RECOVERY-ORIENTED COMPUTING

Crash-Only Software

George Candea and Armando Fox

Stanford University

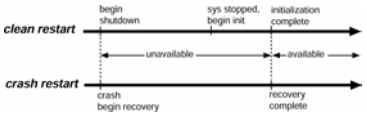
Motivation

- Software bugs
  - a main source of unplanned downtime*
  - most are intermittent/transient
- Fine-grain reboot:
  - easy
  - effective
  - *more or less* predictable
- Need high confidence, simple, well-defined failure semantics

*[*Adams '84*], [*Gray '86*], [*Murphy '95*], [*Chou '97*], [*Murphy '99*], [*Gartner '99*], [*Gartner '01*]

Potpourri of Restarts

- Impractical to guarantee zero crashes → programs must be crash-safe anyway
- Occam's Razor → why have more than one type?
  - Performance (no synch writes in FS → need to flush buffer cache)
- You want fast systems or HA systems ???
  - Performance quest → frail systems
  - Leave performance improvements to Moore's Law
- Crashes are sometimes faster, modulo data loss (WinXP crash reboots for upgrades)



Crash-only software must:
(a) be crash-safe & (b) recover quickly

What Is Crash-Only SW ?

- Crash-only component has PWR switch: stop=crash
  - clean shutdown
  - loss of power
  - kernel panic
  - cure transient failure
- Only one way to go down → only one way to come up: start = recover
- Each component must has a PWR switch → uniform behavior
- Crash-only system = assembly of crash-only components; system PWR switch implemented in terms of components' switches
- PWR switch is external, does not invoke component code, just like
  - `kill -9` for a UNIX process
  - turning off the VM in which a subsystem is running
  - pulling a cluster node's power cable out of the wall

Outline

- Overview
- Requirements for Crash-Only Internet Systems
- Benefits of Crash-Only Designs

What Do We Call Internet Systems ?

- Large scale + HA requirements
- Heterogeneous, individually packaged components (web servers, application servers, databases, etc.)
- Rapid and perpetual evolution → difficult to build and maintain consistent model (key difference from other mission-critical apps)
- Workload = large numbers of relatively short tasks, rather than long-running operations
- Request-reply protocols (e.g., web browsers talking HTTP)
- Single installs (one data center), no WAN
- Prescriptive (CS) vs. descriptive (Physics) laws

Concrete Requirements

- Crash-only = crash-safety + fast recovery
- Intra-component state management:
  1. Persistent state is managed by dedicated stores
  2. State stores are crash-only
  3. Abstractions provided by state store match app's requirements
- Extra-component interactions:
  1. Components are modules with externally enforced boundaries
  2. Timeout-based communication and lease-based resource allocation
  3. TTL and idempotency information carried in requests

1. Persistent State Managed by State Stores

- State management ≠ application logic
- What is application state?
  - application state (user data, control structures, etc.)
  - resources (file descriptors, kernel data structures, etc.)
- Persistent application state lives in dedicated crash-only state stores (DB, NetApp filer, middle-tier persistence layer, etc.)
- Apps become free of persistent state ("stateless")
- Example: three-tier Internet architectures
- Benefit: simpler recovery code for app; state store can be clever about transitioning from one consistent state to another

2. State Stores Are Crash-Only

- Don't push problem one level down
- COTS crash-safe state stores: DB's, NASD's
- Tunable COTS state stores: Oracle DB
- True crash-only state stores: Postgres
  - No WAL, one append-only log
  - Almost instantaneous recovery: mark in-progress txn's failed

3. State Store Abs == App Abstractions

- Persistent state is in stores, so store needs API; all ops on persistent state done through high-level API
- State store abstractions == app-desired abstractions; app should operate on state at its own semantic level
- Example: store customer records in DB, not file system
- Benefit: state store can exploit app semantics and workload characteristics to offer performance and fast recovery
- Berkeley DB: 4 abstractions, 4 APIs

Trend toward Standardization

- Few, specialized state stores:
  - Transactional ACID (customer data → DB)
  - Simple read-only (static web pages & GIFs → NetApp)
  - Non-durable single-access (user session state)
  - Soft state store (web cache)

