Cyber Infrastructure for the Power Grid

Dr. Anurag K. Srivastava, Dr. Carl Hauser, Dr. Dave Bakken

Computation Lecture 3: Overview of Fault-Tolerant Computing
(Not Distributed Systems 1 as I had earlier planned)
Today’s Content

1. Administrivia: Future Alumni Training

2. A Definition of Dependability (6.1)
   A. Basic Definitions
   B. Achieving, Measuring, and Validating Dependability
   C. Fault Assumptions

3. Fault-Tolerant Computing (6.2)

4. Fault-Tolerant Architectures (6.5)

Note: “6.2” is from chapters in an optional text

23. What movie was the WSU fight song sung in?
   a) Conscripts
   b) Volunteers
   c) Shanghai’d
   d) Citizen Kane

24. What fighting force sang the WSU fight song?
   a) Viet Cong
   b) North Vietnamese Army
   c) Khmer Rouge
   d) Bashi-bazouk

25. What is the color of hemorrhoids?
   a) Purple
   b) Purple
   c) Purple
   d) Purple

26. What is the color of concentrated urine?
   a) Gold
   b) Gold
   c) Gold
   d) Gold

27. What is the name of our rivalry game?
   a) Orange Bowl
   b) Fig Leaf
   c) Evergreen Bowl
   d) Apple Cup

1Caution: this statement has not been evaluated by the US Food and Drug Administration
Today’s Content

1. Administrivia

2. A Definition of Dependability (6.1)
   A. Basic Definitions
   B. Achieving, Measuring, and Validating Dependability
   C. Fault Assumptions

3. Fault-Tolerant Computing (6.2)
A Definition of Dependability (6.1)

- **Dependability** deals with having a high probability of behaving according to specification (informal definition)

- Implications
  - Need a comprehensive specification
  - Need to specify not only functionality but assumed environmental conditions
  - Need to clarify what “high” means (context-dependent)
Defining Dependability (cont.)

- **Dependability**: the measure in which reliance can justifiably be placed on the service delivered by a system
  - **Q**: what issues does this definition raise?

- Is there a systematic way to achieve such justifiable reliance?
  - No silver bullets: fault tolerance is an art
  - Prereq #1: know impairments to dependability
  - Prereq #2: know means to achieve dependability
  - Prereq #3: devise ways of specifying/expressing level of dependability required
  - Prereq #4: measure if it the required level of dependability was achieved
Faults, Errors, and Failures

• Some definitions from the fault tolerance realm
  ▪ **Fault**: the adjudged (hypothesized) cause for an error
  ▪ **Note**: may lie dormant for some time
    – Running Example: file system disk defect or overwriting
    – Example: software bug
    – Example: if a man talks in the woods…..
  ▪ **Error**: incorrect system state
    – Running Example: wrong bytes on disk for a given record
  ▪ **Failure**: component no longer meets its specification
    – I.e., the problem is visible outside the component
    – Running Example: file system API returns the wrong byte

• Sequence (for a given component):
  Fault ➔ Error ➔ Failure
Cascading Faults, Errors, and Failures

• Can cascade (if not handled)
  ▪ Scenario: Component 2 uses Component 1
  ▪ Let's see if you can get the terms right..

This is ….
This is ….
This is …. (of Component1)
This is …. (to Component2)
Fault Types

• Several axes/viewpoints by which to classify faults…

• **Phenomenological origin**
  ▪ Physical: HW causes
  ▪ Design: introduced in the design phase
  ▪ Interaction: occurring at interfaces between components

• **Nature**
  ▪ Accidental
  ▪ Intentional/malicious

• **Phase of creation** in system lifecycle
  ▪ Development
  ▪ Operations

• **Locus** (external or internal)

• **Persistence** (permanent or temporary)
More on Faults

- **Independent faults**: attributed to different causes
- **Related faults**: attributed to a common cause
- Related faults usually cause **common-mode failures**
  - Single power supply for multiple CPUs
  - Single clock
  - Single specification used for design diversity
# Scope of Fault Classification

