SPRINGONE2GX
WASHINGTON, DC

Call Tracing

Monish Unni
monish.unni@gmail.com





About Me

Monish Unni
Programmer | Architect
Java | Spring | Scala | NodeJS

Contents

- Background
- Use case
- Core Concepts
- Demo
- Minimal Tracing
- Zipkin Tracing
- Demo
- Production Deployment
- Implementing suggestions
- Exceptions to the model
- Lessons Learnt

Background

- In the beginning we had
  - Clearly specified contracts and roles
  - Well known topology and flow
    - web servers, application servers, services, databases
  - Prescheduled down times
  - Stacktraces were good enough
    - for the most part.

Background

- Today we have microservices that are a
  - Collection of software modules
  - Developed by different teams
  - Different programming languages
  - Span many machines
  - Multiple physical facilities

Background

- Developers have checked in and tested code
- SRE monitors services, queues and databases
  - queue surges
  - database tracing
  - disk corruption
  - load re-balancing

Background

- SRE needs to
  - shutdown and redeploy services
  - map a service to other services
  - map a service to database/tables
  - figure out weak links in a system
  - figure out critical paths

Background

- SRE needs to
  - debug the system
  - figure out where the bottlenecks are?
  - need to determine if request paths are correct?
  - correlate call tree with system behavior.

- In April 2010, Google published a research paper called as “Dapper”

Modern Internet services are often implemented as complex, large-scale distributed systems. These applications are constructed from collections of software modules that may be developed by different teams, perhaps in different programming languages, and could span many thousands of machines across multiple physical facilities.

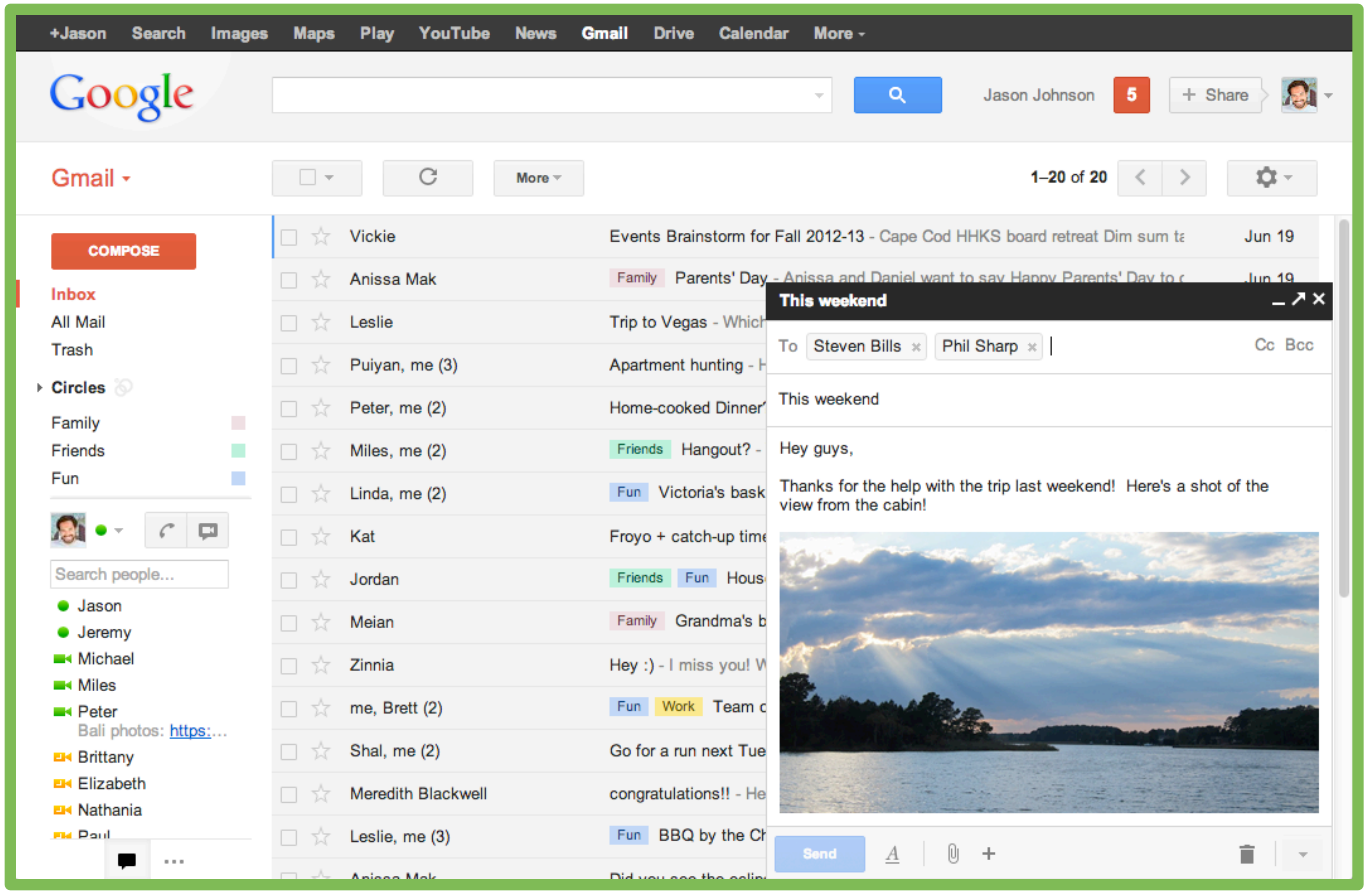
-- Google Dapper Paper

- Using an in-house dapper-like framework, we are able to layer the request demand through the aggregate infrastructure onto a simple visualization
  - Netflix Feb 2015[1]

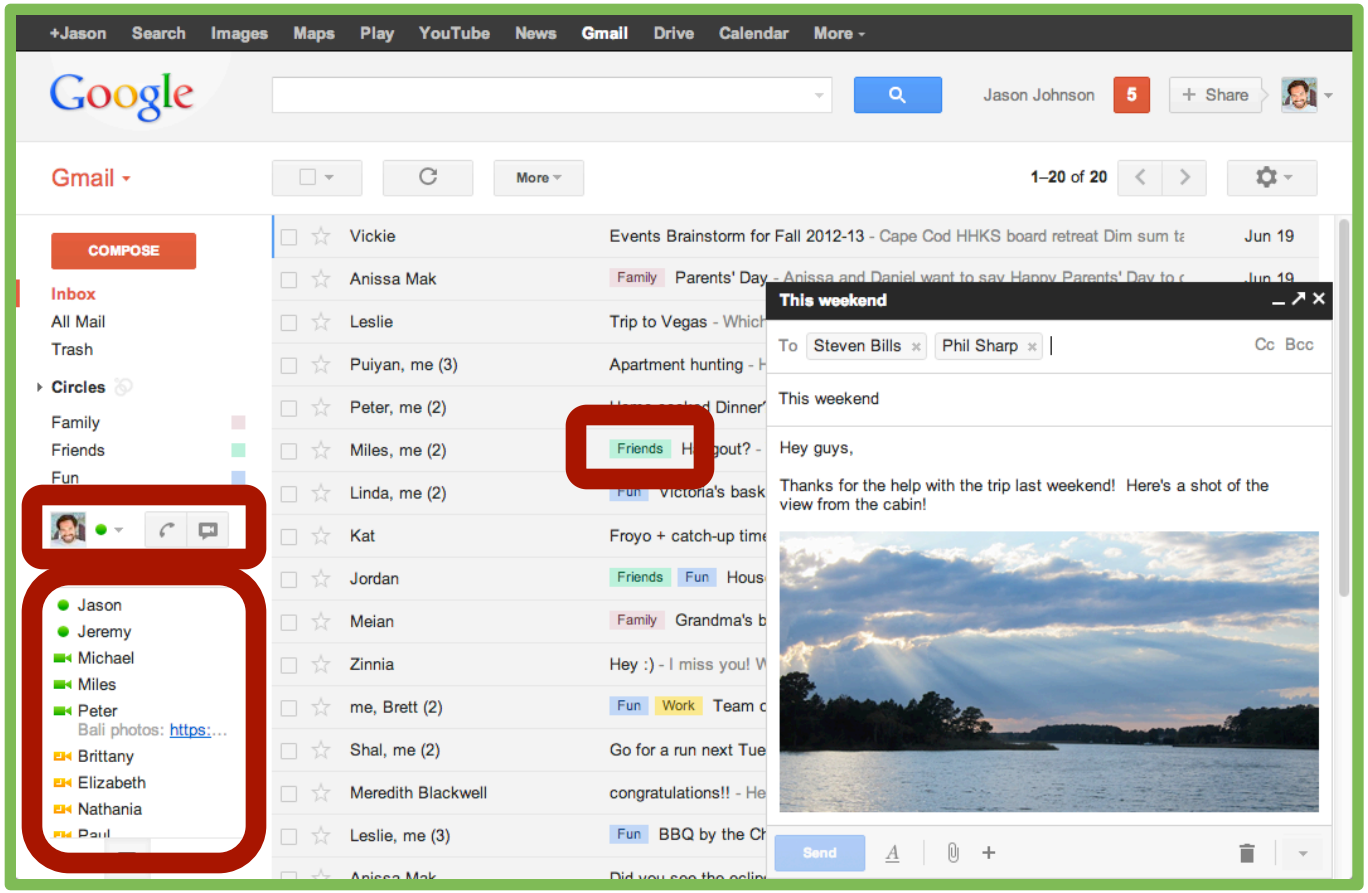
- ..help us gather timing data for all the disparate services involved in managing a request to the Twitter API...
  - Twitter June 2012[2]

- Real-time distributed tracing and call graphs have proven to be indispensable for insight into the service performance of web applications, and the associated cost across the entire technology stack at LinkedIn.
  - LinkedIn Feb 2015[3]
- Talk on “Call Tracing”
  - SpringOne2GX Sept 2015

[1] http://techblog.netflix.com/2015_02_01_archive.html
[2] https://blog.twitter.com/2012/distributed-systems-tracing-with-zipkin
[3] https://engineering.linkedin.com/distributed-service-call-graph/real-time-distributed-tracing-website-performance-and-efficiency



Services ?



Services (guess) .

Contacts –

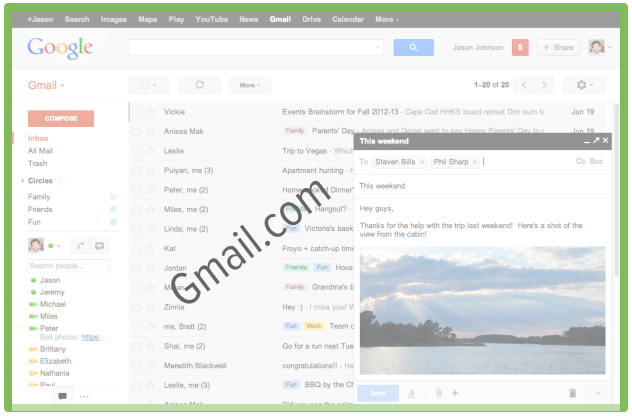
Presence –

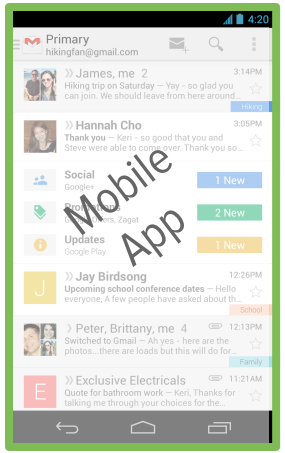
Circles –

Hangout audio –

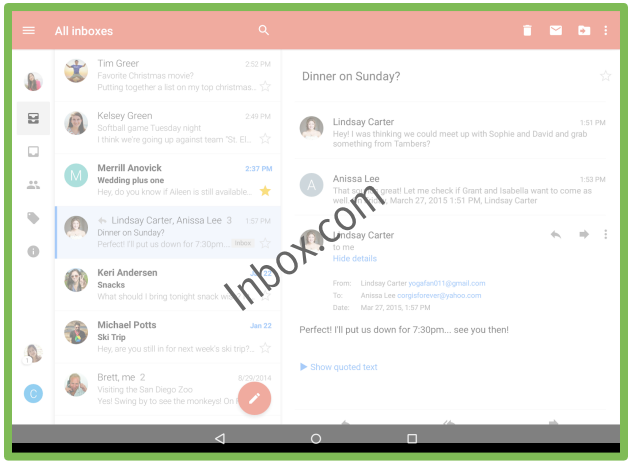
Hangout video –

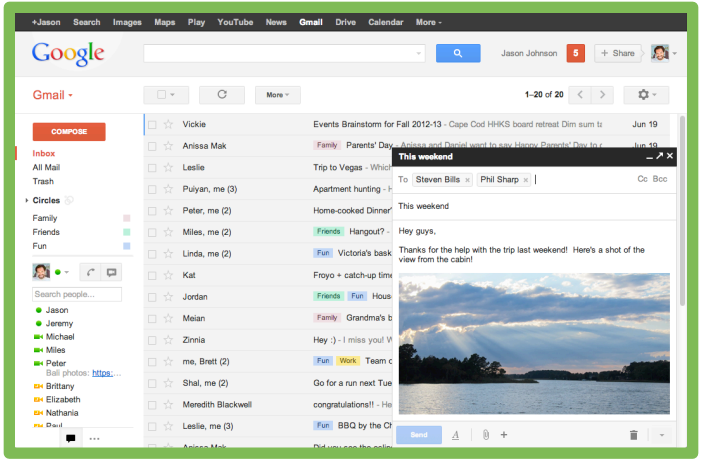
Labels/Tags –

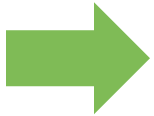


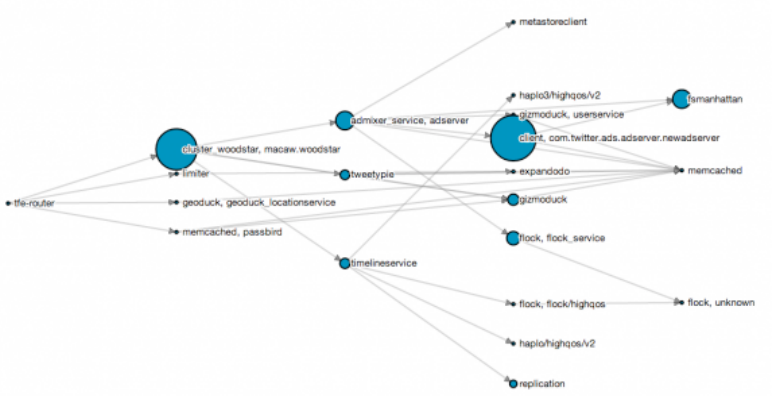


Presence Service



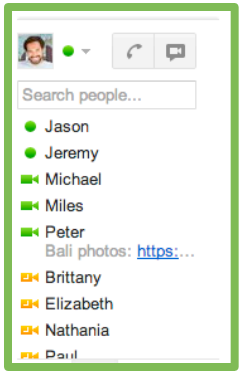




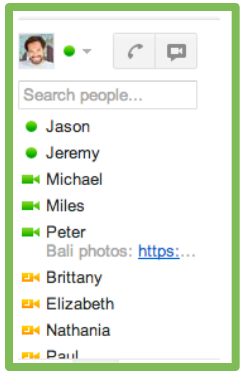


A single request can run across 100s/1000s of machines/services and involve hundreds of different subsystems.

