OpenCluster

Developers guide for communicating with OpenCluster.

Introduction

Data added to the cluster is distributed over a number of OpenCluster nodes. This document describes in detail the communication protocols and processes that occur for a client to add and retreive data in the cluster.

Required Concepts

Before getting into the details of the communication protocol, it is important to first understand how data is distributed safely in the cluster without requiring a master node that controls and tracks the location of data.

Hashkeys

When storing or retrieving data from the cluster, items are normally given a name. That name is converted into a 32-bit number. We use the FNV hashing algorithm to convert a string into a 32-bit number. For example, the key “something” will result in a hashkey of 2118605163 (7e47596b in hex). This hashkey is used to determine which server to send the request for.

Buckets and Hashmasks

Buckets and Hashmasks are essentially the same thing, however, there are some slight elaborations to point out. A Bucket can be thought of as a physical representation of a hashmask, and a hashmask can describe part or all of a bucket.

Each node, when it is connected to the cluster will want to store some of the data that is in the cluster. Clients need to be able to determine which node in the cluster contains the actual data we are looking for.

The Mask

The mask is a 32-bit integer (technically unsigned, but we would never need to use all the bits, so a signed integer should suffice). The mask is utilised to use the hashkey to determine which bucket the data will be in, which also lets us know which server is managing that bucket.

A mask will normally start out as 0x0000000F. This gives us a possible 16 buckets.

If we use the above key from “something” which is 7e47596b, and do a binary AND on it with the mask, we end up with 0x0000000b.

Looking up the server

The client uses those hashmask entries in an array, to associate a hashmask with a server node. When server “A” sends a message saying that it has 4 primary buckets 0002, 0004, 000d, 000e with a mask of 000F, then the client will have an array of 16 elements, and mark those entries with that server connection.

So when it needs to look up a key of “something” which resolves to hash of 0x7e47596b, it AND's that value by the mask (0x0000000F) and ends up with 0x0000000b. It looks up entry 0x0b in the array and finds the server that is responsible for that key.

Primary and Secondary Buckets

The system is designed for redundancy and recovery by making sure there is more than one copy of the data being stored. The client will therefore know which server is responsible, and if it cannot connect to that server, which server to try next. It is not required for a client to maintain the list of secondary servers, however, if the primary server stops responding, it allows the triggering of failover when clients start asking the backup server for data.

NOTE: All reads and writes should go to the primary server. The secondary server should not be queried unless communication is failing with the primary server. The secondary buckets are not used for load balancing. Always communicate with the server that maintains the primary copy of the bucket.

Expanding the Buckets

If the mask is 0x0000000F, that means there are 16 buckets. That isn't very usefull if we have a large amount of server nodes. Therefore, the system has the capability to expand the number of buckets by adjusting the mask.

This will occur when a server has 4 buckets (likely to be two primary and two secondary), and a new node joins the cluster. The existing server will want to send some of its buckets to the new server, but going below 4 buckets will trigger a mask increase. In this example, it will shift the mask by 8-bits. This means going from 0x0000000F to 0x000000FF. Each bucket that it has, will now become 16 new buckets. (4 original buckets now become 64 buckets).

As an example:

The server had the following buckets.

0002, 0006, 000a, 000b.

If we look at 000b with a mask of 000F, when we change the mask to 00FF, we split the bucket into 16 new buckets. And they would be, 000b, 001b, 002b, 003b, 004b, 005b, 006b, 007b, 008b, 009b, 00ab, 00bb, 00cb, 00db, 00eb, 00fb. Notice that the last digit of each of those buckets is still 'b'. This means that all these buckets still match the original hashmask.

This will be done for all the buckets on the server.

Pro Tip: Because of this design, when the server splits a bucket, any transient transactions by clients are still valid because the new hashmasks are still compatible with the old hashmask.

