Dynamic Placement of Continuous Data Processing with Real Time Sensory Readout on IOT Devices

Bachelor thesis by Lukas Holst
Date of submission: October 10, 2021

1. Review: Mira Mezini
2. Review: Ragnar Mogk
Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Computer Science Department

Softwaretechnology

Dynamic Placement of Continuous Data Processing with Real Time Sensory Readout on IOT Devices

Bachelor thesis by Lukas Holst

1. Review: Mira Mezini
2. Review: Ragnar Mogk

Date of submission: October 10, 2021

Darmstadt

Abstract

This thesis explores the potential benefits of offloading program logic from central computing units in a multi-sensory network onto the distributed microprocessors that previously only handled the sensory readout. We developed a new framework that developers can use to build a sensory network processing program to test this hypothesis. Developers can programmatically divide and assign the program to a set of devices. The framework then distributes and runs the program automatically on these devices. The thesis is evaluated using a concrete sensory network scenario and comparing different degrees of distribution of the program onto a fixed set of devices in a case study. The findings are that moving central computations partially to the microprocessors does not decrease throughput and latency significantly. The overall count of network connections is more important than moving computations, with three or more active network connections on the microprocessors leading to a significant decrease in throughput and latency.

Contents

1. Introduction

In our modern and connected society, distributed data gathering and processing happen everywhere. From the IoT domain with smart home automation to facial recognition in the streets, we can find continuous data streams from various sources processed in centralized computing units.

This thesis explores the potential benefits of offloading program logic from central computing units in a multi-sensory network onto distributed microprocessors, which generally only do sensory readouts. We developed a new framework that developers can use to build a sensory network processing program to test this hypothesis. Developers can programmatically divide and assign the program to a set of devices. The framework then distributes and runs the program automatically on these devices. A particular focus thereby lies on the sensory input with a continuous stream of values. The framework supports ten different kinds of sensors as inputs to the system to mimic different IoT constellations and test the mutual influence of different sensors on the distributed computations.

The second part of this thesis evaluates the hypothesis by building a concrete sensory network scenario and comparing different degrees of distribution of the program onto a fixed set of devices in a case study. The case study monitors power usage, throughput, and latency. We monitor the power usage because, in an IoT setup, power usage can be crucial on remotely deployed sensors. Throughput and latency determine the responsiveness and behavior under full load, which is essential for any computer and computer network to determine if it can deliver the desired performance. The concrete scenario is a fire detection system with three sensors that detect fire and two sensors for the control unit. The case study consists of three assignments that have the same program logic but are executed partially on different devices. There are five esp32 microprocessors with one sensor assigned to each esp32. Additionally, there is one x86 based computer that mimics the centralized computing unit. Assignment one does only the sensory readout on the esp32s and all other computations on the computer. Assignment two moves computations partially to the esp32s and does the final combination computations on the computer. The

fully distributed case is represented by assignment three. All computations happen on the esp32s, and only the final output happens on the computer.

There is currently a variety of systems that use distributed programs. Even though distributed gathering and processing is a general idea, as shown by the numerous use cases, the solutions to conquer a specific problem are not. These specific problems like, for example, facial recognition [4] are specifically developed to comply with a particular specification and are not generally applicable. While the facial recognition solution uses distributed computing to conquer the enormous data mass, other domains act differently. In the smart home domain, for example, the currently widely used solution Home Assistant [3] is designed to meet all general use cases in the smart home and IoT domain. It achieves this by having one central computing unit where all IoT devices connect to and which handles all program logic. Home Assistant allows for a simple setup and more direct support for many different sensors and smart devices. However, it also limits the amount of optimization being possible regarding distributed processing. The ”Brain” of the whole network is still a single server. The only task for sensors or controllable devices is to provide information to Home Assistant or listen for commands from it. This kind of setup keeps the sensors and controllable devices doing their initially intended work, but often sensors use generally programmable microprocessors. These microprocessors potentially have unused processing power that we could use to offload computations from the central processing unit to the distributed microprocessors of the sensors. Papers like Reactifi [8] already explored this potential. They found that a microprocessor of a sensor is powerful enough to run custom programs besides its main purpose of handling the sensor without impacting throughput and latency significantly. Reactifi tested this on a single WIFI chip. With Home Assistant, we already discussed one framework, which allows for connecting multiple sensors. However, the degree of distribution is not controllable, as Home Assistant itself is the central control unit. Another well-known tool for distributed processing is Stratosphere [1]. The degree of distribution is controllable, but the tool is optimized for parallel textual input analyzing and does not offer IoT sensory integration. Additionally, the tool is not designed for continuous inputs. Instead, Stratosphere computes one paralleled task to completion with a result. All frameworks comply partially with the requirements. Most frameworks are designed for a specific use case. It is estimated to be more complex to extend or integrate frameworks than developing a new framework.

The case study’s findings are that moving the central computations partially to the microprocessors does not decrease performance significantly. The partially distributed case preserves sub-second responsiveness compared to the fully centralized case. It better utilizes the capabilities of the microprocessor while removing the load from the central

computing unit. The fully decentralized does significantly decrease in performance with an increased delay of up to 2.5 seconds. However, with the complete removal of the centralized unit from the program logic, the performance might be sufficient for some use cases. An additional finding was that the number of network connections and the sending interval of values over those connections hit a microprocessor's performance more than the computations. In assignment three, we observed that the esp32s with three or more network connections did cause a decrease in value throughput and an increase in delay.

2. Example

We introduce the framework and its usability with an example. For the scenario, we want to know if the room temperature is greater than an adjustable threshold. For this, we have a temperature sensor and a dial to select a temperature threshold. For the output, we have a console that prints out the current state.

Graph Description

In the figure 2.1 the data flow graph is shown. We divide the complete program logic into so-called processing nodes. In this example, we use the processing nodes SensorRead, Join, and PrintItems, a subset of currently available processing nodes, to this framework. Each processing node fulfills a specific processing task, where the kind of the node, written in bold text, describes the general task of the node. The subtext describes the user-configurable attribute that specializes general nodes for the specific task. The SensorRead nodes have the user attributes Rotary Encoder and DHT11. The Rotary Encoder SensorRead reads a connected rotary encoder, used as the dial in this setup. The rotary encoder provides the current relative position since the framework started, which is an integer that increases and decreases with the rotation of the sensor. The DHT11 SensorRead reads a connected DHT11 sensor that provides a temperature and humidity value tuple. To understand the Join processing node, we must know that each processing node can have an arbitrary but fixed number of inputs and outputs. SensorRead, for example, has no inputs and one output. It generates values. PrintItems has one input and no outputs, which means it consumes values. Join, on the other hand, has two inputs and one output. It joins two values together. In the graph, the attribute of the Join operator specifies only the output it provides. The Join operator generates a tuple of (bool, int). The bool describes if the temperature inputted by the DHT11 sensor is greater than the threshold, which is the current rotary encoder value. The int holds the current threshold value for a more meaningful print at the end of the program. Figure 2.2 shows the practical

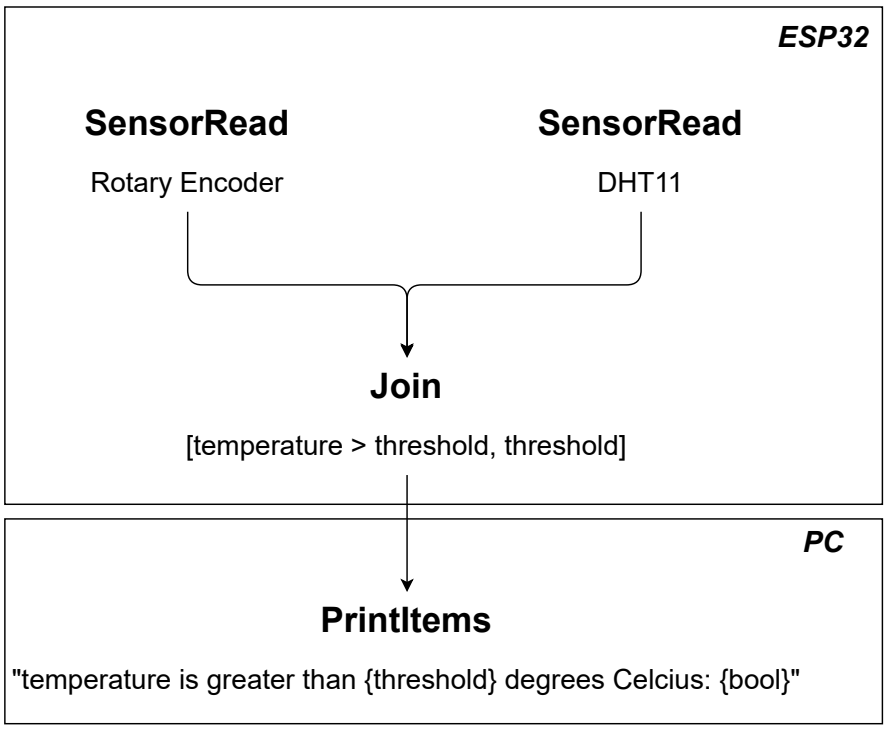

Figure 2.1.: The data flow graph showing the program used in the example chapter.

implementation, which we discuss in more detail in the following part. The last processing node is called PrintItems. This node prints a formatted string to the console for each input item it receives. A format string puts a variable's value at the position specified with curly braces. In this figure, the format string only represents pseudo-code with no actual variables. However, the terms chosen represent values from the Join-Node tuple. The two boxes labeled with ESP32 and PC represent the physical devices that execute the boxed parts. In detail, the ESP32 executes the two SensorReads and Join. The x86-architecture computer, which is labeled PC for personal computer, executes PrintItems.

