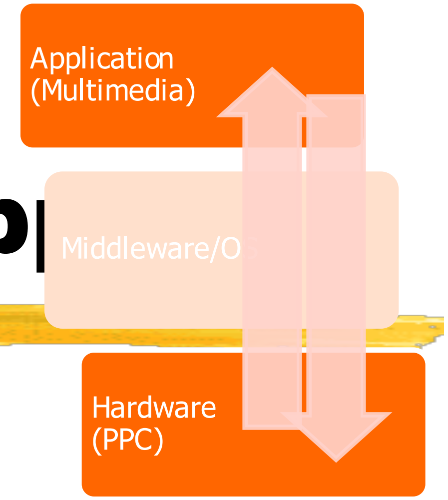
PPC for Multimedia Applications



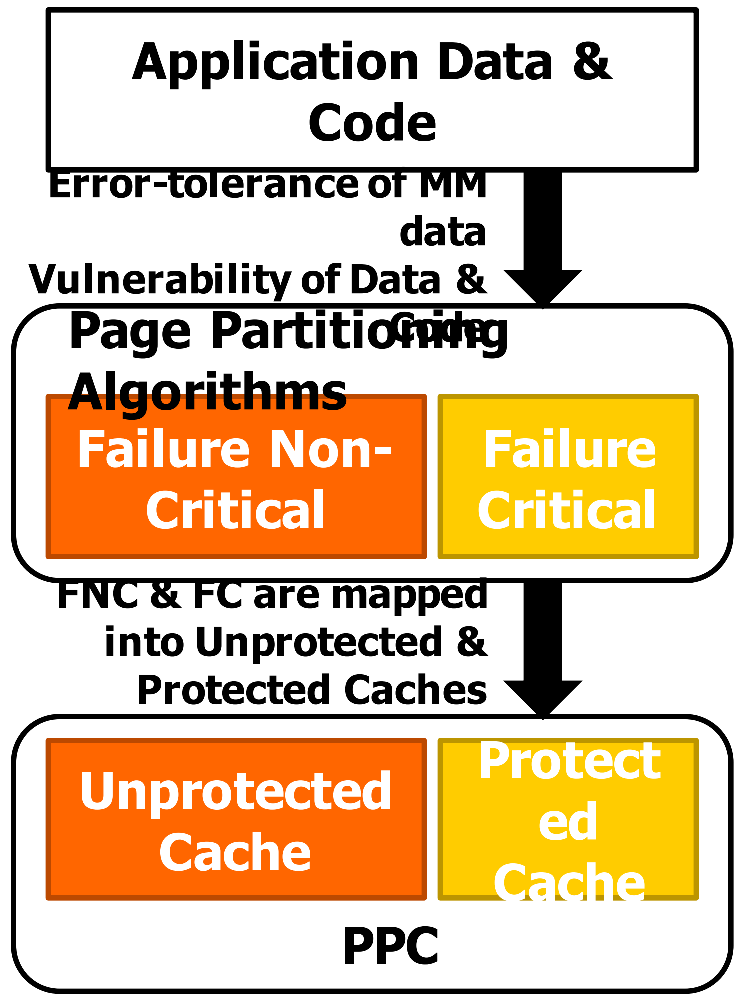
- Propose a selective data protection [Lee, CASES06]
- Unequal protection at hardware layer exploiting **error-tolerance** of multimedia data at application layer
- Simple data partitioning for multimedia applications
  - **Multimedia data** is failure non-critical
  - **All other data** is failure critical



