Debugging Distributed Systems

Today

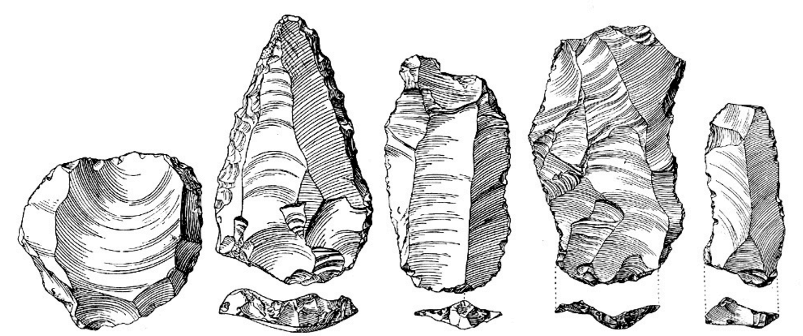
- Cuts, states, and properties
- Deterministic Replay
- Causal Tracing
- Won’t cover verification and model checking (see Logic for Systems!)

In the beginning…

… life was simple

- Activity happening in one thread ~ meaningful
- Hardware support for understanding execution
  - Stack hugely helpful (e.g. profiling, debugging)
- Single-machine systems
  - OS had global view
  - Timestamps in logs made sense

- gprof, gdb, dtrace, strace, top, …



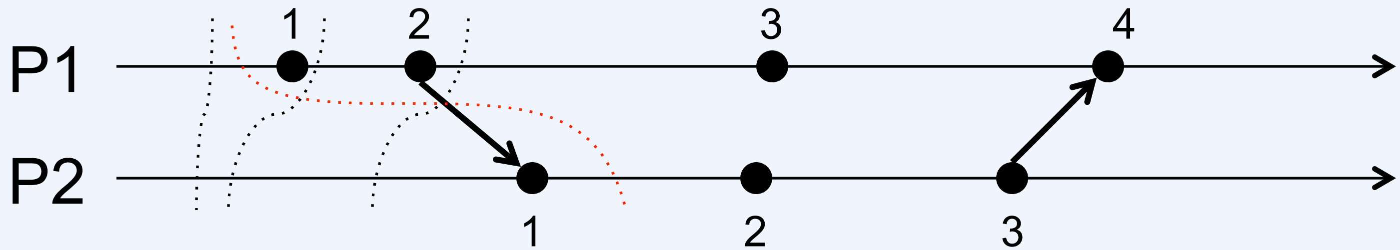
Source: Anthropology: Nelson, Gilbert, Wong, Miller, Price (20

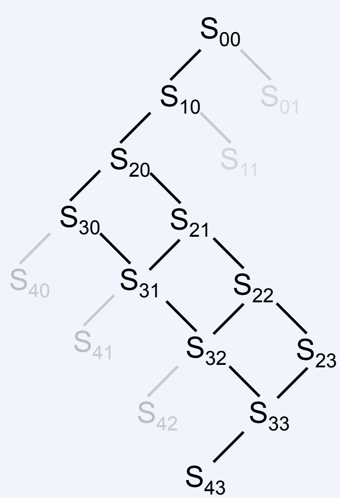
But then things got complicated

- Within a node
  - Threadpools, queues (e.g., SEDA), multi-core
  - Single-threaded event loops, callbacks, continuations
- Across multiple nodes
  - SOA, Ajax, Microservices, Dunghill
  - Complex software stacks
- Stack traces, thread ids, thread local storage, logs all tell a small part of the story

Evolution of a Distributed Program

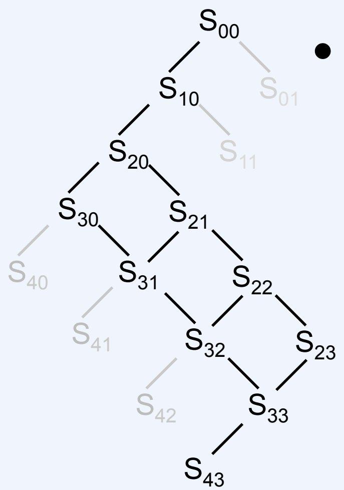
- Execution is a series of transitions between global states of a system





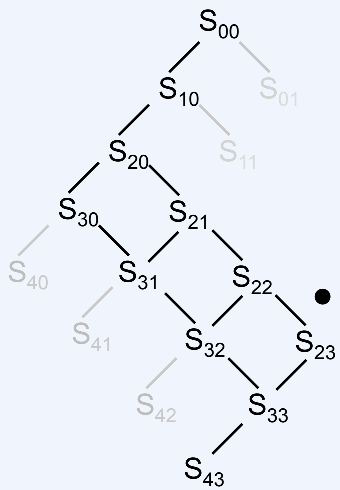
- Lattice of global states
- Paths on this lattice are possible executions, called *linearizations*

Evaluating Global Predicates



- Let Φ(S) be a predicate on a global state
  - E.g., there is a circular dependency (deadlock), or (x + y > z), …
- We want to evaluate two properties of the system:
  - possibly Φ: Φ is true at some point in at least one linearization
  - definitely Φ: Φ is true at some point in all linearizations

