

# Interactive 2 Protocol Specification

Document Version: 1.3.0

## Contents

<b>Vocabulary</b>	<b>2</b>
<b>Server Interaction</b>	<b>3</b>
Endpoint Discovery . . . . .	4
OAuth . . . . .	4
Wire Protocol . . . . .	7
Compression . . . . .	9
none . . . . .	9
gzip . . . . .	10
lz4 . . . . .	10
Error Codes . . . . .	10
<b>Synchronization</b>	<b>11</b>
<b>Methods</b>	<b>12</b>
Core & Authentication . . . . .	12
ready . . . . .	12
setCompression . . . . .	13
onReady . . . . .	14
getTime . . . . .	14
getMemoryStats . . . . .	15
issueMemoryWarning . . . . .	16
setBandwidthThrottle . . . . .	16
getThrottleState . . . . .	17
issueMemoryWarning . . . . .	18
hello . . . . .	18
Participants & Groups . . . . .	19
getAllParticipants . . . . .	20
onParticipantJoin . . . . .	21
onParticipantLeave . . . . .	21
onParticipantUpdate . . . . .	21
getActiveParticipants . . . . .	22
updateParticipants . . . . .	23
createGroups . . . . .	24

getGroups . . . . .	25
updateGroups . . . . .	26
deleteGroup . . . . .	28
onGroupCreate . . . . .	29
onGroupDelete . . . . .	29
onGroupUpdate . . . . .	29
Scene Setup . . . . .	30
Built-In Controls . . . . .	30
Control Positioning . . . . .	31
createScenes . . . . .	32
getScenes . . . . .	33
deleteScene . . . . .	34
updateScenes . . . . .	35
onSceneCreate . . . . .	36
onSceneDelete . . . . .	36
onSceneUpdate . . . . .	37
createControls . . . . .	37
deleteControls . . . . .	39
updateControls . . . . .	39
onControlCreate . . . . .	41
onControlDelete . . . . .	41
onControlUpdate . . . . .	42
Input . . . . .	42
giveInput . . . . .	43
Spark Transactions . . . . .	44
capture . . . . .	44

## Vocabulary

In this specification, the “client” refers to the software running on an end-user’s machine or a third party trusted or untrusted server, consuming the interactive input.. The “server” refers to the software, or mediator, running on Beam’s servers. A “participant” refers to an end user on the Beam website or a Beam app giving input through the mediator. Participants are backed by static, persistent “users” on Beam; a participant is a Beam user who is connected to Interactive. “Sparks” are the virtual currency of Beam. Participants may be charged sparks for specific actions in Interactive.

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in [RFC 2119](#).

## Server Interaction

The client establishes a connection to the server over a websocket, as described below. In this section we lay out the authorization process and initial states that the client may be in. Each of these qualifications are checked in the order that they're presented; a failure at a previous check will precede all subsequent checks.

- When making a handshake to the server, the client **MUST** include an **Authorization** header containing an OAuth bearer token. Failing to pass an authorization or passing an invalid or expired expiration **SHALL** cause the server to close the websocket after a connection with a 4020 code. See **OAuth** for more information.
- Additionally, the client **MUST** include an **X-Interactive-Version** header corresponding to the number version ID that the integration runs. If the version ID is not found, or the user does not have the correct permission to play the game, the connection will be closed with a 4021 code. For example: **X-Interactive-Version: 478210**.
- The client **MAY** include an **X-Interactive-Sharecode** header if it wishes to use an interactive integration which has been shared by its author.
- Additionally, the client **MUST** include an **X-Protocol-Version** header corresponding to the interactive protocol version it speaks. If this server cannot provide the version, a 400 **Bad Request** status code will be returned. The protocol version specified by this document is **2.0**.
- If there is already an interactive connection running for the channel, the connection will be closed with a 4022 code.

When a connection is established to the server, the channel enters a “staging” mode, and after the client signals that it’s ready it enters interactive mode where clients are able to connect and controls appear below the Beam channel.. The channel remains in interactive mode until the connection terminates. Authentication context is preserved throughout the lifetime of the socket.

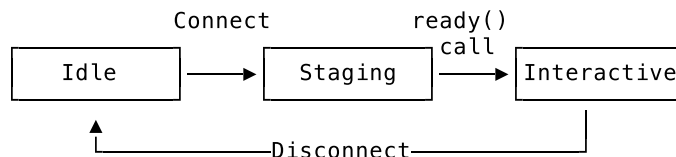


Figure 1: Channel state diagram

For clients unable to set headers when initializing a websocket handshake the client **MAY** include them as query string parameters. Like request headers, the keys **SHALL** be case-insensitive.

## Endpoint Discovery

Beam runs multiple servers in several locations, and will put servers into and remove servers from rotation over time as updates and made and demand shifts. Clients should call the endpoint <https://beam.pro/api/v1/interactive/hosts> to retrieve a list of currently available servers. This returns a list of servers ordered by several factors, including distance to the client and load. A typical response might look like this:

```
[
  {
    "address": "wss://tetris1.dal-09.beam.pro"
  },
  {
    "address": "wss://tetris2.sea-01.beam.pro"
  }
]
```

In general, clients should chose the first returned server, falling back to others if they're unable to connect or they lose connection.

## OAuth

Connections to Interactive must be authenticated using OAuth. Beam implements Bearer and Implicit grants as described in [RFC6749: The OAuth 2.0 Authorization Framework](#). You can view further details and configuration on our [developer website](#); you need to request the `interactive:robot:self` permission in order to connect to the interactive mediator.

An alternative flow is available to interactive applications to avoid the need for opening, embedding browsers, or requiring keyboard input on the client device:

0. Register an OAuth application on the [Beam Lab](#). If you're developing an integration which will run on users' computers, you should not request a client secret.
1. Call `POST /oauth/shortcode` with your `client_id`, `client_secret` (if any) and space-delimited `scope` you want in the request body. This will typically look something like this:

```
POST /api/v1/oauth/shortcode HTTP/1.1
Accept: application/json, */*
Host: beam.pro
```

```
{
  "client_id": "fooclient",
  "scope": "interactive:robot:self"
```

```
}
```

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
```

```
{
  "code": "8WPVHT",
  "expires_in": 120,
  "handle": "Lc7eBcB78d5gZmqH0ajMH3QnmFPrxLGr"
}
```

2. Display the short six-digit `code` to the user and prompt them to enter it on [beam.pro/go](https://beam.pro/go). You can view a user's perspective of this process [here](#).
3. Continuously poll `GET /oauth/shortcode/check/{handle}`. It will give you one of a few statuses back:
  - **204 No Content** indicates we're still waiting on the user to enter the code.
  - **403 Forbidden** indicates the user denied your requested permissions.
  - **404 Not Found** indicates that the handle is invalid or expired.
  - **200 OK** is returned once the user has granted your application access. The response body will contain an `code`.

A poll which results in a 200 response might look something like this:

