Sensor Server

Version:	1.0
Date:	March 22, 2020
Author:	Fred Dijkstra
Doc. id:	wg4ywkl

Document history

Version	Date	Author(s)	Description
1.0	March 22, 2020	Fred Dijkstra (fred@oryxmovementsolutions.nl)	Initial version

Table of contents

Introduction

This document describes the detailed design of the Sensor Server.

The Sensor Server implements the behavior and workflow of an application that controls the operation of a network of orientation sensors by interfacing to a BLE Handler that takes care of the physical interface to these sensors.

The workflow is controlled by instructions, or 'events' received on a GUI events interface.

The Sensor Server also takes care of recording the sensor data into a CSV file which can then be accessed by the GUI to offer it for downloading.

See the "System Architecture" for the context of the Sensor Server in the complete system.

Workflow

An overview of the main workflow as implemented by the Sensor Server is as follows:

1. When the program starts, the GUI and BLE functionality are started.
2. In the GUI the user will see a list of previous recordings (if there are any) that can be downloaded.
3. At one point the user will indicate in the GUI that scanning can be started.
4. The BLE functionality will be configured to start scanning.
5. Detected sensors are identified and sent to the GUI displaying the MAC address.
6. When all the sensors are discovered. The user indicates in the GUI that scanning can be stopped.
7. In the GUI the user will indicate which of the detected sensors need to be connected.
8. The BLE functionality is instructed to connect these sensors.
9. When a sensor connects, this is indicated to the GUI.
10. When all sensors are connected, the user can indicate to disconnect them all or indicate that the connected sensors need to be enabled and start measuring.
11. For each enabled sensor timestamped orientation will start coming in.
12. These orientations will be synchronized to the system time.
13. The user can indicate that a new recording needs to be started.
14. The recording is started and the orientations are stored in a CSV format in a file with the format "`yyyy-mm-dd-hh-mm-ss.csv`".
15. When the user indicates that the recording can be stopped, the file is closed and offered for download in the GUI.

Constants

To process and create the orientation object, the following packages are used by the Sensor Server.

```
var fs                  = require('fs');
var BleHandler          = require('./bleHandler');
var WebGuiHandler       = require('./webGuiHandler');
var FunctionalComponent = require( './functionalComponent' );
var events              = require('events');
```

For the Sensor Server the constants defined are listed in table 1.

Table 1: Constants of the Sensor Server.

Constant	Value	Description
`RECORDINGS_PATH`	`"/data/"`	Relative path where the recordings are stored.
`RECORDING_BUFFER_TIME`	`1000000`	Interval in ms at which data is written in blocks to the recording.

Constructor

To be able to implement the workflow, the properties of the Sensor Server are given in table 2.

Table 2: Properties of the Sensor Server.

Property	Type	Description
`sensors`	`{}`	Collection of sensors that were discovered. A specific sensor is referenced by the MAC address.
`discoveredSensors`	`[]`	The sensors that are discovered.
`connectedSensors`	`[]`	List of sensors that are connected.
`measuringSensors`	`[]`	List of sensors that are enabled and measuring.
`discoveredSensor`	`sensor`	The last discovered sensor.
`sensorList`	`[]`	A list of sensors for which an action needs to be performed (e.g. connecting, enabling).
`fileStream`	`Stream`	Handle to an opened file stream for writing.
`csvBuffer`	`String`	Buffer for CSV data to be able to reduce file access and write in large blocks.
`recordingStartime`	`Unsigned integer`	The system time in us at which the recording was started.
`lastWriteTime`	`Unsigned integer`	The system at which the last write was performed to the recording file.
`lastTimestamp`	`Unsigned integer`	The timestamp of the last orientation during recording.
`connectingSensor`	`sensor`	The sensor currently being connected.
`ble`	`BLE Handler`	Instantiation of the BLE Handler.
`gui`	`Web GUI Handler`	Instantiation of the Web GUI Handler.

To implement the workflow the Sensor Server is constructed as an instantiation Functional Component. This means that the behavior will be defined by a state machine.

Local functions

Overview

A number of local functions are implemented to manipulate these properties. The functions are given in table 3.

Table 3: Local functions.

Function	Arguments	Description
createSensorList	• sensors • addresses	Creates a sensor list from the supplied sensors collection with the given addresses.
removeSensor	• sensor • sensorList	Removes the sensor from the list.
startRecordingToFile	• filename	Creates the file for the recording on the file system.

