Fly Binary Interface

Description of the Fly

Lightweight Binary Interface

Rev 2.0

Table of Contents

Release History

11-9-08	1.0	Nige	Initial release
5-11-08	1.1	Nige	Signal Handling – Autostart script – Socket reuse
16-4-09	1.2	Nige	Multicast responder – Threading improvements
11-9-09	1.3	Nige	Notify with returned Objects
15-8-11	2.0	Nige	UUID based changes – IdEntry etc.
9-1-12	2.0	Nige	Synced with Beta Server version

Fly Lightweight Binary Interface

1 Purpose of Document

This document describes in detail the Lightweight Binary Interface (LBI) that is used to communicate with the Fly Server. This document is complete with respect to laying out the format of all of the messages that can be sent to Fly and the format of all of the expected responses.

In order to achieve language independence and maximum scalability within the Fly Server we describe this binary interface from the 'ground up' so that any party can write any client in any language of their choosing.

2 Builtin and Basic Types and Conventions

The Fly LBI uses byte layouts, endian formatted for the 'builtin' types in Java. This means that the layout of a Java 'int' is the same as that in the Fly Server I.e. types exist within Scala, Java and other languages that do not require mapping to these conventional types and follow a well understood naming convention.

ZnByte = 8 bit word
ZnInt = 4 byte word
ZnLong = 8 byte word

We also define the following basic limits in the value of bytes

Ascii = any valid ascii character (or the first 7 bits of UTF-8)
ByteVal = any valid byte value (0x0-0xFF)

The Fly core understands two basic types – these are String and Object. In most object oriented languages String is a type of Object and in some cases in Fly this is also a valid, in that Strings may be encoded as Objects, however Fly Strings cannot be instanced as NULL but can be the empty string (""). Also Strings have a maximum length of ZnInt.MAX_VAL (32 bit max positive int) whereas objects have a maximum length of ZnLong.MAX_VAL (64 bit max postitive int). More formally …

String = [stringSize : ZnInt] { [Ascii] }

Which reads as a ZnInt which represent size of following ascii character string and then that number of ascii characters.

Object = [objectSize : ZnLong] { [ByteVal] }

Which reads as a ZnLong which represents the size of the following object and then that number of bytes.

As of LBI version 2 we define a UUID as follows

UUID = [mostSigBits : ZnLong | leastSigBits : ZnLong]

UUID must have the properties of at least random (type 4) UUID according to the IETF spec. (http://www.ietf.org/rfc/rfc4122.txt)

3 Message Formats

In order to communicate with the Fly Core it is necessary to format messages that are sent over an open socket connection to the server. Communication is always on a bi directional socket attached to port number 4396 (ref IANA reserved ports doc)

Information is carried in the form of entries – entries are essentially a variable number of 'Objects' (see above) encoded in a stream. 0 length (or null) objects are legitimate within the stream.

In order to support object identity from version 2 onwards we describe two forms of entries, IdEntries that contain a UUID and plain Entries that are used for template matching rather than storage purpose that don't need to include the UUID. Hence …

Entry

Entry = | number of fields : ZnLong |] { [Object] }

The valid of values for 'number of fields' are …

0 = A 'null' entry – i.e. no entry was returned (e.g. Template did not match any objects)

1 to (Integer.MAX_VALUE-1 0x7FFFFFFE) == the number of objects that follow that make up the fields of the entries.

0x7FFFFFFF = An empty entry i.e. an entry with no fields but that none the less exists in the space.

IdEntry

IdEntry = | number of fields : ZnLong | UUID | { [Object] }

The UUID is never optional for non null entires, and is encoded as the most significant 64 bits and then the least sig 64 bits. Hence to following examples are valid, among many …

IdEntry = | 0x0 |

IdEntry = | 0x1 | 0x0123456789abcdef 0x0123456789abcdef | ByteSerilaizedObject |

IdEntry = | 0x7FFFFFFF | 0x0123456789abcdef 0x0123456789abcdef |

Message Headers and Op Codes

Every valid message to the server must start with a ZnInt which is divided into two (short) byte blocks. The two most significant bytes holds the message header 0xFAB1. The two least significant bytes carry the message operation code (Opcode). For example a message header for a read operation would be 0xFAB10000 ^ 0x00000001 = 0xFAB10001

Every reply starts with a ZnLong by convention – this makes writing the client stubs more simple when dealing with asynchronous notify events.

