

NOODLES V0.3: A PROTOCOL SPECIFICATION

NICHOLAS BRUNHART-LUPO

CONTENTS

1. Introduction	2
2. Rationale & Design Goals	2
3. Architecture	2
3.1. Communication	3
4. Concepts	3
4.1. Document	3
4.2. Identifiers	5
4.3. Objects	5
4.4. Tables	5
4.5. Plots	5
4.6. Signals and Methods	5
4.7. Buffers	5
4.8. Mesh	5
4.9. Materials	6
4.10. Textures	6
4.11. Lights	6
5. Common Message Elements	6
5.1. Any Type	6
6. Server Messages	7
6.1. Root Message	7
6.2. Objects	8
6.3. Tables	8
6.4. Buffers	8
6.5. Geometry	9
6.6. Texture	9
6.7. Material	9
6.8. Lights	10
6.9. Signals & Methods	10
6.10. Signal Invoke & Method Reply	10
6.11. Document	10
7. Client Messages	11
7.1. Root Message	11

7.2. Introduction	11
7.3. Method Invocation	11
7.4. Asset Refresh	11
8. Semantics	12
8.1. Tables	12
8.2. Plots	14
8.3. Objects	15
8.4. Scene Semantics	17
9. Operation & Lifecycle	17
9.1. Websocket Messages	17
9.2. Connection	17
Appendix A. Common Flatbuffer Specification	17
Appendix B. Server Message Flatbuffer Specification	36
Appendix C. Client Message Flatbuffer Specification	36

1. INTRODUCTION

This document entails a specification for a distributed scene-graph wireline protocol suitable as a substrate for shared interactive visualizations. It also lays out concepts for the supporting implementations that would provide such visualizations.

2. RATIONALE & DESIGN GOALS

- The intent of this document is simplicity, to get a working version implemented so that further improvements can be identified.
- The structure here is not intended to mirror the use-case of the HTML DOM + Javascript where code is shipped to clients. That would be restrictive, as it requires the clients either interpret or compile and run code on command. Some clients, such as integrated head mounted systems, do not allow compilation, or are not sufficient computing platforms.
- Trying to mirror just the HTML DOM part has issues as well; a number of 3D declarative implementations (like QML 3D), all operate on a scenegraph under the hood. It seems more fruitful to just target the scenegraph for modification, and perhaps (as part of the server library) have a declarative component there.
- A shared document is desired here, as opposed to the standard browser case where every client has their own copy of state.
- Code listings are provided as an example and for exposition only. Clients and servers may be written in any language as long as they conform to the proper wireline protocol.

3. ARCHITECTURE

The system envisions the use of four components, two of which fall under this specification.

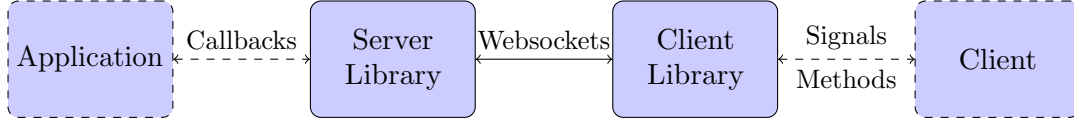


FIGURE 1. System architecture. Note that there may be more than one client. Elements with solid lines fall under this specification.

The Server Library presents a visualization to one or more connected clients through a synchronized scenegraph. Client requests and messages are passed on for handling to the application code, which can manipulate the scenegraph in response. These changes are then published and sent to clients.

The Client Library connects to a server, and maintains the synchronized scenegraph. This scenegraph is query-able by the client. Clients then can interpret and present the scenegraph to the user in the way they see fit. For example, an immersive graphics engine client can draw the scenegraph as is, while a 2D client can choose to present only a subset of the graph. A command line (i.e. Python) client may ignore the scenegraph completely to merely make use of the messaging and method invocation functionality. This also allows each client to customize the interactions available in a way that best aligns with their form factor.

3.1. Communication. Communication between the libraries is achieved over WebSocket connections. All messages are sent over the binary channel of the WebSocket using Flatbuffers.

Client-to-client notification is not supported, and must first pass through the server.

The bulk of communication is from server to client.

This spec is intended to be implemented in a secure network, with the presumption that those that connect to the server are trusted. Provision for security will come later, as is the case with everything, because security is hard and makes my brain bleed.

3.1.1. Flatbuffers. For performance reasons, the *in-situ* capabilities of the serialization medium down-selected available options to Flatbuffers and Cap'n'proto. Both were explored. Table 1 compares the two in rough terms. In the end Flatbuffers won out due to more language support out of the box.

4. CONCEPTS

The objective of the system is to synchronize, as best as possible, the document between the client and the server. This is accomplished through the use of discrete messages.

4.1. Document. The Document represents the visualization. It is an entity-component model, with an Object as the core entity, and Tables being a secondary entity.

The document is implicit. The other elements are explicit.

	Pro	Con
Cap'n'Proto	Created by Protobuf developers, strong pedigree. Excellent JSON inter-operation.	More complex internal formats. Fewer languages supported out of the box. Some packages for other languages are of lower quality. Default serialization code has performance issues. ¹
Flatbuffers	More languages supported out of the box. Simple internal format, more format features (such as maps, field deprecation, evolution). Easier to obtain performance increases using existing serialization code.	API for some languages is horrible. Some languages require schema to have specific design, adding indirection.

