This is an implementation guide for connections between server (Raspi) and client (Android app).

## Handshake/Engine Start

### Client

* 1. When the user long-clicks the engine start-n-stop (ImageView), a pop-up dialog will appear with an EditText in which the user enters server IP and port (server\_ip:port). At the bottom of the dialog there will be an “OK” button. When the user presses “OK” button the client will try contact the server “server\_ip:port/start\_engine(nanohttp\_client\_ip, nanohttp\_client\_port)”. The server will respond back with a message (“OK”). After a successful handshake the engine start-n-stop will change state icon but if the handshake was unsuccessful a toast message will inform the user about the error. The “server\_ip” and “server\_port” will be update from null and -1 to the correct values.

### Server (DONE)

* 1. The server will listen for calls “EngineSystem.handshake(nanohttp\_client\_ip, nanohttp\_client\_port)” from clients. In case this method doesn’t receive any arguments the default will be “nanoClientIP = null” and nanoClientPort = -1.
  2. Server will call hardware method “car.controllers.basic.EngineImpl.start()” to initialize GPIOs,PWM, etc .
  3. *TODO it must return if the engine (GPIOs, PWM, etc) has successfully been initialized.*
  4. *~~POSSIBLE SOLUTION 1:~~* ~~I will use Kotlin coroutines (Future-Callable?) in order to wait for a while for "EngineImpl.start()” to finish and get it’s result (Pair). The “handshake(...)” method will return the String part of the Pair.~~ ‎
  5. *POSSIBLE SOLUTION 2:* Why to use coroutines for a, by default, sequentially execution? The “car.controllers.basic.EngineImpl.start()” will be called sequentially by default!

### Hardware (ALMOST)

* 1. There will be an interface “car.controllers.basic.Engine” with 2 methods “Pair start()” and “Pair stop()”. Singleton “car.controllers.basic.EngineImpl” will inherit from “Engine” will be singleton class. Here will discuss “start()” method.
  2. *How about instead of functions “start()/stop()” to replace them with a “boolean var”?*
  3. *Maybe it would be better if “start” and “stop” methods have the working source at the interface. It would be more extendable when I would like to initialiaze more GPIOs.*
  4. “Start()” method will initialize every pin I need (set mode = input/output, set every pin to 0, etc) and before exiting it will set “engineState = true” (“true” means started, “false” means stopped). Then it will return a Pair<String, Boolean>(Pair(“OK” or error\_message , engineState)).

### Extras (DONE)

* 1. It also needs a function in the client which asks the server which asks the hardware if the engine has successfully started/stopped. The client will have a “val isEngineStarted” with a modified “get” function. This will call a server function “get\_engine\_state()”. This function will be in the “EngineSystem” class. This server function will read the state of the “engineState” from the “Engine/EngineImpl” class.

## Back Button Exit App/Engine Stop

### Client (DONE)

* 1. This action will be activated when the user long-clicks the engine stop ImageView or presses the device back button twice.
  2. This action will executes client’s function “stopEngine()”. This method will call server’s “stop\_engine()” function and will wait for a termination signal (like in “startEngine(...)”).
  3. After a successful termination (“OK” signal) the engine start-n-stop ImageView should change icon to “engine stopped” and if the initial action was triggered by the device’s back button the app should also “finish()”

### Server (DONE)

* 1. The server’s function “EngineSystem.terminateConnection()” will call “ EngineImpl.stop(): String”.

### Hardware

The function “EngineImpl.stop()” will shutdown, unprovision the GPIOS, PWM, etc and whatever it needs in order to have a bugless engine start after this shutdown.

* 1. *How about instead of functions “start()/stop()” to replace them with a “boolean var”?*

## Home Button Pause App/Parking Brake

### Client (DONE differently)

* 1. Parking brake will be activated with long-click at the ImageView. Parking brake will be deactivated the same way. Handbrake is another ImageView and it will not be discussed here.
  2. When the user long-clicks this ImageView the “var parkingBrake: Pair” with modified setter and getter functions at the “.network.ClientTriggeredRequests” will call appropriate server’s functions.
  3. Set function of the “var parkingBrake: Pair”will call server’s “ThrottleBrakeSystem.set\_parking\_brake\_state(state = true/false)” and will wait (using blocking coroutines) for a result from the server function.
  4. Get function of the “var parkingBrake: Pair” will call server’s “ThrottleBrakeSystem.get\_parking\_brake\_state()” and will wait (using blocking coroutines) for a result from the server function.

### Server (DONE differently)

* 1. Server’s function “ThrottleBrakeSystem.set\_parking\_state(state=true/false)” will call hardware’s “var ThrottleBrakeImpl.parkingBrake”with modified “set” function. Server’s function will return a Pair.
  2. Server’s function “ThrottleBrakeSystem.get\_parking\_state” will call hardware’s “var ThrottleBrakeImpl.parkingBrake”with modified “get” function. Server’s function will return a Pair.