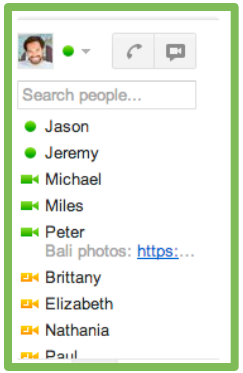
Call Tracing Concepts



Contacts → Presence → Session

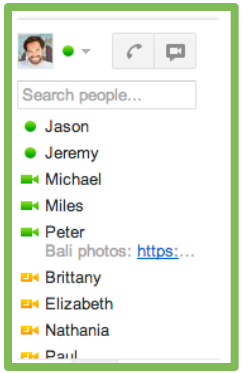


Contacts → Presence → Session

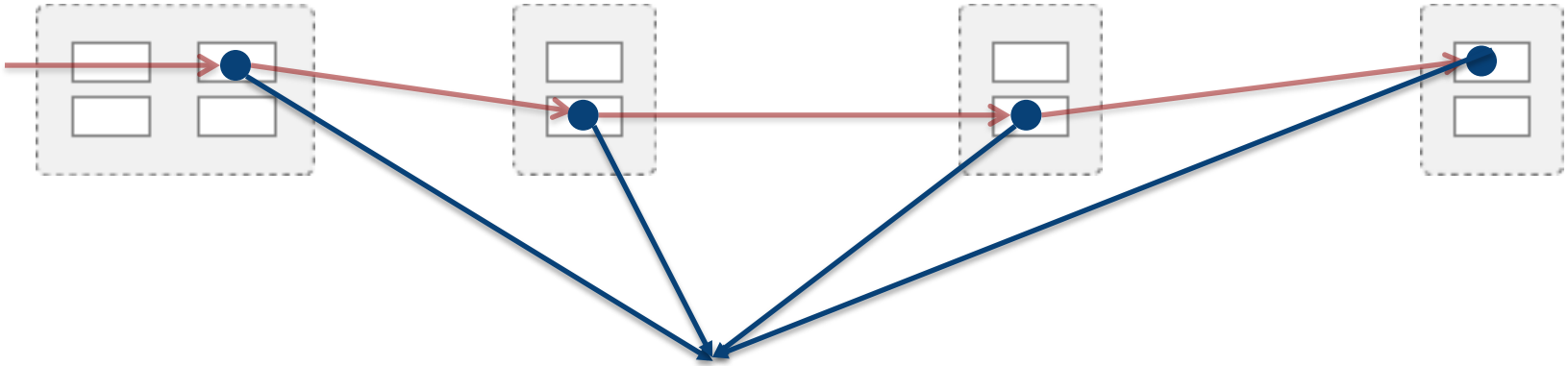


Contacts → Presence → Session

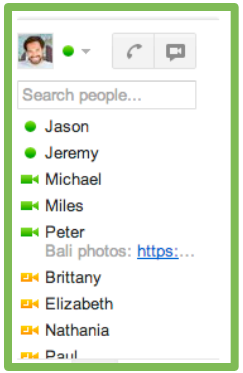
TraceId or GUID



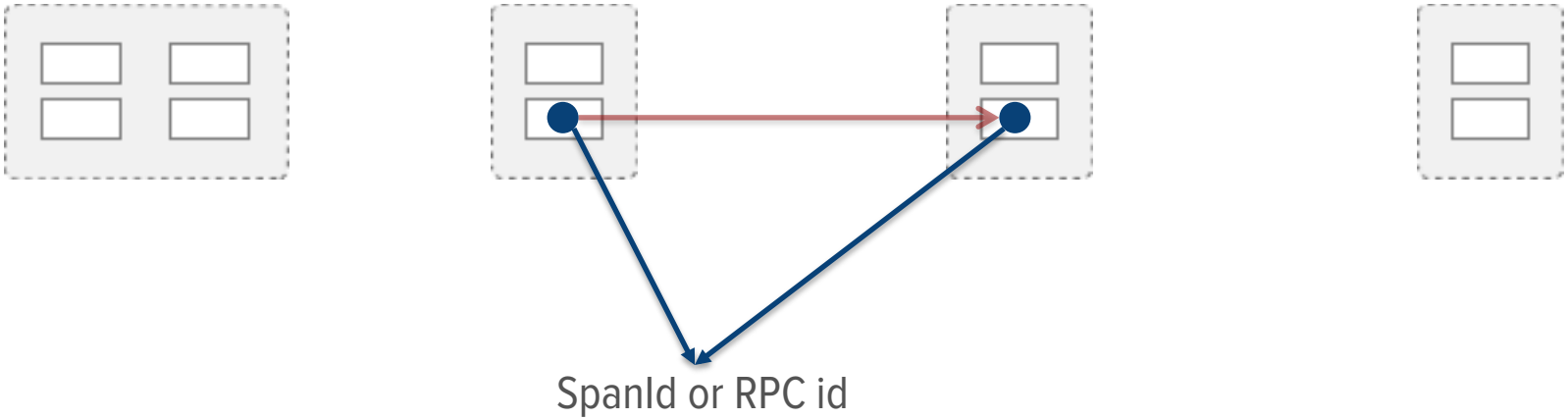
Contacts → Presence → Session



TraceId or GUID



Contacts → Presence → Session

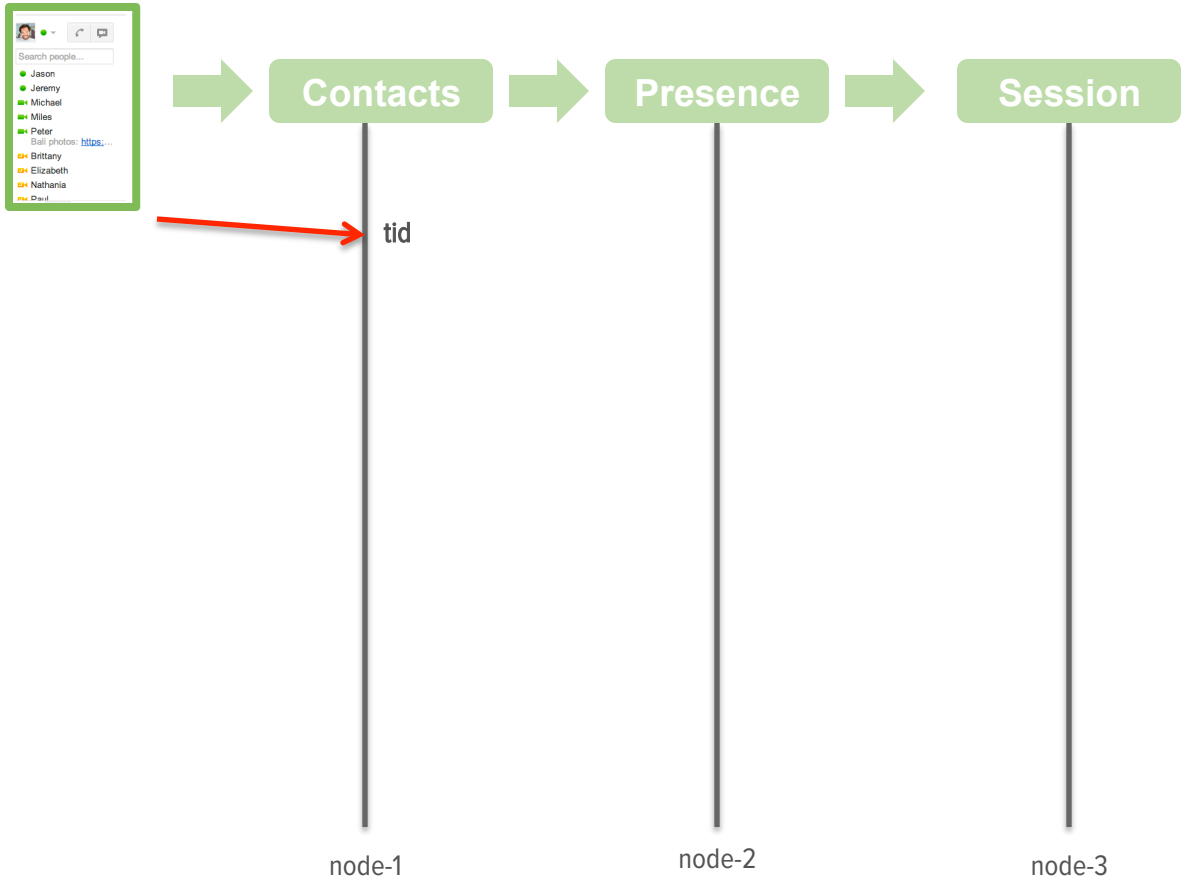


Review

- TraceId
  - a globally unique immutable Id
  - correlates to a request
- SpanId
  - a unique id for an RPC

Demo

- Start two services
  - server_a, server_b
- client calls server_a
  - server_a calls server_b
- show trace UX



Goal

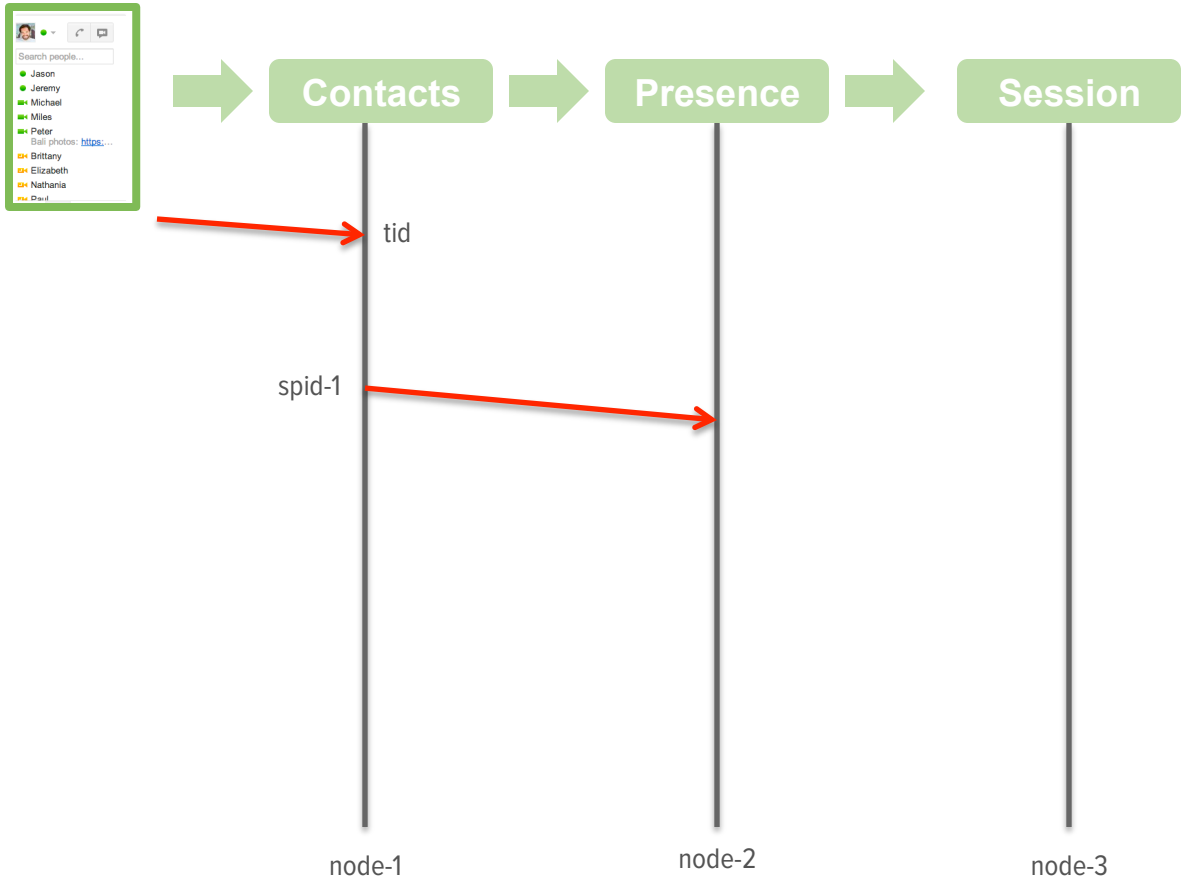
Stitch call tree from local logs

Node Log View

node-1: tid

node-2:

node-3:



Goal

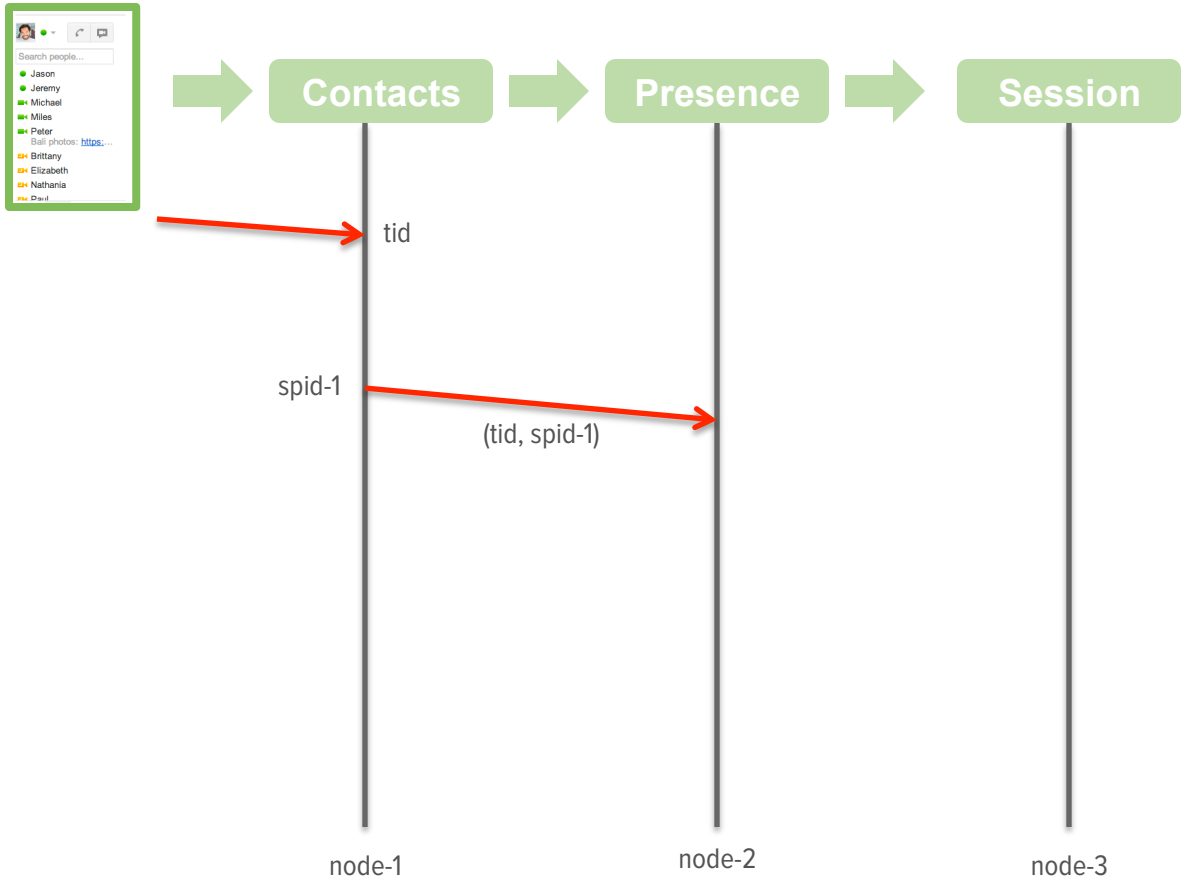
Stitch call tree from local logs

Node Log View

node-1: tid, spid-1

node-2:

node-3:



Goal

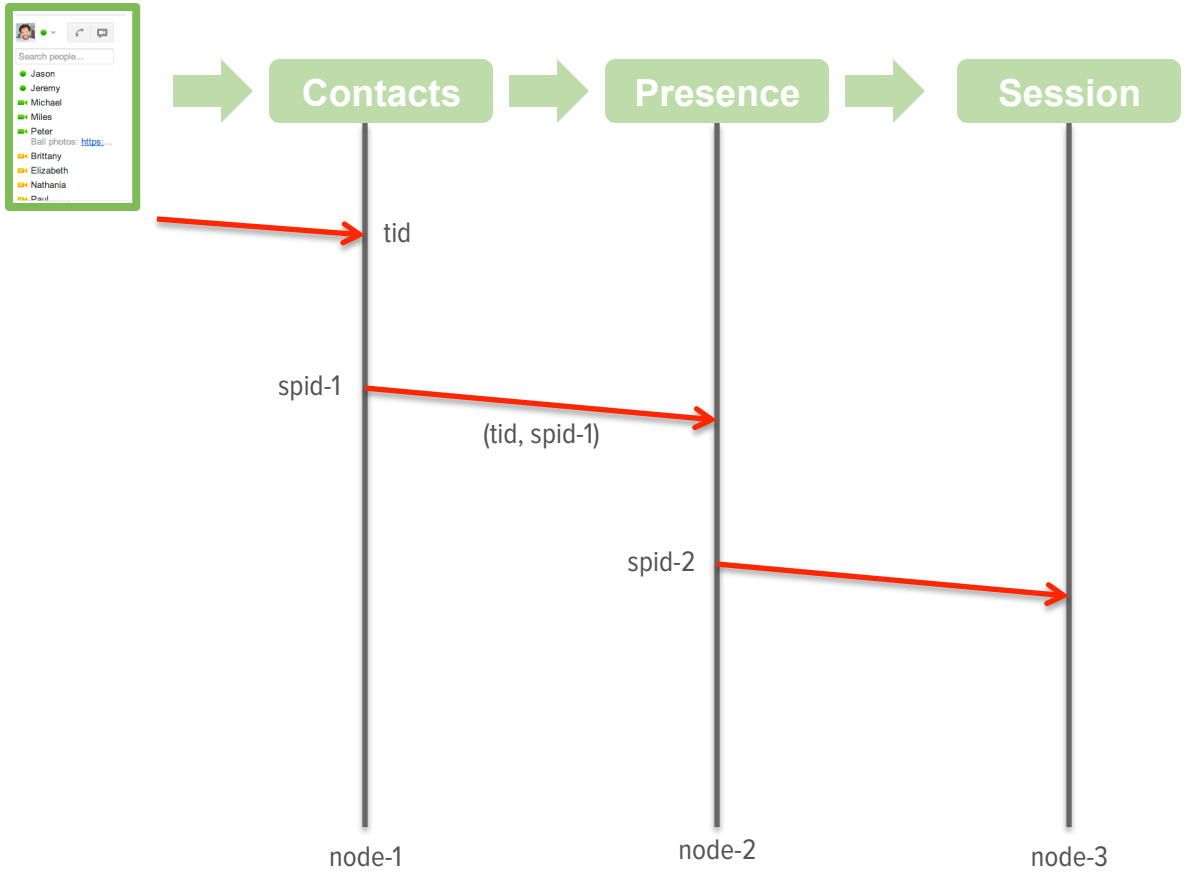
Stitch call tree from local logs

Node Log View

node-1: tid, spid-1

node-2: tid, spid-1

node-3:



Goal

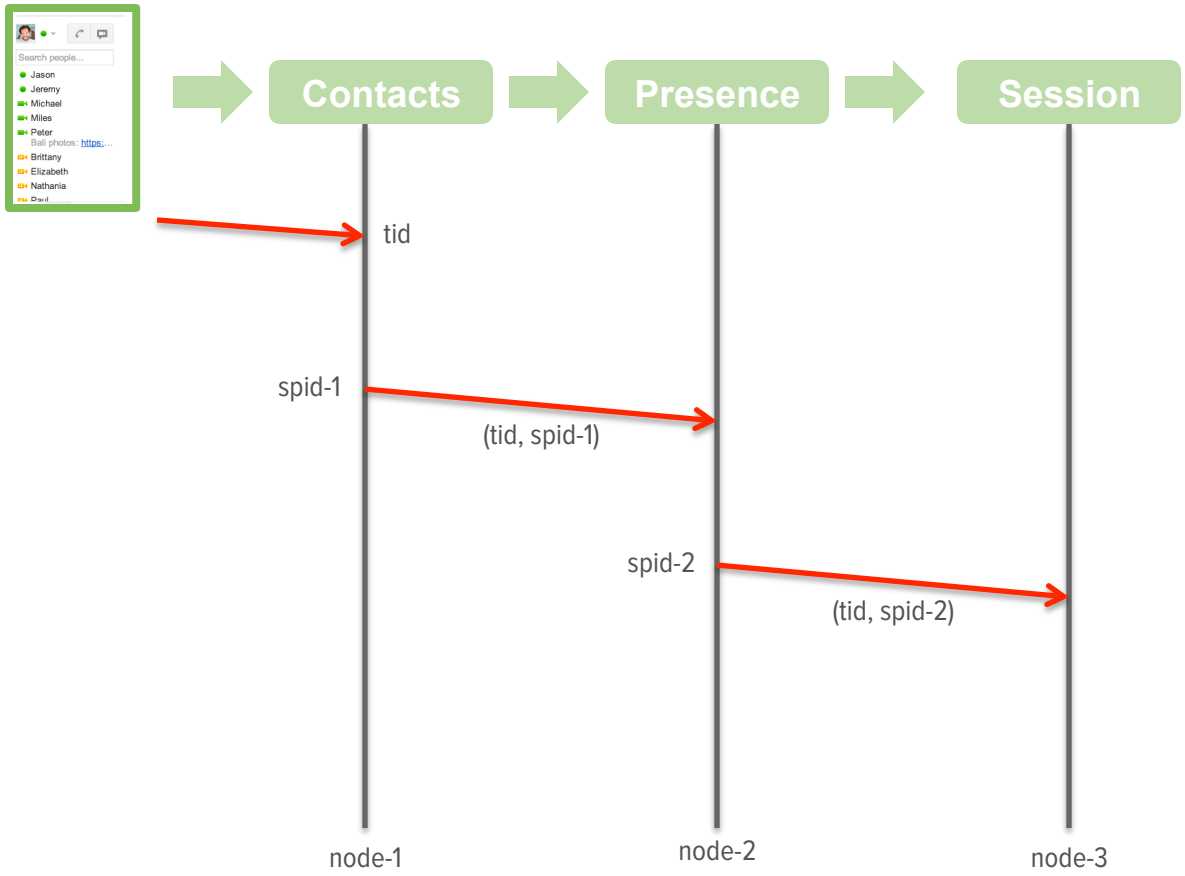
Stitch call tree from local logs

Node Log View

node-1: tid, spid-1

node-2: tid, spid-1, spid-2

node-3:



Goal

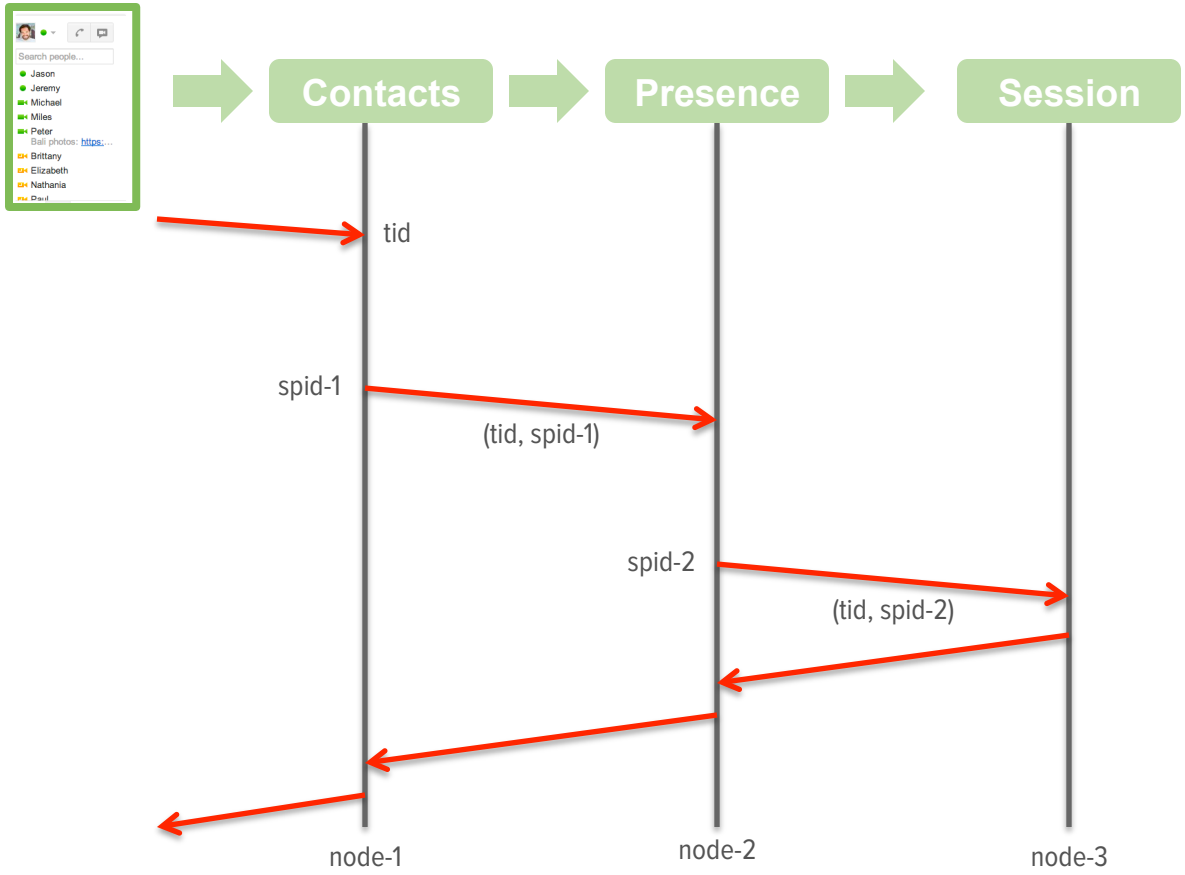
Stitch call tree from local logs

Node Log View

node-1: tid, spid-1

node-2: tid, spid-1, spid-2

node-3: tid, spid-2



Goal

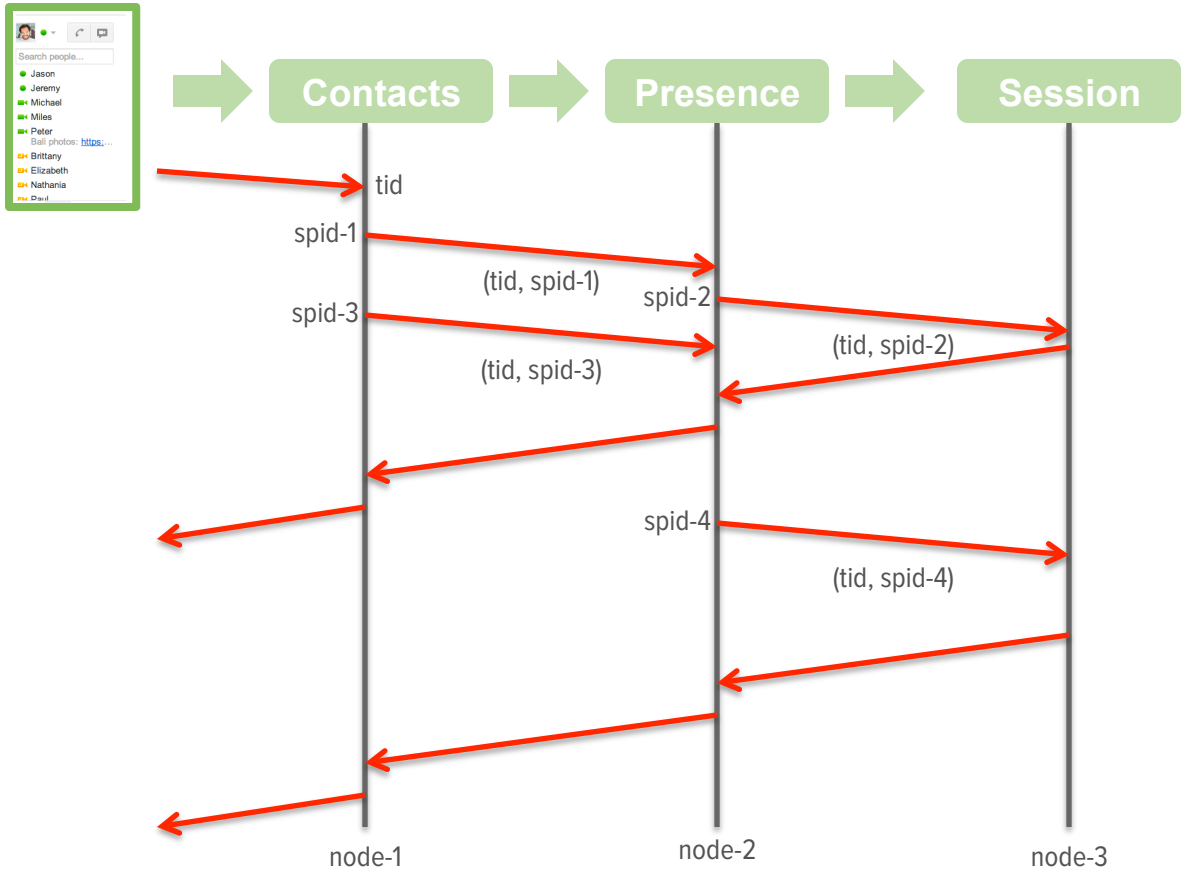
Stitch call tree from local logs

Node Log View

node-1: tid, spid-1

node-2: tid, spid-1, spid-2

node-3: tid, spid-2



Goal

Stitch call tree from local logs

Node Log View

node-1: tid, spid-1, spid-3

node-2: tid, spid-1, spid-2, spid-3

node-3: tid, spid-2, spid-4

Limit

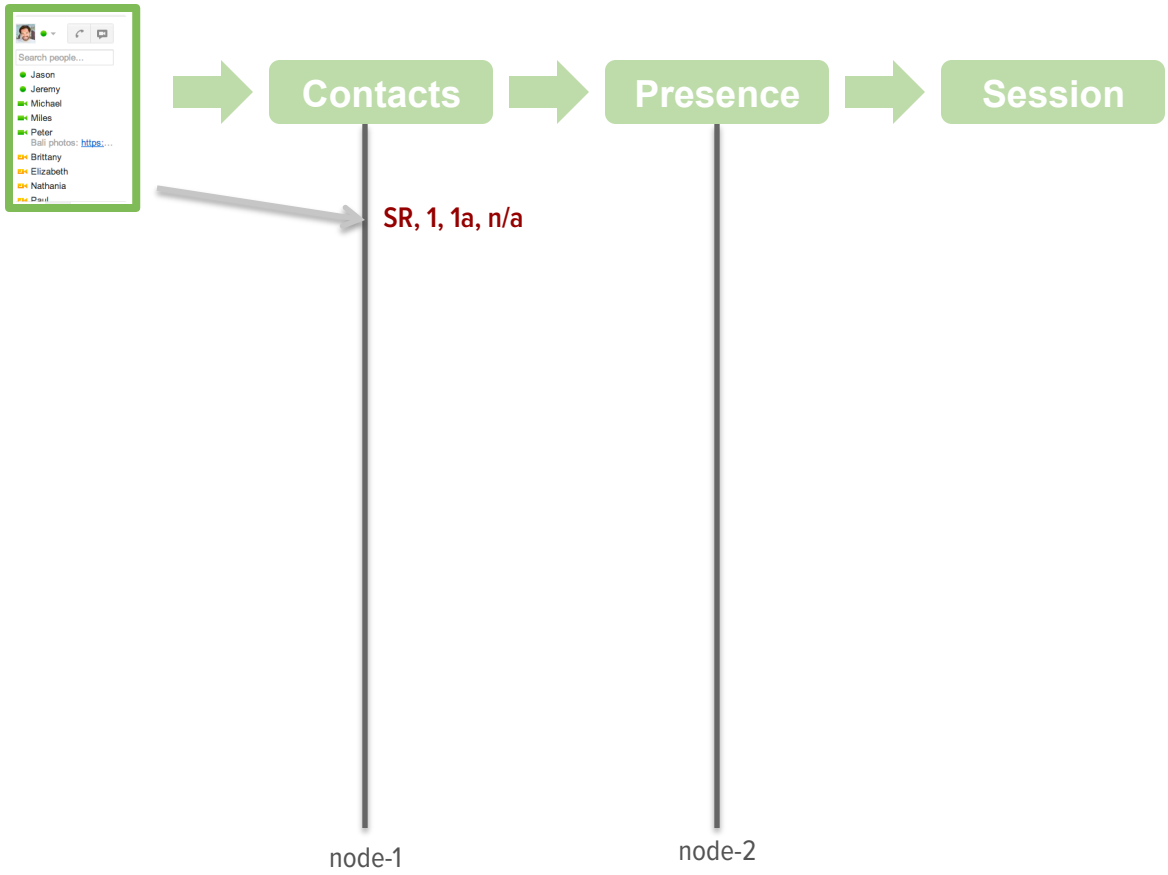
- **can correlate all requests**
- **cannot stitch call tree from logs**

Review

- Black-box instrumentation
  - Pros
    - Application level transparency
    - TraceId, SpanId, Timestamp allow correlation of requests
  - Cons
    - Request Paths or call trees require
      - reasoning based on statistical inference
      - derivation of causal relationships

Uniform Logging Structure

- TraceId, SpanId, ParentId, Event +Timestamp
- Four key events with timestamps
  - Server Receive/Send (SR/SS)
    - duration for a service call
  - Client Send/Receive (CS/CR)
    - duration for a client call
- ParentId
  - Optional – root span has none
  - Previous SpanId



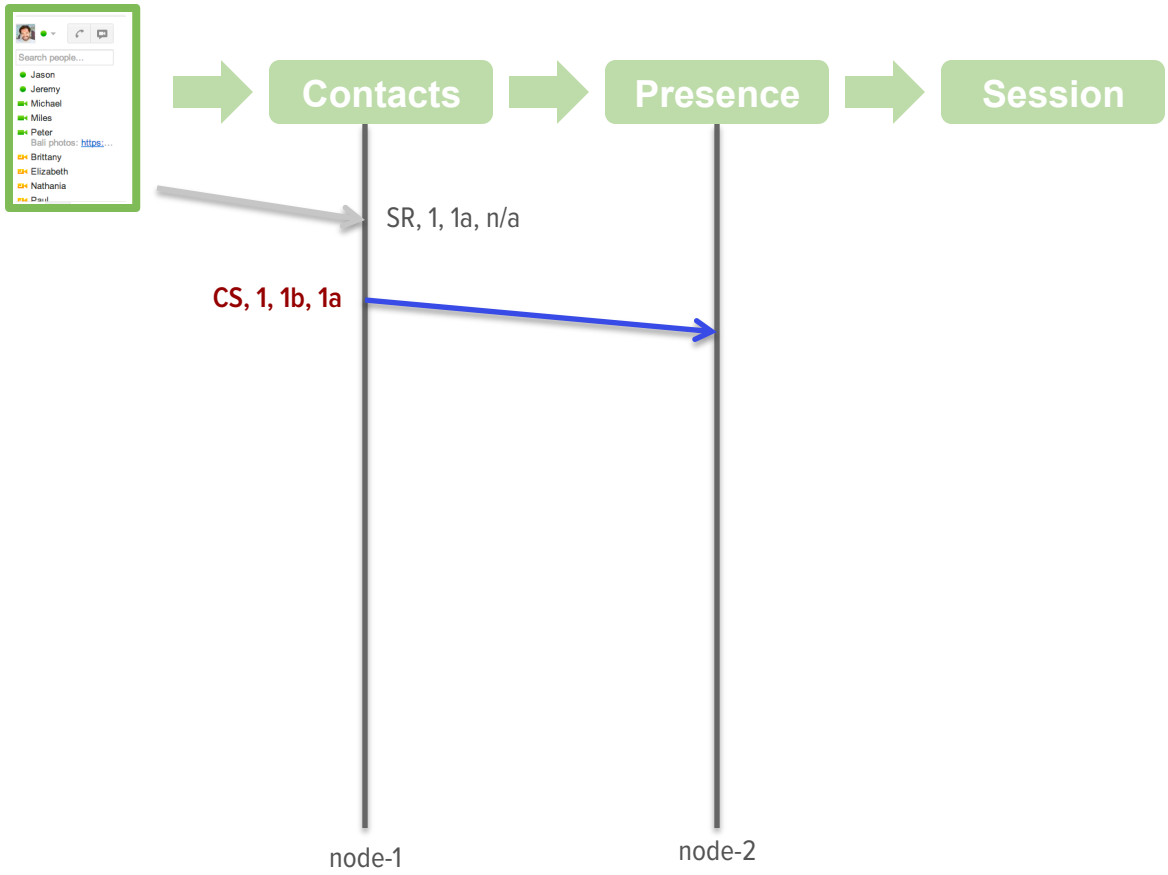
Structure

Event: Server Received (SR)

TraceId: 1

SpanId: 1a

ParentId : N/A



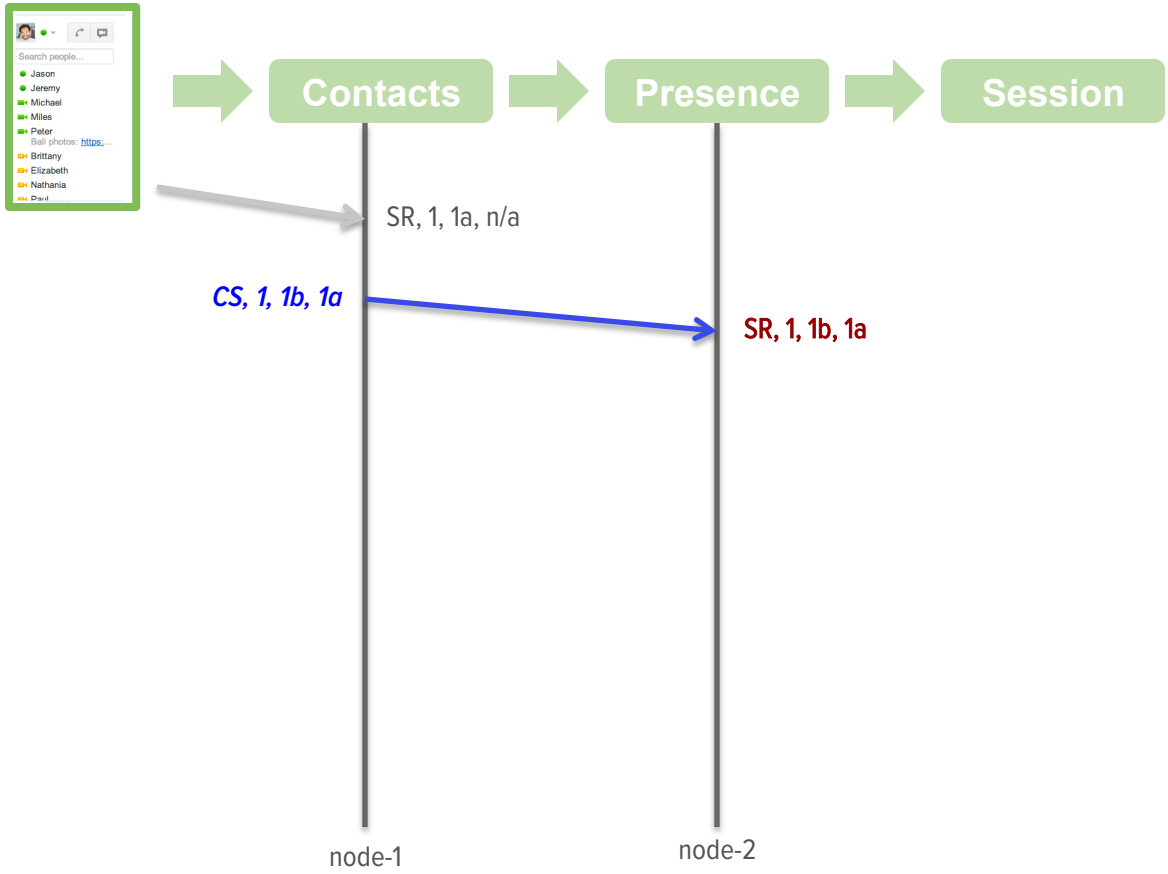
Structure

Event: Client Sent (CS)

