RECOVERY-ORIENTED COMPUTING

On the Way to ROC-2
(JAGR: JBoss + App-Generic Recovery)

George Candea and many others…

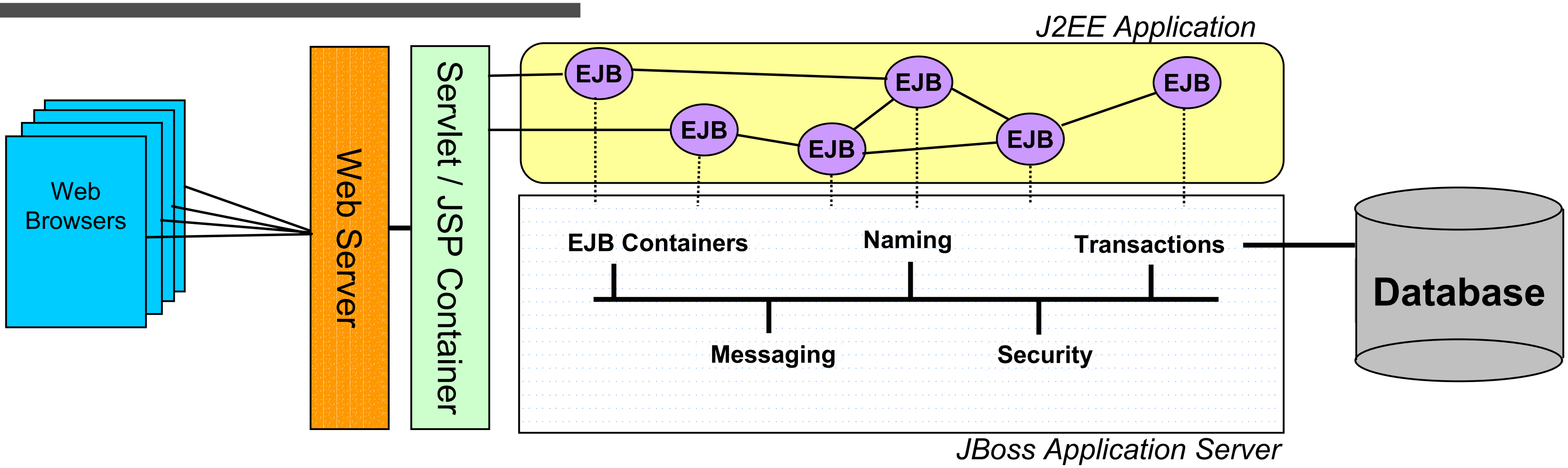
Stanford + Berkeley

Outline

- J2EE and JBoss
- JBoss Additions → JAGR
- µReboots
- Automatic Failure-Path Inference
- Self-Recovery Results
- Discussion

