UNIVERSITÄT ROSTOCK

Distributed Web Interface for Real-Time Spatial Audio Reproduction System

SoundScape Renderer

Héctor Daniel Villacián Ibáñez

Proyecto

Tema: Distribución de un programa de rendering de audio en tiempo real

Título: Distributed Web Interface for Real-Time Spatial Audio Reproduction System

Autor: Héctor Daniel Villacián Ibáñez

Titulación: Telemática

Centro de Lectura: Elektrotechnik

Tutor: Sascha Spors

Departamento: Nachrichtentechnik

Fecha de lectura: 16 de abril de 2013

Resumen

SSR es el acrónimo de SoundScape Renderer (tool for real-time spatial audio reproduction providing a variety of rendering algorithms), es un programa escrito en su mayoría en C++. El programa permite al usuario escuchar tanto sonidos grabados con anterioridad como sonidos en directo. El sonido o los sonidos se oirán, desde el punto de vista del oyente, como si el sonido se produjese en el punto que el programa decida, lo interesante de este proyecto es que el sonido podrá cambiar de lugar, moverse, etc. Todo en tiempo real.

Esto se consigue sin modificar el sonido al grabarlo pero sí al emitirlo, el programa calcula las variaciones necesarias para que al emitir el sonido al oyente le llegue como si el sonido realmente se generase en un punto del espacio o lo más parecido posible. La sensación de movimiento no deja de ser el punto anterior cambiando de lugar.

La idea era crear una aplicación web basada en Canvas de HTML5 que se comunicará con esta interfaz de usuario remota. Así se solucionarían todos los problemas de compatibilidad ya que cualquier dispositivo con posibilidad de visualizar páginas web podría correr una aplicación basada en estándares web, por ejemplo un sistema con Windows o un móvil con navegador.

El protocolo debía de ser WebSocket porque es un protocolo HTML5 y ofrece las “garantías” de latencia que una aplicación con necesidades de información en tiempo real requiere. Nos permite una comunicación full-dúplex asíncrona sin mucho payload que es justo lo que se venía a evitar al no usar polling normal de HTML.

El problema que surgió fue que la interfaz de usuario de red que tenía el programa no era compatible con WebSocket debido a un handshacking inicial y obligatorio que realiza el protocolo, por lo que se necesitaba otra interfaz de red. Se decidió entonces cambiar a JSON como formato para el intercambio de mensajes.

Al final el proyecto comprende no sólo la aplicación web basada en Canvas sino también un servidor funcional y la definición de una nueva interfaz de usuario de red con su protocolo añadido.

Abstract

This project aims to become a part of the SSR tool to extend its capabilities in the field of the access.

SSR is an acronym for SoundScape Renderer, is a program mostly written in C++ that allows you to hear already recorded or live sound with a variety of sound equipment as if the sound came from a desired place in the space.

Like the web-page of the SSR says surely better explained: “The SoundScape Renderer (SSR) is a tool for real-time spatial audio reproduction providing a variety of rendering algorithms.”

The application can be used with a graphical interface written in Qt but has also a network interface for external applications to use it. This network interface communicates using XML messages. A good example of it is the Android client. This Android client is already working.

In order to use the application should be run it by loading an audio source and the wanted environment so that the renderer knows what to do. In that moment the server binds and anyone can use the network interface.

Since the network interface is documented everyone can make an application to interact with this network interface. So the application can have as many user interfaces as wanted.

The part that is developed in this project has nothing to do neither with audio rendering nor even with the reproduction of the spatial audio. The part that is developed here is about the interface used in the SSR application. As it can be deduced from the title: “Distributed Web Interface for Real-Time Spatial Audio Reproduction System”, this work aims only to offer the interface via web for the SSR (“Real-Time Spatial Audio Reproduction System”).

The idea is not to make a new graphical interface for SSR but to allow more types of interfaces and communication. To accomplish the objective of allowing more graphical interfaces this project is going to use a new network interface. By now the SSR application is using only XML for data interchange but this new network interface support JSON.

This project comprehends the server that launch the application, the user interface and the new network interface. It is done with these modules in order to allow creating new user interfaces that can communicate with the server or new servers that can communicate with the user interface by defining a complete network interface for data interchange.

Table of contents

Distributed Web Interface for Real-Time Spatial Audio Reproduction System

INTRODUCCTION

This project aims to become a part of the SSR tool to extend its capabilities in the field of the access.

At first the whole SSR project will be briefly explained so that this project is localized. Then this project will be explained.

This work is mainly divided into five parts the explanations of all the technologies used beyond how they are used here, the network interface that was created to this purpose, the server where the main program is executed, the client where the interface reside and finally a user manual which I personally think can be useful approach of all the functionalities and work that had been done without reading the whole thesis.

SSR

SSR is an acronym for SoundScape Renderer, is a program written mostly in C++ that allows you hear already recorded or live sound with a variety of sound equipment as if the sound came from a place desired in the space.

Like the web-page[1] says surely better explained: “The SoundScape Renderer (SSR) is a tool for real-time spatial audio reproduction providing a variety of rendering algorithms.”

The application can be used with a graphical interface written in Qt but has also a network interface for external applications to use it. This network interface communicates using XML messages. A good example of it is the Android client. This Android client is already working.

In order to use the application you should run it loading an audio source and the wanted environment so that the renderer knows what to do. In that moment the server binds and anyone can use the network interface.

[1] http://spatialaudio.net/ssr/

Since the network interface is documented everyone can make an application to interact with this network interface. So the application can have many user interfaces.

CONTRIBUTION TO SSR

The part that is developed in this project has nothing to do with audio rendering or even reproduction of the spatial audio. The part that is developed here is about the interface used in the SSR application. As it can be deduced from the title: "Distributed Web Interface for Real-Time Spatial Audio Reproduction System", this work aims only to offer the interface via web for the SSR ("Real-Time Spatial Audio Reproduction System").

The idea is not to make a new graphical interface for SSR but to allow more types of interface and communication. To accomplish the objective of allowing more graphical interfaces this project is going to use a network interface.

SSR is already using one. By now the SSR application is using only XML for data interchange but in the future should be able to support more data interchange as for example JSON.

So the main program (from now on server) will in some moment be run in a computer connected to a network and from other computers connected to that network people could connect to the server. The server would now do the hard computing therefore can be run in a fast computer and the interfaces from where the users can access to the program and interact with it can be launched in any little or big device.

In this case the only limitation of the device used is that it should be able to connect to the same network as the server is and to be able to launch a browser, obviously compatible with the technology used.

Since this project will use WAMP subprotocol to communicate between the interface and the SSR application and this subprotocol works with JSON, in order to try the interface, a server for testing purposes has to be made.