TABLE 1. Serialization Format Comparison

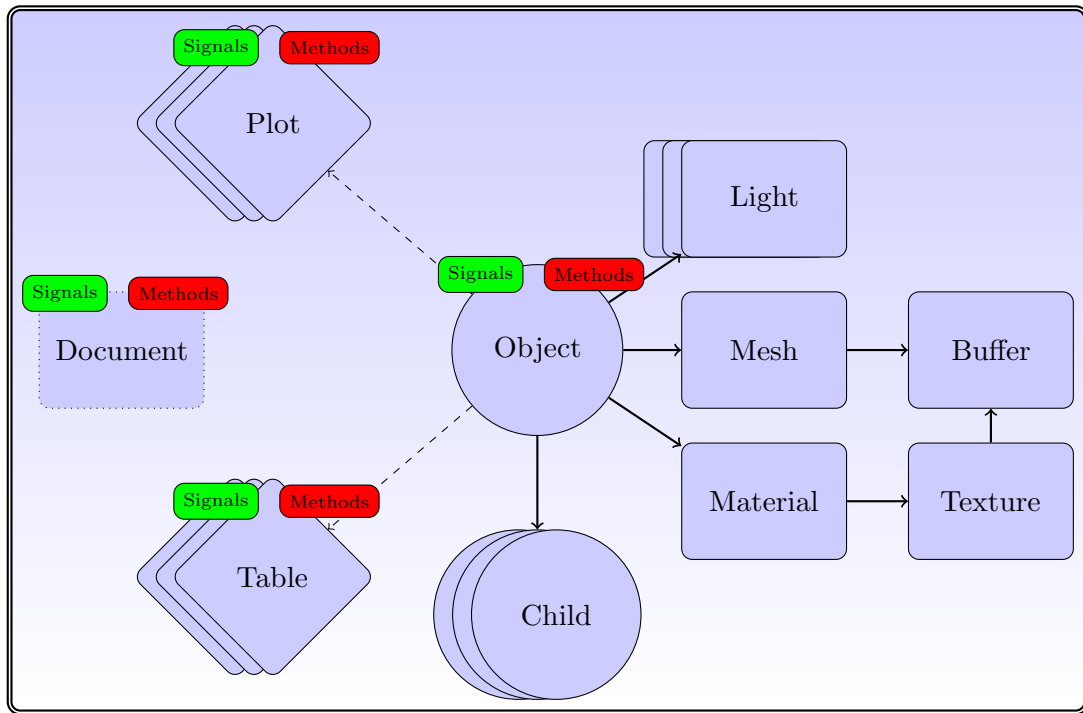


FIGURE 2. Document structure.

4.2. Identifiers. Identifiers are a pair of 32-bit unsigned integers; the first being a slot number, and the second being a generation count. This allows non-hashed storage, as there should be no two elements with the same slot number, so it can be used as an index in an array. The generation number is used to help identify if a slot has been recycled by the server, and thus allow detection of stale identifier use.

An identifier where either the slot and generation are the maximum unsigned integer value is the ‘null’ ID.

4.3. Objects. Each object is provided with an Object ID. Objects are rendered in a hierarchy, starting from a root object with the ID 0. Objects can have any number of children.

Each object is a possibly render-able object, and has a transformation, an optional name, a parent Object ID, a mesh (what to draw), a material (how to draw it), a number of lights, and links to tables. Objects also have a set of string tags, and attached methods and signals. Objects can also be instance rendered.

Objects are mutable.

4.4. Tables. Tables are a structured way to transmit row oriented data. They consist of a header (list of column names), and rows. Attached signals and methods are used to allow clients to modify the data in the table or fetch records (but only when first subscribed to).

4.5. Plots. Plots are a way to transmit and possibly synchronize 2D plots. They consist of either a simple textual plot definition (described below), or a URL to load in a browser.

4.6. Signals and Methods. Signals are notifications from the server to the client. They may contain data, and may come from the document, objects, or tables.

Methods are requests to the server from the client. They may take a set of data parameters, and they may return data as well. They must have a contextual object that they are called on, otherwise they are called on the Document. During the course of a method invocation, signals from the server could be generated.

Each method invocation is tracked by a client-generated arbitrary string. These shall be unique and never re-used. For servers, every method must generate a reply message; the only exception is if the client did not provide an invocation identifier string.

There is a possibility that a method could be called on an object, that is then subsequently deleted, or replaced. In this case, a reply is still generated, and not squashed by the server. Thus a client should be able to handle replies on objects that no longer exist.

Methods and signals are immutable.

4.7. Buffers. A buffer is an opaque block of bytes. This allows for efficient storage and transfer of large assets. These assets can be sent either inline through the WebSocket, or can be supplied through a URL that the client can fetch the buffer from.

Buffers are immutable and referenced from meshes and textures.

4.8. Mesh. Meshes define the geometry that is to be rendered. They consist of references to a buffer for number of components (see Table 2).

Meshes are mutable.

Component	Type	Value Type	Count
Position	Vertex	float	3
Normals	Vertex	float	3
TexCoords	Vertex	unsigned short	2
Colors	Vertex	unsigned byte	4
Lines	Index	unsigned short	2
Triangles	Index	unsigned short	3

TABLE 2. Mesh components

4.9. Materials. This should be a PBR based material, featuring basic elements: base color, metallic, roughness, including an optional texture for base colors. The material only applies to the node it is attached to. Note that though the material is specified in PBR, the client may use Phong or other interpretations of the specified material in order to meet performance goals. The material may also specify that blending should be used; the blending function is src_α and $1 - src_\alpha$.

Materials are mutable.

4.10. Textures. Textures reference images (in Buffers) to be used by a material. Textures are mutable.

4.11. Lights. Lights describe illumination sources. They are mutable.

5. COMMON MESSAGE ELEMENTS

This section discusses common elements to both the server and client message portions of the specification.

5.1. Any Type. The any type is the foundation value type. it is composed as follows:

LISTING 1. Any Definition

```

1  table MaterialID {
2      id_slot : uint32 = 4294967295;
3      id_gen  : uint32 = 4294967295;
4  }
5
6  table GeometryID {
7      id_slot : uint32 = 4294967295;
8      id_gen  : uint32 = 4294967295;
9  }
10
11 table LightID {
12     id_slot : uint32 = 4294967295;
13     id_gen  : uint32 = 4294967295;
14 }
15
16 table ImageID {

```

```

17     id_slot : uint32 = 4294967295;
18     id_gen  : uint32 = 4294967295;
19 }
20
21 table TextureID {
22     id_slot : uint32 = 4294967295;
23     id_gen  : uint32 = 4294967295;
24 }
25
26 table SamplerID {
27     id_slot : uint32 = 4294967295;
28     id_gen  : uint32 = 4294967295;
29 }
30
31 table BufferID {
32     id_slot : uint32 = 4294967295;
33     id_gen  : uint32 = 4294967295;
34 }
35
36 table BufferViewID {
37     id_slot : uint32 = 4294967295;
38     id_gen  : uint32 = 4294967295;
39 }
40
41 union AnyIDType {

```

It is a discriminated union of text, integers, real, or bytes. It also contains a generic list, and string-keyed map. For efficiency, there is also a real-list and integer-list type which allow contiguous storage or access of those elements.

In method and signal API terms, real-lists and integer-lists are coerce-able. That means, for example, that a list of reals may be provided by the `RealList` type, or as a `AnyList` of `Real`.

6. SERVER MESSAGES

Here we discuss the messages that are sent from the server to the client.

Almost all components have strict lifetimes defined by creation and deletion messages. Some messages are also used to update an existing component. Therefore, if a create-update message is received by the client for a component/entity of an ID that it has never seen before, that is the creation milestone. Otherwise, it is an update message. Update messages are treated with certain semantics: either an atomic or non-atomic updates. In non-atomic update, keys in the message add or replace keys in the destination. In an atomic update, the destination is completely replaced by the message.

6.1. Root Message. The server sends messages by the root type `ServerMessages`. This is an array of a union of creation/deletion messages.

LISTING 2. Server Root Messages

6.2. Objects. Objects are created, updated and destroyed in Listing 3. All coordinates are provided in the OpenGL right hand manner.

LISTING 3. Object Messages

Each element is optional, with the exception of the object id.

If the object ID has not been seen before by the client, it is assumed to be a new object. If there is no transform in the message, it is assumed to be the identity.

