Reboot-based High Availability

Turning evil reboots into reliable friends

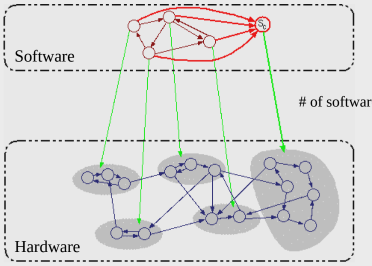
George Candea **Armando Fox**

Stanford University

Motivation

- Software infrastructures = aggregation of industrial strength software components, yet…
  - 40% of total unplanned downtime due to application failure [Gartner]
  - Business losses: average $6.45 million/hour of downtime for brokerage industry [Dataquest]
- Good news: ≥ 90% of bugs in production-quality software are transient [Adams, Gray] → **REBOOT** !
- System management costs >> installation costs

Modeling Software Systems

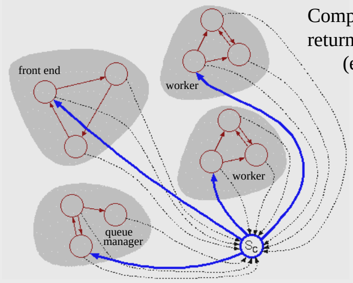


Problem: mapping is not surjective

# of software states << # of hardware states

Bug ⇔ transition to S_C (usually not accounted for)

Modeling Partial Reboots



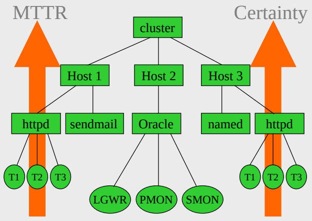
Component can deterministically return to well-known, clean state (e.g., deadlocks in a DBMS)

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

$\lim_{MTTF \to \infty}(Avail) = 1$ **impossible**

$\lim_{MTTR \to 0}(Avail) = 1$ **easier**

Restartability Tree



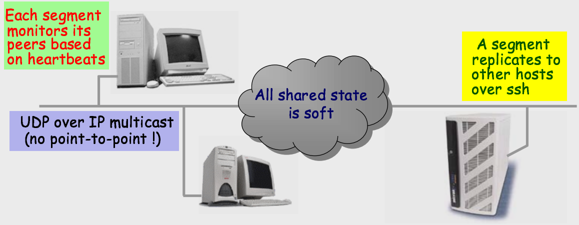
- Captures reboot-related dependencies
- Expresses inter-component hierarchy
- Cannot rely on component to restart cleanly

Parent node:

- reclaims all resources from children
- restarts children

Execution Infrastructure

- Amorphous cooperative of self-managing segments
- Segment = unit of glue between applications and infrastructure
- Can distribute segments locally (LAN) or geographically (WAN)



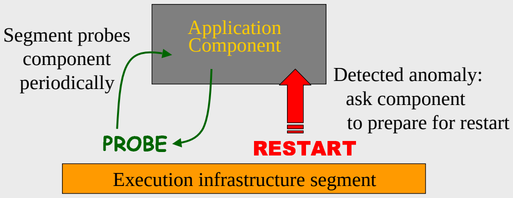
Each segment monitors its peers based on heartbeats

A segment replicates to other hosts over ssh

All shared state is soft

UDP over IP multicast (no point-to-point !)

Component Hooks



Segment probes component periodically

Application Component

PROBE

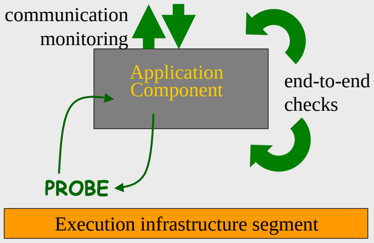
RESTART

Detected anomaly: ask component to prepare for restart

Execution infrastructure segment

- Time to prepare is pre-determined (execution contract)
- Component’s parent restarts it after preparation time elapses

Evaluating Application Progress



communication monitoring

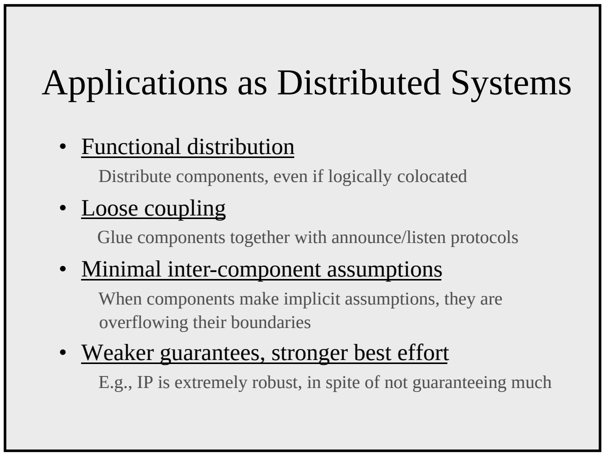
Application Component

end-to-end checks

PROBE

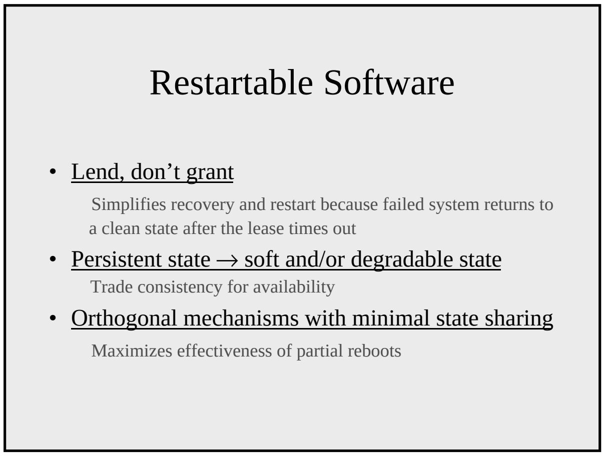
Execution infrastructure segment

2 types of checks: application-specific (PROBE)
domain-specific (e.g., well-known query)



