## STRUCTURE AT A GLANCE

| Part I — Introduction: Dependable Systems (The Ideal-System View) | Goals | 1. Background and Motivation  
2. Dependability Attributes  
3. Combinatorial Modeling  
4. State-Space Modeling |
| Part II — Defects: Physical Imperfections (The Device-Level View) | Methods | 5. Defect Avoidance  
6. Defect Circumvention  
7. Shielding and Hardening  
8. Yield Enhancement |
10. Fault Masking  
11. Design for Testability  
12. Replication and Voting |
14. Error Correction  
15. Self-Checking Modules  
16. Redundant Disk Arrays |
| Part V — Malfunctions: Architectural Anomalies (The Structure-Level View) | Examples | 17. Malfunction Diagnosis  
18. Malfunction Tolerance  
19. Standby Redundancy  
20. Robust Parallel Processing |
22. Degradation Management  
23. Resilient Algorithms  
24. Software Redundancy |
| Part VII — Failures: Computational Breaches (The Result-Level View) | Examples | 25. Failure Confinement  
26. Failure Recovery  
27. Agreement and Adjudication  
28. Fall-Safe System Design |

Appendix: Past, Present, and Future
About This Presentation

This presentation is intended to support the use of the textbook *Dependable Computing: A Multilevel Approach* (traditional print or on-line open publication, TBD). It is updated regularly by the author as part of his teaching of the graduate course ECE 257A, Fault-Tolerant Computing, at Univ. of California, Santa Barbara. Instructors can use these slides freely in classroom teaching or for other educational purposes. Unauthorized uses, including distribution for profit, are strictly prohibited. © Behrooz Parhami

<table>
<thead>
<tr>
<th>Edition</th>
<th>Released</th>
<th>Revised</th>
<th>Revised</th>
<th>Revised</th>
<th>Revised</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Nov. 2020</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
21 Degradation Allowance
“Redundancy is such an ugly word. Let’s talk about your ‘employment crunch’.”

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

Our computers are down, so we have to do everything manually...
# STRUCTURE AT A GLANCE

| Part I — Introduction: Dependable Systems (The Ideal-System View) | Goals | 1. Background and Motivation  
2. Dependability Attributes  
3. Combinational Modeling  
4. State-Space Modeling |
| Part II — Defects: Physical Imperfections (The Device-Level View) | Methods | 5. Defect Avoidance  
6. Defect Circumvention  
7. Shielding and Hardening  
8. Yield Enhancement |
10. Fault Masking  
11. Design for Testability  
12. Replication and Voting |
14. Error Correction  
15. Self-Checking Modules  
16. Redundant Disk Arrays |
18. Malfunction Tolerance  
19. Standby Redundancy  
20. Robust Parallel Processing |
22. Degradation Management  
23. Resilient Algorithms  
24. Software Redundancy |
| Part VII — Failures: Computational Breaches (The Result-Level View) | Methods | 25. Failure Confinement  
26. Failure Recovery  
27. Agreement and Adjudication  
28. Fail-Safe System Design |

Appendix: Past, Present, and Future

---

Nov. 2020 Part VI – Degradations: Behavioral Lapses Slide 5
21.1 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
Diagnose malfunctions and provide capability for the system to work without the modules which are malfunctioning

Degradation management
Adapt: Prioritize tasks and redistribute load
Monitor: Keep track of system operation in degraded mode
Reverse: Return system to the intact (or less degraded) state ASAP
Return: Go back to normal operation
Degradation Allowance Is Not Automatic

A car possessing extra wheels compared with the minimum number required does not guarantee that it can operate with fewer wheels
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
21.2 Diagnosis, Isolation, and Repair

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?)

Recover processes and associated data
- 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
21.3 Stable Storage

Storage that won’t lose its contents (unlike registers and SRAM/DRAM)

Possible implementation method: Battery backup for a time duration long enough to save contents of disk cache or other volatile memory

Flash memory

Combined stability & reliability can be provided with RAID-like methods
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:

$R := \text{bag of received requests with appropriate timestamps}$

if $|R| = k+1 \land \text{all requests identical and from different sources} \land \neg \text{stop}$

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 $\text{stop}$ in stable storage to TRUE
21.4 Process and Data Recovery

Use of logs with process restart

Impossible when the system operates in real time and performs actions that cannot be undone

Such actions must be compensated for as part of degradation management
21.5 Checkpointing and Rollback

If MTTF is shorter than the running time, many restarts may be needed.

