The OpenTracing Semantic Specification

Version: 1.1

Document Overview

This is the "formal" OpenTracing semantic specification. Since OpenTracing must work across many languages, this document takes care to avoid language-specific concepts. That said, there is an understanding throughout that all languages have some concept of an "interface" which encapsulates a set of related capabilities.

Versioning policy

The OpenTracing specification uses a `Major.Minor` version number but has no `.Patch` component. The major version increments when backwards-incompatible changes are made to the specification. The minor version increments for non-breaking changes like the introduction of new standard tags, log fields, or SpanContext reference types. (You can read more about the motivation for this versioning scheme at Issue specification#2)

The Big Picture: OpenTracing's Scope

OpenTracing's core specification (i.e., this document) is intentionally agnostic about the specifics of particular downstream tracing or monitoring systems. This is because **OpenTracing exists to describe the semantics of transactions in distributed systems.** Describing those transactions should not be influenced by how — or how not — any particular backend likes to process or represent data. For instance, detailed OpenTracing instrumentation can be used to simply measure latencies and apply tags in a timeseries monitoring system (e.g., Prometheus); or Span start+finish times along with Span logs may be redirected to a central logging service (e.g., Kibana).

As such, the OpenTracing specification and data modelling conventions have a wider scope than some tracing systems, "and that's okay." If certain semantic behavior is out-of-scope for a particular tracing or monitoring system, said system can summarize or simply ignore the respective data flowing from OpenTracing instrumentation.

The OpenTracing Data Model

Traces in OpenTracing are defined implicitly by their **Spans**. In particular, a **Trace** can be thought of as a directed acyclic graph (DAG) of **Spans**, where the edges between **Spans** are called **References**.

For example, the following is an example **Trace** made up of 8 **Spans**:

```
Causal relationships between Spans in a single Trace


       [Span A]  ←←←(the root span)
           |
    +------+------+
    |             |
[Span B]      [Span C] ←←←(Span C is a `ChildOf` Span A)
    |             |
[Span D]      +---+-------+
              |           |
          [Span E]    [Span F] >>> [Span G] >>> [Span H]
                                         ↑
                                         ↑
                                         ↑
                        (Span G `FollowsFrom` Span F)
```

Sometimes it's easier to visualize **Traces** with a time axis as in the diagram below:

```
Temporal relationships between Spans in a single Trace


--|-------|-------|-------|-------|-------|-------|-------|-> time

 [Span A···················································]
   [Span B·····························································]
      [Span D······························································]
    [Span C·····························································]
         [Span E·····]        [Span F··] [Span G··] [Span H··]
```

Each **Span** encapsulates the following state:

- An operation name

- A start timestamp
- A finish timestamp
- A set of zero or more key:value **Span Tags**. The keys must be strings. The values may be strings, bools, or numeric types.
- A set of zero or more **Span Logs**, each of which is itself a key:value map paired with a timestamp. The keys must be strings, though the values may be of any type. Not all OpenTracing implementations must support every value type.
- A **SpanContext** (see below)
- References to zero or more causally-related **Spans** (via the **SpanContext** of those related **Spans**)

Each **SpanContext** encapsulates the following state:

- Any OpenTracing-implementation-dependent state (for example, trace and span ids) needed to refer to a distinct **Span** across a process boundary
- **Baggage Items**, which are just key:value pairs that cross process boundaries

References between Spans

A Span may reference zero or more other **SpanContexts** that are causally related. OpenTracing presently defines two types of references: `ChildOf` and `FollowsFrom`. **Both reference types specifically model direct causal relationships between a child Span and a parent Span.** In the future, OpenTracing may also support reference types for Spans with non-causal relationships (e.g., Spans that are batched together, Spans that are stuck in the same queue, etc).

