Dependability Overview: Vocabulary and Techniques

George Candea
Stanford University

What Is Dependability ?

= trustworthiness of a computer system with regard to the services it provides

- Reliability = continuous functioning w/out failure
- Availability = readiness for usage
- Safety = avoid catastrophic effects on environment
- Security = prevent unauthorized access and/or handling of information

---

Historical Evolution

1. Make things work and keep them working → *reliability*
   Babbage's analytical engine
   MTBF(vacuum tubes) ˜ computation time
2. Widespread use (accounting, payroll, inventory, billing) → *availability*
3. Critical applications (FAA, early -warning, nuclear power plants, aircraft, ABS) → *safety*
4. Distributed systems → increased node accessibility and vulnerability → *security*

Some Definitions

- Specification = agreed description of a sw system's expected service
- Environment = any external entity that interacts w/ system (present/past/future). Beware of different system boundaries.
- User = that part of the environment that provides inputs to the service and/or receives outputs

- Function = what the system *should do*
- Behavior = what the system *does do*
  (hence, service = abstraction of system behavior)
- Structure = what makes the system *do what it does*

From Fault to Failure

- Failure = deviation of system's service from its spec
- Error = the part of system state that is liable
- Fault = adjudged/hypothesized cause of the error

Fault → Error → Failure

- Failure of one system is fault for another system (programmers, tools, operators, etc.)

… → Fault → Error → Failure → Fault → Error → Failure → …

- Distinction is hard, so make it at the level at which the fault is meant to be prevented or tolerated

---

Bugs = Software Faults

- Heisenbugs = intermittent design/implementation faults; the more you look for them, the more elusive they become
- Bohrbugs = permanent design/implementation faults (solid, easily identified)

Note:

- Intermittent fault ← internal
- Transient fault ← external

Failures

- Byzantine failure = system returns wrong values (named after Byzantine Empire)
- Stopping failure = system activity no longer perceived by user, delivers constant value service
- Omission failure = stopping failure w/ no service being delivered at all
- Crash failure = persistent omission failure

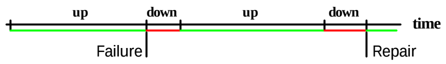
Crash < Omission < Stopping < Byzantine

Classification

- Benign failure: consequences ˜ potential benefit from up-ness (same order of magnitude)
- Catastrophic failure: consequences >> potential benefit from up-ness
- Fail-safe system = only benign failures
- Fail-stop system = only stopping failures (sometimes equivalent to fail-stop)
- Fail-silent system = only crash failures

Statistical Definitions

- Reliability: random variable *R(t)* = probability that system does not fail before time *t*
- Mean time to failure: *MTTF* = E[*R(t)*] (how long expect system to work w/out failure)
- Instantaneous availability: *A(t)* = probability that system is up at time *t* ; then Availability = $\lim_{t \to inf} A(t)$



$$\text{Availability} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}} = \frac{\text{MTTF}}{\text{MTBF}}$$

Techniques

- Three basic approach to faults/errors/failures:

1. Fault Prevention
2. Fault Containment
3. Fault Tolerance

1. Fault Prevention

- Better Software Engineering
- Formal Methods
- Language-Based Mechanisms
- Fault Forecasting

2. Fault Containment

- By Design
- Language-Based Mechanisms
- Virtualization

3. Fault Tolerance

- Redundancy
- Recovery
- Diversity

Overview

1. Fault Prevention
   - Better Software Engineering
   - Formal Methods
   - Language-Based Mechanisms
   - Fault Forecasting
2. Fault Containment
   - By Design
   - Language-Based Mechanisms
   - Virtualization
3. Fault Tolerance
   - Redundancy
   - Recovery
   - Diversity

Formal Methods

- Idea came from traditional engineering disciplines
- Specify and model behavior of a system, and mathematically verify that its design and implementation satisfy functional and safety reqs.
  - L1: formally specify the system (using logic or spec language)
  - L2: spec at 2+ levels; pencil-and-paper proof that concrete levels imply the more abstract levels (e.g., implem. → design)
  - L3: spec and then convince mechanical theorem prover
- Problems:
  - Specs and mechanical prover must be 100% correct
  - Large, complex systems are impossible to verify

Example: Proof-Carrying Code

- Code producer generates formal safety proof (e.g., first-order logic proof for DEC Alpha machine code)
- Code consumer verifies validity of proof with a fast checker
- Advantages:
  - Shifts burden of proof to code producer
  - Only need to trust proof checker
  - Faster than an interpreter
  - PCC maintains safety, even if tampered with
- Disadvantages:
  - Must trust proof checker and safety policy
  - Hard to prove interesting properties automatically
  - Safety is not sufficient for dependability

