Lessons from Writing HTraced: A High-Performance Network Daemon in Go

by Colin P. McCabe
About Me

- I work on the Hadoop Distributed Filesystem and related big data technologies at Cloudera.
- Previously, I worked on the Ceph distributed filesystem.
Overview

- Introduction to Apache HTrace
- HTraced
- Experiences with Golang
- Optimizing Performance
- Conclusion and future plans
- Q&A
Motivations for Apache HTrace

- Diagnosing distributed systems is hard
- Many timeouts and fallbacks
- Performance problems often not repeatable
- Difficult to follow requests across project boundaries and network boundaries
Diagnosing Distributed Systems is Hard
Diagnosing Distributed Systems is Hard
Metrics

- Many different metrics available
  - JMX
  - top
  - vmstat
  - iostat
- Aggregated
- Downsampling over time
Metrics

- Good for getting an overall view of throughput
- Bad for identifying latency problems.
  - Average bandwidth, CPU, disk I/O, etc. numbers often hide significant outliers
- Hard to figure out why
  - Disk I/O stats are low… because of I/O errors? Bottlenecks elsewhere? Low load?
Log Files

- Daemons all generate log files
  - HDFS audit log
  - log4j files
  - Client log files
- Usually stored on the nodes that generated them
- Kept for some length of time, then deleted
Log Files

- Good for getting detailed information about a particular operation or point in time
- Bad for getting a holistic view of a single request. Difficult to correlate what is going on on different systems via logs
- Tradeoff between performance and logging
- Split into many different files
  - Per-host, per-project, per-faculty
HTrace’s Approach

- Distributed Tracing
  - Follow specific requests across the entire cluster
  - Follow requests across network and project boundaries
  - End-to-end tracing on a sampled subset of requests
End-to-End Tracing

- Multiple cluster nodes
- Multiple projects
  - Follow a request from HBase to HDFS
- Multiple languages (app vs. lib)
  - Java, C, C++ language bindings
- Use available storage and compute stack
HTrace Goals

- Support multiple storage and compute backends
  - Not tied to any one RPC, language, framework
- Stable, well-supported client API
- Approximately zero impact when not in use
- Can be used on production clusters
- Integration with upstream big data and Hadoop projects, to allow end-users to enable tracing without writing code.
Trace Spans

- Annotations decompose requests into **trace spans**
- Trace spans can be nested (parent/child relationship)
- Parent can have one different child per system
Trace Spans

- A trace span represents a length of time
  - Description
  - Start time
  - End time
  - Parents
  - Unique Identifier
  - Process ID and IP address
  - Time Annotations
  - Key/Value Annotations
{  
  "a": "f8e9e09c72e388f3fef51b32115beba5",  
  "b": 1448220893721,  
  "e": 1448220893788,  
  "d": "ClientNamenodeProtocol#create",  
  "p": ["f8e9e09c72e388f3dc6778916cf3a5ac"],  
  "r": "FSClient/10.20.190.31" 
}
Sampling

• Tracing all requests generates an enormous amount of data
• It’s usually more useful to do sampling-- to trace only < 1% of requests
• Sampling rate and sampler is configurable
• Sampling is currently done at the level of the entire request
Pluggable Architecture

- htrace4-core is the library for creating spans
- SpanReceivers process spans created by htrace4-core
- htrace-web queries SpanReceiver data
HTrace Graphical Interface
HTracedSpanReceiver

- Easy-to-use SpanReceiver that stores spans in a central daemon
- Indexing, web ui, aggregation in one place
htraced

- Written in Go
- HTTP
  - JSON
- HRPC
  - msgpack
- leveldb
htraced

- storage
  - Optimized for high write throughput
  - Uses multiple leveldb instances to store span data
  - begin time, end time, duration, and span ID are indexed so that range queries are fast
  - leveldb persists data to disk
Why Golang?

