Why do Internet services fail, and what can be done about it?

David Oppenheimer
davidopp@cs.berkeley.edu
ROC Group, UC Berkeley

ROC Retreat, June 2002

Motivation

- Little understanding of real problems in maintaining 24x7 Internet services
- Identify the common failure causes of real-world Internet services
  - these are often closely-guarded corporate secrets
- Identify techniques that would mitigate observed failures
- Determine fault model for availability and recoverability benchmarks

Sites examined

1. **Online service/portal**
   - ~500 machines, 2 facilities
   - ~100 million hits/day
   - all service software custom-written (SPARC/Solaris)
2. **Global content hosting service**
   - ~500 machines, 4 colo facilities + customer sites
   - all service software custom-written (x86/Linux)
3. **Read-mostly Internet site**
   - thousands of machines, 4 facilities
   - ~100 million hits/day
   - all service software custom-written (x86)

Outline

- Motivation
- Terminology and methodology of the study
- Analysis of root causes of faults and failures
- Analysis of techniques for mitigating failure
- Potential future work

Terminology and Methodology (I)

- **Examined 2 operations problem tracking databases, 1 failure post-mortem report log**
- **Two kinds of failures**
  - *Component failure ("fault")*
    - » hardware drive failure, software bug, network switch failure, operator configuration error, …
    - » may be masked, but if not, becomes a…
  - *Service failure ("failure")*
    - » prevents an end-user from accessing the service or a part of the service; or
    - » significantly degrades a user-visible aspect of perf.
    - » inferred from problem report, not measured externally
  - Every service failure is due to a component failure

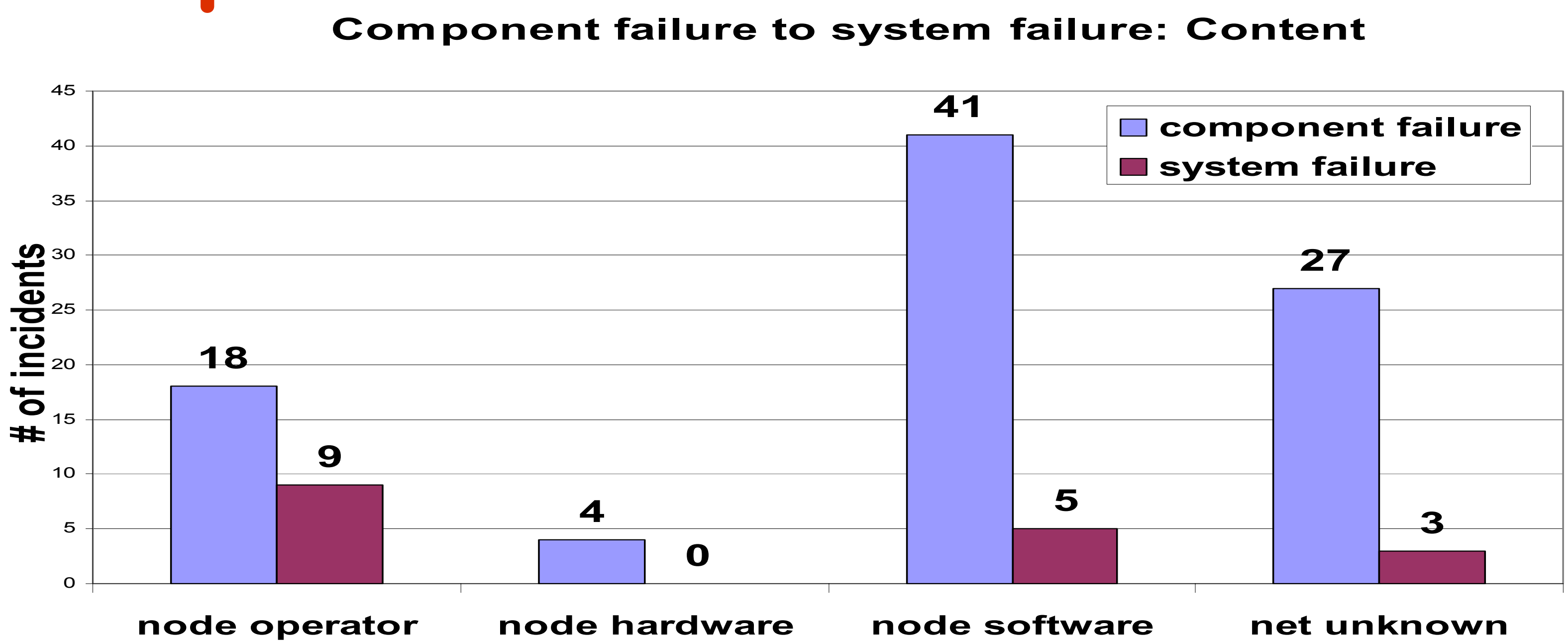
Terminology and Methodology (II)