When a server in the cluster splits its buckets (by shifting the mask), it sends out a hashmask message to each connected node and client informing them of the new hashmasks. When a server in the cluster receives a mask that is bigger than its own, it will also shift to that new hashmask. This means that when any server in the cluster splits their buckets, then rather quickly all the servers in the cluster will split their buckets. These new hashmasks would end up being blasted to each node and client on the cluster, and it is important that everything remains functioning while this occurs. Hence the importance of ensuring that everything remains compatible even if a client doesn't yet know of the change.

The server does not maintain a mix of buckets with different masks. This means that if the server is a backup for a bucket on another node that is of a different mask, it still works, but the bucket is split into smaller buckets on that node (although temporarily).

How does a client handle the splitting of buckets?

The client will need to maintain a list of hashmasks which indicate which server is responsible for each bucket. It will need to handle a special case when the buckets are split into smaller buckets. When a server needs to split a bucket, it splits ALL the buckets that it is responsible for.

For example, On the client side, assume a server has previously sent a hashmask of 0002/000F. But then the server splits the bucket, and now you get 0002/00FF, 0012/00FF, 0022/00FF ... 00E2/00FF, 00F2/00FF. When one server increases the mask, the client list will also need to expand ALL the hashes from all the server to match that higher mask.

This is fairly easy to do, but a little daunting.

Assuming the mask was previously 000F, and is now 00FF. This is easy to detect, so before parsing the new hashmasks from that server.

- Your existing hash array is 16 entries long (mask + 1 == 000F + 1).
- Create a new array that is 256 entries long (mask << 8 == 00FF).
- For each entry in the new array, binary AND that key with the OLD mask.
  For example 0052 AND 000F gives 0002.
- Lookup the old array for that entry
  In this example, 0002. That gives us the server that is responsible.
- When finished filling out the new array, remove the old array.
- Continue parsing the new hashmasks from the server which should now fit in the array.

The process is the same when the mask goes from 00FF to 0FFF.

Communications Protocol

The communications protocol follows a Command/Response format. Meaning that every command issued, should expect one and only one reply. (Although a command may trigger some reciprocal commands).

Forseeing the Future

The binary protocol is designed with performance in mind, but is still required to be flexible so that future changes to the protocol do not result in clients and servers of different versions being unable to communicate with each other.

Therefore some struct rules regarding future changes is required.

1. When a protocol command or reply is published, changes will not be made to those operations.
2. If new functionality is required, new commands (and possibly replies) are required.
3. Do not make changes (or even expand) existing commands and replies.
4. Do not add new replies to an existing command.
5. If you want different data in a reply from an existing command, then you need to create a new command. When a client submits a command, it expects the certain replies to be in a particular format, and should not recieve something different.

The ground rules

Communications are broken up into two steps. A 'command' which could otherwise be called a request, and a 'Reply'. A command should always result in a single reply. If a command should result in a number of logical peices (such as retrieving a full map), then those peices would be wrapped in a container within the reply.

All integers are to be transmitted using network-byte-order.

All strings are transmitted with a 4-byte int which indicates the length of the string, followed by the string data itself. Strings are not null-terminated.

The Header

All communications begin with an 12-byte header.

Short Int	2 bytes	Command (or Reply) Identifier
Short Int	2 bytes	Command being replied to (0 if not a reply)
Integer	4 bytes	User Specified Message ID.
Integer	4 bytes	Payload Length

All commands require a reply. A reply takes the same format as a command, and has the same 12-byte header. A command can generate different replies as needed. A single 'command' will always return a single 'reply', although it may trigger further commands to come. The reply will include a param that indicates the commandID that it is replying to.

The commands are:

Note that there are significantly more commands in the communications protocol than listed here. Only the commands and replies that are used by clients are included.