<table>
<thead>
<tr>
<th>NATURE</th>
<th>ORIGIN</th>
<th>PERSISTENCE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Phenomenological Cause</td>
<td></td>
</tr>
<tr>
<td>Accidental Faults</td>
<td>Physical Faults</td>
<td>X</td>
</tr>
<tr>
<td>Intentional Faults</td>
<td>Human-made Faults</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>System Boundaries</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Internal Faults</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>External Faults</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Phase of Creation</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Design Faults</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Operational Faults</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Permanent Faults</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Temporary Faults</td>
<td></td>
</tr>
</tbody>
</table>

**Usual Labelling**

- **Physical Faults**
  - Physical Faults: X
  - Human-made Faults: X
  - Internal Faults: X
  - External Faults: X
  - Design Faults: X
  - Operational Faults: X
  - Permanent Faults: X
  - Temporary Faults: X

- **Transient Faults**
  - Physical Faults: X
  - Human-made Faults: X
  - Internal Faults: X
  - External Faults: X
  - Design Faults: X
  - Operational Faults: X
  - Permanent Faults: X
  - Temporary Faults: X

- **Intermittent Faults**
  - Physical Faults: X
  - Human-made Faults: X
  - Internal Faults: X
  - External Faults: X
  - Design Faults: X
  - Operational Faults: X
  - Permanent Faults: X
  - Temporary Faults: X

- **Design Faults**
  - Physical Faults: X
  - Human-made Faults: X
  - Internal Faults: X
  - External Faults: X
  - Design Faults: X
  - Operational Faults: X
  - Permanent Faults: X
  - Temporary Faults: X

- **Interaction Faults**
  - Physical Faults: X
  - Human-made Faults: X
  - Internal Faults: X
  - External Faults: X
  - Design Faults: X
  - Operational Faults: X
  - Permanent Faults: X
  - Temporary Faults: X

- **Malicious Logic**
  - Physical Faults: X
  - Human-made Faults: X
  - Internal Faults: X
  - External Faults: X
  - Design Faults: X
  - Operational Faults: X
  - Permanent Faults: X
  - Temporary Faults: X

- **Intrusions**
  - Physical Faults: X
  - Human-made Faults: X
  - Internal Faults: X
  - External Faults: X
  - Design Faults: X
  - Operational Faults: X
  - Permanent Faults: X
  - Temporary Faults: X
Today’s Content

1. Administrivia

2. A Definition of Dependability (6.1)
   A. Basic Definitions
   B. Achieving, Measuring, and Validating Dependability
   C. Fault Assumptions

3. Fault-Tolerant Computing (6.2)
Achieving Dependability (6.1 B)

• Chain of failures likely to cascade unless handled!
  ▪ To get dependability, break that chain somewhere!

• **Fault removal**: detecting and removing faults before they can cause an error
  ▪ Find software bugs, bad hardware components, etc.

• **Fault forecasting**: estimating the probability of faults occurring or remaining in system
  ▪ Can’t remove all kinds easily/cheaply!

• **Fault prevention**: preventing causes of errors
  ▪ Eliminate conditions that make fault occurrence probable during operation
    – Use quality components
    – Use components with internal redundancy
    – Rigorous design techniques

• **Fault avoidance**: fault prevention + fault removal
Achieving Dependability (cont.)

- Can’t always avoid faults, so better tolerate them!
- **Fault-Tolerant System**: a system that can provide service despite one or more faults occurring
  - Acts at the phase that errors are produced (operation)
- **Error detection**: finding the error in the first place
- **Error processing**: mechanisms that remove errors from computational state (hopefully before failure!)

2 Choices:

- **Error recovery**: substitute an error-free state for the erroneous one
  - **Backward recovery**: go back to a previous error-free state
  - **Forward recovery**: find a new state system can operate from
- **Error compensation**: erroneous state contains enough redundancy to enable delivery of error-free service from the erroneous state
Achieving Dependability (cont.)

