# Using the Illuminator Simulation Tool and Embedded Systems to Visually Demonstrate Energy Systems

By Bram Dorland and Alje Vermeer

Student: Bram Dorland (5870399)
Student: Alje Vermeer (5845416)

Thesis committee:
dr.ir. I. (Ioan) Lager,
dr.ir. M. (Milos) Cvetkovic,
dr.ir. J. (Jort) Groen

July 2, 2025

# Abstract

This thesis discusses the integration of the Illuminator software with hardware components, for the purpose of creating a table-top, Plug-and-Play energy system demonstrator. The thesis starts by introducing the motivation for an energy system demonstrator in Chapter 1. An interactive, visual demonstrator can help educate people on energy systems in a way that is more intuitive. The main product should show the power flow on the table, implement Plug-and-Play dynamics, and be scalable.

LED strips are used to visualize power flows in the table-top network, in combination with the Digispark ATtiny85. For determining the topology, static ID pairs were used. A double simulation is used to implement Plug-and-Play dynamics. The first simulation configures the topology used by the second simulation, based on the hardware connections. The second simulation runs the Illuminator simulation. During this simulation, checks are run to see whether a physical connection has changed, such as a cable being unplugged. The simulation and then starts the reconfiguration process again.

Testing the reliability and run-time of the implementation is documented in Chapter 6, with a focus on how well the implementation scales with the size of the simulation. It was concluded that the Digispark's communication with the Raspberry Pi would often stall, requiring error correction to be implemented. Even then, the data transfer to the Digispark from the Raspberry Pi fails on the first try an average of $48\%$ of the time. Determining how long a setup takes to reconfigure was estimated using a computer, since the Raspberry Pi's aren't powerful enough to simulate dozens of models. It was determined that a reconfiguration of 20 models takes about 100 seconds.

# Preface

This thesis was written in the context of a Bachelor Graduation Project for the Electrical Engineering Bachelor at the TU Delft. The project was in collaboration with the developers of the Illuminator Simulation Tool, which is an open-source software designed for simulating energy systems. [1] The goal of this project was to implement the Illuminator on hardware components, namely Raspberry Pi's, in a table-top, Plug-and-Play fashion, such that the simulation is demonstrated visually, and can be manipulated through interaction with the hardware components. Pursuing this goal required expansion of the Illuminator, and the introduction of several embedded systems, in conjunction with the Raspberry Pi's. The coding projects undertaken during this project can be found in a fork of the Illuminator GitHub, called "Illuminator-BAP-EE-2025", under the branch "cluster-BAP-GroupB". [2] The most important pieces of code can be found in the appendix included in this thesis.

We would like to express our gratitude to our daily supervisors, Jort Groen and Despoina Georgiadi, and our supervisor Milos Cvetkovic, for their guidance during the course of this project. We would also like to thank our other group members, Jane Libier, Roos Bonouvrie, Kyrian Rahimatulla, and Andy Zhang, who comprised the other sub-groups of this Bachelor project, and made for a productive and enjoyable experience.

# Contents

# Chapter 1

# Introduction

## 1.1 Energy Transition

The ever growing threat of climate change continues to drive the global transition to clean energy, however integrating clean energy into existing energy grids poses many challenges. Grids that once relied on a rather steady supply of energy from coal and gas plants have to adapt to the intermittency of renewable energy sources (RES), such as wind and solar. Storage and Power-to-X solutions are increasingly necessary in order to manage the highs and lows of supply and demand, and grid congestion is a serious concern.

However, one limitation of the energy transition that is often overlooked among the technical challenges is the general lack of understanding. Energy systems are enormous, complex, and difficult to intuitively grasp. A simple diagram or a supply-demand curve is not enough to demonstrate a village's energy system, or even that of a single house, let alone a vast, ever-changing, intercontinental network.

Though the urgency of the energy transition is well known, the means of achieving a sustainable, clean energy system are not. In order to actually transition to clean energy, the human understanding of energy systems needs to develop. The gap between the immense complexity of energy systems and the general understanding of them needs to be bridged.

# Chapter 2

# State of the Art Analysis

## 2.1 The Illuminator Simulation Tool

The Illuminator Simulation Tool is an open-source software kit developed by the TU Delft, aimed at demonstrating the challenges of energy transition and the benefits of intelligent multi-energy systems. It aims to help utility companies, students, and policy makers better understand the complexities of energy systems, including congestion, intermittency of clean energy, and more. The energy system components (generators, consumers, transmitters, storage facilities) are scaled down in complexity, so that they are more intuitive to understand, and so that they can be realized at the table-top level, using Raspberry Pi mini-computers and 3D models. [3]

In order to get the external simulations to start, a secure shell is started for each model. The secure shell provides a secure remote login to a different device over an insecure network. [4] It's currently used in Illuminator by making a shell file containing all details of the external simulations. It then starts the simulations and the simulations wait for the master to connect to them.

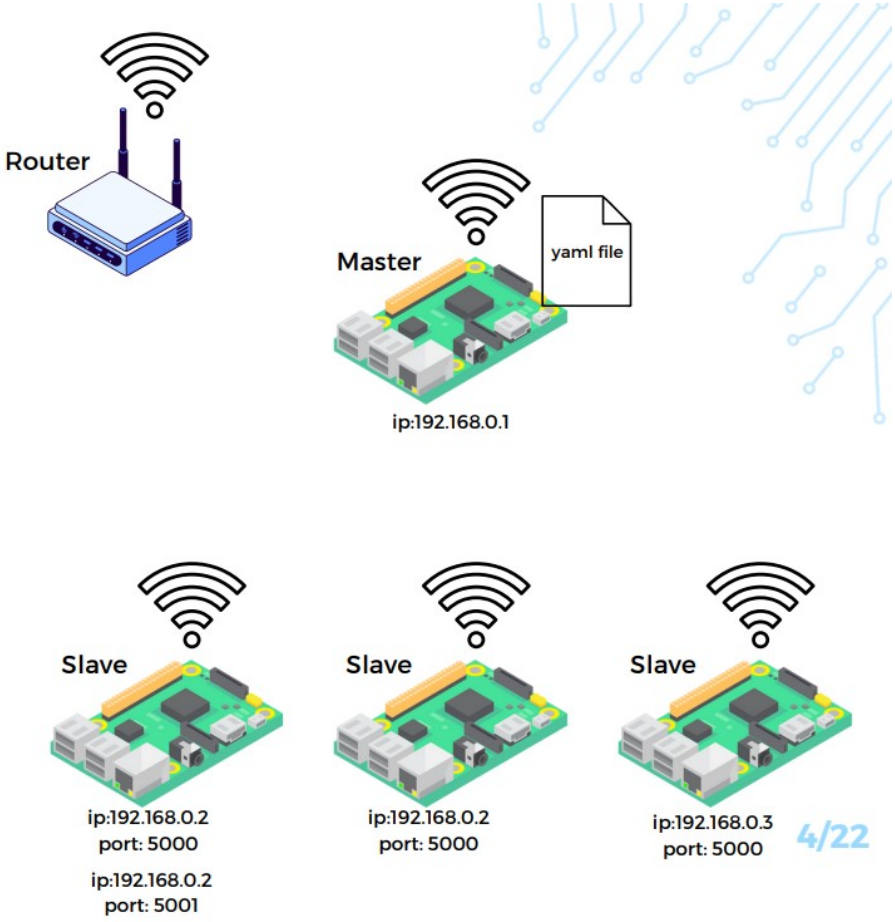



Figure 2.1: Illustration of a Cluster of Raspberry Pi's communicating with a Master Pi over Wi-Fi. A Slave Pi can run mutiple models by assigning different ports, starting at 5000

### 2.1.1 Mosaik

Illuminator uses Mosaik as its basis. Mosaik is a smart grid simulation tool. It has only one model for each node. [5] The current Mosaik version has a high level API and a low level API, which are written in Python. Illuminator implements the high level API as it's engine, which will be used in this report. Illuminator doesn't change that much in the modeling interface. For a model to work, you have to at least implement the step function in your model, but usually you also use the __init__ function to set some things up in the model. [6][7] That makes Mosaik a very useful tool, as one doesn't have to worry about scheduling a simulation. The low-level API does the actual simulation. The current version of Mosaik can not only simulate based on time, but can also simulate based on events. [8] The added benefit of this type of simulator is that you don't have to run the step method at every time point for every model. The time at which the next step should be run can be defined in the current step.

## 2.2 Raspberry Pi

The Raspberry Pi is a mini-computer. It has 4 USB ports, Wi-Fi connection and Ethernet, a power supply port, and an HDMI port. The model used in this report is the Raspberry Pi 3B+. It has 1GB of memory and the processor has an ARM architecture with a frequency of 1,4GHz. It also has 40 GPIO pins for external hardware. [9] The Raspberry Pi usually runs the Raspberry Pi operating system, as this is optimized for the Raspberry Pi's.[10]

The Illuminator simulation demonstrator uses a network of Raspberry Pi's, connected to other physical components. The Raspberry Pi's are responsible for running Illuminator models, visualizing elements of the simulation, and communicating information with other Pi's and physical components. One Raspberry Pi can also run multiple models at once. They can be distinguished by giving each model their own port on the same device. The master can then separate models via the IP address and port combination.

## 2.3 Addressable LED strip

The power flows are visualized using WS2812b LED strips. A micro-controller can control each LED individually by sending a 24-bit color code to the LED strip. It does so for each LED. The first LED sets itself based on the first of the 24 bits and passes the rest of the received bits on to the next LED. If there are no new bits received by the LED for more than 50µs, the LED can receive new data. The timing is very precise, because a one and a zero bit are achieved by sending a high and low voltage for a certain time. [11] This type of protocol is called a non-return-to-zero protocol.

There is a library provided by Adafruit, the Adafruit Neopixel library, that can control the LED strip. This library is used for controlling the LED strip. You only have to specify how many LEDs are on your LED strip. [12] The LED strips are made of a flexible PCB, which can be connected to other components with soldering. This allows the PCB to be controlled by an external component, such as a micro-controller.

## 2.4 ATtiny85 Programmable Chip

For the table-top setup, a micro-chip board that can connect to a Raspberry Pi, via USB, is required. The micro-chip board used is the Digispark ATtiny85 [13], which is a circuit board featuring an ATtiny85 micro-controller [14] with USB programming capabilities. This chip is discussed in detail in Chapter 4. The primary reasons for choosing the Digispark ATtiny85 board are its availability, low cost, and small size. For customers wanting to set up their own table-top networks, these traits are very attractive.

### 2.4.1 Introduction to the Digispark ATtiny85

The Digispark ATtiny85 is a circuit board, featuring an ATtiny85 microprocessor that is USB programmable. The ATtiny85 is comparable to most Arduino's, but with fewer control pins, less RAM, and a smaller size. Even with the small size, the ATtiny85 is capable of several means of communication, both digital and analog. The Digispark was chosen specifically due to its small size (26×12 mm) and USB capability, allowing for easier connection with the LED strips and Raspberry Pi's. The low price and high availability were also attractive properties, since a large focus of the project is replicability and scalability.

The ATtiny85's main function within this project is to control LED strips, used to visualize power flows in the table-top network. Its secondary function is to serve as a means for Raspberry Pi's to determine their connections to other Pi's. The upsides of the Digispark ATtiny85, specifically, is that these functions can be integrated with the existing connection configuration, due to its USB capability.

### 2.4.2 Programming the ATtiny85

The ATtiny85 is programmed through a sketch that is uploaded to the chip via USB. Only one sketch can be uploaded to the chip at a time, with the previous sketch being wiped. A sketch only needs to be compiled and uploaded to the ATtiny once. When the ATtiny is powered, it will wait five seconds for a new sketch upload. Should no upload occur, its current sketch will run for as long as it maintains power. Writing a sketch is done in Arduino IDE, an open-source programming environment meant for programming software and chips. This environment runs a bootloader for the USB/serial programmer, which consumes flash memory.

Programs are saved to the ATtiny85's 8KB of flash memory, reduced to 6KB after the bootloader is loaded. The ATtiny85 also has 512 bytes of EEPROM (Electronically Erasable Programmable Read Only Memory). Flash and EEPROM memory are both saved when power is lost, including between uploads. Flash memory differs from EEPROM memory in that flash memory can only erase and rewrite data in blocks, whereas EEPROM can erase and rewrite data in individual bytes. EEPROM is generally preferred for cases where data needs to be frequently manipulated, since it is more precise. Flash memory is, however, faster than EEPROM, and the ATtiny85 has much more of it.

### 2.4.3 Serial Communication

An important limitation of the Digispark is its lack of hardware serial communication. Communication between the Pi and the ATtiny85 must be done through either software serial or USB-to-serial communication. For the scale of this project, relying on USB-to-serial adapters is not feasible, as they are less easily integrated with the LED strips and connectors, which would add unnecessary complexity to the physical setup. Software serial communication is preferred, provided the ATtiny's limited program storage space allows for it.

There are many software serial options already developed that are compatible with the ATtiny85. The options considered for this project were SoftwareSerial [15] and DigiCDC [16]. These options both utilize the creation of a virtual serial port, or software serial port, through which the serial communication is done. Both also require the addition of their respective libraries to the program, in order to function. Which library to use was determined after testing each library with code written to control the LED strips. It became apparent that there was a conflict between the SoftwareSerial and Adafruit Neopixel [17] libraries, where both tried to use the same vector at the same time. After investigation, it was concluded that there was no feasible workaround for this conflict, so the DigiCDC library was chosen.

