Dapper, a Large-Scale Distributed Systems Tracing Infrastructure.

Sigelman et al.

Google TR 2010

Mid-term survey

- Posted link on Piazza
- 8 responses so far
- Please fill it out to let me know how things are going

Dapper meta-level

- *Google "experience paper": what did they learn from the Dapper experiment? (Also, not peer-reviewed)*
  - Relevant outside of Google? Specific to infrastructure and experiences
  - Even LOC counts are unclear in translation due to supporting infrastructure
- *Is every large-scale paper from google necessarily an interesting paper for the research community? (No..)*
- What **is** a good experiences paper (from a research pov)?
- An experience is unique, in this case to Google! Without this, it wouldn't be an interesting experience paper
- Has influenced many later projects (*span* notion is now prevalent)

Distributed tracing

- *What's the motivation for distributed tracing? Why do we need it? Why are the previous tools we've read about insufficient?*
    - Model checking: *what could have been*
    - Tracing: *what actually happened*
- Isn't MC the superset of traces that you'll observe?
- **M**C: Can't save the traces, requires a property, *huge* state space, … really intended to find bugs. Requires an *accurate* model!
- If tracing is not for bug finding… then what is it for?
    - Measuring *performance (e.g., high latency events; may not be bugs)*
    - Capture information *in production systems (real code: C++, Java)*
- What are the requirements for such a system?
    - Goal: Continuous monitoring at runtime *across as many services as possible* ***(observability)***
    - Low overhead, *transparency (automatic instrumentation, min impact on the SUT), scalability (many services), ubiquity*
    - *Ubiquity: I don't know what I'm going to need/what's important:* ***Capture all, decide what's relevant later***
- Is Dapper a monitoring system? If you think of monitoring as runtime verification, then Dapper is not a monitoring system
- If monitoring is passive observation, then Dapper is a monitoring system — it's observing from the sidelines

Distributed tracing

- How realtime does a tracing system need to be?
    - Dapper: mostly under 15s (median)
    - Your utility determined by how realtime you are
- Enables comprehension of the system: draw a picture of your system => You need tools (on top of data)
    - Well-defined abstraction for the data
    - An API to access/query the data (DAPI)
- Dapper - big success… so, what can we learn to imitate dapper?
    - Adoption: transparency for existing apps (automate the hard parts for whoever the user is)
    - Adoption: a well-defined API to the data that is being collected (they can solve their own problem with the data)
    - General-purpose utility: solve a problem that someone has.. but don't be overly specific (for systems infrastructure). There's a tension between specificity and generality.
    - Adoption: broadly familiar sequence diagram model of execution (extensible for the power users)

Dapper abstractions

- Trees/spans: what does dapper aim/need to represent?
  - Distributed control flow (causal relations)
  - Sequence execution diagram (blocks of exec)
    - Time: “span” of time.
    - *Inter-connected intervals where things happen (tree)*
    - Unique trace identity
    - Optional: user-annotations (key-value map) —- but very widely used, not optional?
    - (Trace ~ Google servicing a single client request)
- Performance focused abstraction => needs realtime (not virtual/logical)

Dapper abstractions

- It lacks many distributed abstractions because it extends the notion of a single machine trace
- ***Maybe this is what makes this abstraction more intuitive (and useful) to developers***
- Finn: A complex tool doesn't mean it has to be complicated to use. An overly intelligent tool could be a liability — a suboptimal, *but easy to understand*, tool could be more useful!
- *Dapper: less intelligence in the tracing system, move the smarts to the end-user tools*
    - Leads to low overhead
- Sub-optimal, but easy to understand — low "emotional overhead". Forces the user to do work that they are comfortable with doing (writing MapReduce queries :-)
    - Think about and *empathize* with your users — that's the path to success!
- Breed *familiarity* with a distributed execution for someone who doesn't know distributed systems that well?
    - Totally ordered paths to represent real call stack and latency
    - Distributed tracing: take a local trace, and extend it to remote machines
    - And can customize with annotation

Dapper design

- *Mostly driven by scalability concerns (terabyte traces / day)*
- Instrument core libraries (e.g., RPC lib)
- Store traces locally on disk, pull when necessary
    - Garbage collect locally. If not used, it never moves over the network.
- Sampling by only tracing 1/n requests
- Additional sampling (fraction of traces included: during collection: hash(tradeID)->[0,1], and give the user a knob in [0,1].
- Adaptive sampling (work in progress): trace frequency based on load; latency is related to number of people who experience it (user-focused metric)
- Tree is not always a tree: these are corner cases and can be dealt with manually
    - RPC-style construction of distributed systems is the norm, so if you support it, you capture most things

Dapper experiences

- User annotations (90% of traces had at least one)
  - Indicates the dumb tracing is.. too dumb?
    - Pull in developers, and let them extend later
- Using traces for policy enforcement/checking (privacy/security: service A should not talk to service B)
- Hunted down developers who disabled Dapper and convinced them to re-enable it (another reason they are successful)
  - Network effect of tool adoption

Next: BigData Compute

- We've covered some basic theory/abstractions, and tools for constructing dist. systems. Now let's look at some complex systems!

- **Big**Data systems ~ cloud-based systems
  - First: **Spark** (analytics)
  - Then, **TensorFlow** (machine learning)
- Both use an important abstraction, **data-flow**