Dependability Lessons from Internetty Systems: An Overview

Stanford University CS 444A / UC Berkeley CS 294-4
Recovery-Oriented Computing, Autumn 01
Armando Fox, fox@cs.stanford.edu

Concepts Overview

- Trading consistency for availability: Harvest, yield, and the DQ principle; TACT
- Runtime fault containment: virtualization and its uses
- Orthogonal mechanisms: timeouts, end-to-end checks, statistical detection of performance failures
- State management, hard and soft state
- Revealed truths: end-to-end argument (Saltzer), software pitfalls (Leveson), and their application to dependability
- Many, many supplementary readings about these topics

Consistency/Availability Tradeoff: CAP

CAP principle (this formulation due to Brewer):

- In a networked/distributed storage system, you can have any 2 of consistency, high availability, partition resilience.
  - Internet systems favor A and P over C
  - Databases favor C and A over P
  - Surely other examples
- Generalization: can you trade *some* of one for more of another? (hint: yes)

Consistency/Availability: Harvest/Yield

- *Yield:* probability of completing a query
- *Harvest:* (application-specific) fidelity of the answer
  - Fraction of data represented?
  - Precision?
  - Semantic proximity?
- Harvest/yield questions:
  - When can we trade harvest for yield to improve availability?
  - How to measure harvest "threshold" below which response is not useful?
- Application decomposition to improve "degradation tolerance" (and therefore availability)

Generalization: TACT (Yu & Vahdat)

- Model: distributed database using anti-entropy to approach consistency
- "Conit" captures app-specific consistency unit (think: ADU of consistency)
  - Airline reservation: all seats on 1 flight
  - Newsgroup: all articles in 1 group
- Bounds on 3 kinds of inconsistency
  - Numerical error (value is inaccurate)
  - Order error (write(s) may be missing, or arrive out-of-order)
  - Staleness (value may be out-of-date)
- "Consistency cost" of operations can be characterized in terms of conits, and bounds on inconsistency enforced

TACT-like example: TranSend

- Early stab at lossy on-the-fly Web image compression, *extensively parameterized* (per user, device, etc.)
- Harvest: "semantic fidelity" of what you get
  - Worst case: the original image
  - Intermediate case: "close" image that has been previously computed and cached
  - Metrics for semantic fidelity?
- Trade harvest for yield/throughput
- TACT-like, though TACT didn't exist then

Another special case: DQ Principle

- Model: read-mostly database striped across many machines
- Idea: Data/Query x Queries/Sec = Data/Sec
- Goal: design system so that D/Q *or* Q/S are tunable
  - Then you can decide how partial failure affects users
  - In practice, Internet systems constraint is offered load of Q/S, so failures affect D/Q for each user
  - Can use some replication of most common data to mitigate effects of reducing D/Q

Fault Containment

- Uses of software based fault isolation and VM technology
  - Protecting the "real" hardware (now will also be used for ASP's)
  - Hypervisor-based F/T
- Orthogonal mechanisms for fault containment
- …and enforcing your assumptions

Extension: Hypervisor-Based Fault Tolerance

- Basic ideas (Bressoud et al, SOSP-15)
  - Use VM's to implement a *hypervisor* that coordinates between a primary process and its backup
  - Instruction *epochs* are separated by periodic S/W interrupts
  - Hypervisor arranges to deliver interrupts only on epoch boundaries
  - Primary and backup can also communicate during "environmental" instructions (so they see same result of I/O, eg)
  - Backup is one epoch "behind" primary, can take over right away
  - Recently applied to JVM by Lorenzo Alvisi et al. at UT Austin
  - Again, successful virtualization requires some lower-level guarantees
- Important concept: critical events occur at points of possible nondeterminism in instruction stream

Orthogonal Mechanisms

- Bunker mentality: *Design with unexpected failure in mind*
  - Minimize assumptions made of rest of system
  - Keep your own house in order, but be prepared to be shot if outside monitoring sees something wrong
  - Design systems to allow independent failure
- In real life (hardware)
  - Mechanical interlock systems
  - Microprocessor hardware timeouts
- In real life (software)
  - Security and safety
  - Deadlock detection and shootdown

Examples of Orthogonality

- examples of orthogonality
  - Software fault isolation/virtualization
  - IP firewalls
  - Deadlock *detection & recovery* in databases - Note: not deadlock *avoidance!*
  - Hardware orthogonal security - Fuses and hardware interlocks; recall the Therac-25
  - Theme: you don't know *why* something went wrong, only *that* something went wrong; and you can usually do fault containment
- What's appealing about orthogonal mechanisms?
  - Small state space - Behavior simple to predict (usually)
  - Allows us to enforce at least some simple invariants - and invariants are your friends