TraceId: 1

SpanId: 1b

ParentId : 1a



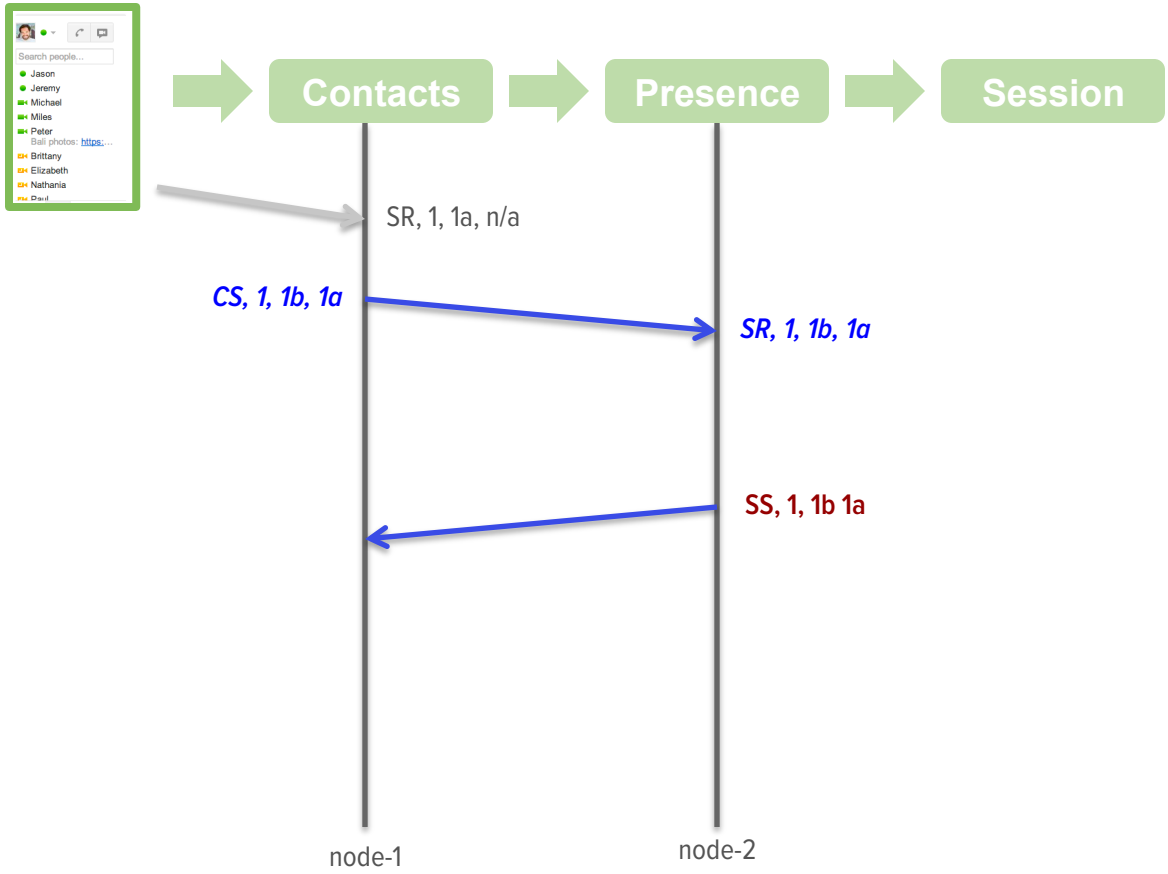
Structure

Event: Server Received(SR)

TraceId: 1

SpanId: 1b

ParentId : 1a



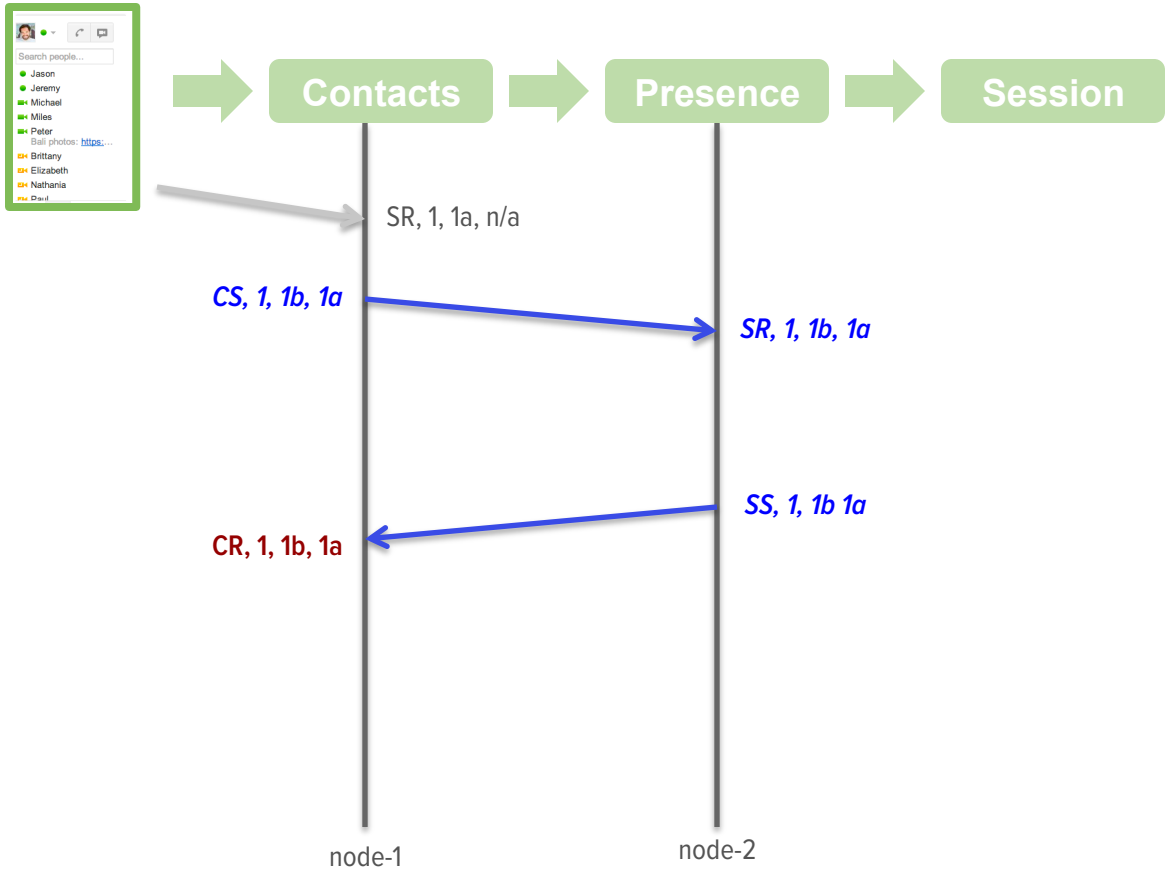
Structure

Event: Server Sent(SS)

TraceId: 1

SpanId: 1b

ParentId : 1a



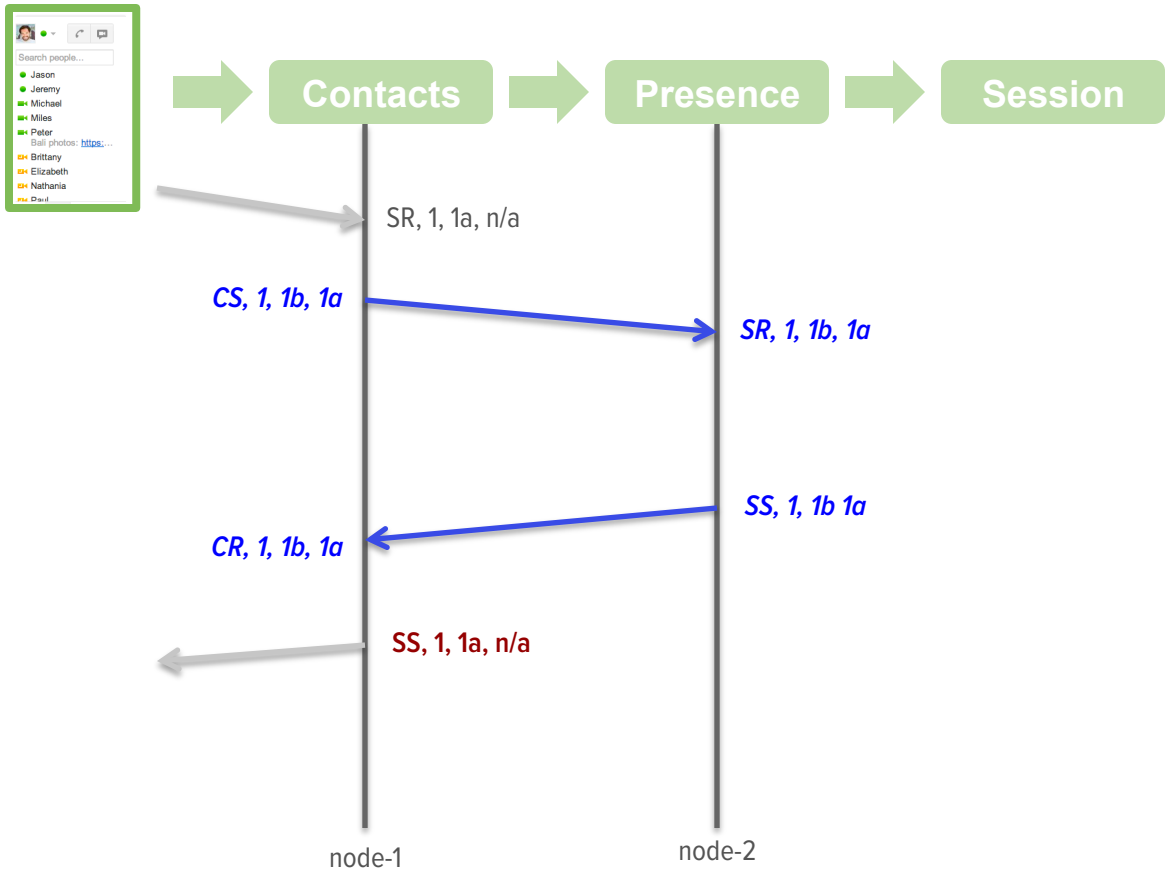
Structure

Event: Client Received(CR)

TraceId: 1

SpanId: 1b

ParentId : 1a



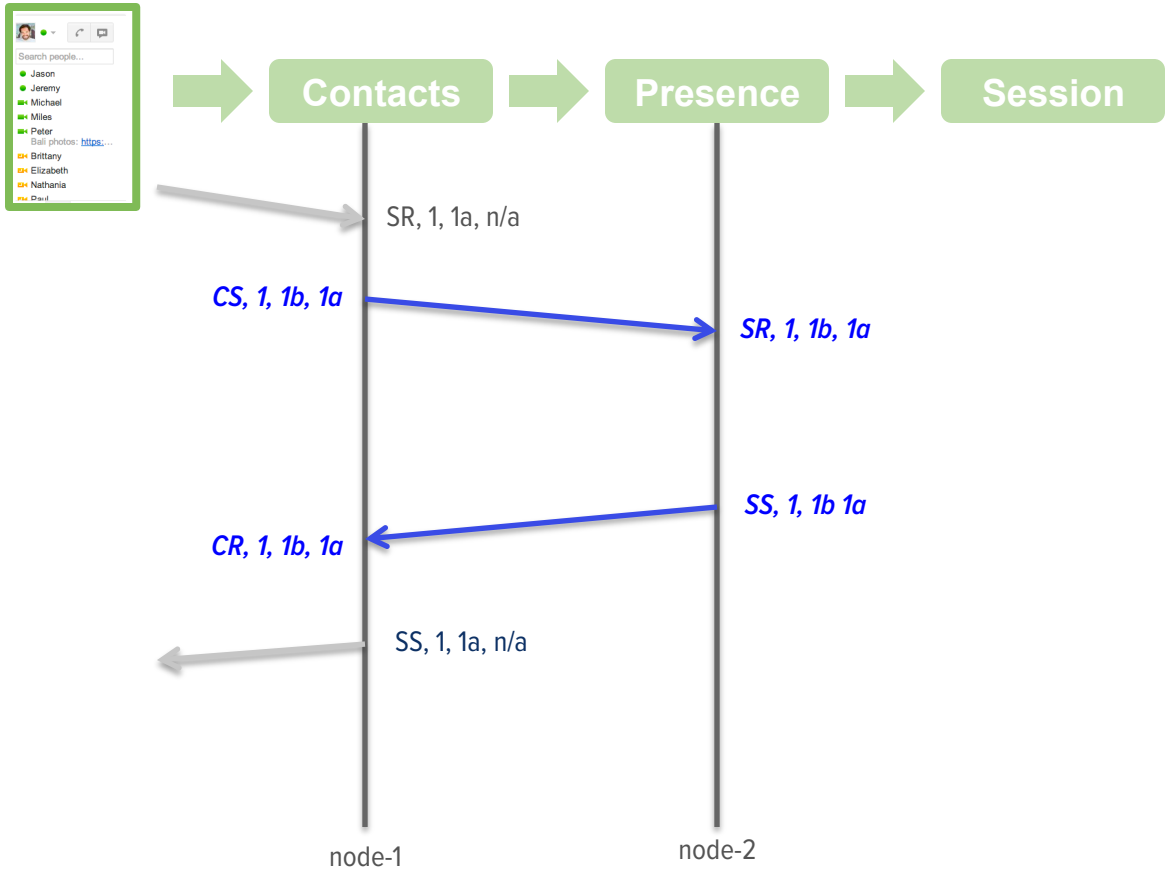
Structure

Event: Server Sent(SS)

