UNIVERSITY OF SOUTHERN DENMARK

Distributed systems controlling Spheros via bluetooth

Report Author:
Anders Kristensen

Course:
Project in real time systems
01.02.2018

Contents

1 Introduction

In this project a distributed system for controlling multiple Spheros around a curse have to be build. The setup can be seen on figure 1. The system consists of a rectangular track with a camera above. On the track there are multible agents/Spheros rolling around. The camera have to track each Sphero and inform the distributed system with the location of them.

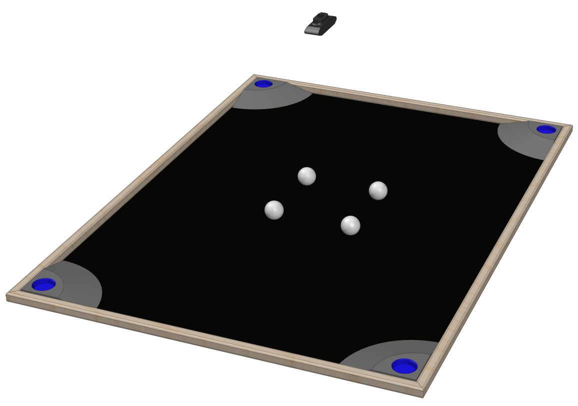


Figure 1: Layout of the track with camera

The distributed system, consisting of 2 Spheros controlled by 2 Raspberry pi's and a Pc acting as a "customer" has to act like a e.g warehouse facility. This means that each Sphero acts like a transport robot that receives an order from the customer to go to three different locations in a sequential order to pick something up. This simulates e.g that a transport robot has to bring one part from one machine to another machine in a sequential order in a production line. The Raspberry's each run the java based JADE middle ware which is a middleware that is used for multi-agent systems complying with the FIPA protocol.
Running the JADE framework makes it possible to create multiple agents communicating with each other using the FIPA protocol. In this particular system there are the following agents:

- 2 Sphero agents - The transport robots
- 3 Machine agents - Acting like physical machines in a warehouse/production facility
- 0 to Many customers

The customer agents are requesting the Sphero-agents to go to each machine in a specific order. If a Sphero-agent is free it agrees to the customers request and starts talking to the machine-agents to find out when they can be served. A Machine-agent can only serve one Sphero-agent at a time and to serve a Sphero-agent the Sphero-agent has to go to a specific place on the track. Each Sphero-agent has its own control algorithm to control its movement. It is being provided by its coordinate on the track by the vision system which streams the information to the network by UDP broadcast.

2 Sphero communication

To be able to control the Spheros from the Raspberry some form of API or driver is needed to be developed. There is an opensource API available written for older Spheros not using BLE. The group was only able to find an API written in Node.js.[4]. As to begin with the aim was to develop a Sphero driver in C or C++ for speed and robustness. Therefore this API could not be used as it was. However it was possible to find an API implemented for the older Spheros written in C++ [2]. This API was selected to be modified for using with BLE and changed so that it complied with the SPRK+'s protocol which are the Spheros used for this project.