PPC for general purpose ap

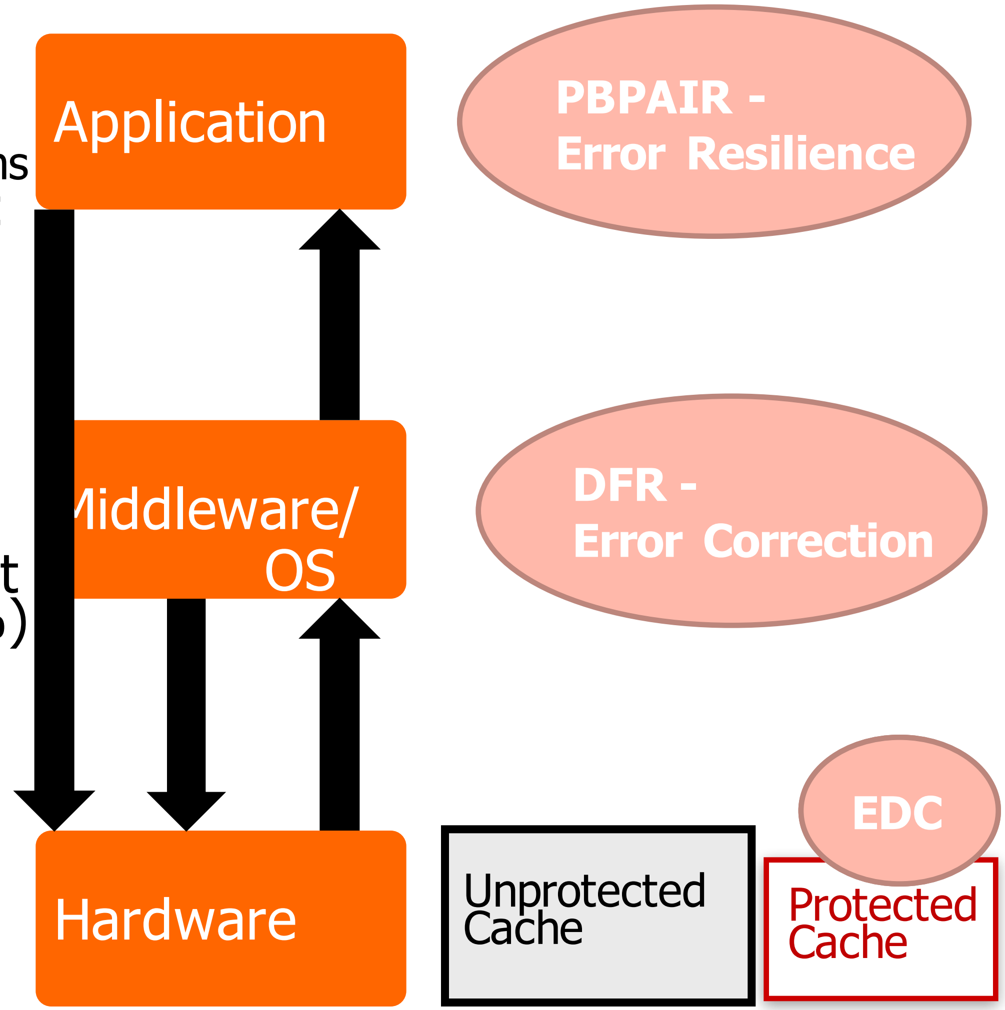


- All data are **not equally failure itical**
- Propose a **PPC** architecture to provide **unequal protection**
  - Support an unequal protection at hardware layer by exploiting error-tolerance and vulnerability at application
- DPExplore [Lee, PPCDIPES08]
  - Explore partitioning space by exploiting vulnerability of each data page
- Vulnerable time
  - It is vulnerable for the time when eventually it is read by CPU or written back to Memory
- **Pages causing high vulnerable time** are failure critical