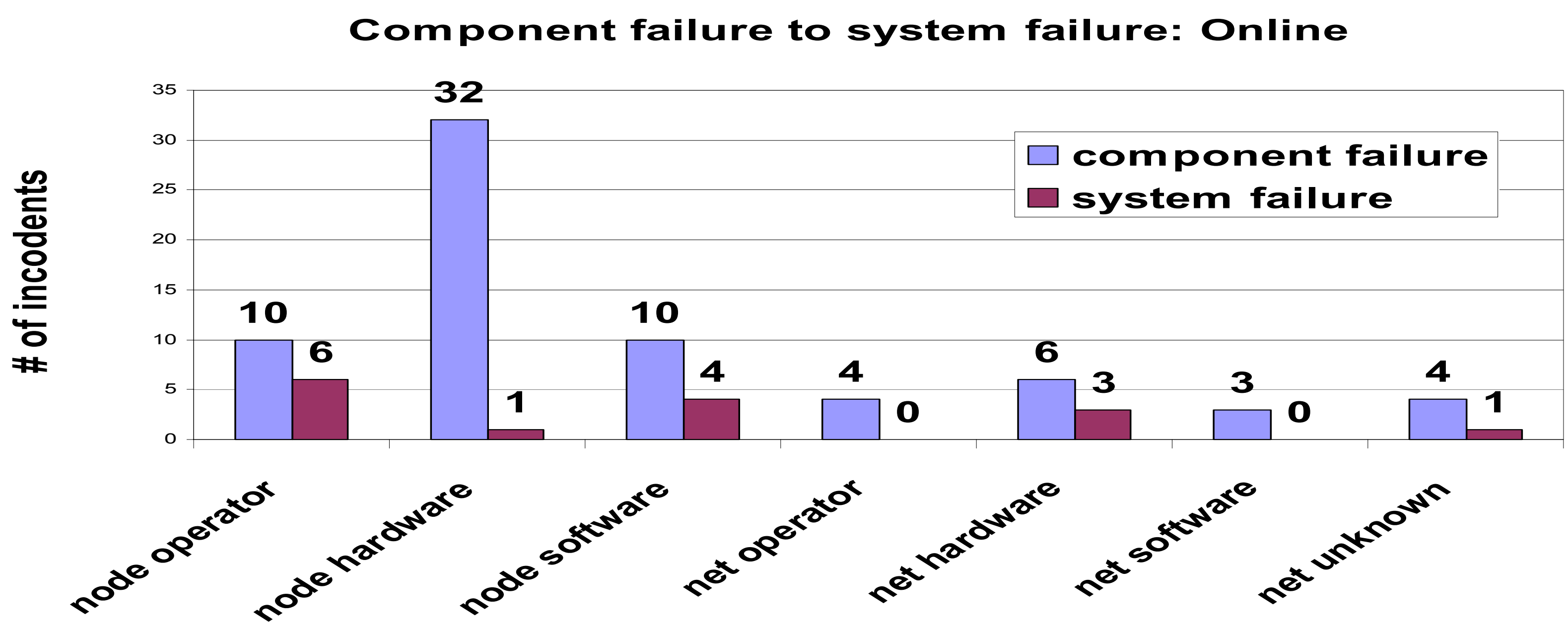
Service	# of component failures	# of resulting service failures	period covered in problem reports
Online	85	18	4 months
Content	99	20	1 month
ReadMostly	N/A	21	6 months

