FLO

Hierarchical Distributed Dataflow

Network Protocol

April 2025

Contents

This document specifies the network protocol used in Flo for communications with remote devices. A device is essentially a publish-subscribe **server** that enables remote networked processes, or **clients**, to access the inputs and outputs of a group of boxes. In brief, the clients send **request** messages to a device, which asynchronously responds by sending one or more **reply** messages back to the client. A client may send requests to (and receive replies from) many devices, and a device may receive requests from (and send replies to) many clients.

This document is a compact specification of the client/device communication protocol, and is structured as follows:

Section 1. **Data-Types & Byte Encodings**: provides formal definitions of all data-types.

Section 2. **Operational Semantics**: defines the a device in response to client requests.

Section 3. **Datagram Packets**: describes how Flo uses UDP to send and receive messages

1. Data-Types & Byte Encodings

This section specifies the composition and byte encoding of a message.

We assume the following basic types:

- **`Bool`** = boolean values, with instances **`true`** and **`false`**.
- **`UInt8`** = unsigned 8-bit integer.
- **`UInt32`** = unsigned 32-bit integer.
- **`UInt64`** = unsigned 64-bit integer.
- **`Float`** = 32-bit floating point numbers.
- **`String`** = character strings.
- **`[T]`** = arrays of values of some type **`T`**.
- **`[K:V]`** = dictionaries with keys/values of types **`K`/`V`**.
- **`[K:V]°`** = an ordered dictionary (i.e. a dictionary that maintains its keys/value pairs in the strict order that they were created or added).
- **`Struct`** = a data container with named/typed members (like a C struct):
    - we will denote a Struct type as in the following example:

```
Foo{
    x : Bool
    y : Float
}
```

    - and we will denote a Struct instance as in this example:

```
Foo{
    x = true
    y = 12.34
}
```

For present purposes:

- all arrays have a maximum size of 2^{32} (with **`UInt32`** indices)
- all dictionaries (ordered or not) have a maximum size of 2^{8}, and thus have with a maximum of 256 entries.

To specify byte encodings, we introduce the following notation:

- The type **`Data`** denotes an array of bytes: **`[UInt8]`**
- **`x:X`** denotes a variable **`x`** of type **`X`**.
- **`™(x)`** or **`™(x:X)`** denotes the 'byte encoding' of an instance **`x`** (of type **`X`**) to an array of bytes.
- **`x + y`** is the concatenation of 2 byte arrays (**`x:Data`** and **`y:Data`**).
- **`x - y`** is array **`x`** with the element **`y`** removed.
- **`x ∈ X`** is **`true`** if either **`x`** is an instance of type **`X`,** or an element of array **`X`**.
- **`0xH`** denotes an hexadecimal number, where **`H`** stands for a sequence of hexadecimal digits **`h ∈ {0..9,A..F}`**, e.g. as used to denote an octet (`0x`**`hh`**) or 32-bit word (`0x`**`hhhhhhhh`**).
- **`b ? x : y`** is a ternary conditional: if **`b`**, then **`x`**, else **`y`**.
- **`a[i]`** variously denotes either i) the **`i`**th element of array **`a`**, ii) the **`i`**th value of an ordered dictionary **`a`**, or iii) the value for key **`i`** in dictionary **`a`**.
- **a[°i]** denotes the **ith** value of an ordered dictionary **a.**
- **`a.keys`** is the (un)ordered list of keys of an (un)ordered dictionary **`a`.**

The following subsections formally define the hierarchical composition of messages.

1.1. Message

The form of a FLO message is captured by a **`Message`** struct type, with members:

- **`token`** : identifies the specific conversation (explained in Section 3.2 below) to which the message belongs.
- **`reply`** : is **`true`** if the message is a reply, **`false`** if it is a request.
- **`payload`** (defined in the next section) contains the content of the message.

We represent the **`Message`** struct type formally as:

```
Message{
     token : UInt64
     payload : Payload
     reply : Bool
}
```

A **`Message`** struct is encoded as:

```
™(m:Message) = ™(m.token) + ™(m.reply) + ™(m.payload)
```

1.2. Payload

The **`Payload`** type represents the contents of a message, and is defined as a union of 6 distinct struct types, the semantics of which are explained in Section 3.2:

```
Payload = union{
     HANDSHAKE{}
     PING{}
     PING_UPDATE{ name:String, skins:[Skin] }
     SUBSCRIBE{ skin:String, out:String, event:Event }
     END_SUBSCRIBE{}
     PUBLISH{ skin:String, events:[String:Event] }
}
```

A **`Payload`** struct is encoded as:

```
™(p:Payload)
    = p ∈ HANDSHAKE ?     [0x00]
    : p ∈ PING ?          [0x01]
    : p ∈ PING_UDPATE ?   [0x02] + ™(p.name) + ™(p.skins)
    : p ∈ SUBSCRIBE ?     [0x03] + ™(p.skin) + ™(p.out) + ™(p.event)
    : p ∈ END_SUBSCRIBE ? [0x04]
    : p ∈ PUBLISH ?       [0x05] + ™(p.skin) + ™(p.events)
```

The **`Box`** and **`Event`** types are described below.