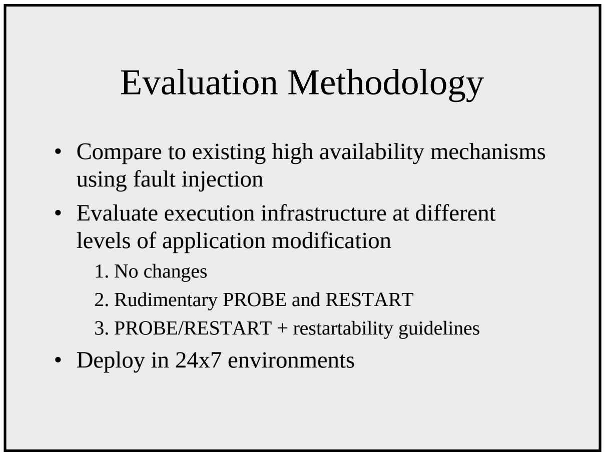
Applications as Distributed Systems

- Functional distribution
  Distribute components, even if logically colocated
- Loose coupling
  Glue components together with announce/listen protocols
- Minimal inter-component assumptions
  When components make implicit assumptions, they are overflowing their boundaries
- Weaker guarantees, stronger best effort
  E.g., IP is extremely robust, in spite of not guaranteeing much



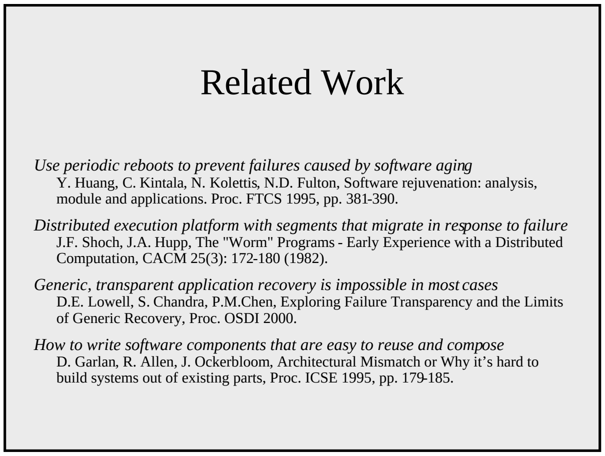
Restartable Software

- Lend, don’t grant
  Simplifies recovery and restart because failed system returns to a clean state after the lease times out
- Persistent state → soft and/or degradable state
  Trade consistency for availability
- Orthogonal mechanisms with minimal state sharing
  Maximizes effectiveness of partial reboots



Evaluation Methodology

- Compare to existing high availability mechanisms using fault injection
- Evaluate execution infrastructure at different levels of application modification
  1. No changes
  2. Rudimentary PROBE and RESTART
  3. PROBE/RESTART + restartability guidelines
- Deploy in 24x7 environments



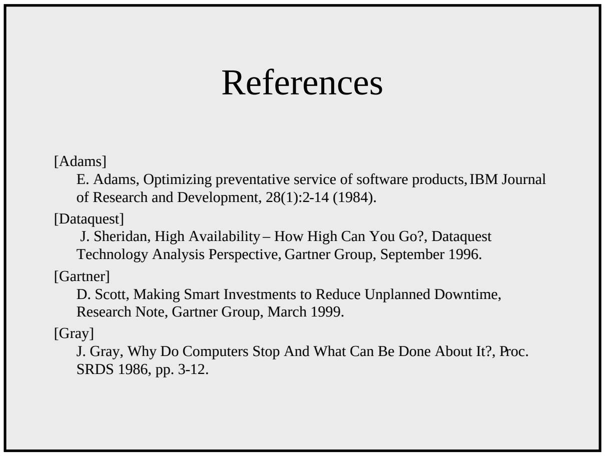
Related Work

Use periodic reboots to prevent failures caused by software aging
Y. Huang, C. Kintala, N. Kolettis, N.D. Fulton, Software rejuvenation: analysis, module and applications. Proc. FTCS 1995, pp. 381-390.

Distributed execution platform with segments that migrate in response to failure
J.F. Shoch, J.A. Hupp, The "Worm" Programs - Early Experience with a Distributed Computation, CACM 25(3): 172-180 (1982).

Generic, transparent application recovery is impossible in most cases
D.E. Lowell, S. Chandra, P.M.Chen, Exploring Failure Transparency and the Limits of Generic Recovery, Proc. OSDI 2000.

How to write software components that are easy to reuse and compose
D. Garlan, R. Allen, J. Ockerbloom, Architectural Mismatch or Why it’s hard to build systems out of existing parts, Proc. ICSE 1995, pp. 179-185.



References

[Adams]
E. Adams, Optimizing preventative service of software products, IBM Journal of Research and Development, 28(1):2-14 (1984).

[Dataquest]
J. Sheridan, High Availability – How High Can You Go?, Dataquest Technology Analysis Perspective, Gartner Group, September 1996.

[Gartner]
D. Scott, Making Smart Investments to Reduce Unplanned Downtime, Research Note, Gartner Group, March 1999.

[Gray]
J. Gray, Why Do Computers Stop And What Can Be Done About It?, Proc. SRDS 1986, pp. 3-12.