Who

I’m an assistant professor at Brown University interested in Networking, Operating Systems, Distributed Systems

X-Trace, with George Porter (UCSD)
PivotTracing, with Jonathan Mace & Ryan Roelke

www.cs.brown.edu/~rfonseca

Much of this work with George Porter, Jonathan Mace, Raja Sambasivan, Ryan Roelke, Jonathan Leavitt, Sandy Ryza, and many others.
Recap

• Last year I talked about why we were tracing like it was 1973 (when TCP/IP were born)
  – Multiple implementations of tracing doing similar but incompatible things
  – Key pattern: causal context propagation
Examples

• **Tracing:**
  – Zipkin, HTrace, TraceView, X-Trace, ...

• **Monitoring**
  – PivotTracing

• **Deadlines:**
  – Timecard (SOSP’13),...

• **Resource Accounting / Management:**
  – Retro (NSDI’15), Quanto

• **Authentication**

• **Others:**
  – Apple’s Activity Tracing, ...

• **I call these meta-applications**
Obligatory ugly hourglass picture

*Causeway (Chanda et al., Middleware 2005) used this term
Recap

• Last year I talked about why we were tracing like it was 1973 (when TCP/IP were born)
  – Multiple implementations of tracing doing similar but incompatible things
  – Key pattern: causal context propagation

• Amazing progress!
  – OpenTracing includes Baggage
  – OpenCtx tackles DCP
This Talk

I’ll describe some of our Brown Tracing Framework

http://brownsys.github.io/tracing-framework/

My real goal: understand the similarities and differences from other projects in the community.
Brown University Tracing Framework

Welcome to the Brown University Tracing Framework repository. This repository contains multiple projects from the Brown University systems research group, such as X-Trace, Retro, and Pivot Tracing.

Full documentation is available at https://brownsys.github.io/tracing-framework

Getting Started

Head over to the tutorials section to begin!

Projects Overview

The tracing framework comprises four main projects:

**Tracing Plane** is the underlying instrumentation library that the other projects are built on. The Tracing Plane provides a generic end-to-end metadata propagation primitive called Baggage that lets you dynamically propagate key-value pairs along the execution path of a request. Baggage provides a primitive similar to thread local variables, but instead at the granularity of a request, as it traverses process, machine, and application boundaries.

**Pivot Tracing** is a dynamic monitoring framework for distributed systems. Users can write high-level monitoring queries that are compiled down into instrumentation and automatically installed into live, running systems. Pivot Tracing uses the Tracing Plane’s Baggage primitive for correlating statistics from multiple places in the system.

**X-Trace** is a distributed causal logging framework. Instead of logging to per-process and per-machine files, X-Trace collects logged messages centrally. X-Trace uses the Tracing Plane’s Baggage primitive to pass identifiers through the system, enabling causally related logging statements to be tied together. X-Trace can automatically hook into existing log4j/commons loggers.

**Retro** is a fine-grained resource consumption, attribution, and actuation library. Retro instruments both system- and application-level resources, such as disk, network, cpu, locks, and queues. Retro aggregates consumption statistics, broken down according to the high-level tenant consuming the resource. Retro provides hooks for throttling the requests of individual tenants. Retro reports measurements globally, enabling centralized resource management policies that react to aggressive resource consumption by throttling the responsible tenants. Retro uses the Tracing Plane’s Baggage primitive to propagate tenant identifiers.

Instrumented Systems

These projects are all instrumentation libraries. They are intended for use with a new or existing system. We also have some pre-instrumented forks of some well known systems:

- HDFS 2.7.2
- Apache YARN 2.7.2
- Hadoop MapReduce 2.7.2
- HBase (coming soon)

[Note: we are currently porting instrumentation for these systems to the latest versions for release. Check back soon]

Note: This documentation is a work in progress. For questions, please use the pivot-tracing-users Google group.
Current Status

• **Tracing Plane**
  – Distributed Context Propagation
  – Generic Baggage Implementation
  – Instrumented HDFS, YARN, MapReduce, HBase*, Zookeeper*, Spark*

• **Tracing Applications**
  – X-Trace, Retro, Pivot Tracing

• **Working on Tracing Plane white paper / reference implementation**
  – Our current implementation still doesn't completely separate the two lower layers (baggage and DCP)
Two main problems

• Developers of target systems only want to instrument their systems once
• Many different meta-applications currently existing, and many more to appear
• **Question:** What is the minimum you have to add to your system so that it can support different meta-applications?

