Threshold-Based Inference of Dependencies in Distributed Systems

Anshul Rastogi
Mentors: Darby Huye, Max Liu, & Dr. Raja Sambasivan
Project partner: Tanmay Gupta

Overview

01 Distributed Systems & Tracing

02 *The Mystery Machine*

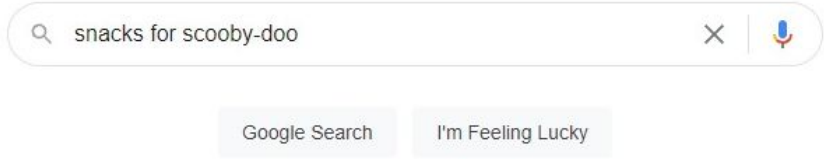
03 *Scooby Systems*

04 Future Work

Distributed systems

Distributed systems are

Networks of devices/machines ("nodes"), such as computers or servers that communicate with one another to complete tasks



Distributed systems surround us.

Some examples:

- Google
- Facebook
- Cellular networks

What is DISTRIBUTED TRACING?

Distributed tracing is when a software (instrumentation) tracks the flow of service requests in a distributed system

This is useful for:

- Debugging
- Regulation
- Analyzing performance

Gives developers information about interactions in the system.

A trace is the data about the request path that results from tracing

https://www.scaleyourapp.com/difference-between-centralized-decentralized-distributed-systems-explained/

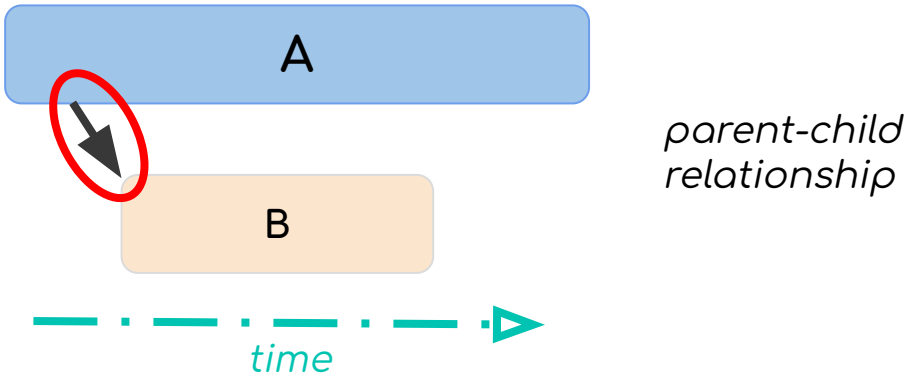
https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/36356.pdf

https://www.uplers.com/wp-content/uploads/2020/04/Top-5-Debugging-Tools-Your-Front-end-Developers-Should-Have-Hands-On.jpg

Tracing: Spans

Span (definition):

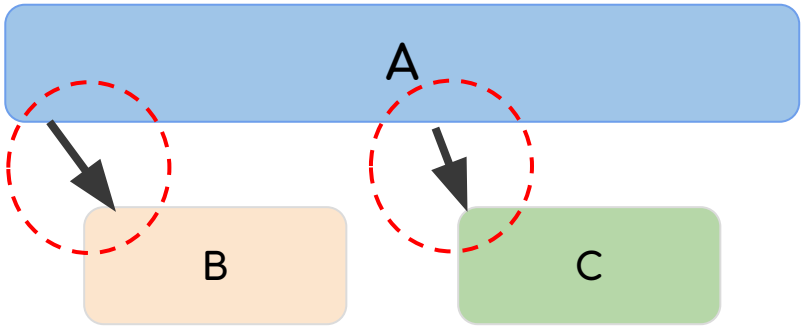
> *"The 'span' is the primary building block of a distributed trace, representing an individual unit of work done in a distributed system."*
>
> *– OpenTracing.io*



parent-child relationship

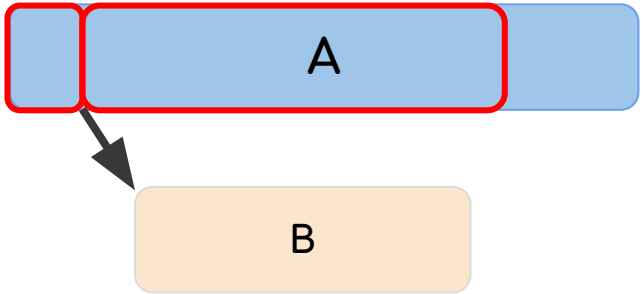
Tracing: Challenges With Spans

- Caller-callee relationships
  - B dependent on A
  - C dependent on A



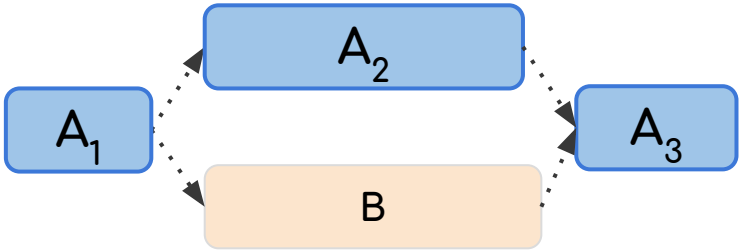
- **Is C dependent on B?**
- Instrumentation could help
- Tedious, hard to do with heterogeneous systems
- How to infer relationship with minimal instrumentation?