10	hello	Initiate communications.
11	capabilities	determine if a command is accepted or not, without issuing the command.
15	shuttingdown	tells the other node that this node is shutting down and not to send any buckets to it, or other data.
20	Goodbye	terminate communications cleanly
30	ping	check that the node is responding.
100	serverlist	sends a list of servers in the cluster.
110	hashmasks	list of all the hashmasks that are in the system.
120	maskupdate	when a set of hashes is being moved to a different server, let the clients and other servers know.
130	newserver	when a new server joins the cluster, a message is sent out to the other members of the cluster.
1000	lock	obtain a cluster lock
1010	unlock	release a cluster lock
2000	set_int	set an integer (32-bit) key/value, overwriting if it exists.
2010	set_long	set an long (64-bit) key/value, overwriting it it exists.
2020	set_string	set a variable length string key/value.
2100	get_int	get a integer (32-bit) key/value pair.
2110	get_long	get a long (32-bit) key/value pair.
2120	get_string	get a variable length string key/value.
2200	get_type	get the type of the data.

Note: To promote performance on the server, CommandID's less than 1000 do not require access to the clustered data. CommandID's over 1000 internally require pipeline locks and need to be done in sequence. The sub-1000 commands do not, and can be handled without going through the more expensive processing pipeline.

DATA PRIMITIVES

Different kinds of data can be stored in the cluster. Several primitives are handled directly.

In the event that these primitives are not enough, anything can be stored as a binary blob, which can be any kind of serialized object. If you serialize using a portable object notation (such as POF, JSON, etc), you can put whatever complex objects that you want in there.

Hint: These data types are also used for parameter passing in network operations.

Type	Bytes	Details
Short Int	2	A 16-bit integer, only used in command parameters. Not a storable type.
Integer	4	A regular (32-bit) integer, signed.
Long	8	A long (64-bit) integer, signed.
String	4 + data	A variable length string. Integer is the length, followed by the data.

MESSAGE STRUCTURES.

Note that some messages are commands, and some are replies.

ack (1)

General acknowlegement. Used when no data is returned.

Type of message:

  Reply

Conditions of use:

  Sent from either client or server.

Payload:

  none

fail (2)

General failure reply. Used when no data is returned.

Type of message:

Reply

Conditions of use:

Sent from either client or server.

Payload:

none

failinfo (3)

General failure reply, to indicate that something failed, but this time return an error code and a descriptive string.

Type of message:

Reply

Conditions of use:

Sent from either client or server.

Payload:

integer - error code.

string - descriptive string of the failure.

tryelsewhere (4)

This reply is returned when the command was not successful because the server is not responsible for the data requested. This can be because the server is leaving the cluster, or because the data has moved around and is now on a different node.

It contains the information of a different server to try.

Type of message:

Reply

Conditions of use:

Sent from server to client.

Payload:

string - server to use

unknown (9)

This reply is returned when the command was not known on the server. If the client is using a newer protocol that has newer commands, and it sends one to a server that doesn't understand it, the server will send this reply. The client library will need to be smart enough to handle these capabilities correctly.

This is only to be used as a reply. There should be no such thing as a 'reply' that is not known.

Type of message:

Reply

Conditions of use:

Sent from server to client.

Sent from server to server.

Sent from client to server.

Payload:

short int - command that was attempted.

hello (10)

This command is used by clients when they connect to a node. It does not contain any additional information, so it remains merely a header. Calling a 'hello' will also result in 'serverlist' and 'hashmasks' information to follow as seperate commands to the client.

Type of message:

Command

Conditions of use:

Sent from client to server.

Payload:

none

Replies:

ack(1), on success.

tryelsewhere(4), if the server is shutting down, or not accepting client connections.

capabilities (11)

This command is used by clients and servers to ensure that the other end knows certain commands. By providing the commandID, the server will respond with an 'ack', or an 'unknown' depending on if it knows it or not. This can be used to avoid using certain commands the server doesn't know how to handle.

Type of message:

Command

Conditions of use:

Sent from client to server.

Sent from server to client.

Sent from server to server.

Payload:

short int - command

Replies:

ack(1), if the command is accepted.

fail(2), if the command is not known.

shutting down (15)

This command is used by server to tell other nodes that it is shutting down. It does not contain any additional information, so it remains merely a header. When an ack(1) is received.