- **Fault Treatment**: preventing faults from re-occurring

Steps:
- **Fault diagnosis**: determining cause(s) of the error(s)
- **Fault passivation**: preventing fault(s) from being activated again
  - Remove component
  - If can’t continue with this removed, need to reconfigure system
Measuring and Validating Dependability

• We’ve practiced fault avoidance & fault tolerance….
  ▪ But how good did we do???
  ▪ **Attributes** by which we measure and validate dependability…

• **Reliability**: probability that system does not fail during a given time period (e.g., mission or flight)
  ▪ **Mean time between failures** (MTBF): useful for continuous mission systems (a scalar)
  ▪ Other quantifications are
    - probability distribution functions (e.g., bathtub)
    - Scalar: **failures per hour** (e.g., 10⁻⁹)

• **Maintainability**: measure of time to restore correct service
  ▪ **Mean time to repair** (MTTR): a scalar measure
Measuring & Validating Dependability (cont).

- **Availability**: prob. a service is correctly functioning when needed (note: many sub-definitions…)
  - **Steady-state availability**: the fraction of time that a service is correctly functioning
    - $\frac{MTBF}{MTBF+MTTR}$
  - **Interval availability** (one explanation): the probability that a service will be correctly functioning during a time interval
    - E.g., during the assumed time for a client-server request-reply

- **Performability**: combined performance+dependability analysis
  - Quantifies how a system gracefully degrades

- **Safety**: degree that system failing is not catastrophic

- **Security**: $\text{Confidentiality} \land \text{Integrity} \land \text{Availability}$

Note: dependability measures vary w/ resources+usage
## Availability Examples

<table>
<thead>
<tr>
<th>Availability</th>
<th>9s</th>
<th>Downtime/year</th>
<th>Example Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>90%</td>
<td>1</td>
<td>&gt;1 month</td>
<td>Unattended PC</td>
</tr>
<tr>
<td>99%</td>
<td>2</td>
<td>~4 days</td>
<td>Maintained PC</td>
</tr>
<tr>
<td>99.9%</td>
<td>3</td>
<td>~9 hours</td>
<td>Cluster</td>
</tr>
<tr>
<td>99.99%</td>
<td>4</td>
<td>~1 hour</td>
<td>Multicomputer</td>
</tr>
<tr>
<td>99.999%</td>
<td>5</td>
<td>~5 minutes</td>
<td>Embedded System (w/PC technology)</td>
</tr>
<tr>
<td>99.9999%</td>
<td>6</td>
<td>~30 seconds</td>
<td>Embedded System (custom HW)</td>
</tr>
<tr>
<td>99.99997%</td>
<td>7</td>
<td>~3 seconds</td>
<td>Embedded System (custom HW)</td>
</tr>
</tbody>
</table>
Today’s Content

1. Administrivia

2. A Definition of Dependability (6.1)
   A. Basic Definitions
   B. Achieving, Measuring, and Validating Dependability
   C. Fault Assumptions

3. Fault-Tolerant Computing (6.2)
Fault Assumptions

• Can’t design to tolerate an arbitrary number and kind of faults! (IMHO; YMMV.)

• **Fault model**: number of classes of faults that have to be tolerated
  - AKA **failure model** (failure of a component being used)
  - 2 main groupings of fault model: omissive and assertive
  - In this segment we mainly deal with interaction faults
    - Q: why?

• Fault model done at atomic level of abstraction not possible or useful to go below
  - Nicely groups lower-level problems at the granularity that you would want to do something about it!
Omissive Fault Group

- **Omissive faults**: component not performing an interaction it was specified to
  - **Crash**: component permanently (but cleanly) stops
    - AKA “fail silent”
  - **Omission**: component periodically omits a specified interaction
    - **Omission degree**: # of successive omission faults
    - Note crash is an extreme case of omission: infinite omission degree
  - **Timing**: component is later (or earlier) than performing specified interaction
    - Note: omission is extreme case of timing fault: infinite lateness
Assertive and Arbitrary Faults

- **Assertive faults**: interactions not performed to spec.
  - **Syntactic**: wrong structure of interaction
    - E.g., sending a float instead of an int
  - **Semantic**: wrong meaning
    - E.g., bad value
    - E.g., temp sensor below absolute zero
    - E.g., Sensor very different from redundant sensors

- **Arbitrary faults**: union of omissive and assertive
  - **Note**: omissive faults occur in the time domain
  - **Note**: assertive faults occur in the value domain
  - Arbitrary can be either
Arbitrary Faults (cont.)