TraceId: 1

SpanId: 1a

ParentId : N/A



Span

CS, SR, SS, CR

Traceid : 1

SpanId : 1b

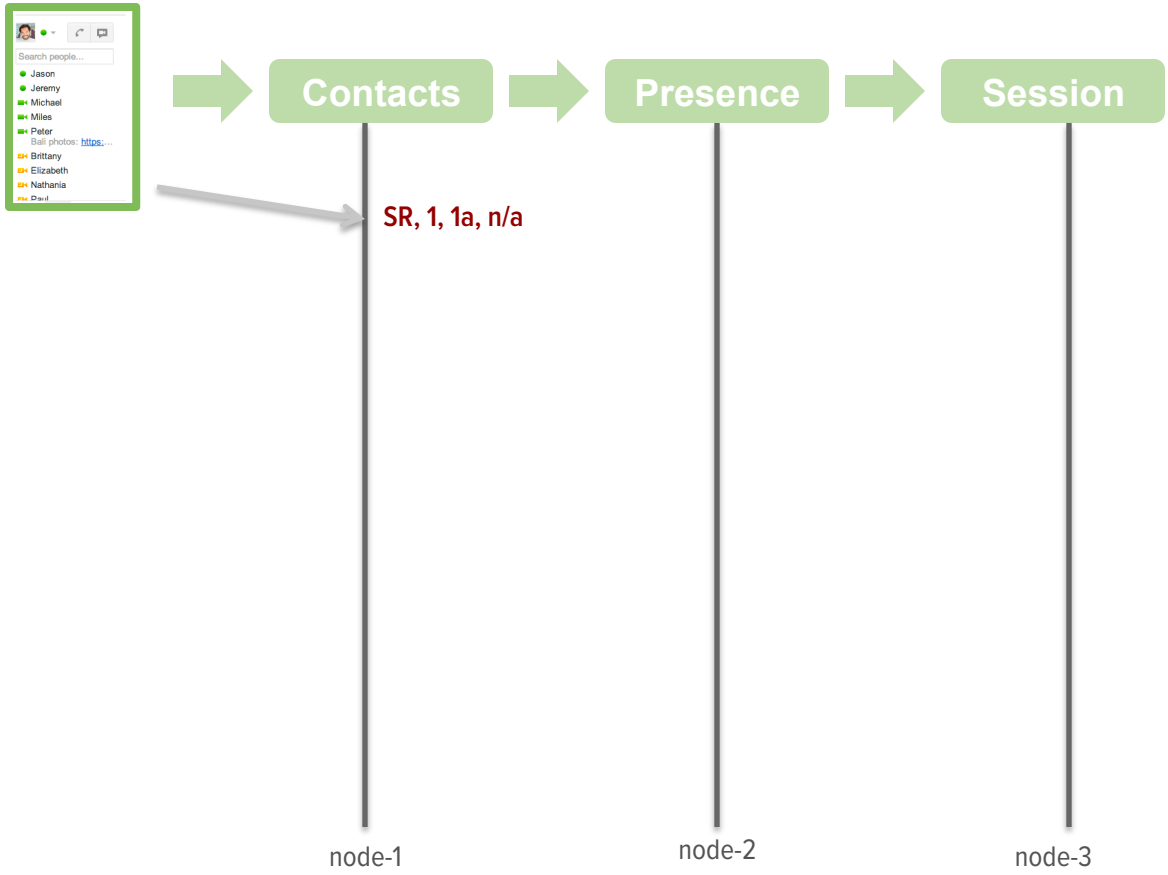
ParentId : 1a

Review

- TraceId
  - remains unique across systems
- Span
  - can identify RPC attributes
  - example: IP Address between hosts
  - Request/Response

Warning!

- We are going from 0 to 100 mph very quickly



Goal

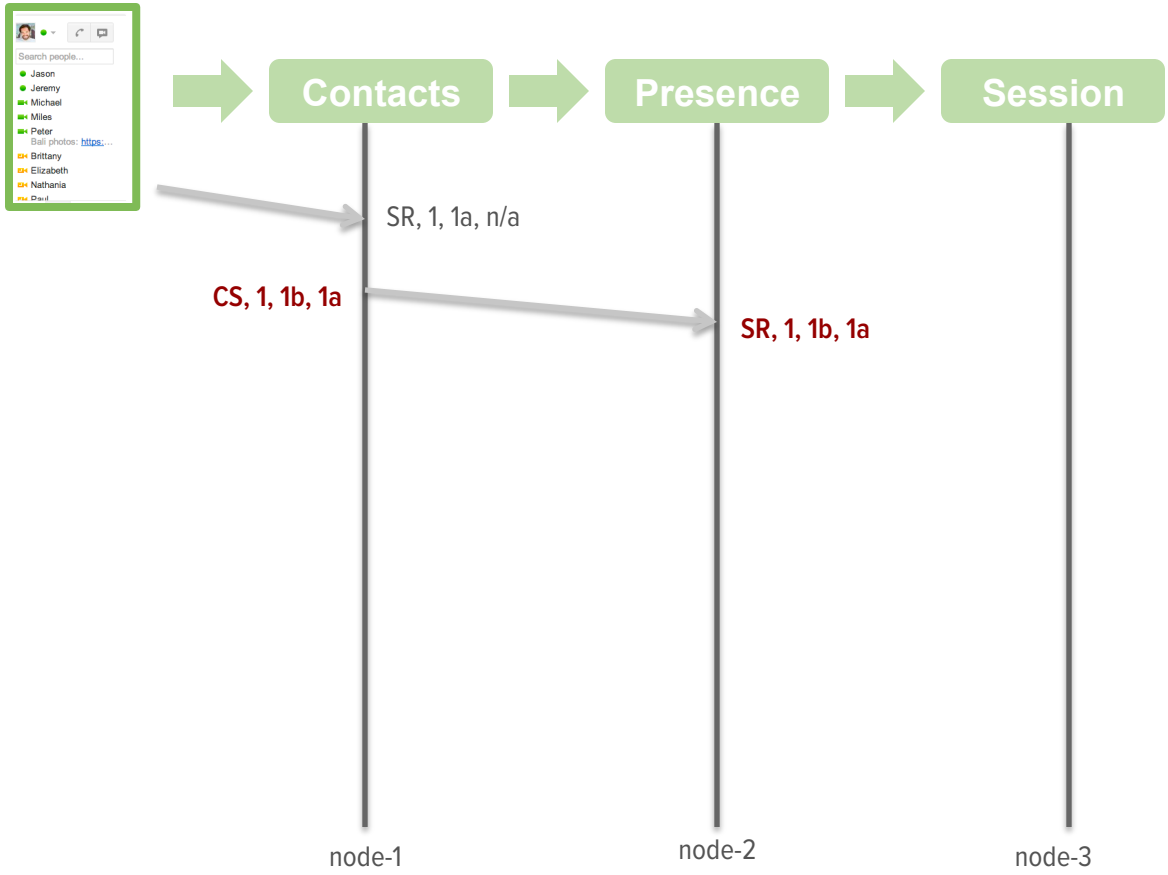
Stitch call tree from local logs

Node Log View

node-1: SR, 1, 1a, n/a

node-2:

node-3:



Goal

Stitch call tree from local logs

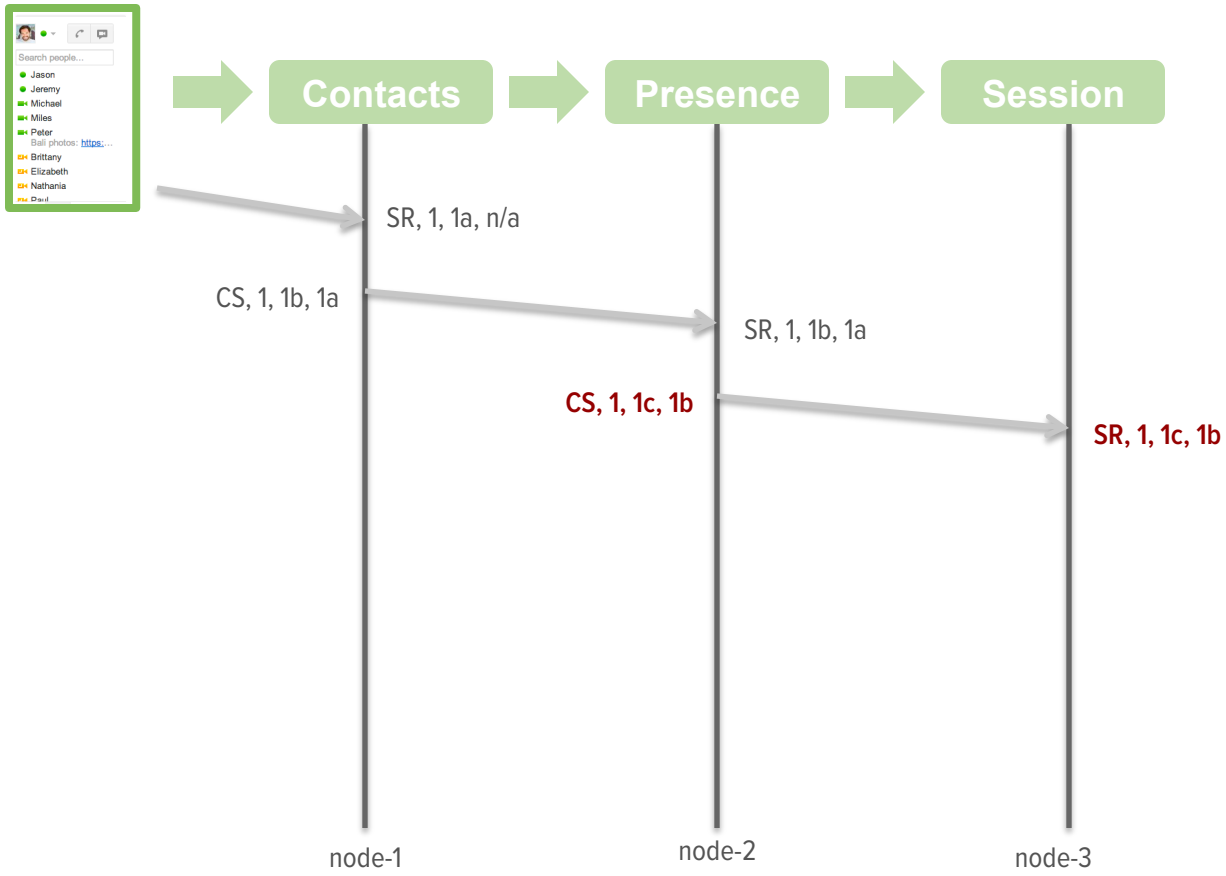
Node Log View

node-1: SR, 1, 1a, n/a

CS, 1, 1b, 1a

node-2: SR, 1, 1b, 1a

node-3:

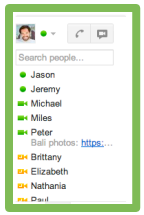


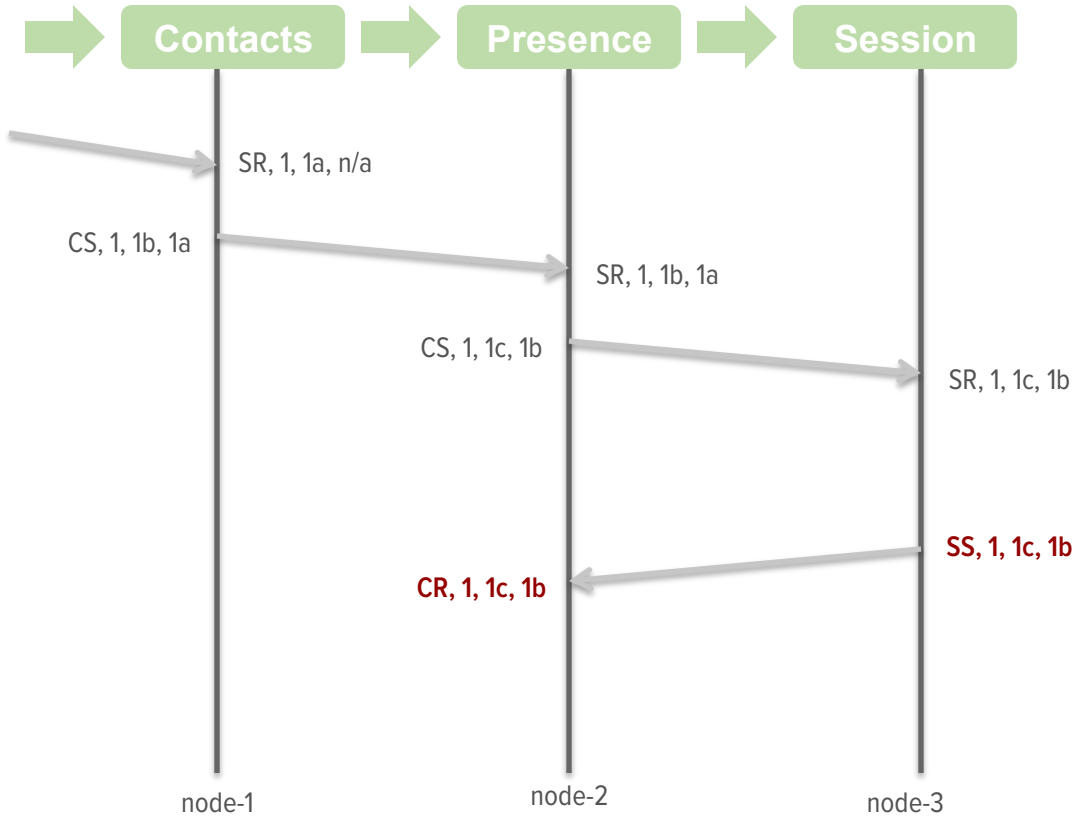
Goal

Stitch call tree from local logs

Node Log View

node-1: SR, 1, 1a, n/a
CS, 1, 1b, 1a
node-2: SR, 1, 1b, 1a
CS, 1, 1c, 1b
node-3: SR, 1, 1c, 1b