(note that the services are not directly comparable)

- **Problems are categorized by "root cause"**
  - first component that failed in the chain of events leading up to the observed failure
- **Two axes for categorizing root cause**
  - ***location***: front-end, back-end, network, unknown
  - ***type:*** node h/w, node s/w, net h/w, net s/w, operator, environment, overload, unknown

Component failure ➡ service failure





Service failure ("failure") causes

	front-end	back-end	net	unknown
Online	72%		28%	
Content	55%	20%	20%	5%
ReadMostly	0%	10%	81%	9%

Front-end machines are a significant cause of failure

	node op	net op	node hw	net hw	node sw	net sw	node unk	net unk
Online	33%		6%	17%	22%			6%
Content	45%	5%			25%			15%
ReadMostly	5%	14%		10%	5%	19%		33%

Operator error is largest cause of failure for two services, network problems for one service

Service failure average TTR (hours)

average TTR in hrs	front-end	back-end	net
Online	9.7	10.2	0.75 *(*)*
Content	2.5	14	1.2 *(*)*
ReadMostly		0.17 *(*)*	1.2

average TTR in hrs	node op	net op	node hw	net hw	node sw	net sw	net unk
Online	15		1.7 *(*)*	0.5 *(*)*	3.7 *(4)*		
Content	1.2				0.23		1.2 *(*)*
ReadMostly	0.17 *(*)*	0.13		6.0 *(*)*		1.0	0.11

(*) denotes only 1-2 failures in this category

Front-end TTR < Back-end TTR

Network problems have smallest TTR

Component failure ("fault") causes

	front-end	back-end	net
Online	76%	5%	19%
Content	34%	34%	30%

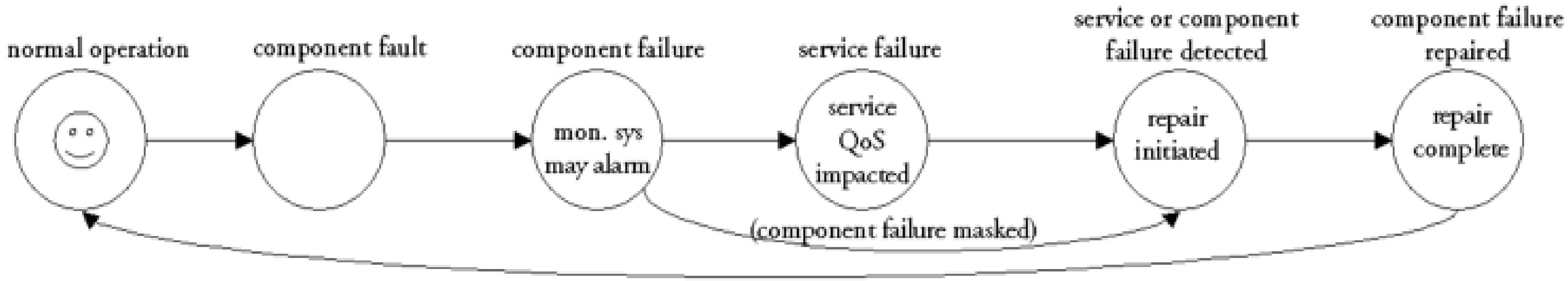
Component failures arise primarily in the front-end

	node op	net op	node hw	net hw	node sw	net sw	node unk	net unk	env
Online	12%	5%	38%	5%	12%	4%	4%	5%	0%
Content	18%	1%	4%	1%	41%	1%	1%	27%	1%

Operator errors are less common than hardware/ software component failures, but are less frequently masked

Techniques for mitigating failure (I)