### 2.4.4 Alternative Development Boards

There are various development boards that perform similarly to the Digispark ATtiny85. The main criteria the boards are graded on are their availability, cost, and size. How easy they integrate with the existing hardware is also taken into account. For example, a board with a USB-C port will be less favourable than a board that can connect directly to USB-A, since that is the port the Raspberry Pi's have. Described below are two of the many development boards that were rejected. Most notably, no alternative boards were found that were directly compatible with a USB-A port, opting instead for Micro USB and USB-C ports.

- **Arduino Micro** [18]: A larger micro-controller board (48×18 mm) based on the ATmega32u4, which has more pins, flash memory, and EEPROM memory than the ATtiny85. [19] While more powerful, the Arduino Micro only comes with USB-C or Micro USB ports, and is typically more expensive than the Digispark ATtiny85. The long, rectangular shape of the board also gives the LED strips less flexibility in movement.
- **Seeeduino Nano** [20]: A micro-controller board (43×18 mm) with comparable properties to the Arduino Micro. This board is built around the Atmega328P, which is very similar to the ATmega32u4. [21] The cost is similar to the Digispark ATtiny85 board, however the Seeeduino Nano has the same drawbacks as the Arduino Micro, in that it is rectangular in shape, and only comes with a USB-C port.

## 2.5 Competitive Simulation Softwares

- **DIgSILENT PowerFactory** [22]: Powerful simulation software with a wide range of functions, including distributed simulation, real-time simulation, and performance monitoring of energy systems.
- **ETAP Power Simulator** [23]: Network modeling, analysis, and simulation software whose subscription is widely customizable, depending on the intended use. Custom options include protection coordination and maintenance, harmonics analysis, and further add-ons for safety, sustainability, etc. Integrates with data services to provide real-time simulation
- **Siemens PSS®SINCAL** [24]: Software platform for electricity network planning, simulation, and analysis. Offers various modules for more specialized functions, such as extensive analysis, parallelized analysis, protection, optimization, and more. Capable of integrating real-time data and time-series data.

# Chapter 3

# Program of Requirements

This project is an expansion of the Illuminator project at the TU Delft. The objectives of the Illuminator project are listed below:

- Present common energy technologies and systems: Illuminator should be able to simulate common systems, such as wind turbines, PV panels and houses
- Table-top design: The kit should be compact, such that common-sized tables can fit it
- Extendibility/Scalability: The design should be able to scale to very large systems. That means that there shouldn't be hard limits in terms of devices and such
- Plug-and-Play capability: The devices should be easy to setup, connect, and disconnect, even while the simulation is running
- Easy on-the-fly reconfiguration of the examples: The simulation should respond to changes in the table-top connections, allowing for interactivity and rapid reconfiguration of the simulation
- Replicability: The setup should be easy to replicate with fresh components, and the simulation should be consistent in its responses to the table-top network

This thesis is one of three parts of a larger Bachelor graduation project. The goals of the over-arching project are to demonstrate the power flows and characteristics of energy systems on a table-top network, and add Plug-and-Play functionality to the simulation. This thesis focuses on visualizing power flows and adding Plug-and-Play. The specific requirements for achieving these goals are broken down using the MoSCoW method, which is a technique for arranging requirements based on their priority:

**Must have**:

- The design must visualize the flow of power between two components in the network.
- The design must be able to automatically configure itself based on the physical hardware configuration.
- The design must be compatible with Illuminator and all its dependencies.

**Should have**:

- Hardware that is provided and available, or has been worked with before, is preferred over newly introduced or unfamiliar hardware.
- It is preferable that the work done during this project is easy to expand on in the future. This includes theoretical designs that couldn't be implemented within the time-span of this project.

**Could have**:

- Remove the need to manually create and run the shell file. This will make running a simulation much simpler for the average user.
- Make the whole simulation an endless loop which can be run indefinitely.

**Won't have**:

- Peer-to-peer networking for creating and running the simulation. This would require rewriting Mosaik to support this, which is not feasible in the given time frame.

# Chapter 4

# Digispark ATtiny85 - LED Strips and IDs

## 4.1 Controlling the LED Strip

The visualization of power flows in the table-top network is done using LED strips. These strips consist of many small segments soldered together, each with an LED, a data line, and voltage rails. The entire strip is built on a flexible PCB. The strips can be easily cut in two with scissors, to shorten the strip's length. This should be done in the allocated, coloured space between two segments, to avoid damaging the circuitry. The strips typically contain around 60 LEDs per meter. The strip can be soldered to pins on the Digispark to allow data to be sent to the LED PCB. For easier connection to a second Raspberry Pi, a second Digispark is soldered to the other end of the LED strip, but not such that this Digispark can direct the LED strip. This second Digispark is affectionately called the "Dummy".

The ATtiny receives a data stream from the Raspberry Pi that determines the parameters of the power flow. This data stream includes the animation speed (1 bytes), direction (boolean, 1 bit), and RGB colours (1 byte per colour). The LED strip therefore has 256 speeds (including zero) and $256^3$ colour combinations with which the power on a line can be visualized. The ATtiny communicates with the LED strip using the Adafruit Neopixel library [17], available on Arduino IDE.

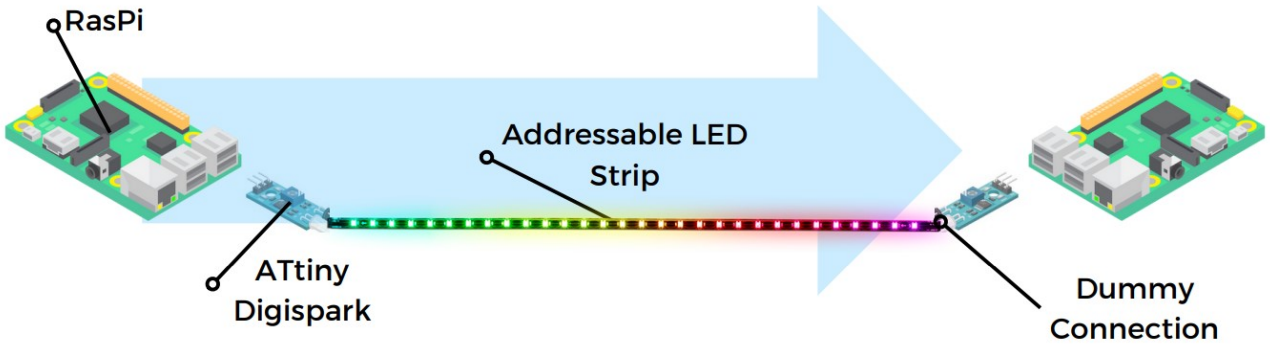



Figure 4.1: Illustration of two Raspberry Pi's connected by a LED strip, via two Digisparks. One of the Digisparks is a "Dummy", in that it does not drive the LED strip. It only serves as an easy connection to another Pi

The program on the ATtiny contains two loop functions for controlling the LED strip. The first function, "requestSpeedFromPi", receives data bytes whenever the serial USB is available. The data bytes represent animation data, sent in the order mentioned, which the function assigns to variables. The second function, "Animate_LEDs", takes the assigned variables and translates them to flow parameters for the LED strip.

### 4.1.1 LED Connection Model

In the table-top network, there will several LED strips visualizing different power flows. The Illuminator needs to allocate the power flow of a transmission line to its corresponding LED strip. This is done via an Illuminator model of type "LED_connection", which is written in the simulation yaml file. This model contains an IP address, and IP port, and a serial port, which collectively determine the Raspberry Pi that a LED strip is connected to. Each LED strip requires its own instance of an LED_connection model.

### 4.1.2 Communication Error Correction

When the ATtiny is writing to the LED strip, all interrupts are blocked. Because the serial USB communication is based on interrupts, data sent while data is being written to the LED strip is lost. This can't be fixed without changing how the libraries work. Instead, a damage control solution was implemented which will be referred to as "error correction". The information that the ATtiny writes to the LED strip comes directly from the Raspberry Pi, via a Python script. The data is sent to the ATtiny in bytes, in a fixed order: first, the animation speed, then direction, then red, then green and finally blue. When the ATtiny receives a byte, it sends a number back that represents the variable it writes that byte to. The Python script checks whether the received number matches its associated variable. In the event that they do not match, the entire data-set is sent again from the beginning, until the number matches the variable again. The ATtiny uses a case statement to cycle through the data stream. An additional error prevention method implemented is that, when the first byte is received by the ATtiny, it ceases writing to the LED strip, to further reduce errors with receiving the other bytes.

An alternative error prevention method that was explored was to represent each variable with two bytes, where the second byte contains the number of the variable the data should be written to. The advantage of this method is that the data stream is not interrupted by the Pi waiting for a number from the ATtiny, checking for a match, and then continuing. This reduces the number of disruptions in the data stream, and reduces delays during data transfer, since there is less error control occurring within the transfer time. The disadvantage of this method is that the data stream is now twice as large, increasing the points of failure during data reception. Overall, the increased risk of data loss with minimal error control was determined to outweigh the advantages of the streamlined transfer.

## 4.2 Determining connections between Pi's - the ATtiny's role

A key feature of the demonstrator is the Plug-and-Play dynamic. A mechanic implemented to achieve this dynamic is the reconfiguration of the simulation, using the hardware connections. The first step in this process is figuring out how the Pi's are connected to each other, to establish a network. Using some form of unique ID from the ATtiny's to pair up connected Pi's was a proposed method from the very beginning. The ATtiny's would each have an ID that, through communication between the Pi's and the ATtiny's, would be linked to another ATtiny's ID. This ID link would establish a connection pair between two Pi's. The implementation of this method went through several iterations, as the limitations of the ATtiny85 model became more apparent.

### 4.2.1 Unique Serial/Random IDs

The initial idea was to use unique serial IDs coded onto the ATtiny85's by the manufacturer. These are totally unique per unit, and could establish a link between two ATtiny's through UART communication. This method had to be revised soon into researching the ATtiny85, as it was quickly discovered that ATtiny85's actually do not have unique serial IDs. The revised method was to hard-code the ATtiny's with unique IDs manually via a sketch. Initially, these IDs would have been totally unique, similarly to serial IDs, by using a random number generator and the ATtiny's EEPROM. However, due to problems with the ATtiny's limited flash memory, the concept of totally unique IDs had to be abandoned, in favour of hard-coding IDs into the program. Figure 4.2 shows how this setup would have worked.

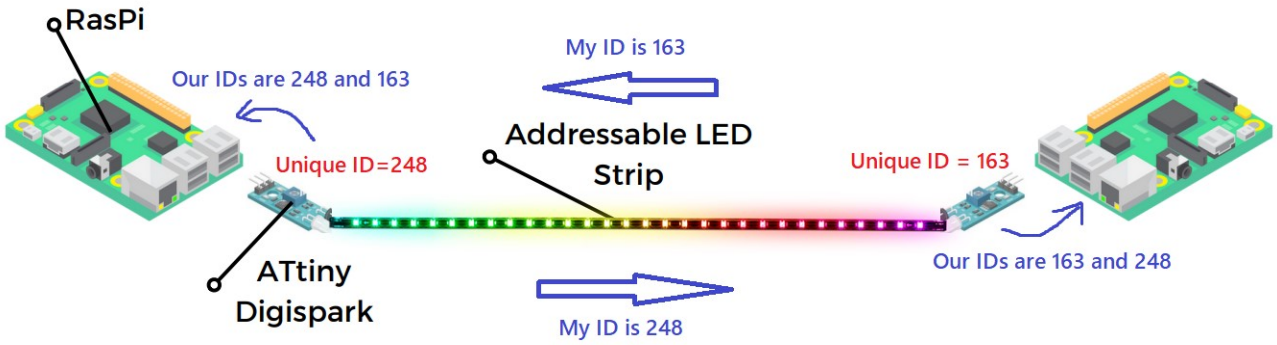



Figure 4.2: Illustration of how Unique IDs on the ATtiny's would pair up Raspberry Pi's, by communicating their IDs to both Pi's via UART

### 4.2.2 Limited Flash Memory

As mentioned earlier, the ATtiny85 has less memory than most of its Arduino counterparts. The Arduino IDE libraries required for serial USB communication and controlling the LED strips (DigiCDC.h and Adafruit_NeoPixel.h, respectively)

are very large. Collectively, these libraries take up $60 - 70\%$ of the available program storage space. This leaves little room for the actual code needed to control the LED strip and communicate with the Raspberry Pi. After including the code to generate unique IDs via the EEPROM, there simply wasn't adequate space left on the chip for UART communication, thus undermining the intended design of using unique IDs to pair up the ATtiny's. Increasing the available space, by reducing elements of the libraries, or writing assembly of our own to mimic their function, was briefly explored. However, time constraints lead to the conclusion that UART would not be a feasible route to take, thus a simpler implementation that didn't require UART was devised.

### 4.2.3 Hard-coded ID Pairs

The final iteration is far simpler in design. As mentioned in 4.1.2, the ATtiny processes the data stream from the Raspberry Pi through a case statement. In the last case, where the byte for blue is received, the ATtiny now also sends back a number that is labelled "ID". This ID is hard-coded to be the same for two ATtiny's on either end of a LED strip, while being different to each other pair of ATtiny's. Two Raspberry Pi's can determine if they are connected simply by comparing the IDs of their ATtiny's. Since the Raspberry Pi also needs to know if it can control the LED strip from the ATtiny, a sender or dummy byte is also sent back to the Raspberry Pi. This implementation sacrifices the more streamlined programming of large numbers of ATtiny's, that the random ID generation method would have allowed, for a straightforward, and noticeably more condensed method. An illustration is shown in Figure 4.3.

