Fault-Tolerant Computing

About This Presentation

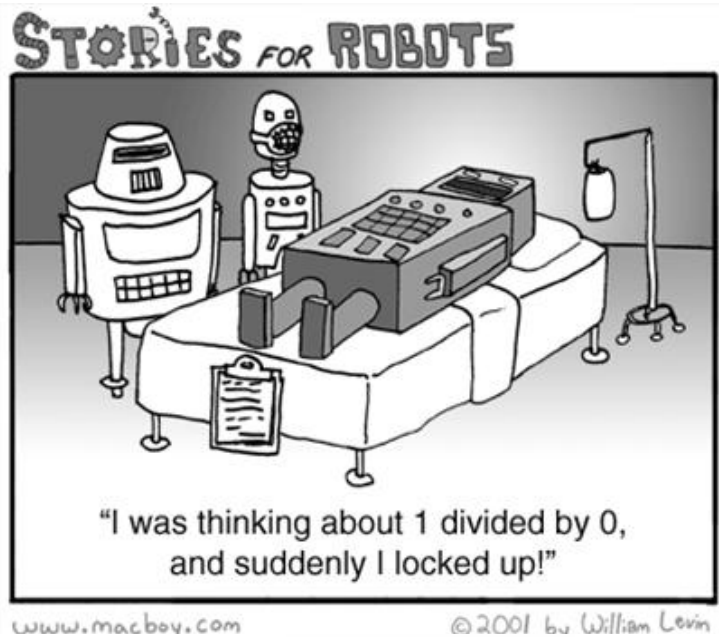
This presentation has been prepared for the graduate course ECE 257A (Fault-Tolerant Computing) by Behrooz Parhami, Professor of Electrical and Computer Engineering at University of California, Santa Barbara. The material contained herein can be used freely in classroom teaching or any other educational setting. Unauthorized uses are prohibited. © Behrooz Parhami

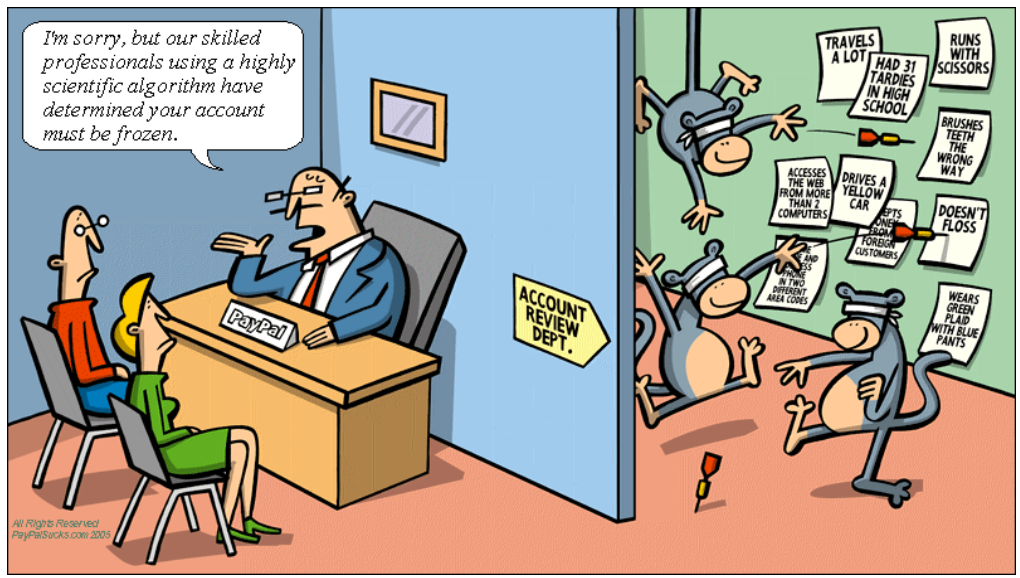
Edition	Released	Revised	Revised
First	Dec. 2006		