© 2003 George Candea

JBoss

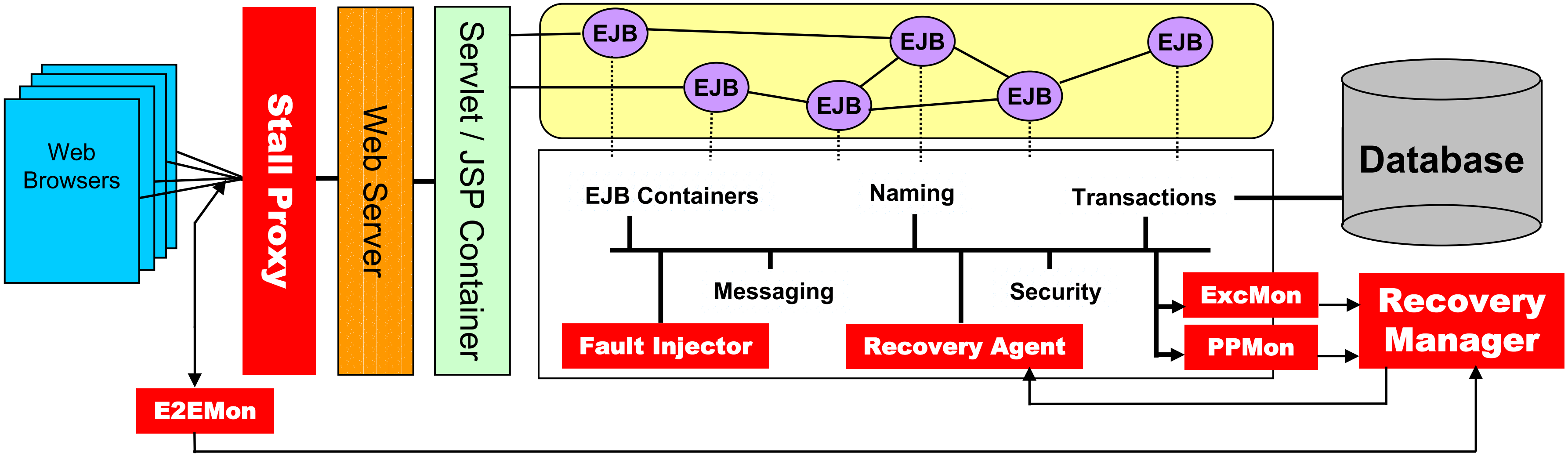


- J2EE enterprise apps = collection of reusable Java modules
- JSPs / servlets invoke EJBs, which invoke other EJBs, ...
- App server = OS for Internet applications (instantiates EJBs, provides runtime svc's, …)

- JBoss = open-source, written entirely in Java, microkernel w/ JMX components
- Downloaded >3.5 million times, JavaWorld '02 Editors' Choice
- Used by >100 large corporations (Dow Jones, WorldCom, etc.)

© 2003 George Candea

JAGR: A Self-Recovering App Server



© 2003 George Candea

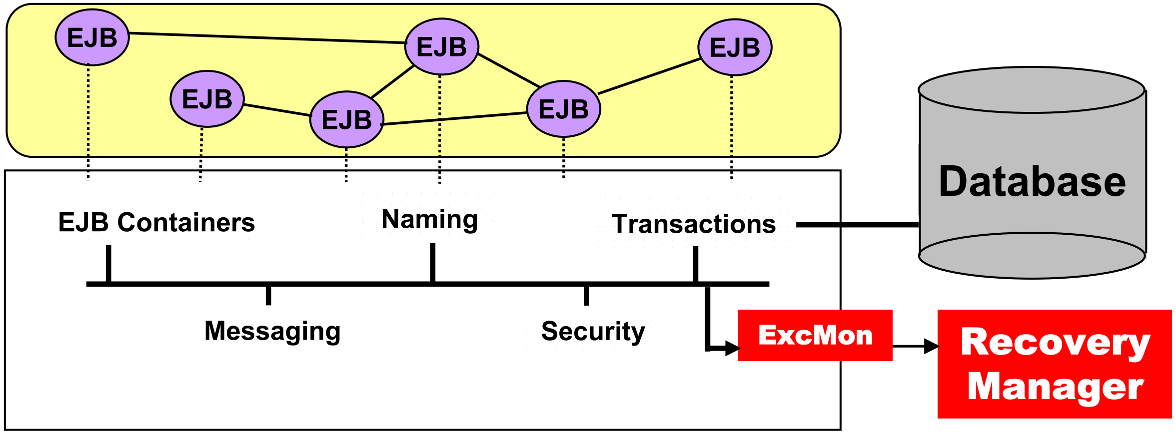
µReboots: Reducing Downtime

Restarted Unit	Duration	Fraction
Reboot Linux server + JAGR + Petstore	357 sec	100.0%
Restart JAGR + Petstore	47 sec	13.2%
Restart Petstore	9 sec	2.5%
µReboot EJB	<1 sec	0.2%

- Surgical reboots were all safe (Linux/ext3fs helps…)
- Various forms of µReboot widely used:
  - Transaction level: deadlock resolution in DBs
  - Process level: web server rejuvenation in Internet portals
  - JVM level: app rejuvenation in enterprise apps
- Finer grain µReboot → less downtime (3 orders of magnitude)
  - Fine-grained workload leads to less lost work

