Using Apache HTrace

- Overview
- Distributed Tracing in Nutshell
- Distributed Tracing in Apache CXF using Apache HTrace
- Configuring Client
  - Configuring tracing header names
- Configuring Server
  - Configuring tracing header names
- Distributed Tracing in Action: Usage Scenarios
  - Example #1: Client and Server with default distributed tracing configured
  - Example #2: Client and Server with nested trace
  - Example #3: Client and Server trace with timeline
  - Example #4: Client and Server with annotated trace (key/value)
  - Example #5: Client and Server with parallel trace (involving thread pools)
  - Example #6: Client and Server with asynchronous JAX-RS service (server-side)
  - Example #7: Client and Server with asynchronous invocation (client-side)
- Distributed Tracing Apache HTrace and JAX-WS support
- Propagating Trace Details To Logs
- Accessing Apache HTrace APIs
- Future Work

The Apache HTrace project has retired on 2018-04-11 and is not developed anymore. The Apache CXF retired the Apache HTrace integration, starting from 3.3.0 release.

Overview

Apache HTrace is a tracing framework intended for use with distributed systems written in java. Since version 3.1.3, Apache CXF fully supports integration with Apache HTrace, both on client side and server side. This section gives a complete overview on how distributed tracing support is supported in JAX-RS applications built on top of Apache CXF.

Distributed Tracing in Nutshell

Distributed tracing, first described by Google in Dapper, a Large-Scale Distributed Systems Tracing Infrastructure paper became increasingly important topic these days. With microservices (aka SOA) gaining more and more adoption, the typical applications are built using dozens or even hundreds of small, distributed pieces. The end-to-end traceability of the requests (or any kind of work performed on user's behalf) is hard task to accomplish, particularly taking into account asynchronous or/and concurrent invocations. Apache HTrace is inspired by Dapper, a Large-Scale Distributed Systems Tracing Infrastructure paper and essentially is a full-fledged distributed tracing framework.

Distributed tracing is additional instrumentation layer on top of new or existing applications. In terms of distributed tracing, span represents a basic unit of work. For example, executing database query is a span. Spans are identified by a unique 128-bit ID. Spans also have other data, such as descriptions, timelines, key-value annotations, the ID of the span that caused them (parent), and process/tracer ID’s (normally IP address and process name). Spans are started and stopped, and they keep track of their timing information. Once span is created, it should be stopped at some point in the future. In turn, trace is a set of spans forming a tree-like structure. For example, if you are running a JAX-RS service, a trace might be formed by a PUT request and downstream work.

From implementation perspective, and in context of Java applications, spans are attached to their threads (in general, thread which created the span should close it). However it is possible to transfer spans from thread to thread in order to model a complex execution flows. It is also possible to have many spans in the same thread, as long as they are properly created and closed. In the next sections we are going to see the examples of that.

Another two important concepts in context of distributed tracing are span receivers and samplers. Essentially, all spans (including start/stop time, key/value annotations, timelines, ..) should be persisted (or collected) somewhere. Span receiver is a collector within a process that is the destination of spans when a trace is running (it could be a console, local file, data store, ...). Apache HTrace provides span receivers for Apache HBase, Apache Flume and Twitter Zipkin. From other side, samplers allow to control the frequency of the tracing (all the time, never, probability driven, ...). Using the sampler is the way to minimize tracing overhead (or just amount of traces) by limiting them to particular conditions.

Distributed Tracing in Apache CXF using Apache HTrace

Apache CXF is a very popular framework for building services and web APIs. No doubts, it is going to play even more important role in context of microservices architecture letting developers to quickly build and deploy individual JAX-RS/JAX-WS services. As it was just mentioned before, distributed tracing is an essential technique to monitor the application as whole, breaking the request to individual service traces as it goes through and crosses the boundaries of threads, processes and machines.

The current integration of distributed tracing in Apache CXF supports Apache HTrace (4.x+ release branch) only in JAX-RS 2.x applications. From high-level prospective, it consists of three main parts:

- TracerContext (injectable through @Context annotation)
- HTraceProvider (server-side JAX-RS provider) and HTraceClientProvider (client-side JAX-RS provider)
- HTraceFeature (server-side Apache CXF feature to simplify Apache HTrace configuration and integration)
Apache CXF uses HTTP headers to hand off tracing context from the client to the service and from the service to service. Those headers are used internally by HTraceProvider and HTraceClientProvider, but are configurable. The default header names are declared in the TracerHeaders class:

- **X-Span-Id**: contains a current span ID

By default, HTraceProvider will try to pass the currently active span through HTTP headers on each service invocation. If there is no active spans, the new span will be created and passed through HTTP headers on per-invocation basis. Essentially, just registering the HTraceClientProvider on the client and HTraceProvider on the server is enough to have tracing context to be properly passed everywhere. The only configuration part which is necessary are span receiver(s) and sampler(s).