Example: What Really Happened on Mars

- Dramatis personae
  - Low-priority thread A: infrequent, short-running meteorological data collection, using bus mutex
  - High-priority thread B: bus manager, using bus mutex
  - Medium-priority thread C: long-running communications task (that *doesn't* need the mutex)
- Priority inversion scenario
  - A is scheduled, and grabs bus mutex
  - B is scheduled, and blocks waiting for A to release mutex
  - C is scheduled while B is waiting for mutex
  - C has higher priority than A, so it prevents A from running (and therefore B as well)
  - Watchdog timer notices B hasn't run, concludes something is wrong, reboots

On Enforcing Your Assumptions

- Orthogonal mechanisms can be used to enforce assumptions about system behavior
  - Infer failure of a peer -> shoot it
  - Assume peers will respond within a specific time -> use timeout to force
- Why is this important?
  - Response to a detected condition may be inappropriate if assumptions are incorrect

Enforcing Invariants Made Easier

- Some other possible replies to fopen():
  - "Maybe later" (NFS soft mount failed this time)
  - "How about a stale copy" (AFS server down, cached copy available, freshness questionable)
  - "Took too long, you consumed too many resources, try again later" (like "HTTP server too busy")
- Essence of the "MIT approach" vs "New Jersey approach"
  - Weaken the guarantee/illusion offered by subsystem
  - Force higher-level app to deal with being told "no"
  - Perhaps wrappers or other mechanisms will be developed to simplify this
  - Makes system more robust: simpler --> easier to understand, plus instills "bunker mentality" in (good) programmers

Soft State

- Soft state and announce/listen
- Soft state and its relation to robustness
- An example of using soft state for managing partial failures

Loose coupling with soft state

- Announce/listen + soft state, vs. hard state
  - "sender" continually sends state messages to "receiver", who may or may not reply/ack
  - If receiver "forgets" state, just wait for next message
  - Example: setting a variable on the server
- Assumptions & challenges
  - Assumption: messages may get lost, receiver may be down, etc.
  - Messages must be idempotent (this is a big one)
  - May not work for real-time-constrained activities
    - Or may require hysteresis to avoid oscillation

Uses of Soft State

- Wide-area Internet protocols, esp. multicast routing
- Scalable Reliable Multicast (SRM)
  - Members of a group session each have soft copies of group state
  - State "repairs" are multicast
  - New members can ask for "fast replay" to catch up
- Related concept: *expiration-based schemes*
  - Web caching: expiration bounds staleness
  - Leases: expiration bounds unavailability of a locked resource due to node failure

Soft State Pros and Cons

+ No special recovery code
  - Recovery == restart
  - "Recovery" paths exercised during normal operation
+ Leaving the group (or dying) doesn't require special code
  - Timeouts/expirations mean no garbage collection
  - "Passive" entities will not consume system resources

– Staleness or other imprecision in answer

– Possible lack of atomicity/ordering of state updates

Next: Exploiting soft state/approximate values for failure masking…

Software Pitfalls (Leveson, Ch. 2)

- Unlike hardware, software is purely abstract design
  - Unlike hardware, software doesn't "fail"
  - Design unconstrained by physical laws
  - Cost of modifying design hidden by lack of physical constraints
  - *Real* state space includes what the hardware and other software are doing, and includes states that don't correspond to states in the abstract model
- Software as a discrete (vs. analog) state system
  - Effect of small perturbation on an analog system vs. on a software discrete system
  - Software can "fail" in a way that is completely unrelated to how the environment/inputs were perturbed

Software Pitfalls, cont'd.

- Exploding some common myths
  - "Reuse of software increases safety" - or, it perpetuates the same (hidden) bugs (e.g. Therac-25)
  - "Formal verification can remove all software-related errors" - unless software/system fails in a way that is *outside* its design point (e.g. overload/thrashing)
  - "GP computers + software are more cost effective [compared to a dedicated purpose-designed system]" - Space Shuttle software costs ~$100M/year to maintain

Revealed Truth: End-to-End Argument

- Don't put functionality in a lower layer if you'll just have to repeat it in a higher layer anyway.
  - If you do put it in a lower layer, make sure it's only a performance optimization.
- Jim Gray: "you can't trust anything"
  - Silent HW failures/incorrectness not that uncommon
  - Examples: ECC memory, disk controllers, OS VM system, etc.
  - Still need end-to-end app-level checks (e.g. Exchange Server)
  - …Is this ominous news for the VM approach?
- To what extent do various fault-masking mechanisms violate (or reinforce) the end-to-end argument?
  - e.g. "transparent" checkpointing and restarting

E2E and Dependability: Some Thoughts

- Use both end-to-end and component-level checks
  - e.g. cross-check answer to a simple query
- Use cross-checks to validate/enforce assumptions about e2e or component-level checks
  - observation: TImeout occurred communicating with component X
  - -> hypothesis: component X is wedged and must be restarted
  - cross-check: look for other signs consistent with X being wedged (*ps, rusage,* etc)
  - enforcement: shoot X, then verify it's gone from process table
- Challenge: requires knowledge of end-to-end semantics
  - Most "classical F/T" approaches *don't* take this tack.