```
GET /api/v1/oauth/shortcode/check/Lc7eBcB78d5gZ... HTTP/1.1
Accept: application/json, */*
Host: beam.pro
```

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
```

```
{
  "code": "r2BTMKCZJJyIuoNE"
}
```

4. The client app should use this code as an OAuth Authorization Code which it can [exchange](#) for access and refresh tokens. The `redirect_uri` in the exchange request is not required and will be ignored, but you must present a valid `client_id`. A request to exchange the tokens might look something like this:

```
POST /api/v1/oauth/token HTTP/1.1
Accept: application/json, */*
```

```

Content-Type: application/json
Host: beam.pro

{
  "client_id": "fooclient",
  "code": "r2BTMKCZJJyIuoNE",
  "grant_type": "authorization_code"
}

HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

{
  "access_token": "pHktaORPcQGejnz48rJQdDWh1AJpevs \
    TWnvKrZW5z2HP3lgEqhp9gzje1YfblI02",
  "expires_in": 21599,
  "refresh_token": "HzCZSviiueoWsfcT6kh6d4n7SHUnfK \
    cFTRI0yHkgykjaCSIT5ctTqUKNTXfWsxfg",
  "token_type": "Bearer"
}

```

- The response will include an `access_token`, which you should send in the Authorization header when you connect to Interactive, prefixed with `Bearer` per standard OAuth behavior. In the above example, the client would then connect to Interactive and present the header `Authorization: Bearer pHktaORPcQGejnz48rJQdDWh1AJpevsTWnvKrZW5z2...`

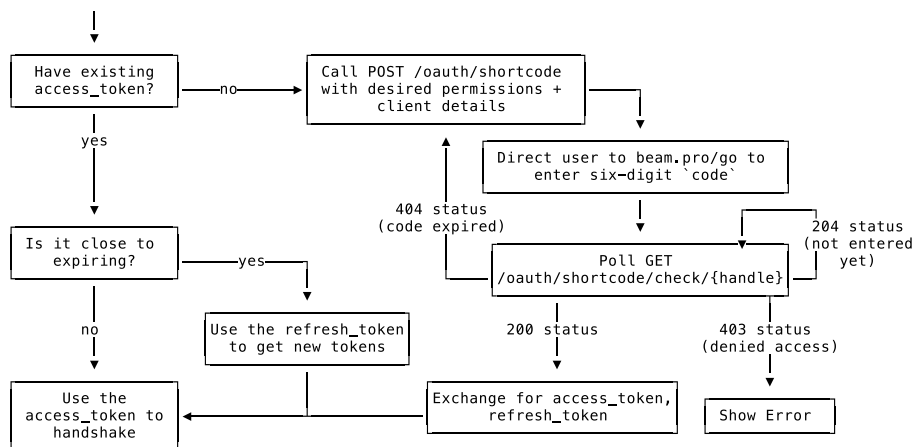


Figure 2: Overview of the mechanics of the alternative OAuth flow.

## Wire Protocol

Interactive’s protocol is similar to the protocol used in Constellation, which is similar to JSON-RPC with additional support for events, with the exception that RPC calls may be bi-directional. Additionally, multiple compression algorithms are supported and may be switched on-demand by the client.

The protocol consists of JSON objects, or “packets”, sent via websocket data frames. Each frame **MUST** contain exactly one JSON packet. There are two packet types: **method** and **reply**. Packet fields which are unused in a given context—empty objects {} in method parameters, for example—**MAY** be set to either **null** or be omitted entirely.

All timestamps used in the protocol are millisecond-precision UTC unix timestamps.

Method packets are sent in a way very similar to JSON-RPC. This is the only packet the client may send to the server. A method may look like the following:

```
{
  "type": "method",
  "id": 123,
  "discard": false,
  "method": "divide",
  "params": {
    "numerator": 16,
    "denominator": 4
  }
}
```

- **type** is **MUST** be set to “method”
- **method** **MUST** be the name of the method to call
- **params** **MUST** be an object, not an array, of named arguments to pass into the method.
- **id** **MUST** be any 32-bit unsigned integer. It’s included in the reply packet and used to correlate replies from the socket. You should ensure that each request has a unique id within your session. Attempting to use a floating point number or an integer outside of the 32-bit range **MAY** result in the number being truncated or overflowing, or **MAY** cause the client to reply with an error.
- **discard** **MAY** be set to **true** if the server does not require a reply for the method call. The client **MUST** effect any state changes regardless of the value of **discard**. The client **MAY** respect the server’s request to discard the successful response, but **MUST** reply with an error if one does occur.

Reply packets are sent in response to method packets. Replies are always sent in response to methods unless the socket closes before they may be sent. Some reply packets may look like the following:

```

{
  "type": "reply",
  "result": 4,
  "error": null,
  "id": 123
}

{
  "type": "reply",
  "result": null,
  "error": {
    "code": 4000,
    "message": "Cannot divide by zero.",
    "path": "denominator"
  },
  "id": 124
}

```

- **type** MUST be set to “reply”.
- **id** MUST be set to the id passed in the corresponding **method** packet, except if an error occurs wherein a packet cannot be parsed as JSON (code 4000), in which case the **id** MUST be set to 0.
- **result** SHALL be the unstructured result of the method. It SHOULD be null if an error occurs, but it MAY be null for successful replies.
- **error** MUST be a JSON object if an error occurred, or null otherwise. If present it MUST always contain a **code** (which is generally an integer between 4000 and 4999, inclusive, in accordance with application-level errors as defined in the websocket specification) and an associated plain text **message**.

It MAY include the path to the property or properties which caused the error, when appropriate. This will be expressed in dot notation relative to the reply **params**. For example, the following object...

```

{
  "foo": 1,
  "bar": [
    2,
    { "baz": 3 }
  ]
}

```

Can be referenced with these paths:

```

foo
bar.0
bar.0.baz

```



Note that if fatal errors occur as a result of a method call, a websocket close frame **MUST** be sent instead of a reply. The close frame's code and associated message **SHOULD** be the same as that which otherwise would have been sent in `reply.error`.

