RECOVERY-
ORIENTED
COMPUTING

To Err is Human

Aaron Brown and David A. Patterson

Computer Science Division
University of California at Berkeley

First EASY Workshop
1 July 2001

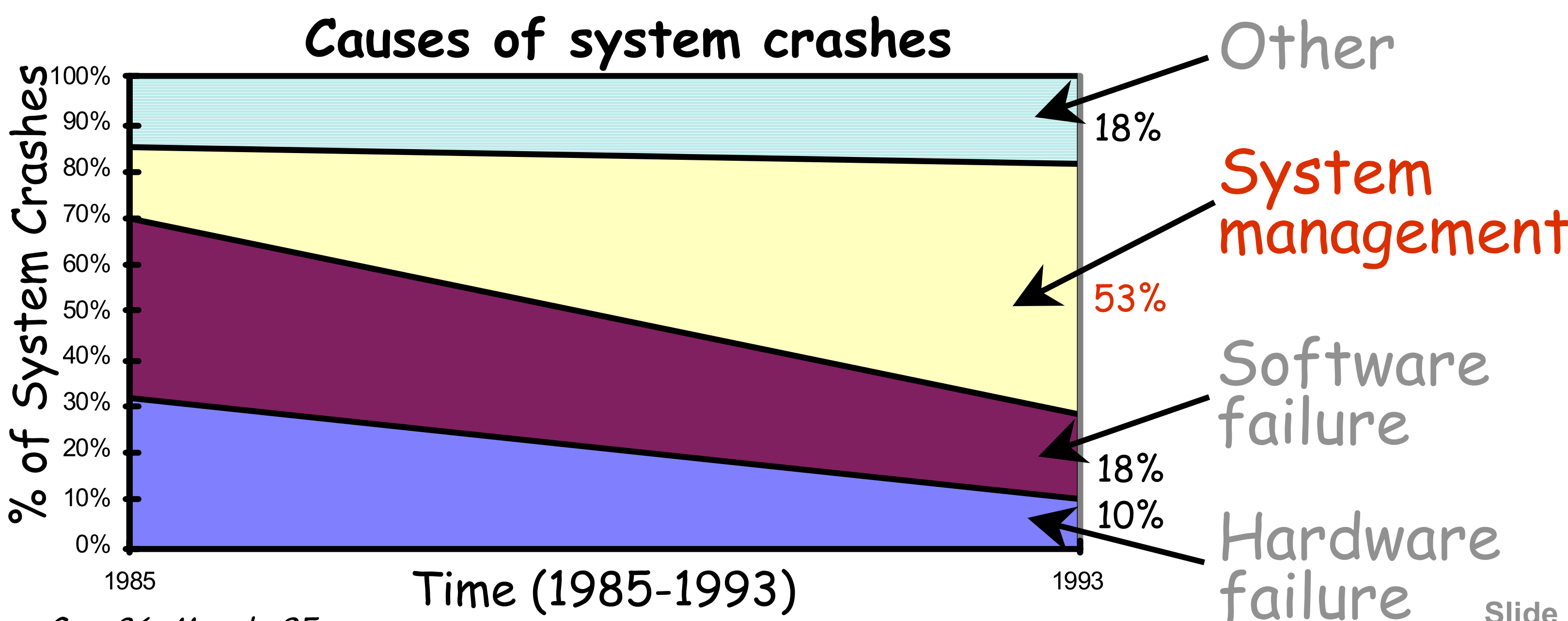
The dependability challenge

- **Server system dependability is a big concern**
  - outages are frequent, especially for Internet services
    - » 65% of IT managers report that their websites were unavailable to customers over a 6-month period
      - 25%: 3 or more outages
    - » EBay: entire site is fully-functioning < 90% of time
  - outages costs are high
    - » NYC stockbroker: $6,500,000/hr
    - » EBay: $ 225,000/hr
    - » Amazon.com: $ 180,000/hr
    - » social effects: negative press, loss of customers who "click over" to competitor