Create sensor list

To return a specific list of sensors by supplying a collection of sensors and the list of addresses, the **`createSensorList()`** function is called.

```
function createSensorList( sensors, addresses )
{
    ...
}
```

The **`sensorList`** array that is to be returned is created first.

```
var sensorList = [];
```

The addresses argument is an array that can be evaluated and then a sensor can be added when it exists.

```
addresses.forEach( function(address)
{
    if( sensors[address] != undefined )
    {
        sensorList.push( sensors[address] );
    }
});
```

The created list can now be returned.

```
return sensorList;
```

Remove sensor

To move a sensor from a sensor list, the `removeSensor()` function can be called.

```
function removeSensor( sensor, sensorList )
{
    ...
}
```

The first step is to determine whether the sensor is in the list. If so, the sensor can be deleted from the list.

```
var idx = sensorList.indexOf( sensor );
if( idx != -1 )
{
    sensorList.splice( idx, 1 );
}
```

Start recording to file

To create and open a file for recording, the **`startRecordingToFile()`** is called, taking the component and the filename without the extension as the arguments.

```
function startRecordingToFile( component, name )
{
    ...
}
```

To create and open the file stream, the createWriteStream() method of the file system object (**`fs`**) is called.

```
component.fileStream = fs.createWriteStream( RECORDINGS_PATH + name + ".csv" );
```

The variables are set to manage the offset in the timing and the batch writing (see "Recording"). This is done by determining the current system time using the **`process.hrtime()`** JavaScript function which returns the number of seconds and nanoseconds. This is used to calculate the number of µs.

```
const hrTime = process.hrtime();
component.recordingStartime = hrTime[0] * 1000000 + hrTime[1] / 1000;
component.lastWriteTime = component.recordingStartTime;
```

Also the **`csvBuffer`** is reset.

```
component.csvBuffer = "";
```

When the stream is opened, the "*open*" event is generated by the file system. In the handler the event is converted into a "*fsOpen*" event.

```
component.fileStream.on( 'open', function()
{
    component.eventHandler( 'fsOpen' );
});
```

When the stream is closed, the "*close*" event is generated by the file system. In the handler the event is converted into a "*fsClose*" event.

```
component.fileStream.on( 'close', function()
{
    component.eventHandler( 'fsClose' );
});
```

Use cases

The implementation of the described workflow can be divided in a number of use cases which are described in this chapter.

Powering-on

When the Application starts it will wait for the BLE radio to be ready. This is indicated by the "*blePoweredOn*" event.

Scanning

When the user presses the button to start scanning, the "*startScanning*" GUI event is received. As a result, the scanning is started by calling the `ble.startScanning()` function. When the BLE radio started scanning, the "*bleScanningStarted*" event is received.

It is up to the user to stop scanning again. This will result in the "*stopScanning*" GUI event and as a result the `ble.stopScanning()` function will be called. When scanning stopped, the "*bleScanningStopped*" event is received which is forwarded as the "*scanningStopped*" GUI event.

Discovering

When scanning, the BLE radio will detect the sensors in the network. This results in a single "*bleSensorDiscovered*" event per discovered sensor. When the sensor is unknown, it is added to the `sensors` collection. To indicate the discovery to the GUI, the "*sensorDiscovered*" GUI event is sent.

Connecting

The user can choose to connect the sensors. This will result in the "*connectSensors*" GUI event containing a list of addresses of the sensors to connect. This list is stored in addresses. The process then starts by calling the `ble.connectSensor()` method for the first sensor in this list.

When a sensor connected, the "*bleSensorConnected*" event will be received. Note that the sensors are connected one-by-one, which seems to work better than trying to connect all sensors all at once; in that scenario sometimes a lot of disconnections can be experienced.

When the sensor is connected, it is moved from the `connectingSensors` list to the `connectedSensors` list by calling the `moveSensor()` function.

The connection is forwarded by sending the "*sensorConnected*" GUI event.

Although it is not expected to happen often, while connecting a sensor, another sensor could disconnect, resulting in the "*bleSensorDisconnected*" event. This is

handled by moving the sensor from the **connectedSensors** list back to the **connectingSensors** list and sending the "*sensorDisconnected*" GUI event.

When the **connectingSensors** list is empty, all the sensors are connected.