Multiple messages **MAY** be concatenated together in a single websocket frame as a JSON array:

```
[
  {
    "type": "method",
    "id": 123,
    "method": "divide",
    "params": {
      "numerator": 16,
      "denominator": 4
    }
  },
  {
    "type": "reply",
    "result": 4,
    "id": "123",
    "error": null
  }
]
```

## Compression

All communication between the client and server **MUST** be initialized with the **none** compression scheme and plain text messages via the **none** encoding **MUST** always be accepted by implementations. This requirement exists so that if the client or server detects a degraded state

The current compression scheme can be changed with a call to the `setCompression` method as described below. Common compression schemes include the following:

### **none**

All implementations **MUST** include the **none** compression scheme and **MUST** select it by default for new connections. In this scheme, JSON objects **MUST** be sent in plain text down textual websocket data frames.

## gzip

Compresses data using the **gzip** algorithm. Messages **MUST** be sent in binary websocket data frames and be prefixed with the uncompressed data's length as an unsigned [variable-length integer](#). In pseudo-code:

```
message = encode_varint(len(packet)) + deflate(packet)
```

The client and server **MUST** reuse the same gzip data stream to send messages, flushing writers to end each frame (using `Z_SYNC_FLUSH`, if using `zlib`). Likewise the readers in both the client and server should treat incoming messages as sequential parts of the same data stream.

Sample code is available in [this repository](#).

## lz4

LZ4 compression may be chosen as an alternative to gzip, providing significantly faster compression and decompression than `zlib` at the cost of a poorer compression ratio. Implementation is similar to **gzip**.

Messages **MUST** be sent in binary websocket data frames and be prefixed with the uncompressed data's length as an unsigned [variable-length integer](#). The client and server **MUST** reuse the same LZ4 data stream to send messages, flushing writers to end each frame. Likewise the readers in both the client and server should treat incoming messages as sequential parts of the same data stream.

Sample code is available in [this repository](#).

## Error Codes

Code	Cause	Method
1011	Sent in a close or method reply if an unknown internal error occurs.	*
1012	Sent in a close frame when we deploy or restart interactive; clients should attempt to reconnect.	*
4000	Error parsing payload as JSON.	*
4001	Error decompressing a compressed frame.	*
4002	Unknown packet type.	*
4003	Unknown method name.	*
4004	Error parsing method arguments.	*
4005	Etag mismatch.	multiple
4006	Unknown or expired transaction ID.	<b>capture</b>
4007	The user doesn't have enough sparks.	<b>capture</b>
4008	Unknown group ID specified.	multiple

Code	Cause	Method
4009	The specified group already exists.	<code>createGroups</code>
4010	Unknown scene ID specified.	<code>multiple</code>
4011	The specified scene already exists.	<code>createScenes</code>
4012	Unknown control ID specified.	<code>multiple</code>
4013	The specified control already exists.	<code>createControls</code>
4014	Unknown control type.	<code>createControls</code>
4015	Unknown participant ID specified.	<code>updateParticipants</code>
4016	Sent in a close frame to the frontend when interactive session has ended.	<code>*</code>
4017	Sent in a close frame if the GameClient exceeds memory usage limits.	<code>*</code>
4018	You cannot delete a default resource.	<code>deleteScene,</code> <code>deleteGroup</code>
4019	Authentication failed.	<code>Initial</code> <code>Connection</code>
4020	The interactive version is not found, or you do not have access to it.	<code>Initial</code> <code>Connection</code>
4021	A different interactive session is already running for the channel.	<code>Initial</code> <code>Connection</code>
4022	The channel is not online. (Participant socket only)	<code>Initial</code> <code>Connection</code>
4099	Bad user input.	<code>giveInput</code>

## Synchronization

The state of users, controls, and scenes can be changed by both the client and the mediator in response to aggregations, where each “document” is owned by the server. Each time a state is changed, a new etag SHALL be assigned. Etags are provided as strings and should be treated as opaque tokens. In order to subsequently modify the state, the requesting party must present its existing etag.

Each of these built-in documents have `meta` properties, which is a map that can be used to nest unstructured, free-form objects that can be consumed by custom frontend controls. Each document in the `meta` property has its own etag.

For example, a full Button control object might look something like this:

```
{
  "controlID": "win_the_game_btn",
  "kind": "button",
  "etag": "43590660",
  "text": "Win the Game",
  "cost": 0,
```

```

    "progress": 0.25,
    "disabled": false
  }

```

In order to update the “disabled” state, the caller should execute a method like:





```

{
  "type": "method",
  "id": 123,
  "method": "updateControls",
  "params": {
    "sceneID": "my awesome scene",
    "controls": [
      {
        "controlID": "win_the_game_btn",
        "etag": "43590660",
        "disabled": true
      }
    ]
  }
}

```

## Methods

This section uses icons beside each method name to denote who may implement them. In general, methods outside of the “core” methods are not required to be implemented by the client.

-  indicates the Beam server **MUST** implement the method.
-  indicates the client software **MUST** implement the method.
-  indicates the Beam server **MAY** implement the method.
-  indicates the client software **MAY** implement the method.

## Core & Authentication

### ready

The client should call the **ready** method after initialization is complete to signal to the mediator that it’s ready to have clients connect and start interacting. It can optionally call the method later with **isReady: false** to enter another initialization phase.

```

{
  "type": "method",
  "id": 123,

```

```

    "method": "ready",
    "params": {
      "isReady": true
    },
    "discard": true
  }

```

- A successful reply:

```

{
  "type": "reply",
  "result": null,
  "error": null,
  "id": 123
}

```

### setCompression S C

`setCompression` changes the compression algorithm used to encode messages. The server's response MUST be sent in the `text` scheme and, if the change is successful, subsequent messages MUST be compressed in the new scheme. If a call to `setCompression` results in the current compression scheme being chosen, the server MUST reset any state associated with the current scheme.

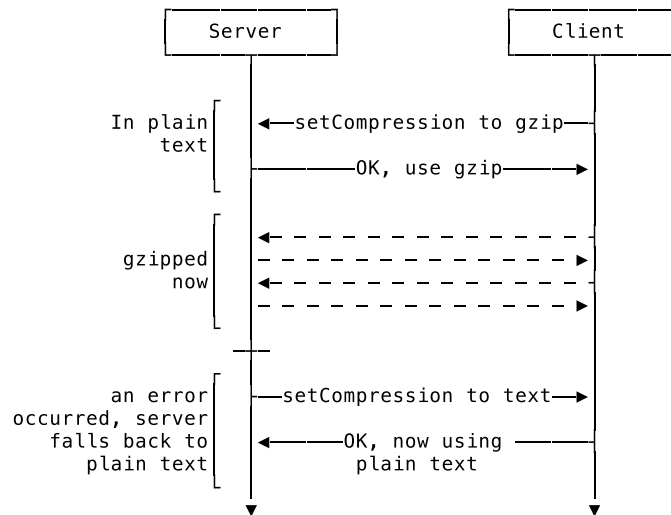


Figure 3: Diagram of compression state in a connection life cycle. The server and client agree to upgrade to gzip, before an error on the server requires a downgrade.