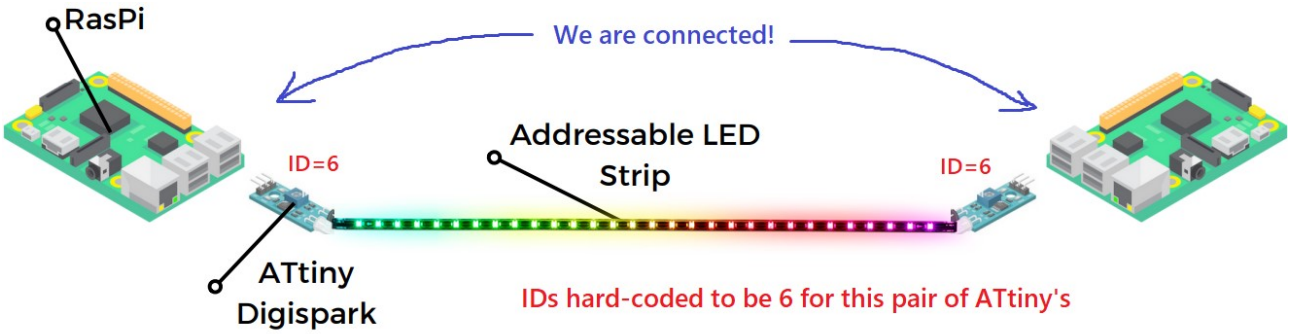



Figure 4.3: Illustration of how ID pairs, hard-coded onto the ATtiny pair, are used to pair up Raspberry Pi's

# Chapter 5

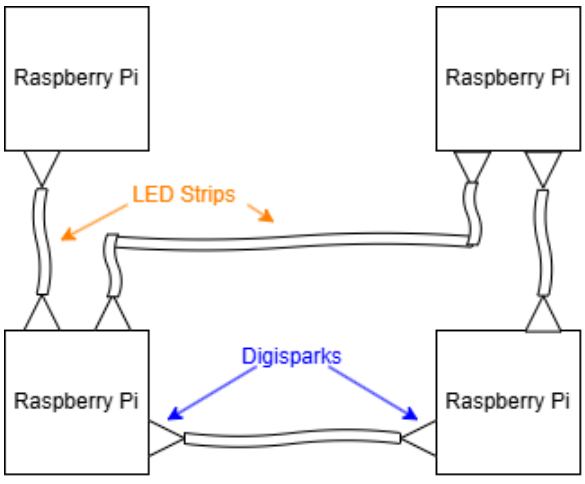
# Reconfiguring the Simulation Topology

The Python script for reconfiguring the topology was constructed in stages, where each subsequent stage could be tested without previous stages impacting the output of that stage. Reconfiguring line connections was addressed first, with several additions as the project progressed. Writing the LED connection models came when the line connections part was already fully functional. Before examining the features of the Python script itself, Section 5.1 explains some of the terminology used in this Chapter, and highlights the simulation's current drawbacks, in the context of the desired Plug-and-Play dynamics.

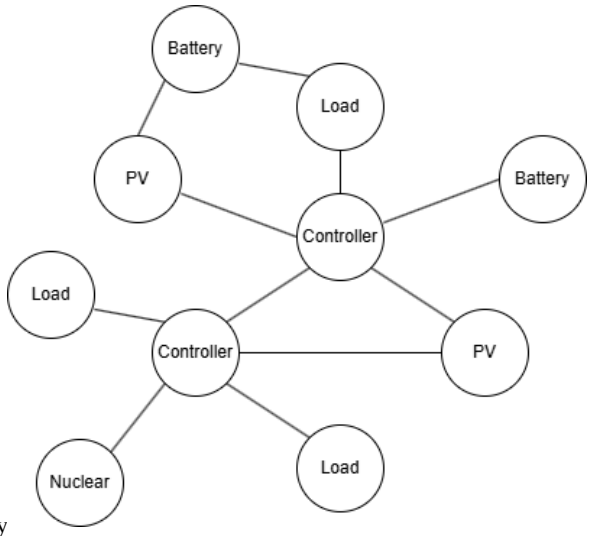
## 5.1 The Topology

### 5.1.1 Physical and Virtual Networks

The physical network is the table-top network, which is made up of various nodes and branches that represent an energy grid. Each node is a Raspberry Pi, and each branch is an LED strip, connected to a Digispark on either end, the Digispark in turn being connected to their Pi's via USB. The Pi's also have WiFi connection and a power cable. The virtual network, on the other hand, instead shows each node as an Illuminator model - Wind, PV Solar, a Battery, etc. - and each branch as a transmission line, with a reactance and a maximum capacity.




(a) Example Graph of a Physical, Table-Top Network of Raspberry Pi's, Digisparks, and LED Strips




(b) Example Graph of a Virtual Network of Energy Models

### 5.1.2 The Static Simulation YAML File

The Illuminator simulation tool starts a simulation by reading information out of a YAML file. YAML is a human-readable data serialization language, often used in configuration and for storing data. This yaml file describes four key elements of the simulation: the scenario, the models, the connections, and the inputs/outputs to monitor. Since YAML is human-readable, it is common for information in the YAML file to be typed in manually.

In pursuit of the desired Plug-and-Play setup, a huge limitation of the simulation yaml file is that it is static. The information is typed in manually and cannot easily be changed on-the-go. The simulation file is also only read before a simulation, never during. This means that, whenever a change is made to the physical connections in the physical

network, the simulation has to be stopped, so that the yaml file's topology can be adjusted, by hand. This process is slow, unintuitive, and prone to error. The remainder of this Chapter discusses the steps taken to automate this process.

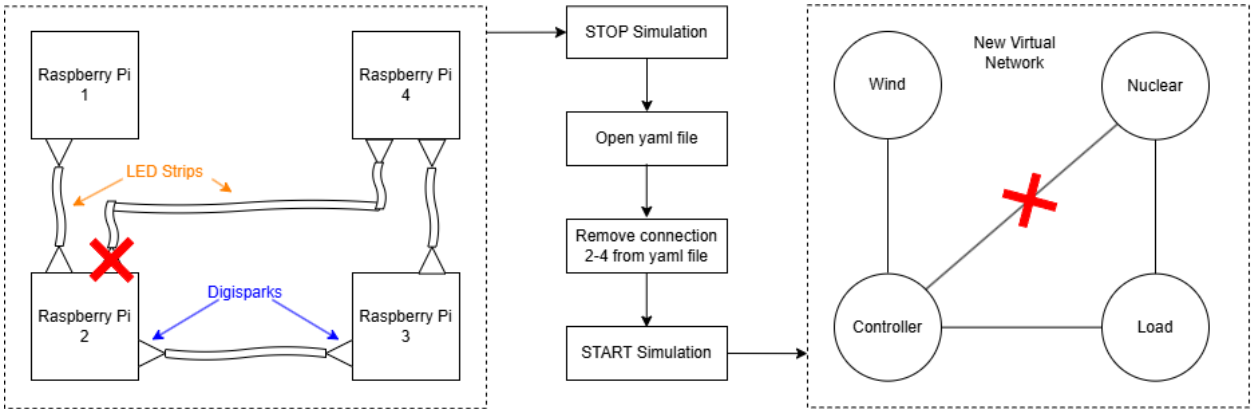



Figure 5.2: Flow Chart showing an example of how a Static Simulation's Virtual Network is adjusted, when a connection is removed from the Physical Network. Each step has to be done manually.

## 5.2 Automated Reconfiguration of Line Connections

Reconfiguring the simulation topology automatically is done via scripts, programmed onto the Master Pi. Like many programs that make up the Illuminator software, these scripts are written in Python. The program needs to determine the new topology after a physical connection has been changed, which the program then needs to translate to a set of valid yaml connections, that it writes to the simulation yaml file. From there, the simulation can be started again with the updated topology. Communicating with the yaml file is done using the PyYAML package. PyYAML can open a yaml file to either read or write to it. Writing to a yaml file wipes everything originally in the file. Adding to a yaml file is done by first reading the file, saving its data to a variable, then writing that data variable straight to the file again, along with the additional data. Writing to a yaml file also removes blank lines and certain indentations that a human may have used to make the file easier to read. This can be avoided by writing to a different file than the one crafted by hand, and using that different file as input for the program. This technique is used in all programs mentioned in this Chapter. Most importantly, writing to a yaml file always writes to the bottom line of the file.

The process of reconfiguring line connections started with examining what PyYAML was capable of: extracting data from a yaml file; writing to a yaml file in a certain format; removing and modifying data in a yaml file, etc. From here, a simple function was written that could write data to a yaml file in the form that the connections key expects. This revealed how the list of dictionaries of connections in the Python script would have to be formatted. The script's task was to take a certain input "Network", and translate them to yaml connections. Testing the script was therefore quite simple: write the script's output to a yaml file, and verify that the connections are correct. The script was also tested with several fixed "Networks", which tested certain properties such as skipping ID values with no connection, and reading the network out of order.

Various elements of determining the new topology have already been covered. Chapter 4 discussed how ID pairs, coded onto the ATtiny's, are used to determine how the Pi's are connected to each other; this Chapter also discussed the LED_connection model in the yaml file. Allocating the correct IP address, IP port, and serial port for each instance of an LED connection will need to be automatically written under the models key of the yaml file. This aspect of the physical topology is independent of the line connections, however their combined implementation places additional restrictions on the layout of the yaml file. This is explained in-depth in Section 5.4.

### 5.2.1 Initial Implementation

The initial implementation of the reconfiguration connected the Illuminator models directly. However, the way the connections are written presented a major threat to the program's scalability. Each model has several inputs and outputs - a csv file, a generated power, an soc, a load, etc. - which each have suffixes. As an example, the output of a wind model with the name "Wind1" is "Wind1.wind.gen", which might be connected to a "Controller1" with input port "Controller1.wind.gen". But "Controller1" may also have a load.dem port, an soc port, a pv.gen port, and several more. Automating the process of writing these connections requires if statement checks for every valid pair of models, and there are dozens of Illuminator models, each with their own suffixes for port names. The conclusion drawn was that this implementation was feasible, but was not well suited to extendability. While discussing integration with other aspects of the project, a more sophisticated implementation was devised.

### 5.2.2 Introducing Stations

The revised implementation was originally proposed for the purposes of power flow analysis, and its additional advantages in the context of reconfiguration led to the idea quickly being adopted.

The network is instead reconstructed to include Stations. A Station is a new Illuminator model, that represents a distribution point. Stations can only connect to other Stations, and to controllable peripherals (fossil generators, batteries, etc.). Any non-Station models are only connected indirectly, via Stations. Every Raspberry Pi must have, bare minimum, a Station. Any other models that a Pi is running are considered connected to that Pi's Station only. The verb "considered" is used very deliberately here: there is no physical connection between the models and the Stations, and only controllable peripherals have a simple connection to a Station written in the simulation's setup files. The models are only fully "considered" connected in the power flow analysis module.

In analysis, the virtual network is reduced to a network that only has Stations as nodes, where each node has a cumulative power generation and power load, based on the other models running on that Pi. Figure 5.3 demonstrates this setup. The Digispark connections and LED strips are excluded from the diagram.

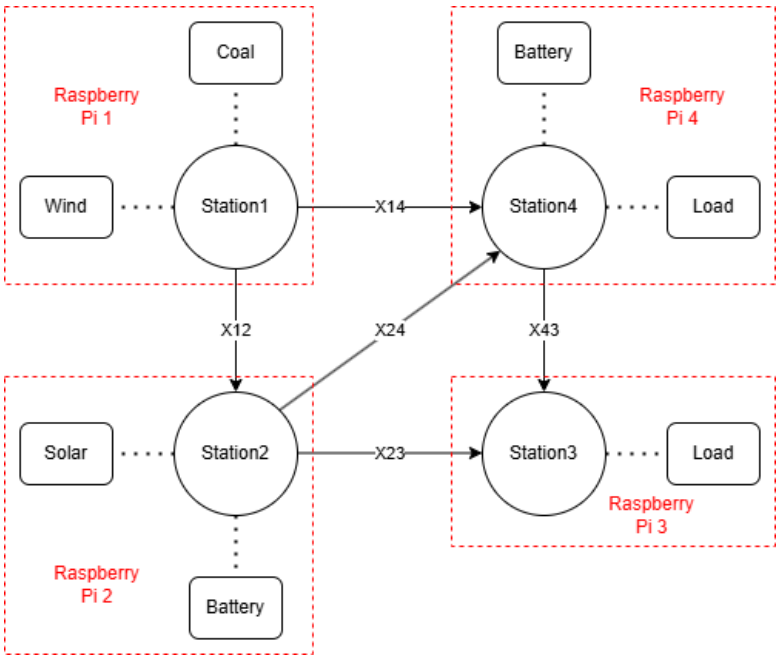



Figure 5.3: Simplified Schematic of the Nodes and Branches in a Virtual Network that uses Stations

The advantage of a network that only considers Stations is that, in the event that the topology is changed, only connections between Stations need to be considered. The generator and consumer models are statically connected. This is massively beneficial for writing connections in the yaml file: as mentioned, connecting Illuminator models directly is incredibly sensitive to the suffixes of each model's inputs and outputs. Writing connections between Stations, by comparison, is very simple, since the Stations only connect to each other via transmit and receive states. What follows is a significant reduction in the number of connection cases that the reconfiguration program has to consider.