When the user wants to disconnect all connected sensors, the "*disconnectSensors*" GUI event is received. The **connectedSensors** list will then be used to call the **disconnectSensor()** method of the BLE Handler for all the sensors until all sensors are disconnected.

Measuring

When the user indicates that a number of sensors must start measuring, the "*startMeasuring*" GUI event is received containing the list of addresses. The corresponding sensors are placed in sensorList using the createSensorList() function.

To enable measuring of a connected sensor, the sensor must be configured by calling the **ble.enableSensor()**.

When this fails, an "*bleSensorerror*" message is received and the **ble.disconnectSensor()** for the sensor is called. Also in this use case sensors could disconnect, which could be the cause for the mentioned error. The disconnection will generate the "*bleSensorDisconnected*" event which will be handled by removing the **connectedSensors** list and the "*sensorDisconnected*" message is sent to the GUI.

For a sensor for which measuring was enabled, the "*bleSensorEnabled*" is received. This is handled by removing the sensor address from **addresses** and adding it to the **measuringSensors** list.

For a measuring sensor orientation data will start coming in via an "*bleSensorData*" event. This data is forwarded to the GUI in the "*sensorData*" GUI event.

When measuring needs to be stopped for all measuring sensors, the "*stopMeasuring*" GUI event is received, which will be handled by disabling all the sensors in **measuringSensors**. For each disabled sensor the "*sensorDisabled*" GUI event is sent.

Recording

When the user indicates that a recording is to be started, the "*guiStartRecording*" GUI event is received containing the filename. The file is created by calling the startRecordingWithFile() function. When new orientation data is received from a sensor, a string is created in the format:

```
<timestamp>,<sensor name>,<address>,<w>,<x>,<y>,<z>
```

This string is added as a new line to the **`cvsBuffer`**. This string is written to file in one write action when the difference between the timestamp and the **`lastWriteTime`** is larger than **`RECORDING_BUFFER_TIME`**.

The recording must be stopped when the "*stopRecording*" GUI event is received. This means that the current **`csvBuffer`** is written to file and that the stream can be closed.

State machine

To implement the described behavior, a state machine is defined for the application. This is done by defining the states, choice-points, state diagram and the state transitions table.

States

For the behavior of the application the states that can be identified are given in table 4.

Table 4: defined states for the state machine.

State name	Description
PoweringOn	The Noble object is initializing and the BLE radio is powered-on.
Idle	The BLE radio is powered-on but not active.
Scanning	The BLE radio is listening, i.e. scanning for advertising peripherals.
Connecting	The discovered sensors are connected one by one.
All connected	All sensors are connected.
Enabling	The process is started to start the measurements on a connected sensor.
Measuring	All connected sensors are measuring.
Disabling	The measurements on the connected sensors are stopped.
Disconnecting	The sensors are disconnecting.
Recording	The data of the connected and measuring sensors are being logged in file.
Closing	The buffered sensor data is written to file and when ready, the file is closed.
Disabling	The measurements are being disabled.