It takes a single string argument, **scheme**, which MUST be set to an array of preferred schemes in order of preference, from greatest to least preference:

```
{
  "type": "method",
  "id": 123,
  "method": "setCompression",
  "params": {
    "scheme": ["lz4", "gzip"]
  }
}
```

- A successful reply will contain the chosen scheme:

```
{
  "type": "reply",
  "result": {
    "scheme": "gzip"
  },
  "error": null,
  "id": 123
}
```

If no preferred common scheme is found, the server MUST fall back to the plain **text** scheme.

## onReady

The server SHALL call this method when the “ready” state changes.

```
{
  "type": "method",
  "id": 123,
  "method": "onReady",
  "params": {
    "isReady": true
  },
  "discard": true
}
```

## getTime

The server SHALL respond to this method with its current time, given as a milliseconds UTC unix timestamp. The client MAY expect this to be reasonably accurate against NTP. It's recommended that the client use this for clock synchronization, particularly when running on devices where the operating system's clock may not be accurate.

```
{
  "type": "method",
  "id": 123,
  "method": "getTime",
  "params": {}
}
```

- A successful reply:

```
{
  "type": "reply",
  "result": {
    "time": 1480432198536,
  },
  "error": null,
  "id": 123
}
```

### getMemoryStats

In response to this method the server will return an object identical to the `params` in `issueMemoryWarning`; a dump of information about the memory cap and current allocations. The `totalBytes` MAY be zero in cases where memory usage isn't capped, namely when using a standalone local build.

```
{
  "type": "method",
  "id": 123,
  "method": "getMemoryStats",
  "params": {}
}
```

- A successful reply:

```
{
  "type": "reply",
  "result": {
    "usedBytes": 1024,
    "totalBytes": 2048,
    "resources": [
      {
        "id": "default",
        "ownBytes": 400,
        "cumulativeBytes": 1024,
        "resources": [/* recursive */]
      }
    ]
  },
  "error": null,
}
```

```

    "id": 123
  }

```

### issueMemoryWarning

The server SHALL call this method when the client reaches a prescribed threshold in their memory limit. It will contain the number of bytes the client has allocated out of the total number of bytes, in addition to a breakdown of how much memory is allocated where for debugging purposes. If the client crosses the memory limit their connection may be terminated with a 4018 error code.

```

{
  "type": "method",
  "id": 123,
  "method": "issueMemoryWarning",
  "discard": true,
  "params": {
    "usedBytes": 1024,
    "totalBytes": 2048,
    "resources": [
      {
        "id": "default",
        "ownBytes": 400,
        "cumulativeBytes": 1024,
        "resources": [/* recursive */]
      }
    ]
  }
}

```

### setBandwidthThrottle

This method may be called on the server to set throttling for certain server-to-client method calls, such as `giveInput`, which could become problematic in very high-traffic scenarios. It implements a [leaky bucket algorithm](#); the client specifies the total bucket capacity in bytes and its drain rate in bytes per second. The `params` contains a map of method names to their throttle rules. Throttling previously enabled on a method can be disabled by setting it to null.

```

{
  "type": "method",
  "id": 123,
  "method": "setBandwidthThrottle",
  "discard": true,
  "params": {
    "giveInput": {
      "capacity": 10000000,

```



```

        "drainRate": 3000000
    },
    "onParticipantJoin": {
        "capacity": 0,
        "drainRate": 0
    },
    "onParticipantLeave": null
}
}

```

- A successful reply:

```

{
    "type": "reply",
    "result": null,
    "error": null,
    "id": 123
}

```

### getThrottleState

This method exposes statistics for the number of dropped and sent packets as a result of throttling rules set up in `setBandwidthThrottle`. It returns the number of sent packets (ones **inserted** into the bucket) and the number of **rejected** packets.

```

{
    "type": "method",
    "id": 123,
    "method": "getThrottleState",
    "params": null
}

```

- A successful reply:

```

{
    "type": "reply",
    "result": {
        "giveInput": {
            "inserted": 354892,
            "rejected": 481
        },
        "onParticipantJoin": {
            "inserted": 0,
            "rejected": 5983
        }
    },
}

```

```

    "error": null,
    "id": 123
  }

```

### issueMemoryWarning

The server SHALL call this method when the client reaches a prescribed threshold in their memory limit. It will contain the number of bytes the client has allocated out of the total number of bytes, in addition to a breakdown of how much memory is allocated where for debugging purposes. If the client crosses the memory limit their connection may be terminated with a 4018 error code.

```

{
  "type": "method",
  "id": 123,
  "method": "issueMemoryWarning",
  "discard": true,
  "params": {
    "usedBytes": 1024,
    "totalBytes": 2048,
    "resources": [
      "id": "default",
      "ownBytes": 400,
      "cumulativeBytes": 1024,
      "resources": [/* recursive */]
    ]
  }
}

```

### hello

The server SHALL call the **hello** method when the connection is authenticated and fully established.

```

{
  "type": "method",
  "id": 123,
  "method": "hello",
  "params": null,
  "discard": true
}

```

## Participants & Groups

The Participant object returned from many methods in this section contains the following properties. Settable properties are tagged, see the [Synchronization](#) section for further details.

- **sessionID**, a unique string identifier for the user in this session. It's used for all participant identification internally, and should be viewed as an opaque token.
- **userID**, indicating the user ID on Beam. This is an unsigned integer and does not vary.
- **username**, the user's name on on Beam, as a UTF-8 string.
- **level**, the user's Beam level, an unsigned integer.
- **lastInputAt**, set to the unix milliseconds timestamp when the user last interacted with the controls.
- **connectedAt**, the unix milliseconds timestamp when the user connected.
- **disabled** (settable), a boolean set to true if the user's input as been disabled.
- **groupID** (settable), a string referencing the group the user belongs to.
- **meta** is a map of custom property names to etagged values; you can use this to attach custom metadata to users which will be visible to custom controls. The values of these are permitted to be any valid JSON.

```
{
  "sessionID": "efe5e1d6-a870-4f77-b7e4-1cfaf30b097e",
  "etag": "54600913",
  "userID": 146,
  "username": "connor",
  "level": 67,
  "lastInputAt": 1484763854277,
  "connectedAt": 1484763846242,
  "disabled": false,
  "groupID": "default",
  "meta": {
    "is_awesome": {
      "etag": 37849560,
      "value": true
    }
  }
}
```

The user always belongs to a group, and is assigned to the magic **default** group when they first join.