Source: InternetWeek 4/3/2000, EBay daily logs (thanks to Patricia Enriquez for data)

Humans cause failures

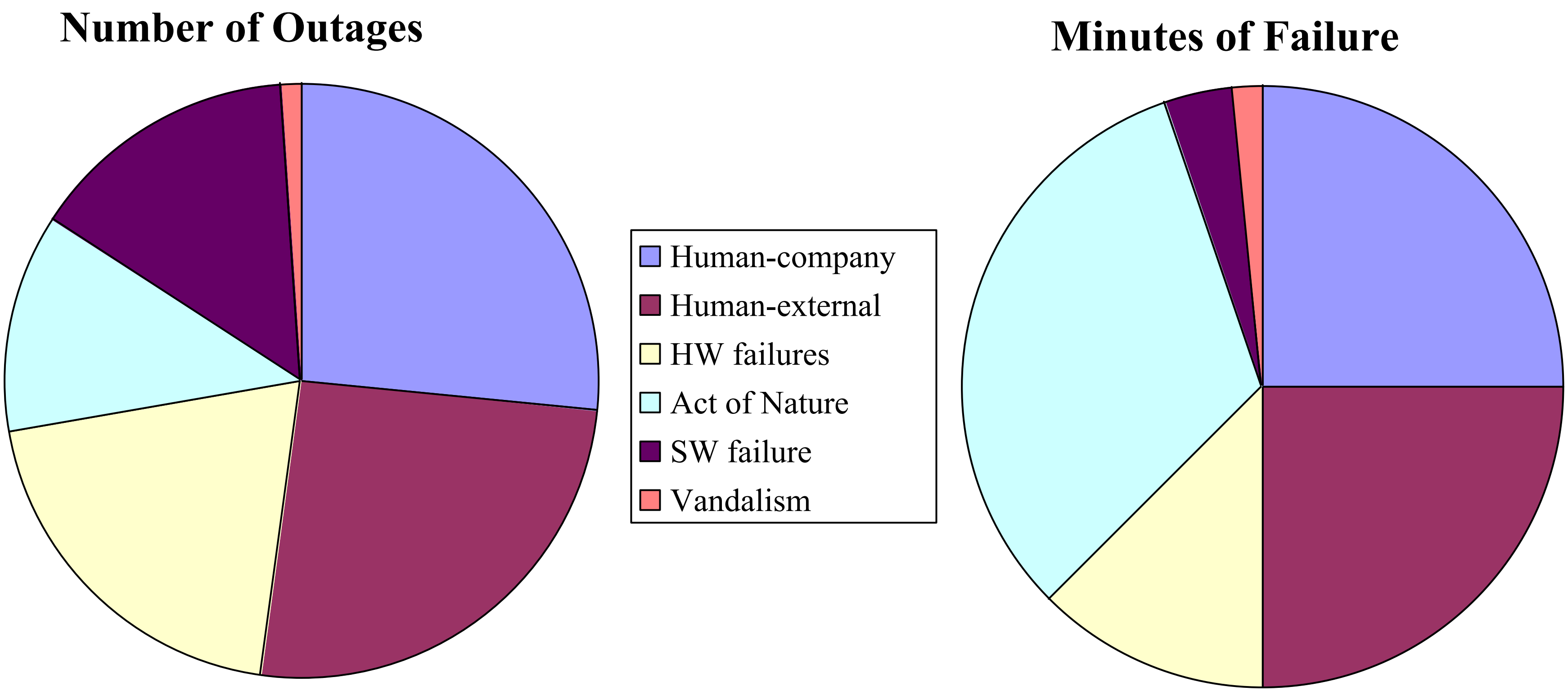
- **Human error is largest single failure source**
  - HP HA labs: human error is #1 cause of failures (2001)
  - Oracle: half of DB failures due to human error (1999)
  - Gray/Tandem: 42% of failures from human administrator errors (1986)
  - Murphy/Gent study of VAX systems (1993):



Sources: Gray86, Murphy95

Humans cause failures (2)

- **More data: telephone network failures**
  - from FCC records, 1992-1994



  - half of outages, outage-minutes are human-related
    - » about 25% are direct result of maintenance errors by phone company workers

Source: Kuhn, IEEE Computer 30(4), 1997.