© 2003 George Candea

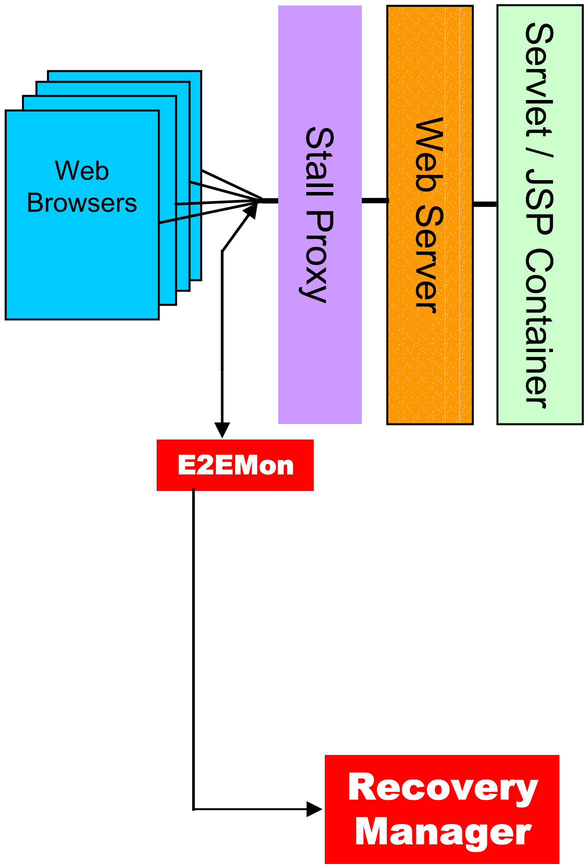
Failure Monitoring: ExcMon



- Instrumented EJB container
- Watch for Java exceptions:
  - Intercept exception in container
  - Parse exception stack
  - Send information to RecoMgr
  - Re-throw exception
- Uses “aspect-oriented programming” feature in JBoss
- Types of failures (fine grained)
  - Unexpected (e.g., runtime OutOfMemoryError)
  - Expected (e.g., app-level EstoreEventException, I/O exceptions)
- Problem: not all exceptions are failures !

© 2003 George Candea

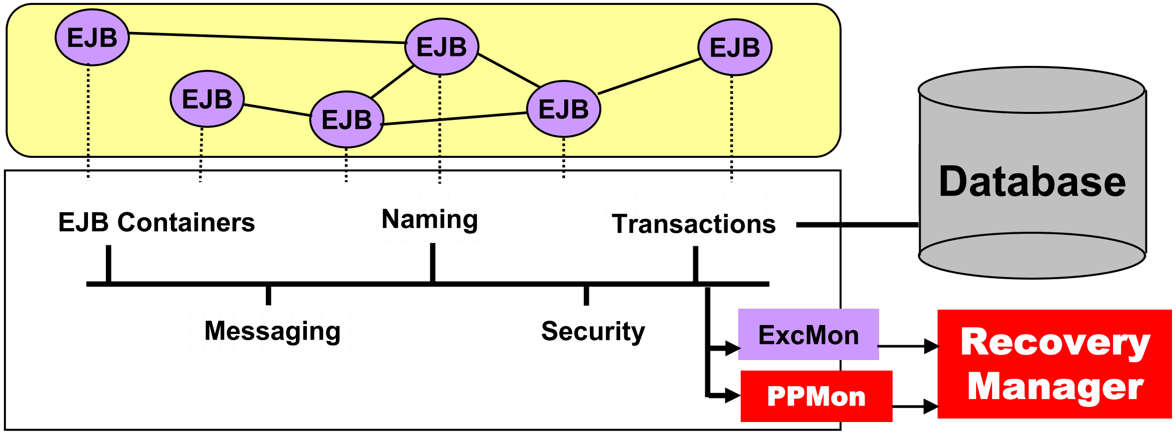
Failure Monitoring: E2EMon



- Simulate a real client:
  - Replay HTTP requests from an application-specific trace file (e.g., browse, mock purchases, user profiles)
  - Check responses
- Types of failures (coarse grained):
  - Network level (e.g., read timeout, connection refused)
  - HTTP level (e.g., HTTP 500 “internal srv error”)
  - HTML level (e.g., empty page, keywords)
- Problem: not all faults result in this kind of failures (e.g., performance degradation)

© 2003 George Candea

Failure Monitoring: PPMon



- Based on Pinpoint (PP)
- Tag client requests and record behavior
- Statistical tech’s + data mining → analyze req’s and capture aggregate behavior
- Compare current behavior to historically-observed “good” behavior → report anomalies
- Failure types:
  - Masked faults (e.g., post-failover, inventory)
  - Fail-stutter behavior
- Problem: Not all anomalies are failures !

© 2003 George Candea

Automatic Failure-Path Inference

- Need a recovery map → dependency graph for application
- How does a fault propagate through system?
- Current options:
  - Reason/construct manually → prone to human failure
  - Static analysis → a priori model does not evolve with app
- Needs to be application-generic !
- Approach:
  - Inject faults
  - Observe system behavior using existing infrastructure
  - Build recovery map
- Two phases:
  1. Pre-deployment: invasive (inject + observe injected faults)
  2. Post-deployment: passive (observe naturally-occurring faults)

