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	Nov. 2006		

Degradation Allowance and Management

"We back up our data on sticky notes because sticky notes never crash."

"I always give 110% to my job. 40% on Monday, 30% on Tuesday, 20% on Wednesday, 15% on Thursday, and 5% on Friday."

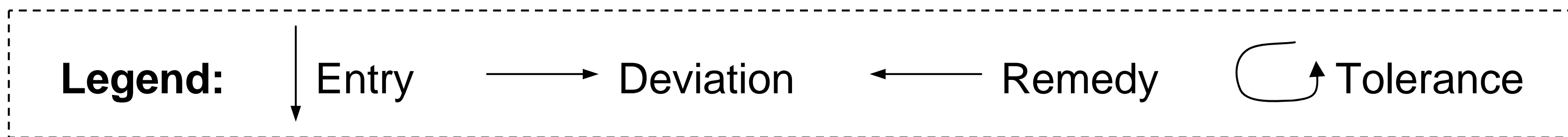
off the mark by Mark Parisi
www.offthemark.com

Multilevel Model of Dependable Computing

Level → Component Logic Information System Service Result

Ideal Defective Faulty Erroneous Malfunctioning Degraded Failed

Unimpaired Low-Level Impaired Mid-Level Impaired High-Level Impaired



Requirements for Graceful Degradation

Terminology: n. Graceful degradation
adj. Gracefully degrading/degradable = fail-soft

Strategies for failure prevention

1. Quick malfunction diagnosis
2. Effective isolation of malfunctioning elements
3. On-line repair (preferably via hot-pluggable modules)
4. Avoidance of catastrophic malfunctions

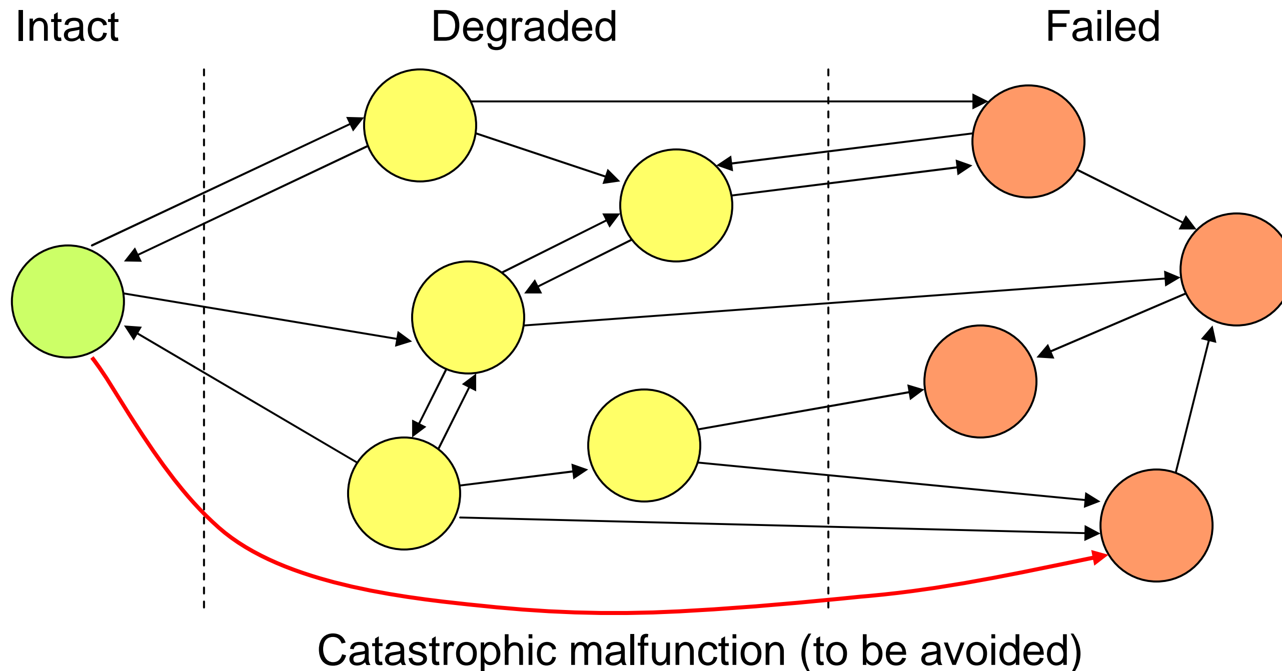
Degradation allowance

Provide capability for the system to function without the modules which have been diagnosed as malfunctioning