Code Description

Figure 2.2 shows the real code corresponding to the data flow graph 2.1, described in the previous paragraph. To understand the framework's behavior, we discuss the script in two parts, split at line 16.

The first part describes the program logic. On running this script part, the framework builds a distribution. The distribution contains the serialized nodes for each device and the future connections between the devices. This process happens automatically and is further described in the IoT Event Framework chapter **??**. The necessary user interaction is to build a program logic and define the devices which should execute this program. For the device definitions, a Device class exists, which is used in lines 1 and 2 to define the esp32 and pc devices. Each device has a network address and a port. It will be explained in detail in the IoT Event chapter. However, the framework itself must be already running on the devices when the script is executed. The IP address and port the framework is started on must match the device definitions in the script. In lines 4 to 13, the processing nodes are defined. Each processing node is a class with a particular constructor, which always has the following structure. The first constructor parameter is the device that will execute the processing node. The next zero, one, or two parameters are the inputs of this processing node. Those are filled with the outputs of previously defined processing nodes. The last parameters are custom parameters that are needed for a specific processing node to function. The processing nodes for this example are defined in the following order. In line 4, the first SensorRead processing node is defined, which reads a connected rotary encoder's value. Next, the second SensorRead is defined in line 7, which reads a DHT11 sensor. The Join operator is defined in line 10. It takes the rotary encoder SensorRead as its first and the DHT11 SensorRead as its second input. The eval_str parameter describes how a value pair should be joined. X and y are fixed variable names, where x holds a value from the first input and y holds a value from the second input. X, in this case,

```
1  esp_32 = Device('192.168.2.182', 8090)
2  pc = Device('192.168.2.163', 8090)
3
4  p0 = SensorRead(esp_32, 'rotary_encoder')
5  # output: current position: 0..1..2..3..4..3..2..
6
7  p1 = SensorRead(esp_32, 'dht11')
8  # output: current temperature + humidity: (25, 70)..(24, 85)..
9
10 p2 = Join(esp_32, p0.out0, p1.out0, eval_str='(y[0] > x, x)')
11 # output: temperature > rotary value, rotary value: (True,
       10)..(False, 35)..
12
13 p3 = PrintItems(pc, p2.out0, format_str='temperature is greater
       than {1} degrees Celsius: {0}', time_frame=100)
14
15 distribution, assignment_order = p0.build_distribution('0')
16
17 for device in assignment_order:
18     device.remove_assignment('0')
19     device.distribute_assignment(distribution)
```

Figure 2.2.: The python code for the example program used in the example chapter.

contains a rotary encoder position, which is an integer. Y contains a DHT11 output, a tuple with the first value representing the current room temperature and the second value representing the humidity. Both values are also integers. The eval_str now builds a tuple. The first part compares the room temperature against the rotary encoder value. The second part is the rotary encoder value that gets dragged along for the final print. These three processing nodes are all defined to be executed on the esp32 device. The last processing node PrintItems gets executed on the pc device and takes the Join output as its input parameter. The node uses a formatting string defined by format_str to format the input tuple into a string and then prints it to a console. On line 15, the distribution is built, which is a serialized version of the previously defined graph. Also, in this line, the program distribution is given an ID, which is '0' in our case. The result is a distribution object with the serialized program graph and an assignment_order list object, containing the involved devices in the optimal order to assign the distribution. Optimal, in this case, means that essentially devices with an earlier part in the graph are put first.

The second part of the script distributes the final distribution onto the involved devices by sending them their parts. The distribution is abstracted by methods included in the device class. Line 17 to 19 is a for loop over all involved devices in the optimal order, like described in the previous paragraph. In our example, the first device in the loop is the esp32 object, and the second is the pc object. To account for previous distributions, we remove a possible earlier distribution with our id from the device. This method handles errors gracefully, where a non-existent earlier distribution results in a no-operation. After this, the new distribution gets sent to the device by the distribute_assignment method. If this does not throw an error, the program is successfully distributed, started, and running on all involved devices.

3. IoT Event Framework

The IoT Event Framework, referred to as the framework, is the first main contribution and is built to run the other main contribution, the case study. The easiest way to understand the framework is with a quick rundown of how the framework is used. The framework is a program that runs on all computers supporting CPython and microprocessors that support MicroPython. Each device must have an active Ethernet or Wifi connection, and the devices must be reachable by each other via an IP address. The framework itself must be deployed onto the device and must then be started. Now we can program a computational model, which is practically a program graph via a script. In this script, we also define on which device each part of the model gets executed. By running this script, we generate an assignment. An assignment includes what each device should execute and what other devices it must connect to and send its computed values. At the end of the script, we can automatically distribute and start the run of this assignment. The framework offers a method for this.

The inputs to this computational model can be any continuous input stream of values. To focus on the IoT domain, and because we want to optimize utilization of microprocessors reading sensor data, the framework supports the readout of a variety of ten kinds of sensors to feed continuous values into the computational model.

In general, this framework offers the possibility of defining and setting up a fixed set of devices and independently defining a computational model. This model can then be programmatically and freely distributed onto this set of devices, binding with the assignment on which device what part of the computational model is executed. The benefit of this free distribution is that we can test the same set of devices with the same computational model to run but with varying degrees of distribution. We can then compare the relative performance of these different assignments on the same computational model. Then we can answer the question if the current smart home model of one centralized computing device is outdated, and computations can be offloaded to microprocessors and keeping the system's performance.

3.1. Micropython

One critical criterion for this framework is that it must run on microprocessors, to support sensory readout. There are two main choices in programming when it comes to microprocessors. The first one is C, and the second one is C++. These low-level languages offer correctly timed program execution because the machine code generation can be statically analyzed. The downside of this low-level programming is the increased code complexity. On x86 and arm-based computers, interpreted languages like Python offer a simpler programming experience because they support garbage collection and dynamic typing. CPython, in particular, also offers the integration of low-level C++ code via a compatibility layer, which can be used on timing-critical code sections. Micropython behaves precisely like a CPython interpreter and supports a subset of the CPython standard. However, it runs on microprocessors like the esp32, used as the microprocessor representative throughout this thesis. The performance also behaves like CPython because Micropython is installed on the esp32 like an operating system. On interaction, one can run code scripts that are interpreted on the fly. Performance critical parts can also be programmed in C++. However, MicroPython also offers special libraries that support the most common use cases, like hardware pin access, timers, or serial communication as Python modules.

The framework is implemented using Python with respect to the MicroPython subset. This allows for one code base which behaves the same on CPython and MicroPython supporting machines. With C or C++, this would not be possible because of hardware-dependent logic, where the compatibility layer on peripheral access and uniform network communication had to be implemented from the ground up. Python abstracts these layers. The downside of an interpreted language is that it is slower than a compiled language, but this is okay for our use case. In the case study, we compare the relative performance of different assignments, and the absolute performance is not essential for us. Micropython also currently only supports single threading, so this framework is designed as a single thread application. A single thread application has implications for the sensory readout, which is further discussed in the Sensors 3.5 section.

3.2. Computational Model

The computational model is a graph definition, where each node does computational work. The nodes are therefore called computational nodes. These nodes can have zero to

n inputs and zero to n outputs. The idea of a node is that it must combine, generate or alter data from the inputs or an external source like sensors and output this data through the outputs or an outer destination like a console print or hardware pin changes. In the computational model, a node with zero inputs is called an input node because it introduces new data into the graph. A node with zero outputs is an output node because it only consumes data from the graph.

From a programmer's standpoint, a node is a class with predefined inputs, outputs, additional attributes, and a method called with these attributes on each framework cycle. These nodes are predefined, and the methods are as general as possible, so the programmer only needs to combine these logic blocks. Additional logic blocks can be added to extend the functionality of the framework. Usually, this is not needed because the logic blocks offer all standard list and stream manipulation functions like join, map, or filter. For example, if we look at the filter node, the node has one input and one output. The inputs and outputs are always lists, and a nodes task is to process the input values and fill the output list. The filter node has one additional attribute, which is the eval_str attribute. Python offers the possibility to execute a string as code in a defined context. The filter node executes the eval_str for each input value and provides the value in the context as x. If the eval_str returns true, the value itself is added to the output stream. Else it is ignored. All other nodes function similarly. The nodes abstract the list and pipeline handling from the programmer to focus on the logic for the computational model.

The nodes are combined by setting the output of one node as the next node's input. The computational model thereby forms a directed a-cyclical graph.

Input Node

The computational model currently supports one kind of input node, which is the Sensor-Read node. This node takes one additional string attribute, which is called sensor. This attribute defines which sensor should be read so that the node can output its values. There are some limitations, with the obvious one being that the sensor must be connected to the device the node is executed on, but this is further explained in the Assignment 3.3 and Sensors 3.5 sections.