Algorithm Design Methods

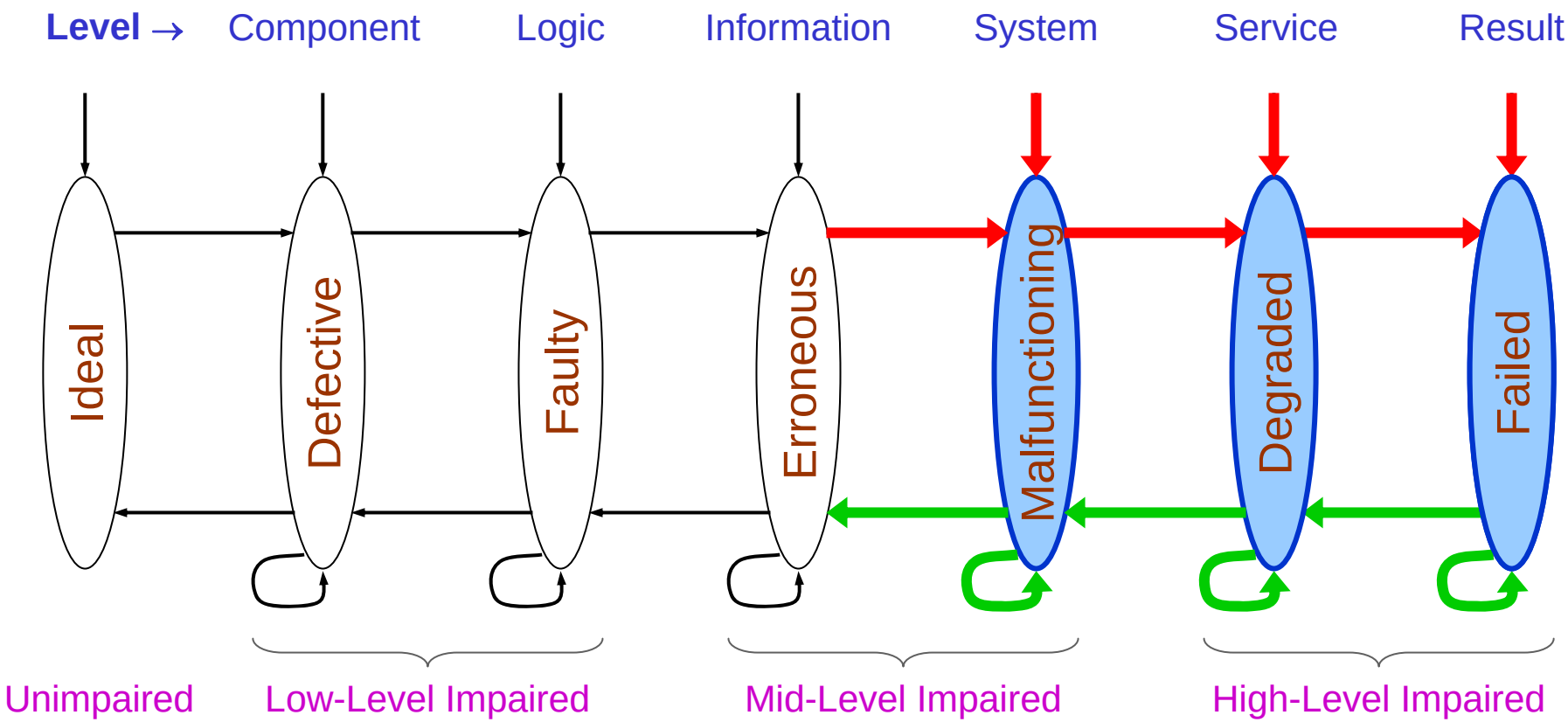
"MEMO: It has come to my attention that every time we solve one problem, we create two more. From now on, all problem solving is forbidden."

"We couldn't afford faster computers, so we just made them *sound* faster."





Multilevel Model of Dependable Computing



COTS-Based Dependable Computing

Many of the hardware and software redundancy methods discussed thus far assume that we are building the entire system (or a significant part of it) from scratch

Some companies with fault-tolerant systems and related services:

Nth Generation Computing: High-availability and enterprise storage systems
Resilience: Emphasis on security
Stratus Technologies: "The Availability Company"
Sun Microsystems: Fault-tolerant SPARC (ft-SPARC™)
Tandem Computers: An early ft leader, part of HP/Compaq since 1997

Question: What can be done to ensure the dependability of computations using commercial off-the-shelf (COTS) components?