Degradation management

Control operation in degraded mode and try to return the system to the intact (or a less degraded) state ASAP

State-Space Model of a Fail-Soft System



Reducing the probability of catastrophic malfunctions

Reduce the probability of malfunctions going undetected

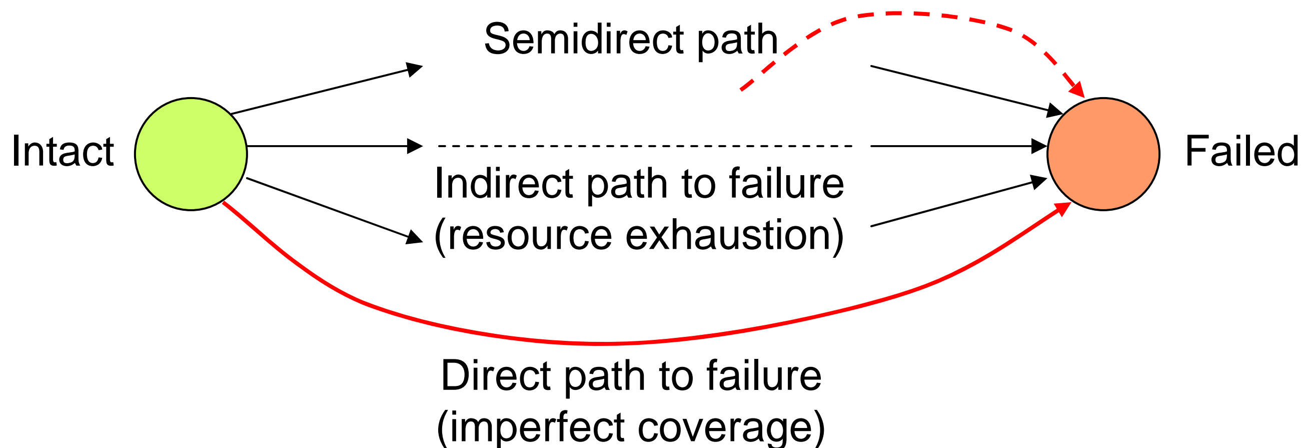
Increase the accuracy of malfunction diagnosis

Make repair rates much greater than malfunction rates (keep spares)

Provide sufficient "safety factor" in computational capacity

Importance of Coverage in Fail-Soft Systems

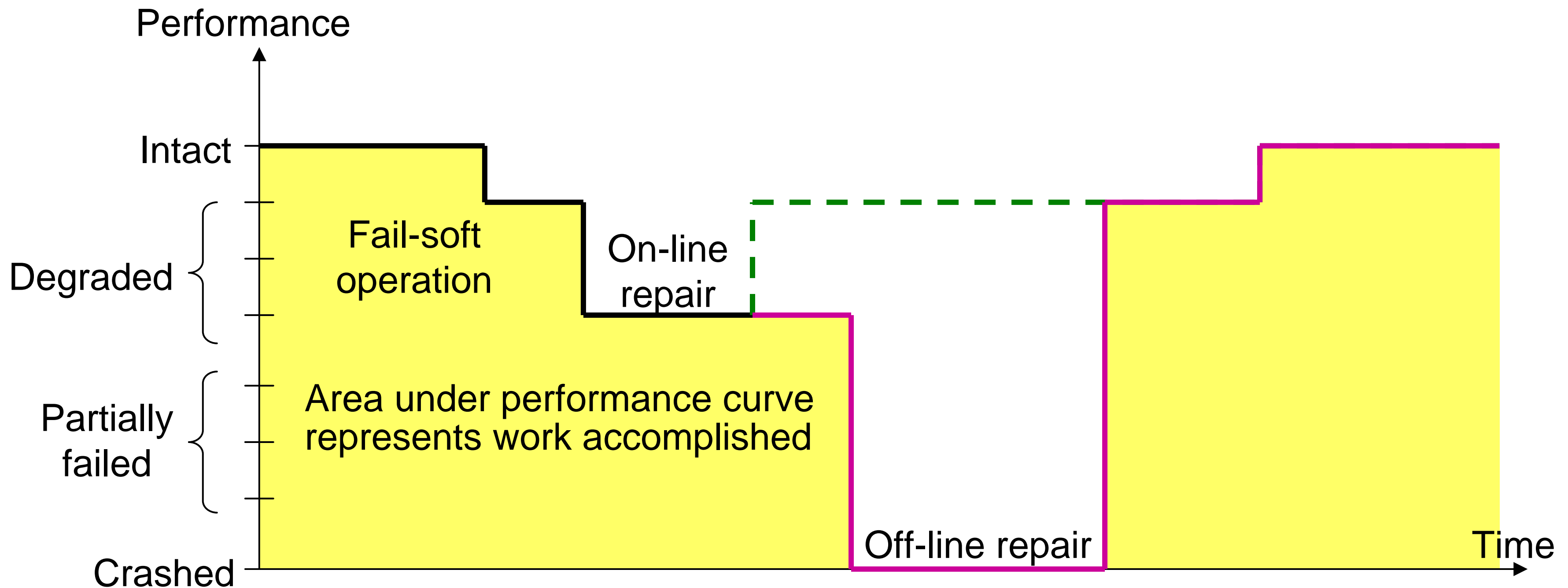
A fail-soft system can fail either indirectly, due to resource exhaustion, or directly because of imperfect coverage (analogous to leakage)