The server provides methods like **getAllParticipants** and **getGroups** which return the state stored on the server. However, the client should store this information and incrementally update it by tracking calls to methods such as

onParticipantJoin; round trip time to Beam servers will often be in the hundreds of milliseconds, a noticeable delay in many situations. This is the reason that more focused methods, such as a method to get only the group names, are not provided—the client should maintain state such that that information can be pulled out of memory.

### getAllParticipants

The server SHALL return all currently connected participants as a reply to this method. This method returns up to 100 participants for each request; participant join dates continuation tokens. Iterating over the list in pseudo-code might look something like the following:

```
def loop_over_all_spectators(fn):
    last_connected_time = 0
    while True:
        result = server.getAllParticipants({ 'from': last_connected_time })
        for s in result['participants']:
            fn(s)

        if not result['hasMore']:
            return

        last = result['participants'][len(result['participants']) - 1]
        last_connected_time = last['connectedAt']
```

An example request:

```
{
  "type": "method",
  "id": 123,
  "method": "getAllParticipants",
  "params": {
    "from": 0
  }
}
```

- A successful reply:

```
{
  "type": "reply",
  "result": {
    "participants": [ /* an array of Participant objects */,
    "total": 4912,
    "hasMore": true
  ],
  "error": null,
```

```
    "id": 123
  }
```

### onParticipantJoin

The server SHALL call this method when a participant joins the integration. It SHALL also be sent to the participant who joined.

```
{
  "type": "method",
  "id": 123,
  "method": "onParticipantJoin",
  "params": {
    "participants": [/* an array of Participant objects */]
  },
  "discard": true
}
```

### onParticipantLeave

The server SHALL call this method when a participant leaves the integration.

```
{
  "type": "method",
  "id": 123,
  "method": "onParticipantLeave",
  "params": {
    "participants": [/* an array of Participant objects */]
  },
  "discard": true
}
```

### onParticipantUpdate

The server SHALL call this method when of the participant's state (for example, its group) is updated. The server will also send this to the updated participant.

```
{
  "type": "method",
  "id": 123,
  "method": "onParticipantUpdate",
  "params": {
    "participants": [/* an array of Participant objects */]
  },
}
```

```

    "discard": true
}

```

## getActiveParticipants [S](#)

The server SHALL return all participants who gave input after the specified threshold time, where the threshold is given as a unix milliseconds timestamp, in ascending order by the time they last gave input. It may return a limited set of results in the case when the list of participants is large. In this case, the client should advance the `threshold` value to the `lastInputAt` time of the last participant and keep calling until `hasMore` is `false`. Bear in mind that you may receive duplicate records for each participant.

Iterating over the list in pseudo-code might look something like the following:

```

def get_active_participants(threshold):
    output = []
    while True:
        result = server.getActiveParticipants({ 'threshold': threshold })
        for participant in result['participants']:
            if participant['id'] not in [s['id'] for s in output]:
                output.append(participant)

        if not result['hasMore']:
            return output

        last = result['participants'][len(result['participants']) - 1]
        threshold = last['lastInputAt']

{
  "type": "method",
  "id": 123,
  "method": "getActiveParticipants",
  "params": {
    "threshold": 1480432203927
  }
}

```

- A successful reply:

```

{
  "type": "reply",
  "result": {
    "participants": [ /* an array of Participant objects */,
    "total": 4912,
    "hasMore": true
  ],
}

```

```

    "error": null,
    "id": 123
  }

```

## updateParticipants [S](#)

Bulk-updates participants objects. Each participant in the array **MUST** contain the participant ID to update, along with zero or more properties which should be updated. The objects **MUST** contain the current **etag** the client thinks that each participant is tagged with. The server will respond with a list of updated participants and their new etags.

The patch **SHALL** either be applied for all participants and properties or fail; in no case will the server apply only a subset of the updates. If a participant is listed who is not connected to the integration, the update to that participant will be ignored.

```

{
  "type": "method",
  "id": 123,
  "method": "updateParticipants",
  "params": {
    "participants": [
      {
        "sessionID": "505cfe7c-123f-40e7-8c78-754103d16531",
        "etag": "100650688",
        "group": "red_team",
        "meta": {
          "is_awesome": {
            "etag": 37849560,
            "value": false
          }
        }
      }
    ]
  }
}

```

- A successful reply:

```

{
  "type": "reply",
  "result": {
    "participants": [ /* an array of Participant objects */ ]
  },
  "error": null,
}

```

```

    "id": 123
  }

```

- An example server response if an etag is mismatched:

```

{
  "type": "reply",
  "result": null,
  "error": {
    "code": 4005,
    "message": "Etag mismatch.",
    "path": "participants.0.etag"
  },
  "id": 123
}

```

- An example server response if the group does not exist:

```

{
  "type": "reply",
  "result": null,
  "error": {
    "code": 4009,
    "message": "Unknown group ID specified.",
    "path": "participants.0.group"
  },
  "id": 123
}

```

## createGroups

**createGroups** creates one or more new groups. Each group can be set to an initial Scene, default to the **default** scene if one is not provided. Group IDs MUST be unique. The client MAY provide an initial etag for the groups.

```

{
  "type": "method",
  "id": 123,
  "method": "createGroups",
  "params": {
    "groups": [
      {
        "groupID": "red_team",
        "etag": "203125580",
        "sceneID": "has_control"
      },
      {

```



```

        "groupID": "blue_team",
        // will be assigned to the default scene
    }
  ]
}
}

```

- A successful reply:

```

{
  "type": "reply",
  "result": null,
  "error": null,
  "id": 123
}

```

- A reply that occurs if the group already exists:

```

{
  "type": "reply",
  "result": null,
  "error": {
    "code": 4010,
    "message": "The specified group already exists.",
    "path": "groups.0.groupID"
  },
  "id": 123
}

```

- A reply that occurs if the scene does not exist:

```

{
  "type": "reply",
  "result": null,
  "error": {
    "code": 4011,
    "message": "Unknown scene ID specified.",
    "path": "groups.0.sceneID"
  },
  "id": 123
}

```

## getGroups

getGroups lists all groups connected to the integration.

```

{
  "type": "method",

```

```

    "id": 123,
    "method": "getGroups",
    "params": {}
  }

```

- A successful reply:

```

{
  "type": "method",
  "id": 123,
  "method": "getGroups",
  "params": {
    "groups": [
      {
        "groupID": "red_team",
        "etag": "203125580",
        "sceneID": "has_control"
      }
    ]
  }
}

```

## updateGroups S

Updates groups that already exist. The array of groups MUST contain each group's ID, along with zero or more properties which should be updated. The objects MUST contain the current **etag** the client thinks that each group is tagged with. The server will respond with a list of the updated groups, including their new etags.

