Distributed Systems Tracing

Michiel Kalkman

Reference system

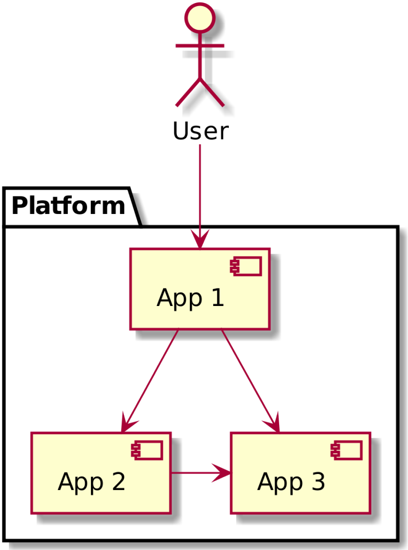

Figure 1: Reference system

Tracing

> *A trace is a representation of a series of causally related distributed events that encode the end-to-end request flow through a distributed system.*

Source

Pillars of Observability

	Logs	Metrics	Tracing
Accounting		X	X
Reporting		X	X
Alerting		X	X
Testing	X	X	X
Diagnostics	X	X	X
Verification	X		X
Auditing	X		

Observability flow

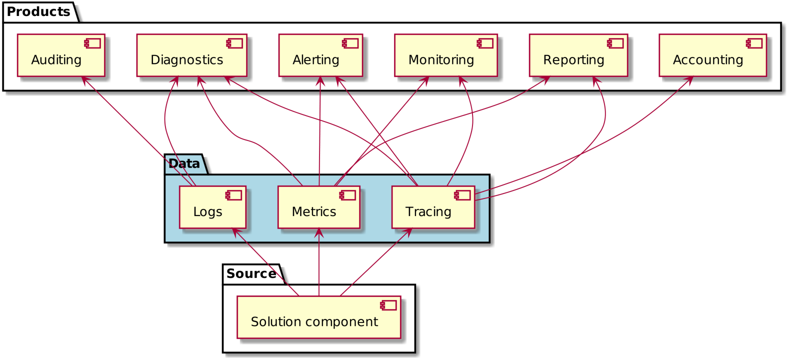

Figure 2: Each component in a solution generates visibility data

Observability flow - Tracing

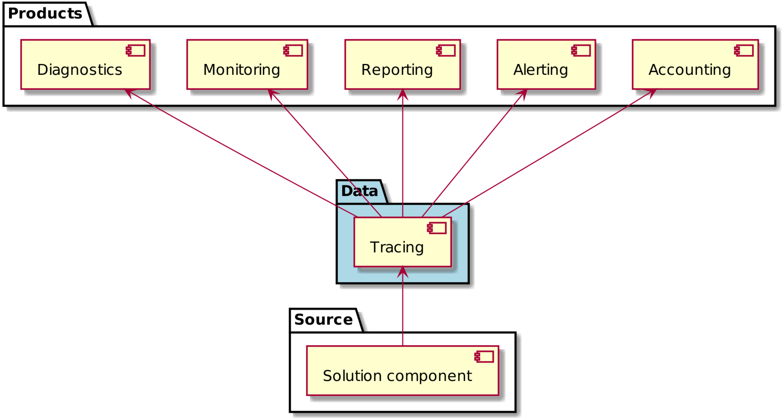

Figure 3: Position of tracing

Logging does not provide Tracing

Log stream

Time	App	Content
15:00.01	App 1	Received request
15:00.01	App 1	Call App 2
15:00.01	App 1	Call App 3
15:00.02	App 3	Received request
15:00.02	App 3	Processing request
15:00.02	App 2	Received request
15:00.03	App 3	Respond to App 1
15:00.03	App 2	Processing request
15:00.04	App 2	Respond to App 1
15:00.05	App 1	Process responses
15:00.06	App 1	Respond to caller

Log stream by application

Time	App 1	App 2	App 3
15:00.01	Received		
15:00.01	Call(App2)		
15:00.01	Call(App3)		
15:00.02		Received	Received
15:00.02			Processing
15:00.03		Processing	Response(App1)
15:00.04		Response(App1)	
15:00.05	Processing	Response(App1)	
15:00.06	Respond(Caller)		

Log issues

- Reliance on timestamps from system clocks
  - Insufficient granularity
  - Synchronization is unreliable
  - No *happens-before* semantics
- Loss of order/sequence, events can be received out-of-order
- Loss of causality, events are unrelated to each other
- Lack of consistent representation, event content is unstructured
- Lack of availability, no garuantee that logging is implemented

Tracing requirements

Uses

> *Traces are used to identify the amount of work done at each layer while preserving causality by using happens-before semantics.*