Static Program Analysis

- Inspect source code, manually or automatically, and find potential bugs (e.g., compilers)
- Example: metacompilation
  - Programmer provides C-like gcc extensions to automatically check or optimize their code; get compiled together w/ src
  - Extensions express accepted rules:

    Syscall must check user pointers before using them
    Don't call blocking function with interrupts disabled
    Disabled interrupts must eventually be re-enabled
  - Found couple thousand bugs in Linux, OpenBSD, and Xok
  - Latest tool: infer these rules automatically, thus not requiring the programmer to write them

Overview

1. Fault Prevention
   - Better Software Engineering
   - Formal Methods
   - Language-Based Mechanisms
   - Fault Forecasting
2. Fault Containment
   - By Design
   - Language-Based Mechanisms
   - Virtualization
3. Fault Tolerance
   - Redundancy
   - Recovery
   - Diversity

Restrictive Languages

- To limit the damage a program can do, limit what can be expressed in the source language (“if you can’t say it, nobody will do it”)
- Typical restrictions
  - Control flow (e.g., no backward branching → finite exec)
  - Type safety, no pointers
- Issues:
  - Language may be too limited and awkward
  - Dependable code must be written in that language
  - Binaries must be tamper-evident
  - Assumes all dev tools are trusted and correct
- Example: SPIN and user-provided kernel extensions written in Modula-3 (type-safe and OO); extensions signed by compiler

---

Overview

1. Fault Prevention
   - Better Software Engineering
   - Formal Methods
   - Language-Based Mechanisms
   - Fault Forecasting
2. Fault Containment
   - By Design
   - Language-Based Mechanisms
   - Virtualization
3. Fault Tolerance
   - Redundancy
   - Recovery
   - Diversity

Fault Forecasting

- Monitor system and infer when it has entered an area of its state space where it’s prone to failure; then fix it
- Internal monitoring: ISTORE
  - Incrementally scaleable, self-maintaining storage appliance
  - Sensors monitor system and communicate changes
  - Software triggers (predicates over system state) get evaluated when something changes and signal potential problems
  - Adaptation code gets invoked to deal with anomalies
- External monitoring: Internet services
  - Statistically model system/network performance behavior
  - A deviation from that model == sign of impending fault

Overview

1. Fault Prevention
   - Better Software Engineering
   - Formal Methods
   - Language-Based Mechanisms
   - Fault Forecasting
2. Fault Containment
   - By Design
   - Language-Based Mechanisms
   - Virtualization
3. Fault Tolerance
   - Redundancy
   - Recovery
   - Diversity

Sandboxing

- Isolate user program in a sandbox where it can execute without harming anything outside the sandbox
- Sandbox = fault domain = code + data segment, suitably aligned
- Configure MMU to fault on accesses/jumps outside of fault domain
- Rewrite the binary to mask off high order bits on addresses to keep them within fault domain
- Redirect system calls through a protected jump table to an arbitrator

---

Dynamic Dataflow Analysis

- Deny potentially unsafe operations
- Quarantine data that may be contaminated (taintperl)

```
print STDERR “Enter file name:”;
$x = <STDIN>;          # tainted
….
$y = “/etc/hosts”;    # clean
$z = “$sysdir/$x”;    # transitively tainted
system(“cat $z”);     # not permitted
system(“cat $y”);     # OK
```

Overview

1. Fault Prevention
   - Better Software Engineering
   - Formal Methods
   - Language-Based Mechanisms
   - Fault Forecasting
2. Fault Containment
   - By Design
   - Language-Based Mechanisms
   - Virtualization
3. Fault Tolerance
   - Redundancy
   - Recovery
   - Diversity

Virtualization

- VM = sw abstraction of a machine on top of another machine
- Can virtualize hw or a language exec environment
- Advantages:
  - Guest can virtualize resources differently from host
  - Excellent way to test and debug (incl. with altered privileges)
  - Run distinct versions of various software, for diversification
  - Create sophisticated sanboxes (e.g., for classified information processing); VM isolation is simple to understand
  - Intercept, control, and monitor access to all resources; could infer when application is about to fail
- Problems:
  - Must rely on integrity and correctness of VM

Virtualization Examples

- 1967: IBM introduces the S/360 model 67 w/ virtual memory; TSS subsystem provides the illusion of multiple 360's w/out virtual memory
- These days: S/390 can run Linux in 10,000s of concurrent VMs (1,000s on production systems)
- IBM's latest offering: z/VM for the z900 mainframe
- Other:
  - VMWare's x86 VM for Windows and Linux
  - RealPC and VirtualPC to simulate x86 on Macs
  - of course... JVM