This server is written in C++ and it works just like if it was the SSR application but without sending neither any audio information nor keeping track of all the changes the interface or interfaces are making. More or less works like a repeater but with more functionality which will be explained in detail later.

To know if the interfaces are connected to the server without changing anything then they can interact between each other just as they would do it in the normal SSR.

The graphical interface part is written with web technologies as HTML5, CSS3, JavaScript and Canvas. Any modern browser should be able to run this interface.

This interface is also a client since it communicates with the server and it uses the network interface, which is also explained below, to communicate either with the server itself or with every part involved through the server.

TECHNOLOGY USED IN THE PROJECT

To be able to accomplish all the objectives, a bunch of technology was used. The normal web solutions along with some libraries and communication tools were used to make things easier.

HTML5/CSS3

HTML [1] is the most extended markup language for showing web pages. HTML only structures the data. That is why CSS is needed. CSS is a style sheet language for XML documents as HTML. So CSS presents the date while HTML structures it.

The election of the fifth version of the HTML is not casualty, it is done like this because it include new audio, video and canvas tags. These tags are very convenient for the project especially canvas tag and for future improvements the audio tag can be very useful.

Every browser understands HTML and CSS so in that way it is also ensuring that the interface is more accessible which is one of the main ideas.

JavaScript

Do not confuse with Java, JavaScript [2] is an interpreted computer programming language. This means that the code is never compiled but executed directly. This property is comfortable for implement it in browser as a client side programming for the internet.

It is very flexible and its code can be separated into many file with almost no restriction. So it is in the hand of the programmer to make it readable or not.

The election of using JavaScript is more like an obligation when doing web application unless third parties tools are plugged to the browser.

Canvas

Canvas, as already said, is a part of HTML 5 [3]. Consist of an element, the canvas tag, which by using JavaScript can be used to paint. It is very low level and of course bit map oriented, which means that whatever it is wanted to keep track of it must be coded apart from canvas.

The election of this technology has nothing to do with making the project easier but lighter. Since canvas do not need so many resources as other technologies such as SVG.

SVG [4] would have been another good option because is vector oriented and XML based. It also handles a structure that for a GUI is better in difficulty terms. And another good thing is the event handlers are assigned to object in SVG which make programming lot easier.

On the other hand this would have led to a weightier interface. And canvas is included in HTML 5 so it is normally well supported.

WebSocket

WebSocket [5] is a protocol that enables a two-way communication over a unique TCP port between clients and server. WebSocket provide ways of sending information to a browser without soliciting it from the client side.

This could have been done by polling [5] although this way would have to use more network bandwidth and surely more delay.

Another approach could have been using a non-embedded technology. In this case the compatibility between devices that can use it and browsers would have decrease.

Finally WebSocket has minimal payload and is full-duplex asynchronous which makes it the best option.

WebSocket has to establish a connection that is requested by the browser. HTTP sees this like a protocol upgrade. It is called the handshake and it can be a little bit tricky.

In order to connect the browser sends a HTTP header to the server as for example:

GET /chat HTTP/1.1

Host: server.example.com

Upgrade: websocket

Connection: Upgrade

Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==

Origin: http://example.com

Sec-WebSocket-Protocol: chat, superchat, wamp

Sec-WebSocket-Version: 13

To prove that the handshake was received, the server has to take two pieces of information and combine them to form a response.

The first information comes from the *"Sec-WebSocket-Key"* header field in the client handshake.

For this header field, the server has to take the value and concatenate this with the Globally Unique Identifier "*258EAFA5-E914-47DA-95CA-C5AB0DC85B11*" in string form, which is unlikely to be used by network endpoints that do not understand the WebSocket Protocol. A SHA-1 hash (160 bits), base64-encoded, of this concatenation is then returned in the server's handshake like this:

HTTP/1.1 101 Switching Protocols

Upgrade: websocket

Connection: Upgrade

Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=

Sec-WebSocket-Protocol: wamp

Any other status code than 101 indicates that the WebSocket handshake has not completed.

From now on data can be transfer by WebSocket protocol.

WAMP

WAMP [6] is a subprotocol that relies on WebSocket und hence on TCP so has no need to establish intercommunication procedure. It only determines the lexis of the messages, nine kinds in total and all with JSON notation, and how they should interact. There are two main ways of communication by using WAMP.

One is by what they called PubSub, which is a system of subscriptions where the client can receive or send messages from the topic it is subscribed.

The second way of communication is by RPC which consist on a client asking for the result of a function that lies on the server. The functions do not need to have a return data.

But for any of this both methods the connection should be ready first. To do so a message from the server to the client to identify it is needed only when the WebSocket connection is done.

The WAMP has a particularity, for choosing the topics it uses a URI so that no topic is ever repeated. This URI's can sometimes be very long or either difficult to follow so WAMP subprotocol also has a way of using prefixes instead of the whole URI.

This prefixes are always done by petition from the client part and are always related to the client.

To make things more clear, this table shows the name of the message, the type ID each message from WAMP has to send at first, direction and category.

Message	Type ID	Direction	Category
WELCOME	0	Server-to-client	Auxiliary
PREFIX	1	Client-to-server	Auxiliary
CALL	2	Client-to-server	RPC
CALLRESULT	3	Server-to-client	RPC
CALLERROR	4	Server-to-client	RPC
SUBSCRIBE	5	Client-to-server	PubSub
UNSUBSCRIBE	6	Client-to-server	PubSub
PUBLISH	7	Client-to-server	PubSub
EVENT	8	Server-to-client	PubSub

Each message is a JSON array with the number coinciding with the type ID. The structure of each one is going to be explained:

WELCOME

[TYPE_ID_WELCOME , sessionId , protocolVersion, serverIdent]

This message allows connecting to the server. The session ID is randomly chosen by the server. The protocol version and identity of the server is also information related to the server.

PREFIX

[TYPE_ID_PREFIX , prefix , URI]

The prefix can be any valid JSON string and will substitute the URI from the right moment the message is send.

CALL

[TYPE_ID_CALL , callID , procURI , ...]

The call message has to include a call ID so that the client knows in the reception which one was the response to which message just in case several were sent. The procedure URI is the one assigned previously to the function that is wanted to call, the rest can be a list of the call argument.

CALLRESULT

[TYPE_ID_CALLRESULT , callID , result]

This call ID has to be the same as in the CALL in order to ensure that is the response to that message and not any other. Obviously the result has to be returned and it is at the end of the array.

CALLERROR

[TYPE_ID_CALLERROR , callID , errorURI , errorDesc]

More or less like the last one but the result data is now an error URI identifying the error and a string errorDesc describing the problem.