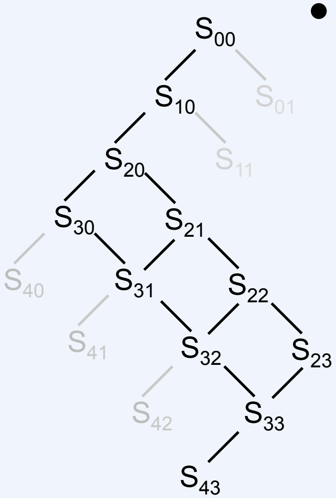
Evaluating Global Predicates

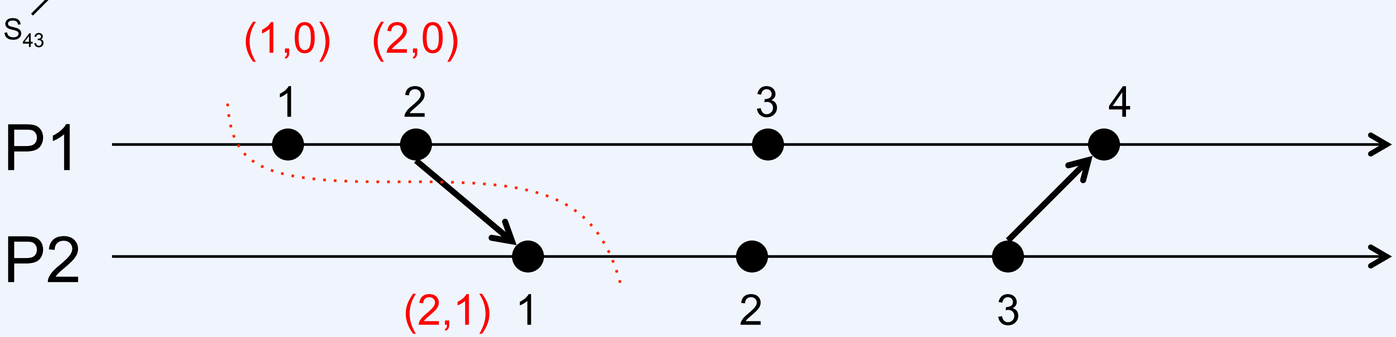


- possibly Φ:
  - Start at some initial state
  - Traverse lattice per level, until Φ is true for at least one state in the current level
- definitely Φ:
  - Start at some initial state
  - Traverse lattice per level, until Φ is true for all states in the current level

How to Obtain Lattice?

- Processes send their successive states, plus vector clocks, to central monitor process
- From vector timestamps, determine which states are reachable from each other
  - Can't violate causality
  - E.g., for $P_2 1(2,1)$ to be in the snapshot, $P_1 2(2,0)$ at least has to be in the snapshot





Deterministic Replay

Deterministic Replay

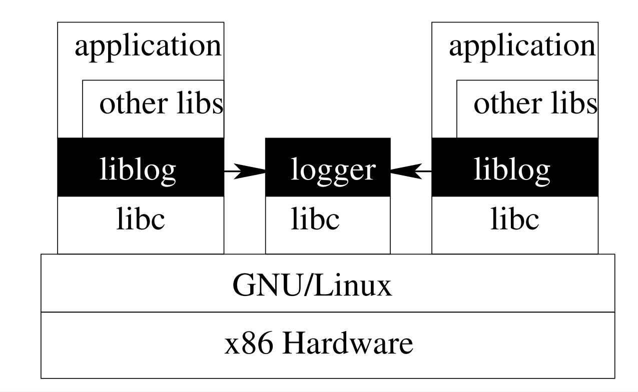
- Recording all events and maintaining vector clocks may be too expensive
  - Too many events?
  - Too many / unknown processes?
- What is the minimum that you would need to record to reproduce an execution?
- (Single process) Debugging with gdb
  - Notice an error
  - Rerun program with same inputs, set watchpoints
- What would make this not work?

Non-Determinism

- Random inputs
- External interactions
  - User input
  - Network messages
  - Interrupts, in general
- We can’t replay the entire world
- E.g., Raft makes extensive use of randomization

Deterministic Replay

- Example: liblog (Altekar et al, 2006)
- Shared library design
- Log content of all messages
  - Receiving process can be replayed independently



- Use Lamport clocks to capture a total order *(they don't need to compare arbitrary timestamps)*
- Challenge: internal concurrency
  - Must either log thread reads/writes to shared memory, or the order of their scheduling (to reproduce races)
- Checkpoints also recorded periodically

liblog Replaying

- Adapted GDB debugger
  - Run application code but replace system calls with the logged system call results
  - For multiple threads, also replace the thread scheduler to read from the recorded thread ordering
  - Console application coordinates several GDB instances, one per process
- Challenges
  - Overhead
  - Log size, checkpoint size
  - Not complete (limited vantage point)