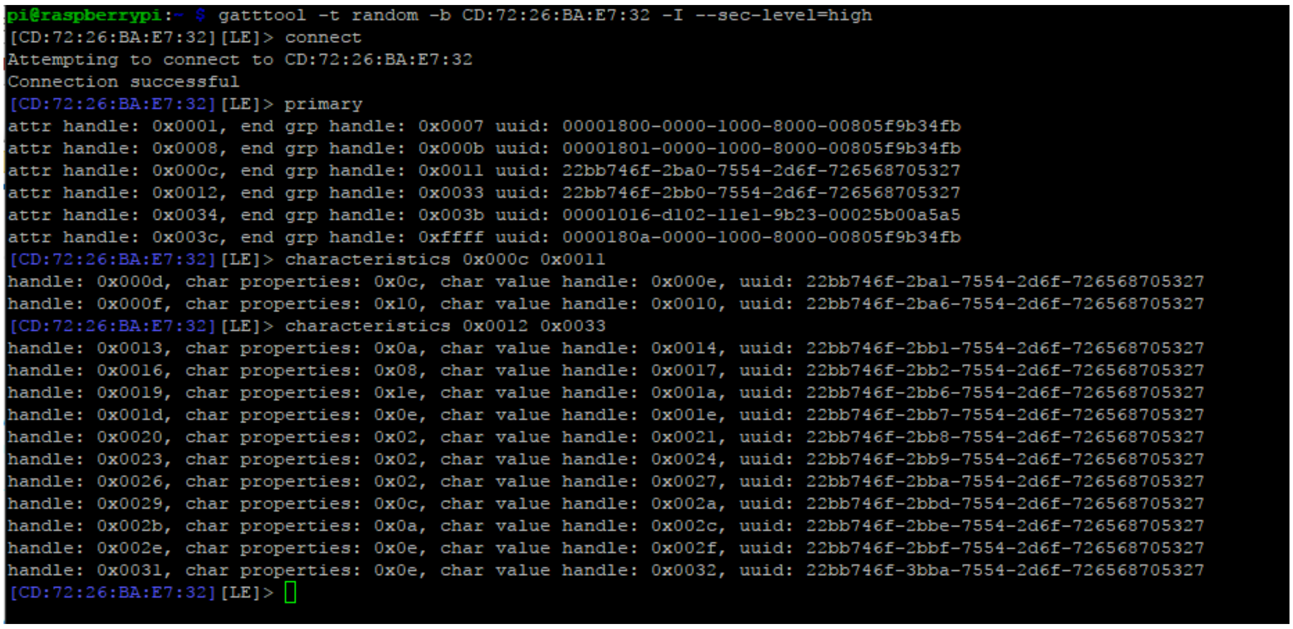
The Spheros are using bluetooth low energy (BLE) to communicate with the Raspberry pi´s. BLE uses "The Generic Attribute Profile" (GATT) to specify how data are transferred between devices [3]. GATT specifies that BLE units must provide Services which contains characteristics. A Service is a collection of data associated with a feature. As an example the Spheros provide a RobotControlService with different characteristics. These services and characteristics are streamed out from the devices with Universally Unique ID´s (UUID) used to find services from other BLE devices.

The Spheros used for this project is of the type SPRK+ which uses BLE. The problem is that it is a bit of a closed system. Not much information about their bluetooth communication is officially published. However, they do have an API reference for how the commands to make the Sphero move, light up etc is put together [1]. Therefore it was necessary to find out which characteristics to write the specific commands to.

To do this the previusly mentioned Node.js API source code was investigated. Here it was possible to find the following lines of code showing the UUIDs used in that API.

```
Adaptor.BLEService = "22bb746f2bb075542d6f726568705327";
Adaptor.WakeCharacteristic = "22bb746f2bbf75542d6f726568705327";
Adaptor.TXPowerCharacteristic = "22bb746f2bb275542d6f726568705327";
Adaptor.AntiDosCharacteristic = "22bb746f2bbd75542d6f726568705327";
Adaptor.RobotControlService = "22bb746f2ba075542d6f726568705327";
Adaptor.CommandsCharacteristic = "22bb746f2ba175542d6f726568705327";
Adaptor.ResponseCharacteristic = "22bb746f2ba675542d6f726568705327";
```