1.3. Skin

A **`Skin`** struct describes a box, giving its name, and listing its input & output types:

```
Skin{
    name : String
    ins : [String:TYPE]°  // ordered dictionary
    outs : [String:TYPE]° // "
}
```

The following struct, for example, describes a box called **`"not"`** with a boolean input named **`"a"`** and a boolean output named **`"¬a"`**:

```
Skin{
    name : "not"
    ins : [ "a" : BOOL ]°
    outs : [ "¬a" : BOOL ]°
}
```

The use of ordered dictionaries (**[K:V]**°) here ensures that inputs and outputs can be uniquely identified BOTH by their name (the dictionary key), AND by their ordinal index within the dictionary (e.g. the 3rd input, 4th output etc.).

A **`Skin`** struct is encoded as:

```
™(s:Skin) = ™(s.name) + ™(s.ins) + ™(s.outs)
```

1.4. Event

An **`Event`** is a container for a value, which may be **`nil`** (indicated by the **`?`**), sent over an arc:

```
Event{ value : Value? }
```

An **`Event`** is encoded as:

```
™(e:Event) = (e.value != nil) ? ™(true) + ™(e.value) : ™(false)
```

1.5. Value

The **`Value`** type is a union of 6 data-types:

```
Value = union{
    Bool,
    Data, // a byte array = [UInt8]
    Float,
    String,
    [V], // where V ∈ { Bool, Float, String, Struct }
    Struct
}
```

A **`Value`** instance is an instance of one of the 6 **`Value`** data-types - e.g.

- **`true`** is a Value instance of type **`Bool`**
- **`[0x00,0x01,0x3B]`** is a Value instance of type **`Data`**
- **`123.456`** is a Value instance of type **`Float`**
- **`"hello world"`** is a Value instance of type **`String`**
- **`["hello","world"]`** is a Value instance of type **`[String]`**
- **`Struct{ uuid="XY", values=["x"=true,"y"=123.456] }`** is a **`Value`** instance of type **`Struct`** (where **`uuid`** uniquely identifies the **`Struct`** type)

Value byte encodings follow their primitive type encodings:

```
™(v:Value) = v ∈ Bool ? ™(v:Bool)
           : v ∈ Data ? ™(v:[UInt8])
           : v ∈ Float ? ™(v:Float)
           : v ∈ String ? ™(v:String)
           : v ∈ [V] ? ™(v:[V]) // V∈{ Bool, Float, String, Struct }
           : v ∈ Struct ? ™(v:Struct)
```

1.6. Types

In order to refer explicitly, within in a message, to the type of a **`Value`** instance, we use the 'meta-type' **`TYPE`** - which is a union of 6 distinct struct types, one for each type of **`Value`**:

```
TYPE = union{
    BOOL{ default:Bool?=nil }
    DATA{ }
    FLOAT{ default:Float?=nil }
    STRING{ default:String?=nil }
    ARRAY{ type:T } // where T ∈ { BOOL, FLOAT, STRING, STRUCT }
    STRUCT{ uuid:String, types:[String:TYPE] }
}
```

For example:

- **`true`** is an instance of **`Bool`**, which is a subtype of **`Value`**,
- but **`Bool`** *per se* is referenced in messages as **`BOOL`**, a subtype of **`TYPE`**.

The **`BOOL`**, **`FLOAT`** and **`STRING`** types all have optional "default" parameters, e.g.

- **`BOOL{ default=true }`** defines a **`BOOL`** type with a default value of **`true`**.

These default values are irrelevant for type equivalence. All **`BOOL`** types are equivalent, regardless of their defaults. Likewise for **`FLOAT`** and **`STRING`** types.