Providing more resources ("safety factor") lengthens the indirect path, thus slowing indirect failures but does nothing to block the direct path

Saturation effect: For a given coverage factor, addition of resources beyond a certain point would not be cost-effective with regard to the resulting reliability gain (same effect observed in standby sparing)

Performability of a Fail-Soft System



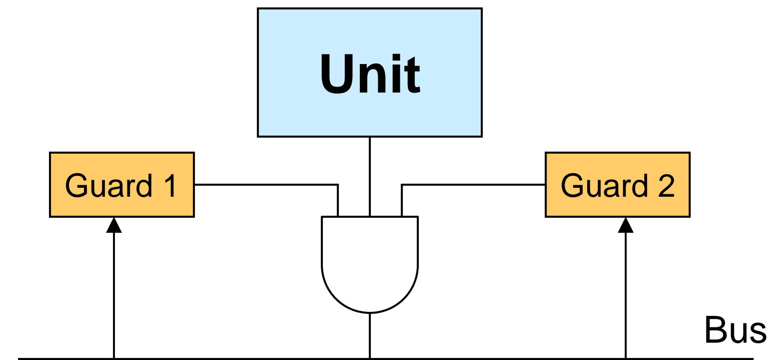
On-line repair: Done by removal/replacement of affected modules in a way that does not disrupt the operation of the remaining system parts

Off-line repair: Involves shutting down the entire system while affected modules are removed and replacements are plugged in

Graceful Degradation after a Malfunction

Diagnose the malfunction

Remove the malfunctioning unit
- Update system (OS) tables
- Physical isolation?
- Initiate repair, if applicable