liblog usage

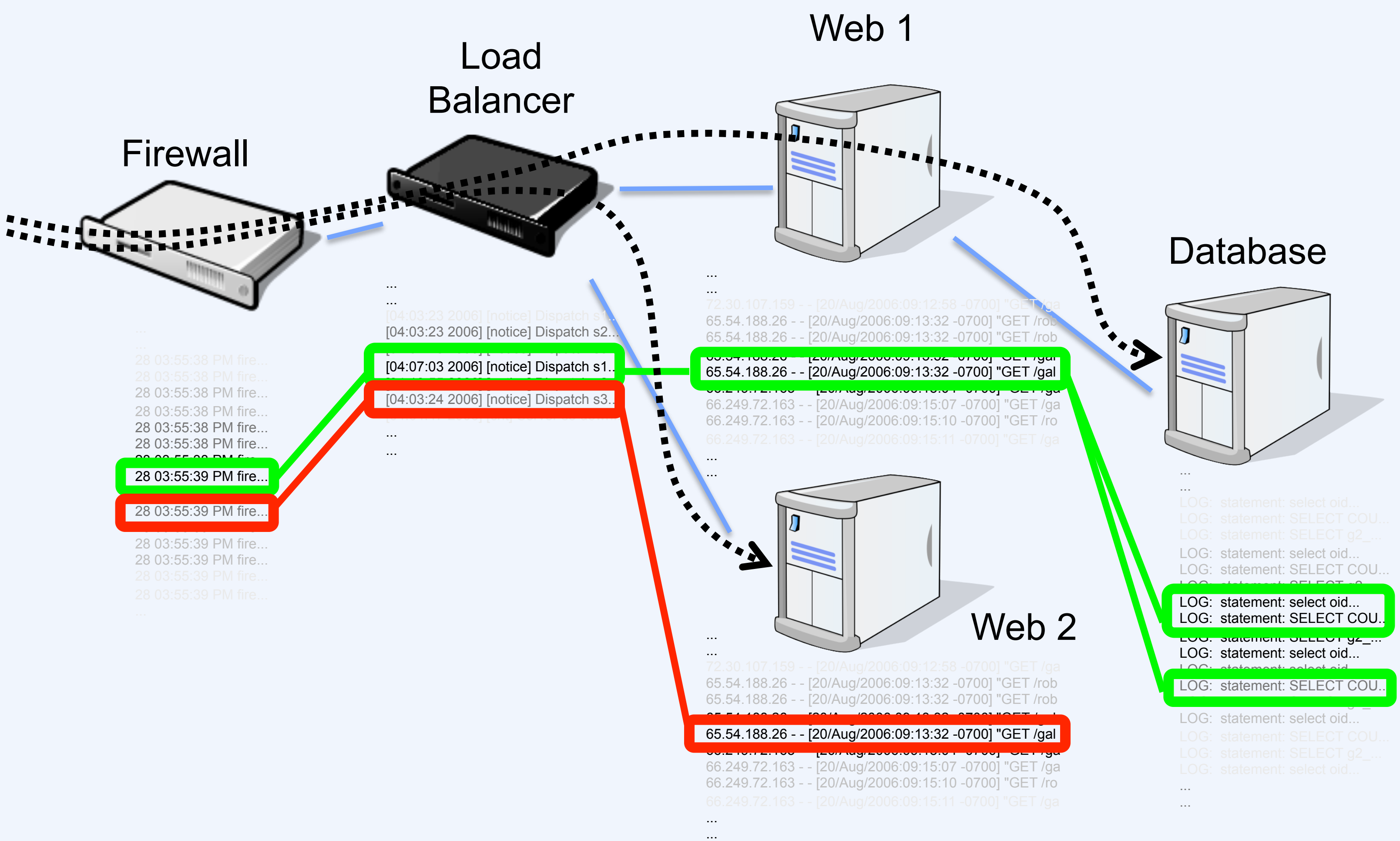
- Original authors found some bugs, erroneous assumptions
- Built a tool, friday, to specify global watchpoints
- Many other examples
  - E.g., Mugshot (Mickens, NSDI 2010) offers deterministic replay of Javascript applications
    - <7% overhead, <80KB of logs per minute
  - Deja Vu for Java (Choi and Srinivasan, '98)
  - Retrospect for MPI programs
  - Also some based on the hypervisor, e.g., XenLR and ReTrace (for VMWare)

