Client Protocol Implementation Guide

Release Date
25.01.2016

Purpose

This document is a guide for developers who plan to implement a client for Hazelcast in any programming language. It heavily uses concepts and definitions from the Hazelcast Open Binary Client Protocol document. It also assumes that you, the developer, have a basic understanding of Hazelcast.

You can write a client in any programming language you choose, so long as the language provides means of TCP/IP socket communication.

This guide will walk you through creating a client that can complete standard operations on distributed objects. It provides examples. You can find an example implementation of a client written in Python at hazelcast-python-client.

Introduction to Hazelcast Open Binary Client Protocol

The Hazelcast Open Binary Client Protocol is a binary protocol specification to identify the on-the-wire format of the data communicated between any client and the Hazelcast cluster. The protocol uses Little-Endian byte order for the wire data binary format.

Hazelcast Open Binary Client Protocol is a client-server protocol. The operations are always initiated by the client. Every request between the client and server is called a “message”. Every message is uniquely identified by a Correlation ID. The Correlation ID is a 4-byte unsigned integer. For each message the client sends to the server, the message must include a unique Correlation ID. Whenever a cluster member sends a response to the request, whether instantly or after the client has registered for an event, the response will have the same Correlation ID as the initial request. The event type response messages always use the Correlation ID initially provided by the registration request message.

The messages may be one of the following types.

1. Request Message
2. Response Message
3. Event Message

Request messages are exclusively sent from the client. All message types must contain a message header. The message header details are described in the message header section of Hazelcast Open Binary Client Protocol.

Note that even if the value for a given field does not require all of the bytes of the data type, it still must be sent over the line as the data type. Since Python manages the number of bytes that an integer uses behind the scenes, you must explicitly allocate the memory for them using the `c types` library as follows:

```
newsize=ctypes.c_uint32(self.size)
newversion=ctypes.c_uint8(self.version)
newflag=ctypes.c_uint8(self.flag)
newtype=ctypes.c_uint16(self.optype)
newcorrelation=ctypes.c_uint64(self.correlation)
newpartition=ctypes.c_int32(self.partition)
newoffset=ctypes.c_uint16(self.data_offset)
```

Figure 1: Explicitly casting variables to certain types

You must take this issue into consideration when you use a programming language like Python or Swift that does memory management behind the scenes.

For a single request, the response message may be a one time response message or it may be in the form of multiple response messages (e.g. events). Events are updates that are communicated to the clients which are registered for listening to specific state changes in the cluster. Your client registers for an event and gets a registration ID (this is different from CorrelationID, which may be used to cancel the registration). The repetitive updates that your client may get from the server always use the same Correlation ID as the original request Correlation ID.

Client Lifecycle

A client typically performs the following steps:

1. Initiate connection to the cluster.
2. Authenticate.
3. Send periodic updates (Heartbeat).
4. Retrieve partition list.
5. Send operation messages and receive responses.
6. Get updates on cluster member changes.
7. Close opened connections.

Please note that a client does not need to implement each possible request and response. A client, as a minimum, may only implement the authentication request, the heartbeat ping message, and another simple request (such as adding an item to a set).

Please see the example code below:

```
client=HazelcastClient()
myQueue=client.getQueue("queue")
myQueue.put("item")
print myQueue.size()
sys.exit("Exiting")
```

Figure 2: Example Client Lifecycle

In this example, a fully-implemented client would:

- Initialize a connection to the server.
- Authenticate itself to the server.
- Add a listener for cluster membership.
- Create a proxy for the IQueue data structure that is named “queue”.
- Get the partitions in the cluster.
- Request to add a String “item” to the queue.
- Request the size of the queue.

- Exit, closing all sockets it opened at the first step.

The steps are not inherently obvious, as there are more steps the client takes than the number of lines in the code. For example, the first three steps take place when the client first connects to the server. When the client calls getQueue(), a proxy is created for the IQueue “queue”. Next, the client gets the partition list. Using this information, it puts the String “item” onto the Queue. After requesting the size of the queue, it prints the size and the process terminates.

The following sections detail each of the possible steps of a client lifecycle.

Initiating Connection to the Cluster

Your client needs to connect to any cluster member in order to start communication. You should provide your client with the IP address and the port number of the member that you want the client to connect with. The client can connect to any member in the cluster.

Your client opens a TCP/IP connection to the cluster member. Your client should initially send 3 bytes to identify the protocol version. Currently, only the following version string can be used: “CB2”. Failure to provide these bytes correctly will cause non-working communication, i.e. the connection is not dropped but the messages will not be interpreted as expected.

