Problem diagnosis is difficult

- For developers of clouds
- For cloud users (i.e., software developers)
  - Must debug own applications
  - Must debug interactions w/cloud
    - E.g., is a slowdown due to other VMs or my app?
Monitoring via perf. counters

- Yields counters of low-level data
  - E.g., CPU time, disk I/Os, etc.
  - E.g., AWS CloudWatch, Ganglia

- **Pros:** Lightweight, commonly available
- **Cons:** Black-box; machine oriented

Logging events of interest

- Yields detailed text describing system’s behavior (e.g., application, OS, VM, etc.)
- Available in most systems (in some form)

- **Pros:** White-box approach
- **Cons:** High overhead; machine-oriented
End-to-end activity tracing

• Similar to logging, but workflow-based
• E.g., Dapper, Stardust, X-Trace, etc.

• **Pros:** White box, shows workflow
• **Cons:** Requires system modifications

Monitoring
A key cloud-specific issue

- Cloud providers and users usually do not wish to share information
- As such:
  - Counters normalized to VM capacity
    - e.g., percentage of AWS instance
  - Provider logs/traces not visible to users

Ganglia [Massie04]

- Designed for HPC environments
- Paper assumes bare-metal hardware
- Collects and aggregates counters
  - Counters can be app or machine specific
  - Within cluster, counters visible everywhere
  - Counters from multiple clusters aggregated
Ganglia architecture

![Ganglia Architecture Diagram]

AWS CloudWatch

- Provides monitoring for all AWS resources
- EC2 counters show VM-normalized values
- Also, can monitor app-specific metrics
End-to-end tracing overview

- Focus of many research efforts for ~10 yrs
- Currently used in Google, Bing, etc.
- Traces show causality-related activity
  - Trace: set of events from different threads/machines merged & sorted by causality
  - E.g., flow of indiv. requests (request flows)
End-to-end tracing implementation

- Tracing infrastructure tracks trace points touched by individual requests
- Some “start” traces (e.g., user request rec’d)
- Others propagate trace ID created at start
- Traces obtained by stitching together trace points accessed by individual requests
- Hard to account for async and batched work

Throttling by Sampling

- Users trace too little or too much
- Limit user bytes added per trace span
- Request sampling to limit global overhead
  - Collects all trace points for a req. or none
  - Hash trace ID to [0,1] and keep if < threshold
  - Allows end-2-end tracing to be “always on”
End-to-end tracing architecture

A few key design questions

- How much representational power?
- DAGs, trees, or paths?
- What causal relationships to preserve?
- Read-after-write, contention, etc.
- How many request flows to sample?
- Where to make sampling decision?
A DAG-based request flow

Nodes show trace points & edges show latencies

Dapper [Sigelman10]

- Google’s impl. of end-2-end tracing
- In use since at least 2008
- Similar in architecture to other examples
- But, optimized for traces expected at Google
- Trace records gathered in external system
- median lat. 15s, 25% of time 98%tile > hrs
Dapper design decisions

- Traces represented as trees of RPCs
- Node contains all work done for an RPC
- Edges indicate new RPC calls/replies
- Core tracing infrastructure + developer adds
- Sampling decision made at request entry
- Based on hash of root ID (keep x% traces)

Example Dapper trace tree
Dapper UI Example

Figure 6: A typical user workflow in the general-purpose Dapper user interface.

End-to-end tracing analysis tools
Spectroscope [Sambasivan11]

- Localizes performance degradations
  - By ID’ing changed request flows

- Output:
  - Groups of before/after request flows
  - Some changes automatically ID’d

Spectroscope workflow

Before degradation request flows

After degradation request flows

Grouping

Structural change identification

Response-time only change identification

Ranking

Presentation
Automatically ID’d changes

- Same structure is same trace points
- Groups w/structural changes
  - Identified via heuristics (e.g. freq. of types)
- Groups w/response-time changes
  - Have identical flows in both periods
  - ID’d via statistical significance testing

Group w/structural changes

Before

- NFS Lookup Call: 10μs
- MDS DB Lock: 20μs
- MDS DB Unlock: 50μs
- NFS Lookup Reply

After

- NFS Lookup Call: 10μs
- MDS DB Lock
- MDS DB Unlock: 350μs
- MDS DB Unlock: 50μs

Developers localize root cause by ID’ing how differences before/after degradation
Group w/response time

Before degradation avg. response time: 110μs
After degradation avg. response time: 1,090μs

NFS Read Call
Avg. 10μs

SN1 Read Start
Avg. 20μs
Avg. 1,000μs

SN1 Read End
Avg. 80μs

Root cause localized by ID’ing responsible interaction

Summary

• Debugging distributed systems is hard
• Performance debugging is harder still
• Monitoring is counting without causation
• But people want examples (traces)
  • Too much statistical analysis slows trust
  • Traces are logistically expensive, quick to rot