© 2003 George Candea

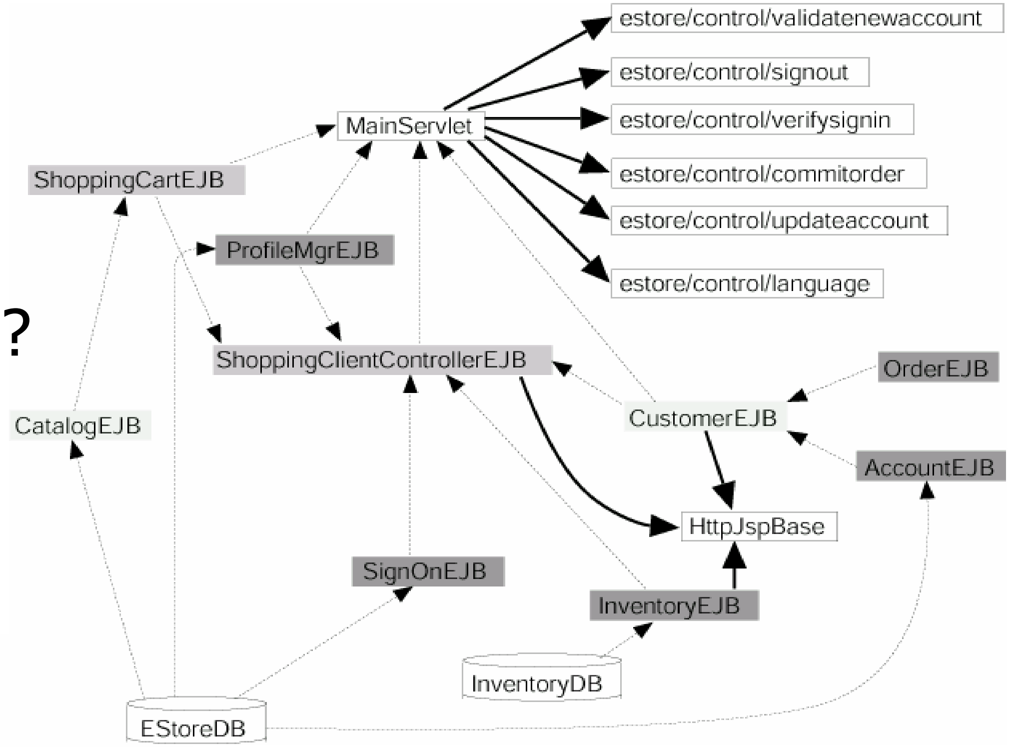
Application-Generic: 2 Different Apps

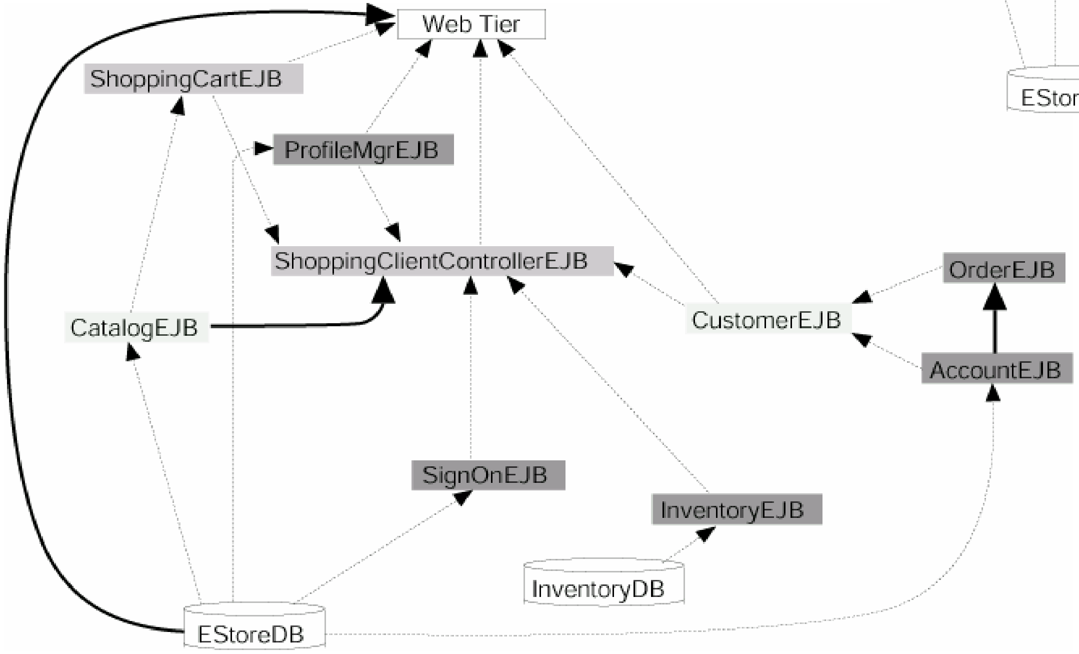
- Petstore
  - Sample J2EE application from Sun
  - E-commerce site: product catalogs, personalization, shopping carts, purchases, shipping, user profiles, etc.
  - 233 Java files, 11 Klines of effective code, 14 DB tables
- RUBiS
  - EBay-like online auction (J2EE): user accounts, customized summary information, item bidding, trustworthiness tracking, etc.
  - 582 Java files, 26 Klines of effective code, 7 DB tables