### 5.2.3 ATtiny Sender/Receiver Status

With the introduction of Stations comes the addition of the "Sender" or "Receiver" status, sent from an ATtiny to its connected Pi. This status indicates whether an ATtiny is the LED controller or the dummy in a given ATtiny pair. The "Sender" status indicates the Station, and subsequently the Pi, from which the animation characteristics are sent to the LED strip. This follows from the transmit/receive connection between Stations: having the transmit Station also be the Station that directs the LED strip is logical and intuitive.

### 5.2.4 Line IDs

Another addition to the reconfiguration, also introduced due to Stations, is the line ID. The line ID is a connection in the yaml file that the engine uses to further draw the topology, for use in power analysis. The line ID's value is copied directly from the ID of the ATtiny pair.

## 5.3 Line Connection Implementation

Figure 5.4 shows a flow chart of each step the Python script takes to determine the new line connections.

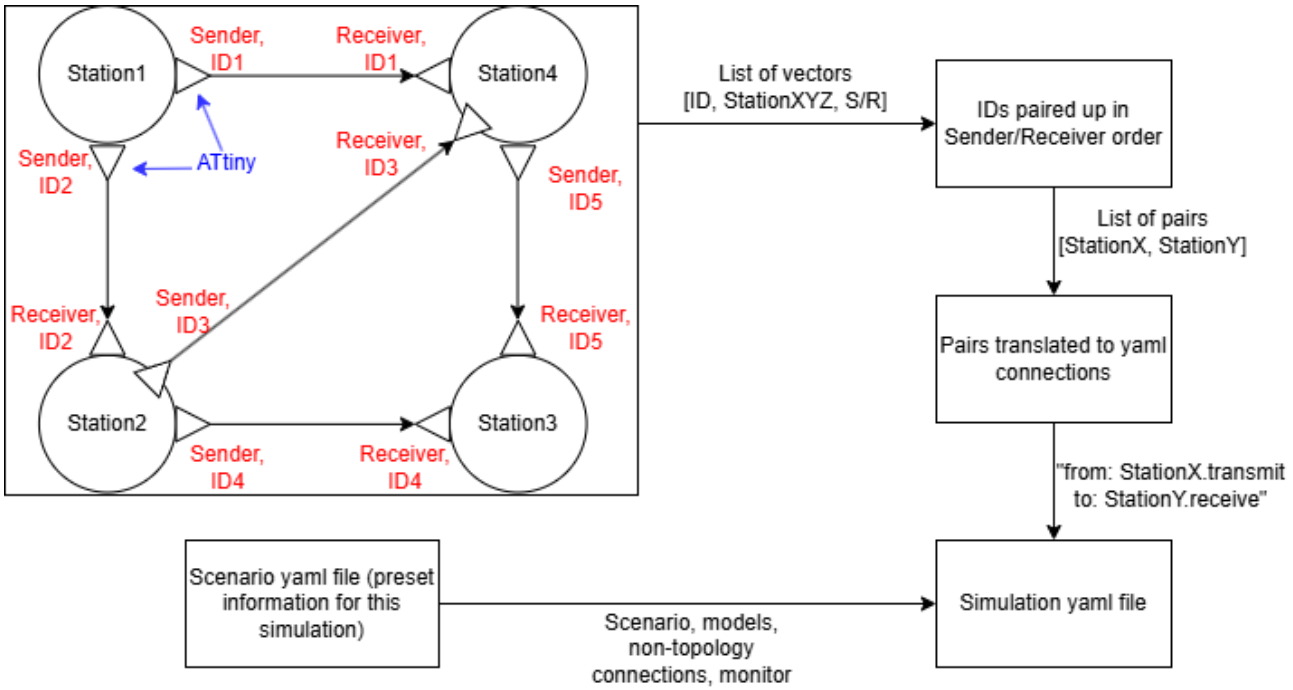



Figure 5.4: Flow chart showing how the Line Connections in the yaml file are reconfigured using the physical components

This implementation places a restriction on the yaml file's layout: the connections key must be the final key in the file before the reconfigured connections are written to the file. This is because the Python script writes the connections to the yaml file using the "yaml.dump(data)" function, which always adds the data to the end of the file. Writing it this way is easier than writing the data to a specific position or key in the yaml file, by a significant margin. The following Section is dedicated to exploring the LED connection model, another aspect of the physical topology, which adds an additional restriction to the yaml file's layout.

## 5.4 Allocating LED connection models

As discussed in Section 4.1, the Raspberry Pi's are responsible for sending the animation data that the ATtiny's pass on to the LED strips. Sending the animation data is done via an Illuminator model type called "LED_connection". This model directs the simulation in sending power from the power flow analysis module to the correct Raspberry Pi, and subsequently the correct LED strip. Each LED strip requires its own unique instance of an LED connection model. The IP address, IP port, and serial port of the Raspberry Pi piloting the LED strip have to be assigned to the respective LED connection model. This is dependent on the physical network, and therefore needs to be automatically adjusted alongside the connections already discussed.

The information for writing the LED connection models is compiled in a list of vectors, called "LED_portmap". Each vector in the list has the IP address, IP port, and serial port, of a given Pi. For each vector in the list, a function in the Python script writes a dictionary that describes an LED connection model with the parameters of that vector. The full list of dictionaries is written to the yaml file under models. Similarly to the line connections, this function is easy to test, by reading the output yaml file and verifying that the models are written exactly as intended.

As with writing the connections, the LED connection models are written to the yaml file using the dump function. The alternative is to modify pre-written LED connection models in the yaml file. Locating a specific key in a yaml file is trivial, however locating headers under a that key, or even headers under headers, is much trickier. The PyYAML package is not designed for this purpose. Text processing tools, such as awk or gawk, were considered, though the investigation and modifications required to allow the use of these tools, most notably the tools only being functional on Linux, was not feasible in the given time frame.

Only using the PyYAML package results in complications, due to the restriction placed on the yaml file's structure by the reconfiguration process: the LED connection models need to be written under the models key, for which the models key has to come last in the file, while the new line connections have to be written under the connections key, for which the connections key has to come last in the file. The solution to this conflict is to remove one of the keys part-way through the process. This is done using the pop function. The static connections in the setup yaml file are read and copied to the

topology variable in the script, and the connections key is popped, leaving the models key as the last key. The data from the setup yaml file, with the connections key popped, is written to the simulation yaml file. The LED connection models are drawn up using IP addresses and ports collected from the Pi's, and also written to the simulation file. The reconfigured topology, which now contains the static connections copied over from the setup files, alongside the new line connections, is written to the simulation file last. Figures 5.5, 5.6, and 5.7 show the how the flow chart in Figure 5.4 is adjusted to include the LED connection models. The flow chart has been separated into three for visual clarity. To summarize the changes, the IP addresses and serial ports are now part of the data collected. These are used to compile a portmap of LED connection models, as well as writing Station-to-LED connections, to direct power flows to the correct LED strip. When writing everything to the final simulation yaml file, the scenario yaml file's static connections are now directed to the connection list in the Python script, rather than directly to the simulation yaml file. The LED portmap is written under the models key, by first popping the connections key altogether. The full connection set is written over last.

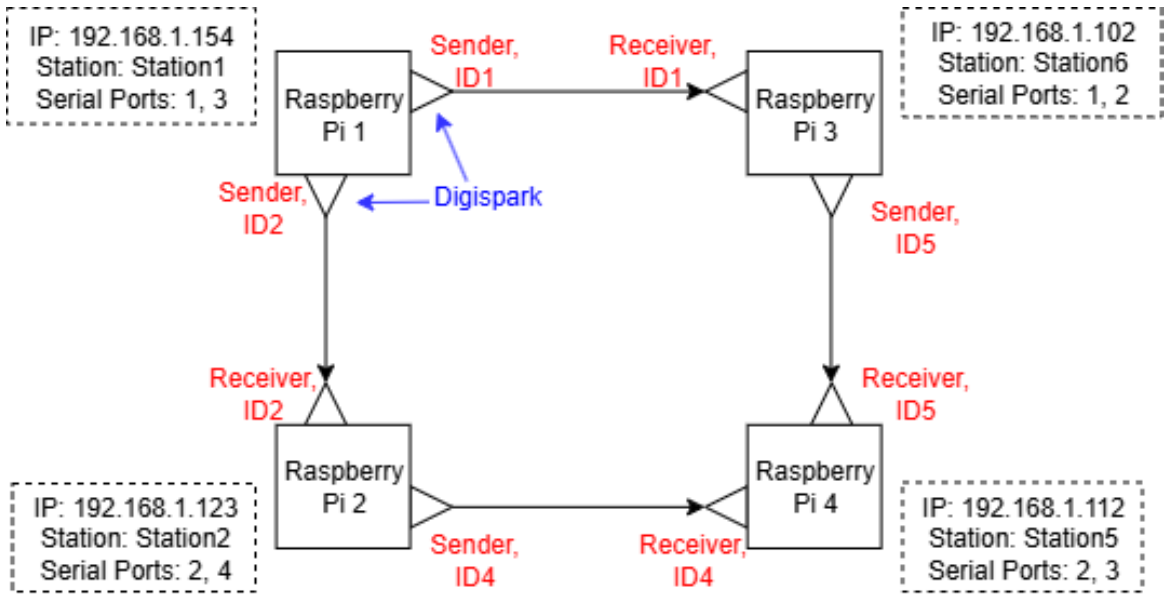



Figure 5.5: Diagram of a Physical Network of Four Raspberry Pi's, with all the Relevant Data that is collected shown. This data includes each Pi's IP address, Station, and active serial ports, as well as the Digisparks' IDs and Sender/Receiver status

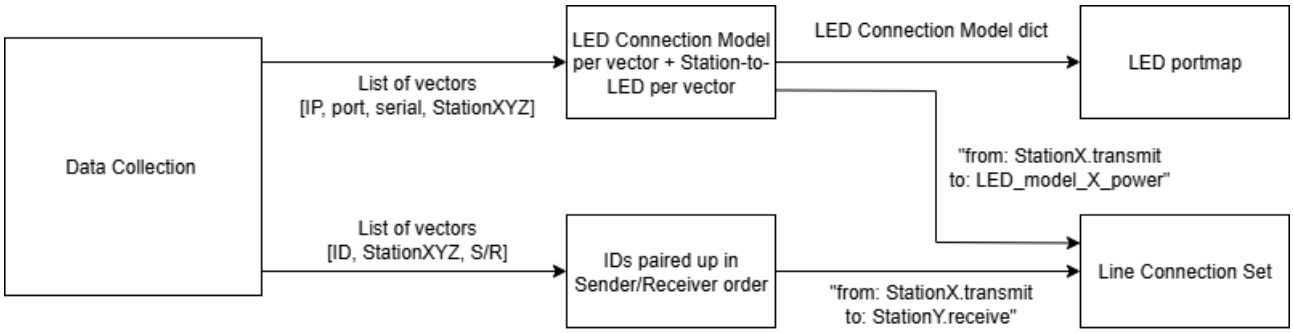



Figure 5.6: Flow Chart showing how the Data collected from the Physical Network is compiled into an LED portmap of LED Connection Models, and a list of Station-to-Station connections and Station-to-LED connections

## 5.5 Double Simulation

The master Pi requires information from the hardware components to reconfigure the simulation. This information is gathered by the slave Pi's and relayed to the master. Naturally, a communication link between the slaves and the master is required for this. The slaves are started using an SSH connection. Then the communication link is established when the simulation on the master is started. Before the master simulation is activated, the yaml file is read. When the simulation is terminated, the links between master and slave are too. This order of operations is problematic when looking to reconfigure the yaml file: reconfiguration requires the slaves to have a communication link with the master, to relay the hardware information.

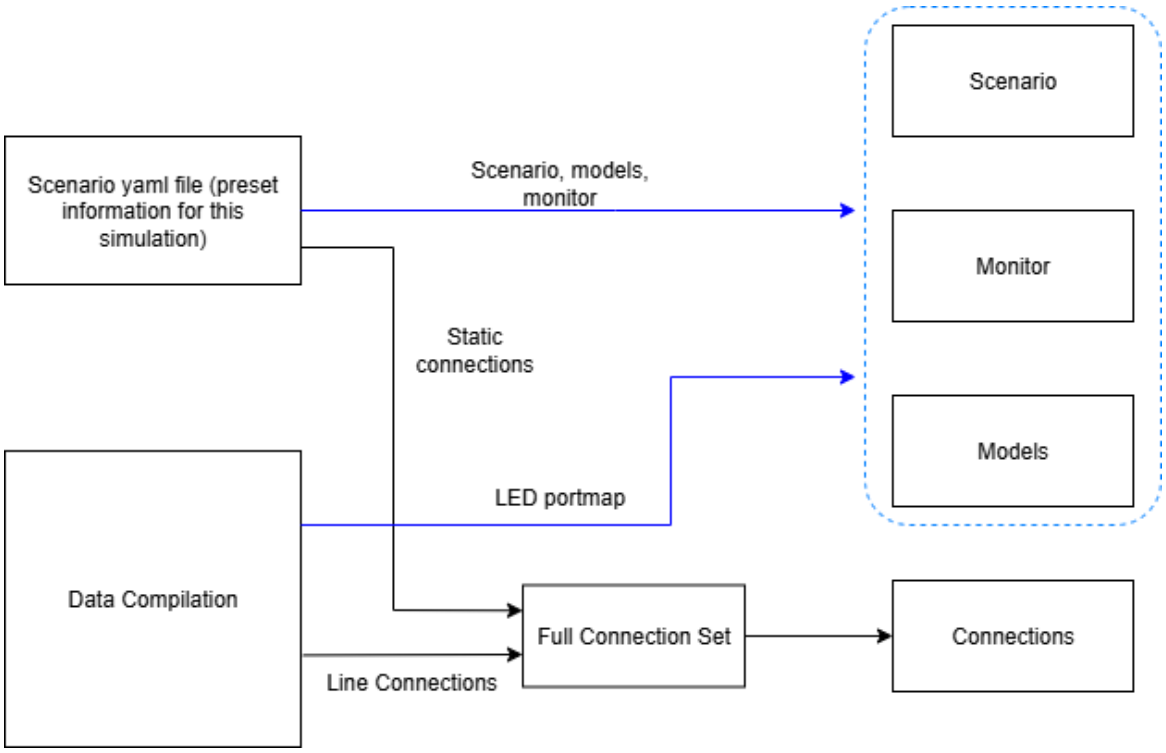



Figure 5.7: Flow Chart showing how the Data compiled is written to the simulation yaml file, alongside the scenario yaml file. Most notably, the static connections of the scenario yaml are written to the Line Connections List. The LED portmap is written to the Models Key, which requires temporary removal of the Connections Key.