Causal Tracing

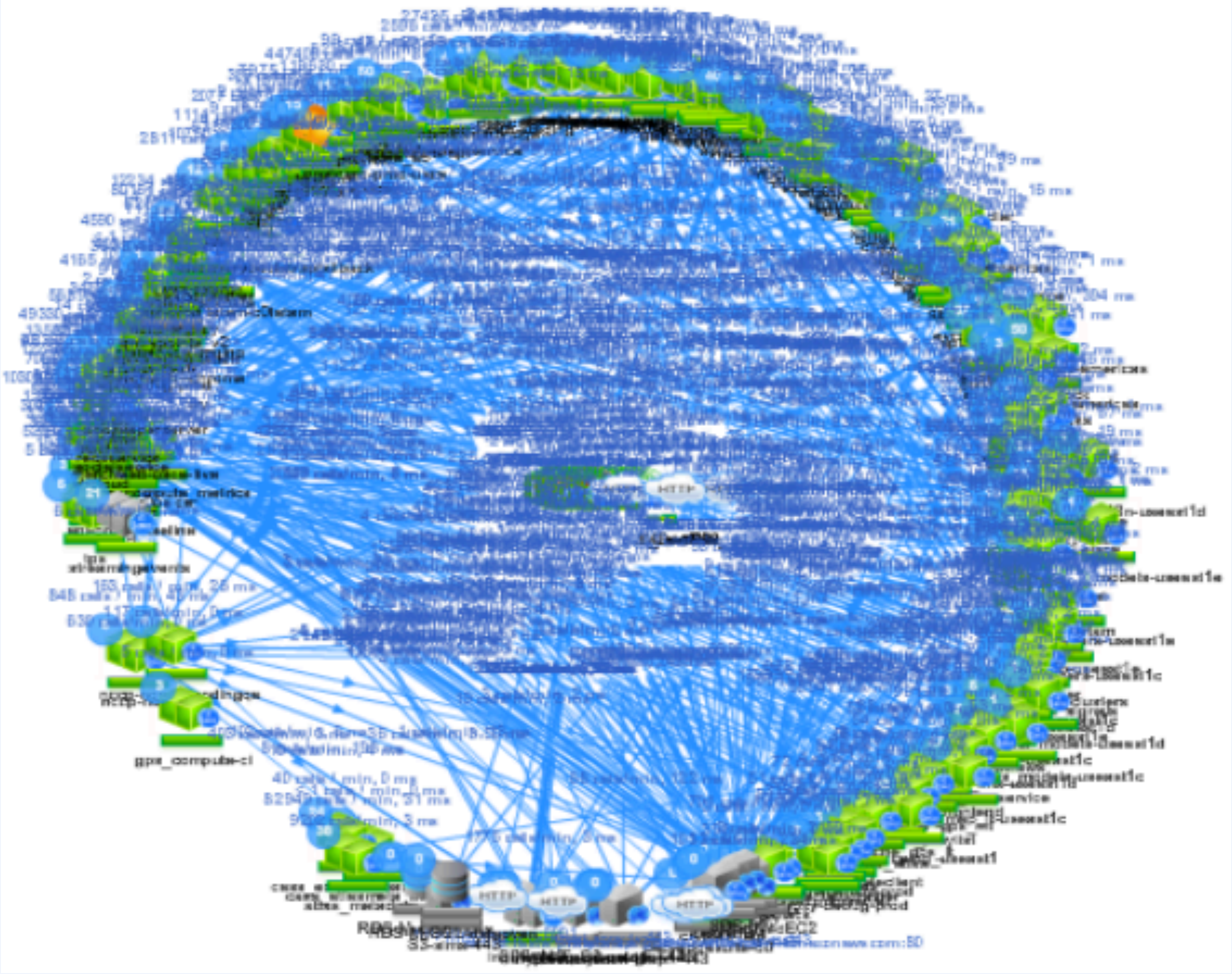
What do people usually do in practice?

- Always have some form of *local* logs
  - But…
  - Yes ☹

Status quo: device centric



This is not so bad, is it?



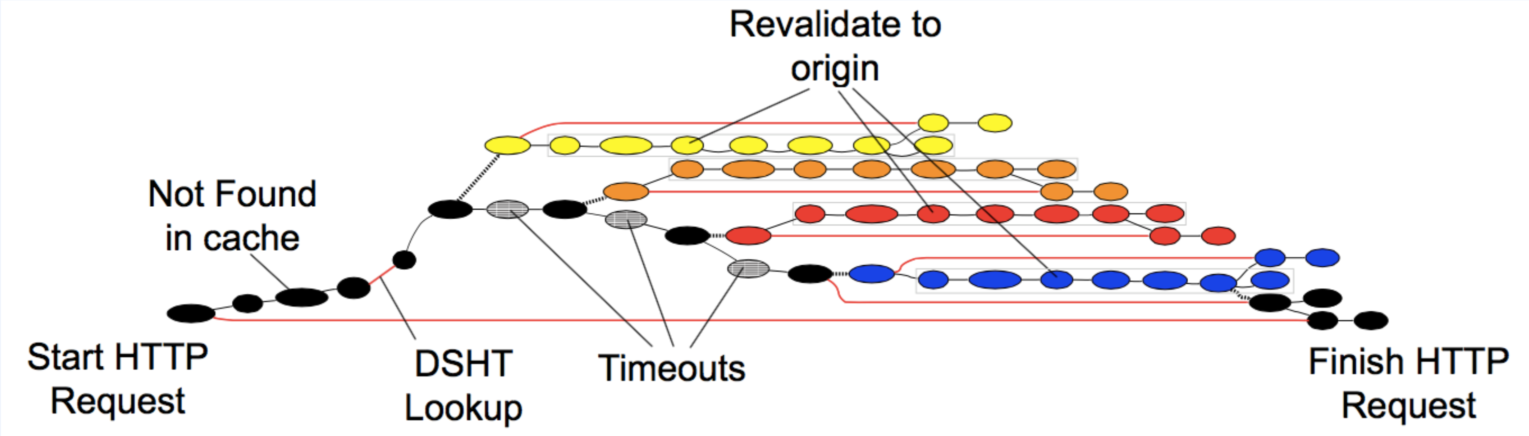
Causal Tracing

- Main idea: capture causality on well defined operations of a distributed system
  - E.g., all actions inside google when you read an email
  - Useful for debugging, performance analysis, root cause analysis of faults
- How can we reconstruct causality?
  - Want to capture (a subset of) the happens before relation, which is a DAG among events
  - Vector clocks
  - Can also directly record the graph!