© 2003 George Candea

Petstore Maps

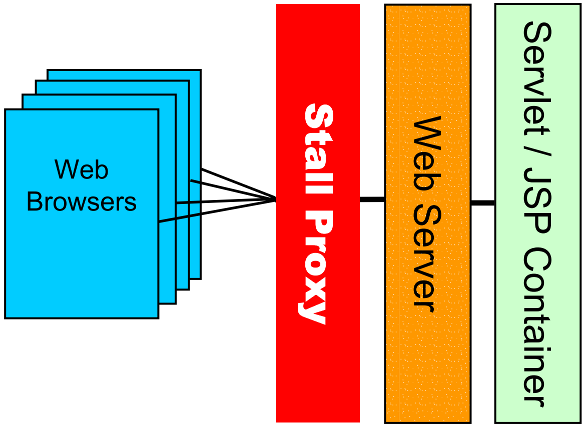
- What does an f-map tell us?
- How do we use a map?





© 2003 George Candea

Stall Proxy



- While recovering → prevent new requests from entering system
- We lose in-transit requests… OK, if recovery is quick
- Fast recovery → can delay requests, instead of turning them away
- Sliding 8-second window
  - after which send back HTTP **`Retry-After`**
  - or have user read and agree to your "new" privacy policy…
- HCI research: <1 sec is fast, >8 sec is distractive

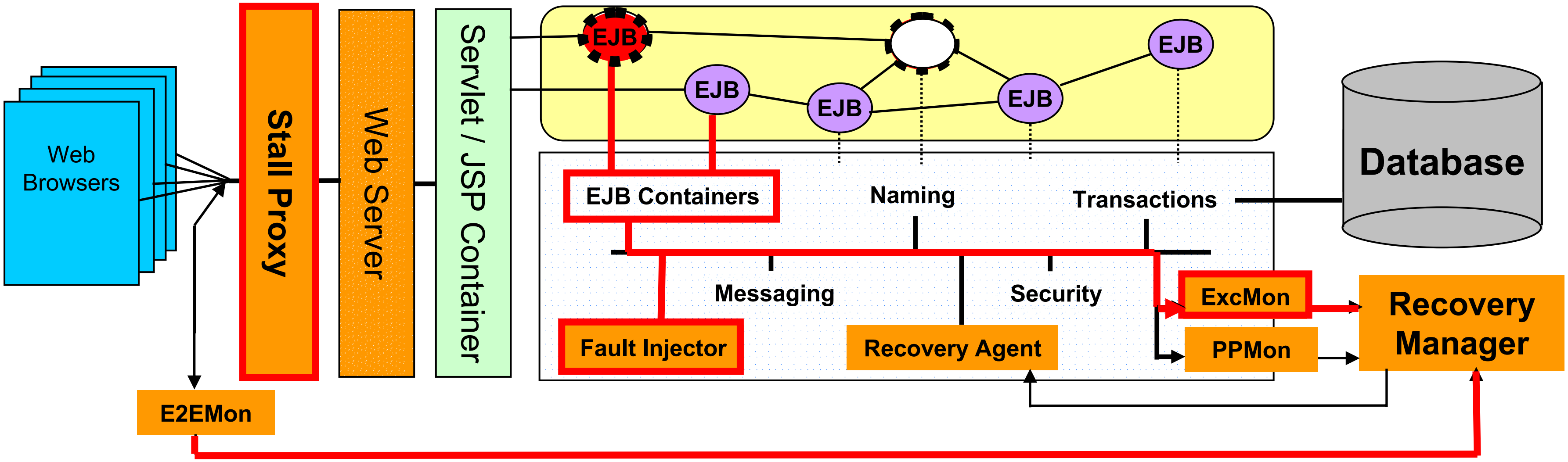
© 2003 George Candea

Outline

- J2EE and JBoss
- JBoss Additions → JAGR
- µReboots
- Automatic Failure-Path Inference
- Self-Recovery Results
- Discussion