### Hardware

* 1. Parking brake is different from handbrake. Parking brake applies to 4 wheels (if the vehicle is awd) and handbrake applies to the rear wheels.
  2. As said at server section, there will be a “var ThrottleBrakeImpl.parkingBrake: Pair” with modified setter and getter. The “set” method will enable/disable parking brake according to the “state” argument. The “get” function will be modified also, because I think that I can get data back from PWM pins. If I cannot get data back from PWM pins then there will be a regular “get” function.

## Handbrake

### Client

* 1. Handbrake applies to the rear wheels only. The user can activate it by clicking the icon. The handbrake must be active while the user touches the icon and inactive when the user releases the icon. The handbrake cannot stay active without touching it (like a physical handbrake with a button). For this, use the parking brake (will not be discussed here).
  2. The client will call server’s function “ThrottleBrakeSystem.set\_handbrake\_state(state = true/false)”. The client will call server’s function in a non-blocking way. This means that the client will not wait for any answer by server. I suppose that during driving the handbrake is necessary (U-turns, etc) so it must not delay neither the server (car) nor the client (driver).
  3. The client will, also, call the server’s function “ThrottleBrakeSystem.get\_handbrake\_state()” to get the state of the handbrake in a blocking way. This function will not be used during driving for the reason I mentioned above but I can be used to reset the state of the icon during engine start/stop, etc.
  4. Do not forget to deactivate the parking brake if it is already enabled. According to my structure of “changeMotionInteractiveIconsStatus()” in client and “set\_throttle\_brake\_system(…) in server in combination with the way I call them and everything should be updated like a charm.

### Server

### Hardware

## Throttle/Brake/Neutral

### Client

### Server

### Hardware

## Steering Left/Right/Neutral

### Client

### Server

### Hardware

## Left Turn Light (modify description for right/neutral also)

### Client

The ImageView will have an initial state, false because the left turn light should be off. When the user presses the image icon (ImageView) the client (Android app) will call a local function “setTurnLightState(direction=left, state=true/false)” which will change the initial state to the new one, change the image to ImageView and will contact server “set\_turn\_light\_state(direction=left, state=true/false)”.

*TODO find another implementation because this will have issues due to concurrency. Imagine the scenario where the user sets the light state to true and the thread to the client reads the previous state which will be false.*

*P~~OSSIBLE SOLUTION 1:~~* ~~A Thread (feedback\_info\_thread) will run continuously and will contact server “get\_turn\_light\_state(direction=left)” to get the current state from the server and call the “setTurnLightState(direction=left, state=true/false)”.~~

*~~POSSIBLE SOLUTION 2~~*~~: A server(NanoHTTP) will run at the client (Android app) which will receive a request from the server (Raspi) about the left turn light state whenever it changes. Then the client should call the local function “setTurnLightState(direction=left, state=true/false)” according to the request data it receives.~~

*POSSIBLE SOLUTION 3*: If method “set\_turn\_light\_state(direction=left, state=true/false” in server always returns a state from the hardware controllers “ElectricController.getTurnLightState(direction=left)” then the client will always be up-to-date with state (true/false) so there is no need to implement *possible solution 1* and *2*. *This solution maybe needs coroutines in order to wait for the “ElectricController.setTurnLightState(direction=left, state=true/false)” to finish its job before getting data from “ElectricController.getTurnLightState(direction=left)”.*

### Server

* 1. The server will be listening “set\_turn\_light\_state(direction=left, state=true/false)” for events. When an event arrives it will call the method on the hardware controllers “ElectricController.setTurnLightState(direction=left, state=true/false)”. Before exiting this method, it will get the state for the left turn light“ElectricController.public val directionMap[“left”]” and will return the result to the client (Android app) as an HTTPResponse to his HTTPRequest to turn on the left turn light “set\_turn\_light\_state(direction=left, state=true/false)”.
  2. This way I ensure that the state (true/false) has reached the hardware controllers and I get the state (true/false) from hardware controllers too. If, for any case, the left turn light will not turn on and the state has successfully reached the hardware controller it would be a hardware issue (controller software or the real hardware).

### Hardware

The “ElectricController” singleton will have a method “ElectricController.setTurnLightState(direction=left, state=true/false)” which receives commands from server only.

This class will have also 2 maps “public val directionMap” and “private val directionMutableMap eith the same content ([“left”]→true/false, [“right”]→true/false, [“neutral”]→true/false) with initial values to “false”. Actually, the “directionMap” will be a public unmutable map reference of the “\_directionMutableMap”.

The method “ElectricController.setTurnLightState(direction=left, state=true/false)” will update (set every key to “false” value and then set the new value to the key from the method’s parameter, example [“direction\_param\_value”]→state\_param\_value) the “directionMap” accordingly and then will apply the changes to the shift register hardware “ synchronized controlShiftRegister()”.

The “synchronized controlShiftRegister()” will get every value from the “directionMap” and update the led values accordingly. Pay **attention** to the different kind of boolean values, the boolean value which indicates if the user wants the left turn light to be on, and the local (to the method) boolean variable which controls the left turn light to change between on/off when the user wants to actually turn the left turn light on. To remember how the shift register works see my example at Tinkercad.