Recording Causality

- Black box approaches
  - Infer causality by observing messages in and out of processes (but can get confused)
    - Project5 (Reynolds et al)
  - BorderPatrol (Koskinen and Janotti, ‘08): assumes knowledge of protocol semantics, better precision
- If application already logs enough information
  - Magpie
- If you can change or are writing application (or libraries)
  - Pip, Dapper, X-Trace, Pinpoint, …

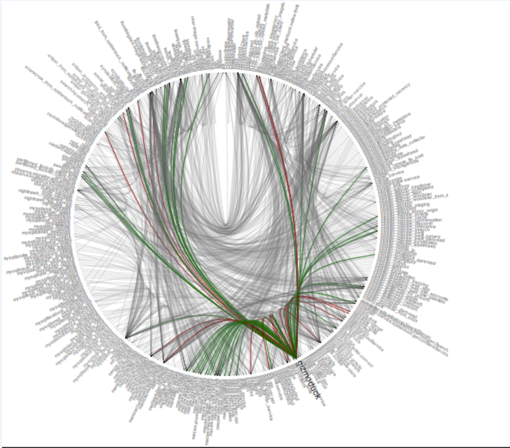
Causal Tracing

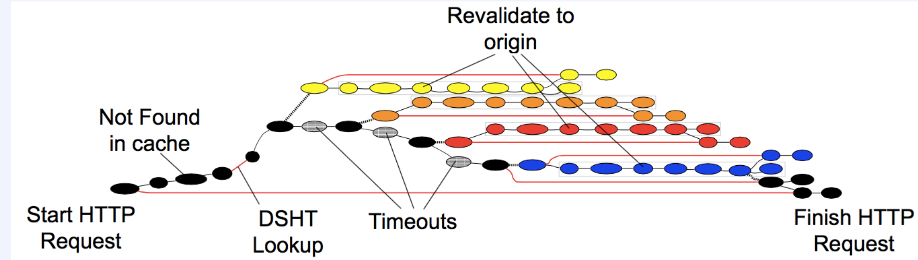


Source: X-Trace

Causal Tracing

Source: App





End-to-End Tracing

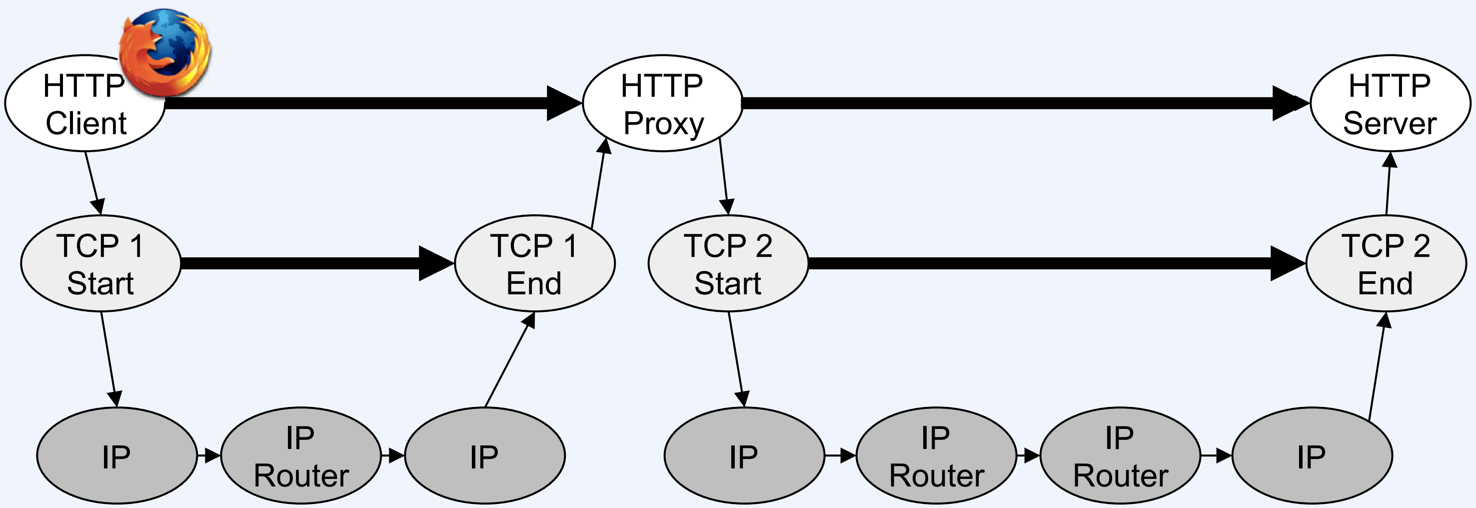
- 2002: Pinpoint
- 2004: Magpie, SDI
- 2005: Causeway
- 2006: Pip, Stardust
- 2007: X-Trace, MS ETW
- 2010: Google Dapper
- 2012: Zipkin, HTrace
- 2013: Node.js CLS
- 2014: Apple Acitivity Tracing
- 2015: Twitter, Prezi, SoundCloud, HDFS, Hbase, Accumulo, Phoenix, Google, Baidu, Netflix, Pivotal, Uber