If the object ID has been seen before and not deleted, it should update the existing object with the elements that are provided in the message. For example, a message for Object ID 5 that contains a transform will only update object 5's transform, and not change other elements. In another example, to detach a material from an object, an update message with a null material ID is used.

Instances of the underlying meshes are specified by a list of matrices, with a matrix per instance. Using column major ordering, Matrix 1, shows how position p , rotation r (as a quaternion), color c , and scale s are specified. Access is assumed to be by column, i.e. $M_0 = p$. Transforms should be applied in the following order: scaling, rotation, translation.

$$(1) \quad \begin{pmatrix} p_x & c_r & r_x & s_x \\ p_y & c_g & r_y & s_y \\ p_z & c_b & r_z & s_z \\ 0 & 0 & r_w & 0 \end{pmatrix}$$

Objects have a definition as to how they should be represented. By default, objects use the **EmptyDefinition**. Objects can also use a **TextDefinition**, which describes that the object should be rendered as a text annotation. Text is to be rendered how the client sees fit, with the orientation to be centered at the object, the text perpendicular to $+Z$ and up being $+Y$. The height of the text must be specified; the text will then automatically use the font information to compute the text width. If the optional width is specified, then the text shall, keeping the proper font aspect ratio, try to fill the bounds provided. For more general 2D content, **WebpageDefinition** can be used to declare that the object is a web page. **RenderableDefinition** is used to declare that the object is supposed to render 3D geometry, with optional instances.

6.3. Tables. Tables are created and destroyed in the following messages.

LISTING 4. Table Messages

Tables have names (which shall be unique), a metadata JSON string, methods, and signals.

6.4. Buffers. Buffers are created and destroyed in the following messages.

LISTING 5. Buffer Messages

Buffers are either inline (in the `bytes` field) or provided as a URL. If a URL is supplied the size of the buffer must be passed as well. If neither is supplied, the server has the data inline, but has deemed it too large to send immediately to avoid stalling clients. In this case, the server would do well to supply the data through another port and use the URL feature, but some servers are unable to do this. In this case, where both the bytes and URL feature are empty, the `url_size` field must still be filled for client pre-allocation. At intervals, the client can send a refresh message to fill in the missing buffers to avoid burdening the websocket (see Section 7.4).

6.5. Geometry. Geometries are created, and destroyed in the following messages.

LISTING 6. Geometry Messages

Limitations in Flatbuffer IDL require some notes here. Geometries are defined by ranges of a buffer for their components. These ranges, in the `ComponentRef` type, require a buffer, and also *require* a start byte offset of the buffer, as well as a byte size field, and the byte stride between vertex elements.

The `min_extent` and `max_extent` fields are required so that clients can efficiently determine culling boundaries.

The position field is required. The other vertex components (`normals`, `texCoords`, `colors`) are optional, but recommended. If there is no normal, the mesh should be rendered without lighting to avoid graphical artifacts. If there are no texture coordinates, the coordinate (0,0) should be assumed for each vertex. If there are no colors, the color (1,1,1,1) should be assumed for each vertex.

Index elements are specified in `lines` and `triangles`. Only one of these should be active. These specify the indices for line segments and triangles respectively. The stride for these components must be zero, i.e. they must be tightly packed.

6.6. Texture. Textures are created, updated, and destroyed in the following messages.

LISTING 7. Texture Messages

Textures, specify a buffer range for an image. For portability, these are to be in the on-disk format for PNG, JPG, or EXR. KTX is also allowed, but the user should be aware that not all clients can support it. Textures shall be interpreted as in the OpenGL coordinate system

6.7. Material. Materials are created, updated, and destroyed in the following messages.

LISTING 8. Material Messages

The `color` key defines colors in the range of 0 – 1 for r, g, b, a . Other PBR parameters are also in the 0 – 1 range. The texture ID field is optional, and could also be null to indicate no texture.

Materials can be updated and are not immutable.

6.8. Lights. Lights are created, updated, and destroyed in the following messages.

LISTING 9. Light Messages

The `color` key defines colors in the range of 0 – 1 for r, g, b . Intensity of the light is unbounded. Note that, for the moment, changing the light type after the initial creation message is not allowed.

6.9. Signals & Methods. Signals and Methods are created, and destroyed in the following messages.

LISTING 10. Signals Messages

Methods must be provided with a human friendly name. Two methods may not share the same name; there is no overloading. Documentation is recommended, but not required, as is return value documentation. Argument information must be provided, at the very least given a name. You may not call a method with more arguments than as specified; use an argument that takes an array type to permit this option.

Signal must be provided with a human friendly name, and also may not share the same name. Arguments follow the same requirements as methods.

6.10. Signal Invoke & Method Reply. Signals may be invoked on the client and client methods replied with the following messages.

LISTING 11. Communication

Either only `on_object` or `on_table` or `on_plot` or none must be set, to indicate context. Signals may NOT be invoked on a context that does not have them attached.

Method replies must have a previously given method invocation identifier (see Section 7.3). If the method could not be executed, an exception field is filled instead of data.

In an exception, the code should represent either one of the predefined error codes in Table 3, or a code in the defined user-code region. A short message should be provided for users; additional data may also be provided for things like nested errors. Given the differences in clients, however, it is likely that such data would be flattened to a string.

Reserved error codes are provided in Table 3 and are designed to match the XMLRPC and JSONRPC codes. Error codes –32768 to –32000 are reserved by the spec.

6.11. Document. The document is implicit, and always exists. It can be modified with the following messages.

LISTING 12. Document Messages

The document may be updated with `DocumentUpdate`, to modify the current methods and signals. It may also be completely reset. The reset message, by quirk of Flatbuffers, may not be empty; ignore any fields within. When a document is reset, all components and objects are to be deleted at that point.

Code	Message	Description
-32700	Parse Error	Given invocation object is malformed and failed to be validated
-32600	Invalid Request	Given invocation object does not fulfill required semantics
-32601	Method Not Found	Given invocation object tries to call a method that does not exist
-32602	Invalid Parameters	Given invocation tries to call a method with invalid parameters
-32603	Internal Error	The invocation fulfills all requirements, but an internal error prevents the server from executing it

TABLE 3. Error Codes

7. CLIENT MESSAGES

In this section we discuss the messages sent by a client.

7.1. Root Message. Client messages are defined as the following root type.

LISTING 13. Client Message Root

7.2. Introduction. The client introduces itself to the server with the following message.

LISTING 14. Introduction Message

The name of the client must not be empty, and should identify a client; host names can be used. The version integer represents the major version of the specification that the client supports. Version 0 implies that the client is a pre-version-1 client.

7.3. Method Invocation. The client asks to invoke a method with the following message.

LISTING 15. Method Invocation

The message must have an invocation identifier; the asynchronous reply will carry that identifier. Identifiers must not be reused.

Either the `on_object` or the `on_table` or `on_plot` or none should be set, to indicate the context of the invocation: on an object, on a table, or on a plot or on the document, respectively. The method can only be called on a context on which it is attached.