`ChildOf` references: A Span may be the `ChildOf` a parent Span. In a `ChildOf` reference, the parent Span depends on the child Span in some capacity. All of the following would constitute `ChildOf` relationships:

- A Span representing the server side of an RPC may be the `ChildOf` a Span representing the client side of that RPC
- A Span representing a SQL insert may be the `ChildOf` a Span representing an ORM save method
- Many Spans doing concurrent (perhaps distributed) work may all individually be the `ChildOf` a single parent Span that merges the results for all children that return within a deadline

These could all be valid timing diagrams for children that are the `ChildOf` a parent.

```
    [-Parent Span---------]
        [-Child Span----]

    [-Parent Span--------------]
        [-Child Span A----]
         [-Child Span B----]
       [-Child Span C----]
        [-Child Span D---------------]
        [-Child Span E----]
```

`FollowsFrom` references: Some parent Spans do not depend in any way on the result of their child Spans. In these cases, we say merely that the child Span `FollowsFrom` the parent Span in a causal sense. There are many distinct `FollowsFrom` reference sub-categories, and in future versions of OpenTracing they may be distinguished more formally.

These can all be valid timing diagrams for children that "FollowsFrom" a parent.

```
    [-Parent Span-]  [-Child Span-]


    [-Parent Span--]
     [-Child Span-]


    [-Parent Span-]
                [-Child Span-]
```

The OpenTracing API

There are three critical and inter-related types in the OpenTracing specification: `Tracer`, `Span`, and `SpanContext`. Below, we go through the behaviors of each type; roughly speaking, each behavior becomes a "method" in a typical programming language, though it may actually be a set of related sibling methods due to type overloading and so on.

When we discuss "optional" parameters, it is understood that different languages have different ways to construe such concepts. For example, in Go we might use the "functional Options" idiom, whereas in Java we might use a builder pattern.

`Tracer`

The `Tracer` interface creates `Span`s and understands how to `Inject` (serialize) and `Extract` (deserialize) them across process boundaries. Formally, it has the following capabilities:

Start a new `Span`

Required parameters

- An **operation name**, a human-readable string which concisely represents the work done by the Span (for example, an RPC method name, a function name, or

the name of a subtask or stage within a larger computation). The operation name should be **the most general string that identifies a (statistically) interesting class of `Span` instances**. That is, `"get_user"` is better than `"get_user/314159"`.

For example, here are potential **operation names** for a `Span` that gets hypothetical account information:

Operation Name	Guidance
`get`	Too general
`get_account/792`	Too specific
`get_account`	Good, and `account_id=792` would make a nice `Span` tag

Optional parameters

- Zero or more **references** to related `SpanContext`s, including a shorthand for `ChildOf` and `FollowsFrom` reference types if possible.
- An optional explicit **start timestamp**; if omitted, the current walltime is used by default
- Zero or more **tags**

Returns a `Span` instance that's already started (but not `Finish`ed)

Inject a `SpanContext` into a carrier

Required parameters

- A `SpanContext` instance
- A **format** descriptor (typically but not necessarily a string constant) which tells the `Tracer` implementation how to encode the `SpanContext` in the carrier parameter
- A **carrier**, whose type is dictated by the **format**. The `Tracer` implementation will encode the `SpanContext` in this carrier object according to the **format**.

Extract a `SpanContext` from a carrier

Required parameters

- A **format** descriptor (typically but not necessarily a string constant) which tells the `Tracer` implementation how to decode `SpanContext` from the carrier parameter
- A **carrier**, whose type is dictated by the **format**. The `Tracer` implementation will decode the `SpanContext` from this carrier object according to **format**.

Returns a `SpanContext` instance suitable for use as a **reference** when starting a new `Span` via the `Tracer`.

Note: required formats for injection and extraction

Both injection and extraction rely on an extensible **format** parameter that dictates the type of the associated "carrier" as well as how a `SpanContext` is encoded in that carrier. All of the following **format**s must be supported by all Tracer implementations.

- **Text Map**: an arbitrary string-to-string map with an unrestricted character set for both keys and values
- **HTTP Headers**: a string-to-string map with keys and values that are suitable for use in HTTP headers (a la RFC 7230). In practice, since there is such "diversity" in the way that HTTP headers are treated in the wild, it is strongly recommended that Tracer implementations use a limited HTTP header key space and escape values conservatively.
- **Binary**: a (single) arbitrary binary blob representing a `SpanContext`

`Span`

With the exception of the method to retrieve the `Span`'s `SpanContext`, none of the below may be called after the `Span` is finished.