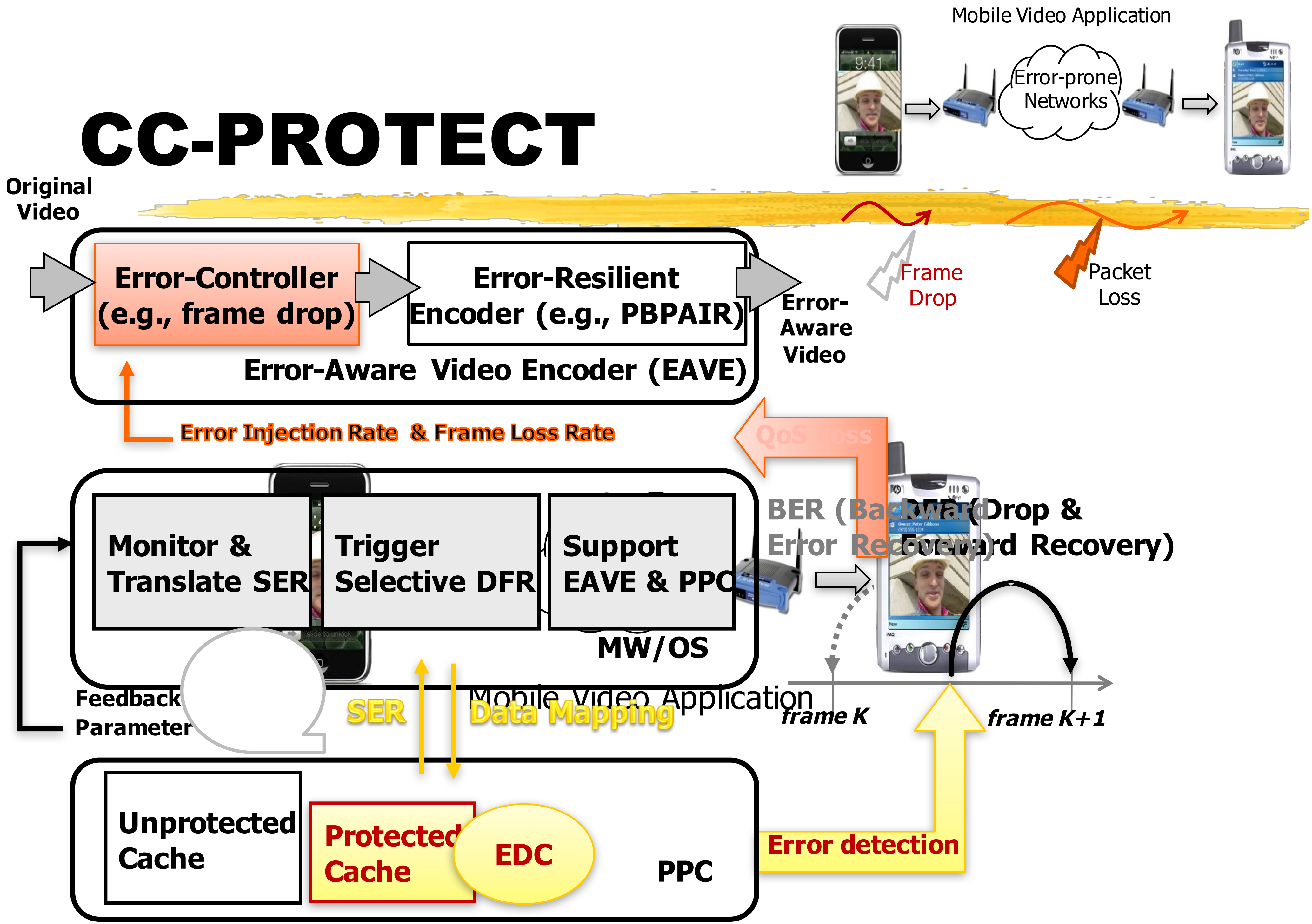


CC-PROTECT

- Approach which **cooperates existing schemes across layers** to mitigate the impact of soft errors on the failure rate and video quality in mobile video encoding systems
  - PPC (Partially Protected Caches) with EDC (Error Detection Codes) at hardware layer
  - DFR (Drop and Forward Recovery) at middleware
  - PBPAIR (Probability-Based Power Aware Intra Refresh) at application layer
- Demonstrate the effectiveness of low-cost (about 50%) reliability (1,000x) at the minimal cost of QoS (less than 1%)