1. Strong Fault Containment Boundaries

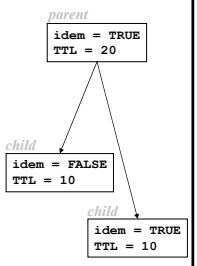
- Components = Modules with externally enforced boundaries
- Isolation achieved using
  - Virtual machines (e.g., VMware)
  - Isolation kernels (e.g., Denali)
  - Java tasks (e.g., Sun's MVM)
  - OS processes
- Example: Ensim and other web service hosting providers
- Staged processing, isolated stages (e.g., HTTP request)

2. Timeouts and Leases

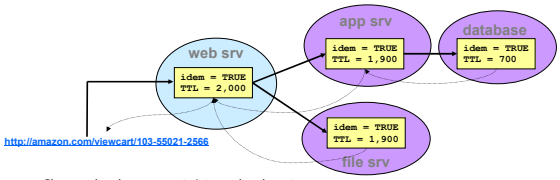
- All communication (RPC or messages) has timeouts → fail-fast behavior for non-Byzantine faults
- Everything is leased, never permanently bound → reduce coupling (persistent state + resources)
- Maximum timeout specified in app-global policy
- Benefit: system never gets stuck

3. TTLs and Idempotency Flag in Requests

- Every request traveling through system carries a context that includes:
  - `idem`: is operation idempotent ?
  - `time-to-live`, after which invoker will assume request has failed (TTL updated at each stage)
- Many interesting requests are idempotent or can be easily be made idempotent (sequence #'s, txn's)
- Benefits
  - sub-request failures are "atomic"
  - request stream is restartable/recoverable



A Restart/Retry Architecture



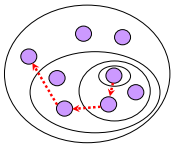
- Glues crash-only components into crash-only systems
- Components that aren't making satisfactory progress or fail get crash-restarted
  - Based on timeouts and/or progress counters = compact representation of progress (e.g., HTTP reply stage)
  - Counters live behind state store and messaging APIs; map state access/messaging activity into per-component progress
  - Components can implement counters capturing app semantics, but are less trustworthy
- Requestor stub infers failure from timeout or *RetryAfter(n)* exception
- Component restart = transient failure, caller resubmits idempotent requests if enough time left → request stream recovers transparently
- Worst case: propagate failure or HTTP/1.1 `Retry-After` to the client

Benefits

- Fewer undefined states
- Robust recovery: exercising recovery code on every startup
  - KLOC/system up faster than bugs/KLOC down → more bugs, software will fail more often, hence need to recover more often
- Transparent sub-system recovery → continuity of service
- Can coerce all non-Byzantine failures into a crash → simple crash-based fault model → easier to write correct recovery code
- Software rejuvenation = preemptive reboot to stave of failure (most effective when using crash, because clean shutdown might not release all resources)
- Trivial migration of tasks (failover, load balancing, reconfiguration) = crash on one node, recovery on the other
- Zero-downtime partial system upgrades = crash old component, recover new one

Recursive Restarts for HA

- We have crash-only components – now what?
- Reduce recovery time by doing partial restarts: attempt recovery of a minimal subset of components
- What if restart ineffective? recover progressively larger subsets
- Chase fault through successive boundaries



- Demonstrated 4x improvement in recovery time on Mercury (stateless, crash-safe satellite ground station)

Ongoing and Future Work

- Crash-only software: one way to go down, one way to come up
- To study:
  - emergent properties
  - not all operations are idempotent (needed for "execute at least once")
  - Limited to request/reply systems (e.g., interactive desktop apps might not work)
- Implement on open-source J2EE app srv – RR-JBoss crash-only
  - Separate J2EE services into separate components
  - Associate contexts with each request
  - Timeout-based RMI (Ninja?) and lease-based allocation
- Crash-only app: ECperf
- Automatic recursive restarts based on f-maps

More…

http://RR.stanford.edu