A number of algorithm and data-structure design methods are available

Some History: The SIFT Experience

SIFT (software-implemented fault tolerance), developed at Stanford in early 1970s using mostly COTS components, was one of two competing "concept systems" for fly-by-wire aircraft control

The other one, FTMP (fault-tolerant multiprocessor), developed at MIT, used a hardware-intensive approach

System failure rate goal: 10^{-9}/hr over a 10-hour flight

SIFT allocated tasks for execution on multiple, loosely synchronized COTS processor-memory pairs (skew of up to 50 μs was acceptable); only the bus system was custom designed

Some fundamental results on, and methods for, clock synchronization emerged from this project

To prevent errors from propagating, processors obtained multiple copies of data from different memories over different buses (local voting)

Limitations of the COTS-Based Approach

Some modern microprocessors have dependability features built in:
Parity and other codes in memory, TLB, microcode store
Retry at various levels, from bus transmissions to full instructions
Machine check facilities and registers to hold the check results

According to Avizienis, however:
These are often not documented enough to allow users to build on them
Protection is nonsystematic and uneven
Recovery options are limited to shutdown and restart
Description of error handling is scattered among a lot of other detail
There is no top-down view of the features and their interrelationships

Manufacturers can incorporate both more advanced and new features, and at times have experimented with a number of mechanisms, but the low volume of the application base has hindered commercial viability

Robust Data Structures

Stored and transmitted data can be protected against unwanted changes through encoding, but coding does not protect the structure of the data

Consider, e.g., an ordered list of numbers
Individual numbers can be protected by encoding
The set of values can be protected by a checksum
The ordering, however, remains unprotected

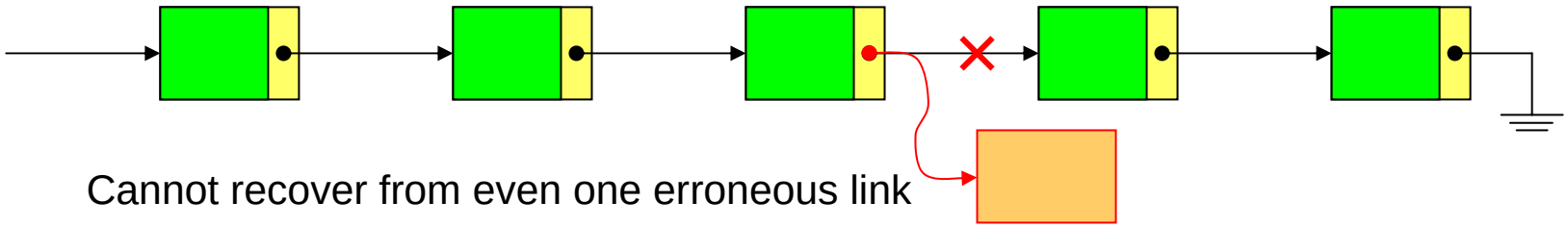
Idea – Use a checksum that weighs each value differently: $(\Sigma j x_j)$ mod A

Idea – Add a “difference with next item” field to each list entry

Can we devise some general methods for protecting commonly used data structures?