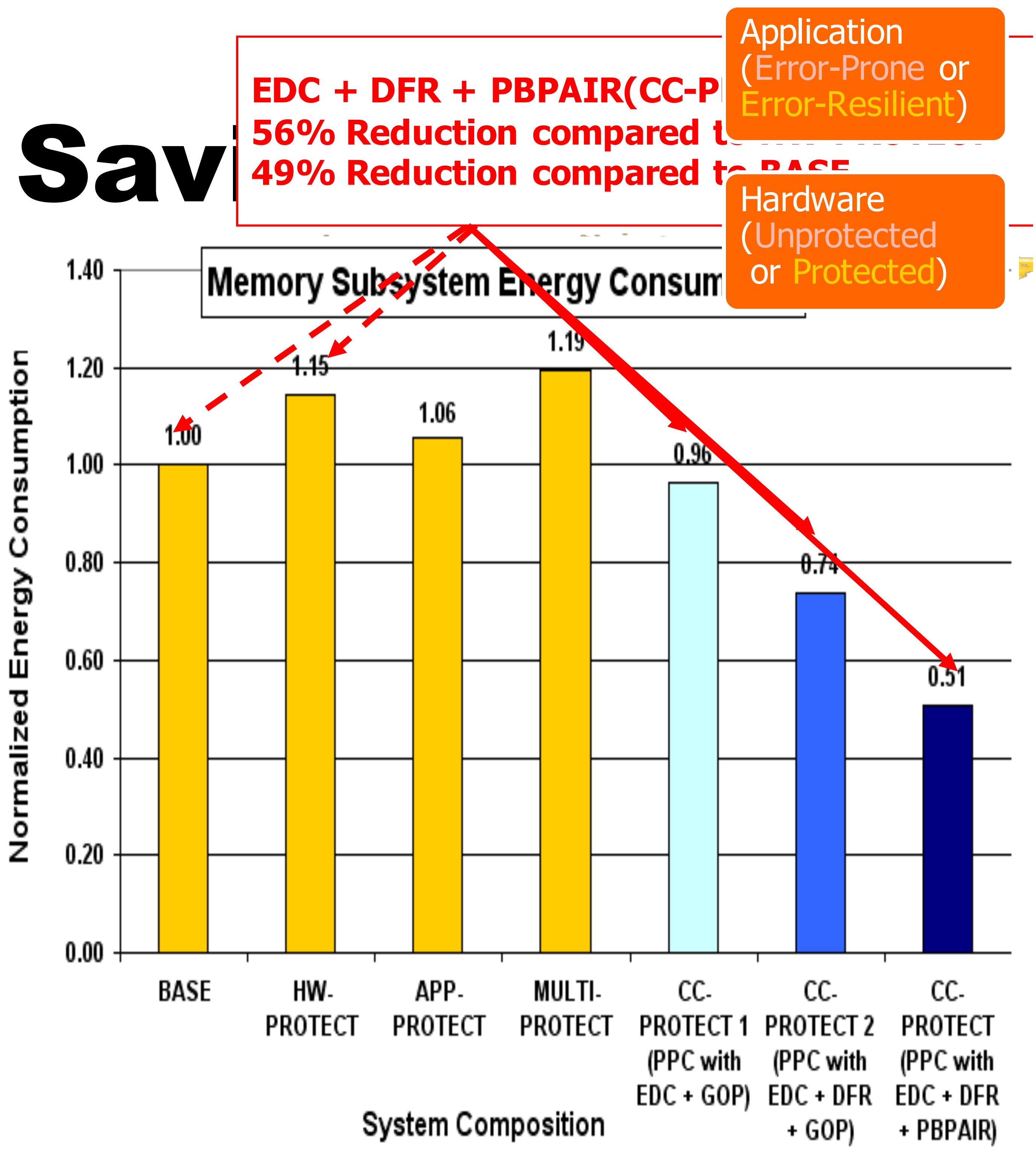
CC-PROTECT



Mobile Video Application
Error-prone Networks
Original Video
Error-Controller (e.g., frame drop)
Error-Resilient Encoder (e.g., PBPAIR)
Error-Aware Video Encoder (EAVE)
Error-Aware Video
Frame Drop
Packet Loss
Error Injection Rate & Frame Loss Rate
QoS Loss
Monitor & Translate SER
Trigger Selective DFR
Support EAVE & PPC
MW/OS
BER (Backward Error Recovery)
DFR (Drop & Forward Recovery)
frame K
frame K+1
Feedback Parameter
SER
Data Mapping
Mobile Video Application
Unprotected Cache
Protected Cache
EDC
PPC
Error detection