The arguments to the method must match the documented method signature.

7.4. Asset Refresh. The client may ask to receive missing buffer content with the following message.

LISTING 16. Buffer Refresh

8. SEMANTICS

8.1. Tables. Tables are a way of exposing record data to clients so that they can either provide an alternative representation of that data or to allow command line clients access to the data. An example of an alternative representation would be a 2D chart that could be provided for a lightweight 2D client instead of a 3D plot. Another approach would be to allow a visual representation to provide a link to details of a certain data point.

Tables consist of columns (with unique names) and rows. Rows are identified by a **Key**, which is an integer. Keys are assumed to be monotonically increasing, starting from 0, that is, new insertions into the database are given a new key larger than any key seen before.

Another useful abstraction is the **Row** type; a row is either an **AnyList** or a **RealList**. A **Column** is the same.

A commonly used notion is the concept of a selection within a table of data. Listing 17 shows, in a JSON-like way, the definition of a Selection object as encoded in a NOODLES Any.

LISTING 17. Selection object definition. Note that the **to** field in the row ranges is exclusive. The **row_ranges** list *must* have an even number of elements.

```

1 {
2   "rows" : [Key, ...], // must be an IntegerList
3   "row_ranges" : [
4     Key from, Key to,
5     ...
6   ] // also must be represented as an IntegerList
7 }
```

8.1.1. Methods & Signals. To query table information, signals and methods are used. These names are restricted and cannot be used by the user application. Note, indexes are all zero-based. Tables 4 and 5 list the data related methods and signals a table can support. The server should not send any data or signals to the client for a given table *unless* a client has expressed interest by calling the subscribe method. This is to avoid stressing clients that have no table interface and to reduce unnecessary network traffic. Further it is up to the server to honor these methods; should the server not support modification, for example, requests will return an exception.

Subscribe. This allows the client to receive signals from the table. Without this, no signal should be sent by the server regarding the table. When this call is made, the server will reply with a **TblInit** object. The full object definition is as follows:

```

1 TblInit : {
2   "columns" : [ string ], // 1
3   "keys" : [ Key ], // 2
4   "data" : [ Column ], // 2
5   "selections" : [ [string, SelectionObject] ] // 3
6 }
```

Method Name	Description
<code>TblInit tbl_subscribe()</code>	Subscribe to changes in the table, receiving initial table state. The client will then receive signals.
<code>void tbl_insert([Column])</code>	Request to add rows of data to the table, as a pack of column segments.
<code>void tbl_update([Key], [Column])</code>	Request to update many rows of data to the table, as a pack of column segments.
<code>void tbl_remove([Key])</code>	Ask to remove a list of keys.
<code>void tbl_clear()</code>	Ask to remove all rows of the table.
<code>void tbl_update_selection(/*snip*/) </code>	Ask to update a selection in the table.

TABLE 4. Table Methods summary

Signal Name	Description
<code>void tbl_reset()</code>	Reinitialize the table. Sent if the table is cleared or reset in some way.
<code>void tbl_updated([Key], [Column])</code>	Rows were updated in the table.
<code>void tbl_rows_removed([Key])</code>	Rows in the table were removed.
<code>void tbl_selection_updated(/*snip*/) </code>	A selection has changed.

TABLE 5. Table Signals summary

Part 1 is a list of columns. This establishes a column order that is used to interpret and pack data values later in other calls and signals. The second elements provide the current data that is in the table, as well as the keys used to identify rows. The third is a pack of the current selections that are available in the table; this is provided as a list of pairs, where the first part of the pair is the string identifier of the selection and the second is the selection object that defines the selection.

Reset. Should the server issue the `tbl_reset` signal, this would imply that the table has been reset, with no data, and no selections, but with the same header.

Insertion. Data may be inserted into the table through both the row and many versions of the call. Note the key cannot be specified. The row length should be equal to the length of the header, and supplied in header order. The many version simply takes a list of rows to be inserted. Insertion success is demonstrated through reception of the `rows_inserted` signal; this signal provides the data inserted along with the keys that were assigned to that row, i.e. the full row of data for all columns.

Update. Data can be updated through both the row and many versions. In this case, as opposed to the insertion functions, the full row, including the key column, is specified in column order, so that the correct row may be updated. Success is indicated through the corresponding update signal.

Removal. Data can be removed by specifying a list of keys to delete. Success will be indicated through the corresponding signal for all clients.

Selection. Data selections can be made through the `update_selection` call. The full signature of the call is as follows:

```
void tbl_update_selection( string, SelectionObject );
```

The first argument denotes the selection to update or add, and the selection object defines what that selection should be updated/initialized to. A selection object that is empty, i.e. specifying no rows or ranges is considered the empty selection and denotes that the selection should be deleted from clients.

This shall trigger the selection update signal. The full signature of the signal is as follows:

```
void tbl_selection_updated( string, SelectionObject );
```

This mirrors the update call, and denotes which selection has changed, and what to change it to.

8.1.2. *Tables Metadata.* Tables are also capable of synchronizing metadata for other purposes. This is exposed as a JSON object.

8.2. **Plots.** To facilitate 2D plot synchronization, multiple optional mechanisms are present. Plots expose a simple definition system, and a URL system.

8.2.1. *Simple.* In the `SimplePlot` object, there is a single member `definition`. This is a JSON encoded object, containing one of several formats.

The first format provides a simple encoded approach:

LISTING 18. Table Metadata for Plot Sync

```

1 definition : {
2     "simple_plot" : SimplePlotInfo ,
3     <other keys>
4 }
5
6 SimplePlotInfo : {
7     "plot_name" : "name",
8     "columns" : [ ColumnInfo ]
9 }
10
11 ColumnInfo : {
12     "column_name" : "<name>",
13     "prefers" : "x" | "y",
14     "color" : "#rrggbb",
15     "range" : [from, to]
16 }
```

In Listing 18, the definition JSON will contain a key called `simple_plot`. This key is a listing of named plots; each plot describes how each column of the table should be treated in an arbitrary simple plot.

Complex 2D Plot Sync: More advanced plotting facilities are forthcoming, but planned to follow a system like: <http://docs.juliaplots.org/latest/attributes/>.

Web. Another option is to directly expose a URL for web access. This allows for complex server-based or other peer to peer 2D synchronization tools.

8.3. Objects. Objects may carry the logical operations.

For simplicity, in this section, we let `vec3 = RealList` and `vec4 = RealList`. When used as arguments, the three component and four component vectors require the exact number of components to be supplied in the list. Otherwise the server will consider that to be malformed, and can reject the call.

8.3.1. *Activator*. For clients, this could be when the user clicks on an object, or presses an interaction button when a wand is over an object.

```
void activate(string)
void activate(int)
list<string> get_activation_choices()
```

It is up to the server application to decide how to handle this ‘activation’. Activation is either in the string or integer form. Activation names can be obtained through the API (example: ‘Click’, ‘Clear Options’). An activation can be triggered by the string, or by an integer index. It makes sense to thus tie the order of names to priorities; a 0 is a primary click, 1 is an alternate click action, etc.