SUBSCRIBE & UNSUBSCRIBE

[TYPE_ID_SUBSCRIBE , topicURI]

[TYPE_ID_UNSUBSCRIBE , topicURI]

The Subscription and unsubscription messages are very similar. The idea is to send the URI from a topic to un/subscribe.

PUBLISH & EVENT

[TYPE_ID_PUBLISH , topicURI , event]

[TYPE_ID_EVENT , topicURI , event]

Every publish or event message has to belong to a topic represented by an URI and then the event is any valid JSON format value. The only difference between this two messages is what send it and what part of the communication is receiving it.

WAMP is used in the project because the messages are transmitted by WebSocket so the payload still low, the possible messages are already defined in a very comfortable way and the data structure is JSON which allows data to be easily parsed by a browser and since is a well-known data structure is not a problem to parse it in other languages.

AutobahnJS

AutobahnJS [7] is a JavaScript client library that implements The WebSocket Application Messaging Protocol (WAMP).

The main features of AutobahnJS are the functions it has for sending and receiving WAMP messages without even having to work with WebSocket. This abstraction layer and the event handlers for subscriptions make this library perfect for implementing WAMP in browsers.

JSON

JSON [8] is a format for data interchange between platforms. It aims to be lighter than XML and in the case of JavaScript is also easier to parse. This is normal because JSON is the JavaScript Object Notation.

JSON is very easy to understand a probe of that is its official web page that explains quite well the structure of JSON in 350 words and five pictures.

The best way of explaining the structure is to use a table from its webpage [9] that resume all fairly well:

object:
{ }
{ members }
members:
pair
pair , members
pair:
string : value
array:
[ ]
[elements]
elements:
value
value , elements
value:
string
number
object
array
true
false
null

string:
""
" chars "
chars:
char
char chars
char:
any-Unicode-character-except-"-or-\-or-control-character
\"
**
\/
\b
\f
\n
\r
\t
\u
four-hex-digits

number:
int
int frac
int exp
int frac exp
int:
digit
digit1-9 digits
- digit
- digit1-9 digits
frac:
. digits
exp:
e digits
digits:
digit
digit digits
e:
e
e+
e-
E
E+
E-

Json-cpp

Json-cpp [10] is a library for C++ that allows parsing easily JSON from a source.

Websocketpp

It [11] is another library for C++ that handles WebSocket connections. The main file of the test Server is based on an example[2] of this library.

NETWORK INTERFACE

One of the important ideas of the SSR tool is that it should be able to use it easily over a network with as many platforms as possible. One or more users should be able to use the program at the same time connecting to the server, which is the one making the hard processing.

This network interface will use WAMP to communicate so any client that knows in advance the communications rules can easily use the application with its own interface. In this point the JSON messages used by WAMP and all WAMP messages which are used for the network interface will be explained. As WAMP messages are JSON arrays the payload of those messages should be also JSON.

It is possible to divide all messages in three main groups. The ones used to connect to the server, the ones that use the RPC method and finally the most used ones which work like Publish & Subscribe method. In the end WAMP[3] offer these two functionalities and manage all the connection oriented mechanisms on which WebSocket relies.

Connection

In order to connect to the server first the WebSocket connection should be established. To choose the WAMP subprotocol the client must send to the server this possibility in the Sec-WebSocket-Protocol section.

After all the WebSocket connection is done and the HTTP communication is upgraded with the possibility of WAMP the server send now only WAMP messages. All these are done by the client. If the user does not have the client locally can always download it with normal HTTP messages by trying to connect to the server with a HTTP connection. When the client is downloaded then it will connect and handle all communication from that moment.

WELCOME message

This message is used only once to connect to the server and with this message the server assign an ID to each connection. The message is always send from the server to each client that attempt to connect. An example of a WELCOME message used in this network interface is:

[0, "v59mbCHDXZ7WTyxB", 1, "Autobahn/0.5.1"]

[2] https://github.com/zaphoyd/websocketpp/tree/master/examples/echo_server
[3] More WAMP information in the WAMP point.

RPC Method

When all the connection processes are already made then the client ask for the complete scene the interface should show in a very first moment. This can be the JSON file loaded by the server if no one has change anything in the scene or an update.

In the test server this does not happen and the changes are not store but only in the client therefore it will always send the first JSON loaded file.

It has no sense to send the complete scene to all clients each time one of them asks for it. To solve that problem the RPC method is used. Like this only the asker will receive the message and depending on the moment it ask for it maybe a different one.

There are two kinds of messages one for the client and one for the server (petition and response).

CALL message

The CALL message in this network interface is only used to ask for the initial scene. But in the future can be used for other things and should be no problem to do this.

It has to be used at the beginning of the connection before any subscription. If the test server would be an actual server this message could be used as refreshment of all the scene, but for accomplishing this the server should keep track of the changes occurred and modify the JSON file at least when a client demand it.

This message is always send from the client to the server. An example of a CALL message in this network interface is:

[2, f651Xd1, "http://spatialaudio.net/ssr/start"]

As can be seen this CALL message, that with the exception of the callID, will always be the same since is the only possible call message and the procURI is always “http://spatialaudio.net/ssr/” start to ask for the scene.

CALL RESULT message

This is the response of a CALL message, so is a message that is only send by the server to a demander client.

As was already said the only CALL RESULT message in the project is, by now, the JSON file loaded by the server. The JSON file structure will be further explained in detail.

An example of a CALL RESULT message is:

[3, f651Xd1, _the full JSON file_]

As can be seen the callID is the same that is received by the server in the CALL message. The explanation of this has nothing to do with this project, is a WAMP specification which has more sense if we would be using more messages of this type.

JSON file

The JSON file is the file loaded by the server. To explain its structure first an auto explanatory JSON file and then the two valid JSON example files from the server are included on the annex at the end of the document. The purpose of adding them in the annex is only about its size. They are very long and to include them here could make the reading harder.

It is possible to make two differences between JSON files: The ones that have loudspeakers and the ones that do not have loudspeakers. This would be also the main difference (for the interface) between binaural rendering and the others. Of course to include loudspeakers in the JSON file from the point of view of a valid JSON is always optional.

And apart from the first environmental information added in the file the only thing more are the sources. They are included in a “sources” pair with a list of their “ID” that is at the same time the pair of their properties of each source.

While adding sources the some properties can be omitted and a default value will be taken for those properties. These default values are:

- "type":"spherical"
- "volume":0
- "mute":false
- "position":[0,0,0]
- "fixed":false
- "channel":0
- "port":""
- "name":""
- "filename":""

In the case of the loudspeakers the default values are:

- "position":[0,0,0]
- "orientation":0
- "subwoofer":false

There is no limit in the number of sources or loudspeakers.