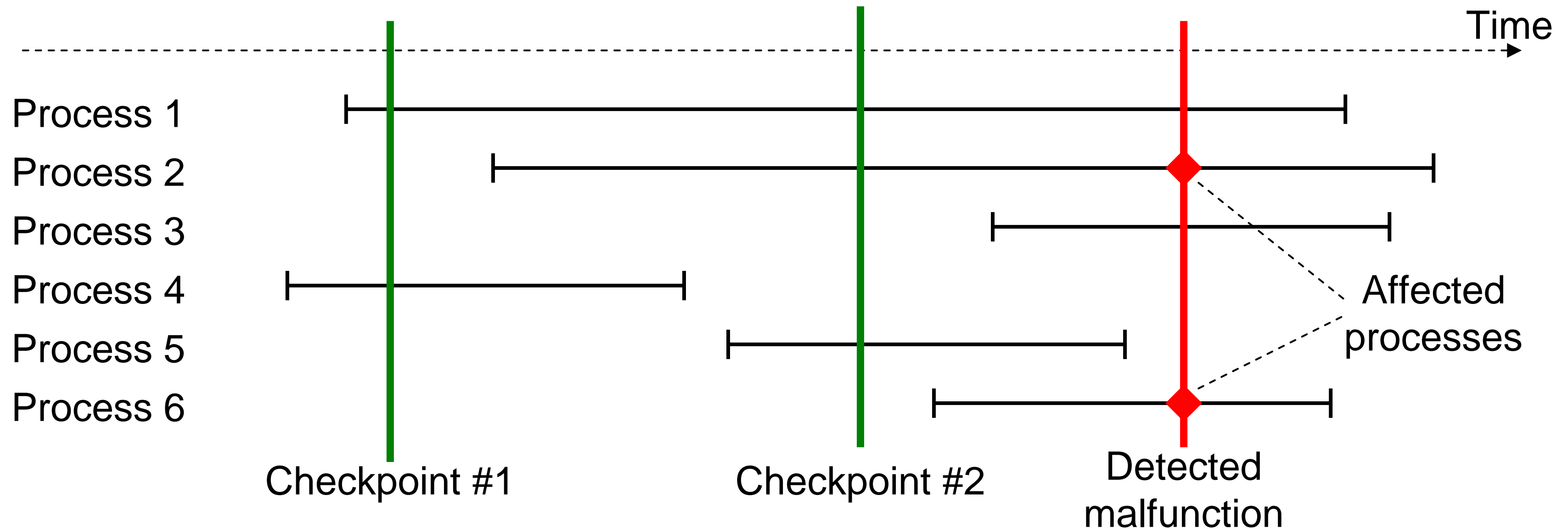


Create new working configuration
- Exclude processor, channel, controller, I/O device (e.g., sensor)
- Avoid bad tracks on disk, garbled files, noisy communication links
- Remove parts of the memory via virtual address mapping
- Bypass a cache or use only half of it (more restricted mapping?)

Perform process and data recovery
- Recover state information from removed unit
- Initialize any new resource brought on-line
- Reactivate processes (via rollback or restart)

Additional steps needed to return repaired units to operating status

Process and Data Recovery



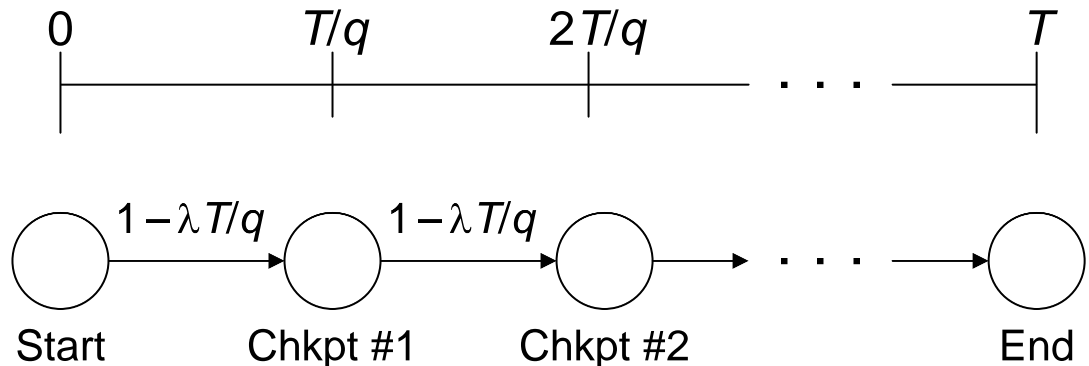
Roll back process 2 to the last checkpoint (#2)
Restart process 6

Rollback or restart creates no problem for tasks that do I/O at the end
Interactive processes must be handled with more care
e.g., bank ATM transaction to withdraw money or transfer funds (check balance, reduce balance, dispense cash or increase balance)

Optimal Checkpointing for Long Computations

T = Total computation time without checkpointing
q = Number of computation segments; there will be $q - 1$ checkpoints
T_{cp} = Time needed to capture a checkpoint snapshot
λ = Malfunction rate

Discrete Markov model: Expected length of stay in each state $1/(1 - \lambda T/q)$, where unit time is T/q



Computation time with checkpointing $T_{total} = T/(1 - \lambda T/q) + (q - 1)T_{cp}$
$= T + \lambda T^2/(q - \lambda T) + (q - 1)T_{cp}$

$dT_{total}/dq = -\lambda T^2/(q - \lambda T)^2 + T_{cp} = 0 \quad \Rightarrow \quad q^{opt} = T(\lambda + \sqrt{\lambda/T_{cp}})$

Example: $T = 200$ hr, $\lambda = 0.01$ / hr, $T_{cp} = 1/8$ hr

$q^{opt} = 200(0.01 + (0.01/0.25)^{1/2}) = 42$; $T_{total}^{opt} \approx 215$ hr

Warning: Model is accurate only when $T/q << 1/\lambda$

Elaboration on Optimal Computation Checkpoints

T = Total computation time without checkpointing (Example: 200 hr)
q = Number of computation segments; there will be $q - 1$ checkpoints
T_{cp} = Time needed to capture a checkpoint snapshot (Range: 1/8-1000 hr)
λ = Malfunction rate (Example: 0.01 / hr)