8.3.2. *Options*. Options are is conceptually the same thing as a combo-box widget.

```
list<string> get_option_choices()
string get_current_option()
void set_current_option(string)
```

A list of choices can be presented for an object, and an option can be set.

8.3.3. *Movable*. Movable objects allows the user to request to change the position of an object.

```
void set_position(vec3 p)
void set_rotation(vec4 q)
void set_scale(vec3 s)
```

Positions, rotations and scales are in the coordinate system of the parent object. The rotation is to be provided as a quaternion, with w being the last component.

8.3.4. *Selection.* Regions of an object can be ‘selected’. What this means is up to the application.

```
void select_region(vec3, vec3, int)
void select_sphere(vec3, real, int)
void select_half_plane(vec3, vec3, int)
void select_hull([vec3], [int], int)
```

The selection API allows for a number of different selection tools. Others can be forged through the use of the movable API, and activators. All coordinates provided are in the object-local coordinate space.

For `select_region`, the selection region is supplied as an axis aligned bounding box, and an option for either additive selection (> 0), deselection (< 0) or replacement ($= 0$). For `select_sphere`, a position and a radius is supplied. For `select_half_plane`, a point and a normal is provided. For `select_hull`, the client provides a list of 3D points, and an index list interpreted as a mesh hull.

To support multiple selections, consider adding options and activators to your object.

8.3.5. *Query.* Objects can be probed to obtain a data value or annotation.

```
[string, vec3] probe_at(vec3)
```

The location (object local coordinates) to be probed is supplied in the argument. As a return value, a revised position is returned (in case the server desires to snap the probe to a different location) and a string containing the data to display.

Note that more complex actions may take place; a user can build their application to add more functionality (or use a different activator), which can instantiate objects for all users to see.

8.3.6. *Annotation and Attention.* The object may request user attention, through the following signals.

```
void signal_attention()
void signal_attention(vec3)
void signal_attention(vec3, string)
```

Multiple overloads are provided. If the signal has no data, the whole object would like attention. If there is a position, a specific object-local coordinate would like attention. If there is a string in addition to that, a message should be displayed at that point.

To attract attention, sounds, client-specific graphical adornment can all be used. For some clients, changing the camera view to include the point of attention can also be done.

8.3.7. *Object Tags.* Objects may be given tags. They are a list of strings. These allow the client to discover capabilities of the Object, or classify an object. Some tags imply the presence of certain methods or signals. Tags prefixed with `noo_` are reserved for use by the system.

Tag	Description
<code>noo_user_hidden</code>	On lists of objects or tree-views, this object should be hidden. Other objects should be visible ²

8.4. Scene Semantics.

8.4.1. *Reporting.* Clients may inform the server of areas of ‘interest’ of the given scene through reporting methods attached to the document.

```
void noo_client_view(vec3 direction, real angle)
```

Note that ‘interest’ is different for different clients. As an example, a desktop client may wish to signal interest via a mouse. An AR system may consider an eye-tracking based approach. For an Immersive VR environment, head direction might be used.

This method, if it exists, should not be called very often; as we are sampling the user, view information can be provided at a human scale, on the order of a second or more.

9. OPERATION & LIFECYCLE

9.1. **Websocket Messages.** The server side shall send the **ServerMessages** message, while clients are restricted to sending **ClientMessages** message.

9.2. **Connection.** Upon the connection of a websocket, the client first sends an introduction message. Any other message is ignored by the server until the introduction is provided.

The server will then send a list of creation messages to build the scene. This could pose a problem; large mesh or texture assets could take a significant amount of time to transfer and attempting to send those all at the start could cause issues ranging from the server being blocked, the client being overwhelmed, or de-synchronization, depending on implementations. In order to avoid this, the server may send creation info of buffers without the data. The client can use placeholder assets, and use the asset refresh mechanism to request asset updates with full information, which it can then use to update the graphical representation.

From this point onward, the client can invoke methods, and the server can send signals and other messages.

APPENDIX A. COMMON FLATBUFFER SPECIFICATION

```
1 // Generate with
2 // flatc --scoped-enums --reflect-names --gen-mutable -c noodles.fbs
3
4 namespace noodles;
5
6
7 /*
```

²This approach (hidden-specified) is chosen, because in a visible-specified, it is difficult to know when to hide the other objects.

```

8
9 Update semantics:
10 - Some objects can be updated, some cannot.
11 - Unless otherwise specified, updates are value-like atomic. That is, the
12   client should reconstruct the local representation of the entity. This is
13   opposed to a non-atomic delta-like update; where only mentioned fields in
14   the
15   table should be updated in the local representation. Note that in non-
16   atomic
17   mode, updates cannot (of course) change the ID of the object. However,
18   updates to the "name" field is to be ignored on the client, and not done
19   by
20   the server.
21
22 Coordinate system semantics:
23 - We follow the GLTF style for coordinates and orientations (right handed).
24 - Units are in SI.
25 - Lengths are in meters.
26
27 */
28 //
29
30 // =====
31
32 // Common Types
33 // =====
34
35 // =====
36
37
38 // This API is defined around handles to things.
39 // Identifiers are tables, due to poor language support for structs.
40 // If any of the slot or gen fields are maximum, then the handle is null.
41
42 table EntityID {
43     id_slot : uint32 = 4294967295;
44     id_gen  : uint32 = 4294967295;
45 }
46
47 table PlotID {
48     id_slot : uint32 = 4294967295;
49     id_gen  : uint32 = 4294967295;
50 }
51
52 table TableID {
53     id_slot : uint32 = 4294967295;
54     id_gen  : uint32 = 4294967295;
55 }
56
57 table SignalID {
58     id_slot : uint32 = 4294967295;
59 }