Overview

1. Fault Prevention
   - Better Software Engineering
   - Formal Methods
   - Language-Based Mechanisms
   - Fault Forecasting
2. Fault Containment
   - By Design
   - Language-Based Mechanisms
   - Virtualization
3. Fault Tolerance
   - Redundancy
   - Recovery
   - Diversity

Information/Data Redundancy

- CRC, EDC/ECC codes, parity checks, etc.
- Replication
  - Primary/Secondary copy
  - Multi-node replication
  - Majority voting
  - Weighted voting
  - Geographical replication: ship transaction log off-site

Processor Redundancy

- Simple example: Triple Module Redundancy
  - Decreased MTTF; so why is it a good idea?
  - Bonus: single point of failure to something simple (a voter)
- Hot/Cold standy and failover (e.g., Tandem Non-Stop)
- Clusters (combines data with processor redundancy)
- Distributed systems problems:
  - Manageability
  - How to reach agreement ?

Distributed Consensus

- Nodes can fail (stop or Byzantine); good nodes need to agree on a value (e.g., time of transaction commit)
- Most famous anthropomorphism in distributed systems: Byzantine Generals problem (Lamport)
  - City surrounded by Byzantine army; attack or retreat? When? Must do it at the same time!
  - Traitorous generals want to deceive loyal generals
  - Can use oral or written (signed messages)
  - What is the max. number of traitors that can be tolerated?
  - Unsigned messages: can tolerate strictly less than 1/3
  - Signed messages: can tolerate any number of faults

Impossibility of Consensus

- Fundamental result, proved in 1985
- Asynchronous system (can make no assumptions about relative speeds of processes) w/ reliable communication
- At most one process can fail (stop or Byzantine)
- Impossible to guarantee consensus in finite time
- Basic reason: cannot distinguish failed nodes from slow nodes
- Consequence: cannot tolerate any fault in async system
- In real world: place upper bounds on communication and processing time; if slow, consider node faulty

Temporal Redundancy

- Perform computation several times, use comparator to generate result
- In face of failure, repeat computation
  - At the basis of reliable message delivery through retransmit

Overview

1. Fault Prevention
   - Better Software Engineering
   - Formal Methods
   - Language-Based Mechanisms
   - Fault Forecasting
2. Fault Containment
   - By Design
   - Language-Based Mechanisms
   - Virtualization
3. Fault Tolerance
   - Redundancy
   - Recovery
   - Diversity

Recovery

- Backward recovery: return system to previous, known-good state (e.g., checkpoint-restart, rollback, reboot)
- Forward recovery: take system to new, good state from where it can continue operating, potentially in degraded mode (e.g., app-specific exception handling)
- Compare to: *compensation* , in which erroneous state is sufficiently redundant to allow continued operation (e..g, compensating transactions, failover, etc.)

ACID

- **A**tomicity = all-or-nothing
  Consistent = only correct state changes
  Isolated = as if running alone, even if concurrent txs
  Durable = committed changes are permanent
- Transaction = unit of work that is ACID
- Write-ahead logging for atomicity and durability
- Hairy algs. for logging and recovery
- Distributed transactions & recovery management
- Multiphase commits

Pros and Cons of ACID

- Advantages:
  - Atomicity is extremely appealing
  - Unambiguous notion of fail-stop
  - Easy and simply to understand and reason about
  - Excellent building block for complex interactions
- Disadvantages:
  - Consistency protocols are hard to get right (design & impl.)
  - Locking and scheduling; deadlock detection
  - Recovery code – worst kind of code: almost never exercised, but absolutely critical when called
  - Performance and correctness antagonistic

Restart-Based Techniques

- One of the largest sources of unavailability: intermittent bugs and transient faults
- Structure systems such that they can be restarted at various fine grain levels without adverse effects
- Use prophylactic and reactive restarts to cure failure
- Properties:
  - Unequivocally returns system to its start state
  - High confidence way to reclaim stale and leaked resources
  - Easy to understand and employ

Overview

1. Fault Prevention
   - Better Software Engineering
   - Formal Methods
   - Language-Based Mechanisms
   - Fault Forecasting
2. Fault Containment
   - By Design
   - Language-Based Mechanisms
   - Virtualization
3. Fault Tolerance
   - Redundancy
   - Recovery
   - Diversity

Diversity

- N-version programming (govt.-funded critical systems)
- More realistic variant: deploy a collection of different releases from various software vendors, to avoid similar fault patterns
- Automated mutation of software

Overview

1. Fault Prevention
   - Better Software Engineering
   - Formal Methods
   - Language-Based Mechanisms
   - Fault Forecasting
2. Fault Containment
   - By Design
   - Language-Based Mechanisms
   - Virtualization
3. Fault Tolerance
   - Redundancy
   - Recovery
   - Diversity