Processing Nodes

The framework currently supports the following processing nodes. All nodes function with an eval_str as explained in the Computational Model section introduction: Map, Filter, Sum, Mean. The additional processing nodes ButtonFilter, ButtonToSingleEmit, and ToggleState, are specific nodes for the case study. ButtonFilter expects boolean input values and outputs one state until the configurable threshold of values from the other state is received. It then flips to that other state. ButtonToSingleEmit also expects boolean input values and outputs true and false after another once for each rising edge, which means false values, followed by true values. ToggleState outputs a boolean value, and the initial state is specified. ToggleState has an eval_str which is evaluated for each input value and toggles the output state if the eval_str returns true.

Join and Duplicate Nodes

To branch, the framework offers one duplication and two join nodes. The Duplicate node copies the input list into two output lists. The first join node called JoinNoFilter combines two inputs using an eval_str, where x holds a value from the first input and y holds a value from the second input. The JoinNoFilter node is designed to propagate changes as fast as possible. It combines all available values, reusing the latest available value from one stream if it has fewer available values than the other. This behavior is equivalent to the combineLatest Event Correlation from the Versatile Event Correlation with Algebraic Effects paper [2]. It potentially generates more values than minimally possible for two input streams due to network delay. However, it ensures a fast propagation of state changes. The advanced join node is called JoinDupFilter and does the same. However, it tests for duplicate outputs and only adds an output value if it is distinct from the previous one. We could use only this Join node, but we want to also test for a maximum load in the case study. The first join node represents this worst-case scenario of the JoinDupFilter node, where all output values are distinct.

Output Nodes

Six nodes are considered output nodes in the computational model: PrintItems, PrintQueue, Monitor, MonitorLatest, ThroughputObserver, CaseStudyDelayObserver. PrintItems and PrintQueue both have one input and print the input items to the device's console the node gets executed on. Both offer a format_str attribute, which includes the input

values into the string and prints it. With the PrintItems node, the format_str is built for each input item and printed to the console. The PrintQueue node only builds the format_str once for all current input items and prints them to the console. The Monitor and MonitorLatest nodes work similarly, but they flush the console screen and show the current queue (Monitor) or the latest input item (MonitorLatest), providing a simpler interface. The ThroughputObserver and CaseStudyDelayObserver nodes are particular nodes for observing the case study. They are explained in the Case Study 4 chapter.

3.3. Assignment

The programmer can specify which node of the computation setup gets executed on which devices with the assignment. Therefore each node has one additional device attribute. During the scripting phase, where the computational model is defined, the programmer can also define several devices through a device utility class, which takes an IP address and port as arguments. These device instances can then be assigned to each node. Every node is executed on precisely one device, and one device can execute any number of nodes. By calling the build_distribution() method, the whole graph gets traversed and serialized. The method returns a tuple with a list of devices as the first argument and the serialized and cut graph parts as the distribution as the second argument. The list of devices is a sorted list, where the devices with nodes nearer to the input nodes are first. The device class has a utility method called distribute_assignment(), which can be called on each device instance with the assignment in the correct order to distribute and run the assignment. For this to work, the framework must run on all defined devices under the defined port. A running assignment can also be stopped and removed by calling remove_assignment() on the device class.

3.4. Pipelines

The inputs and outputs between nodes are called pipelines. There are different kinds of pipelines, which are used automatically, based on where nodes are executed. Suppose two neighboring nodes are executed on the same device and framework instance. In that case, the connecting pipeline will be a local pipeline. Local pipelines are internally represented by a list, which is the fastest option. The second option represents the case where two neighboring nodes are executed on the same device but in different framework instances.

The third option covers the possibility of two neighboring nodes on two different devices. These are both network pipelines, but the second pipeline is faster because the operating system of the executing device can skip parts of the network stack with this connection. For the second and third pipeline options, a new argument does apply, which is specified on a device class instantiation during the assignment phase. It is called time-frame, and it specifies after what period a network pipeline gets flushed and send over the network. The default value is 100 milliseconds. Network pipelines have two implications. The first one is that the framework has two distinct operating modes on pipeline connections. The framework can either flush the pipeline after each time-frame, which is called fixed sending-interval. Alternatively, the framework can flush after the time-frame is met and with at least one value in the pipeline. That is called variable sending-interval. Both modes have advantages and disadvantages, which can be seen in the case study results. Currently, the operating mode cannot be defined programmatically and must be set on framework deployment. The second implication the network connections have regards the assignment order. At the time of assignment for a particular device, each input to a node must already exist. This is a framework constraint and avoids unnecessary network connections but limits the number of distinct assignments that can be done with one computational setup. For example, if we have the graph a->b->c, as well as the devices esp32 and pc, it is legal to have the esp32 executing a,b and the pc executing c. It is illegal to assign a,c to the esp32 and b to the pc. The second variant would fail on assignment because the esp32 would be distributed to first, and the output of b, which is needed to execute c, does not already exist.

3.5. Sensors

What sets this framework apart from other frameworks is the variable distribution of computational setups combined with low-level sensory value access. There are three steps to correctly reading sensory data. The first step is to speak the sensors' language, as each sensor can have its unique protocol and access procedure. The second step is to extract the raw data the sensor returns, and the third step is to either use this data directly if the sensor returns processed data or convert the raw data into a logical representation. For example, the ultrasonic sensor sends out a burst of ultrasonic waves and communicates the echo by setting a pin to state high. It is up to the microprocessor to first measure the delay between burst and echo and then converting this time with the knowledge of the speed of sound into a distance value. The line between the raw data and logical representation is not always clear. That is why this framework automatically does the

first step to abstract the often complex readout from the programmer. It also does the first processing so that developers do not have to worry about correct sensor handling. In the case of the ultrasonic sensor, the SensorRead node provides the raw time delay value and a pre-computed distance value in centimeters for the programmer to use. The framework handles all underlying logic. The readout and processing are done for all supported sensors. The SensorRead will output either a value like a boolean or an integer, or a tuple of these, depending on the chosen sensor. The SensorRead node offers unitary access, which can support almost any sensor that delivers values continuously.

Although the extraction of sensor values is abstracted from the developer, the developer must know exactly the following things about each sensor:

Expected Values and Meaning

It is essential to know the expected output format, upper and lower limits of the SensorRead. Also, it is essential to know which values are invalid because a sensor might have internal logic, which returns unique values to represent an invalid readout. The sensor could also emit values quicker or slower than specified during initial readings or faulty connections. It is also essential to know what the values physically represent. The format, limits and unique values are documented for each python module in the code archive. A quick summary of all supported sensors is provided in the Supported Sensors 3.5.1 subsection.

Readout Time

It is also essential to know how many values can be expected during one time-frame. Most sensors require active communication to deliver values to the microprocessor. The framework has a sequential readout model, where each sensor currently used on a device is read after another. The sequential readout model has one major drawback, which is the waiting time on communication. Even the microprocessors used in the case study are faster than the sensors with which they communicate. The fastest standardized communication protocol used for a supported sensor is the I2C protocol in fast mode with a clock speed of 400kHz. The minimum clock of the esp32 microprocessors used in the case study is 160MHz which equals a factor of 400 in speed difference. A strategy to minimize the influence of this readout model is to use one sensor per esp32, so one sensor cannot slow down the other. If it is unavoidable, sensors with similar speeds should be combined to achieve a near 50/50 time split. There is one kind of sensor readout, which circumvents

these problems. Sensors like the rotary encoder work with interrupts, where a software routine gets bound to the change of a pin state and is called each time the pin changes in a specified manner. For example, the rotary encoder state changes, e.g., it is turned if both connected pins change from high to low or low to high. The direction of the turn is determined by which pins interrupt fired first. These sensors are easily integrated into the existing readout model, where the interrupts always set the current value, and the SensorRead returns this value every cycle. These sensors can also be combined with every other sensor, as they will not slow down the readout of these other sensors. Although it must be noted that with a combination of an interrupt-driven and sequentially read sensor, the SensorRead of the interrupt-driven sensor will output at the same rate as the sequentially read sensor.

3.5.1. Supported Sensors

Button The button is a sensor implementation that supports any button-like object, where a single pin is either kept high or low. An internal pull-up resistor is supported and activated at default. Pull-up means if the pin is not connected, it is high, which means no button press and therefore outputs false. If the pin is grounded, the button is pressed, and the output is therefore true. The readout time is nearly zero as only a single pin state must be read.

CO2 The MH-Z19B CO2 sensor outputs a PWM signal, where the overall time frame of a period with a high and low is always the same, but the timely division of the high and the low determines the current CO2 concentration. This sensor is very slow, with one PWM cycle being about 2 seconds long. The readout is implemented with an interrupt procedure, but the sequential readout model ensures that only if a new valid value arrives, it is propagated through the framework. The output value is a tuple containing the high duration, low duration, and computed concentration of CO2 in ppm. One connectivity note: this sensor needs a 5V operating voltage.