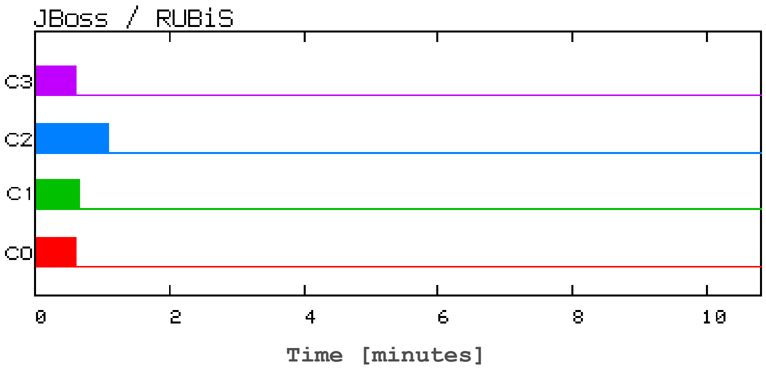
© 2003 George Candea

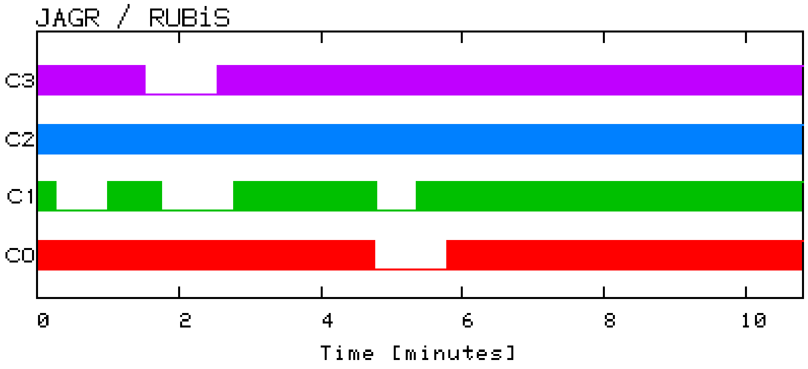
AFPI and Recovery Walkthrough



© 2003 George Candea

Recovering from RUBiS Deadlocks

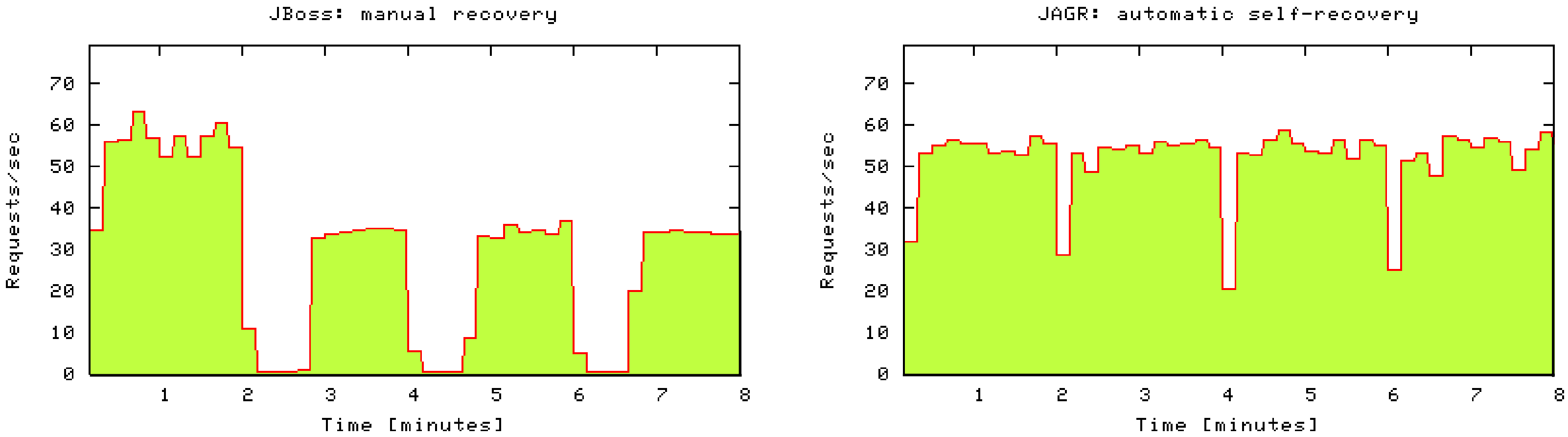




- With a particular workload, can make RUBiS deadlock
- Thick line=up, thin line=down
- Eventually, all clients see the server down
  - JAGR recovers by itself
  - Illusion of continuous uptime for C2 (fast self-recovery masks downtime from end users)

© 2003 George Candea

Availability Improvements



- End user view of service; plot # of successful reqs, averaged over 10-sec intervals
- 3 faults total, injected one every 2 minutes
- JBoss (prompt restart) vs. JAGR (self-µReboot of all EJBs)
- JAGR goodput >= 20 req/sec due to fast recovery + stall proxy
- JBoss: 14,243 requests (green area under the curve)
  JAGR: 25,295 requests
- i.e., 78% improvement in the number of successfully-served requests
  → 78% improvement in availability (actually performability)