The parameters for **`ARRAY`** and **`STRUCT`** types, in contrast, ARE significant:

- The **`type`** parameter in **`ARRAY{type:TYPE}`** specifies a required type for array elements, e.g. **`ARRAY{ type=BOOL }`** defines an array of **`BOOL`**.
  - Two array types **`ARRAY{ type:T1 }`** and **`ARRAY{ type:T2 }`** are equivalent if and only if **`T1`** and **`T2`** are equivalent.
- The **`STRUCT{uuid:String, types:[String:TYPE]}`** type defines a Struct type:
  - **`STRUCT`** types are uniquely identified by their **uuid**.
  - We assume that implementations have access to a common registry of **`STRUCT`** types - which could be implemented as just a simple dictionary keyed by **`STRUCT`** uuids:

```
struct_registry = [ String : STRUCT ]
```

Types are encoded as:

```
™(t:TYPE) = t ∊ BOOL ? [0x01] + (t.default == nil) ? ™(false) :
                                                    ™(true) + ™(t.default)
          : t ∊ DATA ? [0x02]
          : t ∊ FLOAT ? [0x03] + (t.default == nil) ? ™(false) :
                                                    ™(true) + ™(t.default)
          : t ∊ STRING ? [0x04] + (t.default == nil) ? ™(false) :
                                                    ™(true) + ™(t.default)
          : t ∊ ARRAY ? [0x05] + ™(t.type)
          : /* t ∊ STRUCT ? */ [0x06] + ™(t.uuid) + ™(t.types)
```

1.7. Structs

A **Struct** is a data container with named/typed members (like a C struct):

```
Struct{
    uuid : String // uniquely identifies the STRUCT type
    values : [ String:Value ]
}
```

- Struct values must conform to the types listed in the relevant **STRUCT** type, i.e. if **Struct x** is an instance of **STRUCT** type **X** (such that **x.uuid == X.uuid**), then:
  - for all **(key,type)** in **X.types**, **x.values[key] ∊ t**

Struct instances are encoded as:

```
™(s:Struct) = ™(s.uuid) + ™(s.values)
```

1.8. Collections

The array and dictionary collection types are encoded as follows:

- An array **a:[T]** of any element type **T**, with **n:UInt32** number of elements, is encoded as follows (this also covers the array type **Data = [UInt8]**):

  ```
  ™(a:[T]) = ™(n) + ™(a[0]) + ™(a[1]) + … + ™(a[n-1])
  ```

- Both unordered dictionaries **d:[K:V]** and ordered dictionaries **d:[K:V]°** are encoded in the same way: a dictionary with **n:UInt8** number of entries, with (un)ordered key/value pairs $[k_1:v_1,k_2:v_2,\ldots,k_n:v_n]$ (with key type **K**, and value type **V**), is encoded as:

  $$™(d:[K:V]) = [n] + ™(k_1) + ™(v_1) + ™(k_2) + ™(v_2) \ldots + ™(k_n) + ™(v_n)$$

1.9. Primitives

Finally, primitive types, **Bool**, **UInt8**, **UInt32**, **Float** and **String** are encoded as:

- $™(b:Bool) = [\ b\ ?\ n\ :\ 0_B\ ]$, where **n > 0x00** is any non-zero octet.
- **™(u:UInt8) = [u]**
- $™(u:UInt32)=[[(u\&FF000000_H)>>24,(u\&FF0000_H)>>16,(u\&FF00_H)>>8,u\&FF_H]]$
- **™(f:Float)** = follows IEEE_754 floating point encoding standards
- **™(s:String) = ™(n) + utf8(s)**
  - where **utf8(s)** is the UTF-8 standard encoding of the string, and **n:UInt32** is the number of bytes in this UTF-8 encoding

This concludes the definitions of datatypes and their byte encodings.

The next section defines the operational semantics of the requests received by devices.

2. Operational Semantics

This section specifies how devices must behave in response to client requests. For present purposes, a device can be modelled as a struct comprising the device **`name`**, together with a list of the **`skins`** (black-box processes) they publish, their **`clients`** and active subscriptions (which we abbreviated as **`subs`**). The meaning of these elements is explained in the sub-sections below:

```
Device{
    name : String
    skins : [Skin]
    clients : [ String:Bool ]
    subs : [UInt32]
}
```

We will represent a **`client`** as just a string representation of a network address:

```
Client{ address : String }
```

- We will then say that a client **`c:Client`** is "known" to a device **`d:Device`** if:

```
d.clients[ c.address ] != nil
```

In the following sections we use a pre/post-condition distinction for describing actions, where:

`'PRE'` = pre-condition: a necessary prerequisite to performing an action.

`'POST'` = post-condition: defines the action.

