Developer notes

Introduction

This document contains some relevant notes about Remu's structure. The purpose of this document is to shine a light on the more obscure and/or complex parts of the application. For a seasoned developer it might not be necessary to read these notes at all, but it might provide a better understanding of the motivations behind the solutions made in Remu.

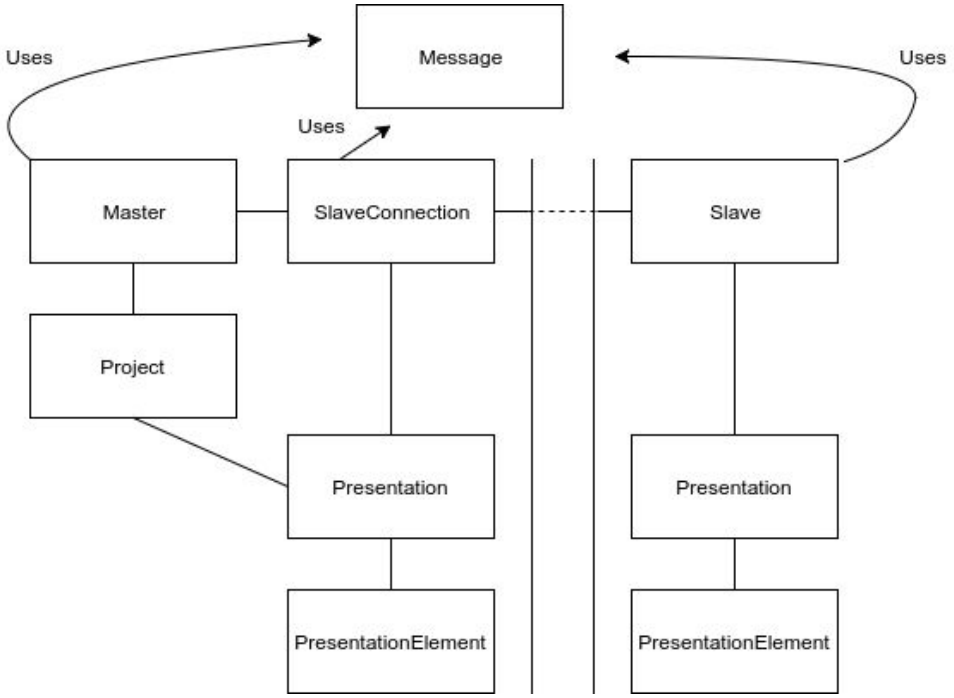
If you, dear reader, partake in the development of Remu, please consider to append this document with notes of your functionality if you see it fit to document it more specifically.

Domain structure

Remu's logic for handling the metadata of a project is divided into 7 classes. Each of them exists for a reason, so in this section all of the classes are briefly described, to get a sense of how a project is seen and handled inside the RemuApp.

Remu's logic can be seen as a hierarchical structure, meaning that when a class is lower in the hierarchy, it handles a more limited responsibility in the application. The only domain class that is somewhat outside of this hierarchy is the Message class, which is used by multiple classes to handle data transfers.

The aforementioned hierarchy is described in the figure below. Please note that the figure is not strict UML, but more of a sketch.



Master, Slave and SlaveConnection:

Even though Master class handles most of the runtime action, Master, Slave and SlaveConnection are seen to exist in the same level in the Remu hierarchy. Reason for this is that Master and Slave are seen as different sides of the same coin, meaning that when the program is advanced enough, Slave can be changed to a Master at any point during runtime, and vice versa. SlaveConnection is seen to be at the same level of hierarchy because it simply represents a single slave to the master, so from the viewpoint of the master, SlaveConnection is the Slave.

Project:

The project class contains metadata of all of the presentations in the project. Project class is created in RemuApp, because it should be able be sent for another slave during runtime when a slave is changed into a master. At the time of writing, this feature is yet to be implemented.

Only the master class and RemuApp may handle the project class. Slaves should not have access to the project class.

Presentation:

Presentation class contains metadata of a single presentation. This means that when showing images from one slave and videos from another, the master has an instance of project that contains two instances of presentations. Presentation knows its visuals through PresentationElement class, which is described below.

PresentationElement:

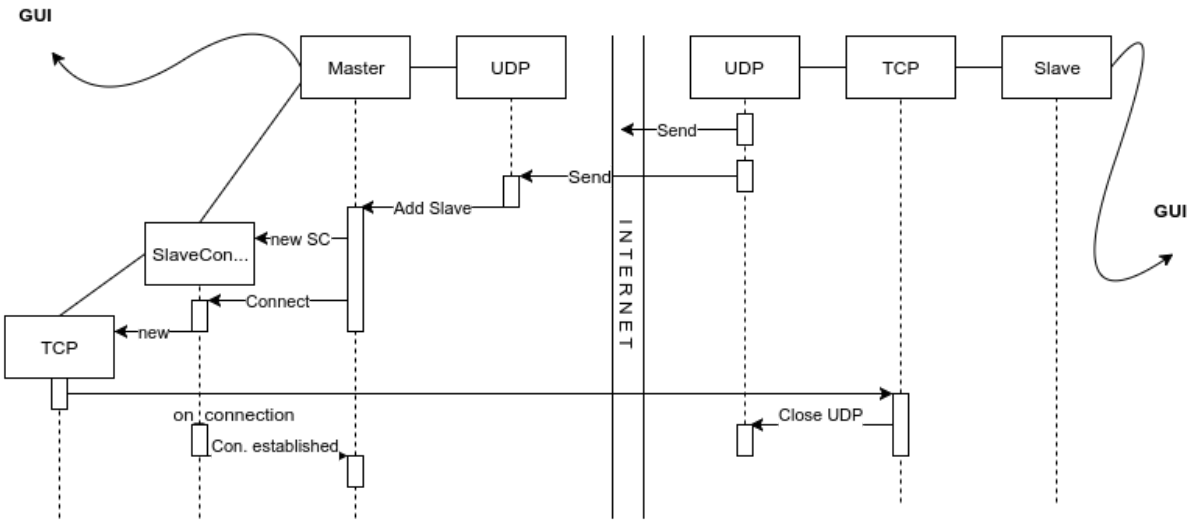
PresentationElement represents a single visual. Most importantly, PresentationElement should be able to represent any kind of visual supported by Remu, whether it be an image, text or video. PresentationElement has no knowledge of its parent at the time of writing, because it doesn't need it. It does know its own element type and source. PresentationElement is also meant to know its content. This can be a bit misleading though, because with videos and images the content is the same as the source. But with text files, the content is the actual text, and not a text file.

Message:

Message class represents messages that are sent between RemuApps, and adds information on the messages it sends. The message class is essentially a JSON wrapper to a JSON message, that is there to add headers to messages.

“The ReMu handshake”

Remu uses UDP protocol to find slaves during runtime, but when a slave is found, the connection between the master and the slaves is handled with TCP. This is because Remu is not supposed to know IP addresses of slaves before a slave informs that it can be used. Because of this, Remu has a way of finding slaves through wireless internet, which is detailed in the figure below.



Text