Early computers had a short MTTF. It was impossible to complete any computation that ran for several hours.

Checkpoints are placed at convenient points along the computation path (not necessarily at equal intervals).

Checkpointing entails some overhead. Too few checkpoints would lead to a lot of wasted work. Too many checkpoints would lead to a lot of overhead.
Why Checkpointing Helps

A computation’s running time is $T = 2 \text{MTTF} = \frac{2}{\lambda}$. What is the probability that we can finish the computation in time $2T$:

a. Assuming no checkpointing

b. Assuming checkpointing at regular intervals of $T/2$

Ignore all overheads.

<table>
<thead>
<tr>
<th>Chkpt #1</th>
<th>Chkpt #2</th>
<th>Chkpt #3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T = 2 \text{MTTF}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Long-running comp.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

47% success probability

25% success probability

$S \rightarrow H \rightarrow C$

$S \rightarrow C$

$e^{-1}$

$e^{-2}$
Recovery via Rollback

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)
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

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:

<table>
<thead>
<tr>
<th>Time in state:</th>
<th>$k$</th>
<th>Availability</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0.922</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0.987</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0.996</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0.997</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>0.997</td>
</tr>
</tbody>
</table>

Alternative:
- Primary site does frequent back-ups

Update requests sent to backup sites
Request is honored after all messages ack’ed

Normal state (primary OK, data available)
Recovery state (primary site is changing)
Checkpoint state (primary doing back-up)
Idle (no site is OK)
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
21.6 Optimal Checkpoint Insertion

There is a clear tradeoff in the decision regarding checkpoints:
- Too few checkpoints lead to long rollbacks in the event of a malfunction.
- Too many checkpoints lead to excessive time overhead.

As in many other engineering problems, there is a happy medium.
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
\( \lambda = \) Malfunction rate

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

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

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

**Example:** \( T = 200 \) hr, \( \lambda = 0.01 / \) hr, \( T_{cp} = 1/8 \) hr
\( q^{opt} = 200(0.01 + (0.01/0.25)^{1/2}) = 42; \quad T_{total}^{opt} \approx 215 \) hr

**Warning:** Model is accurate only when \( T/q \ll 1/\lambda \)
Elaboration on Optimal Computation Checkpoints

\( T = \text{Total computation time without checkpointing} \) (Example: 200 hr)
\( q = \text{Number of computation segments; there will be } q - 1 \text{ checkpoints} \)
\( T_{cp} = \text{Time needed to capture a checkpoint snapshot} \) (Range: 1/8-10 hr)
\( \lambda = \text{Malfunction rate} \) (Example: 0.01 / hr)

Computation time with checkpointing

\[
T_{\text{total}} = T + \lambda T^2/(q - \lambda T) + (q - 1)T_{cp}
\]

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

\[
d^2T_{\text{total}}/dq^2 = 2\lambda T^2/(q - \lambda T)^3 > 0 \quad \Rightarrow \quad \text{Bowl-like curve for } T_{\text{total}}, \text{ with a minimum}
\]

Example:

<table>
<thead>
<tr>
<th>( T_{\text{total}} )</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
<th>50</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>400</td>
<td>300</td>
<td>267</td>
<td>250</td>
<td>222</td>
<td>214</td>
<td>211</td>
<td>208</td>
</tr>
<tr>
<td>1/8</td>
<td>400</td>
<td>301</td>
<td>267</td>
<td>251</td>
<td>225</td>
<td>218</td>
<td>215</td>
<td><strong>214</strong></td>
</tr>
<tr>
<td>1/3</td>
<td>401</td>
<td>302</td>
<td>269</td>
<td>253</td>
<td>229</td>
<td>224</td>
<td><strong>224</strong></td>
<td><strong>224</strong></td>
</tr>
<tr>
<td>1</td>
<td>403</td>
<td>305</td>
<td>274</td>
<td>259</td>
<td><strong>241</strong></td>
<td>243</td>
<td>250</td>
<td>257</td>
</tr>
<tr>
<td>10</td>
<td>430</td>
<td>350</td>
<td><strong>337</strong></td>
<td>340</td>
<td>412</td>
<td>504</td>
<td>601</td>
<td>698</td>
</tr>
</tbody>
</table>