Publish & Subscribe Method

The subscription system adopted is basically several separations of all the possible information flow that can be sent throw the network. So that a client can demand only the information it wills actually use and not all.

This concept is just the contrary from RPC since normally when the server response to the client it response to all the clients connected. This way every client can show an upgraded interface and interact between each other.

These separations are, by now, six but they can be merged or separated easily. They can be differentiated in two main groups.

The first three subscriptions are somehow obligatory so that the interface show something useful. These are the “sources”, “global” and “reference” subscriptions.

The last three subscriptions are separated only because they generate a great amount and continuous of information. It lays in the election of the user depending on the network is using either to receive it or not. These last three are the “masterlevel”, “sourcelevel” and “loudspeakerlevel” subscriptions.

With these differentiations of the type of information sent new interfaces that use only part of it can be easily implemented. Also allows identifying the incoming messages very early, before even parsing the JSON payload.

The separations are administrated as topics in form of a URI as the WAMP specification explain. This kind of implementing topic ensures that the topics names are not repeated.

To choose which topic to subscribe or now the SUBSCRIBE and UNSUBSCRIBE messages are used. In the beginning the client is not subscribe to any topic so should always subscribe in order to receive information.

To send and receive information the PUBLISH and EVENT messages are used. Be aware that each one of these last two messages has a topic also and to receive or send them the client should be subscribed at that topic first.

When a client has subscribed to a topic then the client is able to either wait for messages that are related to the topic or either sends them.

SUBSCRIBE message

For subscribing to a topic a SUBSCRIBE message it is needed to be sent only once by client. This kind of messages is sent from the client to the server only.

These are all the possible SUBSCRIBE messages that can be sent:

[5, "http://spatialaudio.net/ssr/sources"]

[5, "http://spatialaudio.net/ssr/global"]

[5, "http://spatialaudio.net/ssr/reference"]

[5, "http://spatialaudio.net/ssr/masterlevel"]

[5, "http://spatialaudio.net/ssr/sourcelevel"]

[5, "http://spatialaudio.net/ssr/loudspeakerlevel"]

As can be easily seen the topic are URI's. It is used *"http://spatialaudio.net/ssr/"* as the root and then the name from each topic.

UNSUBSCRIBE message

The UNSUBSCRIBE messages are obviously the messages that are sent when information about some topics are no wanted any more.

The possible messages are quite similar to the SUBSCRIPTION messages. The only thing that changes is the number that defines the message.

These are all the possible UNSUBSCRIBTION messages that can be sent:

[6, "http://spatialaudio.net/ssr/sources"]

[6, "http://spatialaudio.net/ssr/global"]

[6, "http://spatialaudio.net/ssr/reference"]

[6, "http://spatialaudio.net/ssr/masterlevel"]

[6, "http://spatialaudio.net/ssr/sourcelevel"]

[6, "http://spatialaudio.net/ssr/loudspeakerlevel"]

PUBLISH message

The PUBLISH messages are messages sent from the client to the server that upgrade the current information of the interface. The information that this messages contains is always changes made by the sender client in its own interface.

Each of these messages are related to a topic URI to which the client should have been subscribed before sending them.

The structure of the payload may change between the different type of information is sent. It will be explained for each kind of payload separated by subscriptions in the Topics point.

EVENT message

The EVENT messages are sent from the server to the client. Just as the PUBLISH messages they have a topic related and the client should be subscribed into it before receiving any.

They have just the same structure than the PUBLISH messages. The structure is also explained in the Topics point.

In the case of the test server even copies the payload. In other server this could be done in other ways such as unite the changes from different sources to one single message, or only send the last messages in an interval and cast aside the ones that are overwritten.

Theoretically, and like the test server work, no interface could change anything without receiving an EVENT message even if it is its own response to a PUBLISH message.

In this last issue the client should wait till the server send the information back but could be possible and have to be expected that the information never came back. For example: a source is moved in client 1 and client 2 delete it, the server receives both messages and internally move and then delete the source but at the end only the delete message is sent as is logical. The client 1 would then never have moved the source.

Topics

As already said these topics can be changed easily and this is only one way of doing it but still the way how the test server and the client function.

The six topics will be explained with examples that can be used either for PUBLISH or by EVENT messages. The examples have both numbers but to change it would not be any problem.

Sources

This topic uses the URI: *"http://spatialaudio.net/ssr/sources"* .

By subscribing to this URI topic is possible to follow the changes made in the sources as well as change the sources yourself.

The event inside the message is a set of this JSON structure where everything is optional:

{"ID_String":{

"type":"String", //For now the possible values {spherical, plane}

"volume":"Integer",

"mute":"Boolean",

"position":[x,y,z], //Array of 3 Integers

"fixed":Boolean,

"orientation":"Integer",

"channel":"Integer",

"port":"String",

"name":"String",

"filename":"String"

"change":"String" //Possible values {add/new, delete/erase}

},

{Another source}

}

While adding new sources the Default values will be the same as in the JSON file:

"type":"spherical",

"volume":0,

"mute":false,

"position":[0,0,0],

"fixed":false,

"channel":0,

"port":"",

"name":"",

"filename":""

Example of a possible PUBLISH message that add one source, delete other one and change the position of a third one:

[7, "http://spatialaudio.net/ssr/sources", {

"sjfU7":{"type":"plain","volume":-6,"change":"add","position":[50,50,0]},

"oIl67":{"position":[10,60,0]},

"58Hjs":{"change":"delete"}

}]

As can be seen, the order of pairs inside an element does not affect the result and merging them is never a problem.

At first it was decided not to add a "change" element for establishing new sources or deleting them but instead adding them every time the ID didn't exist already.

That was not a good idea because the server could confuse a message of new source with the messages from a client that at that moment don't know the source has been removed.

Global

This topic uses the URI: *"http://spatialaudio.net/ssr/global"* .

By subscribing to this URI topic is possible to change and be aware of the global parameters. Such as the situation of the sound if it is playing or not, processing, if someone has rewinded it or even to actually seek.

These following types of JSON can be sent through this topic.

{"play":Boolean}

{"processing":Boolean}

{"rewind":"Boolean"}

{"reset_tracker":"Boolean"}

{"seek":"Integer"}

An example of a rewind EVENT message:

[8,"http://spatialaudio.net/ssr/global", {"reset_tracker":true}]

The seek request is a problem because the duration of the sound-file is not send and even it that case maybe another subscription to the time of the reproduction will be better for synchronization.

Reference

This topic uses the URI: *"http://spatialaudio.net/ssr/reference"* .

By subscribing to this URI topic is possible to change and see the changes in the orientation and position of the reference.