The patch SHALL either be applied for all groups and properties or fail; in no case will the server apply only a subset of the updates.

```

{
  "type": "method",
  "id": 123,
  "method": "updateGroups",
  "params": {
    "groups": [
      {
        "groupID": "red_team",
        "etag": "205390751",
        "sceneID": "lobby"
      }
    ]
  }
}

```

- A successful reply:

```
{
  "type": "reply",
  "result": {
    "groups": [ /* an array of Group objects */ ]
  },
  "error": null,
  "id": 123
}
```

- An example server response if a group does not exist:

```
{
  "type": "reply",
  "result": null,
  "error": {
    "code": 4009,
    "message": "Unknown group ID specified.",
    "path": "groups.0.groupID"
  },
  "id": 123
}
```

- An example server response if the group does not exist:

```
{
  "type": "reply",
  "result": null,
  "error": {
    "code": 4011,
    "message": "Unknown group ID specified.",
    "path": "groups.0.sceneID.value"
  },
  "id": 123
}
```

- An example server response if an etag is mismatched:

```
{
  "type": "reply",
  "result": null,
  "error": {
    "code": 4005,
    "message": "Etag mismatch.",
    "path": "groups.0.etag"
  },
  "id": 123
}
```

## deleteGroup

Removes a group by id, reassigning any users who were in that group to a different one. The server MAY not return an error if the group to remove does not exist.

```
{
  "type": "method",
  "id": 123,
  "method": "deleteScene",
  "params": {
    "groupID": "my awesome group",
    "reassignGroupID": "my other group"
  }
}
```

- A successful reply:

```
{
  "type": "reply",
  "result": null,
  "error": null,
  "id": 123
}
```

- An example server response given if the group to reassign to does not exist:

```
{
  "type": "reply",
  "result": null,
  "error": {
    "code": 4011,
    "message": "Unknown group ID specified."
  },
  "id": 123
}
```

- An example server response given if attempting to delete the default group:

```
{
  "type": "reply",
  "result": null,
  "error": {
    "code": 4019,
    "message": "You cannot delete a default resource."
  },
  "id": 123
}
```

### onGroupCreate

The server SHALL call this method when a new group is created.

```
{
  "type": "method",
  "id": 123,
  "method": "onGroupCreate",
  "params": {
    "groups": [/* an array of Group objects */]
  },
  "discard": true
}
```

### onGroupDelete

The server SHALL call this method when a group is deleted.

```
{
  "type": "method",
  "id": 123,
  "method": "onGroupDelete",
  "params": {
    "groupID": "deleted_group_id",
    "reassignGroupID": "reassigned_group_id",
  },
  "discard": true
}
```

### onGroupUpdate

The server SHALL call this method when a group is updated. This SHALL NOT be called when a participant within the group is updated, only when an attribute of the group (e.g. a **meta** property or the **sceneID**) has changed.

```
{
  "type": "method",
  "id": 123,
  "method": "onGroupUpdate",
  "params": {
    "groups": [/* an array of Group objects */]
  },
  "discard": true
}
```

## Scene Setup

Although scenes and controls can be created via our interactive studio, the game client also has full control over their display and can manipulate them at runtime.

Each control is identified uniquely by its ID, a UTF-8 string, and has an **etag** which must be presented to change that its properties. For an example of what a Button might look like in this form, see the Synchronization section.

## Built-In Controls

The following are a list of built in controls, which can be serialized as JSON into Control objects. Certain properties are static, intrinsic properties of the control, and cannot be modified. These are marked as as “not settable”. Like Participants, controls have a **meta** property which you can use to annotate them with custom data.

- Buttons
  - **controlID: string** (not settable)  
A unique control ID.
  - **kind: string** (not settable)  
Must be set to “button”.
  - **keycode: integer**  
A JavaScript keycode which participant’s use to trigger this button via their keyboard.
  - **text: string**  
The text displayed on a button.
  - **cost: integer**  
The cost in sparks involved in pressing a button. Setting this to a non-zero value will cause a transaction to be created when the button is pressed, and included in the data sent down to the client.
  - **progress: float**  
Renders a process bar on the button. Should be given in the range [0, 1], where 1 causes the bar to be completely full.
  - **cooldown: integer**  
Triggers a cooldown that lasts until the provided unix timestamp.
  - **disabled: bool**  
Disables the control.
  - **Position[]: An array of position objects** Contains an array of position objects as described in Control Positioning.
- Joysticks
  - **controlID: string** (not settable)  
A unique control ID.
  - **kind: string** (not settable)  
Must be set to “joystick”.

- **sampleRate: integer**  
Participant's JoySticks use this to determine how often they should send a move event to the server.
- **angle: number**  
Displays a "halo" on screen at the angle specified. Should be given in the range  $[0, 2\pi)$ .
- **intensity: number**  
Changes the opacity of the halo effect.
- **disabled: bool**  
Disables the control.
- **Position[]: An array of position objects** Contains an array of position objects as described in Control Positioning.

## Control Positioning

Within scenes, controls are rendered by the Beam Frontend on three different grids depending on the screen resolution of the participant. Each grid has a size: "small", "medium", or "large". Based on the participant's screen resolution one of these grids is picked to be displayed. If the participant resizes their screen, the frontend controls will automatically adjust to the best grid for the new resolution.

Measurements within a grid are created with a grid scale where 1 unit on the grid is 12 pixels on the screen.

Each grid has a width and height according to this scale, and a minimum width in pixels the participant's device must have for this grid to be used. Their configurations are as follows:

- Large - Used on large displays.
  - Width: 80
  - Height: 20
  - Minimum Width: 900px
- Medium - Used on smaller displays and large tablets.
  - Width: 45
  - Height: 25
  - Minimum Width: 540px
- Small - Used on Mobile Phones.
  - Width: 30
  - Height: 40
  - Minimum Width: 0px

To specify the location and size of a control on a grid, a position object **MUST** be specified when creating a control dynamically. These are stored in the **position** array of a control. A position **SHOULD** be specified for each grid. The Interactive Studio can be used to experiment with control positioning.

A position object has the following properties:

- **size:** **string**  
The size of the grid this position object is for.
- **width:** **number** The width of this control.
- **height:** **number** The height of this control.
- **x:** **number** The position of this control on the x axis of the grid.
- **y:** **number** The position of this control on the y axis of the grid.