Goal

Stitch call tree from local logs

Node Log View

node-1: SR, 1, 1a, n/a

CS, 1, 1b, 1a

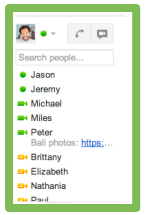
node-2: SR, 1, 1b, 1a

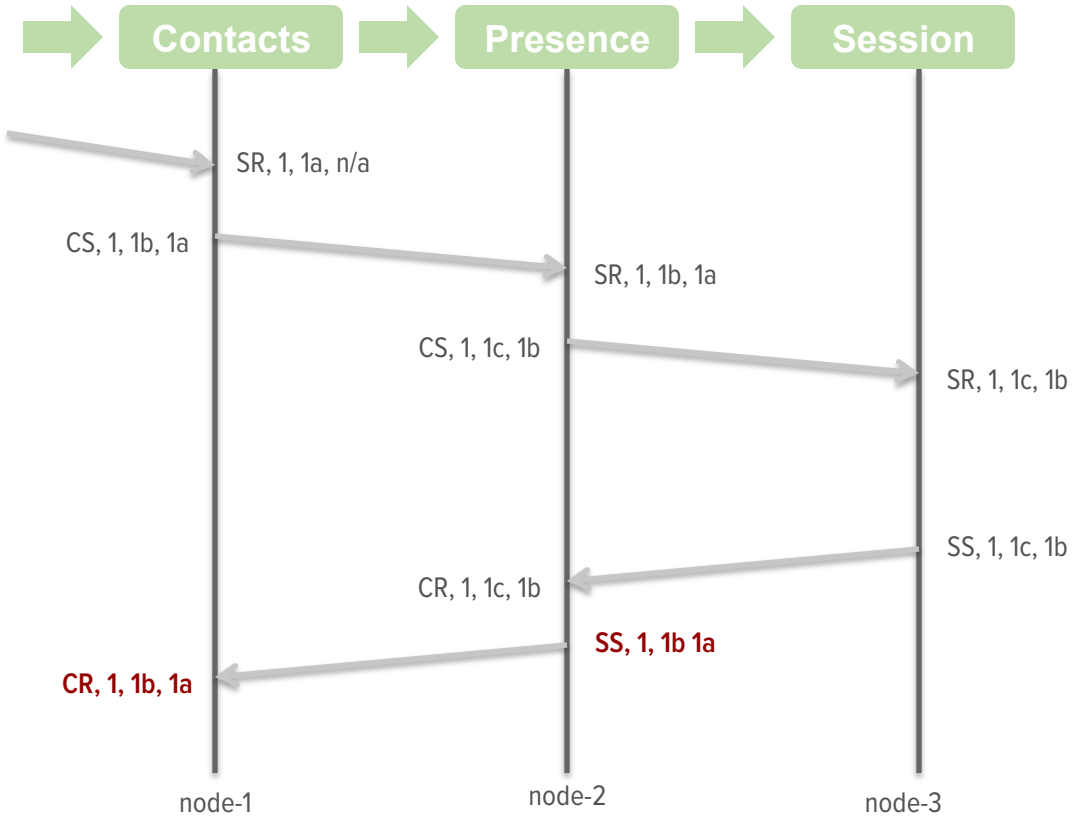
CS, 1, 1c, 1b

CR, 1, 1c, 1b

node-3: SR, 1, 1c, 1b

SS, 1, 1c, 1b



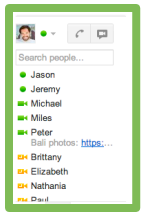


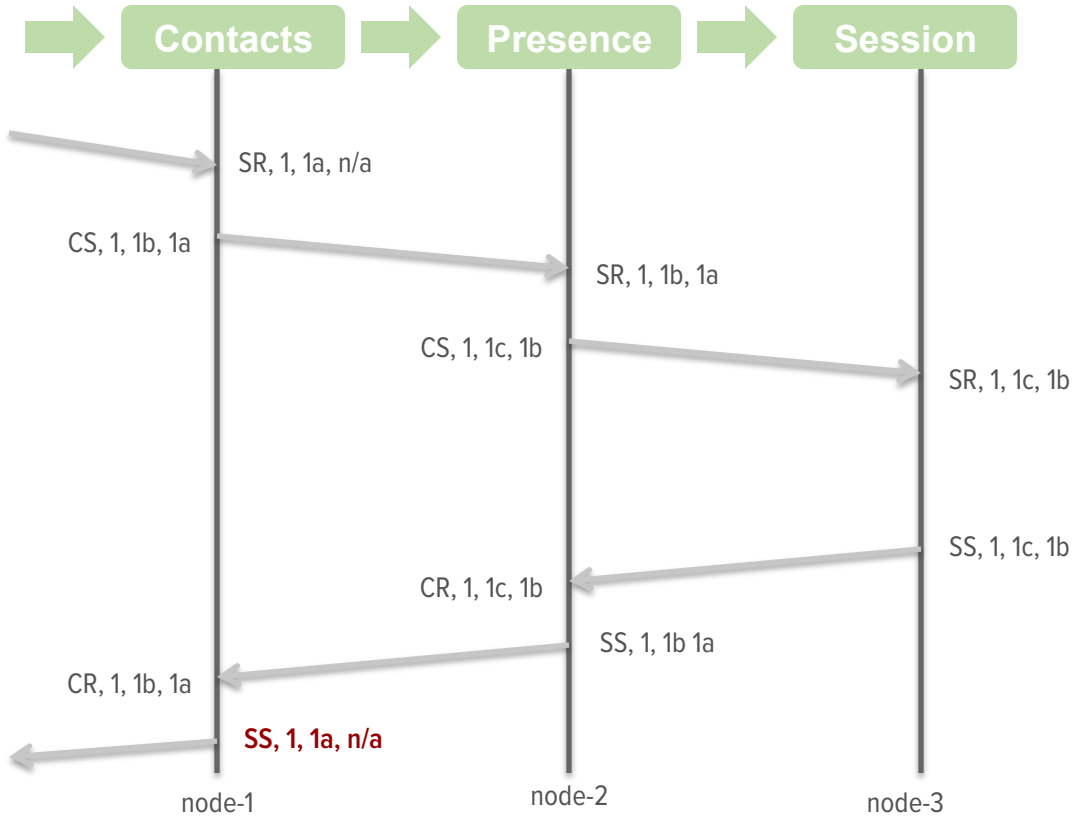
Goal

Stitch call tree from local logs

Node Log View

node-1: SR, 1, 1a, n/a
CS, 1, 1b, 1a
CR, 1, 1b, 1a
node-2: SR, 1, 1b, 1a
CS, 1, 1c, 1b
CR, 1, 1c, 1b
SS, 1, 1b, 1a
node-3: SR, 1, 1c, 1b
SS, 1, 1c, 1b





Goal

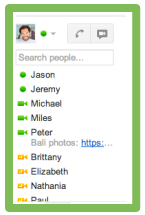
Stitch call tree from local logs

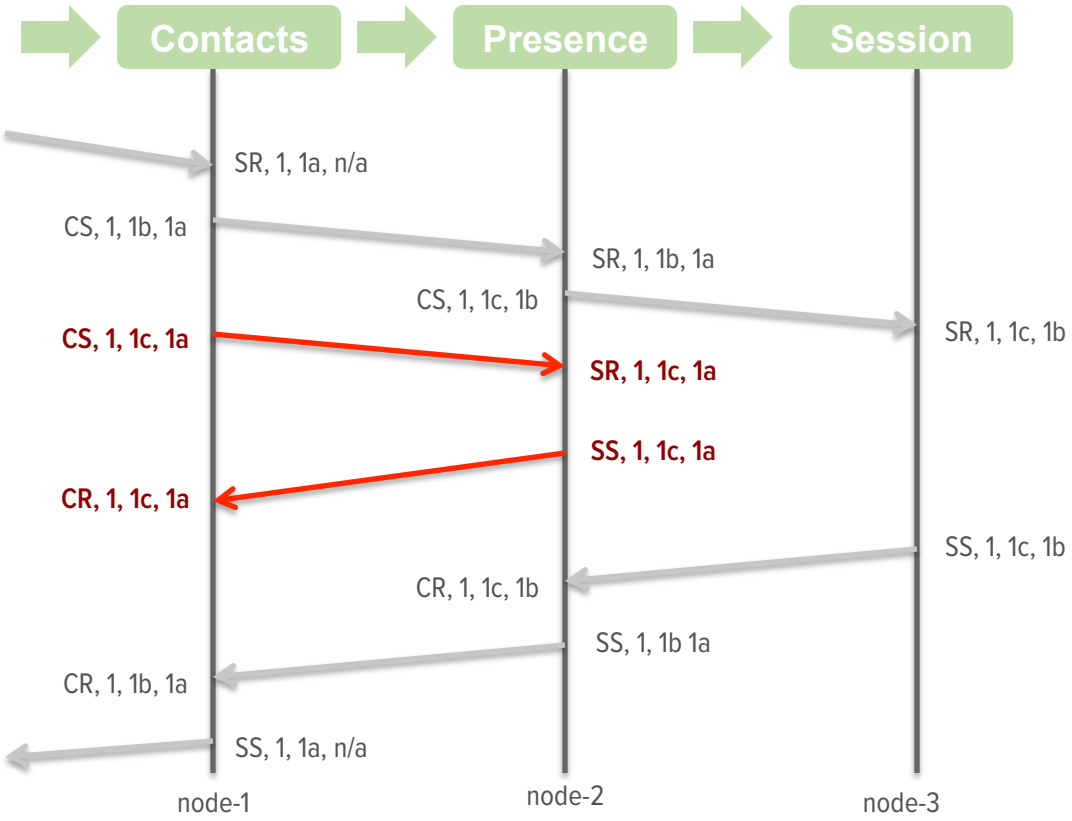
Node Log View

node-1: SR, 1, 1a, n/a
CS, 1, 1b, 1a
CR, 1, 1b, 1a
SS, 1, 1a, n/a

node-2: SR, 1, 1b, 1a
CS, 1, 1c, 1b
CR, 1, 1c, 1b
SS, 1, 1b, 1a

node-3: SR, 1, 1c, 1b
SS, 1, 1c, 1b





Goal

Stitch call tree from local logs

Node Log View

node-1: SR, 1, 1a, n/a
CS, 1, 1b, 1a
CR, 1, 1b, 1a
CS, 1, 1c, 1a
CR, 1, 1c, 1a
SS, 1, 1a, n/a

node-2: SR, 1, 1b, 1a
CS, 1, 1c, 1b
CR, 1, 1c, 1b
SR, 1, 1c, 1a
SS, 1, 1c, 1a
SS, 1, 1b, 1a

node-3: SR, 1, 1c, 1b
SS, 1, 1c, 1b

Review

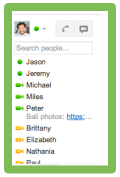
- Annotation based instrumentation
  - Pros
    - End-to-end per request visibility
    - Dependency of a single service on upstream and downstream services
    - Critical path or long tail latency can be identified
  - Cons
    - Requires instrumentation of programs

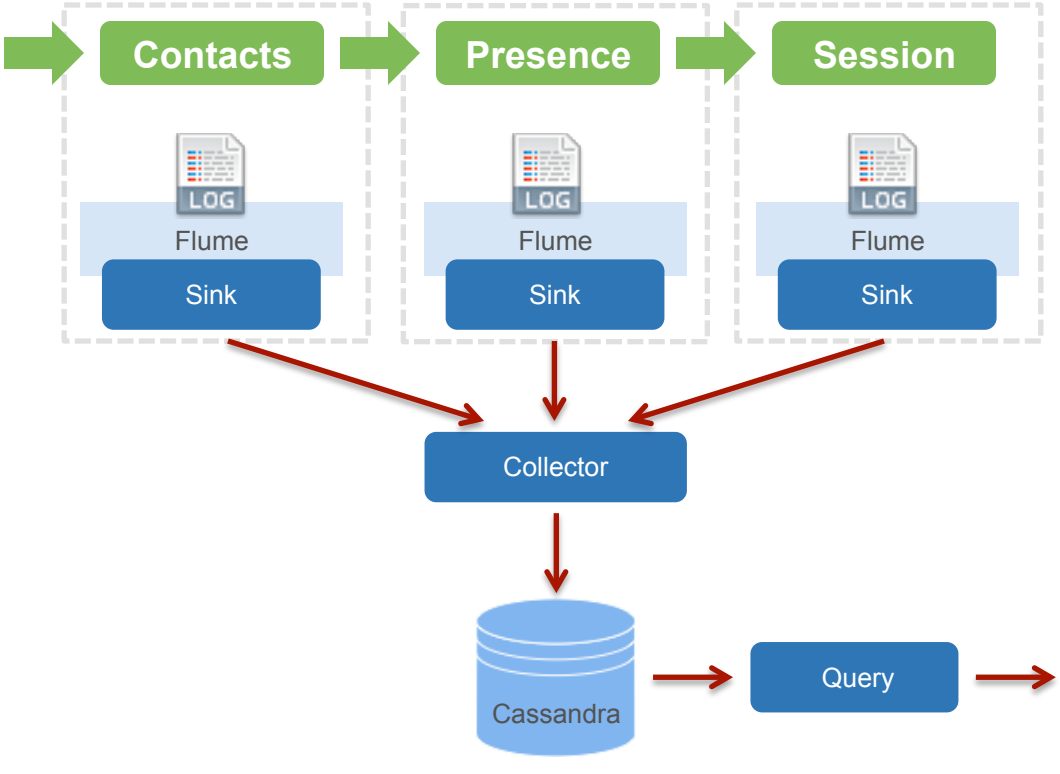
Trace collectors

- Out of band trace collection
  - In band trace data adds application overhead
  - Many systems continue munging data after results are returned
  - Kafka, Flume, Scribe collectors are most popular
  - Syslog-ng requires customization