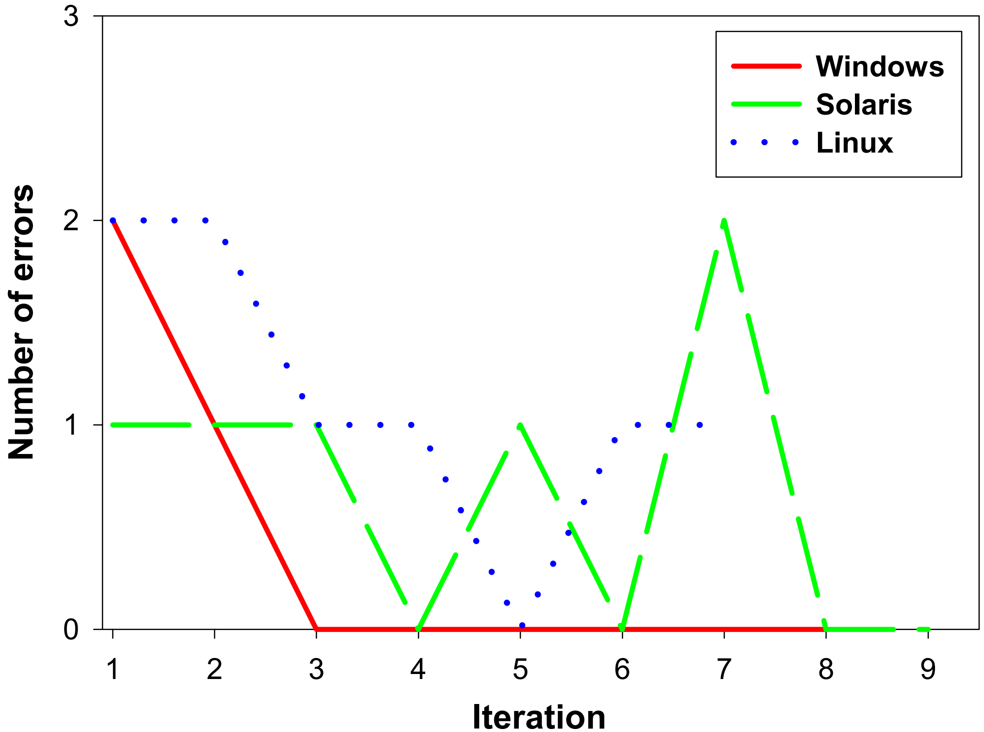
Humans cause failures (3)

- **Human error rates during maintenance of software RAID system**
  - participants attempt to repair RAID disk failures
    - » by replacing broken disk and reconstructing data
  - each participant repeated task several times
  - data aggregated across 5 participants

Error type	Windows	Solaris	Linux
Fatal Data Loss	💣		💣💣
Unsuccessful Repair			💣
System ignored fatal input			💣
User Error – Intervention Required	💣	💣💣	💣
User Error – User Recovered	💣	💣💣💣💣	💣💣
Total number of trials	**35**	**33**	**31**

Humans cause failures (4)

- **Errors occur despite experience:**



- **Training and familiarity can't eliminate errors**
  - mistakes mostly in 1st iterations; rest are slips/lapses
- **System design affects error-susceptibility**

Don't just blame the operator!