\( d^2T_{\text{total}}/dq^2 > 0 \) implies a bowl-like curve for the total computation time, with a minimum at the optimal number of checkpoints.
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

\[ O = \frac{T_{cp} + T_{rb}}{P_{cp}} = \frac{T_{cp}}{P_{cp}} + \lambda(a + bP_{cp}/2) \]

is minimized for: \( P_{cp} = \sqrt{\frac{2T_{cp}}{(\lambda b)}} \)

\( T_{cp} = \text{Time needed to capture a checkpoint snapshot} = 16 \text{ min} \)
\( \lambda = \text{Malfunction rate} = 0.0005 / \text{min} \) \( \text{(MTTF} = 2000 \text{ min} \approx 33.3 \text{ hr}) \)
\( b = 0.1 \)

\[ P_{cp}^{\text{opt}} = \sqrt{\frac{2T_{cp}}{(\lambda b)}} = 800 \text{ min} \approx 13.3 \text{ hr} \]

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

\[ P_{cp}^{\text{opt}} = \sqrt{\frac{2T_{cp}}{(\lambda b)}} = 200 \text{ min} \approx 3.3 \text{ hr} \]
22 Degradation Management
“Budget cuts.”

“....and who's been messing with my desktop configuration?”

“Honesty, would human cloning make a difference?”

Nov. 2020

Part VI – Degradations: Behavioral Lapses

Slide 24
| Part I — Introduction: Dependable Systems | Goals | 1. Background and Motivation
(The Ideal-System View) |
|------------------------------------------|-------|--------------------------|
|                                          | Models| 2. Dependability Attributes
|                                          |       | 3. Combinational Modeling |
|                                          |       | 4. State-Space Modeling   |

| Part II — Defects: Physical Imperfections | Methods | 5. Defect Avoidance
(The Device-Level View) |
|-------------------------------------------|---------|------------------------|
|                                          | Examples| 6. Defect Circumvention
|                                          |         | 7. Shielding and Hardening
|                                          |         | 8. Yield Enhancement     |

(The Circuit-Level View) |
|---------------------------------------|---------|-------------------------|
|                                       | Examples| 10. Fault Masking
|                                       |         | 11. Design for Testability
|                                       |         | 12. Replication and Voting  |

(The State-Level View) |
|-------------------------------------------|---------|----------------------|
|                                          | Examples| 14. Error Correction
|                                          |         | 15. Self-Checking Modules
|                                          |         | 16. Redundant Disk Arrays

| Part V — Malfunctions: Architectural Anomalies | Methods | 17. Malfunction Diagnosis
(The Structure-Level View) |
|-----------------------------------------------|---------|--------------------------|
|                                               | Examples| 18. Malfunction Tolerance
|                                               |         | 19. Standby Redundancy     |
|                                               |         | 20. Robust Parallel Processing

(The Service-Level View) |
|------------------------------------------|---------|---------------------------|
|                                          | Examples| 22. Degradation Management
|                                          |         | 23. Resilient Algorithms   |
|                                          |         | 24. Software Redundancy   |

| Part VII — Failures: Computational Breaches | Methods | 25. Failure Confinement
(The Result-Level View) |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Examples</td>
<td>26. Failure Recovery</td>
</tr>
<tr>
<td></td>
<td></td>
<td>27. Agreement and Adjudication</td>
</tr>
<tr>
<td></td>
<td></td>
<td>28. Fail-Safe System Design</td>
</tr>
</tbody>
</table>

Appendix: Past, Present, and Future

Nov. 2020 Part VI – Degradations: Behavioral Lapses  

Slide 25
22.1 Data Distribution Methods

Reliable data storage requires that the availability and integrity of data not be dependent on the health of any one site

Data Replication

Data dispersion
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 later
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.