### createScenes

Creates new scenes. The sceneIDs can be any valid UTF-8 sequence of characters. You can optionally choose to provide an array of **controls** to set on the scene initially rather than requiring further **addControl** calls. The client **MUST** provide a fully-qualified, tagged control object in this method; the etags provided will be used as their initial values. Either all scenes and controls will be created or an error will be returned, the server **SHALL NOT** apply partial updates.

```
{
  "type": "method",
  "id": 123,
  "method": "createScenes",
  "params": {
    "scenes": [
      {
        "sceneID": "my awesome scene",
        "etag": "252185589",
        "controls": [ // array of control objects
          {
            "controlID": "win_the_game_btn",
            "etag": "262111379",
            "kind": "button",
            "text": "Win the Game",
            "cost": 0,
            "progress": 0.25,
            "disabled": false,
            "meta": {
              "glow": {
                "etag": 254353748,
                "value": {
                  "color": "#f00",
                  "radius": 10
                }
              }
            }
          }
        ]
      }
    ]
  }
}
```



```

    }
  }
}
]
}
]
}
}

```

- A successful reply:

```

{
  "type": "reply",
  "result": {
    "scenes": [/* an array of Scene objects */]
  },
  "error": null,
  "id": 123
}

```

- A response given if the scene already exists:

```

{
  "type": "reply",
  "result": null,
  "error": {
    "code": 4012,
    "message": "The specified scene already exists.",
    "path": "scenes.0.sceneID"
  },
  "id": 123
}

```

## getScenes

**getScenes** returns a list of scenes and their controls and groups.

```

{
  "type": "method",
  "id": 123,
  "method": "getScenes",
  "params": {}
}

```

- A successful reply:

```

{

```

```

    "type": "reply",
    "result": {
      "scenes": [
        {
          "sceneID": "red_team_scene",
          "groups": /* array of Group objects */,
          "controls": /* array of Controls objects */
        },
        // ...
      ]
    },
    "error": null,
    "id": 123
  }

```

### deleteScene

Removes a scene by id, reassigning any groups who were on that scene to a different one. The server MAY not return an error if the scene to remove does not exist.

```

{
  "type": "method",
  "id": 123,
  "method": "deleteScene",
  "params": {
    "sceneID": "my awesome scene",
    "reassignSceneID": "my other scene"
  }
}

```

- A successful reply:

```

{
  "type": "reply",
  "result": null,
  "error": null,
  "id": 123
}

```

- An example server response given if the scene to reassign to does not exist:

```

{
  "type": "reply",
  "result": null,
  "error": {
    "code": 4011,

```

```

        "message": "Unknown scene ID specified."
    },
    "id": 123
}

```

- An example server response given if attempting to delete the default scene:

```

{
    "type": "reply",
    "result": null,
    "error": {
        "code": 4019,
        "message": "You cannot delete a default resource."
    },
    "id": 123
}

```

## updateScenes

Updates scenes that already exist. The array of scenes MUST contain each scene's ID, along with zero or more properties which should be updated. The objects MUST contain the current **etag** the client thinks that each scene is tagged with. The server will respond with a list of the updated scenes, including their new etags.

The patch SHALL either be applied for all scenes and properties or fail; in no case will the server apply only a subset of the updates.

- A successful reply:

```

{
    "type": "reply",
    "result": {
        "scenes": [ /* an array of Scene objects */ ]
    },
    "error": null,
    "id": 123
}

```

- An example server response if a scene does not exist:

```

{
    "type": "reply",
    "result": null,
    "error": {
        "code": 4013,
        "message": "Unknown scene ID specified.",
        "path": "scenes.0.sceneID"
    }
}

```

```

    },
    "id": 123
  }

```

- An example server response if the scene does not exist:

```

{
  "type": "reply",
  "result": null,
  "error": {
    "code": 4011,
    "message": "Unknown scene ID specified.",
    "path": "scenes"
  },
  "id": 123
}

```

- An example server response if an etag is mismatched:

```

{
  "type": "reply",
  "result": null,
  "error": {
    "code": 4005,
    "message": "Etag mismatch.",
    "path": "scenes.0.etag"
  },
  "id": 123
}

```

### onSceneCreate

The server SHALL call this method when a new scene is created.

```

{
  "type": "method",
  "id": 123,
  "method": "onSceneCreate",
  "params": {
    "scenes": [/* an array of Scene objects */]
  },
  "discard": true
}

```

### onSceneDelete

The server SHALL call this method when a scene is deleted.

```
{
  "type": "method",
  "id": 123,
  "method": "onSceneDelete",
  "params": {
    "sceneID": "deleted_scene_id",
    "reassignSceneID": "reassigned_scene_id",
  },
  "discard": true
}
```

### onSceneUpdate

The server SHALL call this method when a scene is updated. This SHALL NOT be called when a control within the scene is updated, only when an attribute of the scene (i.e. a meta property) has changed.

```
{
  "type": "method",
  "id": 123,
  "method": "onSceneUpdate",
  "params": {
    "scenes": [/* an array of Scene objects */]
  },
  "discard": true
}
```

### createControls

Creates one or more new controls in a scene. The client MUST provide a fully-qualified, tagged control object in this method; the etags provided will be used as their initial values.

```
{
  "type": "method",
  "id": 123,
  "method": "createControls",
  "params": {
    "sceneID": "my awesome scene",
    "controls": [/* array of Control objects */]
  }
}
```

- A successful reply:

```
{
  "type": "reply",
  "result": null,
  "error": null,
  "id": 123
}
```

- An example server response if creating a control that already exists:

```
{
  "type": "reply",
  "result": null,
  "error": {
    "code": 4014,
    "message": "The specified control already exists.",
    "path": "controls.0.controlID"
  },
  "id": 123
}
```

- An example server response if the scene does not exist:

```
{
  "type": "reply",
  "result": null,
  "error": {
    "code": 4011,
    "message": "Unknown scene ID specified.",
    "path": "sceneID"
  },
  "id": 123
}
```

- An example server response if the control type is unknown:

```
{
  "type": "reply",
  "result": null,
  "error": {
    "code": 4015,
    "message": "Unknown control type.",
    "path": "controls.0.kind"
  },
  "id": 123
}
```

## deleteControls

Removes one or more controls by their ID.

```
{
  "type": "method",
  "id": 123,
  "method": "deleteControls",
  "params": {
    "sceneID": "lobby",
    "controlIDs": [/ * array of string ids */]
  }
}
```

- A successful reply:

```
{
  "type": "reply",
  "result": null,
  "error": null,
  "id": 123
}
```

## updateControls

Updates control objects already present in a scene. The target scene and array of controls **MUST** be provided. Each control **MUST** contain the ID to update, along with zero or more properties which should be updated. The objects **MUST** contain the current **etag** the client thinks that each control is tagged with. The server will respond with a list of updated controls and their new etags.