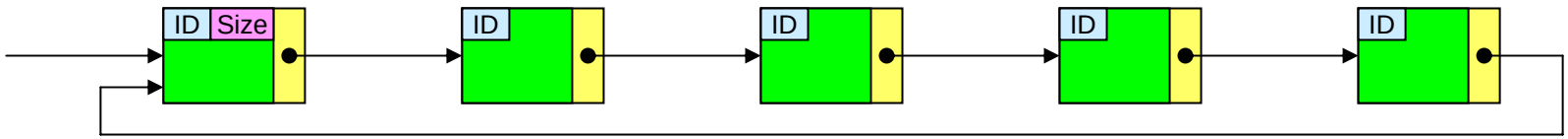
Recoverable Linear Linked Lists

Simple linked list: 0-detectable, 0-correctable

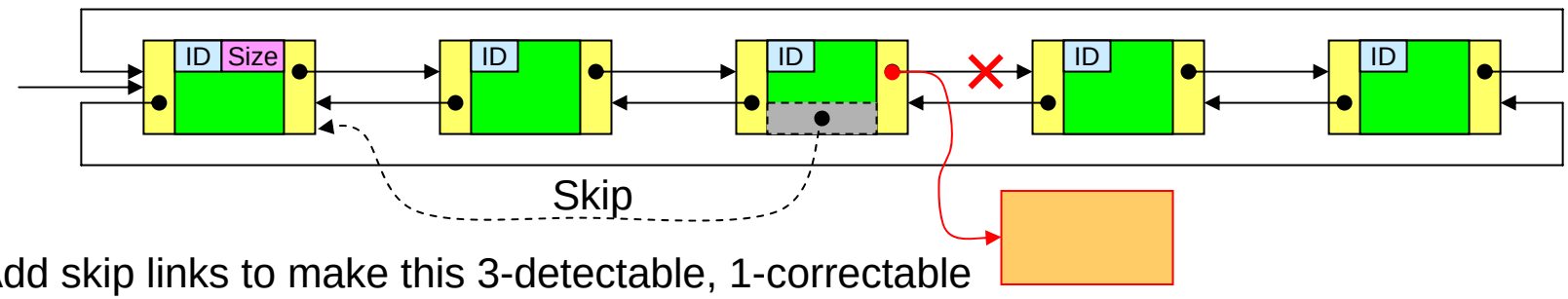


Cannot recover from even one erroneous link

Circular list, with node count and unique ID: 1-detectable, 0-correctable



Doubly linked list, with node count and ID: 2-detectable, 1-correctable



Add skip links to make this 3-detectable, 1-correctable

Other Robust Data Structures

Trees, FIFOs, stacks (LIFOs), heaps, queues

In general, a linked data structure is 2-detectable and 1-correctable iff the link network is 2-connected

Robust data structures provide fairly good protection with little design effort or run-time overhead

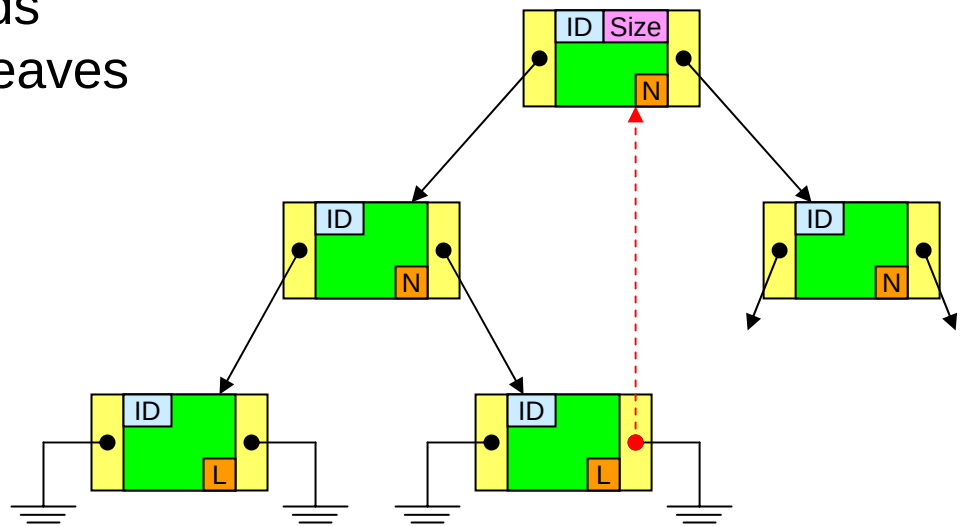
- Audits can be performed during idle time
- Reuse possibility makes the method even more effective

Robustness features to protect the structure can be combined with coding methods (such as checksums) to protect the content

Recoverable Binary Trees

Add “parent links” and/or “threads” (threads are links that connect leaves to higher-level nodes)