It is also worth to mention the way Apache CXF attaches the description to spans. With regards to the client integration, the description becomes a full URL being invoked prefixed by HTTP method, for example: `GET http://localhost:8282/books`. On the server side integration, the description becomes a relative JAX-RS resource path prefixed by HTTP method, i.e.: `GET books, POST book/123`

## Configuring Client

There are a couple of ways the JAX-RS client could be configured, depending on the client implementation. Apache CXF provides its own WebClient which could be configured just like that (in future versions, there would be a simpler ways to do that using client specific features):

```java
final Map<String, String> properties = new HashMap<String, String>();
properties.put(Tracer.SPAN_RECEIVER_CLASSES_KEY, ...);
properties.put(Tracer.SAMPLER_CLASSES_KEY, ...);

/**
 * For example:
 * properties.put(Tracer.SPAN_RECEIVER_CLASSES_KEY, StandardOutSpanReceiver.class.getName());
 * properties.put(Tracer.SAMPLER_CLASSES_KEY, AlwaysSampler.class.getName());
 */

final Tracer tracer = new Tracer.Builder("web-client")
    .conf(HTraceConfiguration.fromMap(properties))
    .build();

final HTraceClientProvider provider = new HTraceClientProvider(tracer);
final Response response = WebClient
    .create("http://localhost:9000/catalog", Arrays.asList(provider))
    .accept(MediaType.APPLICATION_JSON)
    .get();
```

The configuration based on using the standard JAX-RS Client is very similar:

```java
final Map<String, String> properties = new HashMap<String, String>();
properties.put(Tracer.SPAN_RECEIVER_CLASSES_KEY, ...);
properties.put(Tracer.SAMPLER_CLASSES_KEY, ...);

/**
 * For example:
 * properties.put(Tracer.SPAN_RECEIVER_CLASSES_KEY, StandardOutSpanReceiver.class.getName());
 * properties.put(Tracer.SAMPLER_CLASSES_KEY, AlwaysSampler.class.getName());
 */

final Tracer tracer = new Tracer.Builder("jaxrs-client")
    .conf(HTraceConfiguration.fromMap(properties))
    .build();

final HTraceClientProvider provider = new HTraceClientProvider(tracer);
final Client client = ClientBuilder.newClient().register(provider);

final Response response = client
    .target("http://localhost:9000/catalog")
    .request()
    .accept(MediaType.APPLICATION_JSON)
    .get();
```
Configuring tracing header names

To change the default HTTP header names, used to transfer the tracing context from client to server, it is enough to define one property: **TracerHeaders.HEADER_SPAN_ID**. For example:

```java
final ClientConfiguration config = WebClient.getConfig(client);
config.getRequestContext().put(TracerHeaders.HEADER_SPAN_ID, "CUSTOM_HEADER_SPAN_ID");
```

It is very important to keep client and server HTTP headers configuration in sync, otherwise the server won’t be able to establish the current tracing context properly.

Configuring Server

Server configuration is a bit simpler than the client one thanks to the feature class available, **HTraceFeature**. Depending on the way the Apache CXF is used to configure JAX-RS services, it could be part of JAX-RS application configuration, for example:

```java
@ApplicationPath("/")
public class CatalogApplication extends Application {
  @Override
  public Set<Object> getSingletons() {
    final HashMap<String, String> properties = new HashMap<String, String>();
    properties.put(Tracer.SPAN_RECEIVER_CLASSES_KEY, ...);
    properties.put(Tracer.SAMPLER_CLASSES_KEY, ...);
    /**
      * For example:
      *
      * properties.put(Tracer.SPAN_RECEIVER_CLASSES_KEY, StandardOutSpanReceiver.class.getName());
      * properties.put(Tracer.SAMPLER_CLASSES_KEY, AlwaysSampler.class.getName());
      */
    return new HashSet<Object>(Arrays.asList(new HTraceFeature(HTraceConfiguration.fromMap(properties), "tracer")));
  }
}
```

