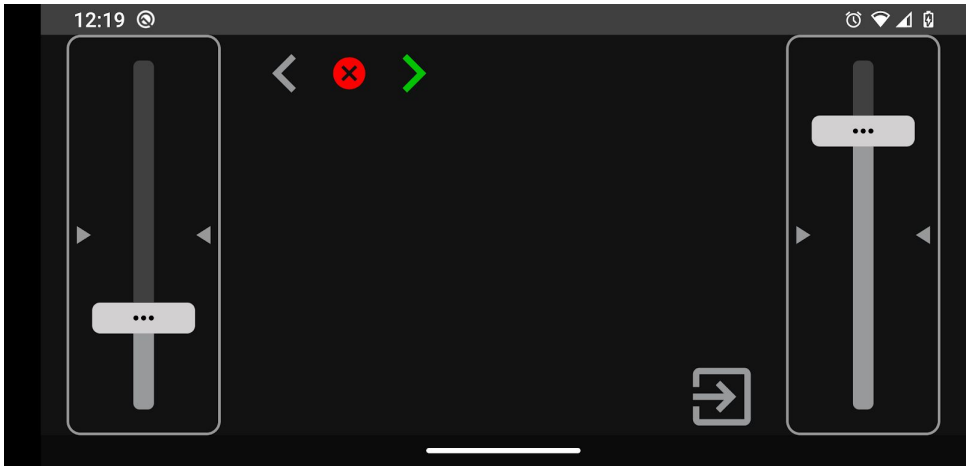
OpenBot Controller technical Overview

Ivo Zivkov
izivkov@gmail.com
Date: January 2nd, 2021

Introduction

The OpenBot controller (controller) is an Android Application for remotely controlling the OpenBot vehicle (robot). It communicates with the main OpenBot android application running on the OpenBot phone. Here is an image of the main screen:



Connecting to the OpenBot

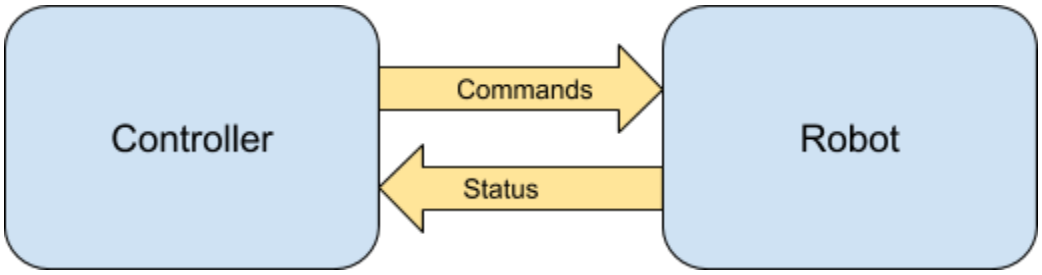
The controller establishes connection to the robot via the Android Nearby Connections API. This technology uses Bluetooth or WiFi to quickly connect to another device. This may change in the future, as we may require WiFi-only connection in order to send video stream from the robot to the controller.

Upon starting, the controller advertises its presence using the Nearby API's. When the "Phone" control is selected on the robot's menu, the robot starts a "Discovery" process, looking for advertisers. This process will be automatically terminated after 1 minute to save batteries, in case no connection is made. If a connection is established, the robot will stop discovering, and the controller will stop advertising. At this point, full communication is established, and the controller can start sending commands to the robot.

Data Flow

After connection, the robot and the controller establish a two-way communication:

1. The controller sends commands to the robot
2. The robot sends status information to the controller.



Data Format

Both **commands** and **status** information are in JSON format. Currently, the following commands are supported:

1. Commands:

`{command: "LOGS"}` - toggle Logging

`{command: "NOISE"}` - toggle Noise

`{command: "INDICATOR_LEFT"}` - turn on left indicator, turn off right indicator if ON

`{command: "INDICATOR_RIGHT"}` - turn on right indicator, turn off left indicator if ON

`{command: "INDICATOR_STOP"}` - turn off right and left indicators

`{command: "NETWORK"}` - toggle network

`{command: "DRIVE_MODE"}` - cycle between drive modes: DUAL, GAME, JOYSTICK

`{driveCmd: {l:0.2, r:-0.34}}` **-** Send a drive command to the robot. The “l” and “r” values indicate the speed of left and right wheels, and can only be between -1 and 1.. Negative numbers indicate reverse. This is how the robot is steered.