Computation time with checkpointing $T_{total} = T + \lambda T^2/(q - \lambda T) + (q - 1) T_{cp}$ (Rollback: $\lambda T^2/(q - \lambda T)$; Checkpointing: $(q - 1) T_{cp}$)

$dT_{total}/dq = -\lambda T^2/(q - \lambda T)^2 + T_{cp} = 0 \quad \Rightarrow \quad q^{opt} = T(\lambda + \sqrt{\lambda/T_{cp}})$
$d^2 T_{total}/dq^2 = 2\lambda T^2/(q - \lambda T)^3 > 0 \Rightarrow$ Bowl-like curve for T_{total}, with a minimum

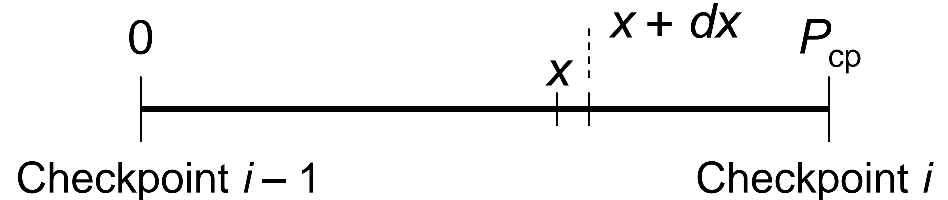
Example:

T_{total} / T_{cp} \ q	4	6	8	10	20	30	40	50
0	400	300	267	250	222	214	211	208
1/8	400	301	267	251	225	218	215	**214**
1/3	401	302	269	253	229	224	**224**	224
1	403	305	274	259	**241**	243	250	257
10	430	350	**337**	340	412	504	601	698
100	700	800	967	1150	2122	3114	4111	5108
1000	3400	5300	7267	9250	19222	29214	39211	49208

Optimal Checkpointing in Transaction Processing

P_{cp} = Checkpointing period
T_{cp} = Checkpointing time overhead (for capturing a database snapshot)
T_{rb} = Expected rollback time upon malfunction detection

Relative checkpointing overhead
$O = (T_{cp} + T_{rb}) / P_{cp}$



Assume that rollback time, given malfunction at time x, is $a + bx$
(b is typically small, because only updates need to be reprocessed)

$\rho(x)$: Expected rollback time due to malfunction in the time interval $[0, x]$

$\rho(x+dx) = \rho(x) + (a + bx)\lambda dx \Rightarrow d\rho(x)/dx = (a + bx)\lambda \Rightarrow \rho(x) = \lambda x(a + bx/2)$

$T_{rb} = \rho(P_{cp}) = \lambda P_{cp}(a + bP_{cp}/2)$

$O = (T_{cp}+T_{rb})/P_{cp} = T_{cp}/P_{cp} + \lambda(a + bP_{cp}/2)$ is minimized for: $P_{cp} = \sqrt{2T_{cp}/(\lambda b)}$

Examples for Optimal Database Checkpointing

$O = (T_{cp} + T_{rb})/P_{cp} = T_{cp}/P_{cp} + \lambda(a + bP_{cp}/2)$ is minimized for: $P_{cp} = \sqrt{2T_{cp}/(\lambda b)}$