AppNeta
AppDynamics
NewRElic

X-Trace

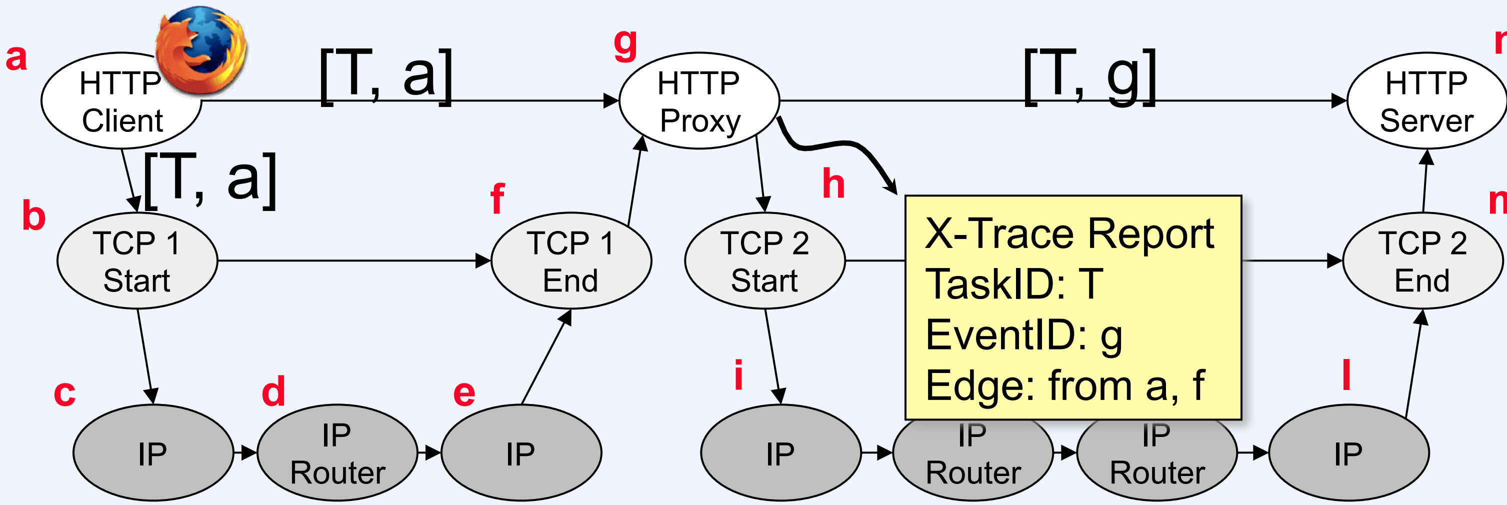
- X-Trace records **events** in a distributed execution a their causal relationship
- Events are grouped into **tasks**
  - Well defined starting event and all that is causally related
- Each event generates a **report**, binding it to one or more preceding events
- Captures full *happens-before* relation
  - For events that are part of the same task
  - Not ideal for systems where everything talks to everything all the time

X-Trace Output



- *Task graph* capturing task execution
  - Nodes: events across layers, devices
  - Edges: causal relations between events

Basic Mechanism

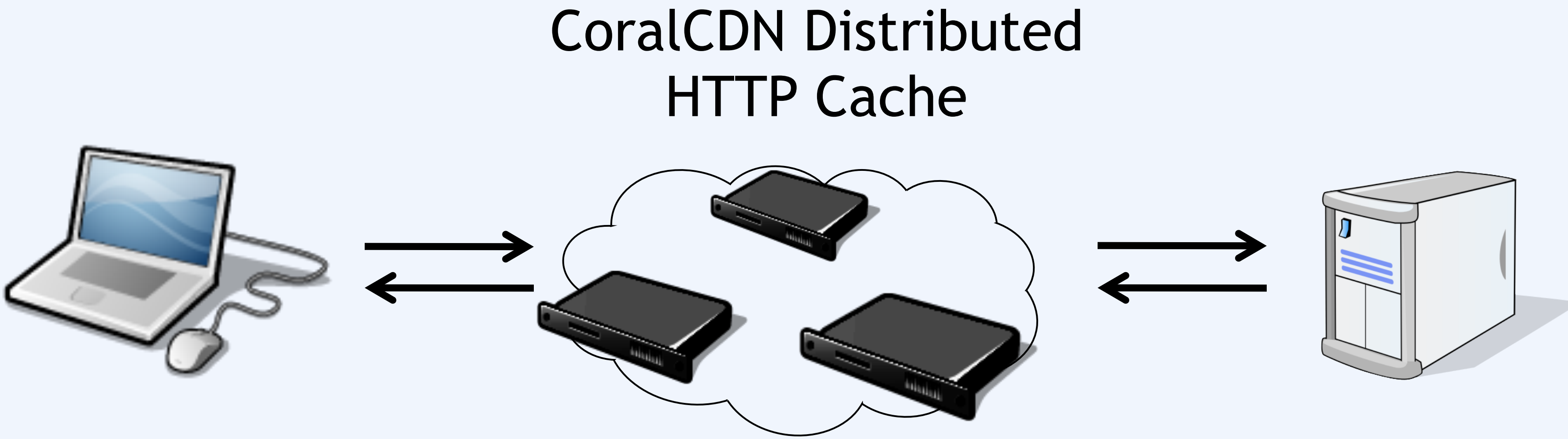


- Each event uniquely identified within a task: [***TaskId***, ***EventId***]
- [TaskId, EventId] **propagated** along execution path
- For each event create and log an X-Trace **report**
  - Enough info to reconstruct the task graph