Only two members are sent and as it was said before it is not a problem to change the order of them.

{

"refPosition":[x,y,z],

"orientation":"Integer"

}

An example of an EVENT message that change the position of the reference and then the orientation:

[8,"http://spatialaudio.net/ssr/reference",{

"refPosition":[58,98,0],

"orientation":49

}]

or

[7,"http://spatialaudio.net/ssr/reference", {"refPosition":[58,98,0]}]

[7,"http://spatialaudio.net/ssr/reference", {"orientation":49}]

Level Topics

These topics are: “masterlevel”, “sourcelevel” and “loudspeakerlevel” topics.

They use the URI’s:

“http://spatialaudio.net/ssr/masterlevel”,

“http://spatialaudio.net/ssr/sourcelevel” and

“http://spatialaudio.net/ssr/loudspeakerlevel”.

These three topics are grouped because they are the level subscriptions, only separated from the rest because they send too much messages to the clients. This way the client can choose to receive them or not.

They are sent as values inside the array of the message.

Several examples of masterlevel and sourcelevel topics:

[8,"http://spatialaudio.net/ssr/masterlevel", -3]

[8,"http://spatialaudio.net/ssr/sourcelevel", -6]

The exception is the loudspeakerlevel topic where the message should look like this:

[8,"http://spatialaudio.net/ssr/loudspeakerlevel",{

"ID_0":[54,8,5,0,2,2],

"ID_1":[5,58,46,1,1,0],

"ID_0":[1,2,5,4,7,54],

"ID_1":[4,7,8,9,5,1]

}]

All the values for each speaker are send for each source inside an array.

This group (level topics) could also contains the subscription of time by now theoretical.

Chronological Example

Just to keep everything clear chronologically the steps that are following in a normal connection are:

1. The server start running and binds in a particular IP and port, also load a JSON file.
2. The client or clients (interfaces) ask for the initial environment using a CALL WAMP message
3. The server response to that client with a CALL RESULT message and as result the message should have the whole loaded JSON file.
4. The client receives it and loads the scene.
5. Now the interface will wait for the user to change something in order to send the changes. The client can also receive messages from other clients which are subscribed in the same topics.

WireShark Captures

The WireShark's captures that are in the CD shows how the real connection takes place. This is a capture of the TCP flow between a client and the server. It can be seen how first the WebSocket connects and then the WAMP messages are send through it.

Stream Content

```
GET / HTTP/1.1
Host: 139.30.207.120:9000
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:19.0) Gecko/20100101 Firefox/19.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: es-ES,es;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
Sec-WebSocket-Version: 13
Origin: http://139.30.207.120:9000
Sec-WebSocket-Protocol: wamp
Sec-WebSocket-Key: IJxIHjtR0hAaeTJ0gI6+cg==
DNT: 1
Connection: keep-alive, Upgrade
Pragma: no-cache
Cache-Control: no-cache
Upgrade: websocket

HTTP/1.1 101 Switching Protocols
Connection: Upgrade
Sec-WebSocket-Accept: fvlm2b/dANbvQluIafcT9IWjX2I=
Sec-WebSocket-Protocol: wamp
Server: WebSocket++/0.2.2dev
Upgrade: websocket

.,[0, "6FGHJQRUVZabjnpz", 1, "Autobahn/0.5.1"]..p...
+...@.......C...........J.............................~.>[3,"0.7dgzjf3337b",
{"duration":62,"level":"String","loudspeakers":
[{"id":"IDLS1","orientation":1.570790,"position":[0,200,0],"subwoffer":false},
{"id":"IDLS2","orientation":-1.570790,"position":[0,-200,0],"subwoffer":false},
{"id":"IDLS3","orientation":0,"position":[200,0,0],"subwoffer":false},
{"id":"IDLS4","orientation":3.141590,"position":[-200,0,0],"subwoffer":false}
],"play":true,"referenceOrientation":0,"referencePosition":[300,200,0],"sources":{"ID_0":
{"change":"new","channel":9,"filename":"name of the playing
file2","fixed":false,"mute":false,"name":"souce name
string","orientation":5,"port":"String","position":
[50,50,0],"type":"spherical","volume":"Integer dB"},"ID_1":
{"change":"add","filename":"name of the playing file","mute":true,"name":"souce name
string2","position":[500,200,0],"type":"plane","volume":54}}}]
..4.r.o.^.\.....].D...U...P...Z...G...G...W...i..E|...I..-....S..5...$...!...+...6..."...
$.......Z.E.v.v...$.u.n.....;.z.5.p...m.(.l.<.l.4.{..
```

The next capture was done during seven sources were moving. There it can be seen how the client group all the sources that are moved together into one only one WAMP message. This Wireshark capture is also in the CD.

Stream Content


```
.~..[8,"http://spatialaudio.net/ssr/sources",{"ID_0":
{"orientation":1.000997243159081,"position":
[549.6559457778931,74.72754192352295,0]},"ID_1":
{"orientation":1.573823043048832,"position":
[1001.655945777893,223.7275419235229,0]},"sID10":
{"orientation":2.088662312031798,"position":
[820.6559457778930,506.7275419235229,0]},"sID2":
{"orientation":1.994039806075765,"position":
[655.6559457778930,372.7275419235229,0]},"sID4":
{"orientation":1.797373574022246,"position":
[1053.655945777894,390.7275419235229,0]},"sID6":
{"orientation":1.451994938794717,"position":
[1048.655945777894,134.7275419235230,0]},"sID8":
{"orientation":1.996384154210983,"position":[975.6559457778937,518.7275419235229,0]}}]
.~..[8,"http://spatialaudio.net/ssr/sources",{"ID_0":
{"orientation":1.000997243159081,"position":
[551.0100269317627,76.08162307739258,0]},"ID_1":
{"orientation":1.573823043048832,"position":
[1003.010026931763,225.0816230773926,0]},"sID10":
{"orientation":2.088662312031798,"position":
[822.0100269317626,508.0816230773926,0]},"sID2":
{"orientation":1.994039806075765,"position":
[657.0100269317626,374.0816230773926,0]},"sID4":
{"orientation":1.797373574022246,"position":
[1055.010026931763,392.0816230773926,0]},"sID6":
{"orientation":1.451994938794717,"position":
[1050.010026931763,136.0816230773926,0]},"sID8":
{"orientation":1.996384154210983,"position":[977.0100269317634,520.0816230773926,0]}}]
.~..[8,"http://spatialaudio.net/ssr/sources",{"ID_0":
{"orientation":1.000997243159081,"position":
[583.5079746246338,130.2448692321777,0]},"ID_1":
{"orientation":1.573823043048832,"position":
```


SERVER

The server, as was already explained, is a test server that can be used to probe the functionality of the graphical interface but do not give the real services of the SSR application. It is based on an example server used in the websocketpp library [12].