- Easy for new open source contributors
  - No wildly divergent C++ coding styles to deal with
  - Memory-safe
  - Garbage collected
- Easy to deploy
  - Generates binaries; can run with no start scripts
  - No problems with CLASSPATH
  - No JVM knobs to turn or max memory size
Why Golang?

- Composition instead of inheritance
  - Elegant standard library
    - net/http
    - io
  - Easy to extend standard library rather than replacing it
  - Tools and libraries, not frameworks
Why Golang?

- Concurrency primitives
  - Goroutines
  - Channels
  - Avoidmutexes or atomics when possible
- Operating system feature aren’t “hidden”
  - Can make hardlinks and symlinks, call getpid
Useful Go Libraries

- Kingpin - a very nice command-line parsing library
  - github.com/alecthomas/kingpin
- Levigo - wrappers around LevelDB
  - github.com/jmhodges/levigo
- Go-Codec - fast serialization / deserialization libraries
  - github.com/ugorji/go/codec
Challenges with Golang

- Repeatable build
  - By default, Go imports the latest versions of dependencies
  - Apache releases must generate the same artifacts each time
  - Need a way of locking the versions of dependencies
{  
  "ImportPath": "git-wip-us.apache.org/repos/asf/incubator-htrace.git",
  "Deps": [
    {
      "ImportPath": "github.com/alecthomas/kingpin",
      "Rev": "afafa8aab106d31c9dc8f5e562b3f30f6246c3d4"
    },
    {
      "ImportPath": "github.com/alecthomas/units",
      "Rev": "6b4e7dc5e3143b85ea77909c72caf89416fc2915"
    }
  ]
}
Challenges with Golang

- Logging
  - No single logging library is as dominant as log4j
  - No standard way of configuring log locations, etc.
- Created our own logging framework
  - $lg$.Debugf("Hello, world! n = %d\n", val);
Optimizing Performance

- Measuring performance
  - Built-in benchmarking tools for unit tests
  - Daemon metrics
Optimizing Performance

- Reducing Serialization Overhead
  - Reduce CPU overhead by using msgpack instead of JSON for writing spans
  - Use Go-Codec for faster implementations of serialization/deserialization
  - Enable unsafe optimizations (i.e. direct memory access)
  - TODO: generated code for serde
Optimizing Performance

- Reduce Garbage Collection Overhead
  - Deserialize lists of spans incrementally rather than all at once
  - Keep most data off-heap inside levelDB
  - Reuse Go objects where appropriate
    - Value types are helpful
  - Compile with a recent version of Go
    - Many, many recent GC improvements
Optimizing Performance

- Reducing write path overhead
  - Use multiple levelDB instances to gain I/O parallelism, disk-level parallelism
  - Pass groups of objects over channels rather than single objects to amortize channel-passing overhead
Optimizing Performance

Nanoseconds per Span Written

Horizontal axis title

Left vertical axis title

Before

After
HTrace Community

- Vibrant upstream community
  - HTrace is an Apache open source Project
  - Contributors from NTT Data, Cloudera, Hortonworks, Facebook, and others
  - The most recent release was 4.1
HTrace Community

- Sharing ideas with other big data projects
  - Hadoop
  - HBase
  - OpenTracing
  - XTrace
  - Twitter Zipkin
Recent Work in HTrace

- More effective error checking in the htrace client
- Optimized RPC format for sending spans to htraced
- Better integration with HDFS
- New web GUI for visualizing spans
- Trace spans are now tagged with IP address or hostname
- Span IDs extended to 128 bits to avoid collisions
HTrace in Cloudera’s Distribution of Hadoop

- Available as a Cloudera Labs “beta” for CDH5.5 and later
- HDFS tracing is supported
- RPMs and debs are available for htrace
Planned

- Improve the HTrace integration in HBase
- Add more annotations to Hadoop span data to get more insight
- Apache Kudu SpanReceiver
- Better integration with cluster management systems
- Improve and test C and C++ support
- Create an aggregate view for spans
HTrace Q & A