The characters in all the strings are ASCII encoded and the first character of the string is sent first on the wire. For example, the client sends the “CB2” string as character ‘C’ (byte value 0x43) first, then character ‘B’ (byte value 0x42), and finally the character “2” (byte value 0x32).

Authenticating

Your client has to perform an authentication for all connections it opens to the cluster. If your client fails to provide the correct authentication parameters, the server disconnects the client. There are two types of authentications:

1. Username/Password authentication: “Authentication” (See Hazelcast Open Binary Client Protocol for message fields) message is used for this authentication. The response is AuthenticationResult message. The result contains the address, UUID and owner UUID information.
2. Custom credentials authentication: “Custom Authentication” message is used. This method sends a custom authentication credentials as a byte array. The response is the same as the username/password authentication.

One of the parameters in the Authentication request is the boolean flag “isOwnerConnection”. Each client is associated with an owner server member; this owner server tracks the availability of the client and if it detects that the client does not exist, it cleans up any resource allocated for that client. The locks acquired by the client is one such resource. You need to set the “isOwnerConnection” to true if the connection will be the owner connection.

Sending Periodic Updates

Your client should send periodic heartbeat messages if no data is transmitted for a timeout duration. This duration is configured at the cluster server members with the "hazelcast.client.max.no.heartbeat.seconds" system property. The default timeout value is 300 seconds. If any data is sent on the connection during this time, there is no need for your client to send an extra heartbeat message. The heartbeat message is of type “ping” message. It has no payload and the response is a header only message. It is important to note that the cluster responding to requests does not count as a heartbeat. Your client is always responsible for maintaining the heartbeat in the connection.

This mechanism is how the server members detect unresponsive clients. If your client fails to send these periodic updates, the server closes the client connection.

Your client may also use the ping messages to detect unresponsive server nodes. For example, if the server does not send a response to a certain number of ping requests, then you can have your client decide to switch to another server member or close the connection.

Getting the Partition List

Your client can get the partition list of the cluster. The partition list tells the client which member handles which partition key. By default there are 271 partitions. Note that this value can be configured in server side. Partition id s starts from zero. For 271 partitions, ids are 0 to 270. Your client can use this information to send the related requests to the responsible member (for the request key if exists) directly for processing.

The request message you have your client send to get the partition list is the “Get Partitions” request message. This message has no payload. The response is the “Get Partitions Result” response message. This response contains the full cluster member list and the member-partition ownership information (using the owner indexes array).

To compute the specific partition ID, see the “Partition ID” under the Message Header section in Hazelcast Open Binary Client Protocol.

Sending and Receiving Operation Messages and Responses

Your client sends request messages and receives and interprets the responses for all of its operations. you can find the possible request/response message types and their field details documented in Hazelcast Open Binary Client Protocol under “Operation Message Types and Responses”.

For any request where data may be placed on a variable partition, your client should submit the request with the correct partition ID. If the request cannot have multiple partitions associated with it, set the partition ID to -1. However, if the partition ID is set to -1 when the client requests an operation where a partition ID is expected, the member will throw an error that says “partition ID cannot be -1”.

Your client can work as a smart client or as a dummy client. See Implementing a Smart Client versus a Dummy Client for additional details.

The response to a client's request message is always one of the following:

1. Regular response message. The response is the message as listed in the protocol specification for the specific request message type.
2. An error message. The response message is of type “ExceptionResultParameters” (see Hazelcast Open Binary Client Protocol for message field details). The cluster member may return this response message with the className field set as “com.hazelcast.core.HazelcastInstanceNotActiveException”. This means that the cluster member is not yet ready for accepting any requests and hence you may have your client either retry the connection later or send the request to another cluster member.

Performing Operations and Requests on a Distributed Data Structure

Performing a “put” operation on the map requires more than just sending one message. First, your client must create a “proxy” for the data structure. A proxy is helpful in two ways:

1. It ensures that the client is actually connected to the object, so it can perform operations on it.
2. It provides you, the developer, with an easy way to mimic the Hazelcast API by creating a Proxy class for an object with methods for every operation they want to implement. We provide an example of this in the Python client’s topic class.

```
class TopicProxy(object):


  def __init__(self,title,connfamily):
     self.title=title
     self.connection=connfamily
     firstpack=proxycodec.createProxy(self.title,"hz:impl:topicService")
     self.connection.adjustCorrelationId(firstpack)
     self.connection.sendPackage(firstpack.encodeMessage())
     response=self.connection.getPackageWithCorrelationId(firstpack.correlation,True)
     if response is not None:
        print "Initialized and connected proxy!"
     else:
        print "Unable to connect to server."


  def publish(self,data):
     msg=topiccodec.TopicPublishCodec.encodeRequest(encode.encodestring(self.title),data)
     retryable=msg.retryable
     self.connection.adjustCorrelationId(msg)


     correlationid=msg.correlation
     self.connection.sendPackage(msg.encodeMessage())
     response=self.connection.getPackageWithCorrelationId(msg.correlation,retryable)
     msg2=ClientMessage.decodeMessage(response)


     return topiccodec.TopicPublishCodec.decodeResponse(msg2)
```

Figure 3: Topic Proxy example

You can easily model a general class from the example above for each data structure you would want to implement. Of course, this is only an example of using proxies. As long as the appropriate bytes are exchanged, the actual code holds little relevance.

You can use this proxy to publish a string to the topic.

Python code:

```
client=HazelcastClient()
mytopic=client.getTopic("my-topic")
update=util.hzstringbytes("Python update on topic")
mytopic.publish(update)
```

Java code:

```
public class Member implements MessageListener<String> {


  public void onMessage(Message<String> message) {
     String sMsg=message.getMessageObject();
     System.out.println("RECEIVED MESSAGE: "+sMsg);
  }


  public static void main(String[] args) throws Exception {
     HazelcastInstance h1 = Hazelcast.newHazelcastInstance();


     ITopic<String> topic=h1.getTopic("my-topic");
     topic.addMessageListener(new Member());
  }


}
```

Figure 4: Publishing a topic update

To use the above example, you first start up a Hazelcast member that implements this MessageListener interface. This will allow you to see updates from the topic. This member subscribes to the topic “my-topic”. Then, run the python script. The member will print “RECEIVED MESSAGE: Python update on topic”.

Getting Updates on Cluster Member Changes

Your client can register for cluster member list updates using the “Cluster Membership Listener” request message, and thus your client can retrieve the list of member server addresses in the cluster. In the Python API, this is the `addMembershipListener` function. The response to this request message are events. The response events can be one of the following:

- The full member list.
- A specific member add/removal to the cluster.
- A member server attribute change.

This information is needed if your client operates as a smart client.

Closing Opened Connections

Your client MUST be the one to close the connection. It can do this by closing the socket it used to connect to the server. The server will release any resources it allocated to the client. If your client does not perform a regular socket close, the server closes the connection and cleans the associated resources after missing the periodic heartbeat messages from the client.

Message Fragmentation

The Hazelcast Open Binary Client Protocol is designed to handle the transfer of large data in a single message. This is achieved using fragmentation and reassembly. If your client sends a large piece of data (e.g. a Map `putAll` request), the message can be divided into multiple pieces (each one is called a fragment). The “Flags” header field contains the two bits that indicate whether the message is the first fragment (BEGIN flag is set), the last fragment (END flag is set), or unfragmented (both BEGIN and END flags are set). The client implementation needs to append the payload parts of each fragment together and compose the actual message. Your client is free to fragment the data in any way you prefer; the cluster server member will be able to reassemble the message if the “flags” field is correctly set.

Performing Serialization

Some operations require you to send/receive binary data in the message payload. These binary data are usually the key and value(s) for the request/response. It is your responsibility to implement a proper serialization for converting the user objects to/from binary format. The "Hazelcast Reference Manual" describes the serialization types used in Hazelcast (please see the Serialization chapter in the Reference Manual). You may implement one of these serializations or develop a custom serialization.

Example usage:

Map get operation - You need to send the key as binary data to the server. Hence you can directly design your client API in terms of binary data, or you should use serialization to convert the user provided key to binary data. The response to the Map get request is also binary data; this data is exactly the same data provided by the user in the Map put operation as the “value” data for the key.

If your client (e.g. Python client) wants to retrieve the value stored in the Map object by another client (e.g. Java client), the clients should both use the same type of serializations for meaningful results.

Please note that the use of Predicates and Query requires that the server be able to deserialize the binary data. Hence, if you want to support these features you should implement Hazelcast serialization support through portable or custom serialization.

Serialization is not in the scope of this document, the protocol works with binary data and it is up to you how you generate the binary data. Please see the Serialization chapter in the Reference Manual for the details of Hazelcast serialization.