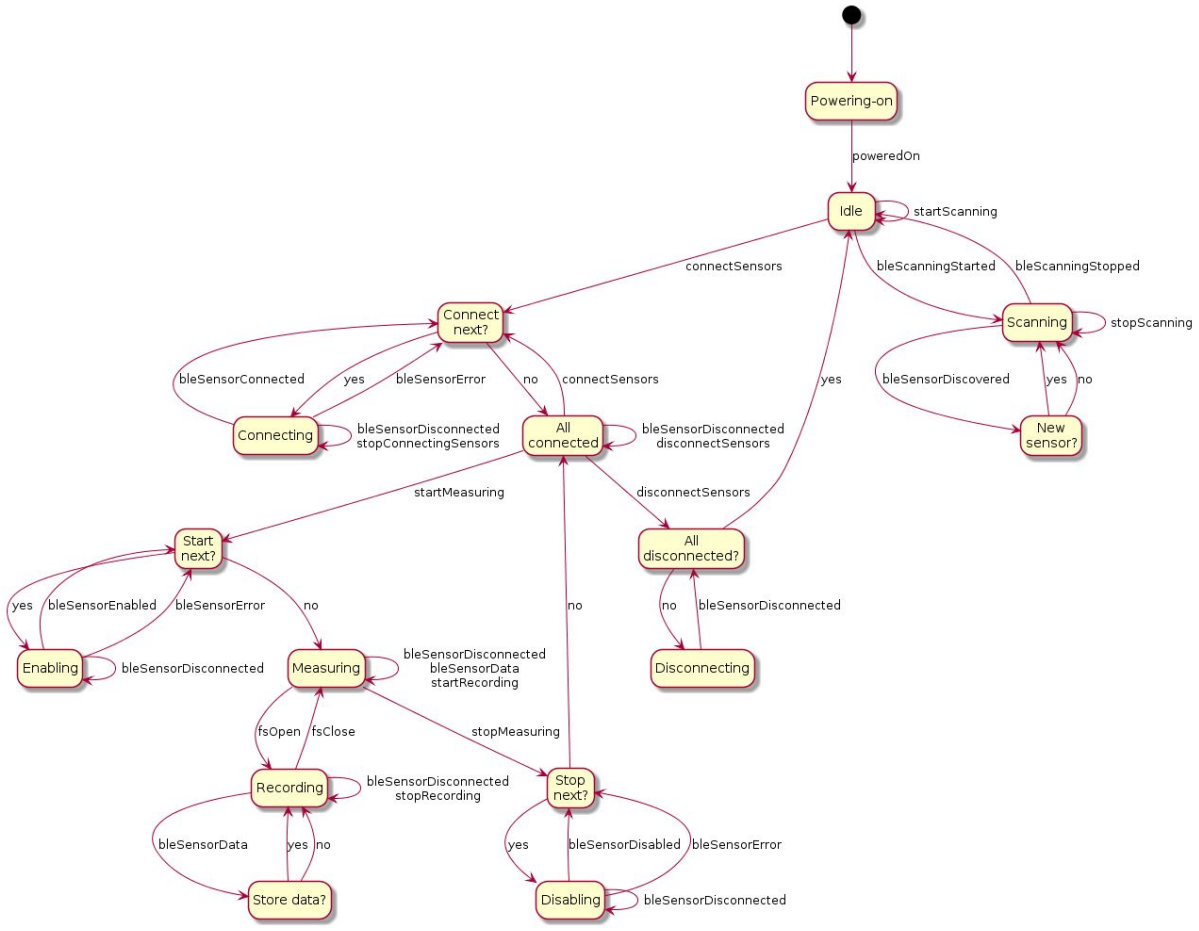
Choice-points

For the behavior of the application the choice-points that can be identified are given in table 5.

Table 5: defined choice-points for the state machine.

Choice-point	Evaluation
Connect next?	Are there any sensors that are not connected yet?
Start next?	Are there any connected sensors that are not measuring yet?
Store data?	Is the data in memory large enough to be written to file?
Stop next?	Are there any connected sensors that are still measuring?
All disconnected?	Are all sensors disconnected?
New sensor?	Is the discovered sensor unknown?

State diagram



State transitions

Use case: Powering-on

State / choice-point	Event	Actions	Next
Powering-on	*blePoweredOn*	• -	*Idle*
Idle	*startScanning*	• Clear **`discoveredSensors`**. • Call **`ble.startScanning()`**.	*Idle*
	bleScanningStarted	• Send "*scanningStarted*" GUI event.	*Scanning*
	connectSensors	• Create **`sensorList`** from **`addresses`**.	*Connect next?*

Use case: Scanning

State / choice-point	Event	Actions	Next
Scanning	*bleSensorDiscovered*	• Set **`discoveredSensor`**.	*New sensor?*
	stopScanning	• Call **`ble.stopScanning()`**.	*Scanning*
	bleScanningStopped	• Send "*scanningStopped*" GUI event.	*Idle*

Use case: Discovering

State / choice-point	Event	Actions	Next
New sensor?	*yes*	• Add sensor to **`sensors`** if not already known. • Add sensor to **`discoveredSensors`**. • Send "*sensorDiscovered*" GUI event.	*Scanning*
	no	• -	*Scanning*