When a node receives this command from another node, it will mark that node as shutting down and will not attempt to move buckets to it, will remove that node as a backup to any other buckets. It will also mark the node to not try and connect to it for a while.

Type of message:

Command

Conditions of use:

Sent from server to server.

Payload:

none

Replies:

ack(1), there is no failure condition that will mean anything.

goodbye (20)

This command is used by servers or clients to indicate that the connection is about to be dropped.

It does not contain any additional information, so it remains merely a header. When an ack(1) is received, the connection can be closed. If this is sent from a client, it will cause all registered events for this client to be cleared, buffers to be reduced, and so on. If the client sends a goodbye and then changes its mind, its capabilities will be diminished, and the server may drop the connection anyway.

If this is sent from a server to a client, it will attempt to wait for an ack(1) from the client, but if the client takes too long to respond, it will drop the connection anyway. If this is sent from a server to a server, then certain synchronisations should already be in place, but any that are outstanding will need to be resolved.

Type of message:

Command

Conditions of use:

Sent from client to server.

Sent from server to client.

Sent from server to server.

Payload:

none

Replies:

ack(1), there is no failure condition that will mean anything.

ping(30)

This command is used to ensure that the server node is responding. The client can send it periodically if it wants, and the servers will send it to each other from time to time.

Type of message:

Command

Conditions of use:

Sent from client to server.

Sent from server to server.

Payload:

none

Replies:

ack(1), there is no failure condition that will mean anything.

serverlist(100)

Sends a list of servers in the cluster.

Type of message:

Command

Conditions of use:

Sent from server to server.

Sent from server to cient.

Payload:

integer - count of the servers in the cluster

string (x count) - connect info (normally in the form "IP:port", but can be any resolvable standard format)

Replies:

ack(1)

tryelsewhere(4), if the server is shutting down, or not accepting client connections.

hashmasks(110)

Sends a list of the hashmasks that this server is responsible for, including backup buckets. The mask also acts as a count and you should expect two integers for each mask. If the mask is 256 (0xFF), then that means that there are 256 hashmask entries.

Type of message:

Command

Conditions of use:

Sent from server to server.

Sent from server to client.

Payload:

integer - mask

integer - buckets in the following list.

row * mask:

integer - hashmasks

integer - instance count (0 = primary, 1 or more = backup).

Replies:

ack(1)

tryelsewhere(4), if the server is shutting down, or not accepting client connections.

hashmask(120)

Sends an updated hashmask.

Type of message:

Command

Conditions of use:

Sent from server to server.

Sent from server to client.

Payload:

integer - mask

integer - hash:

integer - instance count (-1 = not hosted, 0 = primary, 1 or more = backup).

Replies:

ack(1)

tryelsewhere(4), if the server is shutting down, or not accepting client connections.

set_int(2000)

Set an integer (32-bit) key/value, overwriting if it exists.

Payload:

integer - map hash (0 for non-map items)

integer - key hash

integer - expires (in seconds from now, 0 means it never expires)

integer - full wait (0 indicates dont wait for sync to backup servers, 1 indicates to wait).

string - name

integer - value (to be stored).

Replies:

ack(1)

tryelsewhere(4), if the server is shutting down, or not accepting client connections.

2010 set_long set an long (64-bit) key/value, overwriting it it exists.

2020 set_string set a variable length string key/value.

---

get_int(2100)

get a integer (32-bit) key/value pair.

Payload:

integer - map hash (0 for non-map items)

integer - key hash

Replies:

data_int(2105), return the integer data.

nack(), if that data doesnt exist or has been expired.

tryelsewhere(4), if the bucket is being served by a different server.

---

data_int(2105)

The integer result from a key lookup

Reply.

Payload:

integer - map hash (0 for non-map items)

integer - key hash

integer - value

2110 get_long get a long (32-bit) key/value pair.

2120 get_string get a variable length string key/value.

2200 get_type get the type of the data.