X-Trace Library API

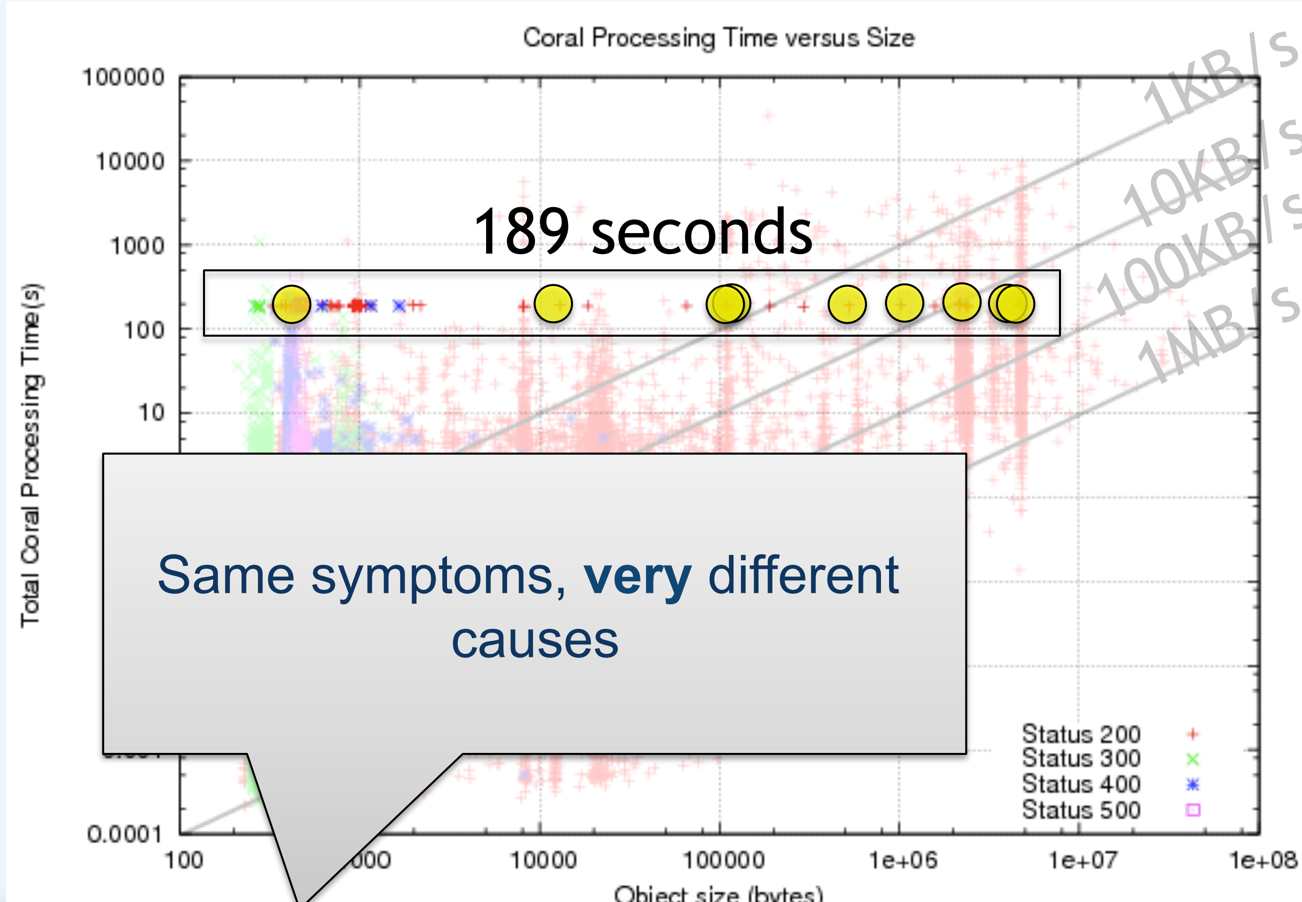
- Handles propagation within app
- Threads / event-based (*e.g.,* libasync)
- Akin to a logging API:
  - Main call is **logEvent(message)**
- Library takes care of event id creation, binding, reporting, etc
- Implementations in C++, Java, Ruby, Javascript