Threads can be added with little overhead by taking advantage of unused leaf links (one bit in every node can be used to identify leaves, thus freeing their link fields for other uses)



Adding redundancy to data structures has three types of cost:

- Storage requirements for the additional information
- Slightly more difficult updating procedures
- Time overhead for periodic checking of structural integrity

Algorithm-Based Error Tolerance

Error coding applied to data structures, rather than at the level of atomic data elements

Example: mod-8 checksums used for matrices

If $Z = X \times Y$ then $Z_f = X_c \times Y_r$

In M_f, any single error is correctable and any 3 errors are detectable

Four errors may go undetected

Matrix M

$$M = \begin{pmatrix} 2 & 1 & 6 \\ 5 & 3 & 4 \\ 3 & 2 & 7 \end{pmatrix}$$

Row checksum matrix

$$M_r = \begin{pmatrix} 2 & 1 & 6 & 1 \\ 5 & 3 & 4 & 4 \\ 3 & 2 & 7 & 4 \end{pmatrix}$$

Column checksum matrix

$$M_c = \begin{pmatrix} 2 & 1 & 6 \\ 5 & 3 & 4 \\ 3 & 2 & 7 \\ 2 & 6 & 1 \end{pmatrix}$$

Full checksum matrix

$$M_f = \begin{pmatrix} 2 & 1 & 6 & 1 \\ 5 & 3 & 4 & 4 \\ 3 & 2 & 7 & 4 \\ 2 & 6 & 1 & 1 \end{pmatrix}$$

Matrix Multiplication Using ABET

If $Z = X \times Y$ then
$Z_f = X_c \times Y_r$

$$X = \begin{pmatrix} 2 & 1 & 6 \\ 5 & 3 & 4 \\ 3 & 2 & 7 \end{pmatrix} \qquad Y = \begin{pmatrix} 1 & 5 & 3 \\ 2 & 4 & 6 \\ 7 & 1 & 5 \end{pmatrix}$$

46 + 20 + 42 = 108 = 4 mod 8

36 = 4 mod 8

$$\begin{pmatrix} 2 & 1 & 6 \\ 5 & 3 & 4 \\ 3 & 2 & 7 \\ 2 & 6 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 5 & 3 & 1 \\ 2 & 4 & 6 & 4 \\ 7 & 1 & 5 & 5 \end{pmatrix} = \begin{pmatrix} 46 & 20 & 42 & 36 \\ 39 & 41 & 53 & 37 \\ 56 & 30 & 56 & 46 \\ 21 & 35 & 47 & 31 \end{pmatrix}$$

Column checksum matrix for X

Row checksum matrix for Y

20 + 41 + 30 = 91 = 3 mod 8

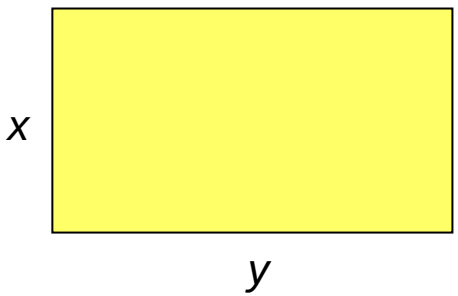
35 = 3 mod 8

Data Diversity

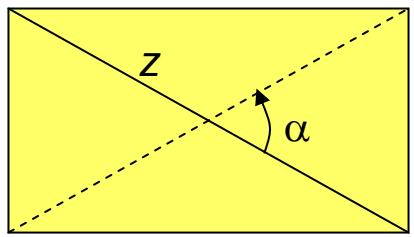
Alternate formulations of the same information (input re-expression)

Example: The shape of a rectangle can be specified:
By its two sides *x* and *y*
By the length *z* of its diameters and the angle α between them
By the radii *r* and *R* of its inscribed and circumscribed circles