- **Original data word and its \( k \) pieces**
- **The \( k \) pieces after encoding (approx. three times larger)**
- **Possible read set of size \( 2k/3 \)**
- **Up-to-date pieces**
- **Possible update set of size \( 2k/3 \)**
- **Reconstruction algorithm**
- **Original data word recovered from \( k/3 \) encoded pieces**
22.2 Multiphase Commit Protocols

The two generals problem: Two generals lead divisions of an army camped on the mountains on the two sides of an enemy-occupied valley. The two divisions can only communicate via messengers. We need a scheme for the generals to agree on a common attack time, given that attack by only one division would be disastrous.

Messengers are totally reliable, but may need an arbitrary amount of time to cross the valley (they may even be captured and never arrive).

G1 decides on $T$, sends a messenger to tell G2.

G2 acknowledges receipt of the attack time $T$.

G2, unsure whether G1 got the ack (without which he would not attack), will need an ack of the ack!

This can go on forever, without either being sure.
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

Coordinators

Wait

- / Begin to all

Yes from all / Commit to all
Yes from all / Commit to all

Commit
Commit

Abort
Abort

Participants

Wait

Begin / Yes

Commit / –
Commit / –

Yes from all / Commit to all
Yes from all / Commit to all

Abort / –
Abort / –

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

Safe from blocking, given the absence of a local state that is adjacent to both a commit and an abort state
22.3 Dependable 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
22.4 Dependable Collaboration

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

Interconnect malfunctions are dealt with by synthesizing reliable communication primitives (point-to-point, broadcast, multicast).

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

- **Crash**: Node stops (does not undergo incorrect transitions).
- **Omission**: Node does not respond to some inputs.
- **Timing**: Node responds either too early or too late.
- **Byzantine**: Totally arbitrary behavior.

---

**UCSB**
Malfunction Detectors in Distributed Systems

Malfunction detector: Distributed oracle related to malfunction detection
Creates and maintains a list of suspected processes
Defined by two properties: completeness and accuracy

**Advantages:**

Allows decoupling of the effort to detect malfunctions, e.g. site crashes, from that of the actual computation, leading to more modular design

Improves portability, because the same application can be used on a different platform if suitable malfunction detectors are available for it

**Example malfunction detectors:**

\( \mathcal{P} \) (Perfect): strong completeness, strong accuracy (min required for IC)

\( \Diamond S \): strong completeness, eventual weak accuracy (min for consensus)

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)
22.5 Remapping and Load Balancing

When pieces of a computation are performed on different modules, remapping may expose hidden malfunctions.

After remapping, various parts of the computation are performed by different modules compared with the original mapping.

It is quite unlikely that the same incorrect answers are obtained in the remapped version.

Load balancing is the act of redistributing the computational load in the face of lost/recovered resources and dynamically changing computational requirements.
Recomputation 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.
22.6 Modeling of Degradable Systems

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).
23 Resilient Algorithms
### STRUCTURE AT A GLANCE

| Part I — Introduction: Dependable Systems (The Ideal-System View) | Goals | 1. Background and Motivation  
2. Dependability Attributes  
3. Combinational Modeling  
4. State-Space Modeling |
|---|---|---|
| Models | 5. Defect Avoidance  
6. Defect Circumvention  
7. Shielding and Hardening  
8. Yield Enhancement |
10. Fault Masking  
11. Design for Testability  
12. Replication and Voting |
14. Error Correction  
15. Self-Checking Modules  
16. Redundant Disk Arrays |
| Examples | Part IV — Errors: Informational Distortions (The State-Level View) | Methods | 17. Malfunction Diagnosis  
18. Malfunction Tolerance  
19. Standby Redundancy  
20. Robust Parallel Processing |
22. Degradation Management  
23. Resilient Algorithms  
24. Software Redundancy |
| Examples | Part VI — Degradations: Behavioral Lapses (The Service-Level View) | Methods | 25. Failure Confinement  
26. Failure Recovery  
27. Agreement and Adjudication  
28. Fail-Safe System Design |
| Examples |

Appendix: Past, Present, and Future

Nov. 2020 Part VI – Degradations: Behavioral Lapses
23.1 COTS-Based Paradigms

Many of the hardware and software redundancy methods 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:

- **ARM**: Fault-tolerant ARM (launched in late 2006), automotive applications
- **Nth Generation Computing**: High-availability and enterprise storage systems
- **Resilience Corp.**: Emphasis on data 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 $\mu$s 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 [Aviz97], 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
23.2 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: $(\sum jx_j) \mod A$

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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>$x - y$</td>
</tr>
<tr>
<td>$y$</td>
<td>$y - z$</td>
</tr>
<tr>
<td>$z$</td>
<td></td>
</tr>
</tbody>
</table>

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
23.3 Data Diversity and Fusion

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 $\alpha$ between them
By the radii $r$ and $R$ of its inscribed and circumscribed circles

Area calculations with computation and data diversity

\[ A = xy \]
\[ A = \frac{1}{2} z^2 \sin \alpha \]
\[ A = 4r(R^2 - r^2)^{1/2} \]
23.4 Self-Checking Algorithms

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

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

Full checksum matrix

$$
M_f = \begin{pmatrix}
\text{dashed} & 2 & 1 & 6 \\
\text{dashed} & 5 & 3 & 4 \\
\text{dashed} & 3 & 2 & 7 \\
2 & 6 & 1 & 1
\end{pmatrix}
$$
Matrix Multiplication Using ABET

If \( Z = X \times Y \) then
\[ Z_f = X_c \times Y_r \]

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

\[
\begin{align*}
46 + 20 + 42 &= 108 = 4 \text{ mod } 8 \\
36 &= 4 \text{ mod } 8 \\
20 + 41 + 30 &= 91 = 3 \text{ mod } 8 \\
35 &= 3 \text{ mod } 8
\end{align*}
\]
23.5 Self-Adapting Algorithms

This section to be completed
23.6 Other Algorithmic Methods

This section to be completed
“Well, what’s a piece of software without a bug or two?”

“We are neither hardware nor software; we are your parents.”

“There’s nothing wrong with your personal finance software. You just don’t have any money.”

“That’s our CIO. He’s encrypted for security purposes.”

“I haven’t the slightest idea who he is. He came bundled with the software.”
## STRUCTURE AT A GLANCE

| Part I — Introduction: Dependable Systems (The Ideal-System View) | 1. Background and Motivation  
2. Dependability Attributes  
3. Combinational Modeling  
4. State-Space Modeling |
|---------------------------------------------------------------|--------------------------|
| Part II — Defects: Physical Imperfections (The Device-Level View) | 5. Defect Avoidance  
6. Defect Circumvention  
7. Shielding and Hardening  
8. Yield Enhancement |
| Part III — Faults: Logical Deviations (The Circuit-Level View) | 9. Fault Testing  
10. Fault Masking  
11. Design for Testability  
12. Replication and Voting |
| Part IV — Errors: Informational Distortions (The State-Level View) | 13. Error Detection  
14. Error Correction  
15. Self-Checking Modules  
16. Redundant Disk Arrays |
| Part V — Malfunctions: Architectural Anomalies (The Structure-Level View) | 17. Malfunction Diagnosis  
18. Malfunction Tolerance  
19. Standby Redundancy  
20. Robust Parallel Processing |
22. Degradation Management  
23. Resilient Algorithms  
24. Software Redundancy |
| Part VII — Failures: Computational Breaches (The Result-Level View) | 25. Failure Confinement  
26. Failure Recovery  
27. Agreement and Adjudication  
28. Fail-Safe System Design |
| Appendix: Past, Present, and Future |

Nov. 2020  
Part VI – Degradations: Behavioral Lapses  
Slide 58
Imagine the following product disclaimers:

**For a steam iron**

There is no guarantee, explicit or implied, that this device will remove wrinkles from clothing or that it will not lead to the user’s electrocution. The manufacturer is not liable for any bodily harm or property damage resulting from the operation of this device.

**For an electric toaster**

The name “toaster” for this product is just a symbolic identifier. There is no guarantee, explicit or implied, that the device will prepare toast. Bread slices inserted in the product may be burnt from time to time, triggering smoke detectors or causing fires. By opening the package, the user acknowledges that s/he is willing to assume sole responsibility for any damages resulting from the product’s operation.
How Is Software Different from Hardware?