Or it could be configured using **JAXRSServerFactoryBean** as well, for example:

```java
final Map<String, String> properties = new HashMap<String, String>();
properties.put(Tracer.SPAN_RECEIVER_CLASSES_KEY, ...);
properties.put(Tracer.SAMPLER_CLASSES_KEY, ...);
final JAXRSServerFactoryBean factory =
  RuntimeDelegate.getInstance().createEndpoint("/" application instance "/", JAXRSServerFactoryBean.class);
factory.setFeatures(Arrays.< Feature >asList(new HTraceFeature(HTraceConfiguration.fromMap(properties), "tracer"));
... return factory.create();
```

Once the span receiver(s) and sampler are properly configured, all generated spans are going to be collected and available for analysis and/or visualization.

Configuring tracing header names

As with the client, to change the default HTTP header names, used to establish the tracing context on the server, it is enough to define single property: **TracerHeaders.HEADER_SPAN_ID**. For example:
final Map<String, Object> headers = new HashMap<String, Object>();
headers.put(TracerHeaders.HEADER_SPAN_ID, "CUSTOM_HEADER_SPAN_ID");

final JAXRSServerFactoryBean sf = new JAXRSServerFactoryBean();
sf.setProperties(headers);
...
sf.create();

It is very important to keep client and server HTTP headers configuration in sync, otherwise the server won’t be able to establish the current tracing context properly.

Distributed Tracing In Action: Usage Scenarios

In the following subsections we are going to walk through many different scenarios to illustrate the distributed tracing in action, starting from the simplest ones and finishing with asynchronous JAX-RS services. All examples assume that configuration has been done (see please Configuring Client and Configuring Server sections above).

Example #1: Client and Server with default distributed tracing configured

In the first example we are going to see the effect of using default configuration on the client and on the server, with only HTraceClientProvider and HTraceServerProvider registered. The JAX-RS resource endpoint is pretty basic stubbed method:

```java
@Produces( { MediaType.APPLICATION_JSON } )
@GET
public Collection<Book> getBooks() {
    return Arrays.asList(
        new Book("Apache CXF Web Service Development", "Naveen Balani, Rajeev Hathi")
    );
}
```

The client is as simple as that:

```java
final Response response = client
    .target("http://localhost:8282/books")
    .request()
    .accept(MediaType.APPLICATION_JSON)
    .get();
```

The actual invocation of the request by the client (with process name jaxrsclient/192.168.0.100) and consequent invocation of the service on the server side (process name jaxrsserver/192.168.0.100) is going to generate the following sample traces:

![Sample traces](image)

Example #2: Client and Server with nested trace

In this example server-side implementation of the JAX-RS service is going to call an external system (simulated as a simple delay of 500ms) within its own span. The client-side code stays unchanged.
@Produces( { MediaType.APPLICATION_JSON } )
@GET
public Collection<Book> getBooks(@Context final TracerContext tracer) throws Exception {
    try(final TraceScope scope = tracer.startSpan("Calling External System")) {
        // Simulating a delay of 500ms required to call external system
        Thread.sleep(500);
        return Arrays.asList(new Book("Apache CXF Web Service Development", "Naveen Balani, Rajeev Hathi")
        );
    }
}

The actual invocation of the request by the client (with process name jaxrsclient/192.168.0.100) and consequent invocation of the service on the server side (process name jaxrsserver/192.168.0.100) is going to generate the following sample traces:

![Sample Trace](image)

**Example #3: Client and Server trace with timeline**

In this example server-side implementation of the JAX-RS service is going to add timeline to the active span. The client-side code stays unchanged.

```java
@Produces( { MediaType.APPLICATION_JSON } )
@GET
g public Collection<Book> getBooks(@Context final TracerContext tracer) throws Exception {
    tracer.timeline("Preparing Books");
    // Simulating some work using a delay of 100ms
    Thread.sleep(100);
    return Arrays.asList(new Book("Apache CXF Web Service Development", "Naveen Balani, Rajeev Hathi")
    );
}
```

The actual invocation of the request by the client (with process name jaxrsclient/192.168.0.100) and consequent invocation of the service on the server side (process name jaxrsserver/192.168.0.100) is going to generate the following sample traces:

![Sample Trace](image)