Use case: Connecting

State / choice-point	Event	Actions	Next
Connect next?	*yes*	• Call **`ble.connectSensor()`** for the first sensor in **`sensorList`**.	*Connecting*
	no	• Send "*allSensorsConnected*" GUI event.	*All connected*
Connecting	*bleSensorDisconnected*	• Remove sensor from **`connectedSensors`**. • Add sensor to **`sensorList`**. • Send the "*sensorDisconnected*" GUI event.	*Connecting*
	stopConnectingSensors	• Clear **`sensorList`**.	*Connect next?*
	bleSensorConnected	• Remove the first element from **`sensorList`**. • Add sensor to **`connectedSensors`**. • Send the "*sensorConnected*" GUI event.	*Connect next?*
	bleSensorError	• Log the error to console.	*Connect next?*
All connected	*disconnectSensors*	• Create **`sensorList`** from **`addresses`**.	*All disconnected?*
	startMeasuring	• Create **`sensorList`** from **`addresses`**.	*Start next?*
	bleSensorDisconnected	• Remove sensor from **`connectedSensors`**. • Send the "*sensorDisconnected*" GUI event.	*All connected*
	connectSensors	• Create **`sensorList`** from **`addresses`**.	*Connect next?*
All disconnected?	*yes*	• Send "*allSensorsDisconnected*" GUI event.	*Idle*
	no	• Call **`ble.disconnectSensor()`** for the first sensor in **`sensorList`**. • Remove the first element from **`sensorList`**.	*Disconnecting*
Disconnecting	*bleSensorDisconnected*	• Remove sensor from **`connectedSensors`**. • Remove sensor from **`sensorList`** if present. • Send a "*sensorDisconnected*" GUI event.	*All disconnected?*

Use case: Measuring

State / choice-point	Event	Actions	Next
Start next?	*yes*	• Call **`ble.enableSensor()`** for the first sensor in **`sensorList`**. • Remove the first element from **`sensorList`**.	*Enabling*
	no	• Send "*allSensorsEnabled*" GUI event.	*Measuring*
Enabling	*bleSensorEnabled*	• Add sensor to **`measuringSensors`**. • Send a "*sensorEnabled*" GUI event.	*Start next?*
	bleSensorDisconnected	• Remove sensor from **`sensorList`** if present. • Remove sensor from **`connectedSensors`**. • Remove sensor from **`measuringSensors`** if present. • Send a "*sensorDisconnected*" GUI event.	*Enabling*
	bleSensorData	• -	*Enabling*
	bleSensorError	• Remove sensor from **`sensorList`**. • Call **`ble.sensorDisconnect()`** for the sensor.	*Start next?*
Measuring	*stopMeasuring*	• -	*Stop next?*
	startMeasuring	• -	*Start next?*
	bleSensorDisconnected	• Remove sensor from **`connectedSensors`**. • Remove sensor from **`measuringSensors`**. • Send a "*sensorDisconnected*" GUI event.	*Measuring*
	bleSensorData	• Send a "*sensorOrientation*" GUI event.	*Measuring*
	startRecording	• Call `startRecordingToFile()` with the received filename.	*Measuring*
	fsOpen	• Send the "*recordingStarted*" GUI event. • Write header to file.	*Recording*
Stop next?	*yes*	• Call **`ble.disableSensor()`** for the first sensor in **`measuringSensors`**. • Remove the first sensor from **`measuringSensors`**.	*Disabling*
	no	• Send "*allSensorsDisabled*" GUI event.	*All connected*
Disabling	*bleSensorDisabled*	• Send a "*sensorDisabled*" GUI event.	*Stop next?*
	bleSensorError	• Call **`ble.sensorDisconnect()`** for the sensor.	*Stop next?*
	bleSensorData	• -	*Disabling*
	bleSensorDisconnected	• Remove sensor from **`connectedSensors`**. • Remove sensor from **`measuringSensors`** if present. • Send a "*sensorDisconnected*" GUI event.	*Disabling*

Use case: Recording

State / choice-point	Event	Actions	Next
Recording	*bleSensorDisconnected*	● Remove sensor from **`connectedSensors`**. ● Remove sensor from **`measuringSensors`**. ● Send a "*sensorDisconnected*" GUI event.	*Recording*
	bleSensorData	● Set **`lastTimestamp`**. ● Add sensor orientation as CSV to **`csvBuffer`**.	*Store data?*
	stopRecording	● Write **`csvBuffer`** to file. ● Close file.	*Recording*
	fsClose	● Send the "*recordingStopped*" GUI event.	*Measuring*
Store data?	*yes*	● Write **`csvBuffer`** to file. ● Set **`lastWriteTime`** to **`lastTimestamp`**.	*Recording*
	no	● -	*Recording*