The solution to this problem is a double simulation. There are two simulations that can be activated, each with their own yaml file. One of the simulations is the existing Illuminator simulation. Its yaml file has the aforementioned scenario, models, connections, etc. The other simulation is used for reconfiguration. Its yaml file has a model that is run on each slave, which has the slave gather information about its connections, its IP address, and so on. The master runs a model that takes the information from the slaves and reconfigures the topology. The yaml file the topology is written to is the same yaml file that the Illuminator simulation reads.

When a simulation procedure is initiated, the reconfiguration simulation is activated first. This simulation runs until the reconfiguration is completed. The reconfiguration is added to an unfinished yaml file, that describes the scenario, monitor, models, and static connections. The reconfiguration simulation then terminates, and the Illuminator simulation activates, reading the now completed yaml file as its input. The Illuminator simulation continues running until a connection is changed. The simulation detects a change via the slave Pi's. On each time step, each slave gathers its connection data and compares it to its connection data at the simulation's start. If there is a difference, the slave alerts the master and the simulation is terminated. The reconfiguration simulation is subsequently activated again. This process can be seen in Figure 5.8.

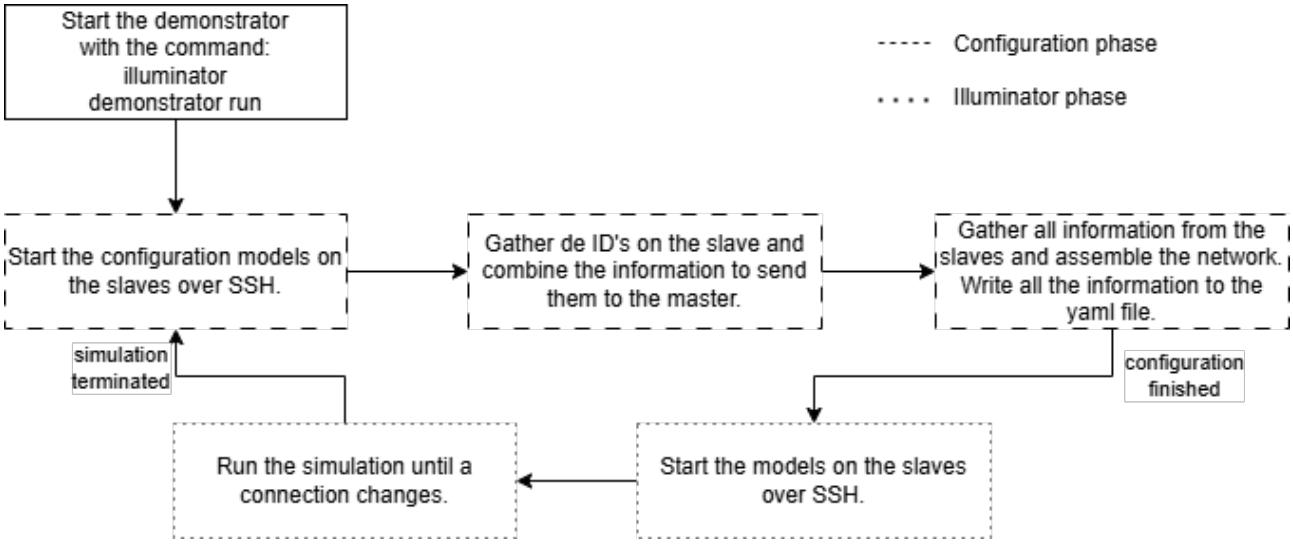



Figure 5.8: Flow chart of the double simulation

# Chapter 6

# Testing Process

## 6.1 ATtiny and the LED strip

Testing the ATtiny code via Arduino IDE is difficult. The ATtiny85 lacks serial hardware communication, and the serial USB communication is not compatible with the Serial Monitor built in to Arduino IDE. Testing the program is therefore not as simple as monitoring the variables as the program runs. Testing the ATtiny program is done via a Raspberry Pi instead. The Raspberry Pi can send and receive information to and from the ATtiny, and perform checks on that information to verify that the ATtiny is performing as expected.

As mentioned in Section 4.1, the ATtiny program has two loop functions: one gathers the animation data from the Raspberry Pi and assigns them to variables, the other translates the animation data and sends it to the LED strip. The function that communicates with the LED strip was not adjusted during this project, and as such did not undergo direct testing. The data gathering function was first tested by having a Raspberry Pi send a single data byte to the ATtiny, wait for the value to be sent back, then send another single data byte, and so on. The next test had the Pi send a full stream of animation data, and verifying that the animation variables were assigned correctly at each step; verification includes checking both the order and synchronization of the variable assignment. This test exposed synchronization errors between the Pi and the ATtiny, which required error control and correction. This was discussed in Subsection 4.1.2.

Even with error correction measures in place, the error rate of the ATtiny was observed to increase with the request rate. A series of tests were conducted, to examine whether the relationship between request rate and error rate was random. The ATtiny was sent 100 read/write requests in quick succession. For each test the delay between requests was increased. The error threshold was set to one second. That is, if a request took longer than one second, it was considered an error. Figure 6.1 shows the average number of errors for each measured delay between requests. From the graph, the error rate appears to be random. More notably, however, is that the error rate is very high, consistently above $40\%$, except for the 0 seconds delay. The delays of 0.5, 1 and 5 seconds have a better error rate than those of 2, 3 and 4 seconds, as those are lower. Although they are lower, the difference doesn't appear to have a clear relation. The 0 seconds delay is considerably lower than the other delays. This can probably be explained by the fact that it then sends the data so fast that the ATtiny isn't able to write the LED strip as it's continuously receiving new data. Bearing in mind that this test is with error correction implemented. The error rate without it would often stall the test completely. This confirms the need for error correction.

## 6.2 Testing the Reconfiguration Simulation

This Section will go in-depth on how the limits of the cluster setup was tested. This was only done for the Reconfiguration Simulation, because the reconfiguration and regular simulation are mostly the same. These tests will not be done on a Raspberry Pi, because it will start freezing when at least 5 models are run on it. Because Illuminator can also be run on a regular computer with Linux on it, the tests are done on a regular computer. It is desired to know how long it takes to reconfigure on the Raspberry Pi's, that's why that will be estimated by measuring the time of a smaller set on the Raspberry Pi and the time on the PC.

### 6.2.1 Methodology

Testing the reconfiguration simulation is done on a PC. This PC is powerful enough to run up to 100 models consistently. The models are run as an external model by setting the IP address to "127.0.0.1", which is the IP address for connecting to itself. This starts a different process with the model running in it. Each model is assigned a different port number, starting at 5000 and ending at 5XXX, where XXX is the number of models. The number of models is limited by the SSH program, because this program has a limited number of connections it can handle. Also the PC hardware eventually limits the amount of models that can be run. It takes time to set up that many models on the slaves, so a delay is set between

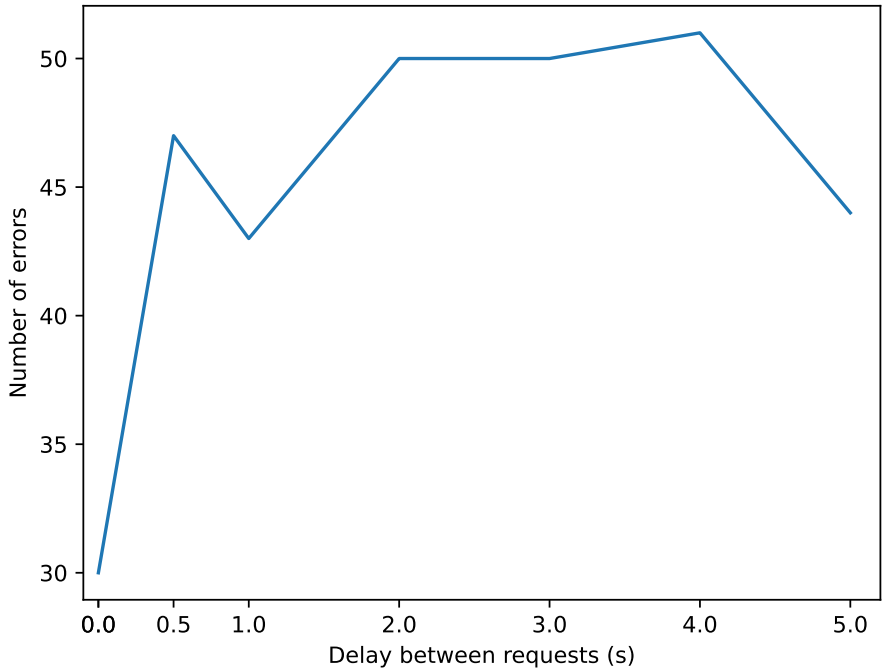



Figure 6.1: Graph showing the number of ATtiny ID requests, out of 100, that experienced an error. The delay time between requests is varied per test

starting the slaves and starting the master. There is a minimum delay time required, depending on the number of models being run, so that all slaves are started before the master is. The delay time chosen for our simulations is slightly higher than this minimum, as an extra precaution.

Figure 6.2 shows the delay times chosen for simulations of varying size. As shown, there are significant increases to the delay time at simulations of 40 and 70 models, and above. This is due to the simulation becoming unexpectedly more unreliable at and beyond these sizes, requiring a sharp increase to the delay time to ensure all the slaves successfully start before the master does. The simulation was also observed to very occasionally crash for 20 models and above. When the delay time was set to 30 seconds or higher, no crashes were observed. Despite this, a delay time below this margin was chosen for simulations with less than 70 models. The frequency of crashes within this range of sizes was low enough to justify reducing the total run time of the simulation, in the majority of cases where a crash didn't occur.

Increasing the simulation size also means the master takes longer to create and reconfigure a topology. The number of models increases the number of nodes in the network, and thus increases the time it takes to write the topology. Total run time is a parameter of the simulation that can indicate the severity of this added delay. The total run time indicates the time taken to process everything that occurs during the reconfiguration simulation. The total run time is measured from the SSH files starting, to the topology being written to the yaml file. It is known from Figure 6.2 what the contribution of the SSH program is to the delay, for a given simulation size. The delay added for actually reconfiguring the topology, aptly named the topology reconfiguration time, can be extrapolated by subtracting the SSH program's delay from the total run time.

Since the Raspberry Pi isn't capable of simulating a lot of models at once, these tests couldn't be done on a Raspberry Pi. The time can be estimated however. A test has been done on the Raspberry Pi with 3 models where this time the whole reconfiguration is done. It has in the reconfiguration phase 3 models running and in the normal simulation phase it has 3 models running on the slave processes and 3 on the master process. It is assumed that the total time taken is linearly connected to the topology reconfiguration time. This means that by calculating the factor of time increase at 3 models, the other times taken can be estimated. Therefore, the test on the PC has also been done with 3 models. With all this, the total reconfiguration time can be estimated from exiting the original simulation until the new simulation.

### 6.2.2 Test Results

Figures 6.3 and 6.4 show lines graphs of the total run time and the topology reconfiguration time for varying simulation sizes, up to simulations with 100 models.