The patch **SHALL** either be applied for all controls and properties or fail; in no case will the server apply only a subset of the updates.

```
{
  "type": "method",
  "id": 123,
  "method": "updateControls",
  "params": {
    "sceneID": "my awesome scene",
    "controls": [
      {
        "controlID": "win_the_game_btn",
        "etag": "262111379",
        "disabled": true,
        "meta": {
          "glow": {
```

```

        "etag": 254353748,
        "value": false
      }
    }
  ]
}

```

- A successful reply:

```

{
  "type": "reply",
  "result": {
    "controls": [ /* an array of Control objects */ ]
  },
  "error": null,
  "id": 123
}

```

- An example server response if a control does not exist:

```

{
  "type": "reply",
  "result": null,
  "error": {
    "code": 4013,
    "message": "Unknown control ID specified.",
    "path": "controls.0.controlID"
  },
  "id": 123
}

```

- An example server response if the scene does not exist:

```

{
  "type": "reply",
  "result": null,
  "error": {
    "code": 4011,
    "message": "Unknown scene ID specified.",
    "path": "controls"
  },
  "id": 123
}

```

- An example server response if an etag is mismatched:

```

{
  "type": "reply",

```



```

    "result": null,
    "error": {
        "code": 4005,
        "message": "Etag mismatch.",
        "path": "controls.0.etag"
    },
    "id": 123
}

```

### onControlCreate

The server SHALL call this method when a new control is created.

```

{
    "type": "method",
    "id": 123,
    "method": "onControlCreate",
    "params": {
        "sceneID": "lobby",
        "controls": [/* an array of Control objects */],
    },
    "discard": true
}

```

### onControlDelete

The server SHALL call this method when a control is deleted.

```

{
    "type": "method",
    "id": 123,
    "method": "onControlDelete",
    "params": {
        "sceneID": "lobby",
        "controls": [
            {
                "controlID": "win_the_game_btn"
            },
            // ...
        ]
    },
    "discard": true
}

```

## onControlUpdate

The server SHALL call this method when a control is updated.

```
{
  "type": "method",
  "id": 123,
  "method": "onControlUpdate",
  "params": {
    "sceneID": "lobby",
    "controls": [/ * an array of Control objects */]
  },
  "discard": true
}
```

## Input

Input objects are polymorphic events that represent actions that a participant is taking. Each input includes an **event** string, such as **mousedown** on buttons, in addition to a short **control** object describing which contain it occurred. There are several types of built-in events which occur for the default controls, and custom controls which you create can fire custom events as well. These are the built-in event types:

- **mousedown**

Fired on a button when a participant presses their mouse on it. It contains a **button** property corresponding to the [JavaScript mouse button](#) that the participant used to click on the button. For touch events or input given on game consoles, the button will always be 0.

Additionally, if the button has an associated cost, a **Transaction** will be created and associated in the **transactionID** property of the params. For example:

```
"params": {
  "participantID": "b2f65dea-429f-4105-a280-745fd5d75945",
  "transactionID": "d69f689e-d4e6-42a4-8d9e-515772f9238f",
  "input": {
    "controlID": "win_the_game_btn",
    "event": "mousedown",
    "button": 0
  }
}
```

- **mouseup**

Fired on a button when a participant releases their mouse. This is identical to `mousedown`, but it will not contain a Transaction; sparks may only be deducted on `mousedown` events.

```

    "params": {
      "participantID": "b2f65dea-429f-4105-a280-745fd5d75945",
      "transactionID": "d69f689e-d4e6-42a4-8d9e-515772f9238f",
      "input": {
        "controlID": "win_the_game_btn",
        "event": "mouseup",
        "button": 0
      }
    }
  }

```

- **move**

Fired on joysticks when the participant moves the joystick. This is not sent continuously if the joystick is not moving, and is throttled so that no more than one event can be fired per used every 50 milliseconds (by default). It contains the `x` and `y` coordinates at which the joystick is positioned, where `[-1, -1]` is the upper left corner and `[1, 1]` is the lower right corner. Their magnitude will be  $||x, y|| \leq 1$ .

```

    "params": {
      "participantID": "b2f65dea-429f-4105-a280-745fd5d75945",
      "transactionID": "d69f689e-d4e6-42a4-8d9e-515772f9238f",
      "input": {
        "controlID": "move_participant",
        "event": "move",
        "x": 0.64,
        "y": -0.1
      }
    }
  }

```

## giveInput

`giveInput` is called on the client in firehose mode when a participant gives input. It contains the participant that gave the input and a list of input objects they provided. This method is not called until after the input is validated and will not be called for disabled participants.

```

{
  "type": "method",
  "id": 123,
  "method": "giveInput",
  "discard": true,
  "params": {

```

```

    "participantID": "b2f65dea-429f-4105-a280-745fd5d75945",
    "input": { /* an Input object */ }
  }
}

```

- A successful reply:

```

{
  "type": "reply",
  "result": null,
  "error": null,
  "id": 123
}

```

## Spark Transactions

Some actions, such as clicking a button with an associated code, can generate spark transactions. Each transaction is assigned a unique integer ID. These transactions work similarly to credit card transactions:

- The server creates “charges” in response to spark associated actions
- The charge is initially “uncaptured”
- You can “capture” the charge to submit it and deduct sparks.

Charges are expired five minutes after they’re created.

### capture

Calling capture SHALL cause the server to attempt to deduct a spark transaction from the participant. The server makes a best-effort to validate the charge before it’s created, blocking obviously invalid ones outright, but when possible the client SHOULD await a successful reply before effecting any associated action.

```

{
  "type": "method",
  "id": 123,
  "method": "capture",
  "params": {
    "transactionID": 1
  }
}

```

- A successful reply:

```

{
  "type": "reply",

```

```
    "result": null,  
    "error": null,  
    "id": 123  
  }
```

- Unsuccessful replies:

```
{  
  "type": "reply",  
  "result": null,  
  "error": {  
    "code": 4006,  
    "message": "Unknown or expired transaction ID.",  
    "path": "transactionID"  
  },  
  "id": 123  
}  
  
{  
  "type": "reply",  
  "result": null,  
  "error": {  
    "code": 4007,  
    "message": "The participant doesn't have enough sparks."  
  },  
  "id": 123  
}
```