RECOVERY-ORIENTED COMPUTING

Automatic Failure-Path Inference

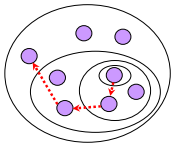
George Candea, Mauricio Delgado, Michael Chen,

Fang Sun, Armando Fox, Pedram Keyani

Stanford University

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-proof satellite ground station)
- How do we navigate the fault boundaries? ...

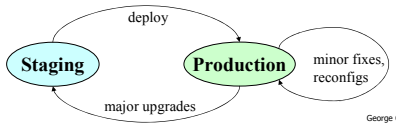
Fault Dependency Graph

- Use a graph that depicts how faults propagate in the system (f-map)
- Challenges:
  1. Problem-determination literature assumes graph is magically available
  2. Internet systems evolve rapidly → hard to keep sys and graph in sync
  3. Many failures result from idiosyncratic system/environment interactions, which can't be guessed just by looking at the app
- Desired process properties:
  - don't use explicit model
  - application generic/independent
  - automatic
  - dynamic

Automatic Failure-Path Inference

- Look at what people do: train by placing themselves in unexpected situations; self-managing systems should do the same → introspection

1. Staging phase (active/invasive):
   - inject faults
   - observe system's reaction
   - add inferred propagation paths to global failure propagation map
2. Production phase (passive/orthogonal):
   - observe system's reaction to "naturally occurring" faults
   - augment failure propagation map

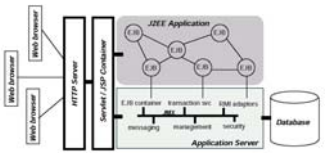


Staging Phase Algorithm

1. Bring system up (infrastructure and application)
2. Each deployment of a component → inspect its interface and infer possible application-visible faults; place potential faults in a global fault list
3. Add environment-related faults (e.g., network partitions, disk I/O faults, out-of-memory)
4. Iterate through list of (component *C*, method *M*, fault *F*) and schedule fault *F* to be raised by *C* on invocation of *M*
5. Generate workload externally to exercise system
6. As components fail, build f-map = directed graph of edges (u, v) indicating that a fault in component u propagated and caused component v to fail (if v handles fault, then no edge)
7. Save f-map and fault list to stable storage, restart app, continue with the next (C,M,F) triplet

- Injection ends when entire list of faults has been exhausted
- Multi-point injections (truly independent faults are seldom in reality):
  1. Take cross product of list of faults with itself and obtain (C1, M1, F1, C2, M2, F2)
  2. Eliminate tuples that have C1=C2
  3. Iterate through list and inject faults
  4. Add previously unseen paths to f-map

Internet Systems / J2EE



- J2EE enterprise apps = collection of reusable Java modules
- JSPs / servlets invoke EJBs, which invoke other EJBs, ...
- EJB = Java component that complies to a certain interface and provides a service
- Deployment descriptor (XML file) conveys run-time characteristics and dependencies; used in deploying the application
- App srv = operating system for Internet applications (instantiates app components in containers, provides runtime system services, integrates with web server to make app web-accessible)
- We use JBoss (open-source J2EE app srv) = microkernel with components held together through JMX

- Large scale + HA requirements
- Heterogeneous, individually packaged components (web servers, application servers, databases, etc.)
- Rapid and perpetual evolution → impossible 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
- Clients are web browsers talking HTTP

Modifications (JBoss → RR-JBoss)

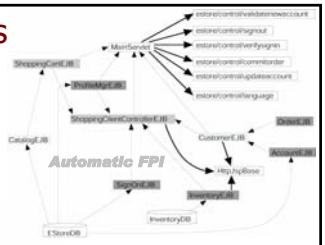
1. Include 2 new JMX services for injection and monitoring: *FaultInjector* and *FailureMonitor*
2. Add hook: whenever a new EJB is deployed, *FaultInjector* is invoked, to reflect EJB interface and populate list w/ exceptions
3. Modify generic EJB container to provide method for scheduling a fault
4. Modify EJB container's log interceptor to capture stack trace when exception is thrown, parse it, and inform *FailureMonitor*

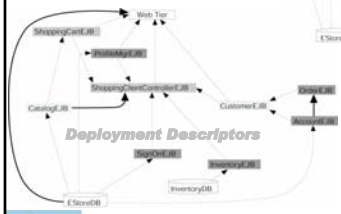
Experiments

- PetStore 1.1.2
  - freely available J2EE "tutorial application" from Sun
  - simulates e-commerce site w/ user accounts, profiles, payments, merchandise catalog, shopping cart, purchases, etc.
- Derive vanilla f-map from deployment descriptors
- Chose to inject Java exceptions = high level, JVM-visible faults
  - low-level bit flips → nondeterministic behavior
  - most manifest low-level problems turn into Java exceptions
- Two types of exceptions:
  - "expected" : declared in bean interfaces
  - "environmental" : resulting from runtime issues (OutOfMemoryError, StackOverflowError, IOException, RemoteException, SQLException)

Comparing f-maps

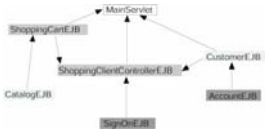
- Are our f-maps at least as good as those obtained by other means?
- If yes, are they better ?





- Missing edges:
  - AccountEJB → OrderEJB: maintained reference, but never used it
  - CatalogEJB → ShoppingClientCtlEJB: didn't even have reference
  - EStoreDB → web tier: only exercised at DB population time
- Additional nodes + edges:
  - HttpJspBase, MainServlet, 6 JSPs: higher resolution, dissected web tier

Fault-Specific f-maps



- Zoom in on dependencies resulting from a specific fault or class of faults
- Targeted recovery when we know the fault that occurred
- f-map obtained by injecting exclusively app-declared exceptions
  - reflects what happens when we isolate it from the environment
- Much simpler (thus more useful) f-map
  - some components missing (ProfileManagerEJB, OrderEJB, InventoryEJB) so no propagation through them

Discussion

- AFPI required no application knowledge
- No performance overhead (we're faster, but that's noise: 94.8 sec vanilla JBoss vs. 93.0 sec RR-JBoss, with 5.8 std. dev.)
- Deployment descriptors can be incorrect; even if correct, will capture paths that might manifest, not only the ones that do manifest
- Use a true call graph tool ? PetStore has 233 Java files w/ 11 KLOC; descriptors are 16 files with 1.5 Klines of XML
- Call graph:
  - might manifest vs. do manifest
  - misses paths that are not due to calls (e.g., memory-gobbling thread)
  - static call graph → need to regenerate every time you change app
  - requires access to source code

Summary

- Automatic Failure-Propagation Inference:
  - \+ automatically and dynamically generates f-maps with no performance overhead
  - \+ no application knowledge required
  - \+ finds dependencies that other analyses might miss, omits dependencies that don't manifest
  - \+ accommodates app evolution
  - \+ obtain high-resolution per-fault-type graphs
  - \- staging phase may take a long time

Future Work

- Make RR-JBoss crash-only
  - Separate J2EE services into separate components
  - Include J2EE services in f-maps
- More complex apps: ECperf (alternately Trade-2, TPC-W, Nile)
- Automatic recursive restarts based on f-maps

More…

http://RR.stanford.edu