|  |  |
| --- | --- |
|  | Francotyp-Postalia |

Navigator Web - Communications

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| # | Version | Date | Authors | Comments/Modifications |
|  | 1.0 | 14.09.2016 | Hendryk Irmischer | initial revision (Demo Prototype) |
|  |  |  |  |  |
|  |  |  |  |  |

This document describes the implementation details of the communication between web clients (users), serial port (device) and all parties between.

Table of Contents

[1. Overview 2](#_Toc461618195)

[2. Server Side – Node.JS 3](#_Toc461618196)

[2.1. Program 3](#_Toc461618197)

[2.2. NPM serialport 3](#_Toc461618198)

[2.3. Serial Link 3](#_Toc461618199)

[2.4. Message Aggregator 3](#_Toc461618200)

[2.5. Message Pipeline 4](#_Toc461618201)

[2.6. WebSocket Server 4](#_Toc461618202)

[2.7. Web Server 5](#_Toc461618203)

[3. Client Side – Angular 2 5](#_Toc461618204)

[3.1. WebSocket Client 5](#_Toc461618205)

[3.2. Web Client 5](#_Toc461618206)

# Overview

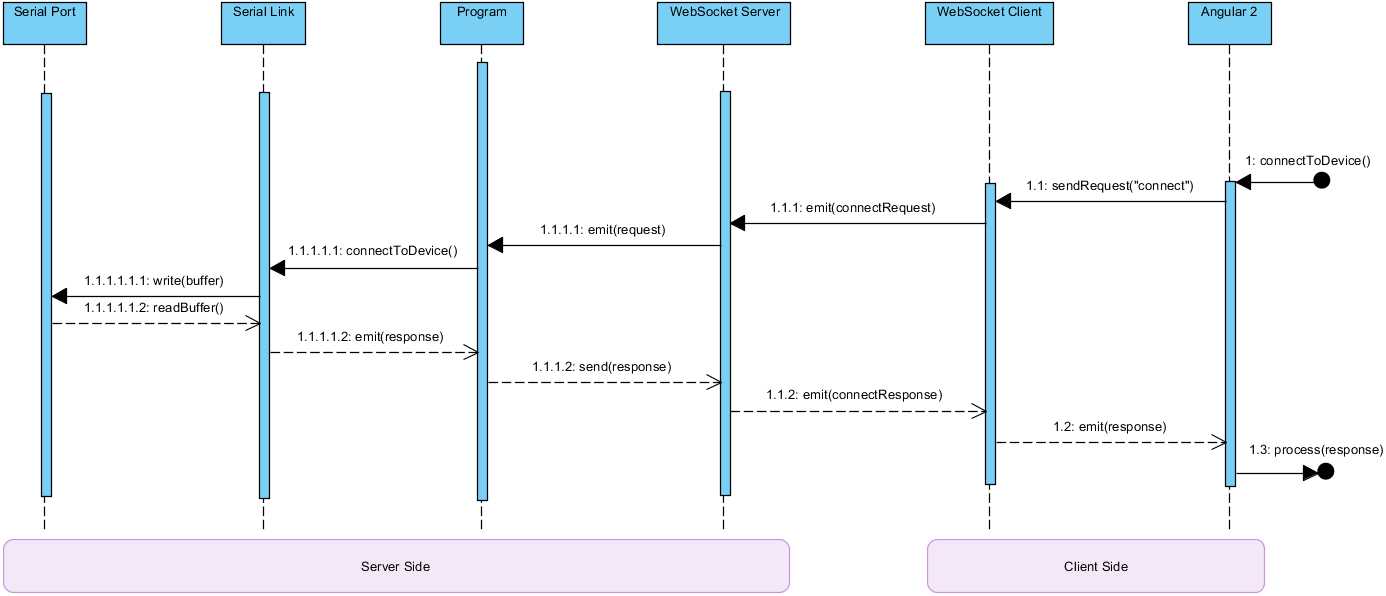


Figure 1 - Example of a communication sequence

|  |  |
| --- | --- |
| **Request** | |
| 1 | User opens the NavigatorWeb site and clicks on the connect button. |
| 1.1 | Angular2 asks the WebSocket client to send the request to the server. |
| 1.1.1 | The WebSocket client emits the request object through the request channel. |
| 1.1.1.1 | The WebSocket server then emits an “request received event” |
| 1.1.1.1.1 | The request has now bubbled up all the way from the client to the root of our Node.JS application. It now knows, that some user clicked on connect. Depending on the request we will decide here, which actions to execute. For the case of a “connect request” we will now ask the Serial Link instance to initiate the connection to the device. |
| 1.1.1.1.1.1 | The Serial Link will then write a specific buffer (a buffer that represents a connect request) onto the serial port. |
| **Response** | |
| 1.1.1.1.1.2 | The device will then send a response, which is being read from the serial port by the Serial Link instance. |
| 1.1.1.1.2 | After completely assembling a message buffer, the serial link will emit that message. |
| 1.1.1.2 | The response message has now arrived at the root of our Node.JS application. It is now being passed to the WebSocket server instance. |
| 1.1.2 | The WebSocket server instance takes the response and sends it through a specific channel to the WebSocket client. |
| 1.2 | The WebSocket client then emits an event, that a message has been received. |
| 1.3 | Now the response can be processed by the web client and business logic will be applied. |