This information could be used in combination with the "gatttool" in linux. By connecting the Raspberry to the SPRK+ through commandline and running the gatttool it was possible to find out which handles belonged to which characteristics. Se figure 2.



```
pi@raspberrypi:~ $ gatttool -t random -b CD:72:26:BA:E7:32 -I --sec-level=high
[CD:72:26:BA:E7:32][LE]> connect
Attempting to connect to CD:72:26:BA:E7:32
Connection successful
[CD:72:26:BA:E7:32][LE]> primary
attr handle: 0x0001, end grp handle: 0x0007 uuid: 00001800-0000-1000-8000-00805f9b34fb
attr handle: 0x0008, end grp handle: 0x000b uuid: 00001801-0000-1000-8000-00805f9b34fb
attr handle: 0x000c, end grp handle: 0x0011 uuid: 22bb746f-2ba0-7554-2d6f-726568705327
attr handle: 0x0012, end grp handle: 0x0033 uuid: 22bb746f-2bb0-7554-2d6f-726568705327
attr handle: 0x0034, end grp handle: 0x003b uuid: 00001016-d102-11e1-9b23-00025b00a5a5
attr handle: 0x003c, end grp handle: 0xffff uuid: 0000180a-0000-1000-8000-00805f9b34fb
[CD:72:26:BA:E7:32][LE]> characteristics 0x000c 0x0011
handle: 0x000d, char properties: 0x0c, char value handle: 0x000e, uuid: 22bb746f-2ba1-7554-2d6f-726568705327
handle: 0x000f, char properties: 0x10, char value handle: 0x0010, uuid: 22bb746f-2ba6-7554-2d6f-726568705327
[CD:72:26:BA:E7:32][LE]> characteristics 0x0012 0x0033
handle: 0x0013, char properties: 0x0a, char value handle: 0x0014, uuid: 22bb746f-2bb1-7554-2d6f-726568705327
handle: 0x0016, char properties: 0x08, char value handle: 0x0017, uuid: 22bb746f-2bb2-7554-2d6f-726568705327
handle: 0x0019, char properties: 0x1e, char value handle: 0x001a, uuid: 22bb746f-2bb6-7554-2d6f-726568705327
handle: 0x001d, char properties: 0x0e, char value handle: 0x001e, uuid: 22bb746f-2bb7-7554-2d6f-726568705327
handle: 0x0020, char properties: 0x02, char value handle: 0x0021, uuid: 22bb746f-2bb8-7554-2d6f-726568705327
handle: 0x0023, char properties: 0x02, char value handle: 0x0024, uuid: 22bb746f-2bb9-7554-2d6f-726568705327
handle: 0x0026, char properties: 0x02, char value handle: 0x0027, uuid: 22bb746f-2bba-7554-2d6f-726568705327
handle: 0x0029, char properties: 0x0c, char value handle: 0x002a, uuid: 22bb746f-2bbd-7554-2d6f-726568705327
handle: 0x002b, char properties: 0x0a, char value handle: 0x002c, uuid: 22bb746f-2bbe-7554-2d6f-726568705327
handle: 0x002e, char properties: 0x0e, char value handle: 0x002f, uuid: 22bb746f-2bbf-7554-2d6f-726568705327
handle: 0x0031, char properties: 0x0e, char value handle: 0x0032, uuid: 22bb746f-3bba-7554-2d6f-726568705327
[CD:72:26:BA:E7:32][LE]>
```

Figure 2: Using gattool to find handles for characteristics on the Spheros

By comparing the UUID's found with the gattool with the ones declared in the Sphero.js it was possible to define the following code in the API. This is going to be used when writing to the bluetooth sockets later on.

```
namespace handle
{
        // RobotControlService  handle 0x000c to 0x0011
        static uint8_t const CMD = 0x0e; //CommandsCharacteristic 0x000e
        static uint8_t const RESP = 0x10; //ResponseCharacteristic 0x0010

        // BLEService  handle: 0x0012 to 0x0033
        static uint8_t const WAKE = 0x2f; //WakeCharacteristic 0x002f
        static uint8_t const TX = 0x14; //TXPowerCharacteristic 0x0014
        static uint8_t const ADOS = 0x2a; //AntiDosCharacteristic 0x002a
}
```

To find out how to put the handles correct in the command sent to the Sphero an example program from the Node.js API was tried executed while running "HCIdump" on the commandline. Doing this it was possible to see exactly how the bytes were arranged on the Hardware control interface" (HCI) and thereby making it possible to replicate it. Another important thing showing up by doing this, is that Sphero SPRK+ needs to be woken up and put in a developer mode in order to respond to commands. Therefore, the C++ API was extended with a wake method:

```
int Sphero::wake()
{
        char message1[8] = { ATT::WRT, handle::ADOS, 0x00, 0x30, 0x31, 0x31 ,0x69 ,0x33 };
        if (write(_bt_socket, message1, sizeof(message1)) < 0)
        {
                return -1;
        }
        char message2[10] =
        { ATT::WRT, handle::TX, 0x00, 0x28, 0x44, 0x00, 0x00, 0x20, 0x00, 0x02 };
        if (write(_bt_socket, message2, sizeof(message2)) < 0)
        {
                return -1;
        }
        char message3[4] = { ATT::WRT, handle::WAKE, 0x00, 0x00 };
        if (write(_bt_socket, message3, sizeof(message3)) < 0)
        {
                return -1;
        }
        return 0;
}
}
```

