Introduction

Working Name
SDK3 Response Time Observability

Revision
#1

Document Authors

- Mike Goldsmith <mike.goldsmith@couchbase.com>

Creation Date
Oct 11, 2019

Proposal Status
DRAFT

Relates To

Motivation

From a diagnostics point of view it is valuable to collect and analyse each operation an SDK performs to help identify problem areas. Each operation goes through a number of phases and tracking each of them both individually and as a whole allows in-depth interrogation.

In addition, it would also be useful to identify if and when an operation takes an unusually long time to complete. This analysis will help aid investigation by providing contextual information that is only available during the operation processing.

API Concepts

Tracing

Tracing is recording details about discrete steps or phases of a request lifecycle, eg request encoding / decoding or dispatching to server. These phases are timed independently and contain additional contextual information. A single request is typically made up of multiple tracing points.

OpenTracing & OpenTelemetry are standardised APIs to structure tracing information in a consistent and predictable manner. This document will not cover all the topics and history of tracing or these APIs (please refer to above links for more complete documentation), and instead will only cover key topics and concepts.

Briefly; tracing data is a nested structure of `Spans` with durations, contextual information and relationships to other `Spans`. A structural example is below:

```
        [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)
```

A good way to visualise the span structure is with an x-axis time graph like the following:

```
––|–––––––|–––––––|–––––––|–––––––|–––––––|–––––––|–––––––|–> time

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

The key concepts of tracing are described below:

Type	Description
Tracer	Top level structure that can be used to create a *SpanBuilder*.
SpanBuilder	Creates and starts *Span*s with contextual information.
SpanContext	Contextual information for a given *Span*. This information can be transported between process / services.

Span	An in-process tracing structure; includes start timestamp, end timestamps and additional contextual information (referred to as *tags* and *baggage*).

SDKs must ship a separate abstraction do not need to carry a OpenTracing mandatory dependency and must supply “addons/modules” to allow plugging OpenTracing and OpenTelemetry subsequently.

Spans

It’s important to uniquely identify operations across each of the services so the full client to server interaction can mapped out. The following table describes each service and how we can identify a single operations.

Service	Description
KV	Operation Opaque NOTE: Opaque is not unique by itself as some SDK implementations are internally incrementing numbers. As part of this RFC, a new Connection ID feature will be added to uniquely identify a client connection with a Couchbase Server and combined with the client's opaque, we have a unique KV operation.
Query	Queries provide a client_context_id field. When a client_context_id field is empty, the SDK will auto-generate a UUID value. *(Not sure if it's used in server logs?)*
Search	Not currently available - created MB-27696 to track.

Analytics	Queries provide a client_context_id field. When a client_context_id field is empty, the SDK will auto-generate a UUID value. *(Not sure if it's used in server logs?)*
Views	Not currently available

NOTE: Both Query and Analytics queries allow the application to override *client_context_id*. Also some SDKs implement a sequence generator that will clash between instances and/or it uses a HTTP library so doesn't know the local host:port.

NOTE: In the case of data structure operations, the top-level operation should be represented within the operation hierarchy with remapped subdoc commands as children. eg MapRemove > Subdoc_Mutate.

Operation Name

Each operation name should represent the activity that is being traced. For example, a KV operation would have a top-level *operation_name* of the operation type, eg GET and a *couchbase.operation_id* tag with the opaque.

Phase Points

The following is a list of phase points that each SDK should implement for each service (KV, View, N1QL, Search and Analytics) and each operation name should match the trace point name.

Name	Description
request_encoding	Request transcoding is when a request content provides content to the server (eg UPSERT) and converts the value from a native object into a common structure (eg JSON).

	NOTE: This can be omitted if the SDK does not perform request encoding or the request does not require have a body.
dispatch_to_server	The time spent sending the request to the server and waiting for a response. This phase encompasses a number of layers, many of which are outside of the SDKs control, for example; task scheduling and writing/reading to the network.
~~response_decoding~~	~~Response transcoding is when a request retrieves content (eg GET) from the server and reconstructs the response body into a native data type.~~ ~~NOTE: This can be omitted if the SDK does not perform response decoding or the response does not require have a body.~~ Due to SDK3 introducing ContentAs which is called by the application to perform the decoding of the bytes into a nativ object, and that the application can wait *any* amount of time between receiving a result and executing ContentAs, it's not a good idea to try to capture decode time at this time.

NOTE: This is different from SDK2 because the native object was decoded as part of the original request, before control was handed back to the application. Attempting to keep references to operations that have been returned to the application and not yet called decode is both dangerous and risks performance. The application may wait an unknown length of time (seconds or minutes), or never call ContentAs.

Client / Connection IDs

Previously, a KV operation was identified using a combination of the local FQDN or IP, port and opaque, however this relies on both the client and server agreeing on network names and is complicated when some components may run under a local NAT, for instance, kubernetes pods.

To address this; the 'hello' request has been extended to be a JSON object with the following properties defined. The server will associate the given *uuid* to the socket and use it when logging slow operations and tracing data.

Name	Description
a (string)	The client agent string - this can not be longer than 200 characters and must be trimmed if exceeds 200. NOTE: This is the same value that was previously sent and includes details such as client name, version, etc.
i (string)	The connection ID is made up of two components separated by a forward slash. The first value will be consistent for all connections in a given client instance, and the second value is per connection. Each part is a randomly generated uint64 number that is formatted as a hex string and padded to 16 characters.

An example key is below