# Server Side – Node.JS

## Program

The Program class represents the root of our whole application. All components are being initialized here within the “bootstrap” method. In addition, all communication flows are being processed here. For all practical purposes, the program class acts as a translator between the serial port (respectively the connected device) and the WebSocket server (respectively the user/browser).

## NPM serialport

The serial port communication is being handled completely by the SerialLink instance, which for its part uses the node module ‘serialport’ to access, read and write to the USB port, where the device has been plugged in. See <https://www.npmjs.com/package/serialport> for further documentation on the library. Technically, the serial port can be used like a normal file stream with the only exception, that an event is being emitted, when new data are being received, respectively when data can be read from serial port.

## Serial Link

The Serial Link class acts as a manager for the communication with the serial port. Messages are being written to and received from the device here. It translates requests into buffers that the device (that is connected via USB) can interpret.

To be able to communicate with a device, the Serial Link needs to establish a connection to the serial port and open it. Only when the serial port has been opened, we are able to read and write messages. To effectively communicate with the device itself and not only with the serial port, we need to also connect to the device by sending a specific buffer that represents a connection request. When the device accepts the connection, it sends back a ‘ConnectConfirm’ message and some other status messages. After receiving the ‘ConnectConfirm’ message, we can assume that we are now connected to the device. To not lose the connection, even when not sending any messages, we send a heartbeat to the device with a predefined interval. To close the connection to the device, we simply send a disconnect message and stop the heartbeat.

Every received buffer is being emitted by the Serial Link to the Program. Because of the fact, that incoming buffers can be incomplete or could contain more than one message (due to the limitation on the static read buffer size) the Program stuffs every received buffer into a message aggregator (see 2.4).

## Message Aggregator

The message aggregator is a rather simple class. Its only responsibility is to assemble fragmented buffers into complete messages. That is the Header, which consists of 2 Bytes (Message Type) + 4 Bytes (Message Size), and the Body with the size given in the Header. After a message has been completely assembled, an event is being submitted, that the assembly of a message has been completed.

The MessageAggregator class also implements the interface ‘IPipelineInput’ that acts as a receiver for the message pipeline (see 2.6).

## Message Pipeline

The message pipeline provides a structured way of processing (save to database; log specific messages) completely assembled messages. The developer can register tasks inside the pipeline that are being executed, after the pipeline input provided new incoming data.

After every registered pipeline task has done its job, the message is being passed to the pipeline output (IPipelineOutput) that in our case is the WebSocket server instance. It then sends the message back to the WebSocket client for further processing.

## WebSocket Server

The WebSocket server is the server endpoint for the communication with the client(s). All messages and events a client needs to get notified about are being sent from here. Depending on the type or message that needs to be sent, a different web socket channel is being used. The currently used channels are listed in the table below:

|  |  |  |  |
| --- | --- | --- | --- |
| **Data Type** | **Channel Name** | **Direction** | **Description** |
| Buffer | Socket.Channels.DATA | Server 🡪 Client | All messages from the device that went through the message pipeline |
| Boolean | Socket.Channels.CONNECTION\_STATUS | Server 🡪 Client | When the connection status changes, the client is being notified via this channel |
| String | Socket.Channels.MESSAGE | Server 🡪 Client | Channel for simple status messages from the server |
| Error | Socket.Channels.EXCEPTION | Server 🡪 Client | Channel for exceptions that occur on the server side |
| Request | Socket.Channels.CLIENT\_REQUEST | Server 🡨 Client | Channel for requests from client to server (e.g.: connectToDevice()) |

Figure 2 - See “/src/common/SocketChannels.ts”

## Web Server

The web server that serves the angular 2 application is using ExpressJS (a node.js-framework). It allows us, to easily define static and dynamic routes. For further documentation see <http://expressjs.com/>.

# Client Side – Angular 2

## WebSocket Client

The WebSocket client acts as the communication endpoint on the client side. For the communication channels being used, see also Figure 2. After receiving messages through one of the channels, the WebSocket client propagates these to the Angular2 application.

## Web Client

The web client is an angular 2 application which acts as an interface to the user. The received messages are being processed here and depending on the message, the UI bindings are updated and business logic is being applied.