Adding Listeners

Listeners communicate multiple responses to a client. You have your client use one of the listener registration messages, e.g. cluster member changes, to register for updates at the cluster. Map entry added listeners and cluster membership listeners are examples of such listeners. The consecutive responses sent from the member to the client uses the same correlation ID as provided by the original request correlation ID. You should not reuse this correlation ID unless it deregisters the listener. Registering for a listener is as simple as the following example:

```
client=HazelcastClient()
map=client.getMap("my-map")
entrylistener=myentrylistener()
map.addEntryListener(entrylistener, True)
```

Figure 5: registering for updates

The addEntryListener request parameters are mapName and includeValue. “True” in the last line indicates that includeValue is set to true. However, entrylistener is not the mapName. The name of a particular data structure is typically initialized when a proxy is initialized; see the above code block. The entrylistener is not written over the wire, but rather it is registered in the client as a callback for when the client receives updates. Your client knows it is properly registered for updates when it receives a response from the cluster in response to its add listener request.

Receiving Updates

When your client receives a response from a member, it should check the FLAGS field on the received message header to see if the EVENT flag is set. If it is set, the message is an event. Otherwise, it is a regular response message. Here is a generic pseudocode for processing a response:

```
For each received client message do {
        Check the FLAGS field to see if the message is of type EVENT
      if (EVENT) then
            {
                  Find the listener callback for the correlationID in the message.
                        Some parts of the Payload may be of typeData (binary). if  desired, these fields may be deserialised.
                        Deliver the received event  message to the callback.
           } // end of if
           // do other processing if any
} // end of do
```

Figure 6: Generic Pseudocode of processing responses

For performance considerations, it is suggested that you design the listener callback so that it does not block the IO thread for a long time. Either have the callback method guarantee to return very quickly, or use another mechanism to deliver the event to the user (e.g. a separate delivery thread).

Providing Error Codes

The server may return an error response. In this case, the payload of the server's message will contain the error message. You may choose to provide direct error codes to your client API user (the one who uses your client's API) or use some other techniques such as exceptions to communicate the error to your API user. See the "Error Message" section of the Protocol Messages document for details of this message. The error response messages are identified by the error code. The full list of error codes are documented in the Protocol Messages document.

For every request message that your client sends to the server, your client may receive an error response, hence your client should handle both the successful response from the server as well as the error situation. You may design different ways to communicate the error to your API users.

Handling Timeouts and Retry

Your client should be able to handle situations where the member may not be able to return the response in an expected time interval. Even if the response to a specific message is not received, the user of your client may or may not retry the request. In the case where the operation was actually executed at the cluster member, the retry operation may cause unwanted effects on the data. Your client should NOT have the retry operation reuse the same correlation ID. Each request, including the retry requests, should use a unique correlation ID.

Common Questions

Maximum Number of Connected Clients Per Server Member

The server member does not expose any limitation on the number of connected clients. The server reserves a certain number of threads for processing the key based and non-key based operations. If too many operation requests go to one server, they are processed with the available processing power of that member server. The number of threads on the member server is configurable. Please see the Hazelcast Reference Manual for configuration details.

Lock Lifecycle Management

When your client is disconnected from the server for some reason (e.g. the client process crashes, the client gracefully terminates by closing the TCP connections, etc.), the cluster cleans the locks acquired by the client.

Implementing a Smart Client versus a Dummy Client

Implementing a dummy client is far simpler than implementing a smart client. To implement a dummy client, you just need to establish a single connection to a member, properly authenticate it to a member, and make it able to request one operation and to decode that same operation.

Implementing a smart client presents much more of a challenge:

- A smart client must register for member updates to connect to members that join a cluster after the client has already started. In addition, it also must know when to drop a connection if a member leaves a cluster.
- A smart client must submit a “Get Partitions” request to compute the partition ID of a request associated with distributed data.
- A smart client must submit requests to the appropriate member that handles the client requests.

Thus, a smart client, while more powerful and robust, is more difficult to implement than a dummy client. A dummy client can still send requests to the wrong member and receive correct responses. The biggest danger in opting for a dummy client over a smart client is if the member the client is connected to goes down. The client will continue to send requests to the member, but the member will not respond. As such, a smart client should be used in a production environment.

Definitions and Acronyms

Terminology	Definition
Member	Hazelcast server instance
Cluster	A virtual environment formed by Hazelcast nodes communicating with each other.

References

1. Hazelcast Open Binary Client Protocol - Version 1.0
2. Hazelcast Reference Manual