After this it was just a matter of rewriting the API to assemble and parse commands according to the SPRK+ API. [1] Furthermore a new bluez adapter was written to be able to send and recieve over BLE. The complete collection of code can be found on:
`https://github.com/miaar7/Sphero`

3 Verifying the API

In order for the people working on the control algorithm positioning the Spheros on the track, some quantification of the timing between a Sphero and a Raspberry had to be investigated.

The first test was to see how stable Sphero was, streaming sensor values back to the Raspberry. Therefore, a small program utilizing the new API was written. The API is created in a way that makes it possible to register a function to be called each time the Sphero object receives a valid message from the Sphero robot. In that way it is possible to log the received sensor value to a file together with a time stamp. From that it was possible to draw some histograms showing the distribution of time difference between each message. See figure 3. Each test was done telling the SPRK+ to stream either one (blue) or four (orange) sensor values at a certain frequency. It is clear streaming at 50Hz gives the best distribution. Here, most of the messages are received with 20ms interval corresponding to 50Hz.

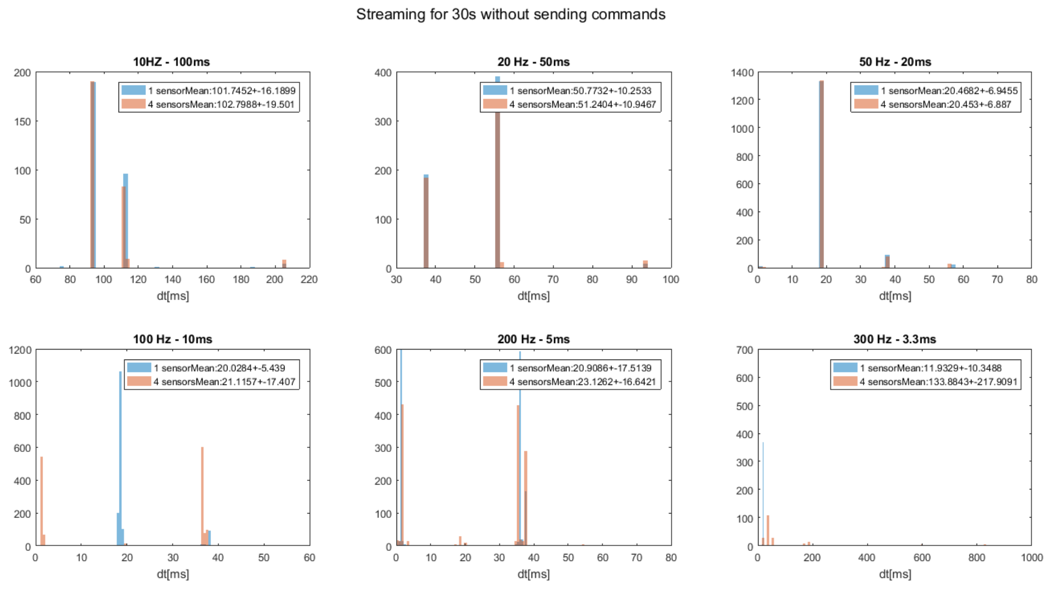


Figure 3: Distribution of time between messages at different stream frequencies

To investigate if it influenced the streaming of sensors when sending data to the SPRK+, two experiments was made. The first messages were sent to the SPRK+ at random time intervals. See figure 4. Here, it is clear that the time interval between each received message starts to spread out. This could be due to some of the messages colliding or the fact that the SPRK+ cannot handle sending and receiving messages close by.