Trace collection overhead

- PLT test
  - collect empirical data
  - tune sampling
  - second round sampling on collection
- Ability to switch off tracing
  - No impact to in band services





Demo

- Generate Traces
- Aspects of Zipkin UX
- Aggregate Services Mapped

Anatomy of a Trace

- Trace
  - root span relating to many spans
- Span
  - set of annotations
- Annotation
  - records time of event + custom
- Sampling
  - boolean

Sampling

- Used to control the overhead
- Accomplish by using Trace Filters
  - Fixed
  - Adaptive
  - Debug

Annotation

- Timestamp events
- SR – server received, SS – server sent
- CS – client sent, CR – client received

Annotation

```
# some event took place, either one by the framework or by the user
struct Annotation {
  1: i64 timestamp              # microseconds from epoch
  2: string value               # what happened at the timestamp?
  3: optional Endpoint host     # host this happened on
  4: optional i32 duration      # how long did the operation take? microseconds
}
```

Annotation.Endpoint

```
# this represents a host and port in a network
struct Endpoint {
  1: i32 ipv4,
  2: i16 port
  3: string service_name  # which service did this operation happen on?
}
```

Span

- TraceId
- name
- Id
- ParentSpanId
- Annotation List
- Binary Annotation List

Span

```
struct Span {
  1: i64 trace_id              # unique trace id, use for all spans in trace
  3: string name,              # span name, rpc method for example
  4: i64 id,                   # unique span id, only used for this span
  5: optional i64 parent_id,  # parent span id
  6: list<Annotation> annotations, # list of all annotations/events that occured
  8: list<BinaryAnnotation> binary_annotations # any binary annotations
}
```

Communicate a Trace

- Protocol headers
  - X-B3-TraceId
  - X-B3-SpanId
  - X-B3-ParentSpanId
  - Sampling
  - ~426 bytes

Instrumention Points

- Each framework maybe different
- The ones implemented
- Spring
  - Events, proxies and interceptors
- Netty
  - Message headers, EventBus
- Loggers
  - Logback, Log4j

Spring Integration

- AbstractReplyProducingMessageHandler
  - service-activator
  - handleRequestMessage
    - inInterceptor
    - handleMessage
    - outInterceptor

Implementing a Library

- To record SR/SS timestamp annotations
  - def preProcess (service: String, headers: Map, annot: Map)
  - def postProcess()
- To record CS/CR timestamp annotations
  - def preClientExec(service: String, headers: Map, annot: Map)
  - def postClientExec()
- Record traces
  - def trace(message: String)

Initialize your System

- List<SpanCollector> spanCollector
  - example: Log4j, Kafka, Scribe, Syslog
- List<TraceFilter> traceFilters
  - example: FixedRate, Adaptive, Debug
- ServerTracer, ClientTracer
  - spanCollector and traceFilter

Pre-process API

- If part of a Trace
  - set state from current trace headers
    - generate a new span
- If not part of a Trace
  - generate a unique traceId
  - may set spanId = traceId
- set serverReceived (SR)

Post-process API

- add annotations
  - host, sessionid, jvm id, threadId
- set serverSend (SS) timestamp annotation

- Operations are symmetrical
  - preClientExec
  - postClientExec

TraceId generation

```
/**
 * This salt is used to prevent traceId collisions between machines.
 * By giving each system a random salt, it is less likely that two
 * processes will sample the same subset of trace ids.
*/
private val salt = new Random().nextLong()
```

- Rejoinder
  - ..when you have no colocated services

TraceId generation

- UUID based
  - https://github.com/cogitate/twitter-zipkin-uuid
  - Util
  - Finagle
  - Ostrich
  - Server

TraceId generation

- UUID support
  - SQL Lite
  - Cassandra

- Circa April 2015

Useful Zipkin APIs

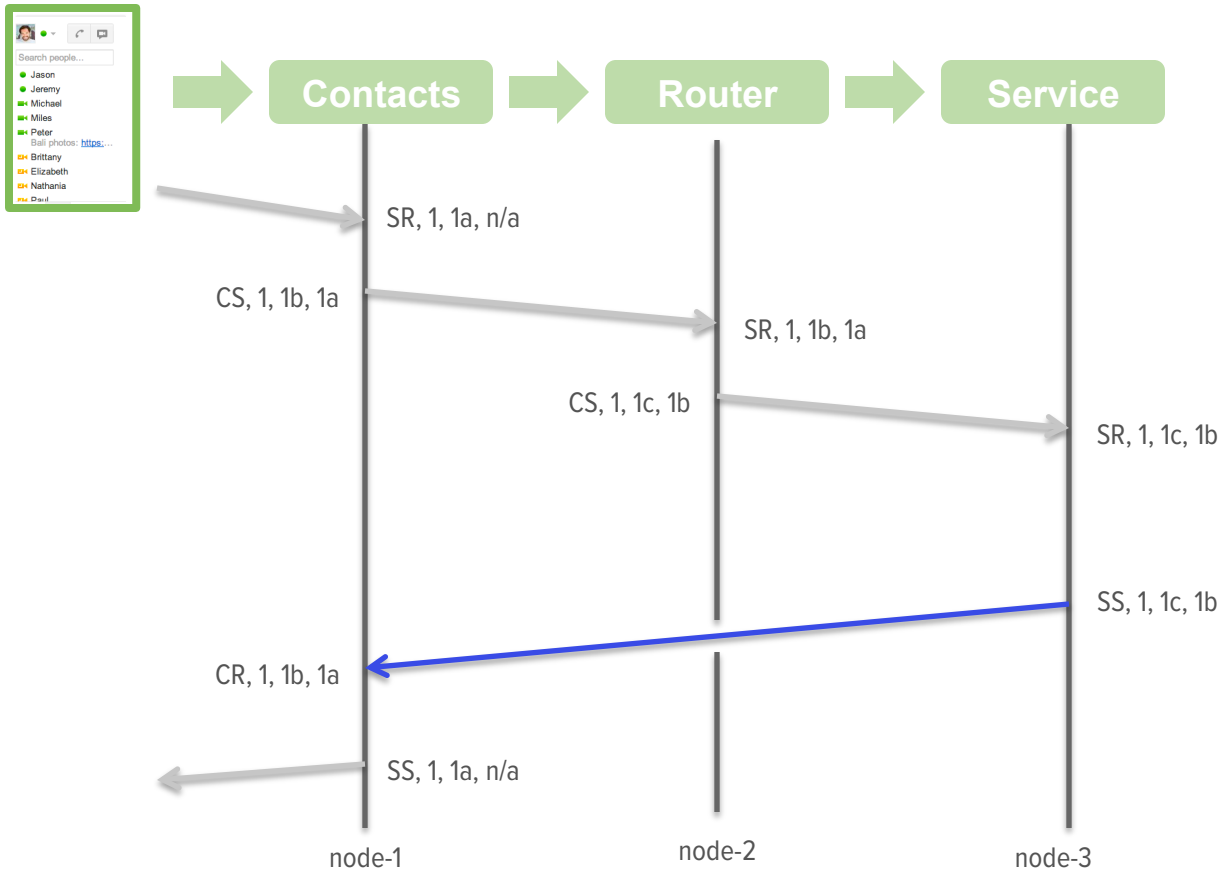
- /traces/{id}
- /api/trace/{id}
- /api/query
- /api/services
- /api/spans

Production coverage

- Common RPC libraries provide leverage
- Some coverage maybe lost due to
  - non-instrumentable legacy systems
  - non-standard control flow

Non standard use cases

- Routers and Forwarders
- Netty like dispatchers
- Non-participating trace nodes
  - To Trace or Not to Trace



Goal

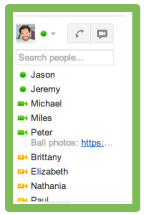
Stitch call tree from local logs

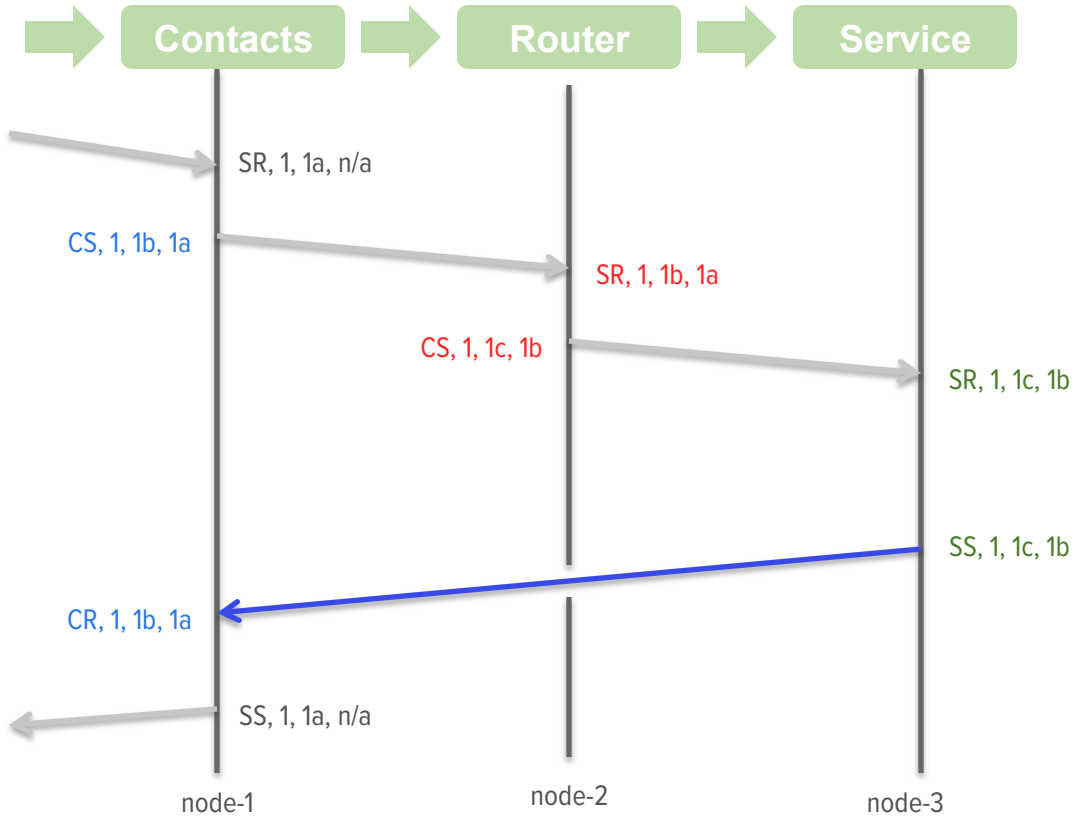
Node Log View

node-1: SR, 1, 1a, n/a
CS, 1, 1b, 1a
CR, 1, 1b, 1a
SS, 1, 1a, n/a

node-2: SR, 1, 1b, 1a
CS, 1, 1c, 1b

node-3: SR, 1, 1c, 1b
SS, 1, 1c, 1b





Goal

Stitch call tree from local logs

Node Log View

node-1: SR, 1, 1a, n/a

CS, 1, 1b, 1a

CR, 1, 1b, 1a

SS, 1, 1a, n/a

node-2: SR, 1, 1b, 1a

CS, 1, 1c, 1b

node-3: SR, 1, 1c, 1b

SS, 1, 1c, 1b

Where's my span?

Non standard use cases

- Reactive, Netty, Asynchronous callbacks
  - Request sent on a different thread
  - Response received on a different thread
- Messaging pattern
  - getSpan (get context)
  - setSpan (set context)

Lessons Learnt

- Development teams use tracing system to debug
  - Surprised with sampling
- Requirement to generate TraceId ALWAYS
- Multipage group tracing
  - UberTrace – List<TraceId>
- Use collectors that you already have
  - add converters
- Monitor
  - Disk Space
  - Collectors buffering

Lessons Learnt

- Dependency Hell
  - control libthrift version
- Mis-understand cassandra
  - CQL
  - Cassandra-CLI
- Need for different consumers
  - Elastic Search
  - Hadoop
- Aggregations
  - Scalding, Algebird, Spark
  - Steep learning curve

Lessons Learnt

- Be ready to interact with INFRA
  - Switch/Port, Dynamic Port Info
  - SSD vs Traditional Hard disk
  - Tools to probe system information
- Network failure points
- Capacity planning and sizing
- Understand Storage systems
  - constraints and limitations

Call Tracing

- Dedicated to a full view of a request's path through SOA
- Based on Google's Dapper paper
- Zipkin, Brave, Node-Tryfer
  - Scala, Java, NodeJS
- Zipkin
  - Collector, Query and UI

Acknowledgements

- SpringOne2GX Team!

- Jeff Smick – Twitter
- Kristof Adriaenssens – Brave
- Node Tryfer - Rackspace
- Eirik Sletteberg - Aggregates

Stay Connected

https://www.linkedin.com/in/cogitate

Q&A