Figure 6.3 shows that the reconfiguration simulation takes longer to complete, as more models are added to the simulation. This is expected, since each model contributes to both start-up time and execution time. This graph is non-

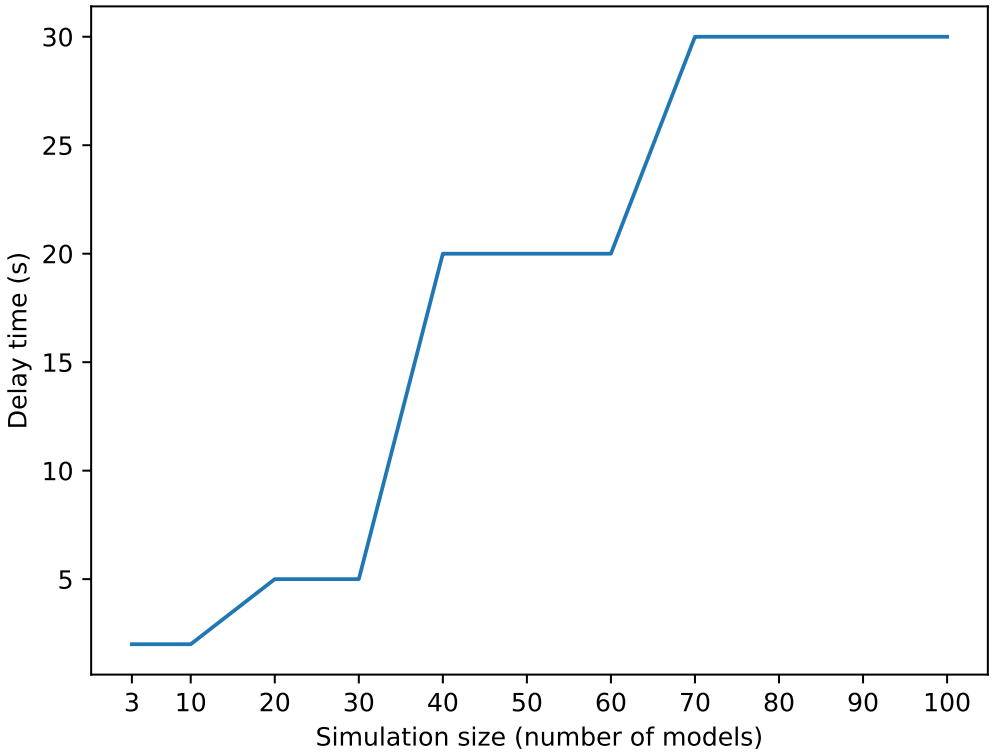



Figure 6.2: Line Graph showing the chosen delay time between starting the Slaves and starting the Master, for varying size simulations. Times chosen for the PC

linear, because the chosen delay time of the SSH program is also non-linear.

To determine the scaling of the topology reconfiguration time, Figure 6.2 is subtracted from this graph, resulting in Figure 6.4. It can be interpreted from Figure 6.4 that the topology reconfiguration time scales linearly with the simulation size. This suggests that the increased reconfiguration time is largely due to the addition of an extra node. How the added node connects to existing nodes is less significant.

It was found that the Raspberry Pi took $38.67$ seconds to restart the simulation with 3 models. With a 15 second correction for the delays between activating the shell files and starting the actual simulation, the total active processing time on the Raspberry Pi is $23.67$ seconds. On the PC, the reconfiguration phase took $2.44$ seconds for the same amount of models. Corrected for the 2 second delay between the shell files and starting the simulation, the active processing time is $0.44$ seconds. By calculating the factor in equation 6.1, the total processing time can be estimated. These results can be found in Figure 6.5. What can be seen from this graph is that, with about 20 models running, the reconfiguration time will be about 100 seconds. It should be mentioned again that this is an estimation, so real world examples may deviate, due to factors such as connection strength, number of models and other processes running on the Raspberry Pi.

$$timefactor = \frac{23.67}{0.44} = 53.80 \tag{6.1}$$

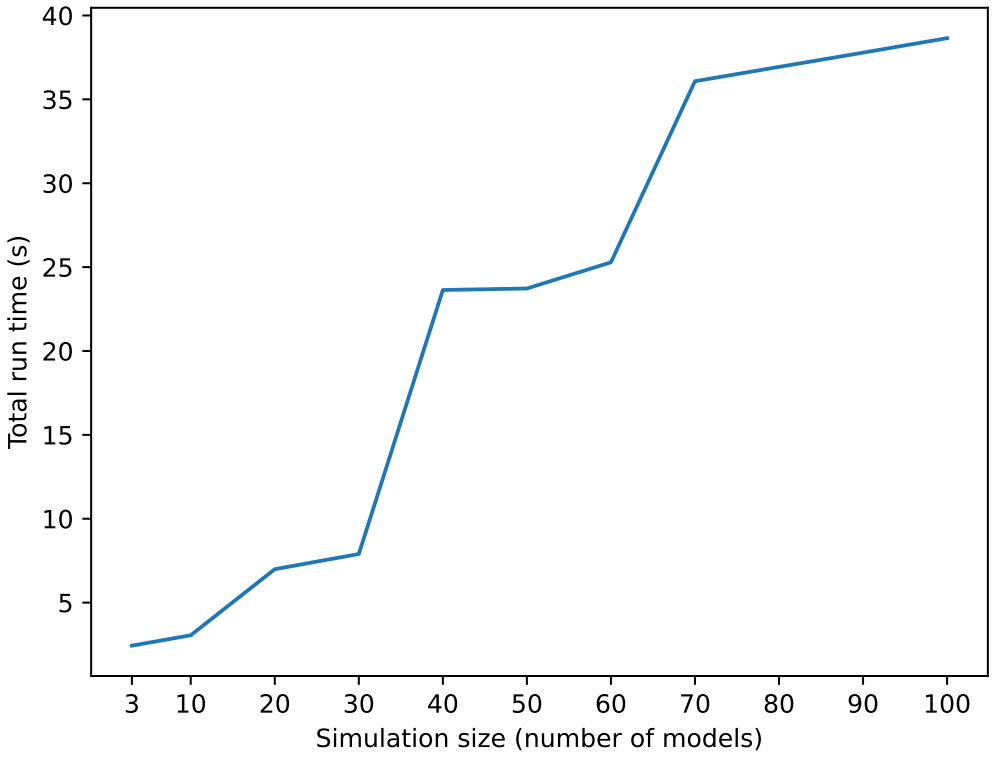



Figure 6.3: Line Graph showing the Total Run Time of the Reconfiguration Simulation, for varying sizes

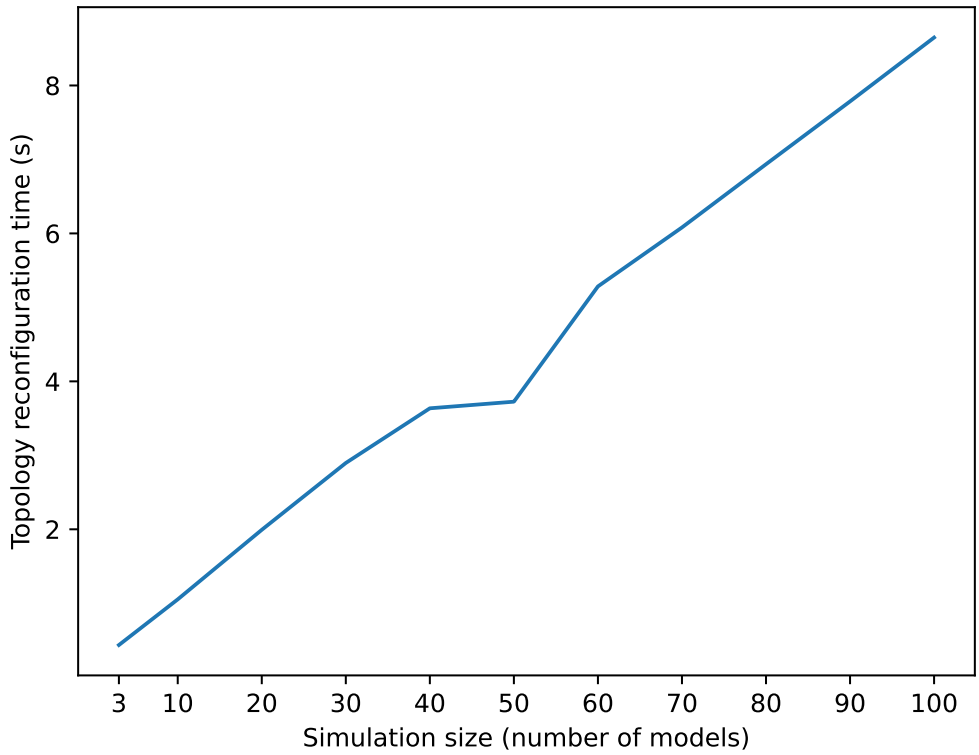



Figure 6.4: Line Graph showing the Topology Reconfiguration Time of the Reconfiguration Simulation, for varying sizes

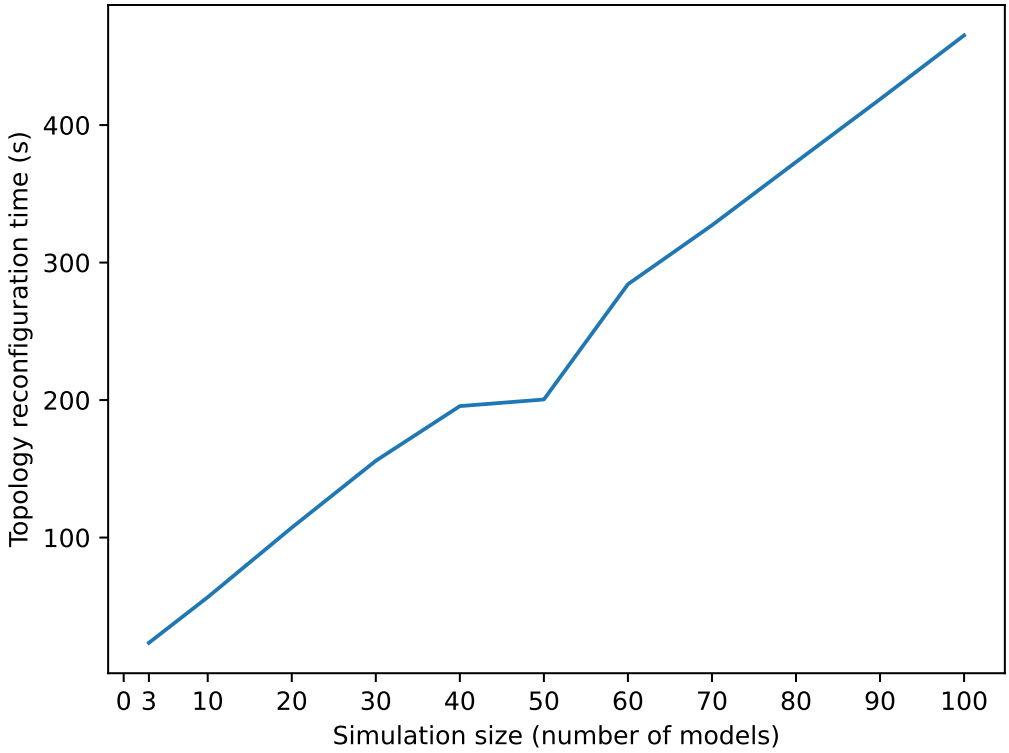



Figure 6.5: Line Graph showing the estimated total reconfiguration time of a simulation, for varying sizes

# Chapter 7

# Discussion and Conclusion

## 7.1 Determining connections using communication over the LED strip

### 7.1.1 UART

In the design discussed in section 4.2.3, the connection pairs are determined by hard-coding ID's. An alternative is to let two slaves communicate to each other over a UART connection. This design would have UART cables attached to the back of the LED strips. The ATtiny would act as a bridge between the USB and UART connection. Each ATtiny would be coded with a totally unique ID, which which it would send to its slave, and to a connected ATtiny, over UART. The slaves on either end of a LED strip would therefore obtain two unique IDs from their connected ATtiny pair. All the slaves would then send a list of all the IDs they received to master, which the master can use to pair up connected slaves, similarly to how it does now. The main advantage of this design would be that you don't have to edit the code for each ATtiny you program. Each ATtiny can contain the exact same program. You can even exchange information during simulation, if a peer-to-peer network is desired. The disadvantage of using UART is that it is more complex than the current design and would require more memory space on the ATtiny; Space that is currently unavailable on the ATtiny. Adding more physical hardware, in the form of UART cables, also complicates the table-top network, especially since the length of the LED strips would be bottle-necked by the length of the UART cable. Ultimately, the UART design was rejected in favour of the existing setup, which already used serial USB communication.

### 7.1.2 WS2812b

The WS2812b could also be used for communication between ATtiny's. This would utilize the data line on the LED strip, the same one used to send the RGB data to the LEDs. This type of communication would be uni-directional. It works by first sending the LED data and then sending extra bits over that same line which can be received by the ATtiny on the other end. The advantage of this method would be that no extra cables would be needed and no hard coded ID's would be necessary. The disadvantage of this method would be that determining the topology gets more complicated as it first needs to be determined which Raspberry Pi can send its ID to the other Pi. Also coding this into the ATtiny requires some additional complex coding, since there are no libraries which support receiving the WS2812b protocol, as this not a typical way of communication. Because coding the WS2812b is a very precise process and also involves writing in assembly, this was not possible in the given time frame and thus this option was rejected.

### 7.1.3 Master-Slave communication

Another alternative would be to use the Universal Serial Interface build in the ATtiny. This allows for asynchronous data transfer using protocols like I2C. The module requires a clock signal, so on the master end it requires some processor time, but since the I2C protocol also has a clock wire, the slave doesn't require processor time. When the data buffer is overflowed an interrupt is raised which indicates data has been transferred. This method can also be used to transfer ID's to the other end of the LED strip. This method was not chosen, as it would make the coding more complex which wasn't viable in the time frame given.

## 7.2 Alternatives ways to visualize Power Flows

An alternative for visualizing would be to show the power flows on a screen. The screen would display the energy system reflected by the hardware connections. It would look similar to a map used in a grid operating center. The screen can be made fully interactive, offering the advantage that you can change the capacity of certain lines during the simulation, or control the grid manually by digitally enabling and disabling certain lines. This would be alongside the interactivity that

the hardware connections provide. Without LED strips included, the Digisparks have the memory space to utilize UART communication, the potential of which was discussed prior in Section 7.1. With this in mind, even a real model of a power line can be created for some nice detail on the table. The power line model would even transport real power in the form of UART signals. The disadvantage of changing the setup this way, aside from adding a screen to the setup, is that the physical visualization of the energy system would be less directly linked to the table-top network. Ideally, all the visual elements would be grouped together, to avoid attention being divided between them.

