Headset Communication Protocol

This document defines an application network protocol for exchanging messages between the server and the headset.

Transport

The client and the server shall use a TCP socket to exchange data.
The designated port number is 15300.

Data Encoding

The client and the server shall exchange messages encoded as JSON. The socket shall be treated as a UTF-8 byte stream.

Messages must be prefixed by their length, encoded as a 32-bit unsigned integer. As in, if a message is 16 bytes long, then it must be prefixed with the following bytes: “00 00 00 10”

Reminder: All network traffic is big endian.

This is needed so that the receiver knows how many bytes to read. JSON strings are variable length. We can reliably read them by first reading the 4 byte length, then reading the JSON string, and finally waiting on more data and repeating the previous two steps.

The server uses the LengthDelimitedCodec implementation provided by tokio-utils.
More detail on that can be found here:
https://docs.rs/tokio-util/latest/tokio_util/codec/length_delimited/

Paradigm

As is true for protocols designed for other video game server applications, this a session level protocol, and thus it should be implemented on top of the transport layer to conform properly to the OSI model. The headset joins the VR exhibit, starting the session, and then the session is active until the VR headset disconnects.

During the session, we present interactions as RPC between the headset and the server. The message IDs describe which procedure is being called, and then the server expects a response. The goal here is to minimize bandwidth usage as much as possible, with the end goal being reduced traffic over the wireless access point.

It is important that all headsets receive messages in order, and that all headsets receive all messages. Thus, the TCP protocol for data transport has been chosen. Building a drop-tolerant

protocol might actually be advantageous, however it would increase the complexity of the protocol specification and implementations.

The order of operation is like this:

- The server performs a procedure call upon the headsets.
- The headsets attempt to fulfill the request, and return a response.

This pattern of operation continues indefinitely until the headset closes the TCP socket.

Each procedure has its own ID. In this document, we will refer to procedures as **Messages**.

Possible Improvement

- Switch from JSON to MessagePack if needed (should be trivial): https://msgpack.org/index.html

- Switch the transport from TCP to QUIC or SCTP if needed. SCTP supports multicasting and might be better for carrying broadcast messages.

Connection Initiation

The server will send the following object when a new headset connects. It indicates the protocol version being used.

```
{
	"version": 1
}
```

In response, the headset is expected to reply with the following object:

```
{
	"id": string
}
```

The headset ID must be a string. The server will use this ID to identify the headset internally and with responses in its API.

The server will then respond with an affirmation that the ID was accepted or denied:

```
{
	"session_name": string or null,
}
```

The server will deny an incoming connection if the given ID collides with that of another headset. If the connection is denied, “accepted” will be false, and the socket will be closed by the server. Otherwise, the server will keep the connection open and expect an acknowledgement before sending further messages.

The “session_name” is a human readable name which is used to identify the headset during the session. The session name will be a string if the connection is accepted, and it will be null if it is denied.

Connection Termination

Headsets should close the TCP socket to end their session.

Messages

All messages will use the following schema:

```
{
      “message”: int,
      “kwargs”: dict
}
```

- message: Indicates the type of the message.
- kwargs: Arguments for the message. Think of the “kwargs” dict in Python.

Example:
{
“Message”: 0x1 (this would be scene change),
“Kwargs”: “scene”: <scene id>(numeric)
}
If successful you can respond with null.

Replies

The headsets should reply with this schema:

```
{
      “error”: boolean,
      “response”: dict || null,
}
```

As an acknowledgement of receiving the message. The response field should be filled in with any extra data that the message requires in return.

Errors

In the case that there was an error, the error flag should be set to “true”.
An error reply should follow the schema below:

```
{
      “error”: true,
      “response”: {
            “message”: <string>,
            “detail”: <any>,
      }
}
```

Message Types

0x0: Query

Query the headset for information.

ID: 0x0
Argument Schema: {} // The unit struct

Response:

```
{
      // We will add fields here as needed
      “query”: <headsetStatus>
}

headsetStatus :
{
      Active: boolean;
      BatteryPercent: int;
}
```

0x1: Change Scene

This message instructs the headset to change to the given scene. The scene ID is the same as what is used in the XML configuration file.

ID: 0x1
Argument Schema:

```
{
      "scene": int
}
```

Response: `null`

If the scene is not valid, an error message could be sent back to the server.

0x2: Focus on Object

This message instructs the headset to focus on an object.

ID: 0x2
Argument Schema:

```
{
      "object": string
}
```

Response: `null`

0x3: Attention Mode (Deprecated)

This message instructs the headset to set or unset attention mode.

ID: 0x3
Argument Schema:

```
{
      "attention": boolean
}
```

Response: `null`

0x4: Transparency Mode

This message instructs the headset to set or unset passthrough mode.

ID: 0x4
Argument Schema:

```
{
      “transparency”: boolean
}
```

Response: `null`

Server Control API

The app will use this to make requests to the server to perform commands, and to obtain information about the headsets.

The app will take the “reins” of the server and control it using this API.

Any endpoint could realistically return with a 500 status code. It should always be handled by any clients.

Endpoints

GET: /headsets

Takes no parameters, returns a list of headsets.

Response body schema:

```
[
	{
		“name”: str,
		“recv”: int,
		“connection_health”: str (one of “Excellent”, “Good”, “Bad”)
	}, … more of these objects …
]
```

GET: /commands

Takes no parameters.
Returns a list of command objects.

Schema (could be changed):

```
[
      {
            “type”: i32,
            “kwargs”: { … message body (varies) … }
      }
]
```

POST: /command/<type>

Tells the server to broadcast a command to headsets.