**Example #4: Client and Server with annotated trace (key/value)**

In this example server-side implementation of the JAX-RS service is going to add key/value annotations to the active span. The client-side code stays unchanged.
@Produces({ MediaType.APPLICATION_JSON })
@GET
public Collection<Book> getBooks(@Context final TracerContext tracer) throws Exception {
    final Collection<Book> books = Arrays.asList(
        new Book("Apache CXF Web Service Development", "Naveen Balani, Rajeev Hathi")
    );
    tracer.annotate("# of books", Integer.toString(books.size()));
    return books;
}

The actual invocation of the request by the client (with process name jaxrsclient/192.168.0.100) and consequent invocation of the service on the server side (process name jaxrsserver/192.168.0.100) is going to generate the following sample server trace properties:

<table>
<thead>
<tr>
<th>Relative Time</th>
<th>Duration</th>
<th>Service</th>
<th>Annotation</th>
<th>Host</th>
</tr>
</thead>
<tbody>
<tr>
<td>418.000ms</td>
<td>1.microseconds</td>
<td>jaxrsserver/192.168.0.100</td>
<td>Client Send</td>
<td>192.168.59.3:80</td>
</tr>
<tr>
<td>418.000ms</td>
<td>1.microseconds</td>
<td>jaxrsserver/192.168.0.100</td>
<td>Server Receive</td>
<td>192.168.59.3:80</td>
</tr>
<tr>
<td>434.000ms</td>
<td>1.microseconds</td>
<td>jaxrsserver/192.168.0.100</td>
<td>Server Send</td>
<td>192.168.59.3:80</td>
</tr>
<tr>
<td>434.000ms</td>
<td>1.microseconds</td>
<td>jaxrsserver/192.168.0.100</td>
<td>Client Receive</td>
<td>192.168.59.3:80</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td># of books</td>
<td>1</td>
</tr>
</tbody>
</table>

Example #5: Client and Server with parallel trace (involving thread pools)

In this example server-side implementation of the JAX-RS service is going to offload some work into thread pool and then return the response to the client, simulating parallel execution. The client-side code stays unchanged.
@Produces( { MediaType.APPLICATION_JSON } )
@GET
public Collection<Book> getBooks(@Context final TracerContext tracer) throws Exception {
    final Future<Book> book1 = executor.submit(
        tracer.wrap("Getting Book 1", new Traceable<Book>() {
            public Book call(final TracerContext context) throws Exception {
                // Simulating a delay of 100ms required to call external system
                Thread.sleep(100);
                return new Book("Apache CXF Web Service Development",
                                 "Naveen Balani, Rajeev Hathi");
            }
        })
    );
    final Future<Book> book2 = executor.submit(
        tracer.wrap("Getting Book 2", new Traceable<Book>() {
            public Book call(final TracerContext context) throws Exception {
                // Simulating a delay of 100ms required to call external system
                Thread.sleep(200);
                return new Book("Developing Web Services with Apache CXF and Axis2",
                                 "Kent Ka Iok Tong");
            }
        })
    );
    return Arrays.asList(book1.get(), book2.get());
}

The actual invocation of the request by the client (with process name jaxrsclient/192.168.0.100) and consequent invocation of the service on the server side (process name jaxrsserver/192.168.0.100) is going to generate the following sample traces:

Example #6: Client and Server with asynchronous JAX-RS service (server-side)

In this example server-side implementation of the JAX-RS service is going to be executed asynchronously. It poses a challenge from the tracing prospective as request and response are processed in different threads (in general). At the moment, Apache CXF does not support the transparent tracing spans management (except for default use case) but provides the simple ways to do that (by letting to transfer spans from thread to thread). The client-side code stays unchanged.
The actual invocation of the request by the client (with process name jaxrsclient/192.168.0.100) and consequent invocation of the service on the server side (process name jaxrsserver/192.168.0.100) is going to generate the following sample traces:

Example #7: Client and Server with asynchronous invocation (client-side)

In this example server-side implementation of the JAX-RS service is going to be the default one:

```java
@Produces( { MediaType.APPLICATION_JSON } )
@GET
public Collection<Book> getBooks() {
  return Arrays.asList(
    new Book("Apache CXF Web Service Development", "Naveen Balani, Rajeev Hathi")
  );
}
```