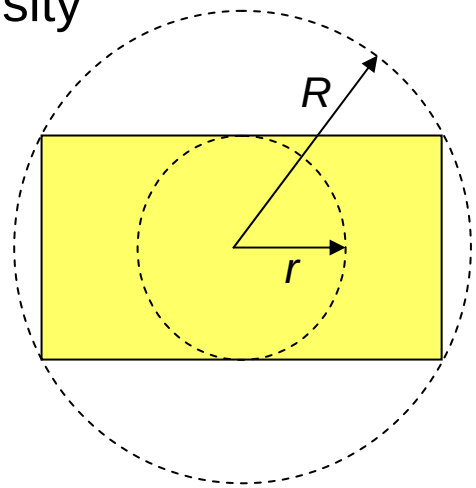
Area calculations with computation and data diversity



$A = xy$

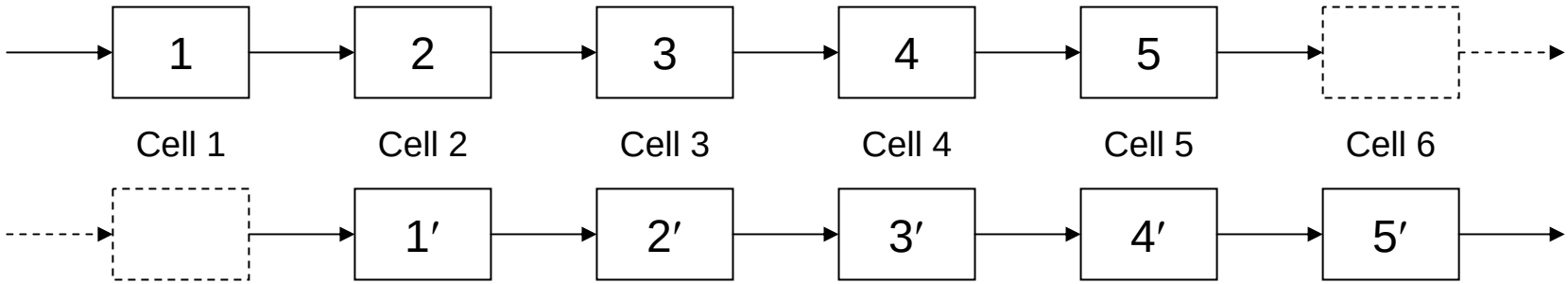


$A = \frac{1}{2} z^2 \sin \alpha$



$A = 4r(R^2 - r^2)^{1/2}$

Recomputing with Shift in Space



Linear array with an extra cell can redo the same pipelined computation with each step of the original computation shifted in space

Each cell *i* + 1 compares the result of step *i* that it received from the left in the first computation to the result of step *i* that it obtains in the second computation

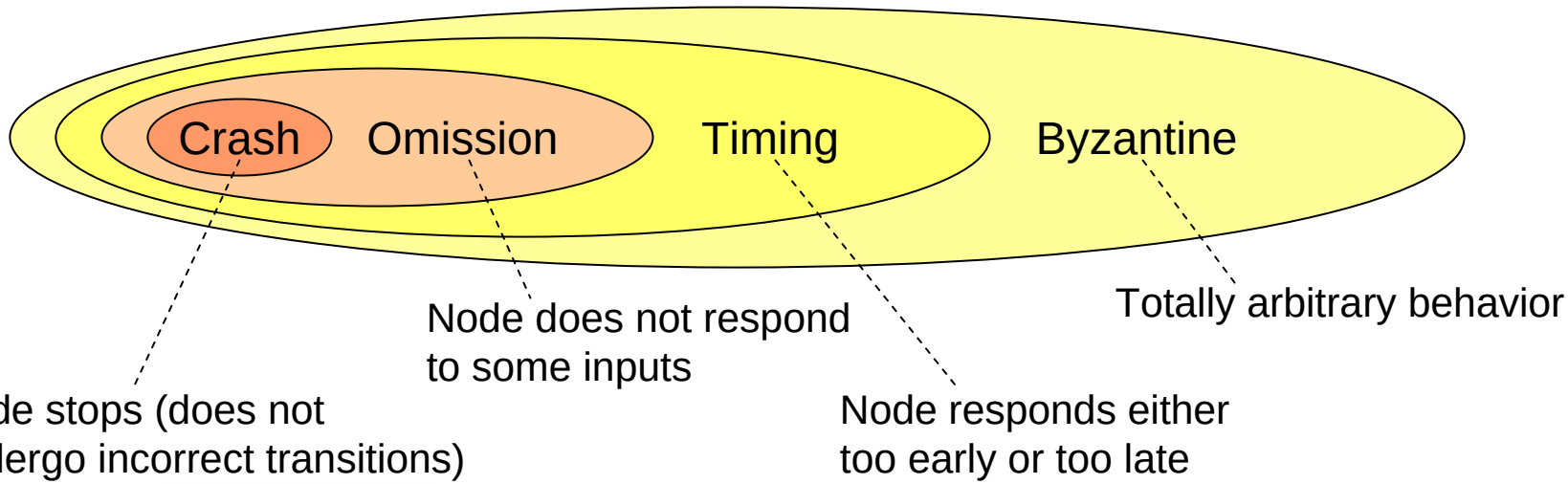
With two extra cells in the linear array, three computations can be pipelined and voting used to derive highly reliable results