© 2003 George Candea

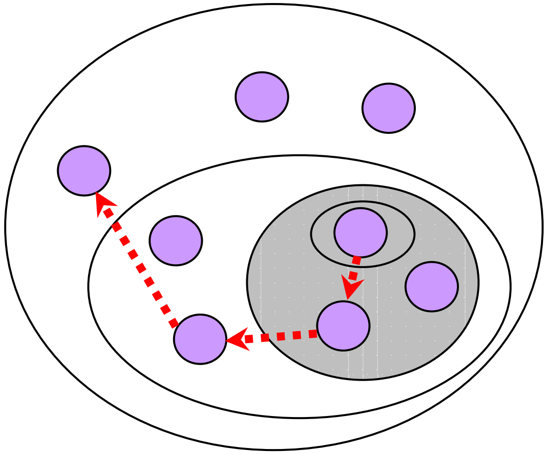
Discussion

- Why care, if we have redundancy + failover
  - Orthogonal: reducing recovery time of individual nodes is important
  - CNN.com cluster on Sep. 11, 2001
    8:46 **84K** req/sec (AAL 11 hits Tower 1)
    8:55 **129K** req/sec
    9:00 **229K** req/sec (doubling every 5 minutes !)
    9:03 Servers start thrashing; sysadmins unable to ssh into cluster
    11:15 CNN.com goes down
    11:30 HTML service restored, no images
    16:15 **1,110K** req/sec
    - slow recovery and lack of self-recovery lead to collapse
- No a priori models → no "expected" failure modes → robustness

© 2003 George Candea

Looking ahead: Recursive µReboots

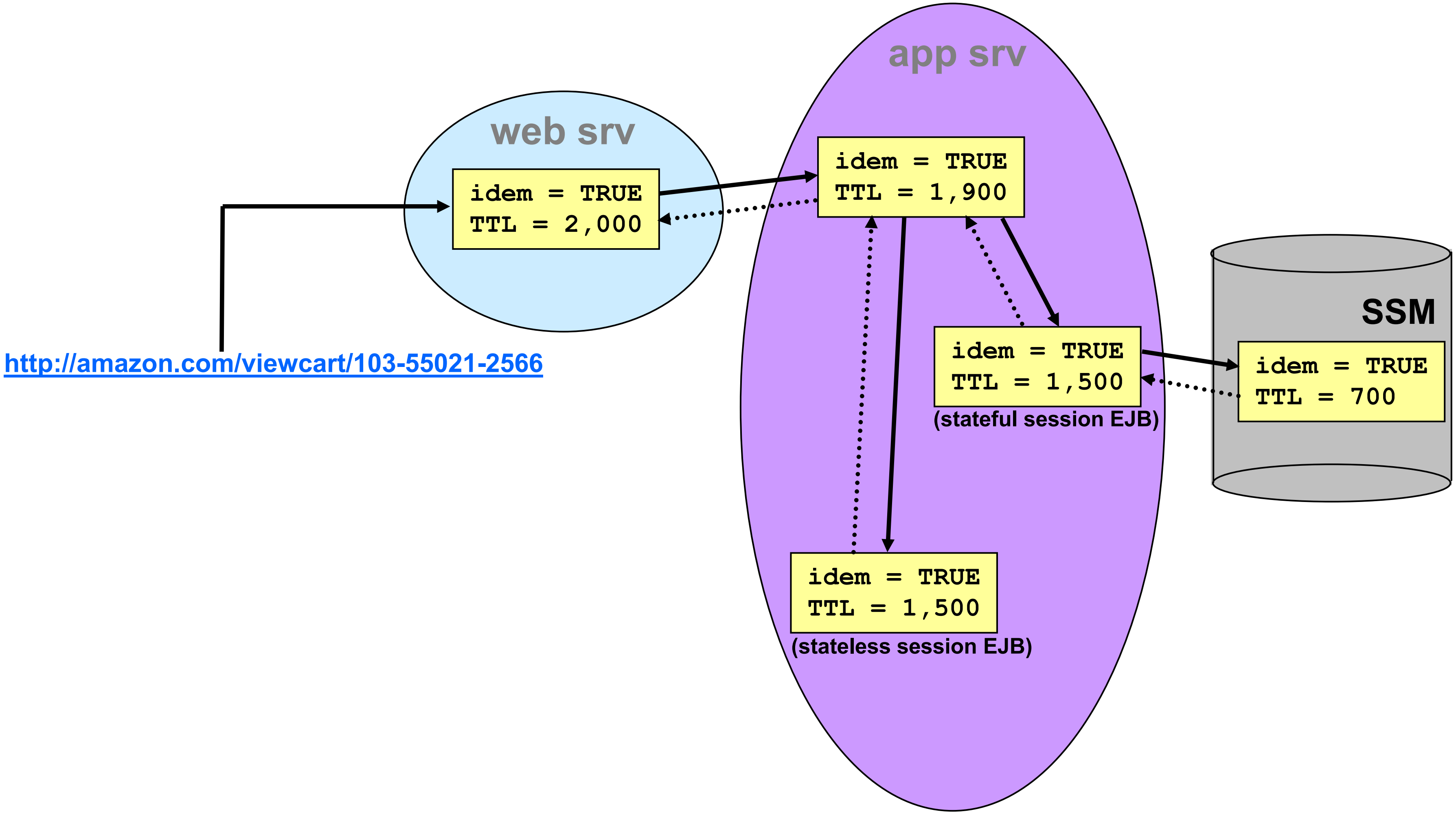
- F-map indicates how faults propagate → recovery policy
- First attempt recovery of a minimal subset of components