2.1. General

The device's **`clients`** dictionary holds a list of known **`client`** addresses together with a boolean value indicating whether a **`client`** needs to be notified of the device's **`name`** and **`skins`** (using the **`PING_UPDATE`** messages, described in Section 2.3 below). This is **`true`** by default for every **`client`** when it is first added to the list (see **`HANDSHAKE`** messages in Section 2.2 below), but also whenever the device's **`name`** and/or **`skins`** change, hence:

`PRE:` for device **`d:Device:`** either **`d.name`** or the contents of **`d.skins`** changes

`POST:` i) **`d`** sets all values in its **`clients`** dictionary to **`true`** as follows:

```
for all c in d.clients.keys{
    d.clients[ c ] = true
}
```

This setting triggers the device to send its **`name`** and **`skins`** details to all **`clients`** in response to their next ping (Section 2.3).

ii) **`d`** cancels all previous subscriptions:

```
d.subs = []
```

2.2. Handshake

A client sends a handshake request to a device as a way to 'introduce itself', and allow the device to check its authenticity and authorisation.

`PRE:` device **`d:Device`** receives a handshake request, **`q`**, from an authentic and authorised client **`c:Client`**, with:

- **`q.payload ∈ HANDSHAKE{}`**

`POST:` i) **`d`** updates its client dictionary as follows:

```
d.clients[ c.address ] = true
```

This setting triggers the device to send its **`name`** and **`skins`** details to the **`client`** on the next **`client`** ping (Section 2.3 below).

ii) **`d`** immediately sends the following handshake reply to **`c`**:

```
Message{
   token = q.token
   payload = HANDSHAKE{}
   reply = true
}
```

This reply informs the client that it has been accepted as valid by the device. Receipt of the reply by the client marks a successful handshake.

2.3. Ping

Following a successful handshake (Section 2.2), the client periodically (e.g. every second) 'pings' the device to check its availability.

`PRE:` **`d:Device`** receives a request, **`q`**, from a known client **`c:Client`**, with:

- **`q.payload ∈ PING{}`**

`POST:` IF **`d.clients[c.address] == true`**, THEN:

i) **`d`** updates its client dictionary as follows:

```
d.clients[ c.address ] = false
```

This setting prevents the device from sending its **`name`** and **`skins`** details to the client on the next client ping.

ii) **`d`** immediately sends the following reply to **`c`**:

```
Message{
   token = q.token
   payload = PING_UPDATE{ name = d.name, skins = d.skins }
   reply = true
}
```

This reply informs the client of the device **`name`** and the **`skins`** it hosts. If a client does not receive a **`PING_UPDATE`** reply (for any reason), it should stop pinging the device, and try sending another **`HANDSHAKE`** request (`Section 2.2`).

ELSE: **`d`** immediately sends the following reply to **`c:`**

```
Message{
   token = q.token
   payload = PING{}
   reply = true
}
```

This reply just informs the client that the device is available.

2.4. Subscribe

A **`client`** can ask a device to send notifications of changes to the values of any **`output`** of any **`skin`**. This is done by sending a subscribe request to the device:

`PRE:` **`d:Device`** receives a request, **`q,`** from a known client **`c:Client`**, with:

- **`q.payload ∈ SUBSCRIBE{ skin:String, out:String, event:Event }`**

The request payload identifies a specific output (**`q.out`**) of a specific skin (**`skin`**) for which the client wishes to receive notifications. For a successful subscription, is required that:

```
q.event = Event{ value = nil }

d.skins[q.skin]?.outs[q.out] != nil
```

`POST:` i) **`d`** adds the message token to its subscriptions list:

```
d.subs = d.subs + [ q.token ]
```

This setting marks the subscription as "active".

ii) **`d`** immediately sends the following reply to **`c`**:

```
Message{
   token = q.token
   payload = SUBSCRIBE{
        skin = q.skin,
        out = q.out,
        event:Event{ value = d.skins[q.skin]?.outs[q.out] }
   }
   reply = true
}
```

iii) repeat from step (ii) whenever the value of **`d.skins[q.skin]?.outs[°q.out]`** changes AND for as long as **`q.token ∈ d.subs`** (see next section).

2.5. End Subscribe

A **`client`** can ask a device to stop sending subscribed notifications:

`PRE:` **`d:Device`** receives a request, **`q`**, from <u>known</u> client **`c:Client`**, with:

- **`q.payload ∈ END_SUBSCRIBE`**

AND the request corresponds to an active subscription: **`q.token ∈ d.subs.`**

`POST:` i) **`d`** immediately sends the following reply to **`c`**:

```
Message{
   token = q.token
   payload = END_SUBSCRIBE
   reply = true
}
```

This reply informs the **`client`** that the subscription has ended.

ii) **`d`** removes the message token from its subscriptions list:

```
d.subs = d.subs - q.token
```

This marks the subscription as not active.