COTS-Based Reliable Distributed Systems

Distributed systems, built from COTS nodes (processors plus memory) and interconnects, have redundancy and allow software-based fault tolerance implementation

Interconnect malfunctions are dealt with by synthesizing reliable communication primitives (

Node malfunctions are modeled differently, with the more general models requiring greater redundancy to deal with



Reliable Communication

Point-to-point message: encoding + acknowledgment + timeout

Reliable broadcast: message guaranteed to be received by all nodes

Forwarding along branches of a broadcast tree, with possible repetition (duplicate messages recognized from their sequence numbers)

Positive and negative acknowledgments piggybacked on subsequent broadcast messages (P broadcasts message m_1, Q receives it and tacks a positive ack for m_1 to message m_2 that it broadcasts, R did not receive m_1 but finds out about it from Q's ack and requests retransmit)

Atomic broadcast: reliable broadcast, plus the requirement that multiple broadcasts be received in the same order by all nodes (much more complicated to ensure common ordering of messages)

Causal broadcast: if m_2 is sent after m_1, any message triggered by m_2 must not cause actions before those of m_1 have been completed

Reliable Group Membership Service

A group of processes may be cooperating for solving a problem

The group's membership may expand and contract owing to changing processing requirements or because of malfunctions and repairs

Reliable multicast: message guaranteed to be received by all members within the group

ECE 254C: Advanced Computer Architecture – Distributed Systems (course devoted to distributed computing and its reliability issues)

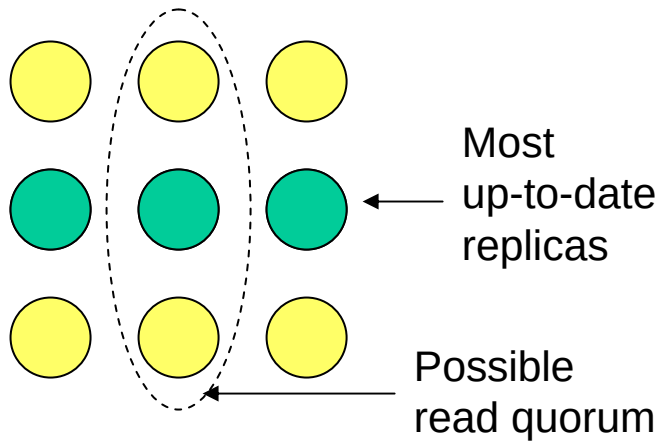
Data Replication

Resilient objects using the primary site approach

Active replicas: the state-machine approach
Request is sent to all replicas
All replicas are equivalent and any one of them can service the request
Ensure that all replicas are in same state (e.g., via atomic broadcast)

Read and write quorums

Example: 9 replicas, arranged in 2D grid
Rows constitute write quorums
Columns constitute read quorums
A read quorum contains the latest update



Maintaining replica consistency very difficult under Byzantine faults
Will discuss Byzantine agreement next time

Data Dispersion

Instead of replicating data objects completely, one can divide each one into *k* pieces, encode the pieces, and distribute the encoded pieces such that any *q* of the pieces suffice to reconstruct the data