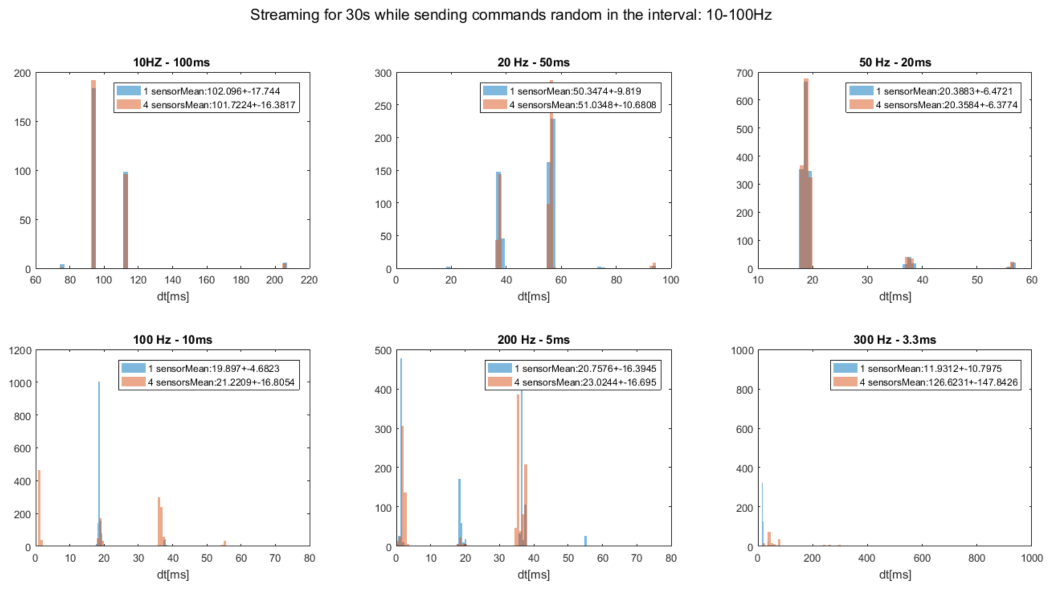


Figure 4: Distribution of time between messages at different stream frequencies. Sending messages to SPRK+ with random time intervals

The second test was to send a command each time the SPRK+ receives a message. In that way there should in principle not be any colliding packages and the SPRK+ do not have to parse incoming commands while streaming a sensor value. The result of this can be seen on figure 5. Here it is seen that most of the messages are received with a constant interval when streaming and thereby sending with a frequency of 50Hz

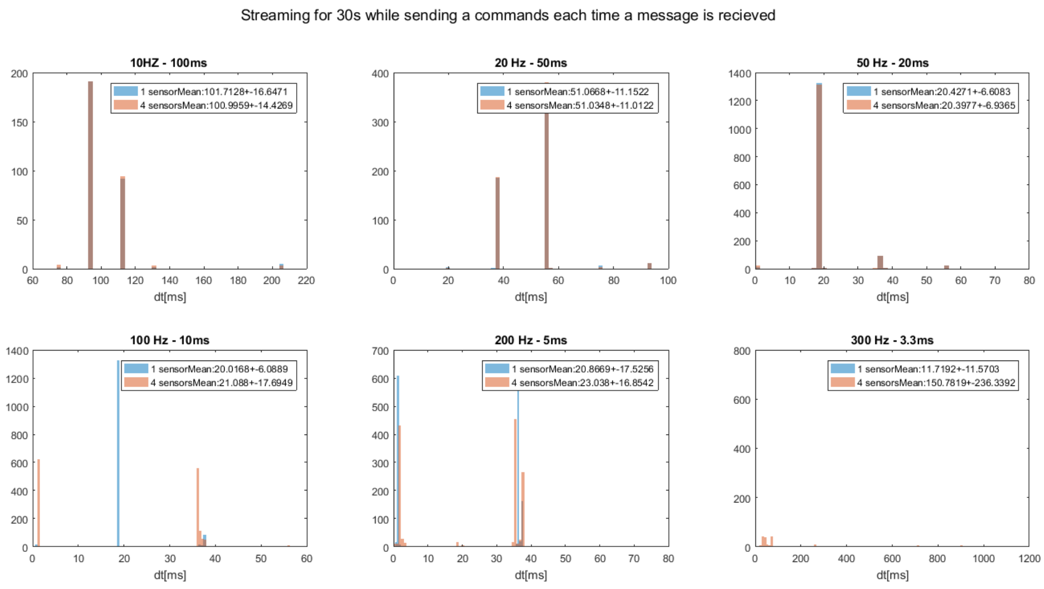


Figure 5: Distribution of time between messages at different stream frequencies. Sending messages to SPRK+ synchronously