## 7.3 Reconfiguration without Stopping

As discussed in Section 5.5, reconfiguring the simulation requires the master and slaves to have a communication link. It also requires the Illuminator simulation to re-read the yaml file, after the connections have been adjusted. With the current simulation process, this was not feasible, so a double simulation setup was implemented. This section will discuss an alternative method that was considered, and remained a possible option until fairly late into the project's life-span. It was ultimately rejected, in favour of the double simulation.

The alternative method saw major modifications made to the simulation process, which would allow the simulation to re-read its yaml file during simulation. This, in turn, would allow the simulation to reconfigure without needing to restart. For the purposes of Plug-and-Play, this is a major benefit. The drawbacks of this method are that the process of reconfiguring the topology, as well as the power flow analysis on the topology, have to run in each time step of the simulation. The slave Pi's also need to request IDs from the ATtiny's each step, which, as discussed in Section 6.1, is very prone to error. Despite these drawbacks, the upsides were appealing enough that there was significant discussion on whether to try this method instead of the double simulation. It was ultimately decided that this method would be too time-consuming for the scope of this project, due to the aforementioned modifications to how the simulation runs.

## 7.4 Using the Stations as Abstraction

In the current version, there is an Illuminator model that directs the power flow output of a given station to a given LED strip, with each LED strip having its own instance of this model. This can cause problems when there are multiple models running on the same Pi. The maximum is about 5, and if you already have a station and a normal model running, there is not a lot of models left for LED connections. Additionally, running extra models for each LED strip increases the processing time of the simulation, and introduces more points of failure.Since each Pi is guaranteed to have a station, there is the option to direct the power flows directly through the station, removing the LED connection model entirely. This would require adjustments to the station model, such that it can receive multiple power inputs and know which LED strip to direct each power to. Handling all the connections in one model complex and there was not enough time to implement it, though the potential is there.

## 7.5 Comparison to Competitors

The integration of a table-top network introduces hardware limitations to the simulation. Compared to its competitors, such as DIgSILENT and ETAP PS, the Illuminator simulation running on this table-top network will ultimately be slower and less accurate. However, the upside of having physical components cannot be under-stated. This simulation kit trades accuracy and speed for interactivity and visual aids. The kit is much more intuitive to use and interpret than its competitors that are exclusively software-based.

## 7.6 Conclusion

This thesis discussed the steps taken to create an interactive, Plug-and-Play energy system demonstrator using the Illuminator simulation tool, alongside hardware components and Python scripts. First, the motivation for the demonstrator was described. The transition to clean and renewable energy introduces intermittency and distributed power to today's energy systems. The technical knowledge and understanding of such systems is lacking. An interactive, intuitive simulator can help visually show how such a system works.

A state-of-the-art analysis was conducted on various components involved in this project, such as the Illuminator simulation tool itself, which uses Mosaik, a smart grid simulation tool, as its basis; The simulation is run on a cluster network of Raspberry Pi mini-computers, which utilize a slave-master dynamic. The Pi's are responsible for running Illuminator models and communicating information to and from other components; LED strips, directed by Digispark ATtiny85 micro-controllers, are used to visualize the flow of power between Pi's. The Digisparks communicate with the Pi's via serial USB, due to their lack of serial hardware. Their primary task is to translate the Pi's animation data about the power flow to the LED strip.

The LED strip has a Digispark on both ends, for easier connection to Pi's, with only one of the Digisparks actually directing the LED strip. Each pair of Digisparks is hard-coded with an ID, which they send to their connected Pi's. Whether there is a connection between two Raspberry Pi's can be determined by comparing these IDs. Error correction between the Pi and the ATtiny was implemented, because the communication with the Raspberry Pi is sometimes blocked by the LED strip code on the ATtiny, leading to synchronization errors.

Each Pi runs a set of energy models alongside a Station. Physical connections only correspond to connections between Stations, not energy models, to make writing the connections more scalable. Connections between Pi's are used by a Python script that reconfigures the topology of the simulation. The Illuminator simulation reads its scenario, models, connections, and monitored inputs and outputs off of a simulation yaml file. The Python script uses information from the Raspberry Pi's to modify the models and connections of the simulation yaml file. More specifically, connections between Stations are modified, and LED connection models are written for each LED strip in the network, using the IP address, IP port, and serial port of their respective Raspberry Pi.

A second simulation was constructed to perform the reconfiguration process, due to the Illuminator simulation deactivating the SSH program upon termination. The SSH program is required for the Pi's to communicate with the master computer. Whenever a connection in the table-top network is changed, the Illuminator simulation self-terminates, and the reconfiguration simulation starts up. When the reconfiguration is complete, it self-terminates, and the Illuminator simulation starts up again.

The process of testing each implemented feature, in isolation and when integrated with accompanying features, was discussed. This included examining the error rate for serial communication with the ATtiny. The error rate was found to be random, independent of the request rate. The algorithm for reconfiguring the topology was tested extensively in isolation, since its output was very specific and easy to verify. The reconfiguration simulation as a full entity was tested for different size simulations, to determine how its total run time scaled. The run time of starting the SSH program was found to scale non-linearly, with exceptional increases at simulations above 40 and 70 models. The topology reconfiguration time was found to scale linearly with size, indicating that the addition of an extra node contributes far more to the total run time than the relevant connections of that node do. It was estimated that the total reconfiguration time at 20 nodes is about 100 seconds. It scales linearly like the

Lastly, a few alternatives to the implementations described in this report were discussed. Rather than using Digisparks with LED strips, the power flows in the network could instead be visualized on a screen. The table-top network can then instead be connected with simpler cables. A screen provides an overview of the energy system, and can be programmed with interactivity, however it may struggle with clarity for very large systems. The possibility of using UART communication rather than serial USB would allow for unique IDs on the Digisparks, if the flash memory limitation of the Digispark can be overcome. Using a screen instead of LED strips would make this option more feasible. Significant modifications to the simulation process could allow for constant reconfiguration, removing the need for the simulation to terminate between configurations. These have some advantages over the designs used in this report, but were ultimately not pursued, though their potential in future developments is intriguing.

With this work done on the Illuminator, another unique feature has been added to the Illuminator project. This will create a more intuitive alternative for expensive simulation software. Visualizing energy systems using table-top components will help broaden people's understanding, and allow for more enlightened decision making.

# Bibliography


[1] T. Delft. (2025), [Online]. Available: https://github.com/Illuminator-team/Illuminator.

[2] B. D. Alje Vermeer. (2025), [Online]. Available: https://github.com/Kyr-updog/Illuminator-BAP-EE-2025/tree/cluster-BAP-GroupB.

[3] A. Fu, R. Saini, R. Koornneef, A. van der Meer, P. Palensky, and M. Cvetković, "The illuminator: An open source energy system integration development kit," in *2023 IEEE Belgrade PowerTech*, 2023, pp. 01–05. DOI: 10.1109/PowerTech55446.2023.10202816.

[4] T. Ylonen and C. Lonvick, "The secure shell (ssh) protocol architecture," Tech. Rep., 2006.

[5] S. Schütte, S. Scherfke, and M. Tröschel, "Mosaik: A framework for modular simulation of active components in smart grids," in *2011 IEEE First International Workshop on Smart Grid Modeling and Simulation (SGMS)*, 2011, pp. 55–60. DOI: 10.1109/SGMS.2011.6089027.

[6] S. Schütte, S. Scherfke, and M. Sonnenschein, "Mosaik-smart grid simulation api," *Proceedings of SMARTGREENS*, pp. 14–24, 2012.

[7] S. Lehnhoff, O. Nannen, S. Rohjans, *et al.*, "Exchangeability of power flow simulators in smart grid co-simulations with mosaik," in *2015 Workshop on Modeling and Simulation of Cyber-Physical Energy Systems (MSCPES)*, 2015, pp. 1–6. DOI: 10.1109/MSCPES.2015.7115410.

[8] M. Büscher, A. Claassen, M. Kube, *et al.*, "Integrated smart grid simulations for generic automation architectures with rt-lab and mosaik," in *2014 IEEE International Conference on Smart Grid Communications (SmartGridComm)*, 2014, pp. 194–199. DOI: 10.1109/SmartGridComm.2014.7007645.

[9] Raspberry Pi Ltd., *Raspberry pi hardware documentation*. [Online]. Available: https://www.raspberrypi.com/documentation/computers/raspberry-pi.html.

[10] Raspberry Pi Ltd., *Raspberry pi os documentation*. [Online]. Available: https://www.raspberrypi.com/documentation/computers/os.html.

[11] Adafruit. "Ws2812b, intelligent control led integrated light source." (2025), [Online]. Available: https://cdn-shop.adafruit.com/datasheets/WS2812B.pdf.

[12] Adafruit. "Adafruit neopixel library." (2025), [Online]. Available: https://adafruit.github.io/Adafruit_NeoPixel/html/index.html.

[13] JOY-IT. "Joy-it® digispark microcontroller." (2019), [Online]. Available: https://joy-it.net/files/files/Produkte/ARD-Digispark/ARD-Digispark-Manual.pdf.

[14] A. Corporation. "Atmel 8-bit avr microcontroller with 2/4/8k bytes in-system programmable flash." (2013), [Online]. Available: https://ww1.microchip.com/downloads/en/devicedoc/atmel-2586-avr-8-bit-microcontroller-attiny25-attiny45-attiny85_datasheet.pdf.

[15] Arduino. "Softwareserial library." (2015), [Online]. Available: https://docs.arduino.cc/learn/built-in-libraries/software-serial/.

[16] Digistump LLC. "Digistumparduino." (2015), [Online]. Available: https://github.com/digistump/DigistumpArduino.

[17] Adafruit. "Adafruit neopixel." (2015), [Online]. Available: https://docs.arduino.cc/libraries/adafruit-neopixel/.

[18] Arduino. "Micro." (2025), [Online]. Available: https://docs.arduino.cc/hardware/micro/#features.

[19] A. Corporation. "Atmega16u4/atmega32u4." (2015), [Online]. Available: https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7766-8-bit-AVR-ATmega16U4-32U4_Datasheet.pdf.

[20] S. Studio. "Seeeduino nano." (2023), [Online]. Available: https://wiki.seeedstudio.com/Seeeduino-Nano/.

[21] A. Corporation. “Atmega328p.” (2015), [Online]. Available: https://ww1.microchip.com/downloads/aemDocuments/documents/MCU08/ProductDocuments/DataSheets/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf.

[22] DIgSILENT. “Powerfactory.” (2025), [Online]. Available: https://www.digsilent.de/en/powerfactory.html.

[23] etap. “Etap power simulator.” (2025), [Online]. Available: https://etap.com/power-simulator.

[24] Siemens. “Pss®sincal electricity modules.” (2025), [Online]. Available: https://www.siemens.com/global/en/products/energy/grid-software/planning/pss-software/pss-sincal/pss-sincal-electricity.html.

# Appendix A

# Digispark Program

This is the program uploaded to the Digispark via USB. The program controls the LED strips using animation data from an external source, in this case a Raspberry Pi. The constants ”ID” and ”Dummy” should be modified for each pair of Digisparks that are connected by a LED strip. The ID needs to be the same for both of them, and unique to other pairs used in the setup. The Dummy is 1 for the Receiver, and 2 for the Sender. Refer to Section 4.1 and Subsection 5.2.3 for more information.

```
#include <Adafruit_NeoPixel.h>
#include "DigiCDC.h"
#include "EEPROM.h"

#define LED_PIN 0    // WS2812B Data Line on PB0 (Pin 0)
#define LED_COUNT 20 // Number of LEDs
#define ID 1 //the ID of the attiny
#define DUMMY 2 //1 is dummy, 2 is controller

Adafruit_NeoPixel strip(LED_COUNT, LED_PIN, NEO_GRB + NEO_KHZ800);

uint16_t animationSpeed = 0;  // Default speed, max 65535
uint8_t direction = 0;
uint8_t r = 10;
uint8_t g = 10;
uint8_t b = 10;
uint8_t i = 0, j = 1;

void setup() {
  SerialUSB.begin();  // Initialize USB Serial
  strip.begin();
  strip.show();

  //ID = EEPROM.read(0);
  //delay(2000);
  //SerialUSB.write(ID);
}

void loop() {
  requestSpeedFromPi();  // Ask Raspberry Pi for speed
  if (j == 1){
    Animate_LEDs(animationSpeed, r, g, b, direction);  // Run animation
  }
}

void requestSpeedFromPi() {
    if (SerialUSB.available()) {
      switch (j){
        case 1:
          animationSpeed = SerialUSB.read();
```

```
            SerialUSB.write(j);
            j++;
            break;
          case 2:
            direction = SerialUSB.read();
            SerialUSB.write(j);
            j++;
            break;
          case 3:
            r = SerialUSB.read();
            SerialUSB.write(j);
            j++;
            break;
          case 4:
            g = SerialUSB.read();
            SerialUSB.write(j);
            j++;
            break;
          case 5:
            b = SerialUSB.read();
            SerialUSB.write(j);
            j = 1;
            SerialUSB.write(ID);
            SerialUSB.write(DUMMY);
            break;
          default:
            j = 1;
            SerialUSB.write(6);
            break;
        }
      }
      else{
        SerialUSB.refresh();
      }
}

 // Cyan Flow Animation with dynamic speed
 void Animate_LEDs(int wait, uint8_t r, uint8_t g, uint8_t b, bool direction) {

  if (wait == 0){
    for (uint8_t pos=0; pos < LED_COUNT; pos++){
      strip.setPixelColor(pos, strip.Color(r, g, b));
    }
    strip.show();
    SerialUSB.delay(10);
    return;
  }


  int pos = 0;
  if (direction == 0){
    pos = LED_COUNT - i - 1;
    }
  else{
    pos = i;
  }
  strip.clear();
  strip.setPixelColor(pos, strip.Color(r, g, b));  // Cyan color
  strip.show();
  SerialUSB.delay(wait);
```

```
    i++;
    if (i > LED_COUNT){
      i = 0;
    }
}
```

# Appendix B

# Python Scripts

## B.1 Reconfiguration of the simulation yaml file

This script has a function that determines the LED connection models, and several functions related to writing the line connections. Each function is briefly explained. The script can be located in the GitHub under src/illuminator/models/-TopologyMaker/Dynamic_yaml_scenario_module.py

The function ”write_LED_portmaps” takes a list of vectors of the form ”[IP address, IP port, serial port]”, and writes LED connection models for each vector. Each vector corresponds to a LED strip and its pilot Raspberry Pi.