Software unreliability is caused predominantly by design slips, not by operational deviations – we use *flaw* or *bug*, rather than *fault* or *error*.

Not much sense in replicating the same software and doing comparison or voting, as we did for hardware.

At the current levels of hardware complexity, latent design slips also exist in hardware, thus the two aren’t totally dissimilar.

The curse of complexity

The 7-Eleven convenience store chain spent nearly $9M to make its point-of-sale software Y2K-compliant for its 5200 stores. The modified software was subjected to 10,000 tests (all successful). The system worked with no problems throughout the year 2000. On January 1, 2001, however, the system began rejecting credit cards, because it “thought” the year was 1901 (bug was fixed within a day).
Software Development Life Cycle

- Project initiation
- Needs
- Requirements
- Specifications
- Prototype design
- Prototype test
- Revision of specs
- Final design
- Coding
- Unit test
- Integration test
- System test
- Acceptance test
- Field deployment
- Field maintenance
- System redesign
- Software discard

Software flaws may arise at several points within these life-cycle phases

- Evaluation by both the developer and customer
- Implementation or programming
- Separate testing of each major unit (module)
- Test modules within pretested control structure
- Customer or third-party conformance-to-specs test
- New contract for changes and additional features
- Obsolete software is discarded (perhaps replaced)
What Is Software Dependability?

Major structural and logical problems are removed very early in the process of software testing.

What remains after extensive verification and validation is a collection of tiny flaws which surface under rare conditions or particular combinations of circumstances, thus giving software failure a statistical nature.

Software usually contains one or more flaws per thousand lines of code, with < 1 flaw considered good (Linux has been estimated to have 0.1).

If there are $f$ flaws in a software component, the hazard rate, that is, rate of failure occurrence per hour, is $kf$, with $k$ being the constant of proportionality which is determined experimentally (e.g., $k = 0.0001$).

Software reliability: $R(t) = e^{-kft}$

The only way to improve software reliability is to reduce the number of residual flaws through more rigorous verification and/or testing.
Residual Software Flaws

Input space

Flaw

Not expected to occur
24.2 Software Malfunction Models

Software flaw/bug $\Rightarrow$ Operational error $\Rightarrow$ Software-induced failure

“Software failure” used informally to denote any software-related problem

Removing flaws, without generating new ones

New flaws introduced are proportional to removal rate
Software Reliability Models and Parameters

For simplicity, we focus on the case of no new flaw generation

Assume linearly decreasing flaw removal rate ($F = \text{residual flaws}, \tau = \text{testing time, in months}$)

\[
\frac{dF(\tau)}{d\tau} = -(a - b\tau)
\]

\[
F(\tau) = F_0 - a\tau (1 - b\tau/(2a))
\]

Example: $F(\tau) = 130 - 30\tau(1 - \tau/16)$

Hazard function

\[
z(\tau) = k(F_0 - a\tau (1 - b\tau/(2a)))
\]

In our example, let $k = 0.000132$

\[
R(t) = \exp(-0.000132(130 - 30\tau(1 - \tau/16))t)
\]

Assume testing for $\tau = 8$ months:

\[
R(t) = e^{-0.00132t}
\]
The Phenomenon of Software Aging

Software does not wear out or age in the same sense as hardware.

Yet, we do observe deterioration in software that has been running for a long time.

So, the bathtub curve is also applicable to software.

Reasons for and types of software aging:

- Accumulation of junk in the state part (reversible via restoration)
- Long-term cumulative effects of updates (patches and the like)

As the software’s structure deviates from its original clean form, unexpected failures begin to occur.

Eventually software becomes so mangled that it must be discarded and redeveloped from scratch.
More on Software Reliability Models

Linearly decreasing flaw removal rate isn’t the only option in modeling.

Constant flaw removal rate has also been considered, but it does not lead to a very realistic model.

Exponentially decreasing flaw removal rate is more realistic than linearly decreasing, since flaw removal rate never really becomes 0.

How does one go about estimating the model constants?