DHT11 The DHT11 sensor outputs a temperature (Degrees Celsius) and a humidity reading (both integers), propagated into the framework as a tuple. The readout is done via a Micropython library, but it is a bit fragile. Long cables of about 20cm can lead to failed readings, which is why this is the only readout with an exception handling, and the exception is currently ignored. The internal logic of the DHT11 also only updates about once every second. That is why the reading is also only triggered about once every second. The speed of the readout itself is fast, and the influence on other sensor readouts is irrelevant due to the long idle time. The output

value of the sensor is only propagated through the framework once if it is a new reading.

Gyro The MPU6050 uses the I2C protocol in fast mode. Fourteen bytes can be read and combined into seven integer values representing the following: acceleration X, Y, Z, temperature, gyro X, Y, Z. This could be speed optimized to only read a subset of these, but for convenience, the whole 14 bytes are read each cycle and outputted as a seven tuple. One connectivity note: This sensor needs a 3.3V operating voltage.

Hall The esp32 has an integrated hall sensor that returns a positive or negative integer if a magnet is present, depending on the pole facing the esp32. The closer the magnet is to the esp32, the higher the readings get. The framework uses a function provided by Micropython to read the value. The readout is near-instant, and there is no idle time needed.

Potentiometer A potentiometer, or poti, is just a variable resistor and most microprocessors have an integrated analog to digital converter that can be bound to the input pin. The esp32 has a DAC, outputting a number between 0 and 4095, depending on the input voltage set by the poti. A unique configuration the esp32 offers is to set the attenuation. The attenuation defines the voltage range from 0 to n that will be mapped to 0-4095. The options are 1V, 1.3V, 2V, 3.6V.

Rotary Encoder The rotary encoder works by having a turn-able nob with resting positions and two input pins that determine the current state. On resting states, the inputs are either both low or high. When turned, one of the pins will change state first. The first state change determines the direction that the nob is turned. The logic for this is built in the framework and works through interrupts. The interrupt routines account for current and following valid states, thereby eliminating debouncing, which is usually a big problem with rotary encoders.

Temperature The esp32 has an integrated temperature sensor that outputs its temperature in Fahrenheit. This readout happens near-instant and has no idle time. It should be noted that the temperature is not the room temperature but the internal temperature of the central chip.

Touch The touch sensor is a simplified version of the button sensor because there are special sensors with a capacitive touch field and internal logic that output 1 (full voltage) on touch and 0 on no-touch. They do not need pull-up resistors. The SensorRead outputs true if the input is 1 and false if the input is 0. The readout is near-instant and needs no idle time.

Ultrasonic The ultrasonic sensor works by sending a burst of waves and waiting for the echo. The delay multiplied by the speed of sound and divided by two gives the object's distance. The logic to initiate the burst and wait for the echo is built in the framework. The SensorRead returns a tuple with the delay in microseconds and the computed distance in centimeters. If the delay is -1, no echo was received. The echo can take up to 20 milliseconds, which results in a theoretical maximum readout of 50 values per second. In the case study, a consistent, practical maximum of 30 values per second is achieved.

4. Case Study

In this chapter, a case study will be conducted to evaluate the new framework under the aspect of centralized to decentralized computing. The framework is specifically designed for continuous data use cases and dynamic programmatic processing node placement. We evaluate the framework and thereby our thesis by having a fixed scenario and computational setup, which is then assigned in three configurations with different node-to-device assignments.

For the concrete scenario, a smart fire detection system was chosen, where multiple inputs are evaluated, and only a combination of events leads to an alarm being issued. This scenario is closely related to modern smart home solutions where multiple sensory values can be evaluated and trigger events based on programming, like, for example, Node-RED [5]. Actually, the case study will more closely model a smart home connectivity solution because fire detection systems must be hardwired to ensure reliability, but it will mimic the behavior of a fire detection system.

The framework's behavior is observed by specially designed framework nodes, the ThroughputObserver and CaseStudyDelayObserver, that are integrated into the computational setup at multiple locations. We specifically do not involve external tools for this, which could interfere with the framework and require an extra setup.

4.1. Computational Setup

The computational setup describes the real-world scenario used for this case study and its implementation with the tools the framework offers. The difficulty was to choose a scenario where the variety of sensors supported by this framework could be tested and keeping it close to a real-life scenario. The variety of sensors is important because one aspect is that the sensors operate at widely different speeds, which impacted performance

significantly in pre-testing. The variety also more closely resembles real-world applications, where any combination of sensors can be used. The setup uses as few nodes as possible to mimic the desired behavior. We could artificially increase the node number, but because the number of esp32s is fixed to five and we have more than five nodes, we can already fulfill the edge case of full and equal distribution with assignment three. There is no need to increase the node number further.

Specific Scenario Setup

The setup uses the dht11, co2, and ultrasonic sensors as fire detection inputs. The ultrasonic sensor is used as a smoke detection system, which is entirely inaccurate because smoke does not reflect sound and can not be detected by this sensor. Smoke detectors use ionization and, or photoelectric detection. Photoelectric detection, for example, works by setting up a light source, and when smoke deflects the light, it deflects onto a light-sensor and activates the smoke detector [6]. However, for testing purposes, it is sufficient to mimic this behavior by setting a boundary condition that every detection closer than 10cm counts as smoke detected. Everything further away counts as clean sight. The co2 sensor is used, as suggested by the name, to read the co2 concentration in the air. An average co2 concentration should not exceed 2000ppm in closed rooms because values larger than this lead to increased fatigue [7]. However, there can be wrong measures because humans exhale a higher co2 concentration than they inhale. This error is accounted for by setting a higher concentration boundary. The concentration boundary is 3000ppm, which is well beyond what is comfortable for humans and should only be reached in emergency cases. The last sensor is the dht11 sensor, which emits a value pair of temperature and humidity. Only the temperature is used as a fire indicator. The dht11 reads temperatures from 0 to 50 degrees Celsius. In this case study, the threshold is 45°C because it leaves some headroom for the sensor, so we do not need a maximum reading. The relative humidity readings go from 20 to 80 percent, but the air humidity during a fire depends on the burned material and its humidity. Therefore we cannot put a direct connection between humidity and fire detection. For a more realistic approach, the dht11 could be exchanged for a dht22 sensor because the temperature scale increases to a scale from -40 to +80 °C. This increased range leads to a more accurate distinction between an overall high temperature and a fire.

In addition to these detection sensors, there is also a simple control unit that toggles the three sensors on and off. An off-sensor does not contribute to the detection resulting in the detection solely dependent on the activated inputs. This control unit allows two

possible logical setups for this case study. The sensors can either be combined with an OR, which results in off sensors emitting a logical false. The result is that any sensor can trigger a fire alarm output, and with no sensor activated, the alarm state will be false. This approach has the advantage of an early trigger, where any sensor is reliable enough to detect fire independently. The other logical combination is an AND, where all sensors must trigger to set off the alarm. Therefore, the logical state a sensor must emit when deactivated is true to keep fire detection with partially activated sensors working. With all sensors being deactivated, the system output is true, and the system will output an alarm. This approach can be chosen if only a combination of events and sensors detecting fire is valid, where a single sensor might falsely get triggered from other environmental influences. We choose that our sensors are reliable and independent from each other. The computational setup is therefore implemented using the first approach with the OR combinatory logic. The control mechanism is implemented using a button and a rotary encoder. With the rotary encoder, one can select which sensor to toggle, and a button push toggles the state of the currently selected sensor.

Graph Description

The data flow graph in figure 4.1 shows the complete program logic, without the observing and assigning part, which is explained in the following two sections of the case study. Look at the Example chapter 2 for an explanation of how to read the figure. This paragraph focuses on how this graph resembles the fire detection systems' program logic. The overall setup uses five sensors as the inputs to the system, which are the SensorRead nodes. At the top of the graph, the SensorRead processing node for the rotary encoder and the button can be found. The rotary encoder value is processed in the Map processing node, where the modulo operator is used to map any input integer to a value in $[0, 3]$. The button SensorRead is processed with a ButtonFilter, which eliminates debouncing. The output is then processed in a ButtonToSingleEmit processing node, which only outputs true once for a series of true values, else false. The mapped rotary encoder value and processed button value are joined and then duplicated together. The joined duplication avoids multiple duplication nodes, introducing unnecessary network connections and pipelines into the system. Each of the three duplication outputs is then fed into a ToggleState processing node, which toggles between true and false if the selected value of the rotary encoder equals 0, 1, or 2 respectively and the processed button value is true. Each ToggleState is joined together with its fire detection sensor, and the join outputs the sensor value if the ToggleState emits true for the activated state. If ToggleState is false, the join's output is also false to comply with the OR behavior explained in the previous paragraph. The output