Spoiler: DCP!
Tracing Plane

• **Three layers:**
  – Instrumentation Layer (DCP)
  – Data Layer (Baggage)
  – (Meta-)Application Layer

• **Orthogonal:**
  – How to extract information from the instrumented systems
Instrumentation Layer

- **Main concern: distributed context propagation**
  - "Propagate opaque context until told to stop"
- **Agnostic to contents of metadata**
  - Context is opaque set of bytes, interpreted by clients
- **Implemented by: developers of instrumented systems**
  - Know the program flow, concurrency, and messaging structure, ...
- **Implementation:**
  - Implicit or explicit
  - Thread-local storage, Continuation-local storage, instrumented queues, RPC libraries, etc.
  - Upcalls to signal events, or obtain specialized behavior
- **Used by**
  - Data layer, or directly by (meta) applications
Instrumentation Layer API

↑ start(ctx)
↑ stop()
↑ fork()   (↑ are upcalls)
↑ join(incomingCtx)  //OC join
↑ serialize(ctx)  //OT inject
↑ deserialize(bytes)  //OT extract

set(ctx)
get()
Data Layer

• **Main concern: common functions on metadata**
  
  — **Baggage abstraction**: inline map of key -> value
  
  — Basic value is actually a *set* of values
    
    • Generic join behavior is UNION
    
    • Can be specialized by upcalls
  
  — Multiplexing: Namespaces
    
    • Map of (namespace, key) -> set of values

• **Implemented by**: baggage maintainers

• **Used by**: tracing applications
Data Layer API

pack(namespace, key, value) // = get
unpack(namespace, key)       // = set
repack(namespace, key, value)
remove(namespace, key)
↑fork(), ↑join(), ↑start(), ↑stop()
Join behavior

Goals:

- not require type-specific behavior everywhere
- Enable lazy evaluation

Set UNION as default, override by upcall
Correct Counting

At I(e): join(x=10, x=10), should be (x=10)
At II(e): join(x=10, x=10), should be (x=20)
Solution

• Default is UNION: only one of two equal entries
  – Case I is correct
• Case II?
  – Application adds an id, or tag, to the value:
    \[ x+=(10,"c"), \ x+= (10,"d") \]
  – Union is then \( x=(10c, 10d) \)
  – Can lazily update sum later
Application Layer Examples

Can use the data layer, or go directly to the instrumentation layer if baggage is overkill
X-Trace

• **Builds graphs of causally-related events**
• `XTrace.startTask()`
  taskID = <new random TaskID>
  Baggage.pack("X-Trace", "TaskID", taskID)
• `XTrace.logEvent(message)`
  currentEvent = <new random EventID>
  parentIDs = Baggage.unpack("X-Trace", "ParentEventID")
  X-TraceLogger.log(taskID, currentEvent, parentIDs)
  Baggage.repack("X-Trace", "ParentEventID", currentEvent)
• **On Baggage.join():**
  – assert(|TaskID| \leq 1)
  – ParentEventID is SET
• **May want to log an event on fork:**
Interval Tree Clocks

- **onStart():** create new empty clock: \((i, o)\), store in baggage as "ITC"
- Custom fork() and join() upcalls, no spans in sight
- **onFork():** fork\((i, e)\) -> split id, copy event
- **onJoin():** join\(((i_1, e_1), (i_2, e_2))\) -> merge ids, events
- **logEvent:** increase event part allowed by id
- **Caveat:** this is brittle, as it requires ALL forks and joins to be instrumented; dangling edges are bad.
Zipkin

- **Store TraceID, SpanID, ParentID, Sampled, Flags in Baggage**!
- **StartSpan**
  - Checks if Sampled
  - Copies SpanID into ParentID
- **Annotations**
  - Can be added to baggage and logged on Finish, or logged when created consulting TraceID, SpanID

Edit: this was deemed a bad idea at the workshop. This description talks about what goes into the baggage and how the API (could be) used by Zipkin. There is a whole other side of what gets logged by Zipkin, which is not part of the metadata propagation. The data layer does not dictate what gets propagated in band, and what does not.
Pivot Tracing

Baggage explicitly follows execution

Evaluated inline during a request
(no global aggregation needed)

Tuples are accumulated locally in PT Agent

Periodically reported back to user
e.g., every second

From a In A
Join b In B On a -> b
GroupBy a.procName
Select a.procName, SUM(b.delta)
PivotTracing

• Generalization of the counter example
• Each query stores its tuples under a unique key
• Each tuple has a unique id
  – to properly handle concurrency
• `onJoin()` not bound (default set union applied)
• Different aggregations happen *lazily*
  – Pack, Unpack, Emit
Retro

• Propagates TenantID across a system for real-time resource management
• Allows several policies – e.g., DRF, LatencySLO
• At start of request, get tenantID, store in baggage
• onStart(), onStop() used to measure this tenant’s running time
• When logging resource usage, or throttling, access tenantID in baggage

Jonathan Mace, Peter Bodik, Madanlal Musuvathi, and Rodrigo Fonseca. Retro: targeted resource management in multi-tenant distributed systems. In NSDI ’15
Why This Layering?

• **Separation of concerns**
• **Independent evolution**
• **Network effects**

**Main design decisions:**
- DCP required by all, Spans not
- Baggage common base data format/abstraction
- General join, enables lazy custom joins

**We view these as IP and TCP:**
- DCP is the minimum required by all
- Baggage is optional, but extremely convenient for what most people want to do
- Can build applications on top of either
Questions

Don't want callback handlers for every component of baggage, for every data type. Is this a required assumption?

Do other applications that don't require Spans justify our design?

Is the propagation the same for all applications?

What do people think?
Thank you!
“10th Rule of Distributed System Monitoring*”

“Any sufficiently complicated distributed system contains an ad-hoc, informally-specified, siloed implementation of causal tracing.”

*This is, of course, inspired by Greenspun’s 10th Rule of Programming