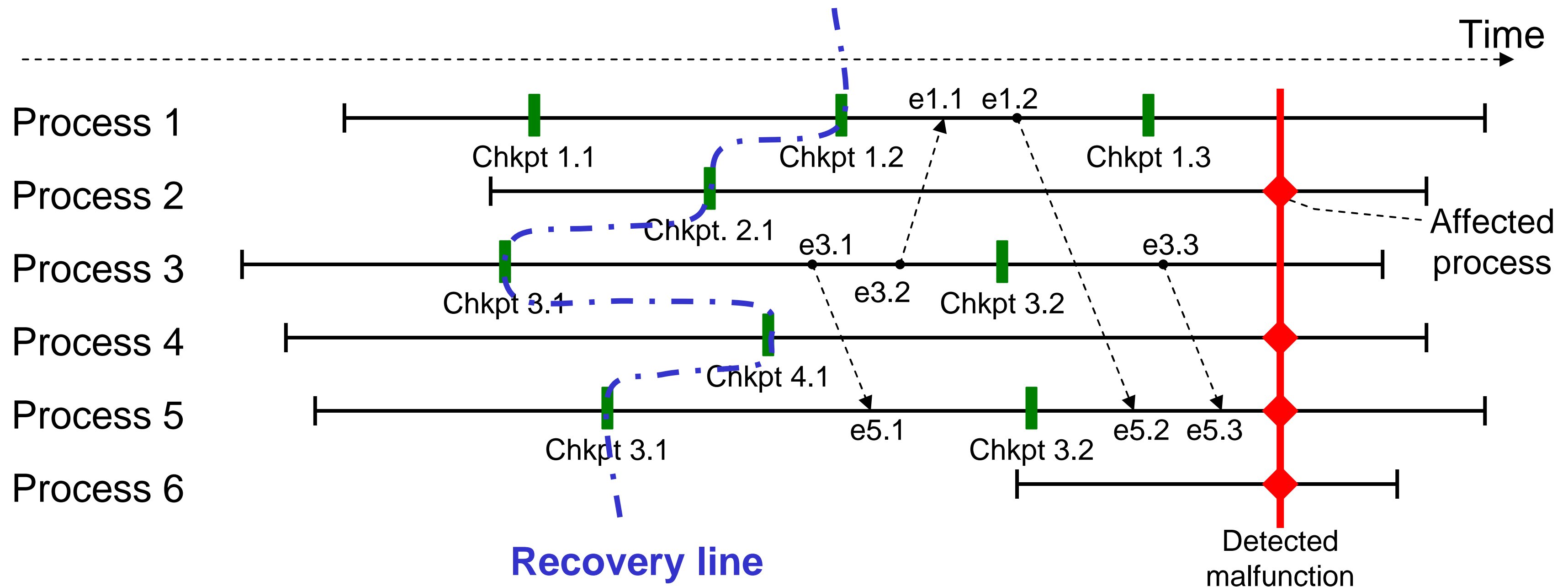
T_{cp} = Time needed to capture a checkpoint snapshot = 16 min
λ = Malfunction rate = 0.0005 / min (MTTF = 2000 min ≈ 33.3 hr)
$b = 0.1$

$P_{cp}^{opt} = \sqrt{2T_{cp}/(\lambda b)}$ = 800 min ≈ 13.3 hr

Suppose that by using faster memory for saving the checkpoint snapshots (e.g., disk, rather than tape) we reduce T_{cp} to 1 min

$P_{cp}^{opt} = \sqrt{2T_{cp}/(\lambda b)}$ = 200 min ≈ 3.3 hr

Asynchronous Distributed Checkpointing



For noninteracting processes, asynchronous checkpoints not a problem

When one process is rolled back, other processes may have to be rolled back also, and this has the potential of creating a domino effect

Identifying a consistent set of checkpoints (*recovery line*) is nontrivial

Checkpointing for Data

Consider data objects stored on a primary site and *k* backup sites (with appropriate design, such a scheme will be *k*-malfunction-tolerant)

Each access request is sent to the primary site
Read request honored immediately by the primary site
One way to deal with a write request:
  Update requests sent to backup sites
  Request is honored after all messages ack'ed

Alternative: Primary site does frequent back-ups

If one or more backup sites malfunction, service is not disrupted
If the primary site malfunctions, a new primary site is "elected"
  (distributed election algorithms exist that can tolerate malfunctions)

Analysis by Huang and Jalote:
  Normal state (primary OK, data available)
  Recovery state (primary site is changing)
  Checkpoint state (primary doing back-up)
  Idle (no site is OK)

Time in state:
↑fn of *k*
↓fn of *k*

k	Availability
0	0.922
1	0.987
2	0.996
4	0.997
8	0.997

Maintaining System Consistency

Atomic action: Either the entire action is completed or none of it is done

One key tool is the ability to ensure atomicity despite malfunctions

Similar to a computer guaranteeing sequential execution of instructions, even though it may perform some steps in parallel or out of order

Where atomicity is useful:
Upon a write operation, ensure that all data replicas are updated
Electronic funds transfer (reduce one balance, increase the other one)