Source

Event causality

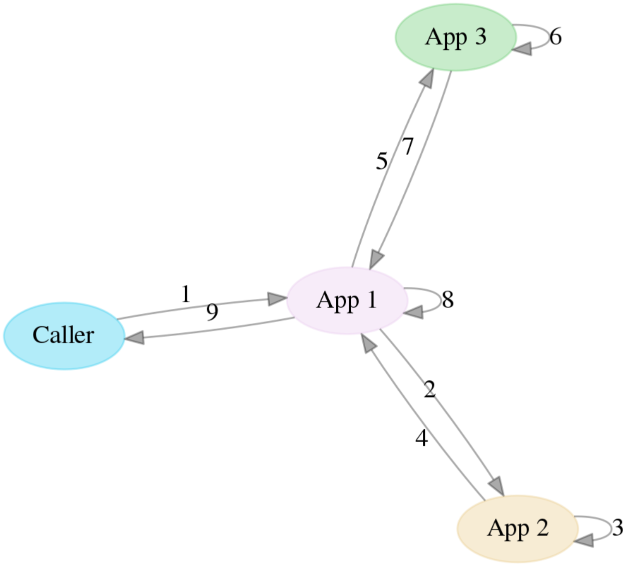

Figure 4: Events as a directed graph, showing causal relations

Event causality over time

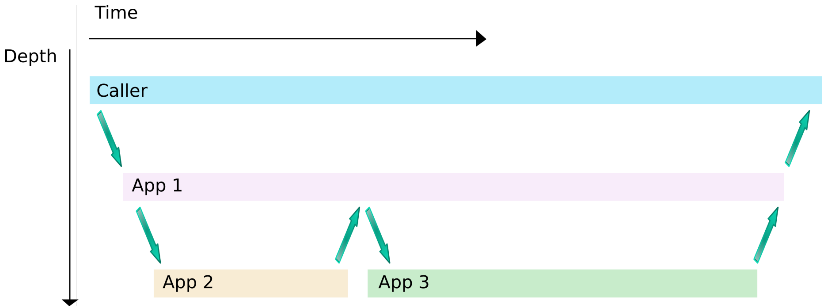

Figure 5: Events as a flame chart

Verification

- Does each microservice call the policy agent for all incoming requests?
- Does each incoming HTTP request trigger the creation of an audit log?
- Is the cache hit rate for service X within expected range?

Accounting and Reporting

Record

Example for collector storage (sqlite3) - implementations will differ

```
CREATE TABLE traces (
  unique_id   STRING,   /* id: unique */
  service_id  STRING,   /* id: service */
  function_id STRING,   /* id: service function */
  client_id   STRING,   /* id: user, system */
  starttime   INTEGER,  /* timestamp */
  endtime     INTEGER,  /* timestamp */
  duration    INTEGER,  /* ms - total time */
  cpu         INTEGER,  /* ms - processing state */
  io          INTEGER,  /* ms - processing state */
  wait        INTEGER,  /* ms - processing state */
  details     BLOB      /* trace-specific data */
);
```

Reporting- Utilization

```sql
SELECT service_id, function_id, sum(cpu), sum(io),
       sum(wait), sum(duration) as total
       FROM traces
       GROUP BY function_id
       ORDER BY total DESC;
```

Service	Function	cpu	io	wait	total
preferences	get	6084	6747	8262	21093
preferences	update	4965	4261	4841	14067
shopping_cart	add_item	3844	4523	4608	12975
user_management	get_user	4181	3820	3493	11494
user_management	list_users	3090	3290	2772	9152
user_management	update_user	2065	2893	2766	7724
shopping_cart	list_items	2538	2739	2403	7680
shopping_cart	remove_item	1948	2169	1720	5837

Reporting - SLA

```sql
SELECT service_id, function_id,
       COUNT(unique_id) as breaches
       FROM traces
       WHERE duration > 500
       GROUP BY service_id, function_id
       ORDER BY breaches;
```

Service	Function	Number of breaches
preferences	get	24
preferences	update	18
shopping_cart	add_item	17
user_management	get_user	16
user_management	update_user	10
user_management	list_users	9
shopping_cart	remove_item	8
shopping_cart	list_items	7

Reporting - Affected users

```
SELECT client_id,
       COUNT(unique_id) AS breaches
       SUM(duration) as total, AVG(duration),
       FROM traces
       WHERE duration > 500
       GROUP BY client_id
       ORDER BY breaches DESC;
```

Client	Breaches	Total time	Avg time
john	32	22887	715
jack	28	18069	645
joe	25	17175	687
jill	24	15990	666

Reporting - Advantages

- Tracing data is consistent across protocols
  - No intermediate extraction step (i.e. from log events)
  - Can be implemented for any protocol (RPC, MQ, custom, etc)
- Tracing data represents actual client experience
  - Can be extended to include the actual client in the trace
- Tracing data contains trace-specific details
  - Immediate answer to *'why is this trace slow?*