Besides the websocketpp is in charge of the WebSocket connections still the WAMP subprotocol has to be programed and its decisions.

Even though it is a good possible approach to an integrated server in the SSR still do not give some of the functionality it should. Do not take the real data from the renderer like the sound therefore the sound is not sent neither the levels. Another thing is not done is to save the changes that occur in the interface. So the server do not keep track on what is displayed in the interfaces but only retransmit the changes that the clients generate.

The SSR application it is written in C++ and works with libboost libraries to handle connectivity. These libboost libraries are also used by the websocketpp library used to manage the WebSocket connections. This coincidence was not casualty and was a reason to use websocketpp in the server so the integration in a future is easier.

Apart from the websocketpp and the jsoncpp library used to parse JSON messages no other library is needed by the program to be compiled.

This server consists of three files.

These three files can be divided into the main program and a class with its header. The name of the main program file is *"server.cpp"* and the class name is clients so "clients.cpp" and "clients.hpp" because is the class that handles the clients and organize them in the topics.

The main program is in charge of the WebSocket connection and it is based in an example from the websocketpp library. This library is going to handle all the connections and offer us the next event handlers[4]:

Handler	Trigger	What the server does here
validate(connection)	Accept or reject an incoming connection	Confirm the selection of the subprotocol WAMP
on_open(connection)	Successful new connection	Add the new connection to the list and send the WELCOME message
on_interrupt(connection)	Connection was manually interrupted	Remove the connection that has closed the browser or disconnected
on_close(connection)	Connection closed (after opening)	
http(connection)	Process an HTTP connection	Send the whole webpage for establishing WebSocket connection
on_message(connection, message)	Message received	"tosend" method is called here. Then the messages are sent back

There are more handlers in the documentation but this are the only ones are used.

The "Clients" class has these usable methods:

void add(websocketpp::server::connection_ptr con);

This method adds a new connection to the list of clients connected. It is called in the on_open handler, in the server.

bool addSubscription(websocketpp::server::connection_ptr con,

[4] http://www.zaphoyd.com/websocketpp/manual/reference/handler-list Maybe the documentation differs a bit but for now that's the latest official documentation

std::string URI);
This method returns a true when a subscription has been correctly added to a connection. False is returned if it is not possible.

bool subtract(websocketpp::server::connection_ptr con);

This method erases a connection from the list of clients. This happens normally because the browser is closed.

Bool tosend(std::string &inmsg,
std::vector<websocketpp::server::connection_ptr> &w);

The "tosend( )" method is which makes almost all the work once all the subscriptions and connections are establish.

This method has two parameters the message that was received from the server and a vector of connections with as first the connection that had send it. This way the class now knows which connection has send what message. They are passed by reference.

The method returns a Boolean which is false if the server do not need to send anything. For example a SUBSCRIBE or WELCOME message. But if the server needs to send back a message the method returns a true. In this case the method has already overwritten the vector of connections with all the ones which should receive the message that is also overwritten.

In resume the server pass the connection of the sender and the message and then if true send to all the connections passed back the message is also passed back without checking anything.

The real decisions, like what to be send and to who, are done by the parser. Depending on what message and from who has received it make the choice. The message is parsed with the jsoncpp library and those choses are just the same as already said in the Network Interface point.

CLIENT /GRAPHICAL INTERFACE

The client/interface is just a normal webpage that any modern browser should have no problem to show. The difference it is make between client and interface is only a conceptual separation from the function of each part of the webpage. Even though this deference in the code it is not so clear.

Client is referred to the code that makes the webpage connect to the server and handles everything related with the WAMP subprotocol.

Interface is referred to the part of the webpage that actually "draws" the GUI and interact with the user.

Implementation

The webpage has an html file called index.html were all the pure html and CSS code lays. Then there are two more folders.

Images folder contains the two images used by the graphical interface that by the way were taken from the original SSR.

The JS folder contains all the JavaScript code which is the core of the webpage. Inside this folder are eight JavaScript files.

Client

For the client can be said that three JavaScript files are making most of the work.

At first the communication.js file makes all the WebSocket connection to the server and the subscriptions petitions. All the messages received are handled here and send them to a parser. For this AutobahnJS library is used with part of its handlers and methods.

The parser.js file is called from communication and is the bridge to the interface. Here the messages are parsed and the information that the parser took from them is changed in the graphical interface.

At last the netranslator.js file is not fully finish because still the integration not ready. But is a translator for the data used in the interface and the data needed by the SSR. For example a translation from 100 pixels from the graphical interface to one meter from the SSR.

Graphical Interface

The usability of graphical interface will be in the manual further explained.

The implementation of the graphical interface is divided in five files. The bigger of those is the environment.js which is in charge of the actual visualization of the scene without any object. Also control the events such as the mouse or keyboard. The zooming and panning is also work for this file which calls all the objects that appear in the graphical interface. This objects are more or less separated in the rest of the files.

Control.js file is in charge of displaying the controls, rotating.js is the file for displaying the reference and the little ball, speaker.js is the one for the loudspeakers and sources.js for sources.

All is painted in canvas and each shape has its own method “contain(x, y)” which is useful for the handlers to know by iterating which shape is clicked. This is made like this so that each shape is aware of itself and making possible interaction without making a single file that takes care of everything.

To actually draw the shapes the HTML 5 canvas is used. This way of "painting" do not keep track on what is painted each time so is by JavaScript classes how the tracking is been done.

USER MANUAL

Getting Started

Compiling the code and the JSON file…

Server

To get the server compiled all these libraries are needed:

- websocketpp
- libboost_system
- libboost_date_time
- libboost_program_options
- libboost_thread
- libboost_regex

Once the server is compiled, two possible and optional arguments can be written in the command line to execute it.

The port in which the server binds is one of them. If no port is selected by default the 9000 port will be the one establish to make the WebSocket connection.

On the other hand the environment in a JSON file can be loaded. By default "enviroment.json" file will be loaded. The only requirements for this file are that the file name should end in ".json" and should be a valid JSON and also valid according to the defined network interface.

In the moment the server is running any client can start communicating with it. If we do not have an interface the browser can ask for it by going to the IP and port where the server binded.

As an example with the server executed in a computer with this IP: 100.1.2.3 would look like this:

./ssr_server 9080 myfile.json

Client/Interface

The requirements for the browser are to be compatible with WebSocket and Canvas at least.

Following the example above then a browser should browse to:

http://100.1.2.3:9080

The server then will offer the whole scene to the interface and everything it is needed to connect to the server using WebSocket. Since this moment all the information is sent through WebSocket and to keep using the interface the server cannot stop.