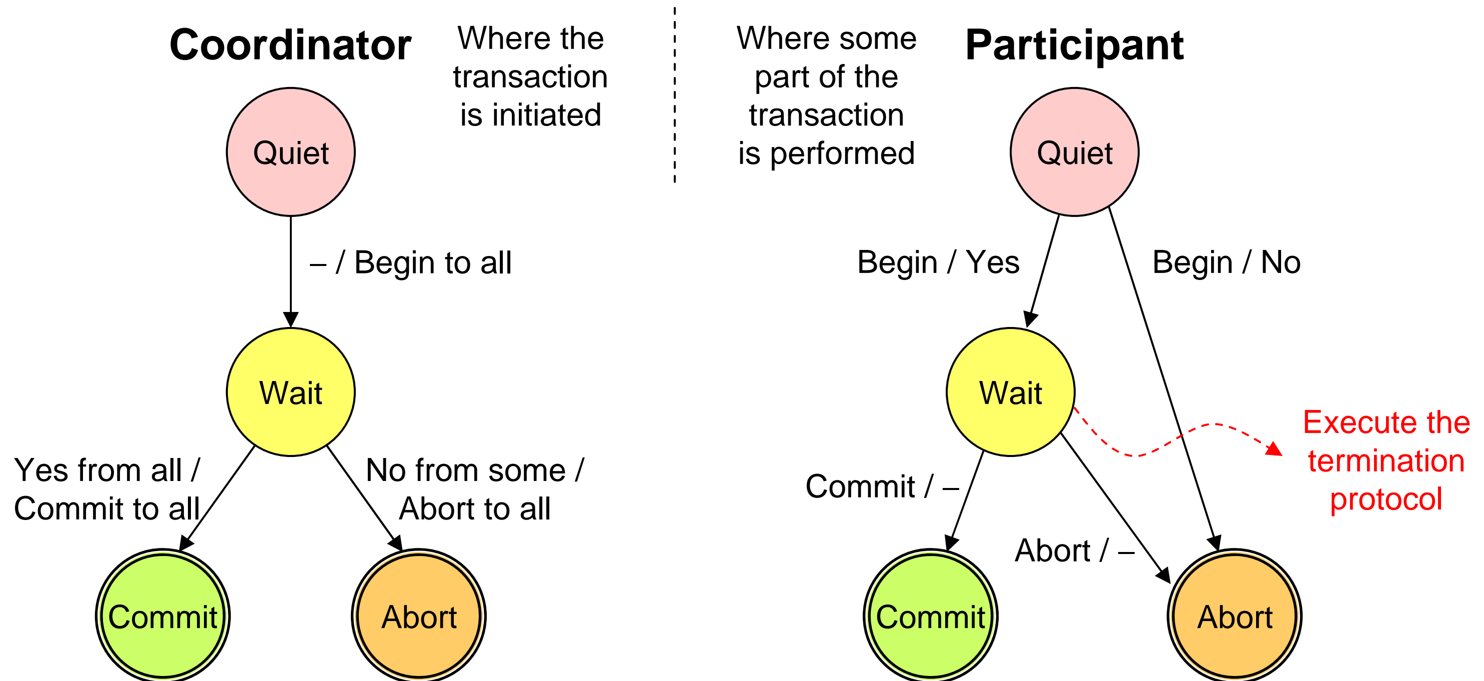
In centralized systems atomicity can be ensured via locking mechanisms

Acquire (read or write) lock for desired data object and operation
Perform operation
Release lock

A key challenge of locks is to avoid deadlock (circular waiting for locks)