- **How techniques could have helped**



- **Techniques we studied**
  1. testing (pre-test or online-test)
  2. redundancy
  3. fault injection and load testing (pre- or online)
  4. configuration checking
  5. isolation
  6. restart
  7. better exposing and diagnosing problems

Techniques for mitigating failure (II)

technique	# of problems mitigated (/19)
online testing	11
redundancy	8
online fault/load injection	3
configuration checking	3
isolation	2
pre-deployment fault/load injection	2
restart	1
pre-deployment correctness testing	1
better exposing/monitoring errors (TTD)	8
better exposing/monitoring errors (TTR)	8

Comments on studying failure data

- **Problem tracking DB may skew results**
  - operator can cover up errors before manifests as a (new) failure
- **Multiple-choice fields of problem reports much less useful than operator narrative**
  - form categories were not filled out correctly
  - form categories were not specific enough
  - form categories didn't allow multiple causes
- **No measure of customer impact**
- **How would you build an anonymized meta-database?**

Future work (I)

- **Continuing analysis of failure data**
  - New site? (e-commerce, storage system vendor, …)
  - More problems from Content and Online?
    - » say something more statistically meaningful about
      - MTTR
      - value of approaches to mitigating problems
      - cascading failures, problem scopes
    - » different time period from Content (longitudinal study)
  - Additional metrics?
    - » taking into account customer impact (customer-minutes, fraction of service affected, …)
  - Nature of original fault, how fixed?
  - Standardized, anonymized failure database?

Future work (II)

- **Recovery benchmarks (akin to dependability b/m's)**
  - use failure data to determine fault model for fault injection
  - recovery benchmark goals
    - » evaluate existing recovery mechanisms
      - common-case overhead, recovery performance, correctness, …
    - » match user needs/policies to available recovery mechanisms
    - » design systems with efficient, tunable recovery properties
      - systems can be built/configured to have different recoverability characteristics (RAID levels, check-pointing frequency, degree of error checking, *etc.*)
  - procedure
    1. choose application (storage system, three-tier application, globally distributed/p2p app, *etc.)*
    2. choose workload (user requests + operator preventative maintenance and service upgrade)
    3. choose representative faultload based on failure data
    4. choose QoS metrics (latency, throughput, fraction of service available, # users affected, data consistency, data loss, degree of remaining redundancy, …)

Future Work (III)

- **Recovery benchmarks, *cont.***
  - issues
    - » language for describing faults and their frequencies
      - hw, sw, net including WAN, operator
      - allows automated stochastic fault injection
    - » quantitative models for describing data protection/recovery mechanisms
      - how faults affect QoS
        - isolated & correlated faults
      - like to allow prediction of recovery behavior of single component and systems of components
    - » synthesizing overall recoverability metric(s)
    - » defining workload for systems with complicated interfaces (*e.g.*, whole "services")