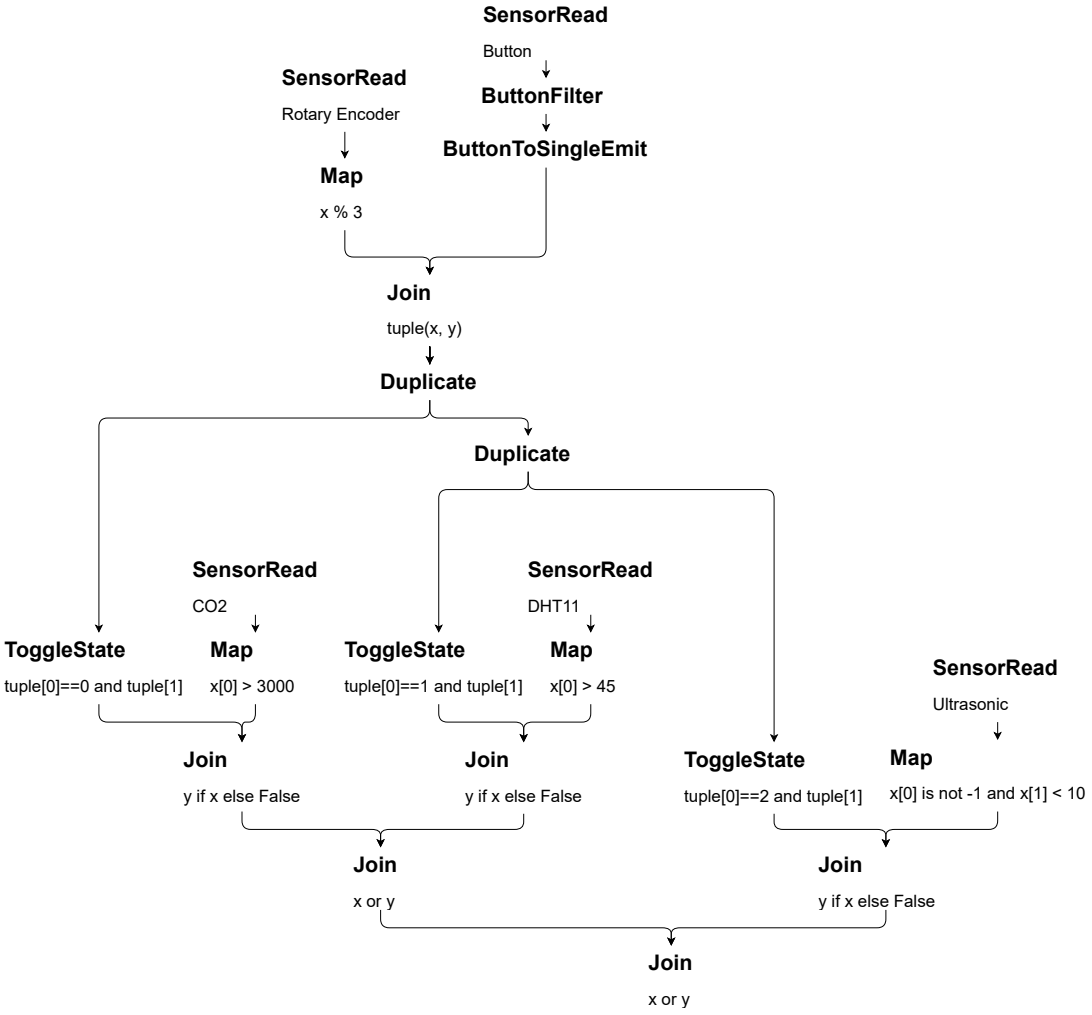

Figure 4.1.: The data flow graph showing the case study program.

values of the join operators are then joined together with or operators to comply with the desired behavior. This description only leaves the SensorReads of the fire detection inputs open for an explanation. Those are the SensorReads of CO2, DHT11, and Ultrasonic. The raw sensor values are mapped to comply with the desired behavior explained earlier. The CO2 value is checked for being larger than the set threshold of 3000ppm. From the DHT11 value tuple, only the temperature value is used and checked if it is larger than 45°C. The Ultrasonic is checked specially because the SensorRead outputs a value pair. The first value is the raw sensor output, which is the delay between send and echo. The second value is the computed distance in cm the object is detected from the sensor in normal air. The Map processing node now checks if the raw sensor value is not -1, which signals an invalid reading because the sensor did not receive an echo. If this is okay, it is then checked if the computed distance is smaller than 10 cm. This description concludes the program logic.

Code Description

The code section corresponding to the data flow graph can be found in figure 4.2 and 4.3. This code is similarly structured to the example code section in figure 2.2, which is explained in detail in the Example chapter 2. The functionality of this code section is explained in detail in the previous paragraph with the data flow graph explanation. There are some newly introduced processing nodes. ButtonFilter has one special parameter, flip_threshold, which defines how many readings must be the same before the output flips to this value. ButtonToSingleEmit and Duplicate have no special parameters, but with Duplicate being the only processing node with two outputs, out0 and out1. The last new processing node is ToggleState, with two special parameters. eval_str defines if the output state should be toggled, and initial_state defines which state should be output at program start. The number of devices in the code section is linked to the number of devices available for testing. Also, the distribution program in the code section is just the first assignment. The code for the other assignments looks the same. Only the node-to-device assignments are different.

4.2. Assignment

With the logical and programmatic setup done, we will look at the assignments next. An assignment is a distribution configuration of a computational setup, which defines what

```
esp_32_1 = Device('192.168.2.177', 8090)
esp_32_2 = Device('192.168.2.146', 8090)
esp_32_3 = Device('192.168.2.182', 8090)
esp_32_4 = Device('192.168.2.162', 8090)
esp_32_5 = Device('192.168.2.124', 8090)
pc_local_0 = Device('192.168.2.163', 8090)

raw_co2 = SensorRead(esp_32_1, 'co2')
raw_dht11 = SensorRead(esp_32_2, 'dht11')
raw_ultrasonic = SensorRead(esp_32_3, 'ultrasonic')
raw_rotary_encoder = SensorRead(esp_32_4, 'rotary_encoder')
raw_button = SensorRead(esp_32_5, 'button')

selection_int = Map(pc_local_0, raw_rotary_encoder.out0,
    eval_str='int(x/2) % 3')

button_filtered = ButtonFilter(pc_local_0, raw_button.out0,
    flip_threshold=1)
button_single_emit = ButtonToSingleEmit(pc_local_0,
    button_filtered.out0)

joined_selection = Join(pc_local_0, selection_int.out0,
    button_single_emit.out0, eval_str='(x, y)')

duplicator_0 = Duplicate(pc_local_0, joined_selection.out0)
duplicator_1 = Duplicate(pc_local_0, duplicator_0.out0)

co2_toggle = ToggleState(pc_local_0, duplicator_0.out1,
    eval_str='x[0]==0 and x[1]', initial_state=True)
temperature_toggle = ToggleState(pc_local_0, duplicator_1.out0,
    eval_str='x[0]==1 and x[1]', initial_state=True)
distance_toggle = ToggleState(pc_local_0, duplicator_1.out1,
    eval_str='x[0]==2 and x[1]', initial_state=True)

co2_filtered = Map(pc_local_0, raw_co2.out0, eval_str='x[2] > 800')
temperature_filtered = Map(pc_local_0, raw_dht11.out0,
    eval_str='x[0] > 45')
distance_filtered = Map(pc_local_0, raw_ultrasonic.out0,
    eval_str='x[0] is not -1 and x[1] < 10')
```

Figure 4.2.: Part 1: The python code for the case study computational setup.

```
co2_bool = Join(pc_local_0, co2_toggle.out0, co2_filtered.out0,
    eval_str='y if x else False', zip_oldest=False)
dht11_bool = Join(pc_local_0, temperature_toggle.out0,
    temperature_filtered.out0, eval_str='y if x else False')
ultrasonic_bool = Join(pc_local_0, distance_toggle.out0,
    distance_filtered.out0, eval_str='y if x else False')

joined = Join(pc_local_0, co2_bool.out0, dht11_bool.out0,
    eval_str='x or y', zip_oldest=False)
alarm_ser = Join(pc_local_0, joined.out0, ultrasonic_bool.out0,
    eval_str='x or y', zip_oldest=False)

output = PrintItems(pc_local_0, alarm_ser.out0, time_frame=100)

distribution, assignment_order = output.build_distribution('0')

for device in assignment_order:
    device.remove_assignment('0')
    device.distribute_assignment(distribution)
```

Figure 4.3.: Part 2: The python code for the case study computational setup.

devices will execute what part of the computational setup. On the one hand, we have multiple challenges and constraints, limiting the number of assignments feasible for the case study. On the other hand, we have multiple variables that can change the system's behavior. In the following, we will first look at the assignments and then the reasons why these assignments were chosen.

Specific Assignments

The case study consists of three assignments, which can be found in the figures 4.4, 4.5, 4.6. The graphs shown in these figures are simplified versions of the graph representing the case study computational setup 4.1. Each bundle of nodes surrounded by a dotted line gets assigned to one esp32 microprocessor. The remaining nodes get assigned to the "high performance" pc framework instance. Each assignment represents a different degree of offloading computations onto the esp32 microprocessor, with assignment one representing the minimum offloading possible and assignment three representing full offloading.