Message : Ping
OpCode = 0
Format = | Header |
Reply = | tagCount : ZnLong | { [tag : String] }
Note : The tag "FlySpace" will always be returned even if the server is started without any tags on the command line. E.g. a server started with "fly test market data" will respond with "FlySpace test market data"

Message : Read – *LBI2*
OpCode = 1
Format = | Header | type channel : ZnInt | template : Entry | waitTime : ZnLong |
Reply = | IdEntry |
Note : WaitTime may be ignored by the server if the wait functionality is being done in the client through a polling strategy. The example implementation ignores the parameter and returns once the read is complete because the waitTime behaviour is implemented in the client stubs and not in the server.

Message : Take – *LBI2*
OpCode = 2
Format = | Header | type channel : ZnInt | template : Entry | waitTime : ZnLong |
Reply = | IdEntry |
Note : WaitTime may be ignored by the server if the wait functionality is being done in the client through a polling strategy. The example implementation ignores the parameter and returns once the take is complete because the waitTime behaviour is implemented in the client stubs and not in the server.

Message : Write – *LBI2*
OpCode = 3
Format = | Header | type channel : ZnInt | entry : IdEntry | leaseRequested : ZnLong |
Reply = | leaseGranted : ZnLong |

Message : Notify : DEPRECATED
Do not use (see Notify section below)
OpCode = 4

From LBI version 2 this form of Notify is deprecated.

Any implementing server should emit a deprecated warning and not set up the notify, returning a notify token of 0.

Reply = | notifyToken : ZnLong |

Message : Snapshot
OpCode = 5
Note : Reserved in namespace, in case there is ever a need to optimise snapshots in the Fly core. Ignored or emits unimplemented warning.

Message : ReadMany – *LBI2*
OpCode = 6
Format = | Header | type channel : ZnInt | template : Entry | limit : ZnLong | ignore : ZnLong |
Reply = | entryCount : ZnLong | { [IdEntry] }
Note : Entry count may be 0 but may not be null.

Message : TakeMany – *LBI2*
OpCode = 7
Format = | Header | type channel : ZnInt | template : Entry | limit : ZnLong |
Reply = | entryCount : ZnLong | { [IdEntry] }
Note : Entry count may be 0 but may not be null.

Message : WriteMany
OpCode = 8

Note : Reserved in 'opspace', in case there is ever a need to optimise writeMany in the Fly server. Ignored or emits unimplemented warning. Use of type channels means that preambles must be done for each object written in the client so each must be done as a single client side write. Some local client optimisations are possible in the client if we take a collection hence this appears in the client interface definitions (section 5), and for future versions of the server.

Message : Statistics
OpCode = 9
Format = | Header | typeName : String |
Notes :
If typeName is null (0 length) then dumps all stats.
If typeName is valid typeName then all stats for that channel
If typeName is an invalid typeName you get 0 stats count.

Reply = | statsCount : ZnLong | { [Stats] }
Stats = | typeName : String |
typeChannel : ZnInt |
entryCount : ZnLong |
totalReads : ZnLong |
matchedReads : ZnLong |
totalTakes : ZnLong |
matchedTakes : ZnLong |
writes : ZnLong |
writeNotifyTemplates: ZnLong |
takeNotifyTemplates : ZnLong

Notes :

For spaces that store a large number of types this operation can be intrusive on the performance of the Fly Server although the cost may be very low on a multiprocessor core/OS because it doesn't assert any locks. Type names can be used, for example, to subscribe to only the object types that are of interest to the client, performance tools should poll this method only as often as necessary. Also type format inspection tools can be written that use the TYPE_STRUCUTRE_PREAMBLE replies from a FlyPrime call. See below.

Also note section 6 for example on Stats output.

4 Type Structure Preamble

Before any Fly operations can be performed on the server it is necessary to apply to the server for a type channel token (typeChannel in the above definitions) The typeChannel is the key to the entry store that is used by this and possibly many other clients concurrently to perform the space operation.

The only way to gain a valid type channel is to apply to the core with an EntryLayout and call the classStructurePreamble operation. We define the EntryLayout to use a type name which is an agreed Ascii (UTF-8 encoded) String value. For programming language interoperation the clients may need to provide maps between the class or type name given in the client and the type name

EntryLayout = | typeName : String |
typeChannel : ZnInt |
fieldInfoCount : ZnInt | { [FieldInfo] }
FieldInfo = | fieldType : String | fieldName : String |

Notes ; In calls to the type structure preamble the typeChannel field is ignored, it can be set to any value, good practice is to set the channel to 0x0.

Message : TypeStructurePreamble
OpCode = 10
Format = | Header | EntryLayout |
Reply = | replyCode : ZnInt | EntryLayout |

In the reply the type channel is set to the type channel for this object type. If the Code is 0 then the entryLayout given to the fly server matches it's view of the entry layout and it is fine to proceed to use other methods.