The first thing that the interface shows is the JSON file that has being send. From now on everything can be changed. The interface fits all the space that is in the browser and for now only if it is refresh this will change.

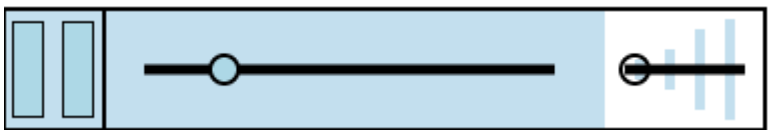
All the interfaces are communicated so the things they show should be the same in all. The moving parts can be moved by other interface, it is recommendable not to move the same thing at the same part because then strange movements will be seen even though nothing will crash.

Visible Parts

The visible parts are the shapes and all the representation such as menus the interface is able to draw.

Controls

The controls are located at the left down corner always, even with zooming or panning, they are fixed there. Had the Play/Pause button along with the seeking bar (not operative yet) and the volume. The Play/Pause button can be pressed to send the message only if the client has subscribe to the global subscription as well as the seeking bar or the volume that can also be changed.

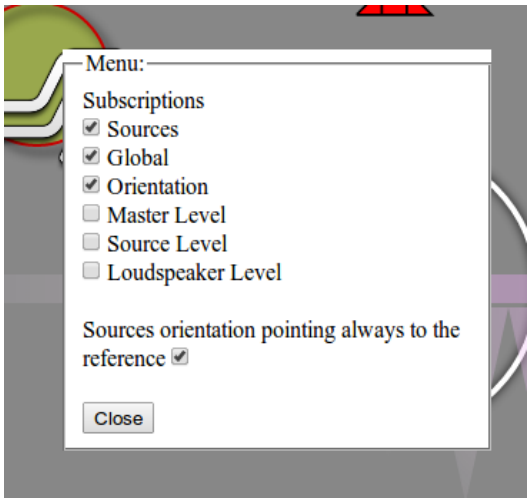


Logo

The logo of the SSR is at the top right corner and in order to make it usable, the menu button is there placed.

Menu

Appears only when the logo is clicked and it still let to see and interact with the interface below. The menu allows changing the subscriptions and the "Sources orientation pointing always to the reference" option.

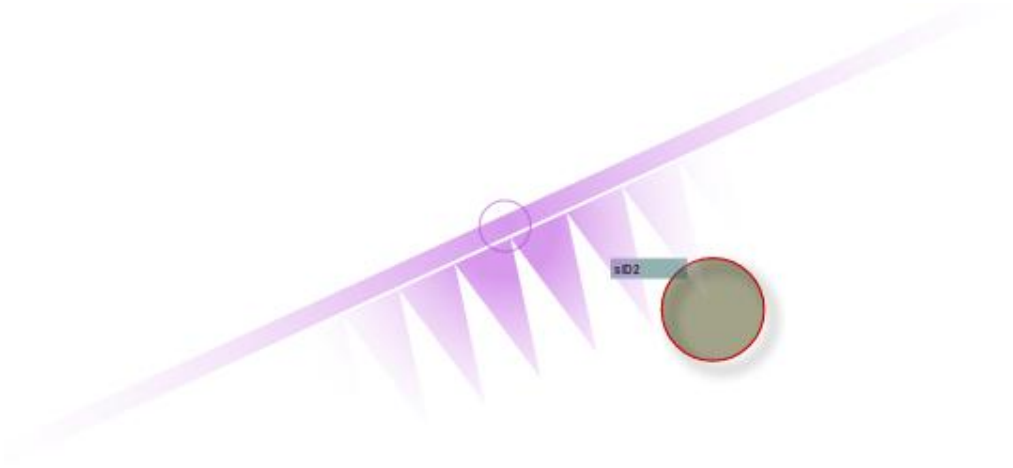


Sources

In the form of circles the sources are a movable part of the interface. They can be separated in two different types: the plane and spherical wave sources.

The type of the wave of the sources can change during the execution of the tool.

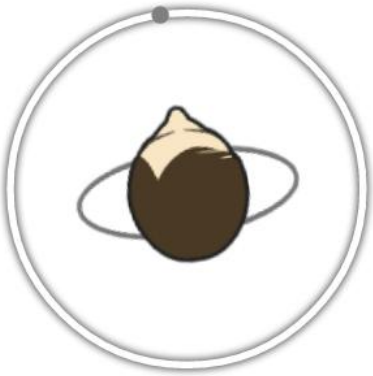
The plane waves have the particularity of being a smaller circle but same clickable area and a representation of where is the wave generated. Even is not infinite gives the idea.

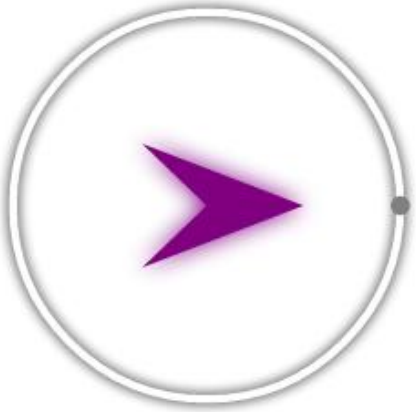


Reference

The reference is the other movable part of the interface, it can also rotate. Depending on the configuration the shape of the reference can be a head when is

binaural rendering (without loudspeakers) or some sort of arrow. All the spherical shapes point to it if it is desired in the menu.





Loudspeakers

The loudspeakers should emulate the real configuration and are always moved or rotated as the reference does. By them own are not clickable and are shown like a simple loudspeaker in red and black.

Online Features

The features everyone can try in the interface and will affect other interfaces. Every feature here need a subscription to work.

Play/Pause

It is located at the left part of the controls and visually can be a triangle or two rectangles depending on if it is playing or not.

To be able to do this the client should has subscribed to the global subscription.

Moving sources

The interface is designed so that multiple sources can be moved. To click on them is the way to select them and also panning can be done without deselecting them. To deselect the sources only, a click where nothing movable is, is needed.

Once the desired sources are selected click to one of them and drag it to move all the selected sources together.

To be able to do this the client should has subscribed to the sources subscription.

Changing the wave type from the source

When a source is selected a rectangle with the source name is display. Only by clicking it the wave type can be change.

To be able to do this the client should has subscribed to the sources subscription.

Muting sources

To change between mute and unmute the 'm' key should be pressed so that the selected sources change between both states.

To be able to do this the client should has subscribed to the sources subscription.

Adding sources

By double clicking somewhere a new source will be displayed in the place where the double click has occurred.

To be able to do this the client should has subscribed to the sources subscription.