```
{
  "a":"couchbase-net-sdk/2.4.5.0 (clr/4.0.30319.42000) (os/Microsoft Windows NT 
10.0.16299.0)",
  "i":"66388CF5BFCF7522/18CC8791579B567C"
}
```

NOTE: The indentation is for visibility where the JSON that is sent to the server should be compressed to remove whitespace.

Tracing Interface

When implementing Response Time Observability, the SDK must mark all interfaces as volatile (and preferably keep them internal). They are very likely to change in a later time frame and users must not rely on them.

Threshold Logger

To help identify when an operation is exceeds a reasonable threshold, each SDK will implement a *ThresholdLogger* that will track operations that exceed the threshold and log them periodically.

The threshold logger receives completed spans and verifies if an operation has exceeded the given threshold for the operation type. Operations that exceed the threshold are periodically logged with a total count and a sample of the slowest ones per service.

The following are configuration properties for threshold logger.

Property Name	Description
Interval	The interval between executions that processes the collected operation spans. *Default value: 10000 (10 seconds)* *Expressed as milliseconds.*
SampleSize	The maximum number of items to log per service. *Default value: 10*
KVThreshold	The KV operation operation threshold. *Default Value: 500000 (500 milliseconds)*

	Expressed as microseconds.
ViewsThreshold	The View query operation threshold. *Default Value: 1000000 (1 second)* *Expressed as microseconds.*
QueryThreshold	The N1QL query operation threshold. *Default Value: 1000000 (1 second)* *Expressed as microseconds.*
SearchThreshold	The FTS query operation threshold. *Default Value: 1000000 (1 second)* *Expressed as microseconds.*
AnalyticsThreshold	The Analytics query operation threshold. *Default Value: 1000000 (1 second)* *Expressed as microseconds.*

NOTE: The per-service operation and query floors are a guide for default values and how they are expressed. How the SDK receives these values can be represented by whatever is idiomatic to the SDK.

Slow operation summaries are to be logged at the **INFO**, or equivalent, level.

This RFC does not go into how logging should be configured as each SDK and environment is different and the standard SDK logging infrastructure / abstraction should be used. Also, some logging infrastructure implementations provide ways to send log entries to an alternative source, eg LogStash, so this RFC will not go into alternatives storing locations at this time.

See each SDKs documentation on how to configure logging:

https://developer.couchbase.com/documentation/server/current/sdk/dotnet/collecting-information-and-logging.html

ThresholdSummary

When the ThresholdLogger executes to process any collected spans, each of the spans that is to be output should be transformed into this structure.

Property	Description
operation_name	The operation name is operation type that the application used with the eg get, get_and_touch, upsert, etc
last_operation_id	The last operation ID. eg KV opaque, n1ql context ID. Note: for KV operations that use the opaque for the the operation ID should use the 0x prefix to indicate it's a hex value.
last_local_address	The local socket hostname / IP and port, separated by a colon. For example: 255.123.11.134:54321
last_remote_address	The server hostname / IP and port separated by a colon.

	For example: 10.112.170.101:11210
last_local_id	The last connection ID used to send a packet to the server. For example: *66388CF5BFCF7522/18CC8791579B567C*
total_us	The total time taken for the operation. *Expressed as microseconds.*
encode_us	The calculated sum of all encode sub-spans. *Expressed as microseconds.*
dispatch_us	The calculated sum of all dispatch sub-spans. *Expressed as microseconds.*
server_us	The calculated sum of all server duration sub-spans. Only present if server durations are enabled. *Expressed as microseconds.*
last_dispatch_us	The time taken for the last dispatch to server. *Expressed as microseconds.*

NOTE: Any of the above properties can be omitted if there is no appropriate value to be used.

An example output from the threshold logging tracer:

```
[
   {
     "service":"kv",
     "count":75,
     "top":[
       {
         "operation_name":"get",
         "last_operation_id": "0x21",
         "last_local_address":"10.211.55.3:52450",
         "last_remote_address":"10.112.180.101:11210",
         "last_local_id": "66388CF5BFCF7522/18CC8791579B567C",
         "total_duration_us":18908,
         "encode_us":256,
         "dispatch_us":825,
         "server_duration_us":14
       },
       {
         "operaion_name":"set",
         "last_operation_id": "0x22",
         "last_local_address":"10.211.55.3:52450",
         "last_remote_address":"10.112.180.101:11210",
         "last_local_id": "66388CF5BFCF7522/18CC8791579B567C",
         "total_duration_us":11468,
         "encode_us":3832,
         "dispatch_us":565
       },
       {
         "operaion_name":"get",
         "last_operation_id": "0x23",
         "last_local_address":"10.211.55.3:52450",
         "last_remote_address":"10.112.180.101:11210",
         "last_local_id": "66388CF5BFCF7522/18CC8791579B567C",
         "total_duration_us":2996,
         "encode_us":4,
         "dispatch_us":2829
       },
       {
         "operaion_name":"set",
         "last_operation_id": "0x24",
         "last_local_address":"10.211.55.3:52450",
         "last_remote_address":"10.112.180.101:11210",
         "last_local_id": "66388CF5BFCF7522/18CC8791579B567C",
         "total_duration_us":2777,
         "encode_us":15,
         "dispatch_us":2627
       },
       {
         "operaion_name":"set",
         "last_operation_id": "0x25",
         "last_local_address":"10.211.55.3:52450",
         "last_remote_address":"10.112.180.101:11210",
         "last_local_id": "66388CF5BFCF7522/18CC8791579B567C",
         "total_duration_us":1331,
         "encode_us":16,
         "dispatch_us":1206
       }
     ]
   }
 ]
```

Server Durations

As part of the spans that dispatch a request to a server, it is desirable to track the server-side duration to help identify where a requests duration is spent (SDK, Server or in-between) . Most services provide a mechanism to retrieve the duration, as described below.

KV - Flexible Framing Extras

KV operation server duration timings are encoded into the response using "flexible framing extras" (https://github.com/couchbase/kv_engine/blob/master/docs/BinaryProtocol.md#response-header-with-flexible-framing-extras).

Server duration is enabled via Hello negotiation with the feature code (0x10) and when enabled the response packet has the following changes:

- Magic byte is 0x18 (instead of 0x81)
- Header byte index 2 indicates the total extras length
- Framing extras is encoded as a series of variable length FrameInfo structures and is is returned directly after the header and before the regular extras byte

FrameInfo

Each FrameInfo has a descriptor byte followed by a variable number of bytes depending on the type of information.

Bit	Description
0-3	Frame type descriptor
4-7	Length

The initial design of framing extras defines one structure, the server duration. This is a two byte floating number described in microseconds with a variable precision and is encoded/decoded as follows:

```
encoded = (micros * 2) ^ (1.0 / 1.74)
decoded = (encoded ^ 1.74) / 2
```

Decoding examples:

C

```
std::chrono::microseconds Tracer::decodeMicros(uint16_t encoded) const {
  auto usecs = uint64_t(std::pow(encoded, 1.74) / 2);
  return std::chrono::microseconds(usecs);
}
```

.NET (C#)

```
public static double Decode(ushort encoded)
{
  return System.Math.Pow(encoded, 1.74) / 2;
}
```

An example unit test to prove the encoded values are being decoded as expected:

```
[TestCase((ushort) 0, 0.0)]
[TestCase((ushort) 1234, 119635.03533802561)]
[TestCase((ushort) 65535, 120125042.10125735)]
public void Can_Decode_Server_Duration(ushort encoded, double expected)
{
  var decoded = Math.Pow(encoded, 1.74) / 2;
  Assert.AreEqual(expected, decoded);
}
```

N1QL, Analytics Query Server Duration

N1QL, FTS and Analytics queries are all dispatched over HTTP with both the request and response encoded as JSON. The query duration time is returned in the response body JSON in the *metrics*.*elapsedTime* property for N1QL and Analytics and the root property *took* for FTS. This value is expressed as microseconds, so is the same precision as other metrics.

Because the request durations form part of the response body it cannot be read independently of the response body. This is troublesome for streamed responses as the server duration timings may not be readable until after all results have been read, which could be a long time depending on how the application processes them. To help distinguish the different phases, a streamed request should be made up of the following spans:

Phase Name	Description
request_encoding	The amount of time taken to prepare the JSON request body ready for dispatching to the server.
dispatch_to_server	The amount of time taken between sending the request to the server and when the response HTTP header has been read. The response body has not been processed yet.

N1QL Profiling Baggage (optional)

N1QL returns additional metrics related to a query execution as part of the payload and can be added as tags to the query span. A summary of the default properties is below:

Property	Description
elapsedTime	The time taken between receiving the request and returning a response. The property is used for the server-duration property described above.

	Expressed as milliseconds.
executionTime	The time taken to execute the query, not taking into account preparing the response. *Expressed as milliseconds.*
resultCount	The number of result sets.
resultSize	The number of rows.

N1QL also provides an enhanced profiling mode where additional details are recorded and returned in the response JSON. Profiling is enabled by setting a request property and the extra information described in the enhanced profiling could be converted into Span Tags to provide in-depth analysis but is not required at this stage.

As an example, the available N1QL tags may look like this. Because the profile details are returned in a multi-level JSON structure, each level must be concatenated together using an underscore. Primitive types (integers, strings, boolean) map easily in a one to one with their tag name and arrays should be concatenated together with a double underscore '__'. Also note any non alphanumeric characters should be trimmed, eg "~version" becomes "version".

For example, given the below structure, it would create the following tags:

```
"profile": {
  "phaseTimes": {
    "authorize": "1.544104ms"
  },
  "phaseCounts": {
    "fetch": 1,
    "primaryScan": 1
  },
  "~version": [
    "2.0.0-N1QL",
    "5.1.0-1256-enterprise"
  ]
}
```

Name	Value
profile_phaseTimes_authorise	"1.544104ms"
profile_phaseCounts_fetch	1
profile_phaseCounts_primaryScan	1
profile_version	"2.0.0-N1QL__5.1.0-1256-enterprise"

Views

There is not currently a way to retrieve the server duration for a view query.

Enhancements to Timeout Exceptions

Currently, error handling may return messages that are generic in nature. Tracing information must be added to key messages such as Timeouts to help correlate data between when a timeout happens and when the response is received from the server.

Consider this line of Java code:

```
JsonDocument fetched = bucket.get("u:king_arthur", 1, TimeUnit.MICROSECONDS);
```

It will always throw a RuntimeException, since the timeout specified is too short for even a kernel context switch, let alone to get network IO done. When catching the exception and printing the message, one gets "java.util.concurrent.TimeoutException". This does not help the user understand possible causes.

Tracing information will be appended log messages, with the format:

```
{existing message} {compresed_json_object}
```

For example, the above exception log entry would look like:

```
java.lang.concurrent.TimeoutException:
{"s":"kv:get","i":"0x7b1","i":"002c2b0d250e6fc5/002c2b0c723e11c5","b":"default","l":"
_10.157.77.74:16584"_,"r":"my.server.io:11210"," t":2500000}
```

Where appropriate, Timeout exceptions should be extended to append the following information:

Property	Description
Operation name	One of: kv, view, n1ql, fts, cbas
Operation ID	The operation's unique identifier, as described in the trace information.
Local endpoint	The local hostname / IP and port, formatted with a colon. For example: localhost:12345
Time observed before timeout	The amount if time observed before the timeout occurred. Expressed as microseconds.
Server last dispatched to	The last server the operation was dispatched to. Note this may be empty as a server may not have been selected if a timeout is that low.

Orphaned Responses

For packets that come in whose request may have timed out, it is important to be able to know if the server duration was related directly or indirectly. However, we also do not want to overwhelm the log when there may be a big burst of operations that have timed out and the responses come in delayed putting the system into a pathological state.

For this, we have specific handling of orphaned responses.

NOTE: Orphan response is not related to Threshold logging and works independently. It is possible to replace the Tracer implementation and still have orphan response logging enabled.

Terminology

Orphan response -> A response received on a connection whose original requester (or requesters in the case of request deduplication) is no longer in scope.

We only maintain the key information from each orphan, utilising the Operation Context (described below) structure.

Implementation Notes

Prerequisites: Generate a random uint64 client instance and connection IDs.

In implementation, an SDK will change any timeout return messages to include the operation type, instance ID and opaque along with the hostname and timeout duration specified by the app. Separately, as network requests complete that are not associated with any operations in scope, aggregate them for logging, limiting the number to the top 10 by reported server duration on a per ten second basis.

This results in timed out log messages from the application which can be further categorized into a few possible causes with further correlation:

Bucket 1: No orphan responses are received and the connection is eventually disrupted. This likely means network issues.

Bucket 2: Only some map up to an orphan response. All orphan responses are from one server. The topmost orphan response has a large server duration and the subsequent responses are fast (or become fast) and have increasing opaques for this individual client log. This means one slow operation caused congestion per MB-10291 (or other causes).

Bucket 3: Only some map up to an orphan response. orphan responses are from multiple servers. The orphan responses have small server durations relative to the timeout value. This means general environmental issues at the client, the network or at the servers not severe enough to cause connection disruption caused the issue.

Bucket 4: Only some map up to an orphan response. orphan responses are from one server. The orphan responses have small server durations relative to the timeout value.

This means an environmental problem (e.g., THP being enabled) at the server in question is causing the issue.

Because there is no way to order the orphaned responses, the insertion order should be maintained up to the sample size.

Example Telemetry

The following table describes the properties that will form a JSON object that will be appended after the existing timeout message:

Operation Context Properties

Property	Description
s (string)	Service Type - one of: kv, view, n1ql, search, analytics KV operations should be in the format: kv:{operation_type} eg kv:get, kv:upsert
i (string)	Operation ID - the service specific identifier for a given operation KV use opaque (with a 0x prefix and hex formatted) eg "0x7b1" N1QL & Analytics use the context ID which is a GUID.
c (string)	Connection ID (optional) eg KV would use the new connection ID described above
b (string)	Bucket name (optional)

l (string)	Local endpoint host & port (optional)
r (string)	Remote endpoint host and port
t (string)	Timeout value (optional) Should be present for timeout contexts
d (string)	Server Duration (optional) Should be present for orphaned contexts

A Java example would look like this:

```
java.lang.concurrent.TimeoutException
{"s":"kv:get","i":"0x7b1",c":"002c2b0d250e6fc5/002c2b0c723e11c5","b":"default","r":"_
10.157.77.74:16584"_,"r":"my.server.io:11210"," t":2500000}
```

Logged orphaned responses, here with a sample size of 2, every ten seconds while there are orphans to log:

```
2018-01-08 15:36:51,903 [16] WARN Couchbase.Core.KVNode - Orphaned responses
observed:
[{"service":"kv","count":2,"top":[{"s":"kv:get","i":"0x71b","c":"002c2b0d250e6fc5/002
c2b0c723e11c5","l":"192.168.1.101:11210","r":"10.112.181.101:12110","d":123},{"s":kv:
upsert","i":"0x71c","c":"121345/13321/7612","l":"192.168.1.101:11210","r":"10.112.181
.101:12110","d":43}]}]
```

Example Orphaned response output:

https://gist.github.com/MikeGoldsmith/147b85e960378a47bcb0169581952af1

Given the two output examples above, the log entries can be used to correlate operations to identify if a timeout was caused by server duration or was due to something else. The operation_id 0x_0769020a and connection ID_ *002c2b0d250e6fc5/002c2b0c723e11c5* appears in both logs.

Orphaned operation summaries are to be logged at the WARN, or equivalent, level.

Signoff

Language	**Team Member**	**Signoff Date**	**Revision**
Node.js	Brett Lawson		
Go	Charles Dixon		
Connectors	David Nault		
Python	Ellis Breen		
Scala	Graham Pople		
.NET	Jeffry Morris		
Java	Michael Nitschinger		
C	Sergey Avseyev		
Ruby	Sergey Avseyev		