Chapter 1

Developing a Client Library

This chapter is relevant for developers who are creating their own client library that communicates with the Couchbase Server. For instance, developers creating a library for language or framework that is not yet supported by Couchbase would be interested in this content.

Couchbase SDKs provide an abstraction layer your web application will use to communicate with a cluster. Your application logic does not need to contain logic about navigating information in a cluster, nor does it need much additional code for handling data requests.

Once your client makes calls into your client library the following should be handled automatically at the SDK level:

- maintain direct communications with the cluster,
- determining cluster topology,
- distribute requests to the cluster; automatically make reads and writes to the correct node in a cluster,
- direct and redirect requests based on topology changes,
- handle and direct requests appropriately during a failover.

The cluster exposes its services using several protocols. For management and query APIs the client must be able to communicate HTTP protocol with JSON encoding for payloads (in most of the cases, but in several places more simple URL-form encoding might be required. For data operations, like Upsert or Get, more efficient binary protocol must be implemened. In this document we will call this protocol MCBP (stands for Memcache Binary Protocol). The protocol derives from memcache binary protocol, although it does not aim to be compatible with existing memcache client libraries.

This document uses OCaml code snippets, because at the time of writing there is no supported SDK for this language, therefore the implementation will be free from legacy implementation decisions. For the same reason we assume that the cluster exposes APIs that became available in Couchbase Server 7.0 and don't cover any deprecated features.

1.1 Bootstrap process

This section describes a number of different behaviours within a client. We first start by discussing how individual MCBP connections should be established and authenticated. We then further

describe the overall connection behaviour for an entire cluster, the process for connecting to a particular bucket, along with a number of required optimizations to these processes to enhance the applications performance in various scenarios.

1.1.1 MCBP Connection Sequence

Individual MCBP connections are described as being "connected" only once the connection has been completely initialized at a protocol level (`HELLO`, `ERRMAP`, `AUTH`). A connection can be either a cluster connection or a bucket connection, the differentiation solely being whether a `SELECT_BUCKET` has yet been performed on the connection. Note that fetching a configuration from a connection is NOT considered part of the initialization of a memcached connection and is explicitly discussed outside the scope of the memcached initialization, though in the case of the pipelining that is performed, a valid operation may be sent as part of the initial connection pipeline sequence in order to reduce new-connection operational round trip time (RTT) to 1.

All MCBP connections should first establish an appropriate network connection to the target node via TCP. Once the connection is established, the client must perform a sequence of commands to initialize the connection. This sequence must be performed in the order given in order to allow correct behaviour. This sequence MUST be dispatched in a single batch (except in the case of `SASL_CONTINUE`) without waiting for responses. The entire batch of initialization commands will be handled in sequence upon receiving responses, as described further below in this section.

`HELLO` A HELLO operation must be dispatched to the node with a list of requested features along with a client and connection identifiers as defined in "Client Connection ID".

1.2 Unified Client Agent

The SDKs send an actual HTTP `User-Agent` header to services like query or analytics, but they also need to send a trimmed-down version via a MCBP `HELLO` negotiation.

```
cb-<SDK-Identifier>/<Version> (<System-Information>)
```

1.2.1 SDK-Identifier

Mandatory field. The identifier allows to uniquely distinguish between the different SDKs used. The following identifiers must be followed by the implementations:

- .NET: `dotnet`
- Go: `golang`
- NodeJS: `nodejs`
- C: `lcb`
- PHP: `php`
- Python: `python`
- Java: `java`
- Scala: `scala`

- Kotlin: `kotlin`
- JVM Core-IO: `jvm-core`
- Ruby `ruby`
- Rust `rust`

We are going to use `ocaml`, because the document uses OCaml as example implementation.

1.2.2 Version

Mandatory field. The version must include the significant digits of the current followed SemVer standard and can optionally include a platform-dependent suffix.

`major.minor.patch[suffix]`

If for whatever reason a version cannot be determined, `0.0.0` must be used.

1.2.3 System-Information

Optional field. The system information is the most flexible piece since it will inevitably be different for each platform. Nonetheless a structure like the following must be followed to help with automated parsing:

`(<os>; <platform>)`

More entries might be added with `;` (semicolon) as separator, but we recommend including OS and Platform descriptions as first two entries. Note that if system infromation is not available, the surrounding braces must be omitted completely. If only one entry is available, the `;` separator must be omitted.

os The OS identifier, maybe including the architecture. For example: `Unix/64`

platform If the SDK is running in a platform/runtime, include information. This could also include application server information. For example: `OCaml/4.11.1`

1.3 Client Connection ID

Each client instance must have unique identifier. In addition, it must generate unique identifiers for every network connection.

Each MCBP connection MUST serialize both client and connection identifiers using the following JSON format, and send it as a key with `HELLO` command during connection sequence. The server will associate the given uuid to the socket and use it when logging slow operations and tracing data.

`a` (string) The unified client agent string — this can not be longer than 200 characters and must be trimmed if exceeds 200.

`i` (string) The connection ID is be 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.

The example key is below

```
{
  "a":"cb-ocaml/1.0.0 (Unix; OCaml/4.11.1)",
  "i":"000ae6bd34d5ee12/48e7bce204c4e9ad"
}
```