```

```
50     id_gen  : uint32 = 4294967295;
51 }
52
53 table MethodID {
54     id_slot : uint32 = 4294967295;
55     id_gen  : uint32 = 4294967295;
56 }
57
58 table MaterialID {
59     id_slot : uint32 = 4294967295;
60     id_gen  : uint32 = 4294967295;
61 }
62
63 table GeometryID {
64     id_slot : uint32 = 4294967295;
65     id_gen  : uint32 = 4294967295;
66 }
67
68 table LightID {
69     id_slot : uint32 = 4294967295;
70     id_gen  : uint32 = 4294967295;
71 }
72
73 table ImageID {
74     id_slot : uint32 = 4294967295;
75     id_gen  : uint32 = 4294967295;
76 }
77
78 table TextureID {
79     id_slot : uint32 = 4294967295;
80     id_gen  : uint32 = 4294967295;
81 }
82
83 table SamplerID {
84     id_slot : uint32 = 4294967295;
85     id_gen  : uint32 = 4294967295;
86 }
87
88 table BufferID {
89     id_slot : uint32 = 4294967295;
90     id_gen  : uint32 = 4294967295;
91 }
92
93 table BufferViewID {
94     id_slot : uint32 = 4294967295;
95     id_gen  : uint32 = 4294967295;
96 }
97
98 union AnyIDType {
99     EntityID,
```

```

100     TableID,
101     SignalID,
102     MethodID,
103     MaterialID,
104     GeometryID,
105     LightID,
106     ImageID,
107     TextureID,
108     SamplerID,
109     BufferID,
110     BufferViewID,
111     PlotID
112 }
113
114 union InvokeIDType {
115     EntityID,
116     TableID,
117     PlotID
118 }
119
120 table AnyID {
121     id : AnyIDType (required);
122 }
123
124 table MapEntry {
125     name  : string (key);
126     value : Any;
127 }
128
129 table Text { text : string; }
130 table Integer { integer : int64; }
131 table IntegerList { integers : [int64]; }
132 table Real { real : double; }
133 table RealList { reals : [double]; }
134 table Data { data : [byte]; }
135 table AnyList { list : [Any]; }
136 table AnyMap { entries : [MapEntry]; }
137 // due to a limitation of FB, we can't have structs in a union. Therefore
138     ...
138 table AVec2 {
139     x : float;
140     y : float;
141 }
142
143 table AVec3 {
144     x : float;
145     y : float;
146     z : float;
147 }
148

```

```
149 table AVec4 {
150     x : float;
151     y : float;
152     z : float;
153     w : float;
154 }
155
156 union AnyType {
157     Text,
158     Integer,
159     IntegerList,
160     Real,
161     RealList,
162     Data,
163     AnyList,
164     AnyMap,
165     AnyID,
166     AVec2,
167     AVec3,
168     AVec4,
169 }
170
171 table Any {
172     any : AnyType;
173 }
174
175 enum Format : byte {
176     U8,
177     U16,
178     U32,
179
180     U8VEC4,
181
182     U16VEC2,
183
184     VEC2,
185     VEC3,
186     VEC4,
187
188     MAT3,
189     MAT4,
190 }
191
192 // Misc Types
193     =====
194 struct Vec2 {
195     x : float;
196     y : float;
197 }
```

```
198
199 struct Vec3 {
200     x : float;
201     y : float;
202     z : float;
203 }
204
205 struct Vec4 {
206     x : float;
207     y : float;
208     z : float;
209     w : float;
210 }
211
212 struct Mat3 {
213     // for javascript compat, we have to expand the below:
214     // components : [float : 9];
215     c1 : Vec3;
216     c2 : Vec3;
217     c3 : Vec3;
218 }
219
220 struct Mat4 {
221     // for javascript compat, we have to expand the below:
222     // components : [float : 16];
223     c1 : Vec4;
224     c2 : Vec4;
225     c3 : Vec4;
226     c4 : Vec4;
227 }
228
229 struct BoundingBox {
230     aabb_min : Vec3;
231     aabb_max : Vec3;
232 }
233
234 struct RGB {
235     r : uint8;
236     g : uint8;
237     b : uint8;
238 }
239
240 struct RGBA {
241     r : uint8;
242     g : uint8;
243     b : uint8;
244     a : uint8;
245 }
246
```

```

247 //
=====
248 // Server Messages
=====
249 //
=====

250
251 table MethodArg {
252     // What is the name for this method argument?
253     name : string (required);
254
255     // Documentation for users; what does this argument do?
256     doc  : string;
257
258     // Optional hint of the type of this argument
259     // Any value in AnyType (as text) is valid.
260     hint : string;
261
262     // Optional Control hint for gui editors.
263     // Currently known values
264     // - 'EditCheckbox'
265     // - 'EditSlider:min:max:step'
266     editor_hint : string;
267 }
268
269 // Create a new method
270 table MethodCreate {
271     // Depending on the application, multiple methods might have the same
272     // name
273     // This can cause some confusion; avoid it by prefixes, etc.
274
275     id          : MethodID (required); // The new method's ID
276     name        : string (required);   // Non-unique name of method
277     documentation : string;             // Optional docstring for method
278     return_doc   : string;             // Optional return value
279     documentation
280     arg_doc      : [ MethodArg ];      // Arguments to method
281 }
282
283 // Destroy a method
284 table MethodDelete {
285     id : MethodID (required);
286 }
287
288 //
=====

```

```

288 // Create a new signal
289 table SignalCreate {
290     id          : SignalID (required); // The new signal's ID
291     name        : string (required);   // Non-unique name of signal
292     documentation : string;             // Optional signal docstring
293     arg_doc     : [ MethodArg ];        // Data provided with signal
294 }
295
296 // Delete a signal
297 table SignalDelete {
298     id : SignalID (required);
299 }
300
301 //
=====
302
303 // The 'has no visual representation' type
304 table EmptyDefinition {
305     // Tables cannot be empty, which breaks the variant model.
306     // In any case, this field is completely ignored.
307     padding : bool = false;
308 }
309
310 // Render this entity as text
311 table TextDefinition { // Text plane, normal -z, up is +y, center: obj
    origin
312     text      : string (required); // String to render
313     font      : string (required); // Approximate font to use (e.g. Arial)
314     height    : float = .25; // The height of the text plane.
315     width     : float = -1; // Optional width of text, infer from height if <
        0
316 }
317
318 // Render this entity as a web page. This is done by defining a plane on
    which
319 // to paint the page
320 table WebpageDefinition {
321     url      : string (required); // Where should we fetch the page?
322     height   : float = .5;         // the physical height of the page 'plane'
323     width    : float = .5;         // the physical width of the page
324 }
325
326 table InstanceSemantic {
327     view      : BufferViewID;
328     // bytes between instance matrices. For best performance, there should
        be
329     // no padding.
330     stride    : uint64 = 0;
331 }

```



```

332
333 // Render this entity as a mesh
334 table RenderableDefinition {
335     material      : MaterialID (required);
336     mesh          : [GeometryID] (required);
337     instances     : InstanceSemantic; // optional
338     instance_bb   : BoundingBox; // optional override for instanced object
339         culling
340 }
341 union Representation {
342     EmptyDefinition,
343     TextDefinition,
344     WebpageDefinition,
345     RenderableDefinition
346 }
347
348 struct EntityVisibility {
349     // Should this entity even be visible? By default all renderable items
350     // are
351     // visible, but there are times when you want to switch something off
352     // temporarily
353     visible : bool;
354 }
355
356 // Create or update an entity.
357 // Non-atomic update semantics
358 table EntityCreateUpdate {
359     // either the new id of the entity or the entity to update
360     id      : EntityID (required);
361     name    : string; // optional name of this entity
362     parent  : EntityID; // optional parent of this entity. DO NOT CREATE
363         L00PS
364     transform : Mat4; // optional transform; if missing, assume
365         identity
366
367     representation : Representation; // optional drawable representation
368
369     lights      : [LightID]; // optional lights attached
370     tables      : [TableID]; // optional tables attached
371     plots       : [PlotID]; // optional plots attached
372     tags        : [string]; // optional tags
373     methods_list : [MethodID]; // optional attached methods
374     signals_list : [SignalID]; // " " signals. avoid "signals" for Qt.
375
376     // optional region of influence, for interaction; edge case for when a
377     // user is clicking empty space, but you want this entity to catch it.
378     influence : BoundingBox;
379
380     visibility : EntityVisibility; // optional visibility