- Use handbook: public ones, or compiled from in-house data.
- Match moments (mean, 2\textsuperscript{nd} moment, . . .) to flaw removal data.
- Least-squares estimation, particularly with multiple data sets.
- Maximum-likelihood estimation (a statistical method).
24.3 Software Verification and Validation

Verification: “Are we building the system right?” (meets specifications)
Validation: “Are we building the right system?” (meets requirements)

Both verification and validation use testing as well as formal methods

Software testing
Exhaustive testing impossible
Test with many typical inputs
Identify and test fringe cases

Example: overlap of rectangles

Formal methods
Program correctness proof
Formal specification
Model checking

Examples: safety/security-critical

Railway interlocking system [Hlavaty 2001]
Cryptography device [Kirby 1999]
Smart cards [Requet 2000]
Automated lab analysis test equipment [Bicarregui 1997]
 Formal Proofs for Software Verification

Program to find the greatest common divisor of integers \( m > 0 \) and \( n > 0 \)

\[
\begin{align*}
\text{input} & \quad m \text{ and } n \\
x & := m \\
y & := n \\
\text{while} & \quad x \neq y \\
\quad \text{if} & \quad x < y \\
\quad & \quad y := y - x \\
\quad \text{else} & \quad x := x - y \\
\text{endif} \\
\text{ endwhile} \\
\text{output} & \quad x = \text{gcd}(m, n)
\end{align*}
\]

\( m \) and \( n \) are positive integers
\( x \) and \( y \) are positive integers, \( x = m, y = n \)
\text{Loop invariant: } x > 0, y > 0, \text{gcd}(x, y) = \text{gcd}(m, n)

Steps 1-3: “partial correctness”
Step 4: ensures “total correctness”

The four steps of a correctness proof relating to a program loop:
1. Loop invariant implied by the assertion before the loop (precondition)
2. If satisfied before an iteration begins, then also satisfied at the end
3. Loop invariant and exit condition imply the assertion after the loop
4. Loop executes a finite number of times (termination condition)
Software Flaw Tolerance

Flaw avoidance strategies include (structured) design methodologies, software reuse, and formal methods.

Given that a complex piece of software will contain bugs, can we use redundancy to reduce the probability of software-induced failures?

The ideas of masking redundancy, standby redundancy, and self-checking design have been shown to be applicable to software, leading to various types of fault-tolerant software.

“Flaw tolerance” is a better term; “fault tolerance” has been overused.

Masking redundancy: N-version programming
Standby redundancy: the recovery-block scheme
Self-checking design: N-self-checking programming

Sources: *Software Fault Tolerance*, ed. by Michael R. Lyu, Wiley, 2005
24.4 N-Version Programming

Independently develop \( N \) different programs (known as “versions”) from the same initial specification.

The greater the diversity in the \( N \) versions, the less likely that they will have flaws that produce correlated errors.

**Diversity in:**
- Programming teams (personnel and structure)
- Software architecture
- Algorithms used
- Programming languages
- Verification tools and methods
- Data (input re-expression and output adjustment)
Some Objections to N-Version Programming

Developing programs is already a very expensive and slow process; why multiply the difficulties by $N$?

Diversity does not ensure independent flaws (It has been amply documented that multiple programming teams tend to overlook the same details and to fall into identical traps, thereby committing very similar errors)

Imperfect specification can be the source of common flaws

With truly diverse implementations, the output selection mechanism (adjudicator) is complicated and may contain its own flaws

Cannot produce flawless software, regardless of cost

This is a criticism of reliability modeling with independence assumption, not of the method itself

Multiple diverse specifications?

Will discuss the adjudication problem in a future lecture
Reliability Modeling for N-Version Programs

Fault-tree model: the version shown here is fairly simple, but the power of the method comes in handy when combined hardware/software modeling is attempted.

Probabilities of coincident flaws are estimated from experimental failure data.