Energy Savi

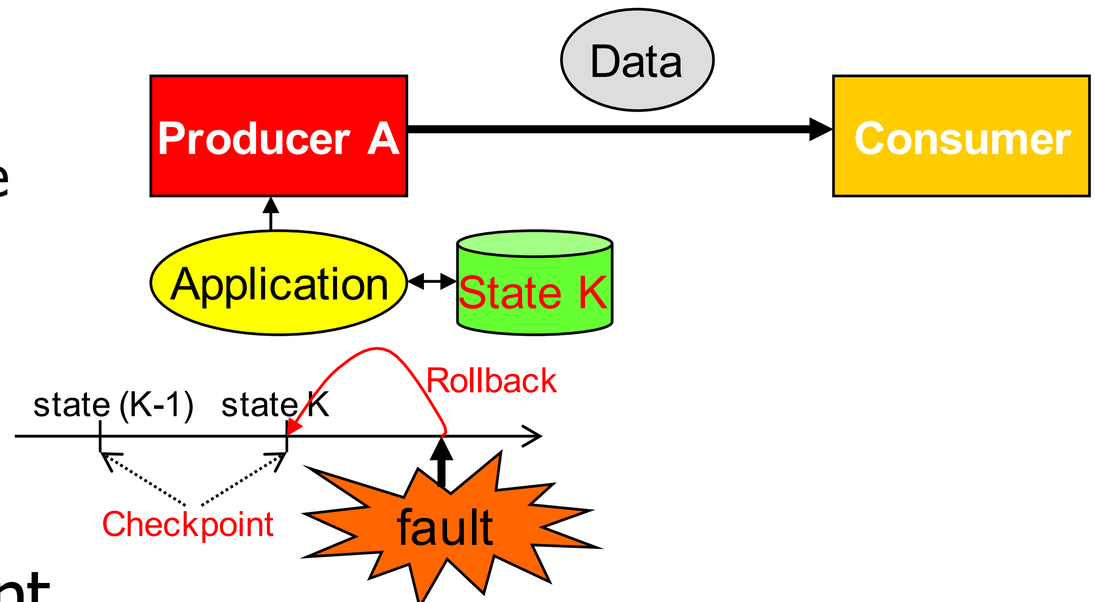
- BASE = Error-prone video encoding + unprotected cache
- HW-PROTECT = Error-prone video encoding + PPC with ECC
- APP-PROTECT = Error-resilient video encoding + unprotected cache
- MULTI-PROTECT = Error-resilient video encoding + PPC with ECC
- CC-PROTECT1 = Error-prone video encoding + PPC with EDC
- CC-PROTECT2 = Error-prone video encoding + PPC with EDC + DFR
- CC-PROTECT = error-resilient video encoding + PPC with EDC + DFR



4) Checkpoints & Rollbacks

- Checkpoints and Rollbacks
  - Checkpoint
    - A copy of an application's state
    - Save it in storage immune to the failures
  - Rollback
    - Restart the execution from a previously saved checkpoint

➔ Recover from transient and permanent hardware and software failures



Message Logging

- Tolerate crash failures
- Each process periodically records its local state and log messages received after
  - Once a crashed process recovers, its state must be consistent with the states of other processes
  - Orphan processes
    - surviving processes whose states are inconsistent with the recovered state of a crashed process
  - Message Logging protocols guarantee that upon recovery no processes are orphan processes

Message logging protocols

- Pessimistic Message Logging
  - avoid creation of orphans during execution
  - no process p sends a message m until it knows that all messages delivered before sending m are logged; quick recovery
  - Can block a process for each message it receives - slows down throughput
  - allows processes to communicate only from recoverable states; synchronously log to stable storage any information that may be needed for recovery before allowing process to communicate

Message Logging

- Optimistic Message Logging
  - take appropriate actions during recovery to eliminate all orphans
  - Better performance during failure-free runs
  - allows processes to communicate from non-recoverable states; failures may cause these states to be permanently unrecoverable, forcing rollback of any process that depends on such states