```

```

378 }
379
380 table EntityDelete {
381     id : EntityID (required);
382 }
383
384 //
=====
385
386 table SimplePlot {
387     // this plot uses a simple language. TBD
388     definition : string (required);
389 }
390
391 table URLPlot {
392     // this plot is defined as a webpage.
393     url : string (required);
394 }
395
396 union PlotType {
397     SimplePlot,
398     URLPlot
399 }
400
401 // non-atomic update semantics
402 table PlotCreateUpdate {
403     // ID of the plot to either create or update
404     id : PlotID (required);
405     name : string; // optional name of the plot
406     table_ref : TableID; // optional link to a table of data
407     type : PlotType (required); // type of this plot
408     methods_list : [MethodID]; // optional attached methods
409     signals_list : [SignalID]; // optional attached signals
410 }
411
412 table PlotDelete {
413     id : PlotID (required);
414 }
415
416 //
=====
417
418 table InlineSource {
419     bytes : [byte] (required);
420 }
421
422 table URLSource {
423     url : string (required);

```

```

424 }
425
426 union BufferSource {
427     InlineSource, // either an inline set of bytes
428     URLSource      // or a URL to load
429 }
430
431 // A buffer describes a source of bytes to read from
432 table BufferCreate {
433     // ID of the buffer to either create or update
434     id      : BufferID (required);
435     name    : string; // optional name of this buffer
436     size    : uint64; // Size of buffer, if missing or zero, invalid
437
438     // where does the data come from?
439     source  : BufferSource;
440 }
441
442 table BufferDelete {
443     id      : BufferID (required);
444 }
445
446 //
447     =====
448
449 enum ViewType : byte {
450     UNKNOWN,
451     GEOMETRY_INFO, // Data contains geometry information (vertex, index)
452     IMAGE_INFO,    // Data contains an image
453 }
454
455 // Defines a subrange of a buffer, with a hint as to the data contained
456 // within.
457 table BufferViewCreate {
458     // ID of the buffer view to either create or update
459     id      : BufferViewID (required);
460     name    : string; // optional name of this view
461     source_buffer : BufferID (required); // Buffer this view looks at
462
463     type    : ViewType; // Type hint for this buffer
464     offset  : uint64 = 0; // Offset into the buffer of this range
465     length  : uint64 = 0; // Length of this range
466 }
467
468 table BufferViewDelete {
469     id      : BufferID (required);
470 }

```

```

470 //
471 // =====
472 // A reference to a texture
473 table TextureRef {
474     texture_id      : TextureID (required);
475
476     // texture coordinate transform. If missing, identity
477     transform       : Mat3;
478
479     // texture coordinate channel of a mesh to be used in mapping
480     texture_coord_slot : uint8;
481 }
482
483 table PBRInfo {
484     base_color      : RGBA; // default is 255 for all channels.
485     base_color_texture : TextureRef; // assumed to be SRGB. no premult
486     alpha
487     metallic        : float = 1;
488     roughness       : float = 1;
489     metal_rough_texture : TextureRef; // assumed to be linear. ONLY RG used
490 }
491
492 // non-atomic update semantics
493 table MaterialCreateUpdate {
494     // ID of the new material or the material to update
495     id      : MaterialID (required);
496     name    : string;
497
498     pbr_info      : PBRInfo; // if missing, assume defaults.
499     normal_texture : TextureRef; // if missing, no normal mapping
500
501     occlusion_texture      : TextureRef; // assumed to be linear. ONLY R
502     occlusion_texture_factor : float = 1;
503
504     emissive_texture : TextureRef; // assumed to be SRGB. ignore A.
505     emissive_factor  : Vec3;
506
507     use_alpha      : bool = false;
508     alpha_cutoff   : float = .5;
509
510     double_sided : bool = false;
511 }
512
513 table MaterialDelete {
514     id      : MaterialID (required);
515 }
516

```

```

517 //
518 // =====
519 union ImageSource {
520     BufferViewID, // Either a reference to a buffer
521     URLSource // Or a URL to load from
522 }
523
524 // Images may come in a variety of formats
525 // - PNG
526 // - JPEG
527 // - KTX2
528 // PNG and JPEG should be supported, KTX2 may be ignored
529 table ImageCreate {
530     // ID of the image to create
531     id : ImageID (required);
532     name : string; // name of this image
533
534     // color space information must be ignored
535     source : ImageSource;
536 }
537
538 table ImageDelete {
539     id : ImageID (required);
540 }
541
542 //
543 // =====
544 // Textures may be assumed to be in SRGB. If so, they must be decoded to
545 // linear
546 // before use in shaders, etc.
547 table TextureCreate {
548     id : TextureID (required);
549     name : string;
550
551     image : ImageID (required);
552     sampler : SamplerID; // optional, if missing default sampler
553 }
554
555 table TextureDelete {
556     id : TextureID (required);
557 }
558 //
559 // =====

```

```

560 enum MagFilter : byte {
561     NEAREST,
562     LINEAR,
563 }
564
565 enum MinFilter : byte {
566     NEAREST,
567     LINEAR,
568     NEAREST_MIPMAP_NEAREST,
569     LINEAR_MIPMAP_NEAREST,
570     NEAREST_MIPMAP_LINEAR,
571     LINEAR_MIPMAP_LINEAR,
572 }
573
574 enum SamplerMode : byte {
575     CLAMP_TO_EDGE,
576     MIRRORED_REPEAT,
577     REPEAT,
578 }
579
580 table SamplerCreate {
581     id : SamplerID (required);
582     name : string;
583
584     mag_filter : MagFilter;
585     min_filter : MinFilter;
586
587     wrap_s : SamplerMode = REPEAT;
588     wrap_t : SamplerMode = REPEAT;
589 }
590
591 table SamplerDelete {
592     id : SamplerID (required);
593 }
594
595 //
596     =====
597
598 // Lights are defined to mirror the GLTF punctual light extension
599 // A point light source
600 table PointLight {
601     range : float = -1;
602 }
603
604 table SpotLight {
605     //Direct light along -Z
606     range : float = -1;
607     inner_cone_angle_rad : float = 0;