Conclusion

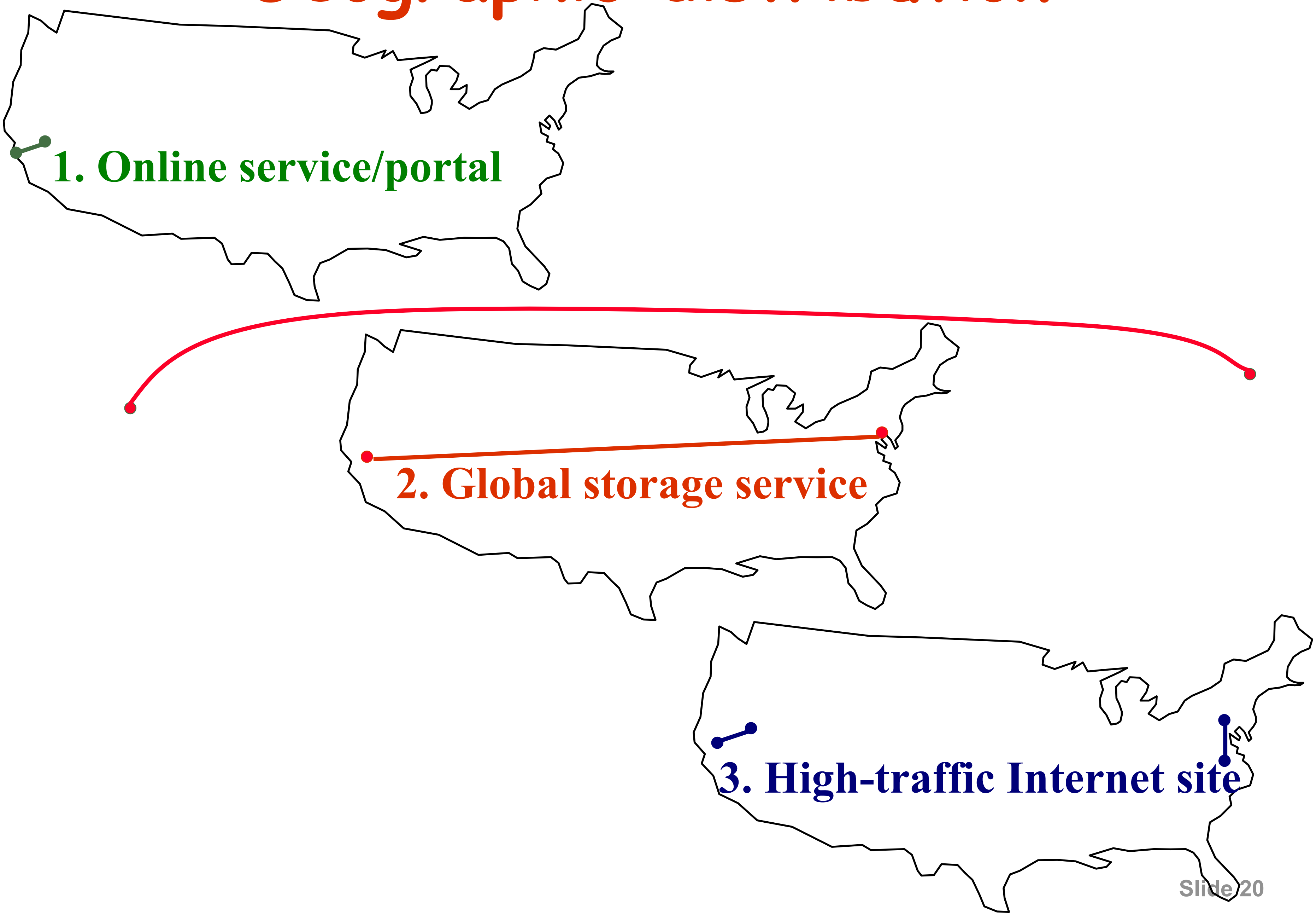
- **Failure causes**
  - operator error #1 contributor to service failures
  - operator error most difficult type of failure to mask; generally due to configuration errors
  - front-end software can be a significant cause of user-visible failures
  - back-end failures, while infrequent, take longer to repair than do front-end failures
- **Mitigating failures**
  - online correctness testing would have helped a lot, but hard to implement
  - better exposing, monitoring for failures would have helped a lot, but must be built in from ground up
  - for configuration problems, match system architecture to actual configuration
  - redundancy, isolation, incremental rollout, restart, offline testing, operator+developer interaction are all important (and often already used)