2. Status

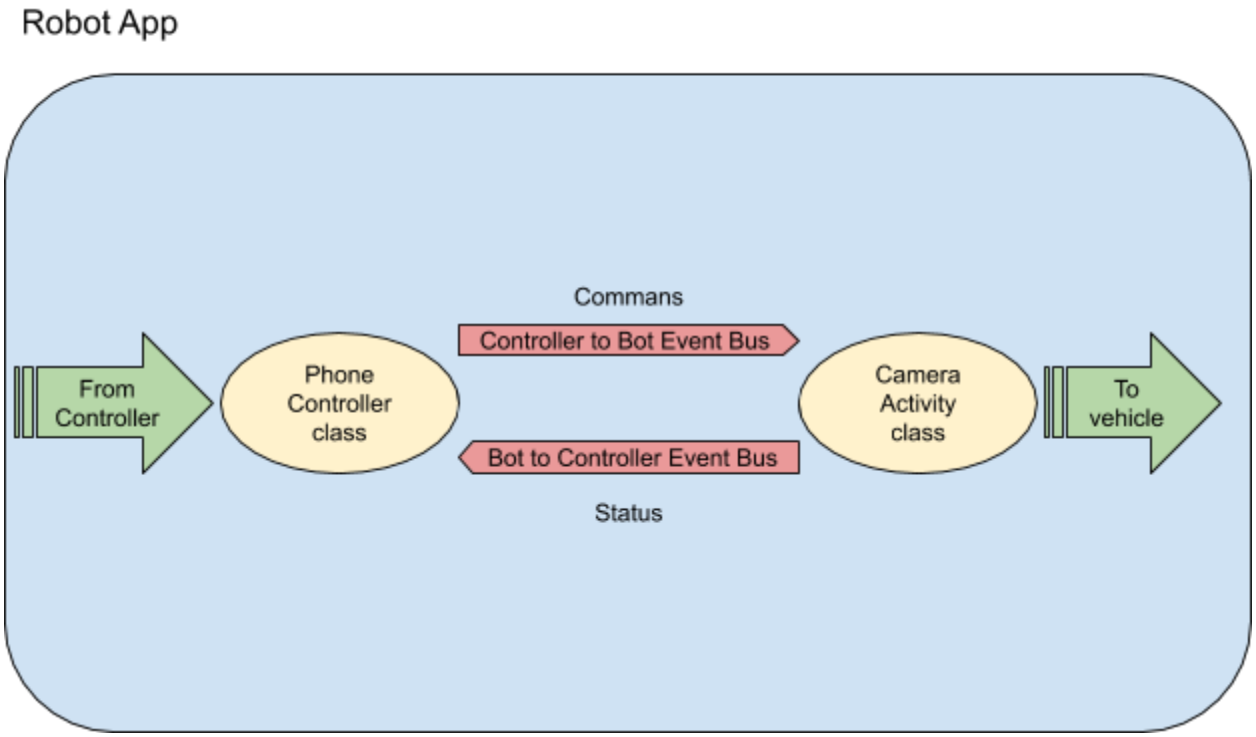
Status data is sent from the robot to controller to indicate the current state of the robot. The controller can display its components according to this information. The status can be sent at any time. The robot application observes for changes in its state, and sends status data to the controller when this happens. Status data can be sent one value at a time, or in bulk.

```
{status: {"LOGS": "true | false"}

{status: {"NOISE": "true | false"}

{status: {"NETWORK": "true | false"}

{status: {"DRIVE_MODE": "game | dual | joystick"}

{status: {"INDICATE_LEFT": "true|false"}

{status: {"INDICATE_RIGHT": "true|false"}

{status: {"INDICATE_STOP": "true|false"}
```

Or bulk mode:

```
{status: {"INDICATE_STOP": "false", "NOISE": "true",
"DRIVE_MODE": "dual"}
```

How Commands and Status is handled in the Robot



The robot app provides two channels of communication between the main `CameraActivity` class and the `PhoneController` class.