Other readings (will be on Web)

- Virtualization and sandboxing: other readings
  - Hypervisor-Based Fault Tolerance (Bressoud et al., SOSP-15)
  - Disco: Running Commodity OS's on Scalable Multiprocessors (Bugnion et al., ACM TOCS 15(4) and SOSP-16)
  - Recursive Virtual Machines in Fluke (Ford et al., OSDI 96)
  - Efficient Software-Based Fault Isolation (Lucco & Wahbe, 1993)
  - JANUS: an environment for running untrusted helper apps (Goldberg et al., USENIX 1996 Security Conference)
- Harvest, yield, consistency, availability
  - Harvest, Yield, and Scalable Tolerant Systems (Fox & Brewer, HotOS-VII)
  - TACT (Yu & Vahdat, SOSP-18 and others)

Other readings

- Reliability in cluster based servers
  - Lessons From Giant-Scale Services (Brewer, IEEE Internet Computing; draft on Web)
  - Cluster-based Scalable Network Services (Fox, Brewer et al, SOSP-16)

Putting It All Together

Berkeley SNS/TACC: an application-level example of several of these techniques in action:

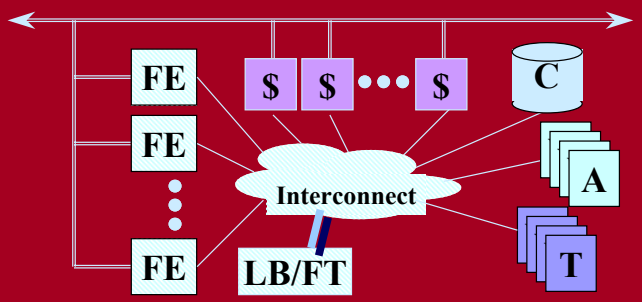
- Supervisor-based redundancy for both availability and performance
- Loose coupling and announce/listen to circumvent SPF for supervisor
- Orthogonal mechanisms to account for legacy code vagaries
- Normal-operation and failure-recovery code paths are the same

TACC/SNS

- Specialized cluster runtime to host Web-like workloads
  - TACC: transformation, aggregation, caching and customization--elements of an Internet service
  - Build apps from composable modules, Unix-pipeline-style
- Goal: complete separation of **ility* concerns from application logic
  - Legacy code encapsulation, multiple language support
  - Insulate programmers from nasty engineering

"Starfish" Availability: LB Death

- FE detects via broken pipe/timeout, restarts LB

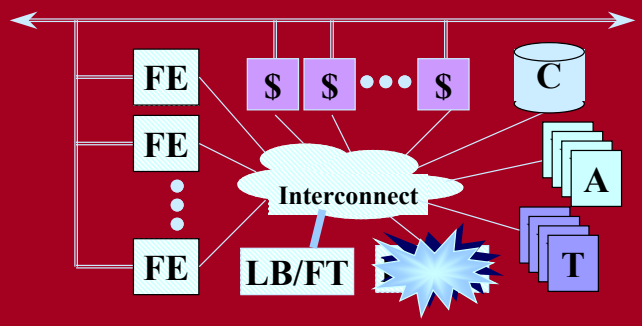


"Starfish" Availability: LB Death

- FE detects via broken pipe/timeout, restarts LB

New LB announces itself (multicast), contacted by workers, gradually rebuilds load tables

If partition heals, extra LB's commit suicide
FE's operate using cached LB info during failure

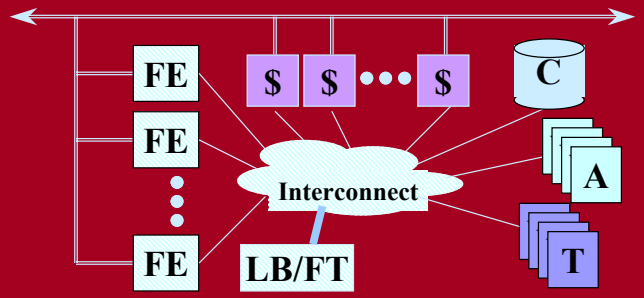


"Starfish" Availability: LB Death

- FE detects via broken pipe/timeout, restarts LB

New LB announces itself (multicast), contacted by workers, gradually rebuilds load tables

If partition heals, extra LB's commit suicide
FE's operate using cached LB info during failure



SNS Availability Mechanisms

- Soft state everywhere
  - Multicast based announce/listen to refresh the state
  - Idea stolen from multicast routing in the Internet!
- Process peers watch each other
  - Because of no hard state, "recovery" == "restart"
  - Because of multicast level of indirection, don't need a location directory for resources
  - Timeouts and restarts everywhere
- Load balancing, hot updates, migration are "easy"
  - Shoot down a worker, and it will recover
  - Upgrade == install new software, shoot down old
  - Mostly graceful degradation