Tracing: Spans vs. Segments



Spans

- Many relationships between parts of A and B
  - E.g. asynchronous concurrency missed



Segments

- A_1 happens before A_2 and B; A_2 and B have no dependency on one another
- Requires more instrumentation

The Ideal Trace: Introduction

Developers use tracing to determine causality within a system.

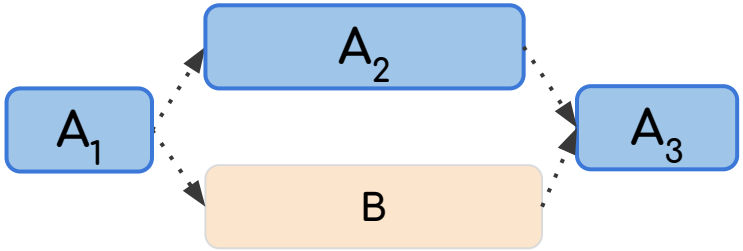
Therefore, causal relationships should be thoroughly represented throughout the entire system in a trace

To have enough information, more instrumentation can be used

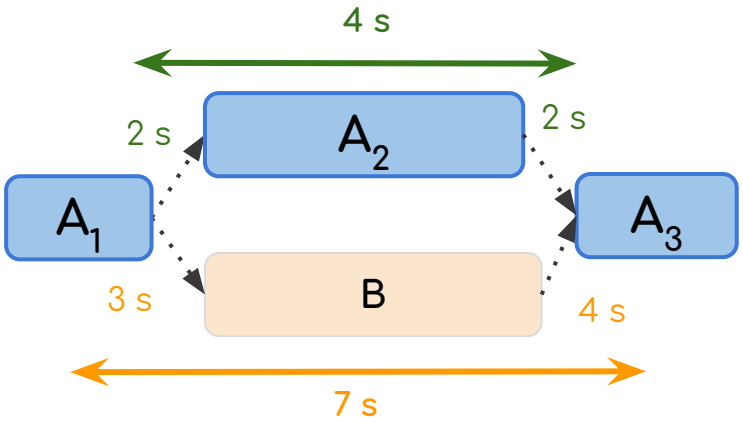
The Ideal Trace: Causality

- *Happens-before* relationship
  - A_1 must occur for B to occur

- *Concurrent* relationship
  - Segments can happen in any order or simultaneously
  - No dependency between segments
  - e.g. B and A_2



The Ideal Trace: Applications



Critical Path: longest path in trace

Determines minimum request path latency, shows amount of "slack" in system

"Slack Analysis"

Overview

01 Distributed Systems & Tracing

02 *The Mystery Machine*

03 *Scooby Systems*

04 Future Work

The Mystery Machine: GCM

The Mystery Machine produces a **Global Causal Model** (**GCM**)

- to approach the ideal trace, *The Mystery Machine* uses many traces rather than just one to infer dependencies while minimizing instrumentation
- uses a *segment-based model* – more clarity
- shows happens-before dependencies between every segment across the traces such that the dependencies hold for *every trace*
- presents inferred GCM of all system interaction to user

The Mystery Machine: Limitations

- **Problem:** Assumes enough traces to capture every interaction in request paths
- **Problem:** Assumes that all traces are correct, leaving no room for error such as:
  - Clock skew
  - Anomalies in structure (caused by bugs)

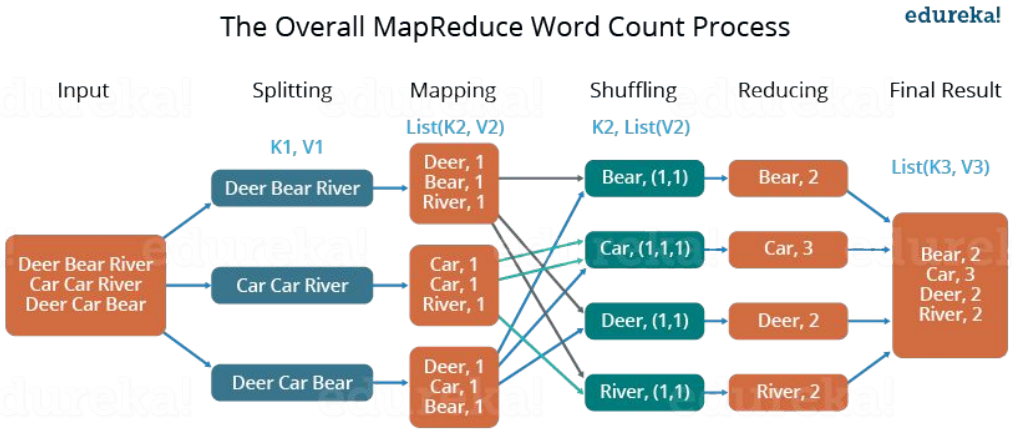
Overview

User-Defined Threshold

- Address **rigidity** of *The Mystery Machine*
  - Increase tolerance
  - Flukes should not affect GCM
- Introduce threshold – relationship is only removed from final model if violations to relationship exceeds threshold
- User has freedom to choose threshold amount