- **Psychology shows that human errors are inevitable** [see J. Reason, *Human Error*, 1990]
  - humans prone to *slips* & *lapses* even on familiar tasks
    - » 60% of errors are on "skill-based" automatic tasks
  - also prone to *mistakes* when tasks become difficult
    - » 30% of errors on "rule-based" reasoning tasks
    - » 10% of errors on "knowledge-based" tasks that require novel reasoning from first principles
- **Allowing human error can even be beneficial**
  - mistakes are a part of trial-and-error reasoning
    - » trial & error is needed to solve knowledge-based tasks
      - like problem diagnosis and performance tuning
    - » fear of error can stymie innovation and learning

What can we do?

- **Human error is inevitable, so we can't avoid it**

  "If a problem has no solution, it may not be a problem, but a fact, not to be solved, but to be coped with over time" — Shimon Peres

- **We must build dependable systems that can cope with human error**
  - and even encourage it by supporting trial-and-error
  - allow operators to learn from their mistakes
- **We must build benchmarks that measure dependability in the face of human error**
  - "benchmarks shape a field" and motivate progress

Dependability benchmarks & humans

- **End-to-end dependability benchmarks ("TPC")**
  - **model:** complete system evaluated for availability/QoS under injected "upset-load"
  - **goal:** measure overall system dependability *including human component, positive and negative*
  - **approach:** involve humans in the benchmark process
    - » select "best" administrators to participate
    - » include maintenance, upgrades, repairs in upset-load
  - **benefits:** captures overall human contribution to dependability (both positive and negative)
  - **drawbacks:** produces an upper-bound measure; hard to identify human contribution to dependability

Dependability benchmarks (2)

- **Dependability microbenchmarks**
  - **model:** component(s) tested for susceptibility to upsets
  - **goal:** isolate human component of dependability
    - » system's propensity for causing human error
    - » dependability impact of those errors
  - **approach:** usability experiments involving humans
    - » participants carry out maintenance tasks and repairs
    - » evaluate frequency and types of errors made
    - » evaluate component's resilience to those errors
  - **benefits:** direct evaluation of human error impact on dependability
  - **drawbacks:** ignores positive contribution of humans; requires large pool of representative participants

Human participation in benchmarks

- **Our approaches require human participation**
  - significantly complicates the benchmark process
  - hard to get enough trained admins as participants
  - makes comparison of systems difficult
- **Can we eliminate the human participation?**
  - end-to-end benchmarks need a human behavior model
    - » if we had this, we wouldn't need system administrators!
  - microbenchmarks require only a human error model
    - » but, human errors are inherently system dependent
      - function of UI, automation, error susceptibility, ...
    - » may be possible to build a model for a single system, but no generalized benchmark yet
    - » *good place for future research . . .*

Dependable human-operated systems

- **Avoiding human error**
  - automation: reducing human involvement
    - » SW: self-tuning, no-knobs, adaptive systems, ...
    - » HW: auto-sparing, configuration, topology discovery, ...
    - » but beware of automation irony!
  - training: increasing familiarity with system
    - » on-line training on realistic failure scenarios in a protected sandbox
  - *avoidance is only a partial solution*
    - » some human involvement is unavoidable
    - » any involvement provides opportunity for errors

The key to dependability?

- **Building tolerance for human error**
  - accept inevitability of human involvement and error
    - » focus on *recovery*
  - **undo:** the ultimate recovery mechanism?
    - » ubiquitous and well-proven in productivity applications
    - » familiar model for error recovery
    - » enables trial-and-error interaction patterns
  - undo for system maintenance
    - » "time-travel" for system state
    - » must encompass all hard state, including hardware & network configuration
    - » must be flexible, low-overhead, and transparent to end user of system

Conclusions

- **Humans are critical to system dependability**
  - human error is the single largest cause of failures
- **Human error is inescapable: "to err is human"**
  - yet we blame the operator instead of fixing systems
- **We must take human error into account when building dependable systems**
  - in our system designs, by providing tolerance through mechanisms like undo
  - in our dependability evaluations, by including a human component in dependability benchmarks
- **The time is ripe for human error research!**
  - the key to the next significant dependability advance?