The `PhoneController` class handles the interface to the Controller connection. In this case, it uses the `NearbyConnection` class to receive and send data to the remote controller. Once data is received from the controller, the `PhoneController` class sends an RX event to the CameraActivity via the `ControllerToBotEventBus`. The `ControllerToBotEventBus` class handles all the details, so to send command event is really simple:

```
ControllerToBotEventBus.emitEvent(new JSONObject({command: ”LOGS”}));
```

To receive a status events from the `CameraActivity` class is also simple. Subscribe to status events like this:

```
BotToControllerEventBus.getProcessor().subscribe(event ->
handleThisEvent(event));
```

In our case, we simply send back the status information to the controller using the `NearbyConnection` class:

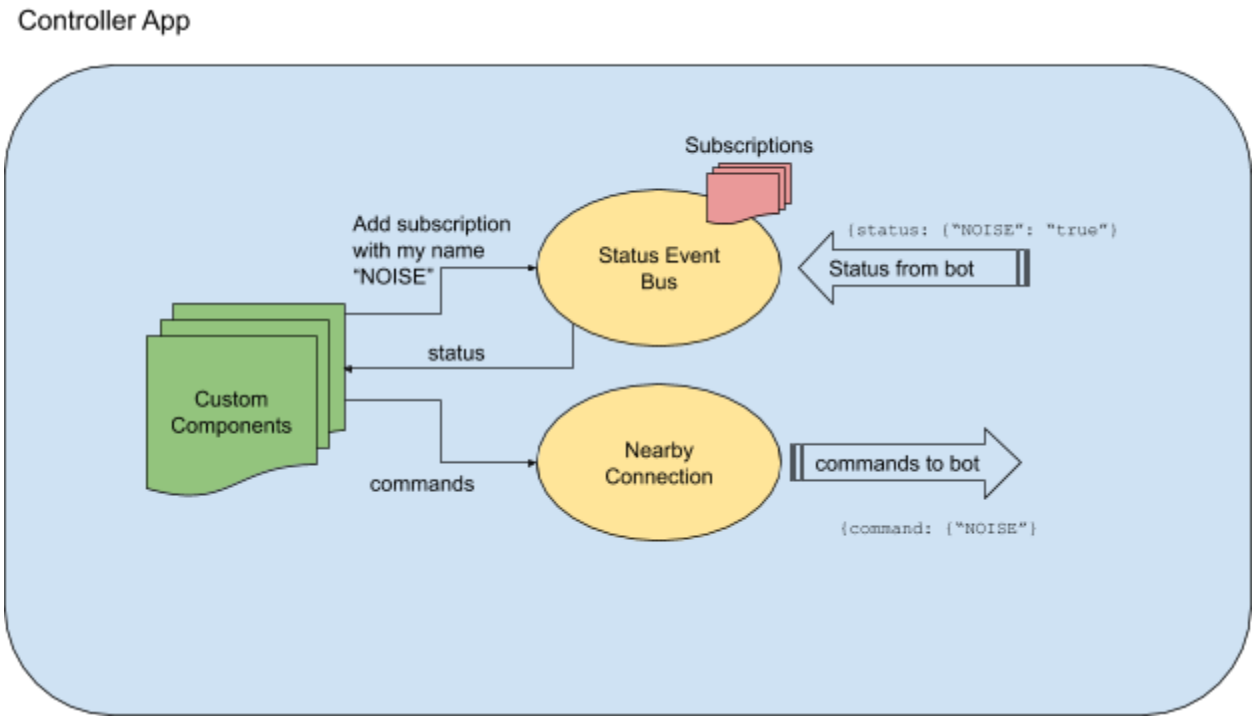
```
BotToControllerEventBus.getProcessor().subscribe(event -> send(event));
```

… where the `send(event)` functions send the data to the controller.

Note that using event buses makes it easy for other remote devices to issue commands to the robot, without the need to change any code in the CameraActivity and NetworkActivity classes. For example, if we are adding a web client for controlling the robot, we can create a class similar to the `PhoneController` to receive commands from the web server in any format, then just create and emit an event and send it via the `ControllerToBotEventBus`. This way, the rest of the app becomes a black box.