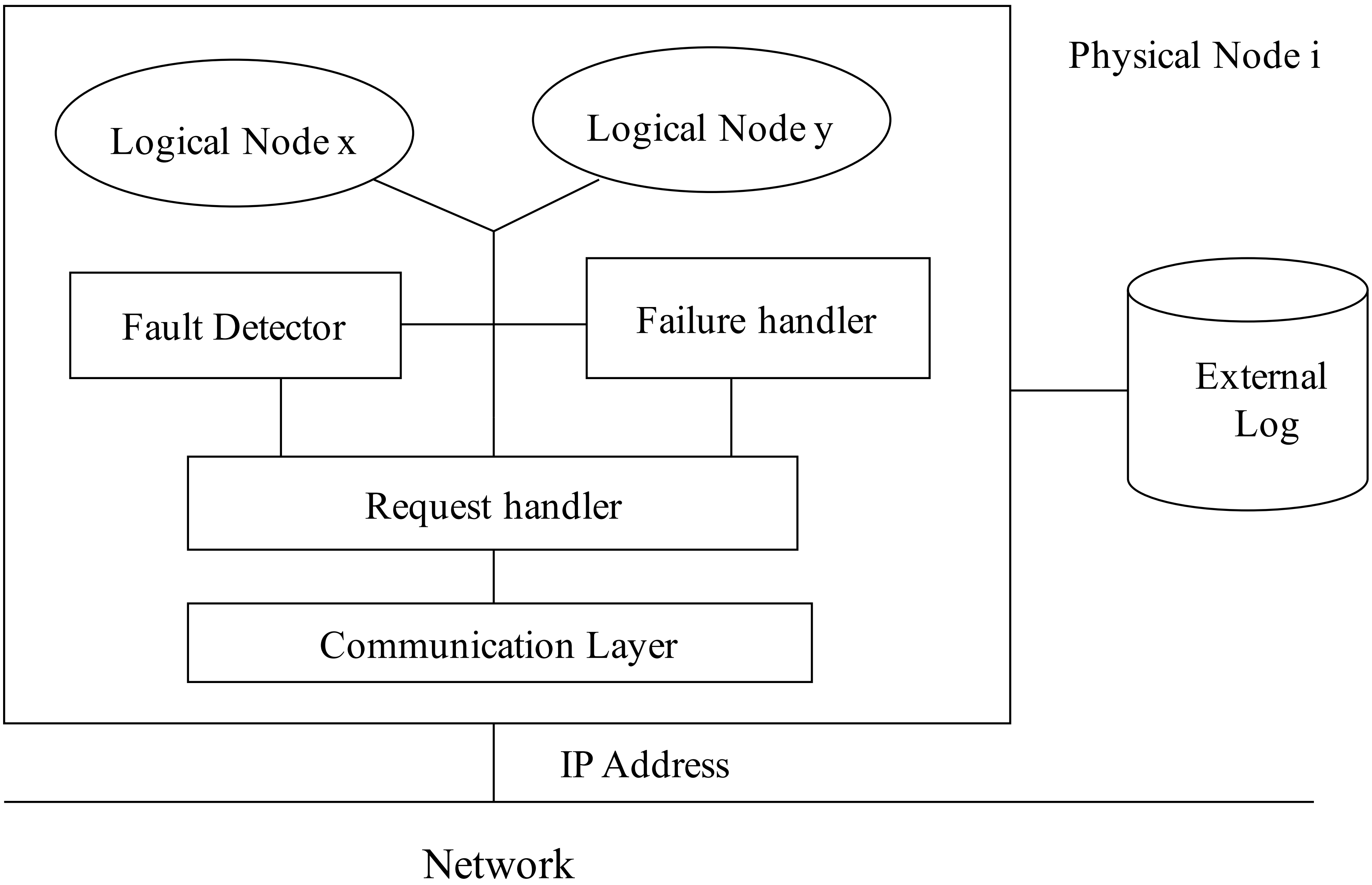
Causal Message Logging

- Causal Message Logging
  - no orphans when failures happen and do not block processes when failures do not occur.
  - Weaken condition imposed by pessimistic protocols
  - Allow possibility that the state from which a process communicates is unrecoverable because of a failure, but only if it does not affect consistency.
  - Append to all communication information needed to recover state from which communication originates - this is replicated in memory of processes that causally depend on the originating state.

KAN – A Reliable Distributed Object System (UCSB)

- Goal
  - Language support for parallelism and distribution
  - Transparent location/migration/replication
  - Optimized method invocation
  - Fault-tolerance
  - Composition and proof reuse
- Log-based forward recovery scheme
  - Log of recovery information for a node is maintained externally on other nodes.
  - The failed nodes are recovered to their pre-failure states, and the correct nodes keep their states at the time of the failures.
- Only consider node crash failures.
  - Processor stops taking steps and failures are eventually detected.

Basic Architecture of the Fault Tolerance Scheme



Egida (UT Austin)

- An object-oriented, extensible *toolkit* for low-overhead fault-tolerance
- Provides a library of objects that can be used to *compose log-based rollback recovery protocols*.
  - Specification language to express arbitrary rollback-recovery protocols
  - Checkpointing
    - independent, coordinated, induced by specific patterns of communication
  - Message Logging
    - Pessimistic, optimistic, causal

AQuA

- Adaptive Quality of Service Availability
- Developed in UIUC and BBN.
- Goal:
  - Allow distributed applications to request and obtain a desired level of availability.
- Fault tolerance
  - replication
  - reliable messaging