Test Configurations

In addition to the three assignments, there are three variables that are chosen to be meaningful and influential to the framework's behavior. The first two variables are the sending behavior and sending interval, explained in section 3.4. We test the fixed and variable sending interval with each a 50ms and 100ms time-frame. The time-frames are chosen to test at the edge of what an esp32 microprocessor running an interpreted language is capable of, which shows in the case study results. The third variable is to either use the JoinNoFilter or JoinDupFilter node kind in the computational setup. The JoinNoFilter represents the worst-case scenario where each value pair is distinct and tests a full load on the setup. The JoinDupFilter represents the more realistic scenario of multiple joined variable pairs carrying the same information, introducing unnecessary load on the network connections that can be filtered out. These variables result in eight configurations for each assignment and, therefore, in a total of 24 test runs.

Physical Restrictions

The physical resources for the assignments are one sensor of each supported sensor kind, five esp32 microprocessors, and a personal computer with a Ryzen 2700 CPU with eight

cores (16 logical). The assignments are configured to load the esp32 controllers with different amounts of processing nodes. The framework is launched once on each currently involved esp32, while the personal computer has multiple instances of the framework running. All of the observing processing nodes are assigned to ”high performance” instances of the personal computer to not interfere with the performance of the esp32 microprocessors. Additionally, there is one framework instance, which acts as the ”high performance” computer. This instance represents a centralized server in a typical IoT setup, where one computer connects to all devices and processes their data. From early testing, it was found that the serial interface of Micropython on the esp32 controller is slow and can therefore not be used by the case study as the final output. This exclusion implicates two options that are used in the assignments. The first option is to use a hardware output like an LED as the final output. This option is not used in the case study because it cannot be measured digitally easily. The second option is to use the ”high performance” instance to print the output directly to a personal computer console like it is done with the observables. This option is used for all other case study assignments and can also be found in the code for the computational setup 4.3.

Challenges

Three particular challenges restricted the variety of assignments. One challenge was to ensure the ”high performance” instances will not slow down the framework. The data generated from an esp32 microprocessor cannot force full utilization of one of the personal computer CPU cores because it is about 15 times slower (3.7GHz to 240MHz), and the Ryzen architecture is more advanced than the esp32 architecture. Nevertheless, the case study takes precautions by not exceeding the 16 framework instances count on the personal computer. Another challenge is that SensorRead nodes can only be executed on esp32s. The third challenge comes from the pipeline assignment order, where data can only flow forwards on devices. This flow order is further explained in section 3.4. With these restrictions, the three assignments mentioned in paragraph two were chosen to mimic the different degrees of offloading while being complying with the framework’s limitations.

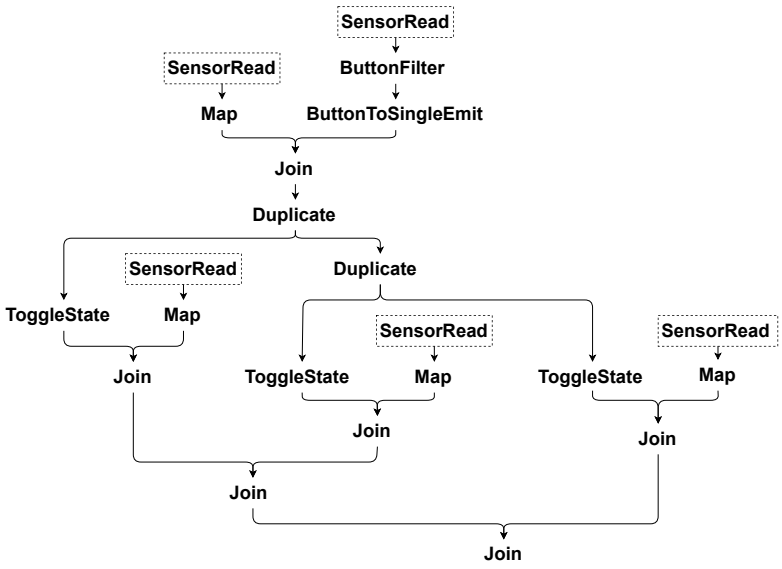

Figure 4.4.: A simplified data graph showing separations for case study assignment 1.

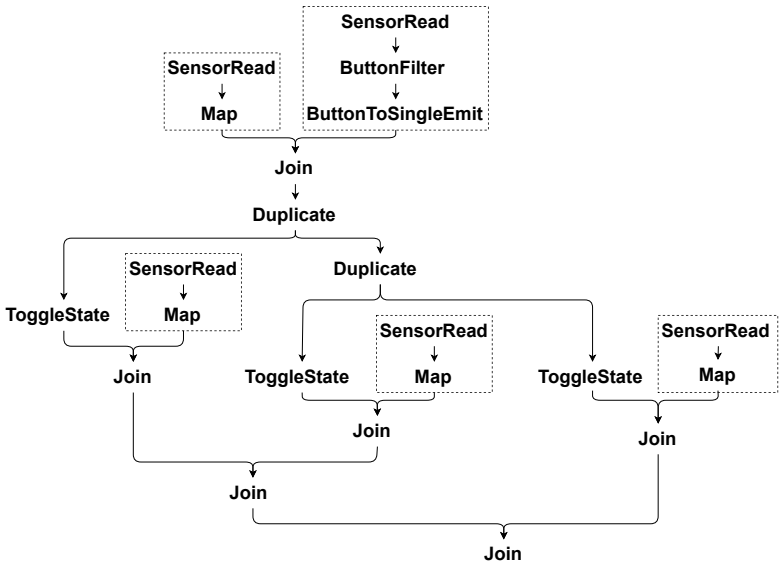

Figure 4.5.: A simplified data graph showing separations for case study assignment 2.

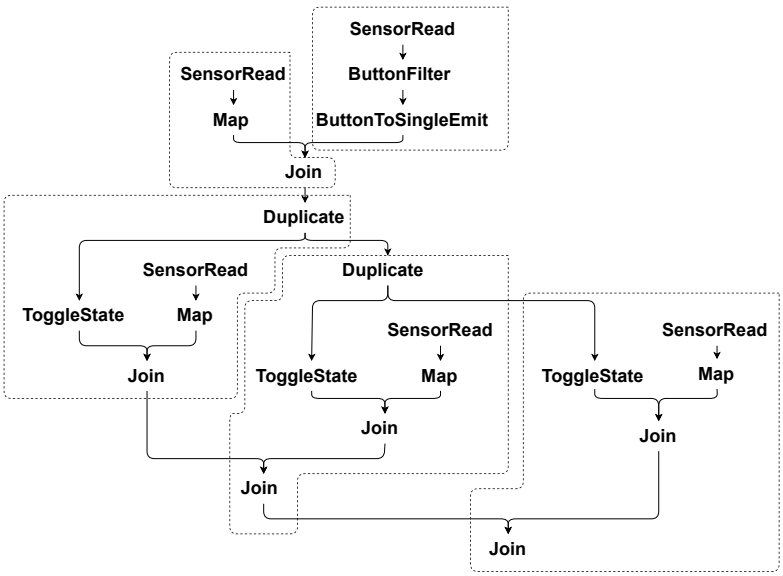

Figure 4.6.: A simplified data graph showing the fully distributed separation for case study assignment 3.

4.3. Observables

The case study monitors the assignments under the aspects of power consumption, throughput, and delay. The general goal is to determine the benefits and downfalls of moving computations to a more or less powerful device. Another aspect is the distance from the sensory readout and the further processing. The power consumption is essential because microprocessors are used for their efficiency to operate in remote use cases. They often rely on alternative power sources like solar panels or batteries, and even if they have line power, they mainly operate with no downtime, which must not result in a massive power bill.

Power Consumption

The first aspect, the power consumption, is monitored using a variable power supply, which has an amp-meter. This amp-meter is monitored in person because there is no option to log the display digitally, and in addition, there is not much fluctuation expected on the power

usage. All esp32 microprocessors are powered on the same power supply. This collective powering results in collective monitoring and gives an individual average by dividing the total power usage by the number of esp32's used in the current assignment. The current is measured in ampere and is first divided by the number of esp32s (5) and then multiplied with the input voltage of 5, which gives a power value in Watt. Conveniently the measured total current equals the individual power usage. The high-performance framework instances are not monitored for power usage because the personal computer (windows environment) cannot be restricted to run the framework instances solely. The power consumption therefore fluctuates, and readings are inaccurate.

Throughput

The system's throughput is monitored in values per second, where each value is one pipeline item traveling through the system. This monitoring is done by integrating multiple ThroughputObserver nodes into the computational setup. There is one ThroughputObserver after each SensorRead node and one after the final Join node. The ThroughputObserver nodes are always paired with a Duplicate node, placed after the node to observe. One output goes to the next node in the computational setup, and one output goes to the ThroughputObserver. This assigning is done to be able to redirect the value flow onto pc framework instances. The Duplicate node is executed on the device, which executes the node to observe, and the ThroughputObserver is executed on a pc framework instance. If the Duplicate node would be emitted and a passing-through ThroughputObserver would be used, the observing node must be executed on the device which executes the node to observe. This execution restriction is because of the pipeline constraints introduced in the IoT Event Framework chapter. This restriction could influence the device's performance and will be avoided.