Two-Phase Commit Protocol

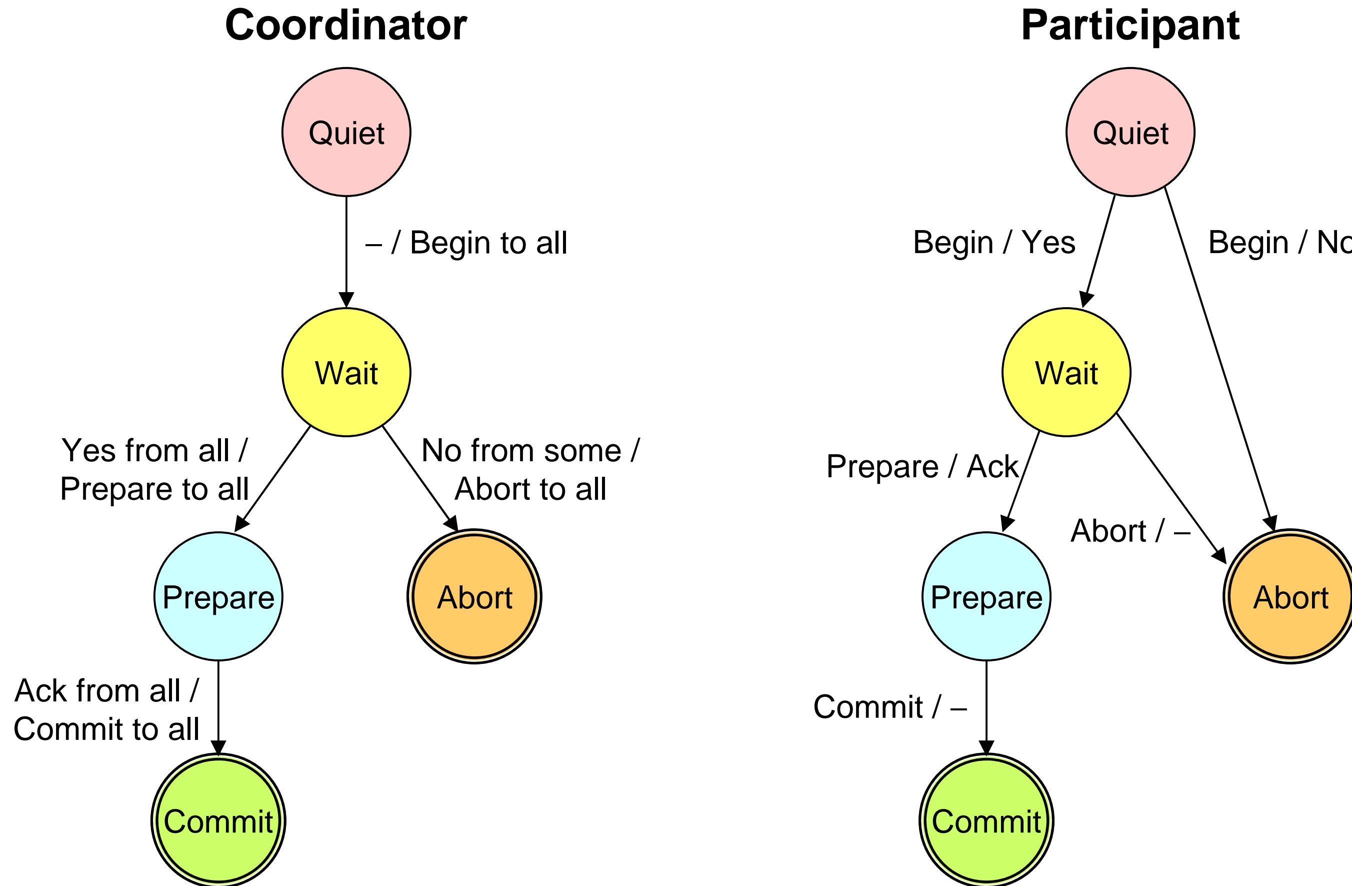
Ensuring atomicity of actions in a distributed environment



To avoid participants being stranded in the wait state (e.g., when the coordinator malfunctions), a time-out scheme may be implemented

Three-Phase Commit Protocol

Two-phase commit is a blocking protocol, even with timeout transitions



Malfunction-Stop Modules

Malfunction tolerance would be much easier if modules simply stopped functioning, rather than engage in arbitrary behavior

Unpredictable (Byzantine) malfunctions are notoriously hard to handle

Assuming the availability of a reliable stable storage along with its controlling s-process and (approximately) synchronized clocks, a k-malfunction-stop module can be implemented from $k + 1$ units

Operation of s-process to decide whether the module has stopped:

R := bag of received requests with appropriate timestamps
if $|R| = k+1$ $\wedge$ all requests identical and from different sources $\wedge \neg$ *stop*
then if request is a write
    then perform the write operation in stable storage
    else if request is a read, send value to all processes
else set variable *stop* in stable storage to TRUE

Malfunction-Tolerant Scheduling

Scheduling problems are hard even when resource requirements and availability are both fixed and known a priori

When resource availability is fixed, the quality of a schedule is judged by:
(1) Completion times (2) Meeting of deadlines

When resources fluctuate, deadlines may be met probabilistically or accuracy/completeness may be traded off for timeliness