Scooby Systems: Scalability

- Implemented in Hadoop MapReduce for scalability
- MapReduce allows the program to execute efficiently across a cluster of devices
- Have had successful runs of MapReduce implementation on small "pseudo-clusters" (simulated clusters) as we don't currently have access to an actual distributed system



Overview

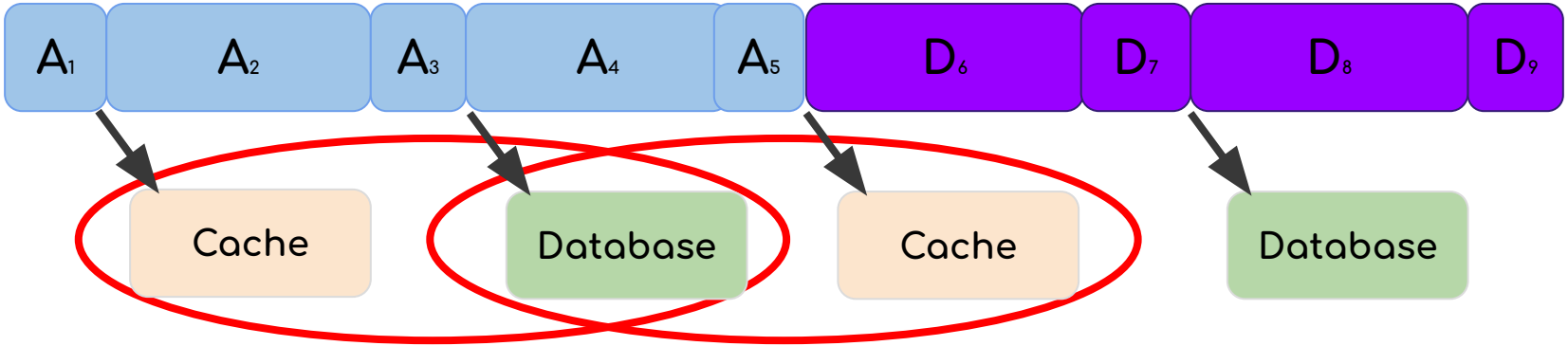
01 Distributed Systems & Tracing

02 *The Mystery Machine*

03 *Scooby Systems*

04 Future Work

Addressing Repeats



- **Problem:** *The Mystery Machine* assumes that there are no repeats of the same segment or event
- GCM concludes no dependency between B and C because *The Mystery Machine* sees both `B -> C` and `C -> B`
- But what if `B -> C` is the true structure?

Next Steps: Backtrace Depth

Without further instrumentation, need a way to distinguish repeats
Add more information:

- Currently *Scooby Systems* increases specificity in instances of a segment by associating them with the process ID
- Backtrace is more accurate – e.g. A-cache and D-cache instead of repeats of just cache
- But what if A calls cache and database twice in the same trace?
    - Could increase backtrace depth – include A's parent information
    - However, at some point, would be unable to consolidate information about cache

Moving Forward

- Implement backtrace information
- Proposing solutions / further analysis of *The Mystery Machine* limitations
- Evaluation
  - Traces from DeathStarBench

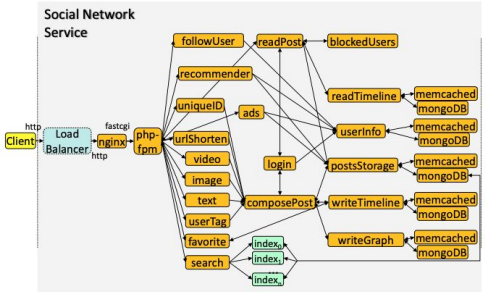


Figure 4. The architecture (microservices dependency graph) of *Social Network*.

ACKNOWLEDGEMENTS

Thanks to **Darby Huye**, **Max Liu**, **Raja Sambasivan**, & D.O.C.C. Lab for their guidance

Thank you to **Tanmay Gupta** for being my PRIMES partner for the first iteration of *Scooby Systems* in 2021.

Thanks to the **PRIMES Program** for providing this opportunity


THANK YOU FOR YOUR ATTENTION
ANY QUESTIONS?
makeameme.org