Delay

The delay is measured in milliseconds between the longest graph distance between nodes. The CaseStudyDelayObserver node has two inputs, taken by placing two Duplicators at different but connected places in the computational setup. The delay is measured explicitly by knowing the computational setup and comparing system changes. An earlier approach did assume a one-to-one relationship and compared the delay between incoming messages. This approach is not feasible because the Join operator reuses older values from one input if no new values are present to combine with the new values from the other

input. The Join operator is further explained in the IoT Event Framework chapter. Because of this non-one-to-one relationship between values, we need to know how the first input to the CaseStudyDelayObserver affects the second input. The CaseStudyDelayObserver is placed after the Button SensorRead node and after the final Join node. The Button SensorRead outputs true if the button is pressed, else false. The final Join outputs true if any activated sensor outputs true, else false. The CaseStudyDelayObserver now works by comparing the time the button is pressed to when the final output Join flips its state. Therefore the rotary encoder selection points to one sensor that always returns true, and the other two sensors return false. The conditions are adapted such that only the button press changes the final output.

4.4. Results

Power Consumption

Three different scenarios can differentiate the overall measured power consumption. At idle, with the framework running, each esp32 uses about 0.3 Watts of power. Under load, the power consumption differs between the fixed and variable sending intervals. With a fixed sending interval, the power usage is between 0.55 and 0.6 Watts. With a variable sending interval, the power usage fluctuates between 0.33 and 0.55 Watts when using the JoinDupFilter node and uses about 0.5 to 0.55 Watts with the JoinNoFilter node. This fluctuation suggests that the Wifi module uses a significant amount of power, which correlates with the usage time of network connections. The power usage can therefore be reduced, by reducing the values sent over the network, which reduces the active time of the Wifi chip.

Chart Structure

The delay and throughput are measured for each test. Seven observing nodes take measurements during each test. Each of these observing nodes has one dedicated page of charts. On these pages, the structure is always the same. From top to bottom, there are three rows, and each row represents one assignment. From top to bottom: assignments one, two, three. On each assignment, there are three variables to toggle, which results in eight tests per assignment. For better readability, these are split into two diagrams with four tests. On the left are the tests with a fixed sending interval, and on the right are the

tests with a variable sending interval. The colors are also consistent throughout the tests, with the pairs (purple, 50ms JoinDupFilter), (green, 50ms JoinNoFilter), (blue, 100ms JoinDupFilter), (yellow, 100ms JoinnoFilter). The chart pages do not contain a legend, except for the first chart A.1. The legend of this chart is valid for all other charts.

Throughput

The throughput is measured after the input nodes, which are the different SensorRead nodes. Additionally, the total throughput is measured after the final Join node of the computational setup 4.1.

For the button, co2, dht11, and rotary encoder, the throughput mimics the expected throughput. The co2 A.4 and dht11 A.5 SensorReads output about one value per second and behave about the same on each assignment, which means the esp32's are not overloaded by additional computations because they do not decrease on throughput. The button A.3 and rotary encoder A.7 SensorReads do show a decrease in values per second on the second and third assignment. This decrease is expected because the theoretical speed on outputting values is determined by the operating speed of the esp32 and not the sensors themselves. The decrease in throughput on assignment three does show more significantly on the rotary encoder because the Join operator introduces an additional network connection with input values. It should be observed that it makes no significant difference if the sending interval is variable or fixed for the throughput of the button and rotary encoder inputs. This throughput is plausible because the variable sending interval should behave exactly like the fixed sending interval if enough values are produced. In contrast, the time-frame seems to have a significant impact on the throughput performance. The green and red lines on the button and rotary encoder throughput, which have a 50ms time-frame, are consistently lower on all tests than the blue and yellow lines, representing the 100ms time-frame. This throughput means a short sending interval keeps the framework significantly more busy with network connections and lowers its throughput. It should be noted that the decrease is about 10 to 15 items per second but stays mostly above 40 values per second. The two Join node variants do not seem to have any impact on these four input's throughput.

The ultrasonic throughput A.6 does behave differently. In contrast to the button and rotary encoder inputs, which are read instantly, or the co2 and dht11, which are read after a fixed time interval of one second, the ultrasonic sensor is read on each framework cycle not instantly. The ultrasonic sensor sends out a burst of waves and waits for the echo, which takes 20 milliseconds in the worst case where no echo is received. With an average

of 30 values per second on assignments one and two, this results in $1000ms/30 = 33.3ms$ and $33.3ms - 20ms = 13.3ms$ as the average cycle time for the framework on this esp32. The human factor can explain the higher values per second on assignment two. There may have been an arm resting unintentionally in front of the sensor, which causes valid echos that take less than 20ms, and the throughput jumps up significantly. Another anomaly to observe is assignment three. Let us take the 30 values per second for the ultrasonic sensor as the practical maximum possible for this computational setup. Then the charts of assignment three for this sensor show a significant overload on the esp32. The values per second count drops significantly below 30 on each test. The variable sending interval combined with the JoinDupFilter node performs the best, likely because the network communication is greatly reduced. The JoinDupFilter leads to only sending values if the button is pressed and thereby introducing a state change. The variable sending interval then only propagates these values and does not send empty lists.

The total throughput A.8 has two different behaviors. With the purple and blue lines representing the test runs with the JoinDupFilter node, the output fluctuates between one and zero values per second on all assignments. The manual button press is the only state-changing input to the system, so this is the expected behavior. In contrast, the green and yellow lines represent a maximum value output for the assignment and configuration with the JoinNoFilter node. Here we can notice a steady decrease in values per second from assignment one to three, where the jump from one to two is more significant than the jump from two to three. This observation does not consider the green line on assignment three. As discussed in the delay subsection, the green line assignment with 50ms time-frame and JoinNoFilter increased massively in delay. On the left side diagram, showing the fixed sending interval, some pipelines lost connection after about 140 seconds during testing. The diagram shows this with an apparent dip in throughput at this point. The right diagram green line, showing the variable sending interval, also vastly increased in delay. With equalizing at 20 seconds, this is so far of that it is considered a failing setup compared to the other tests.

Delay

The delay measuring was done with a custom node, which introduced some measuring errors in assignments one and two. In figure A.1 the raw delay data of assignment 1 with a fixed sending interval is shown. In this diagram, there is a significant amount of about 2000ms spikes in delay. These high delays could not be confirmed visually during testing, where the output state was monitored all the time. For assignments one and two,

the delay was always in the sub-second area. Therefore the delay diagrams in figure A.2 of assignments one and two are filtered to only show values below 1000ms. It can be seen that the delay is similar for assignments one and two, with both showing a more significant amount of spread and a slightly higher delay on the fixed sending-interval than on the variable sending-interval. For assignment three, the delay readings were all correct. In this assignment, the fixed sending interval performed better with an average delay of 1.2 seconds regarding the 100ms time-frame. The 50ms time-frame on the fixed sending interval did not perform well. The run with the JoinNoFilter node (green line) quickly jumped to more than 22 seconds in delay and partially crashed due to pipeline buffer overflows. The JoinDupFilter node (purple line) seemed to be okay until about 240 seconds, where the delay increased to more than 10 seconds. Here the test run time was not enough to explore if this setup also fails or settles at a delay of about 15 seconds. It can be assumed that the 50ms time-frame, together with multiple pipelines, e.g., network connections, overload the esp32, and it suffers a significant performance loss. In addition, a run at maximum throughput capacity quickly leads to system failure. This time-frame observation is also accurate for a variable sending interval. The 100ms time-frame is consistent, with a delay of about 2.5 seconds. The 50ms time-frame does work on the JoinDupFilter test run (purple line) because this results in an average of under one value per second sent over the network, but the JoinNoFilter test run (green line) quickly jumps to a 20-second delay. It is assumed that the delay of 20 seconds holds in this test because the network buffers are full and there are only new values added if the network allows to send new values, but it is unclear why it works in this case and fails in the fixed sending-interval case. Therefore it is assumed to be unique for this framework, and all 50ms time-frame tests in assignment three are considered non-practical.

Final Result

These observations lead to the final result that we need to differentiate between centralized use cases, where a powerful computer is available, and the fully distributed use cases, where we need to rely on microprocessor performance solely.

On centralized use cases, assignments 1 and 2, it is possible to offload computations onto involved microprocessors without a significant loss in throughput. The average of even 30 values per second (ultrasonic throughput A.6) in the worst case is well above what is needed to achieve a total sub-second delay for a computational setup. The 50ms / 100ms time-frame, fixed and variable sending interval, or the JoinDupFilter / JoinNoFilter configurations did not significantly impact throughput or delay on any test and any

observable. This result shows that a minimum of 100ms time-frame, together with a variable sending interval and the JoinDupFilter is probably the optimal choice for this kind of setup because it decreases network activity and therefore lowers the power usage of the microprocessors.

The distributed use case, assignment 3, is feasible for non-sub-second critical use cases because, with multiple network connections, an esp32 spends significantly more time handling these connections, which is lost for handling the sensory computations. This loss is due to the esp32 handling network connections serially in line with the sensory computations. To have enough time left for any sensory computations at all, the time-frame on which values are send must be at least 100ms. Values below this time-frame lead to the esp32 being only occupied with handling network connections and having no time to do enough sensory computations in the case study computational setup. This behavior shows through the significantly increased delay (purple line, green line, assignment 3, delay A.2). It leads to the 100ms time-frame, variable sending interval, and JoinDupFilter configuration being the optimal choice to reduce network activity. The optimal choice for the delay is the same configuration with a fixed time-frame, which cuts the delay in half from 2.5 seconds to 1.2 seconds. This reduction is probably due to Wifi optimizations being done in the background if there is an expected send and receive request for each device.