Table 5.6  Error characteristics for four-version configurations

<table>
<thead>
<tr>
<th>Category</th>
<th>BY-CASE</th>
<th></th>
<th></th>
<th>BY-FRAME</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Number of cases</td>
<td>Frequency</td>
<td>Number of cases</td>
<td>Frequency</td>
<td></td>
</tr>
<tr>
<td>$F_0$ - no errors</td>
<td>322010</td>
<td>0.65052</td>
<td></td>
<td>2613781410</td>
<td>0.9998951</td>
</tr>
<tr>
<td>$F_1$ - single error</td>
<td>152900</td>
<td>0.30889</td>
<td></td>
<td>2719200</td>
<td>0.001040</td>
</tr>
<tr>
<td>$F_2$ - two coincident</td>
<td>16350</td>
<td>0.03303</td>
<td></td>
<td>2070</td>
<td>0.00000079</td>
</tr>
<tr>
<td>$F_3$ - three coincident</td>
<td>3700</td>
<td>0.00747</td>
<td></td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>$F_4$ - four coincident</td>
<td>40</td>
<td>0.00008</td>
<td></td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>Total</td>
<td>495000</td>
<td>1.00000</td>
<td></td>
<td>2614055400</td>
<td>1.000000</td>
</tr>
</tbody>
</table>

Source: Dugan & Lyu, 1994 and 1995
Applications of N-Version Programming

Back-to-back testing: multiple versions can help in the testing process

B777 flight computer: 3 diverse processors running diverse software

Airbus A320/330/340 flight control: 4 dissimilar hardware/software modules drive two independent sets of actuators

Some experiments in N-version programming

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Specs</th>
<th>Languages</th>
<th>Versions</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Halden, Reactor Trip</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>[Dah79]</td>
</tr>
<tr>
<td>NASA, First Generation</td>
<td>3</td>
<td>1</td>
<td>18</td>
<td>[Kei83]</td>
</tr>
<tr>
<td>KFK, Reactor Trip</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>[Gme80]</td>
</tr>
<tr>
<td>NASA/RTI, Launch Interceptor</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>[Dun86]</td>
</tr>
<tr>
<td>UCI/UVA, Launch Interceptor</td>
<td>1</td>
<td>1</td>
<td>27</td>
<td>[Kni86a]</td>
</tr>
<tr>
<td>Halden (PODS), Reactor Trip</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>[Bis86]</td>
</tr>
<tr>
<td>UCLA, Flight Control</td>
<td>1</td>
<td>6</td>
<td>6</td>
<td>[Avi88]</td>
</tr>
<tr>
<td>NASA (2nd Gen.) Inertial Guidance</td>
<td>1</td>
<td>1</td>
<td>20</td>
<td>[Eck91]</td>
</tr>
<tr>
<td>UI/Rockwell, Flight Control</td>
<td>1</td>
<td>1</td>
<td>15</td>
<td>[Lyu93]</td>
</tr>
</tbody>
</table>

Source: P. Bishop, 1995
24.5 The Recovery Block Method

The software counterpart to standby sparing for hardware

Suppose we can verify the result of a software module by subjecting it to an acceptance test

\[
\begin{array}{|c|c|}
\hline
\text{ensure by} & \text{acceptance test} \\
\text{else by} & \text{primary module} \\
\text{else by} & \text{first alternate} \\
\text{else by} & \text{last alternate} \\
\text{else fail} & \text{else fail} \\
\hline
\end{array}
\]

e.g., sorted list
e.g., quicksort
e.g., bubblesort

The acceptance test can range from a simple reasonableness check to a sophisticated and thorough test

Design diversity helps ensure that an alternate can succeed when the primary module fails
The Acceptance Test Problem

Design of acceptance tests (ATs) that are both simple and thorough is very difficult; for example, to check the result of sorting, it is not enough to verify that the output sequence is monotonic.

Simplicity is desirable because acceptance test is executed after the primary computation, thus lengthening the critical path.

Thoroughness ensures that an incorrect result does not pass the test (of course, a correct result always passes a properly designed test).

Some computations do have simple tests (inverse computation). Examples: square-rooting can be checked through squaring, and roots of a polynomial can be verified via polynomial evaluation.

At worst, the acceptance test might be as complex as the primary computation itself.
24.6 Hybrid Software Redundancy

**Recoverable N-version block scheme = N-self-checking program**

Voter acts only on module outputs that have passed an acceptance test.

**Consensus recovery block scheme**

Only when there is no majority agreement, acceptance test applied (in a prespecified order) to module outputs until one passes its test.

More General Hybrid NVP-AT Schemes

(a) Legend

(b) 5VP

(c) ALT1