Controller Internals

The controller app is written in Kotlin. Most of the app's logic is handled by custom components which send their own commands, and receive status information related to them from the robot. The custom components may change their appearance depending on the status they have received from the robot. For example, if the robot reports that the Noise function has been turned on, the Noise button will change its color to reflect that state. Drive control commands are sent from the Right and Left `DualDriveSeekBar` components, which appear as sliders on each side of the controller screen.



Components listen on "TouchEvents" from the user and send commands to the controller. Each component will send a different command. For example a Noise button will send command:

```
{commad: "NOISE"}
```

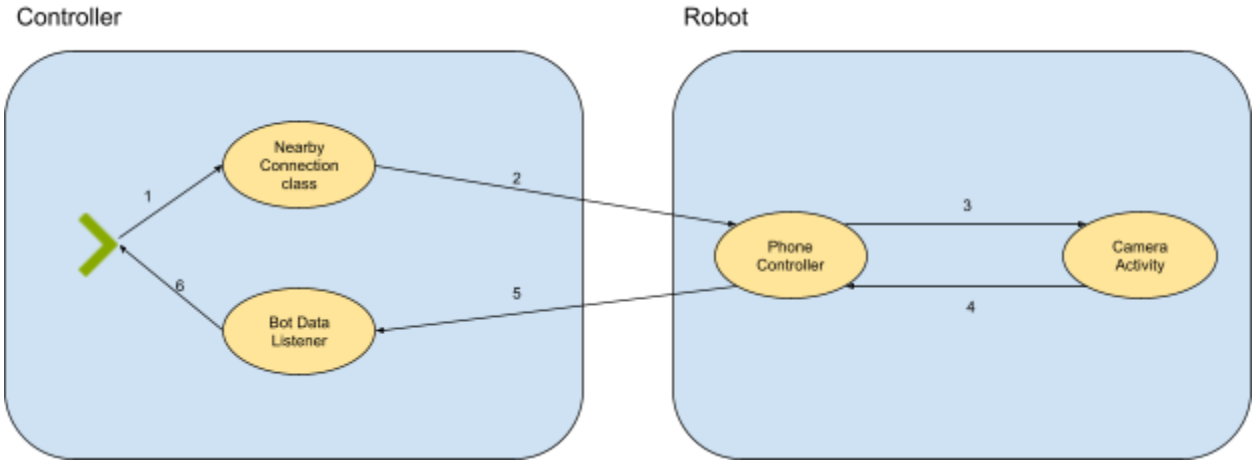
Each component has a name associated with it. To get status events, the component will subscribe with `StatusEventBus` class to receive events on this subject (name). In the example above, the status message would be:

```
{status: {"NOISE", "true"}
```

The `BotDataListen` class (not shown) will match the event name "NOISE" with the name of the subject "NOISE" and will instruct `StatusEventBus` to send the event on this subject. The component which subscribed to the "NOISE" subject will receive the value, and will update its appearance.

Putting it all together

Let us trace all the steps that take place when the user presses the "INDICATOR_RIGHT" button on the controller (this is the blinking green arrow on the top-left of the first diagram in this document).



1. The user presses the green arrow button. This generates `TouchEvent`, and invokes the `send()` method on the `NearbyConnection` class to send **`{command: "INDICATOR_RIGHT"}`** command to the controller.
2. The NearbyConnection class sends the command to the controller.
3. `PhoneController` class in the robot app detects a new command has arrived, and passes it on to the `CameraActivity`
4. The event is received by the `CameraActivity` class, and the command is passed to the `Vehicle` class (not shown) to turn on the signal in the robot.
   This state change is detected and the `CameraActivity` class sends a status command:

`{status: "INDICATOR_RIGH", "true"}`

to the `PhoneController` class.

5. The `PhoneController` re-sends the status back to the controller, and there it is received by `BotDataListener` class.
6. `BotDataListener` routes the status message to the correct custom control via the `StatusEventBus` class and using `INDICATOR_RIGH` as the subject. The component receives the status messages and starts animation to blink every second. It will stop the animation if it receives **`{status: "INDICATOR_RIGH", "false"}`** message.