Backup Slides

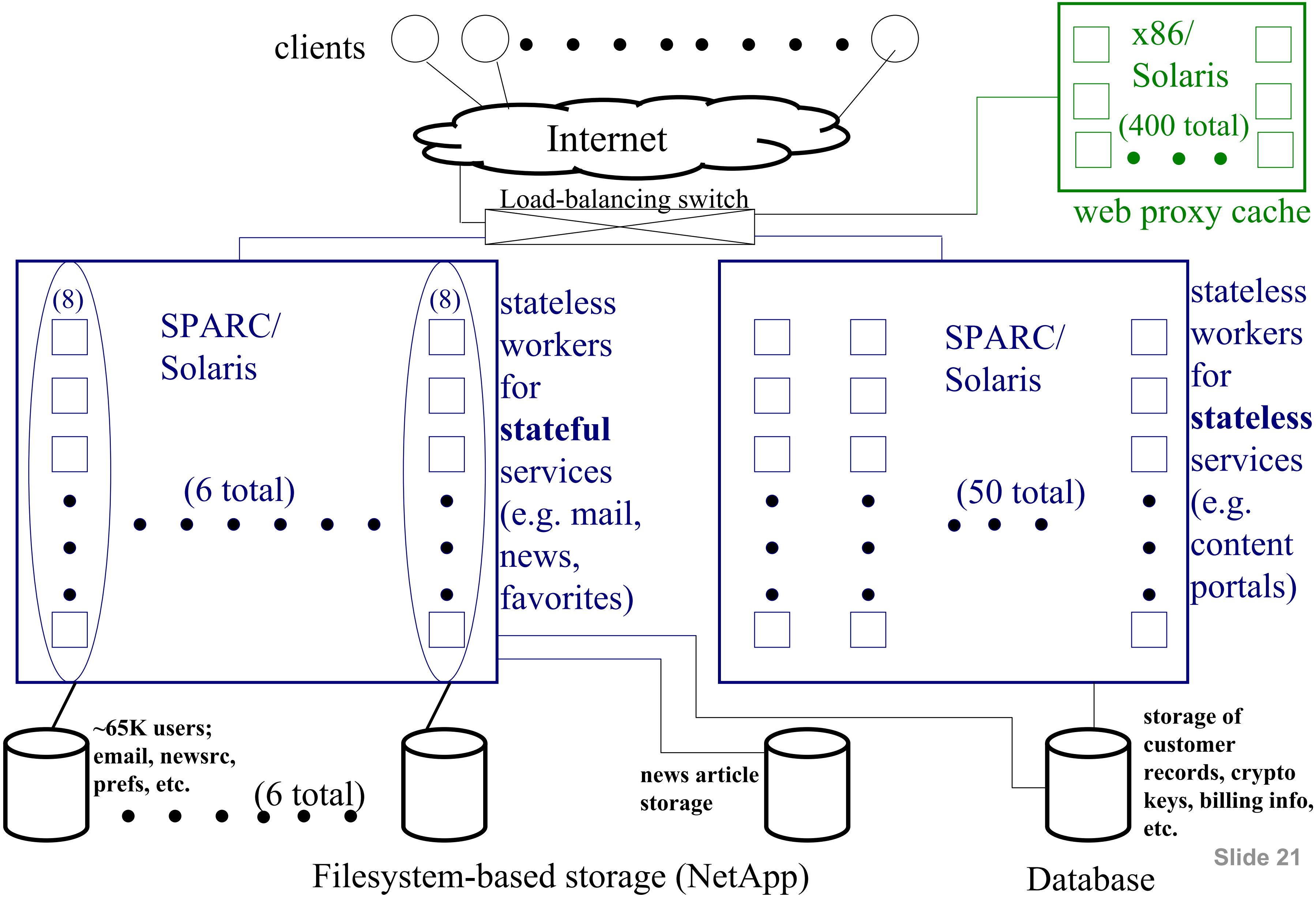
Techniques for mitigating failure (III)

technique	implementation cost	potential reliability cost	performance impact
online correctness testing	medium to high	low to medium	low to medium
redundancy	low	low	very low
online fault/load injection	high	high	medium to high
config checking	medium	zero	zero
isolation	medium	low	medium
pre-deployment fault/load injection	high	zero	zero
restart	low	low	low
pre-deployment testing	medium to high	zero	zero
better exposing/ monitoring failures	medium	low (false alarms)	low