Deleting sources

If the double click is done in an actual source, it will be deleted.

To be able to do this the client should has subscribed to the sources subscription.

Moving and rotating the reference

The reference can be moved along the environment just as the sources. But to rotate it a little ball is displayed in the circumference around it and by dragging it the reference can be rotated.

To be able to do this the client should has subscribed to the reference subscription.

Volume

The volume still not working because no sound is now sent or received

Offline Features

To be able to use these features the client do not need to subscribe. The changes made by these features do not change the behavior in other client or interfaces.

Panning

By clicking in an area where neither sources nor the reference is and moving the cursor without un-pressing the mouse, all the space will move together with the cursor. To the exception of the controls and the logo which will always stay still.

Zooming

For zooming the wheel mouse is needed and moving it up or down will zoom or un-zoom **the scene**. The reference to do the zooming will be always the cursor and as in the panning was said the controls and the logo will never be zoomed.

Menu

The menu is situated where the logo of the SSR is. Once in the menu the subscriptions can be checked or unchecked and the "Sources orientation pointing always to the reference" also can be chose.

This last option allows moving the spherical wave sources always looking to the reference or in the other hand keeps at every moment the same orientation.

The "Sources orientation pointing always to the reference" option is a little tricky if it is taken into account that several clients can be using different options. So an interface with this option unchecked can actually see how some sources are moving pointing to the reference and vice versa.

POSSIBLE IMROVMENTS

The server can be improved in three different ways. First of all the server could be integrated in the real program of SSR and work with real rendered data.

Another possible improvement is the possibility of sending some messages with a bit of delay in order to mix them up into one only message.

For the last improvement the first one should be finished, and is to keep track of all the changes that the clients send to it and when a new client connects offer it the updated scene JSON.

The client can have a lot more of functionality for example:

- SOLO feature.
- Adjusting the interface size to the browser size in real time.
- The way of adding sources

The last improvement to have really good version of a graphical interface would be obviously to allow playing the sound. Another subscription for transmitting time could be arranged and the sound can be send through another protocol more prepared for this issue.

ACRONIMS

XML: EXtensible Markup Language.

HTML 5: HyperText Markup Language fifth revision.

CSS3: Cascading Style Sheets.

SVG: Scalable Vector Graphics.

HTTP: Hypertext Transfer Protocol.

RPC: Remote Procedure Call.

WAMP: WebSocket Application Messaging Protocol.

JSON: JavaScript Object Notation.

REFERENCES

[1] HTML5 Draft, http://www.w3.org/html/wg/drafts/html/master/Overview.html.

[2] JavaScript rfc, http://www.apps.ietf.org/rfc/rfc4329.html.

[3] HTML 5 Canvas, http://www.w3.org/TR/2009/WD-html5-20090825/the-canvas-element.html.

[4] SVG, http://www.w3.org/Graphics/SVG/IG/resources/svgprimer.html.

[5] Peter Lubbers, Frank Greco, HTML5 Web Sockets, http://www.w3.org/html/wg/drafts/html/master/Overview.html.

[6] WAMP, http://wamp.ws/spec.

[7] AutobahnJS, http://autobahn.ws/js.

[8] JSON rfc, http://www.ietf.org/rfc/rfc4627.txt.

[9] JSON, http://json.org/.

[10] json-cpp, http://sourceforge.net/projects/jsoncpp/.

[11] websocketpp, https://github.com/zaphoyd/websocketpp/.

[12] example server, https://github.com/zaphoyd/websocketpp/tree/master/examples/echo_server.

[13 CSS, http://www.w3.org/Style/CSS/.
]

MORE REFERENCES

For everything related with web technology: www.w3schools.com/

WebSocket: http://www.websocket.org/quantum.html

Canvas: http://simonsarris.com/blog/140-canvas-moving-selectable-shapes

ANNEX

Auto explanatory JSON

{

"level":*<string>***,**

"play":*<true | false>***,**

"refPosition":[*<number>(meters, x)***,***<number>(meters, y)***,***<number>(meter ,z)***],**

"orientation":*<number>(angle)***,**

"duration":*<number>(seconds)***,**

"sources":{

"*<string>(ID from Source)***":{**

"type":*<string>(Spherical | plane)***,**

"volume":*<number>(dB)***,**

"mute":*<true | false>***,**

"position":[*<number>(meters, x)***,***<number>(meters, y)***,***<number>(meter ,z)***],**

"fixed":*<true | false>***,**

"orientation":*<number>(angle)***,**

"channel":*<number>***,**

"port":*<string>***,**

"name":*<string>***,**

"filename":*<string>***},**

"*<string>(ID from other Source)***":{**

...

}

},

"loudspeakers":[

"*<string>(ID from Loudspeaker)***":{**

"position":[*<number>(meters, x)***,***<number>(meters, y)***,***<number>(meter ,z)***],**

"orientation":*<number>(angle)***,**

"subwoofer":*<true | false>***,},**

"*<string>(ID from other Loudspeaker)*"**:{**

…

}

]}

JSON example without loudspeakers

{

"level":"String",

"play":false,

"referencePosition":[300,300,0],

"referenceOrientation":19,

"duration":62,

"sources":{

"ID_0":{

"type":"spherical",

"volume":"Integer dB",

"mute":false,

"position":[50,50,0],

"fixed":false,

"orientation":5,

"channel":9,

"port":"String",

"change": "new",

"name":"souce name string",

"filename":"name of the playing file2"}

```
        }
}
```

JSON example with loudspeakers

```
{
"level":"String",
"play":true,
"referencePosition":[300,200,0],
"referenceOrientation":0,
"duration":62,
"sources":{
        "ID_0":{
        "type":"spherical",
        "volume":"Integer dB",
        "mute":false,
        "position":[50,50,0],
        "fixed":false,
        "orientation":5,
        "channel":9,
        "port":"String",
        "change": "new",
        "name":"souce name string",
        "filename":"name of the playing file2"},
        "ID_1":{
        "type":"plane",
        "volume":54,
        "mute":true,
        "change":"add",
```

```
        "position":[500,200,0],
        "name":"souce name string2",
        "filename":"name of the playing file"}
        },
"loudspeakers":[
        {"id":"IDLS1",
        "position":[0,200,0],
        "orientation":1.57079,
        "subwoofer":false},
        {"id":"IDLS2",
        "position":[0,-200,0],
        "orientation":-1.57079,
        "subwoofer":false},
        {"id":"IDLS3",
        "position":[200,0,0],
        "orientation":0,
        "subwoofer":false},
        {"id":"IDLS4",
        "position":[-200,0,0],
        "orientation":3.14159,
        "subwoofer":false}
        ]
}
```