Recovery-Oriented Computing

To Err is Human

For more information:

{abrown,patterson}@cs.berkeley.edu
http://roc.cs.berkeley.edu

Backup slides

Recovery from human error

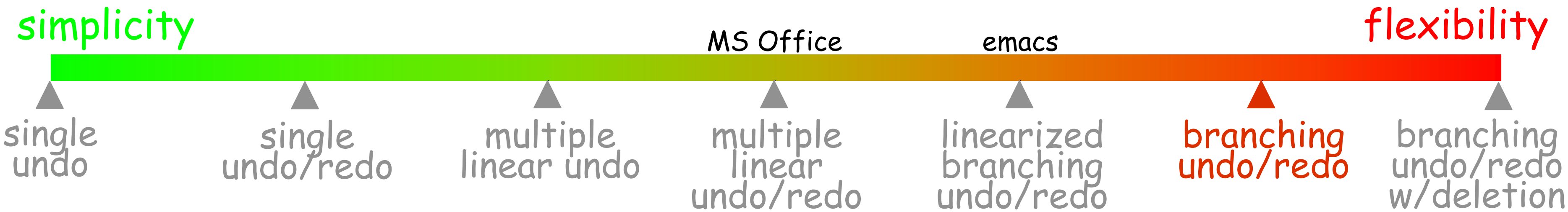
- **ROC principle: recovery from human error, not avoidance**
  - accepts inevitability of errors
  - promotes better human-system interaction by enabling trial-and-error
    - » improves other forms of system recovery
- **Recovery mechanism: Undo**
  - ubiquitous and well-proven in productivity applications
  - unusual in system maintenance
    - » primitive versions exist (backup, standby machines, ...)
    - » but not well-matched to human error or interaction patterns

Undo paradigms

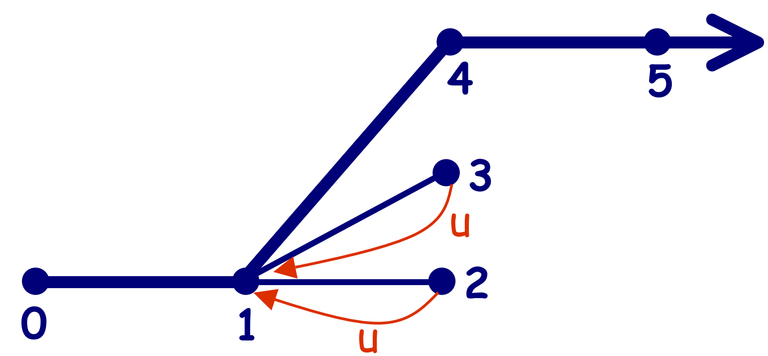
- **An effective undo paradigm matches the needs of its target environment**
  - cannot reuse existing undo paradigms for system maintenance
- **We need a new undo paradigm for maintenance**
  - plan:
    - » lay out the design space
    - » pick a tentative undo paradigm
    - » carry out experiments to validate the paradigm
- **Underlying assumption: service model**
  - single application
  - users access via well-defined network requests

Issue #1: Choice of undo model

- Undo model defines the view of past history
- Spectrum of model options:



- Important choices:
  - undo only, or undo/redo?
  - single, linear, or branching?
  - deletion or no deletion?
- Tentative choice for maintenance undo



trial-and-error history pattern

More undo issues

2) Representation

- does undo act on **states** or **actions**?
- how are the states/actions named? TBD

3) Selection of undo points

- granularity:
  - » undo points at each state change/action?
  - » or at checkpoints of some granularity?
- are undo points administrator- or system-defined?