A last test was made to see what the round trip delay was. By the round trip delay it is meant the time it takes from sending a command to the SPRK+ telling it to make a change on the hardware and until the sensor value measuring this hardware change has been streamed back to the Raspberry. For doing this, a sine wave was send to be put on the motor output while the SPRK+ was told to stream the real PWM output for that motor. The result of this can be seen on figure 6. The phase shift between the send sine wave and the received is approximately 100[ms]. This means that it takes a maximum of 100[ms] from sending a command to the command has been parsed and executed on the SPRK+ until the change is read by the Raspberry again.

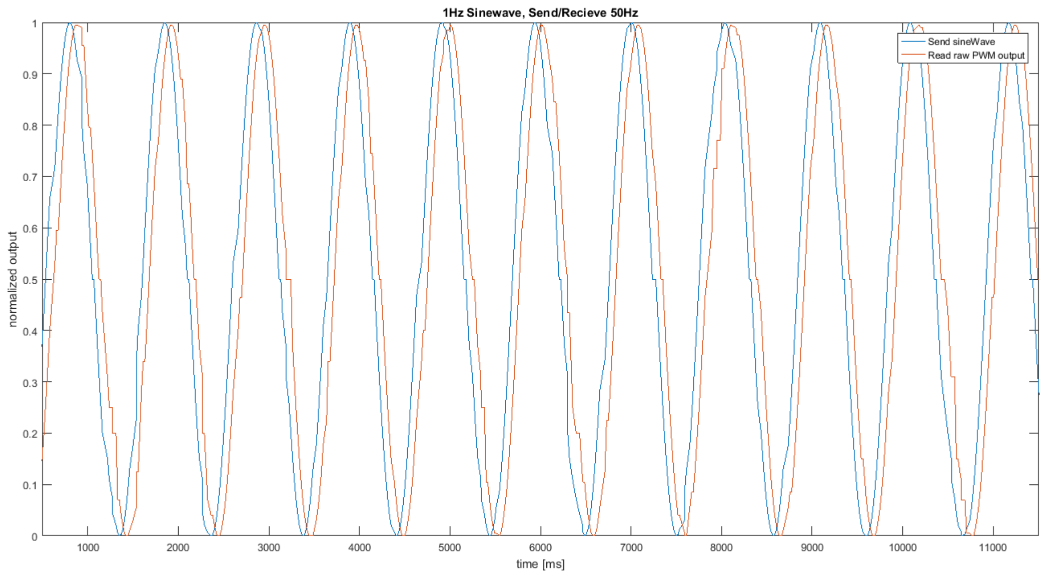


Figure 6: Phase shift between the sine wave send to the SPRK+ and the one received

4 Conclusion

By utilizing two opensource APIs for Spheros it was possible to make a new one for the Sphero SPRK+ model which uses BLE. Furthermore by only utilizing HCI on the Linux system it was possible to create a stable and somehow fast API where it is possible to stream sensors at almost constant 50Hz while sending commands synchronously. Therefore, it is now possible to utilize the API in this and future projects where controlling the Spheros have to be done. By running the API on a Raspberry Pi 3b it is also possible to use the collected data about timing issues for further enhancement of possible control algorithms. The complete API can be found on:
`https://github.com/miaar7/Sphero`
main.cpp includes the code for the tests included in this report.

References

[1] *About qt*, `https://sdk.sphero.com/api-reference/api-packet-format/`, (Accessed on 21/12/2017),

[2] *Api for older spheros*, `https://github.com/slock83/sphero-linux-api`, (Accessed on 12/19/2017),

[3] *Bluetooth specification*, `https://www.bluetooth.com/specifications`, (Accessed on 21/12/2017),

[4] *Sphero.js*, `https://github.com/orbotix/sphero.js`, (Accessed on 12/19/2017),