Retrieve the `Span`'s `SpanContext`

There should be no parameters.

Returns the `SpanContext` for the given `Span`. The returned value may be used even after the `Span` is finished.

Overwrite the operation name

Required parameters

- The new **operation name**, which supersedes whatever was passed in when the `Span` was started

Finish the `Span`

Optional parameters

- An explicit **finish timestamp** for the `Span`; if omitted, the current walltime is used implicitly.

With the exception of the method to retrieve a `Span`'s `SpanContext`, no method may be called on a `Span` instance after it's finished.

Set a `Span` tag

Required parameters

- The tag key, which must be a string
- The tag value, which must be either a string, a boolean value, or a numeric type

Note that the OpenTracing project documents certain "standard tags" that have prescribed semantic meanings.

Log structured data

Required parameters

- One or more key:value pairs, where the keys must be strings and the values may have any type at all. Some OpenTracing implementations may handle more (or more of) certain log values than others.

Optional parameters

- An explicit timestamp. If specified, it must fall between the local start and finish time for the span.

Note that the OpenTracing project documents certain "standard log keys" which have prescribed semantic meanings.

An aside: "Logging" in general, and what it means in OpenTracing

"Logging" is an overloaded term in our industry; one could reasonably argue that all tracing is just a particularly organized form of logging. OpenTracing "logs" are really just key:value maps that describe a particular moment within the context of a Span.

While it's possible to redirect general-purpose process-level logging into OpenTracing, doing so requires care. For instance, logging statements that aren't anchored in specific transactions or traces may not make sense within a tracing system. That said, in enviroments where an overwhelming fraction of conventional logging statements already refer to distributed transactions, `tee`ing that logging data into OpenTracing is reasonable and often beneficial.

The granularity of Span logs is intended to be finer than typical "info"-style logging in process-level logging frameworks. Since tracing systems usually have a smart, all-or-nothing per-trace sampling mechanism, the verbosity within a single trace can be higher than what would be appropriate for a process as a whole — especially when that process contends with high concurrency.

Set a baggage item

Baggage items are key:value string pairs that apply to the given `Span`, its `SpanContext`, and **all `Span`s which directly or transitively *reference* the local `Span`**. That is, baggage items propagate in-band along with the trace itself.

Baggage items enable powerful functionality given a full-stack OpenTracing integration (for example, arbitrary application data from a mobile app can make it, transparently, all the way into the depths of a storage system), and with it some powerful costs: use this feature with care.

Use this feature thoughtfully and with care. Every key and value is copied into every local *and remote* child of the associated Span, and that can add up to a lot of network and cpu overhead.

Required parameters

- The **baggage key**, a string
- The **baggage value**, a string

Get a baggage item

Required parameters

- The **baggage key**, a string

Returns either the corresponding **baggage value**, or some indication that such a value was missing.

`SpanContext`

The `SpanContext` is more of a "concept" than a useful piece of functionality at the generic OpenTracing layer. That said, it is of critical importance to OpenTracing *implementations* and does present a thin API of its own. Most OpenTracing users only interact with `SpanContext` via references when starting new `Span`s, or when injecting/extracting a trace to/from some transport protocol.

In OpenTracing we force `SpanContext` instances to be **immutable** in order to avoid complicated lifetime issues around `Span` finish and references.

Iterate through all baggage items

This is modeled in different ways depending on the language, but semantically the caller should be able to efficiently iterate through all baggage items in one pass given a `SpanContext` instance.

`NoopTracer`

All OpenTracing language APIs must also provide some sort of `NoopTracer` implementation which can be used to flag-control OpenTracing or inject something harmless for tests (et cetera). In some cases (for example, Java) the `NoopTracer` may be in its own packaging artifact.

Optional API Elements

Some languages also provide utilities to pass an active `Span` and/or `SpanContext` around a single process. For instance, `opentracing-go` provides helpers to set and get the active `Span` in Go's `context.Context` mechanism.