- **Tentative maintenance undo choices in red**

More undo issues (2)

4) Scope of undo

- "what state can be recovered by undo?"
- single-node, multi-node, multi-node+network?
- on each node:
  - » system hardware state: BIOS, hardware configs?
  - » disk state: user, application, OS/system?
  - » soft state: process, OS, full-machine checkpoints?
- tentative maintenance undo goals in red

More undo issues (3)

5) Transparency to service user

- ideally:
  - » undo of system state preserves user data & updates
  - » user always sees consistent, forward-moving timeline
  - » undo has no user-visible impact on data or service availability

Context: other undo mechanisms

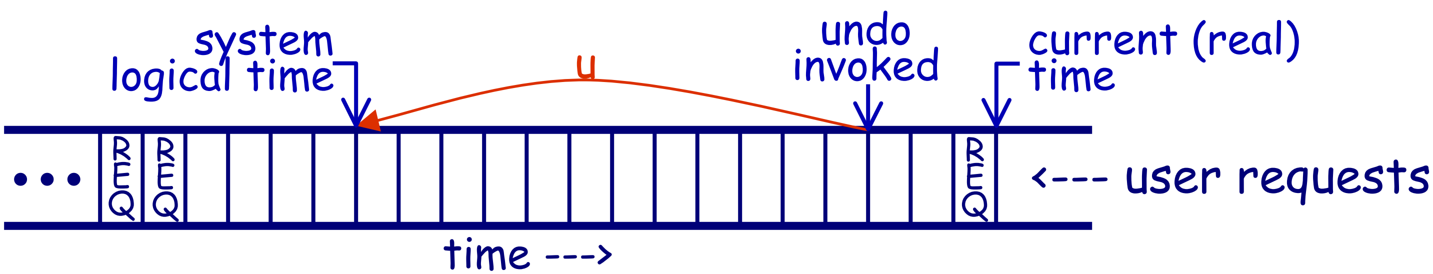
Design axis / Undo mech.	Undo model	Representation	Undo-point selection	Scope	Trans-parency
Desired maintenance-undo semantics	branching undo/redo	state, naming TBD	automatic checkpoints	all disk & HW, all nodes & network	high
Geoplex site failover	single undo	state, unnamed	varies; usu. automatic checkpoints	entire system	high
Tape backup	single or multiple linear undo	state ad-hoc naming	manual checkpoints	disk (1 FS), single node	low
GoBack®	linearized branching undo/redo	state, temporal naming	automatic checkpoints	disk (all), single node	low-medium
Netapp Snapshots	multiple linear undo	state, temporal naming	manual checkpoints	disk (all), single server	low
DBMS logging (for txn abort)	single undo	hybrid, unnamed	automatic checkpoints	single txn, app-level	high

Implementing maintenance undo

- **Saving state: disk**
  - apply snapshot or logging techniques to disk state
    - » e.g., NetApp- or VMware-style block snapshots, or LFS
    - » all state, including OS, application binaries, config files
  - leverage excess of cheap, fast storage
  - integrate "time travel" with native storage mechanism for efficiency
- **Saving state: hardware**
  - periodically discover and log hardware configuration
  - can't automatically undo all hardware changes, but can direct administrator to restore configuration

Implementing maintenance undo (2)

- **Providing transparency**
  - queue & log user requests at edge of system, in format of original request protocol
  - correlate undo points to points in request log
  - snoop/replay log to satisfy user requests during undo



- **An undo UI**
  - should visually display branching structure
  - must provide way to name and select undo points, show changes between points

Status and plans

- **Status**
  - starting human experiments to pin down undo paradigm
    - » subjects are asked to configure and upgrade a 3-tier e-commerce system using HOWTO-style documentation
    - » we monitor their mistakes and identify where and how undo would be useful
  - experiments also used to evaluate existing undo mechanisms like those in GoBack and VMware
- **Plans**
  - finalize choice of undo paradigm
  - build proof-of-concept implementation in Internet email service on ROC-1 cluster
  - evaluate effectiveness and transparency with further experiments