2.6. Publish

A **client** can send values to the **inputs** of any of a device's **skins**. This is done by sending a publish request to the device:

PRE: **d:Device** receives a request, **q**, from <u>known</u> client **c:Client**, with:

- **q.payload ∈ PUBLISH{ skin:String, events:[String:Event] }**

The request identifies specific inputs (**q.events.keys**) of a specific skin (**q.skin**) to which the client wishes to publish events. For a given key, **k**, in **q.events.keys**, the publish request will only succeed if:

```
d.skins[q.skin]?.ins[k] != nil
```

POST: **d** SHOULD update all specified black-box inputs:

```
let B = the black-box process described by d.skins[q.skin]
for all k in q.events.keys{
  d SHOULD set the value of input k of B to q.ins[i].value
}
```

This ends the operational semantics of the protocol for communicating with remote devices.

The next section provides a brief note on the default Flo implementation of the protocol using UDP (user datagram packets) network communications.

3. Datagram Packets

By default, Flo uses UDP[1] (user datagram protocol) for sending and receiving messages.

It is common in UDP implementations to set a maximum buffer size, typically just a few kB, on individual packets. The size of the messages exchanged using the device protocol depends primarily on the size of their payloads, which may vary from small **Bool** values (a single byte), to very large byte arrays (e.g. encoding images or video streams). In order to handle payloads whose size exceeds the maximum buffer size for datagram packets, we introduce a protocol for decomposing messages into packets.

The basic approach is as follows:

- for a message **m**, with token **m.token** and byte encoding **™(m):**
  - sequentially split the encoding into **n** small bytes arrays, a_i (each smaller than the maximum buffer size), such that:

    $$\text{™}(m) = a_1 + a_2 + \ldots + a_n$$

  - encode each of these small arrays as follows:

    $$\text{™}(a_i) = \text{™}(m.token) + \text{™}(j) + a_i$$

    - where if **i == n** then **j = (i + flag)**, else **j = i**, and **flag** is a single (most significant) bit to indicate the maximum index
  - and then send the arrays as individual packets in any order:
    - assuming all the packets arrive, the prefixed token and index (i.e. **™(m.token) + ™(j)**) carries all the information necessary to reconstitute the original complete message.

[1] https://en.wikipedia.org/wiki/User_Datagram_Protocol

3.1. Reference Implementation

Below is a Swift reference implementation for the Datagram packet algorithm described above.

First we define a couple of constants:

```
let MSG_HEADER_SIZE = 4 + 2 // size of token + packet index
let PKT_N_FLAG = 0x8000 // flag identifying the maximum index
```

The constraint on the maximum size of a packet is then defined as:

```
let MAX_BUF_SIZE = 4096 // the actual buffer size is not important
let MAX_PACKET_SIZE = MAX_BUF_SIZE - MSG_HEADER_SIZE
```

The following code then decomposes a message into packets:

```
fstatic var decomp_counter:UInt32 = 0
func decompose(_ bytes:inout [UInt8],_ f:(inout [UInt8])->()){
   // unique identifier for this group of packets
   Manager.decomp_counter += 1
   // n = number of packets to send:
   let n = Int(ceil(Float32(bytes.count)/
                            Float32(Packet.MAX_PKT_LEN)))
        // encode the packet group id
        let C = Manager.decomp_counter.bytes
        // break the msg up into 'n' packets
        for i in 0..<n{
            // start index of packet
            let j = i * Packet.MAX_PKT_LEN
            // end index of packet
            let k = min(bytes.count,(i+1) * Packet.MAX_PKT_LEN)
            // encode the index as a countdown ..
            var idx = UInt16(n-i)
            // .. so the 1st index = the total number of packets
            if i == 0{
                idx |= Packet.PKT_N_FLAG // flag the first index
            }
            // encode the packet
            var pk = C + idx.bytes + Array<(UInt8)>(bytes[j..<k])
            f(&pk) // each packet is passed to the function f
        }
    }
```

To reconstitute the message, you need to keep a record of all the packets received that have the same initial token - e.g. in a dictionary with the following form:

```
                    pkt_group_id    pkt_index  pkt_data
    packets = [       UInt32    : [   UInt16 : [UInt8] ]()
```

Then, for a given message token, **t**:

- if a packet index, `i`, matches the `PKT_N_FLAG` (i.e. `i & PKT_N_FLAG > 0`), then the index is the same as the total number, `n`, of packets into which the original message was decomposed.
- when all `n` packets have been received, the byte encoding of the original message can be rebuilt by concatenating all the packet data, in reverse index order (see Swift code above):

  $™(m) = a_n + a_{n-1} + \ldots + a_2 + a_1$

  where: a_i = `packets[t][i]`