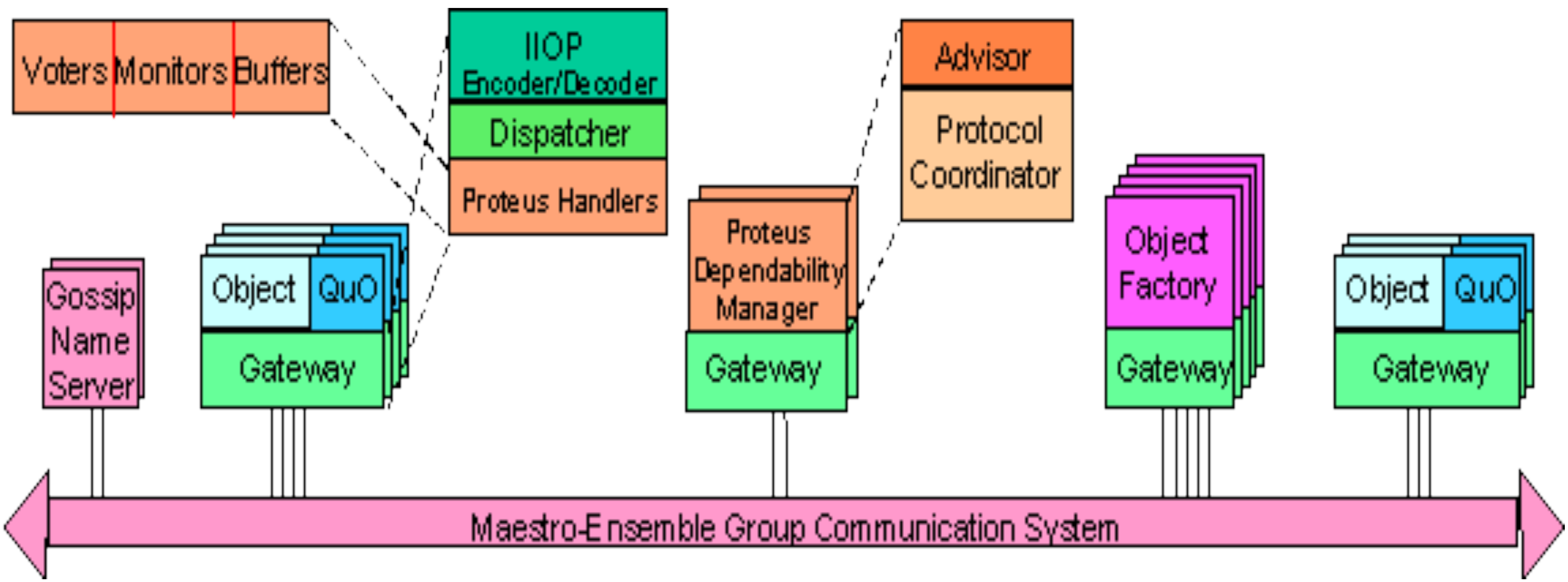
Features of AQuA

- Uses the *QuO* runtime to process and make availability requests.
- *Proteus* dependability manager to configure the system in response to faults and availability requests.
- *Ensemble* to provide group communication services.
- Provide *CORBA* interface to application objects using the *AQuA* gateway.

Group structure

- For reliable mcast and pt-to-pt. Comm
  - Replication groups
  - Connection groups
  - Proteus Communication Service Group for replicated proteus manager
    - replicas and objects that communicate with the manager
    - e.g. notification of view change, new QuO request
    - ensure that all replica managers receive same info
  - Point-to-point groups
    - proteus manager to object factory

AQuA Architecture



Fault Model, detection and Handling

- Object Fault Model:
  - Object crash failure - occurs when object stops sending out messages; internal state is lost
    - crash failure of an object is due to the crash of at lease one element composing the object
  - Value faults - message arrives in time with wrong content (caused by application or QuO runtime)
    - Detected by *voter*
  - Time faults
    - Detected by *monitor*
  - Leaders report fault to *Proteus*; *Proteus* will kill objects with fault if necessary, and generate new objects

5) Recovery Blocks

- Recovery Blocks
  - Multiple alternates to perform the same functionality
    - One Primary module and Secondary modules
    - Different approaches

1) Select a module with output satisfying acceptance test
2) Recovery Blocks and Rollbacks
    - Restart the execution from a previously saved checkpoint with secondary module

➔Tolerate software failures