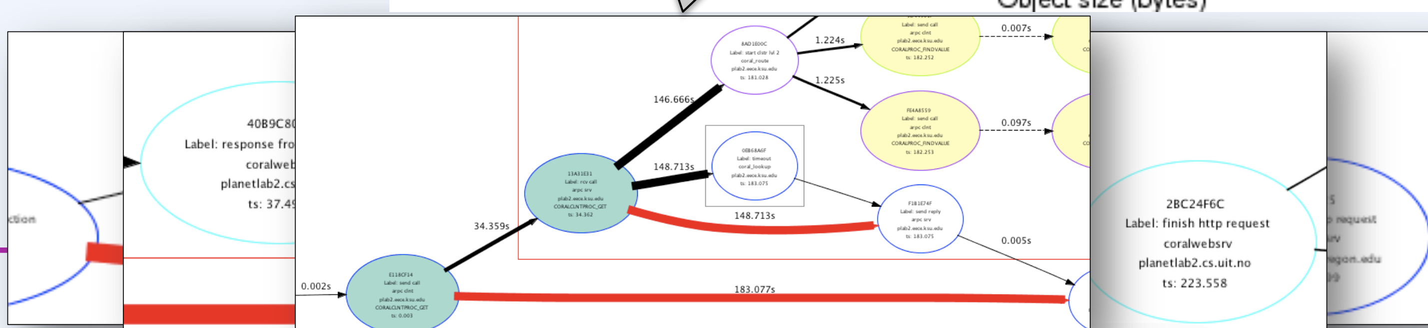
Example: CoralCDN



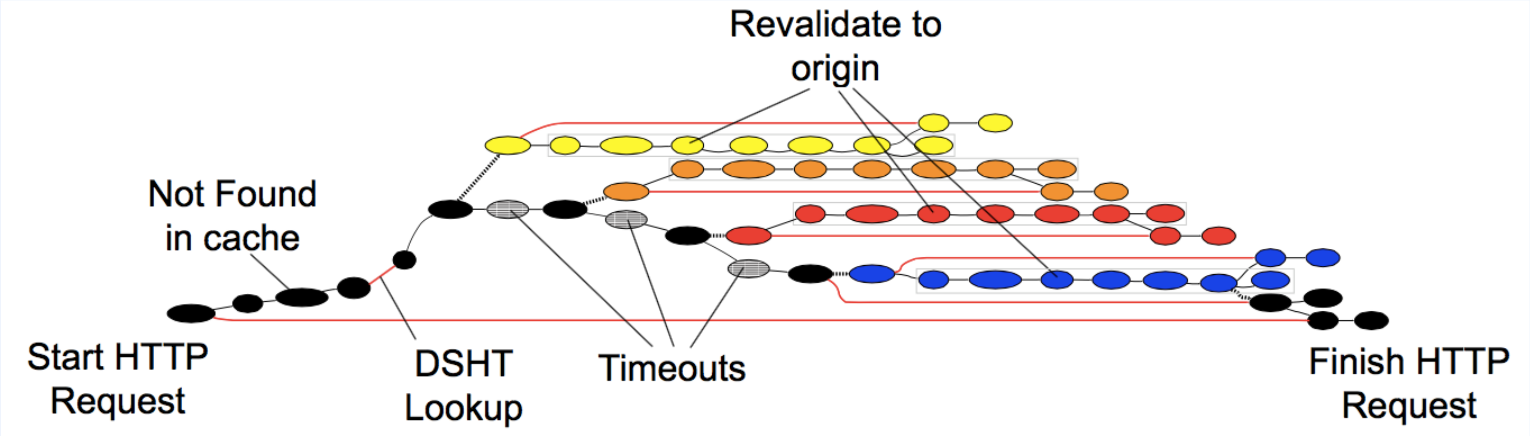
CoralCDN Response Times

- 189s: Linux TCP Timeout connecting to origin
- Slow connection Proxy -> Client
- Slow connection Origin -> Proxy
- Timeout in RPC, due to slow Planetlab node!





Critical Path

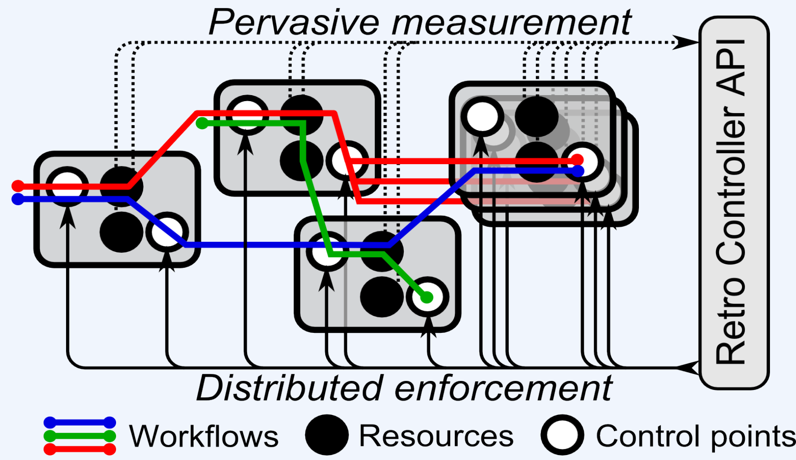


End-to-End Tracing

- Propagate metadata along with the execution*
  - Usually a request or task id
  - Plus some link to the past (forming DAG, or call chain)
- Successful
  - Debugging
  - Performance tuning
  - Profiling
  - Root-cause analysis
  - …

- Propagate metadata along with the execution

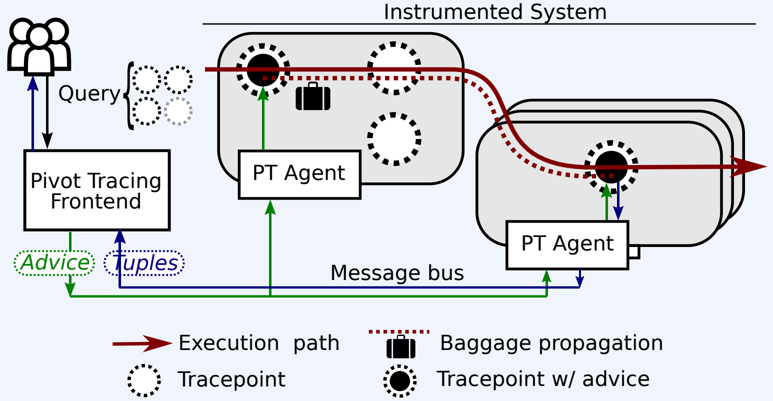
Retro



- Propagates TenantID across a system for real-time resource management
- Instrumented most of the Hadoop stack
- Allows several policies – e.g., DRF, LatencySLO
- Treats background / foreground tasks uniformly

Jonathan Mace, Peter Bodik, Madanlal Musuvathi, and Rodrigo Fonseca
. Retro: targeted resource management in multi-tenant distributed systems. In *NSDI '15*

Pivot Tracing



- Dynamic instrumentation + Causal Tracing

```
From incr In DataNodeMetrics.incrBytesRead
 Join cl In First(ClientProtocols) On cl -> incr
 GroupBy cl.procName
 Select cl.procName SUM(incr.delta)
```

- Queries → Dynamic Instrumentation → *Query-specific metadata* → Results
- Implemented generic metadata layer, which we called *baggage*

Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca.
Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. SOSP 2015

Causal Metadata Propagation

- We are currently working on a generic abstraction for causal metadata propagation to enable multiple simultaneous uses of this data