While the JAX-RS client implementation is going to perform the asynchronous invocation:

```java
final Future<Response> future = client
  .target("http://localhost:8282/books")
  .request()
  .accept(MediaType.APPLICATION_JSON)
  .async()
  .get();
```
In this respect, there is no difference from the caller prospective however a bit more work is going under the hood to transfer the active tracing span from JAX-RS client request filter to client response filter as in general those are executed in different threads (similarly to server-side asynchronous JAX-RS resource invocation). The actual invocation of the request by the client (with process name `jaxrsclient/192.168.0.100`) and consequent invocation of the service on the server side (process name `jaxrsserver/192.168.0.100`) is going to generate the following sample traces:

![Sample traces](image)

Distributed Tracing Apache HTrace and JAX-WS support

Distributed tracing in the Apache CXF is built primarily around JAX-RS 2.x implementation. However, JAX-WS is also supported but it requires to write some boiler-plate code and use Apache HTrace API directly (the JAX-WS integration is going to be enhanced in the nearest future). Essentially, from the server-side prospective the in/out interceptors, `HTraceStartInterceptor` and `HTraceStopInterceptor` respectively, should be configured as part of interceptor chains. The `span` receiver should be configured manually though, using Apache HTrace API, for example:

```java
final Map<String, String> properties = new HashMap<String, String>();
final HTraceConfiguration conf = HTraceConfiguration.fromMap(properties);
Trace.addReceiver(new StandardOutSpanReceiver(conf));

final JaxWsServerFactoryBean sf = new JaxWsServerFactoryBean();
...sf.getInInterceptors().add(new HTraceStartInterceptor(Phase.PRE_INVOKE, new AlwaysSampler(conf)));
sf.getOutInterceptors().add(new HTraceStopInterceptor(Phase.PRE_MARSHAL));
...sf.create();
```

Configuring right phases for interceptors is very important. The recommended phase for in-interceptor is `PRE_INVOKE` while for out-interceptor is `PRE_MARSHAL`. If wrong phases are being used, response or/and request headers could be ignored or not processed.

Similarly to the server-side, client-side needs own set of out/in interceptors, `HTraceClientStartInterceptor` and `HTraceClientStopInterceptor`. Please notice the difference from server-side: `HTraceClientStartInterceptor` becomes out-interceptor while `HTraceClientStopInterceptor` becomes in-interceptor. For example:

```java
JaxWsProxyFactoryBean factory = new JaxWsProxyFactoryBean();
...
factory.getOutInterceptors().add(new HTraceClientStartInterceptor(sampler));
factory.getInInterceptors().add(new HTraceClientStopInterceptor());
...factory.create();
```

Propagating Trace Details To Logs

In order to have better correlation between ongoing traces and logs, Apache CXF since 3.1.11 / 3.2.0 releases distributes a helpful extension for Logback users, `LogbackSpanConverter`. This converter can be used to complement log records with current trace details, such as tracer id and span id. For example, here is a simple `logback.xml` configuration file.
In this case the tracing details will be propagated to each log record in following format: <tracer_id>, span: <span id>. For example:

```
[INFO] [-, -] 2017-03-11 14:40:13.603 org.eclipse.jetty.server.Server Started @2731ms
[INFO] [tracer-server/192.168.0.101, span: 6d3e0d975d4c883c7592f4c2317dec22]
2017-03-11 14:40:24.013 com.example.rs.PeopleRestService Getting all employees
[INFO] [tracer-server/192.168.0.101, span: 6d3e0d975d4c883c7592f4c2317dec22]
2017-03-11 14:40:28.017 com.example.rs.PeopleRestService Looking up manager in the DB database
```

The special [-, -] placeholder indicates that no trace details are being available at the moment of logging the record.

### Accessing Apache HTrace APIs

The Apache CXF abstracts as much of the tracer-specific APIs behind TracerContext as possible. However, sometimes there is a need to get access to Apache HTrace APIs in order to leverages the available instrumentations. To make it possible, TracerContext has a dedicated unwrap method which returns underlying Tracer instance.

```java
@GET
@Path("/search")
@Produces(MediaType.APPLICATION_JSON)
public JsonObject search(@QueryParam("q") final String query, @Context final TracerContext tracing) throws Exception {
    final Tracer tracer = tracing.unwrap(Tracer.class);
    // ...
}
```

### Future Work

The Apache CXF is very proud to offer Apache HTrace integration. At the current stage, it was a conscious decision to keep the minimal API and provide the set of necessary features only. However, there is a strong commitment to evolve not only Apache HTrace integration, but the distributed tracing support in general.