OpenTracing

Single standard + implementation

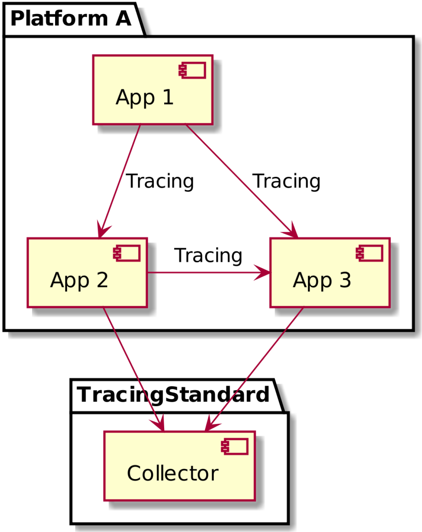

Figure 6: Visibility across **application** boundaries

Multiple standards + implementations

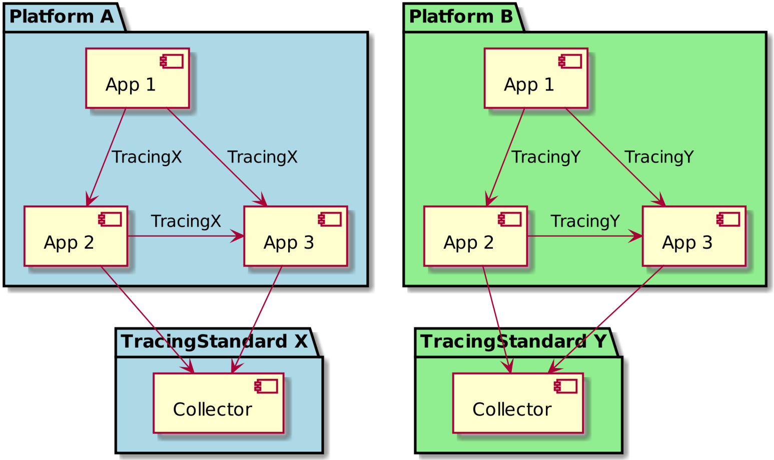

Figure 7: Incompatible tracing standards

Incompatibility

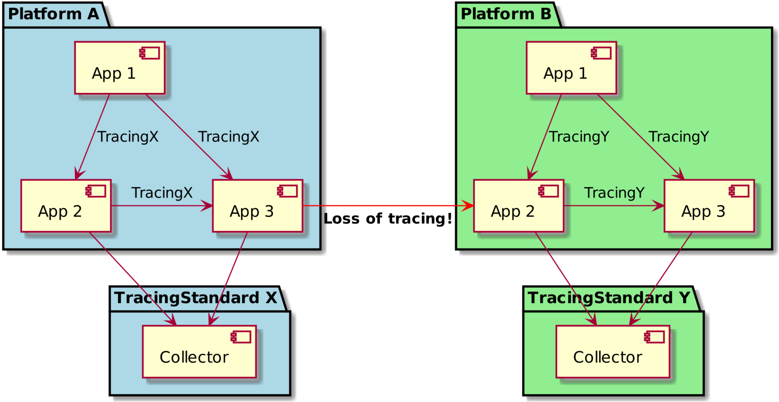

Figure 8: No visibility across **platform** boundaries

Vendor-neutral standard + multiple implementations

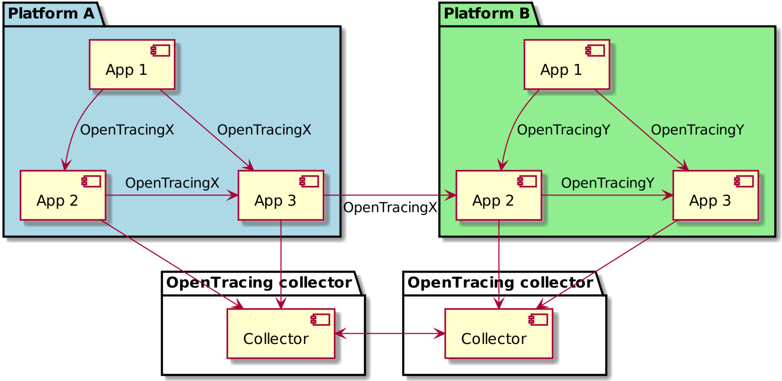

Figure 9: Visibility across **platform** boundaries

Background

OpenTracing is,

- an API specification, not a standard or an implementation
- vendor-neutral and a project under the CNCF
- inspired by Google Dapper paper

OpenTracing nouns

- **Trace** : The description of a transaction as it moves through a distributed system.
- **Span** : A named, timed operation representing a piece of the workflow. Contains key/value pairs and logs
- **Span context** : Trace information that accompanies the distributed transaction. Contains trace ID and span ID

Spans

Each Span has,

- An **operation name**
- Start and finish timestamps
- A **Span context** containing
  - **Baggage Items** : key:value pairs that cross process boundaries
  - Implementation-dependent state needed to refer to a span across a process boundary

HTTP Trace-Context headers

These fields are being standardized

field	format	description
`trace-id`	128-bit; 32HEXDIG	ID of entire trace
`span-id`	64-bit; 16HEXDIG	ID of caller span (parent)

HTTP B3 headers

These fields are used by Zipkin-derived systems

field	format	description
`X-B3-TraceId`	64, 128-bit	ID of trace, every span shares this ID
`X-B3-SpanId`	64-bit	Position of current operation in trace tree. May be derived from `TraceId`

Header propagation

Generic requirements

- Incoming request handling
  - Generating new `spanId`
- Session or context handling (storing the trace information)
- Outgoing request handling
  - Passing tracing information via metadata, headers, etc
- Incoming response handling
- Outgoing response handling