The function ”determine_connected_pairs” takes a list of vectors of the form ”[ID, StationX, Sender/Receiver]” and loops through ID values from 1 up to the highest ID in the list. For each ID, it pairs up Stations that share an ID. These pairs are added to a ”connected_pair_array”, arranged such that the Sender Station is always first.

The function ”def_write_topology” takes the connected pair array as input, and loops through each row in the array. It writes a dictionary for each row, that contains ”from : StationX.transmit”, ”to : StationY.receiver”, and ”line_id : ID”. This is compiled in a list of dictionaries. Before this function does all this, however, it first opens the scenario yaml file to read, copies the data under the ”connections” key to its topology list, and then pops the connections key. The remaining data is then dumped to the simulation yaml file, which is originally empty.

The last function, ”write_scenario_LEDs_and_connections”, takes the outputs of the other functions and writes it to the simulation yaml file. The order is crucial, with the simulation file’s scenario data coming first, then the LED connection models, and lastly the connections.

```
import yaml
import numpy as np

def write_LED_portmaps(LED_model):
    LED_model_array = np.asarray(LED_model, dtype=object)
    LED_portmap = []

    for i in np.arange(len(LED_model_array)):
        Pi_IP_address = LED_model_array[i, 0]
        ip_port = LED_model_array[i, 1]
        serial_port = LED_model_array[i, 2]

        LED_portmap.append({'name': f'LED_model_{i+1}', 'type': 'LED_connection'
            ,
                            'connect': {'ip': Pi_IP_address, 'port': ip_port},
                            'parameters': {'min_speed': 0, 'max_speed': 0.6, '
                               direction': 0, 'port': serial_port},
                            'input': {'speed': 5}
                            })
    return LED_portmap

def determine_connected_pairs(Network):            #this function creates an
   array of all Station pairs in S/R order

    highest_ID = np.max(np.array(np.array(Network)[:,0], dtype = int))  #
       determine highest ID value in the Network
```

```
    ID = np.arange(1, highest_ID + 1)                                   #create
        ID list with range 1 up to highest ID + 1
    connected_pair_array = np.empty((0, 3))                             #create
        a blank-by-2 array
    for i in np.arange(len(ID)):                                        #loop
        over every ID value
        connected_pair = []                                             #
            redefine connected_pair list before each ID value
        for model in np.arange(len(Network)):                           #loop
            over the Network array
            if Network[model][0] == ID[i]:                              #compare
                Network's 0th element to ID's ith element
                connected_pair.append(Network[model][1])                #add
                    Network's Station element to the connected_pair list
                connected_pair.append(Network[model][2])                #add
                    Network's Sender/Receiver element to list
                connected_pair.append(i+1)
        if (len(connected_pair) == 0):                                  #if this
            ID value had no connections, skips
            continue

        if (len(connected_pair) == 6):
            if ('Sender' in connected_pair[1]):
                connected_pair_array = np.append(connected_pair_array, [[
                    connected_pair[0], connected_pair[3], connected_pair[2]]],
                    axis=0)
                #add connected Station pair to next row in array in standard
                    order (for S/R)
            else:
                connected_pair_array = np.append(connected_pair_array, [[
                    connected_pair[3], connected_pair[0], connected_pair[2]]],
                    axis=0)
                #add connected Station pair to next row in array in reverse
                    order (for S/R)
    return connected_pair_array



def write_topology(connected_pair_array, key, filename, write_file):
    with open(f'{filename}.yaml', 'r') as f:        #opens a yaml file to read
        data = yaml.safe_load(f)                        #loads the yaml data in
            safe mode
        topology_list = (data[f'{key}'])                #copies everything under
            a given "key:" to a list
        data.pop(key)                                   #pops the "key:" and
            everything underneath
    with open(f'{write_file}.yaml', 'w') as file:   #opens a different yaml file
        to write to
        yaml.dump(data,file,sort_keys=False)            #writes the original
            yaml data, excluding the popped key, to said different yaml file
                                                    #the purpose of this
                                                        operation is to copy the
                                                        static connections to the
                                                         topology
                                                    #and then remove them in a
                                                        new intermediary yaml
                                                        file
                                                    #this leaves the models:
                                                        section at the bottom of
                                                        the file, for the
                                                        LED_portmapping
```

```
    from_model = ''
    to_model = ''
    line_id = ''

    for i in range(np.shape(connected_pair_array)[0]):
        if ('Station' in connected_pair_array[i, 0]) and ('Station' in
            connected_pair_array[i, 1]):
            from_model = connected_pair_array[i, 0]+'.transmit'
            to_model = connected_pair_array[i, 1]+'.receive'
            line_id = 'line_'+connected_pair_array[i, 2]

            topology_list.append({'from': from_model, 'to': to_model, 'line_id':
                line_id})
    return topology_list

def write_scenario_LEDs_and_connections(filename,write_file,LED_portmap,
    topology):
    with open(f'{filename}.yaml', 'r') as f:          #opens a yaml file to read
        data = yaml.safe_load(f)                      #loads the yaml data in safe
            mode
    with open(f'{write_file}.yaml', 'w') as file:     #opens a different yaml file
        to write to
        connections = {'connections' : topology}            #defines a dict with
            connections: {topology}, to recover the popped
                                                            #connections:
                                                                section from the
                                                                original yaml
                                                                file
        yaml.dump(data,file,sort_keys=False)                #writes the read
            yaml data to said different yaml file
        yaml.dump(LED_portmap,file, sort_keys=False)        #writes the LED
            models to the models: section (which should be at the bottom)
        yaml.dump(connections,file,sort_keys=False)         #writes the
            connections to the connections: section
    print('simulation file connections updated')

if __name__ == "__main__":
    Network = [
            [1, 'Station1', 'Sender'],
            [1, 'Station4', 'Receiver'],
            [2, 'Station1', 'Receiver'],
            [2, 'Station2', 'Sender'],
            [12, 'Station2', 'Sender'],
            [12, 'Station3', 'Receiver'],
            [4, 'Station3', 'Receiver'],
            [4, 'Station4', 'Sender'],
            [7, 'Station2', 'Receiver'],
            [7, 'Station4', 'Sender']
           ]

    #LED_Model = [ip, ip_port, serial_port]
    LED_Model = [['192.168.137.150', 5023, 'dev/ttyACM0'],
                 ['127.0.0.1', 5023, 'dev/ttyACM1'],
                 ['127.0.0.1', 5024, 'dev/ttyACM0']
                 ]

    LED_portmap = write_LED_portmaps(LED_Model)
    print(LED_portmap)
    connected_pair_array = determine_connected_pairs(Network)
    print(connected_pair_array)
    topology = write_topology(connected_pair_array, 'connections', 'simple_test2
```

```
    ', 'simulation_file')
print (topology)
write_scenario_LEDs_and_connections('simulation_file', 'simulation_file',
    LED_portmap)
```

## B.2 LED strip controller

This script contains the function with error checking built in, as well as the code for the tests done for this part.

```
#order: speed (2 bytes), dir, r, g, b
from serial import Serial
from time import sleep, time

def sendPixelData(connection: Serial, animationSpeed: int, direction: bool, red:
    int, green: int, blue: int):
    """This function sends the data needed for the ATtiny to the ATtiny. When
        this data is send, it receives the ID of the ATtiny.

    Args:
        connection (Serial): The serial connection to the ATtiny (should be
            between 0 and 255)
        animationSpeed (int): Animation speed (the time between each light
            switch)
        direction (bool): The direction of the animation
        red (int): Value of the red light (should be between 0 and 255)
        green (int): Value of the green light (should be between 0 and 255)
        blue (int): Value of the blue light (should be between 0 and 255)

    Returns:
        ATtiny ID
    """
    do_send = False
    case = 0

    if animationSpeed > 255 : animationSpeed = 255
    if direction > 1 : direction = 1
    if red > 255 : red = 255
    if green > 255 : green = 255
    if blue > 255 : blue = 255 #don't send invalid values

    while not do_send:
        checks = [False, False, False, False, False] #if all phases are done at
            the end, the transfer is good, otherwise it should be redone.
        do_send = True
        if case <=1:
            connection.write(animationSpeed.to_bytes(1, "big"))
            case = int.from_bytes(connection.read(1), 'big')+1
            if case != 2:
                do_send = False
            else:
                checks[0] = True #when done, check

        if case == 2:
            connection.write(direction.to_bytes(1, "big"))
            case = int.from_bytes(connection.read(1), 'big')+1
            if case != 3:
                do_send = False
            else:
                checks[1] = True
```

```
        if case ==3:
            connection.write(red.to_bytes(1, "big"))
            case = int.from_bytes(connection.read(1), 'big')+1
            if case != 4:
                do_send = False
            else:
                checks[2] = True

        if case <=4:
            connection.write(green.to_bytes(1, "big"))
            case = int.from_bytes(connection.read(1), 'big')+1
            if case != 5:
                do_send = False
            else:
                checks[3] = True

        if case <=5:
            connection.write(blue.to_bytes(1, "big"))
            case = int.from_bytes(connection.read(1), 'big')+1
            if case != 6:
                do_send = False
            else:
                checks[4] = True
                id = int.from_bytes(connection.read(1), 'big')
                return_dummy = int.from_bytes(connection.read(1), 'big')

        if False in checks:
            do_send = False
            case = 0

    if return_dummy == 2: #number to string
        return_dummy = 'sender'
    else:
        return_dummy = 'dummy'
    return id, return_dummy

if __name__ == "__main__":#testing function; usually doesn't run as you only run
    the function of this script
    connection = Serial("/dev/ttyACM0", timeout=1) #if it takes at least the
       timeout, then it errors
    for j in [0,0.5,1,2,3,4,5]: #all delay times for easier automated testing
        file = open("delay_"+str(j)+".csv", 'w')
        for i in range(100):
            begin_time = time()
            ID = sendPixelData(connection, 100, True, 30, 50, 50)
            file.write(str(time()-begin_time)+"\n")

            sleep(j)

        print(f"done:■{j}")
```

## B.3 USB connection change detector

This script contains the USB connection detector. It check if a connection has changed. It should be noted that it is displayed here as standalone model, but it can also be implemented within a station model.

```
import serial.tools
import serial.tools.list_ports
from illuminator.builder import IlluminatorModel, ModelConstructor
import mosaik_api_v3 as mosaik_api
```

```
import serial
import time as t

class USBdetector(ModelConstructor):

    parameters = {"start": True}
    outputs = {"USBchange": False}

    def __init__(self, *args, **kwargs):
        result = super().__init__(*args, **kwargs)
        self.connections = serial.tools.list_ports.comports() #store the initial
            connections
        return result

    def step(self, time: int, inputs: dict=None, max_advance: int=1) -> None:
        current_connections = serial.tools.list_ports.comports()

        old_connections = []
        for port in self.connections:
            old_connections.append(str(port))

        self.set_outputs({"USBchange": False})

        for port in current_connections:
            if str(port) not in str(old_connections): #detects if all comports
                are still present (port change or added)
                 self.set_outputs({"USBchange": True})

        if len(old_connections) != len(current_connections): #detects if a
            comport is removed
             self.set_outputs({"USBchange": True})

        t.sleep(1) #making sure it doesn't run the detection all at once as that
            would remove the use of it.

        return time + 1


if __name__ == "__main__":
    mosaik_api.start_simulation(USBdetector(), "USB connections")
```

## B.4 Testing script for the double simulation

This function is a part of the main file. It contains all code that runs for the demonstrator to run.

```
@demonstrator_app.command("run")#run for the demonstrator part of the
    illuminator
def scenario_run():
    "Runs a demonstration scenario using a YAML file. Keep in mind that this
        only works on linux. The starter file should be under the cli folder."

    #while True:
    file = open("data.csv", 'a') #for writing the results to a file for easy
        analysis
    for _ in range(10): #It is on 10 now for testing purposes. Make it while
        True and it runs forever.
         begin_time = time.time() #the start time
         os.system('src/illuminator/cli/starter.sh') #start the slaves

         time.sleep(30)#time to let all slaves start up
```

```
makeSim = Simulation("src/illuminator/cli/starter.yaml") #the path to
    the startup simulation (is only 1 step)
makeSim.run()

os.system('./run.sh') #start real simulation slaves

time.sleep(5)#time to let all slaves start up

global simulation #a dirty way for letting the simulation be able to
    exit mosaik
simulation = Simulation('simulation.yaml') #yaml file will always be
    written to the same location
simulation.run()
file.write(str(time.time()-begin_time)+"\n")#measure the time taken
```