```

```

608     outer_cone_angle_rad : float = 0.7853981633974483; // PI/4.0
609 }
610
611 table DirectionLight {
612     //Direct light along -Z
613     range : float = -1;
614 }
615
616 union LightType {
617     PointLight,
618     SpotLight,
619     DirectionLight,
620 }
621
622 // non-atomic update semantics
623 table LightCreateUpdate {
624     id          : LightID (required);
625     name        : string;
626
627     color       : RGB; // Linear space, default pure white
628     intensity   : float = 1.0;
629
630     light_type  : LightType; // after being set once, updates ignored
631 }
632
633 table LightDelete {
634     id : LightID (required);
635 }
636
637 //
638
639 // Renderable primitive types
640 enum PrimitiveType : byte {
641     POINTS,
642     LINES,
643     LINE_LOOP,
644     LINE_STRIP,
645     TRIANGLES,
646     TRIANGLE_STRIP,
647     TRIANGLE_FAN // Not recommended, some hardware support is lacking
648 }
649
650 enum AttributeSemantic : byte {
651     POSITION, // for the moment, must be a vec3.
652     NORMAL,  // for the moment, must be a vec3.
653     TANGENT, // for the moment, must be a vec3.
654     TEXTURE, // for the moment, is either a vec2, or normalized u16vec2
655     COLOR,   // normalized u8vec4, or vec4

```

```

656 }
657
658 // Interpret a buffer view as a strided pack of vector or matrix elements
659 table Attribute {
660     view      : BufferViewID;
661     semantic  : AttributeSemantic;
662
663     // some semantics may have a channel; there could be multiple of these
664     // attributes for this mesh. For now
665     // textures and colors.
666     // may have extra channels. Implementations need not support more than
667     // 1
668     channel   : byte;
669
670     stride    : uint64 = 0; // bytes between elements
671     format    : Format;     // format of the element
672
673     minimum_value : Vec4; // optional bounds for this attribute
674     maximum_value : Vec4; // optional bounds for this attribute
675
676     // are the elements normalized?
677     // for example a normalized U8: 0 -> 0, 255 -> 1.
678     normalized : bool = false;
679 }
680
681 table IndexSemantic {
682     view      : BufferViewID;
683     // bytes between indicies. for performance, recommend all indicies be
684     // tightly packed, with no padding.
685     stride    : uint64 = 0;
686     format    : Format;     // format of the indicies, u8, u16, u32
687 }
688
689 table GeometryPatch {
690     attributes : [Attribute];
691
692     // optional, if missing, non-indexed primitives only
693     indicies   : IndexSemantic; // u8, u16, u32
694
695     type : PrimitiveType = TRIANGLES;
696
697     material : MaterialID; // Material to use for rendering this patch
698 }
699
700 table GeometryCreate {
701     id      : GeometryID (required); // id of the new geometry
702     name    : string;
703     patches : [ GeometryPatch ];
704 }

```



```

705 table GeometryDelete {
706     id : GeometryID (required);
707 }
708
709 //
=====
710
711 // non-atomic update semantics
712 table TableCreateUpdate {
713     // ID of the new table to create
714     id          : TableID (required);
715     name        : string; // name of the table
716     meta        : string; // application defined metadata
717     methods_list : [ MethodID ]; // methods attached
718     signals_list : [ SignalID ]; // signals attached
719 }
720
721 table TableDelete {
722     id : TableID (required);
723 }
724
725 //
=====
726
727 // Update the core document properties
728 // non-atomic update semantics
729 table DocumentUpdate {
730     methods_list : [ MethodID ];
731     signals_list : [ SignalID ];
732 }
733
734 // Ask to reset the document. All entities, objects, tables, etc, are now
735 // invalid.
736 table DocumentReset {
737     padding : bool; // these things cannot be empty, so...
738 }
739
740 //
=====
741
742 // Clients should invoke the given signal on the given context
743 table SignalInvoke {
744     // ID of signal to invoke
745     id : SignalID (required);
746
747     // if not set, the context is on the document
748     context : InvokeIDType;

```

```

749
750     // Arguments to the signal
751     signal_data : AnyList;
752 }
753
754 // Information about method exceptions
755 // This is modelled after JSONRPC error handling and exceptions.
756 table MethodException {
757     code      : int64; // required (but not expressable in fbs)
758     message   : string; // optional
759     data      : Any;    // optional
760 }
761
762 // A reply to a method invocation.
763 table MethodReply {
764     invoke_ident      : string (required); // the client provided invoke
765                          ident
766     method_data       : Any;                // optional, method return value
767     method_exception  : MethodException; // optional, possible exception
768 }
769 //
770
771 union ServerMessageType {
772     MethodCreate,
773     MethodDelete,
774     SignalCreate,
775     SignalDelete,
776     EntityCreateUpdate,
777     EntityDelete,
778     BufferCreate,
779     BufferDelete,
780     BufferViewCreate,
781     BufferViewDelete,
782     MaterialCreateUpdate,
783     MaterialDelete,
784     TextureCreate,
785     TextureDelete,
786     SamplerCreate,
787     SamplerDelete,
788     ImageCreate,
789     ImageDelete,
790     LightCreateUpdate,
791     LightDelete,
792     GeometryCreate,
793     GeometryDelete,
794     TableCreateUpdate,
795     TableDelete,

```

```

796     DocumentUpdate,
797     DocumentReset,
798     SignalInvoke,
799     MethodReply
800 }
801
802 table ServerMessage {
803     message : ServerMessageType;
804 }
805
806 // Root type for server messages. This is the only type that is to be sent
807 // from // the server.
808 table ServerMessages {
809     messages : [ ServerMessage ];
810 }
811 //
812 // =====
813 // Client Messages
814 // =====
815 //
816 // Introduction of the client to the server, must be the first message sent
817 // by
818 // the client, and the server will not respond until it gets such a message
819 table IntroductionMessage {
820     client_name : string (required); // A human-friendly name of the client
821 }
822
823 // Client asks to invoke a method or function, i.e. RPC
824 table MethodInvokeMessage {
825     method_id : MethodID (required);
826
827     // Context, or which thing should this method be invoked on
828     // If not set, it is on the document
829     context : InvokeIDType;
830
831     // An optional, client created identifier for this invocation request.
832     // If blank, the server will not send any response, i.e. fire and
833     // forget
834     // semantics.
835     invoke_ident : string;
836
837     // Arguments for this method.
838     method_args : AnyList;
839 }

```

```
838 union ClientMessageType {
839     IntroductionMessage,
840     MethodInvokeMessage,
841 }
842
843 table ClientMessage {
844     content : ClientMessageType (required);
845 }
846
847 // Root type for client messages, this is the type that must be sent from
848 // clients.
849 table ClientMessages {
850     messages : [ ClientMessage ] (required);
851 }
```

APPENDIX B. SERVER MESSAGE FLATBUFFER SPECIFICATION

```
1 include "noodles.fbs";
2
3 namespace noodles;
4
5 root_type ServerMessages;
```

APPENDIX C. CLIENT MESSAGE FLATBUFFER SPECIFICATION

```
1 include "noodles.fbs";
2
3 namespace noodles;
4
5 root_type ClientMessages;
```