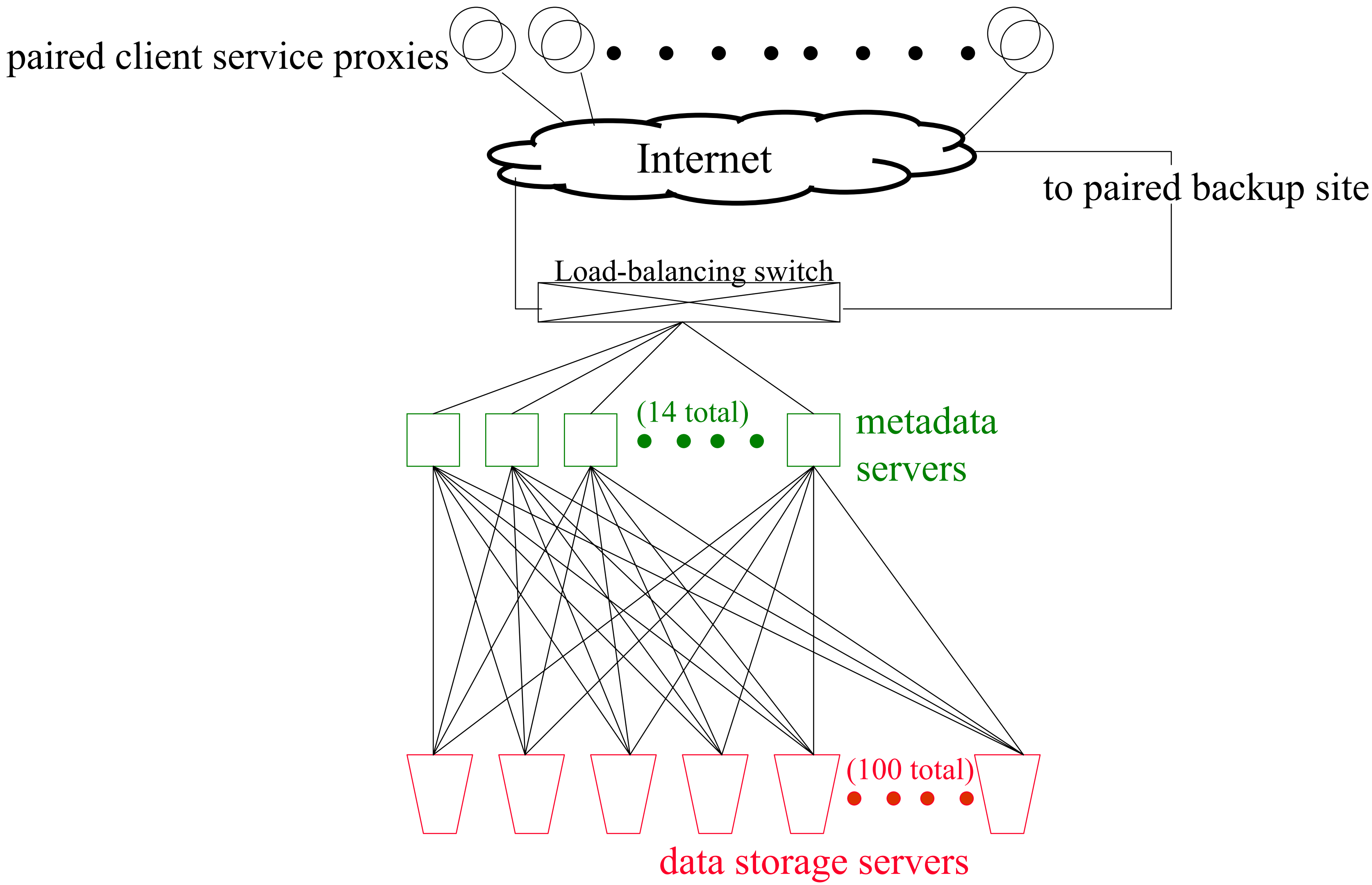
Geographic distribution



1. Online service/portal site



2. Global content hosting service site



3. Read-mostly Internet site