- What if µReboot(s) ineffective? → recover progressively larger subsets
- Chase fault through successive fault boundaries
- If reboot-based strategy doesn't work, notify operator

© 2003 George Candea

Looking ahead: Restart/Retry Architecture



© 2003 George Candea

Benefits of Architecture

- Transparent sub-system recovery → failure masking and continuity of service
- Can do hot bug fixes and upgrades = crash old component, recover new one (modulo API changes)
- Zero-downtime rolling rejuvenation of components (µReboot components to prevent failure from resource exhaustion)
- Trivial migration of tasks (e.g., for failover, load balancing, reconfiguration) = crash on one node, recovery on the other

© 2003 George Candea

Further Information

http://crash.stanford.edu

http://reboot.stanford.edu

© 2003 George Candea

Overflow Slides

© 2003 George Candea

Reboots: Good and Bad

GOOD

- If properly designed, will unequivocally bring recovered system to start state = best understood, best tested
- Reclaim leaked/stale resources (memory, fd's) → rejuvenation
- Easy to understand/employ → implement, debug, automate
- Frequent use: most bugs in prod-quality sys are transient/intermittent!

BAD

- Software not designed to tolerate (Hardware is much better)
- Can lead to extended downtime
- Data corruption/loss

Fix the BAD, exploit the GOOD…

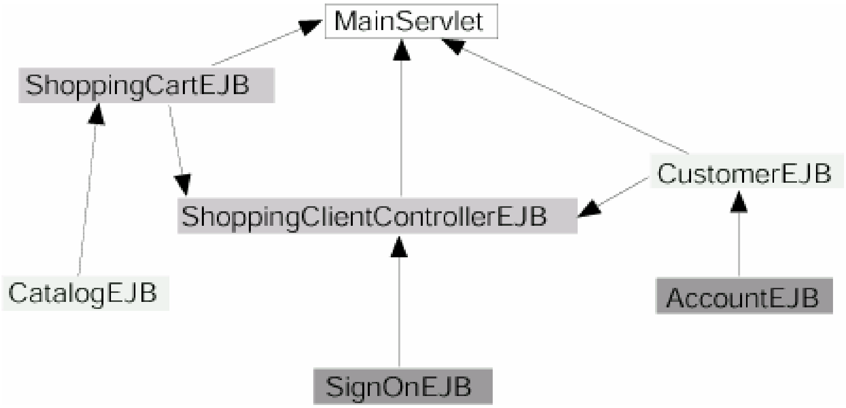
© 2003 George Candea

AFPI: Invasive Phase

- Injected faults = Java exceptions (high level, better coverage)
- Every time an EJB C is deployed, use reflection to discover each of its methods (M1, M2, …) and, for each method, the thrown exceptions ($F1_{M1}$, $F2_{M1}$, …); add tuples <C, Mi, Fj_{Mi} > to list L of faults
- Also add environmental exceptions as tuples (network-related, disk I/O, memory-related, etc.)
- Once entire application is deployed,
  - Iterate through list, and inject one fault at a time
  - Place load on application, using LoadGen
- As components fail, the monitors report to the Recovery Manager
- RecoMgr builds fault dependency map by adding edges for each fault propagation
- To simulate correlated faults, use cross-products LxL, LxLxL, etc.
- After deployment, continue with passive phase, and update the recovery map as application evolves

© 2003 George Candea

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

© 2003 George Candea