• Causes of arbitrary faults
  ▪ Improbable but possible sequence of events
  ▪ A bug
  ▪ Deliberate action by intruder

• Byzantine faults: subset of arbitrary
  ▪ Generally defined as sending bad values and often inconsistent semantic faults (“two-faced behavior”)
  ▪ One counter-example sub-case: a malicious early timing fault
    – Really a forged interaction
    – Non-malicious early timing fault happened to my lab machines in fall 2000…
Caveat: it’s a Byzantine (and Machievellian) world out there….

“You've got to ask yourself one question. Do you feel lucky? Well, do you... Punk”
Coverage

• To build a FT system you had to assume a fault model
  ▪ But how good (lucky?) were you in your assumptions???

• Q: which is “better”
  ▪ A system tolerating two arbitrary faults
  ▪ A system tolerating two omission faults
  ▪ A system tolerating one omission and one arbitrary fault

• Coverage: given a fault, it’s the probability that it will be tolerated

• Assumption coverage (informally): the probability that the fault model will not be violated
Causes of Failures

• Jim Gray (RIP) survey at Tandem (1986)
  ▪ Still relevant today

• Causes of failures (“How do computers fail…”)
  ▪ Plurality (42%) caused by incorrect system administration or human operators
  ▪ Second (25%) software faults
  ▪ Third: environmental (mainly power outage, but flood/fire)
  ▪ Last: hardware faults

• Lessons for the system architect (“…and what can be done about it?”)
  ▪ Dependability can be increased by careful admin/ops
  ▪ SWE methodologies that help with fault prevention and removal can significantly increase reliability
  ▪ Software fault tolerance is a very critical aspect
Today’s Content

1. Administrivia

2. A Definition of Dependability (6.1)
   A. Basic Definitions
   B. Achieving, Measuring, and Validating Dependability
   C. Fault Assumptions

3. Fault-Tolerant Computing (6.2)
Fault-Tolerant Computing (6.2)

• Recall: FT computing is techniques that prevent faults from becoming failures
  ▪ Quite a span of mechanisms…

• FT requires some kind(s) of redundancy (examples?)
  ▪ **Space redundancy**: having multiple copies of a component
  ▪ **Time redundancy**: doing the same thing more than once until desired effect achieved
    - Can be redone same way or different way
  ▪ **Value redundancy**: adding extra information about the value of the data being stored/sent
Error Processing

- Facets of error processing
  - **Error detection**: discovering the error
  - **Error recovery**: utilize enough redundancy to keep operating correctly despite the error
    - **Backward error recovery**: system goes back to a previous state known to be correct
    - **Forward error recovery**: system proceeds forward to a state where correct provision of service can still be ensured
      - Usually in a degraded mode
  - **Error masking**: providing correct service despite lingering errors
    - AKA **error compensation**
    - E.g., receiving replies from multiple servers and voting
Distributed Fault Tolerance (DFT)

- Modularity is important for FT
- DFT sys. built of nodes, networks, SW components
  - Key goal: decouple SW components from HW they run on
  - This modularity greatly helps reconfiguration and replication
Distributed Fault Tolerance (cont.)

• If right design techniques used, you can replace HW or SW components without changing the arch.

• Also lets you provide incremental dependability
  ▪ Adding more replicas
  ▪ Hardening fragile ones (fault prevention)
  ▪ Making more resilient to severe faults (fault tolerance)

• Can also support graceful degradation: system does not collapse quickly at some point, service provided at lower level
  ▪ Slower
  ▪ Less precise results

• Modularity also helps support heterogeneity
  ▪ Usually with distributed object middleware