If the reply code is not zero then the fly server's view of the entry layout is different to the client's view of the entry layout and the client should either report the mismatched entry layout to the application code, or should attempt to build a structure bridge between the clients view of the structure of the entry and the servers view of the structure of the entry and then reapply to the server via another call to the TypeStructurePreamble method continually, until an agreement on the structure the entry type is reached (i.e. 0 is returned)

5 Notifies

Since version 1.3 the notify system has been replaced with the following. It is implied that if any notify call is used, the client will set up a mechanism (normally a thread) to listen for async replies on the connected socket for notify replies of this format. Replies will only return on the socket which placed the notify request. If this socket is closed (stub is terminated) the notify will not be sent or received.

Message : NotifyWrite
OpCode = 20
Format = | Header | type channel : ZnInt | template : Entry | notifyLease : ZnLong |

Async Reply = | NOTIFY_HEADER = -1L : ZnLong | notifyToken : ZnLong |

Message : NotifyTake
OpCode = 21
Format = | Header | type channel : ZnInt | template : Entry | notifyLease : ZnLong |

Async Reply = | NOTIFY_HEADER = -1L : ZnLong | notifyToken : ZnLong |

Message : NotifyWriteReturningObject – *LBI2*
OpCode = 22
Format = | Header | type channel : ZnInt | template : Entry | notifyLease : ZnLong |

Async Reply = | NOTIFY_HEADER = -2L : ZnLong | notifyToken : ZnLong | entry : IdEntry |

Message : NotifyTakeReturningObject – *LBI2*
OpCode = 23
Format = | Header | type channel : ZnInt | template : Entry | notifyLease : ZnLong |

Async Reply = | NOTIFY_HEADER = -2L : ZnLong | notifyToken : ZnLong | entry : IdEntry |

6 Fly Client Interfaces

Although not strictly part of the definition of the Fly Server interface, these interfaces act as a guide-post to the extent of the services that must be provided by clients in any language to qualify as a Fly client implementation. Minimally clients must implement the 'FlyPrime' interface fully without class structure change support (although this may been seen as essential in some languages)

Basic

```
interface FlyPrime  {
      long  [granted lease] write(Serializable entry, long leaseTimeInMillis);
      Serializable [read entry]  read(Serializable entry, long waitTime);
      Serializable [taken entry] take(Serializable entry, long waitTime);
      Serializable snapshot(Serializable template);
}
```

Notifies

```
interface NotiFly extends FlyPrime {

      boolean [setUp OK]  notifyWrite(Object template, Notifiable handler, long
leaseTime);


boolean [setUp OK]  notifyTake(Object template, Notifiable handler, long
leaseTime);

}
```

Multi Ops

```
interface MutliFly extends FlyPrime {

      long [granted lease] writeMany(Collection entries, long lease);
      Collection [matching entries] readMany(Serializable template, long
matchLimit);
      Collection [matching entries] readMany(Serializable template, long
ignoreInitialMatches, long matchLimit);
      Collection [matching entries] takeMany(Serializable template, long
matchLimit);

}
```

All Together

```
interface Fly extends NotiFly, MultiFly { }
```

7 Multicast Responder

From version 1.2 of Fly, a multicast responder thread has been added so that applications can find Fly instances on a local network without any prior knowledge of the location or number of fly server instance that are currently running.

The responder follows the pattern of the Ping operation on the server to a greater or lesser extent, and likewise any code in any language that can issue and listen for multicast packets can provoke any instance of fly running on the local network to reply with its command line tags.

Fly listens on port 4396 within the multicast group "232.43.96.232". We send the string FAB1 for reasons of symmetry and for debugging purposes but any packet will provoke a response from the server.

When it receives a packet, as above, the responder sends back a packet on port 4397 within the same multicast group "232.43.96.232" that contains the list of tags returned by the Ping operation. The tag “FlySpace” will always be returned even if the server is started without any tags on the command line. E.g. a server started with “./fly test market data” will respond with “FlySpace test market data” in the response packet, formatted as ASCII 8 bit text in a ‘c’ style.

On a multiprocessor system with enough cores, a call to the responder will be run without any impact on the performance of the core of the server. The responder holds no locks and runs on an independent thread (lightweight process). However care should be taken when building lookup caches, for example, to not flood the network with multicast packets.

8 Fly Stats Output Format

In previous version of this document we described a binary output format of the Stats example app for the Fly stats interface. However since the popularity of Splunk, Flume, etc this example has been change to a simple log format that is ‘Splunk friendly’. That is it is easy for Collectors/Forwarders to scoop up and send to whatever analysis tools are being used.

Source code from the Stats app are given in the distribution.