The time-frame seems to be the most critical design choice for handling the network connectivity for both scenarios. For this computational setup, the optimal choice is 100ms, but for other computational setups with more complicated computations on each microprocessor, the optimal choice may be higher.

5. Related Work

Distributed Program Execution

A common choice for distributed program execution is Stratosphere [1], which processes big data inputs and offers general programming on multiple levels. Stratosphere only supports one-time inputs, where continuous data input is not intended. That is why we developed our framework, which supports the opposite, with a continuous data input being the intended and supported choice.

Generalized Embedded Programming

The ReactiFi [8] language offers a reactive programming solution to generally programming microprocessors. However, it is limited to Wifi chips and access to their PHY, MAC, and IP layer. With the protocols and example implementations provided by the Arduino community, our framework can support any sensory input. The microprocessor only has to provide generally accessible pins, which widens the input scope from internal sensory inputs to supporting external sensory inputs.

Event Correlation

The combination of differently frequent events in a network environment presents a challenge on how to combine two event streams. The Versatile Event Correlation Paper [2] works explicitly on this problem, and the Join operator used in our framework is inspired by the combineLatest correlation approach, which is presented as one example of correlation in the paper.

6. Conclusions

This thesis paper looked at the IoT domain and how current solutions often pursue a centralized approach with microprocessors delivering sensory data to a central computing unit, which handles the combinatory logic. We then formulated the question of optimization potential, where microprocessors could potentially be used more effectively by offloading computations from the centralized computer onto these devices. To test this, we created a framework based on Python, which can be run on the esp32 microprocessor using MicroPython and can also be run on all computers supporting CPython. This framework allows programmatically distributing a computational setup onto variable sets of devices by defining computational nodes and executing these on the desired devices. These computational nodes allow for a continuous input stream from various supported sensors. In addition to supporting the standard value manipulation (filter, join, map), the delay and throughput can also be monitored using particular nodes. In the case study, we defined a computational setup that mimics a fire detection system. We then created three assignments that mimic different distribution amounts by executing either the sensory readouts, some additional computations, or the complete computational setup on the five esp32 microprocessors. To test variable conditions, we also defined the variables: time-frame (50ms/100ms), fixed/variable sending interval, and Join operator (JoinDupFilter/JoinNoFilter).

During testing, we found that the network connectivity is the significant bottleneck on esp32 microprocessors because they need to handle connections serially. It can impact the performance of the actual sensory computations. We thereby found two use cases, which can be recommended for future use. The first one keeps the centralized processing approach, but the initial computations can be offloaded to the microprocessors without losing sub-second responsiveness. The centralized computer handles the network connectivity and keeps the number of connections low on the microprocessors. The ideal configuration, which does not increase delay and keeps network activity low, is the 100ms time-frame, variable sending interval, and JoinDupFilter configuration. The second use case regards the fully distributed approach, where no centralized computer is needed. This assignment

is considerably slower, with a delay of 1.2 to 2.5 seconds, due to the increased number of network connections being handled by the microprocessors. It is, therefore, suitable for non-sub-second delay critical setups, where the absence of a centralized computer might be needed to ensure low power usage. This approach is also more susceptible to overload from network handling. We found that a minimum time-frame of 100ms is needed for our computational setup to ensure the absence of buffer overflows. The number of incoming values per second is just of secondary importance. The testing did not suggest an influence on delay from this variable. Our steady maximum delivered by one sensor was 100 values per second, and our minimum from another sensor was under one value per second.

7. Future Work

Four potentials and issues could not be addressed in this thesis.

The first one regards the sensory readout itself. In the case study, we only looked at one sensor per microprocessor. The framework handles multiple sensors serially, where during one cycle, every sensor is read after another. This serial readout was not applied in the case study but could be potentially optimized because a slow sensor will slow down the readout of a fast sensor. If the clock speed and execution times on the microprocessor are known, with a compiled language like C++, the readout of sensors could be combined because waiting on one sensor could be used to signal another sensor. The other option is to use the known speeds of sensory readouts to equalize the number of readouts so that the slow sensor gets only read half the time, and the fast sensor is read the rest of the time. This time division presents the optimal case of cutting both numbers of readout values per second only in half.

The second potential issue regards the network connections, which are handled serially in this framework. This serial handling greatly simplifies the network logic. It keeps a consistent state during each cycle because if a value is found, it is read fully in that one cycle and is available for further processing. The issue is that the network stack does not work serially and packages of one connection might arrive interleaved with packages from another connection. Therefore the current solution relies on the network buffer to hold values of a connection until it is read. This solution could be optimized by keeping a software buffer for each connection, which is filled for each connection, and a connection must therefore not return no or a full value. However, it can instead read every available chunk, and this is cycled for each connection until a connection has acquired a full value. Because of our network stack with a length stamped, JSON-encoded message, this has to be self-implemented, and this is no trivial task because correct error handling must be ensured.

The third potential optimization addresses the issue of the centralized computers not being powerful enough to handle all network connections and computations, thereby running in

the same bottleneck issue as the esp32 on a bigger scale with bigger computational setups. This bottleneck issue could probably be solved in two ways. The first solution might be to divide the network connectivity and computations into different processes, which can effectively communicate via inter-process communication. Inter-process communication has a lower overhead than network communication. The second possible solution is to have multiple computers assigned an equal number of network connections, thereby dividing the upcoming work.

The last potential regards the issue of networks being divided and some devices not being reachable by others. This reachability issue includes especially the problems of firewalls and one-way filters and is maybe solvable by bridge devices or proxy connections.

Bibliography

[1] Alexander Alexandrov et al. “The Stratosphere platform for big data analytics”. In: *VLDB J.* 23.6 (2014), pp. 939–964. DOI: 10.1007/s00778-014-0357-y. URL: https://doi.org/10.1007/s00778-014-0357-y.

[2] Oliver Bracevac et al. “Versatile event correlation with algebraic effects”. In: *Proc. ACM Program. Lang.* 2.ICFP (2018), 67:1–67:31. DOI: 10.1145/3236762. URL: https://doi.org/10.1145/3236762.

[3] *Home Assistant*. URL: https://www.home-assistant.io (visited on 09/10/2021).

[4] Andrey N. Kokoulin et al. “Hierarchical Convolutional Neural Network Architecture in Distributed Facial Recognition System”. In: *2019 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*. 2019, pp. 258–262. DOI: 10.1109/EIConRus.2019.8656727.

[5] *Node-RED*. URL: https://nodered.org (visited on 09/10/2021).

[6] Kapila K. Pahalawatta and Richard Green. “Particle Detection and Classification in Photoelectric Smoke Detectors Using Image Histogram Features”. In: *2013 International Conference on Digital Image Computing: Techniques and Applications (DICTA)*. 2013, pp. 1–8. DOI: 10.1109/DICTA.2013.6691509.

[7] Usha Satish et al. “Is CO2 an indoor pollutant? Direct effects of low-to-moderate CO2 concentrations on human decision-making performance”. In: *Environmental health perspectives* 120.12 (2012), pp. 1671–1677.

[8] Artur Sterz et al. “ReactiFi: Reactive Programming of Wi-Fi Firmware on Mobile Devices”. In: *Art Sci. Eng. Program.* 5.2 (2021), p. 4. DOI: 10.22152/programming-journal.org/2021/5/4. URL: https://doi.org/10.22152/programming-journal.org/2021/5/4.

A. Graphs

The legend is valid for the diagram on this page and all the following diagram pages.

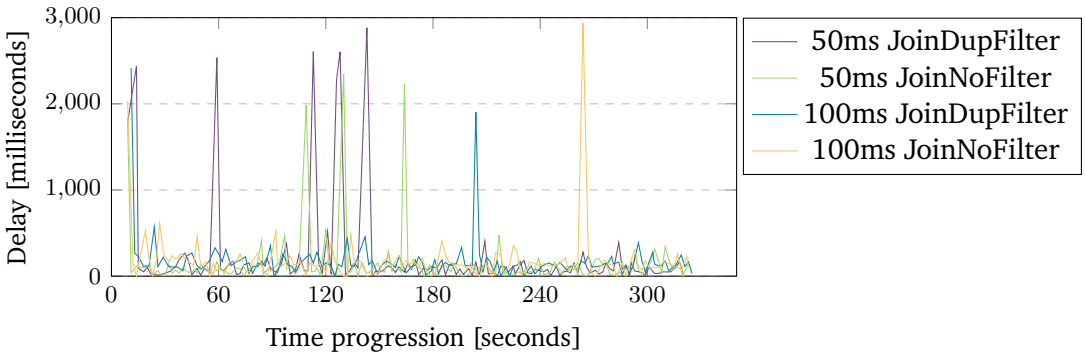

Figure A.1.: Delay measurement of assignment 1 with a fixed sending interval, without high value cutoff.

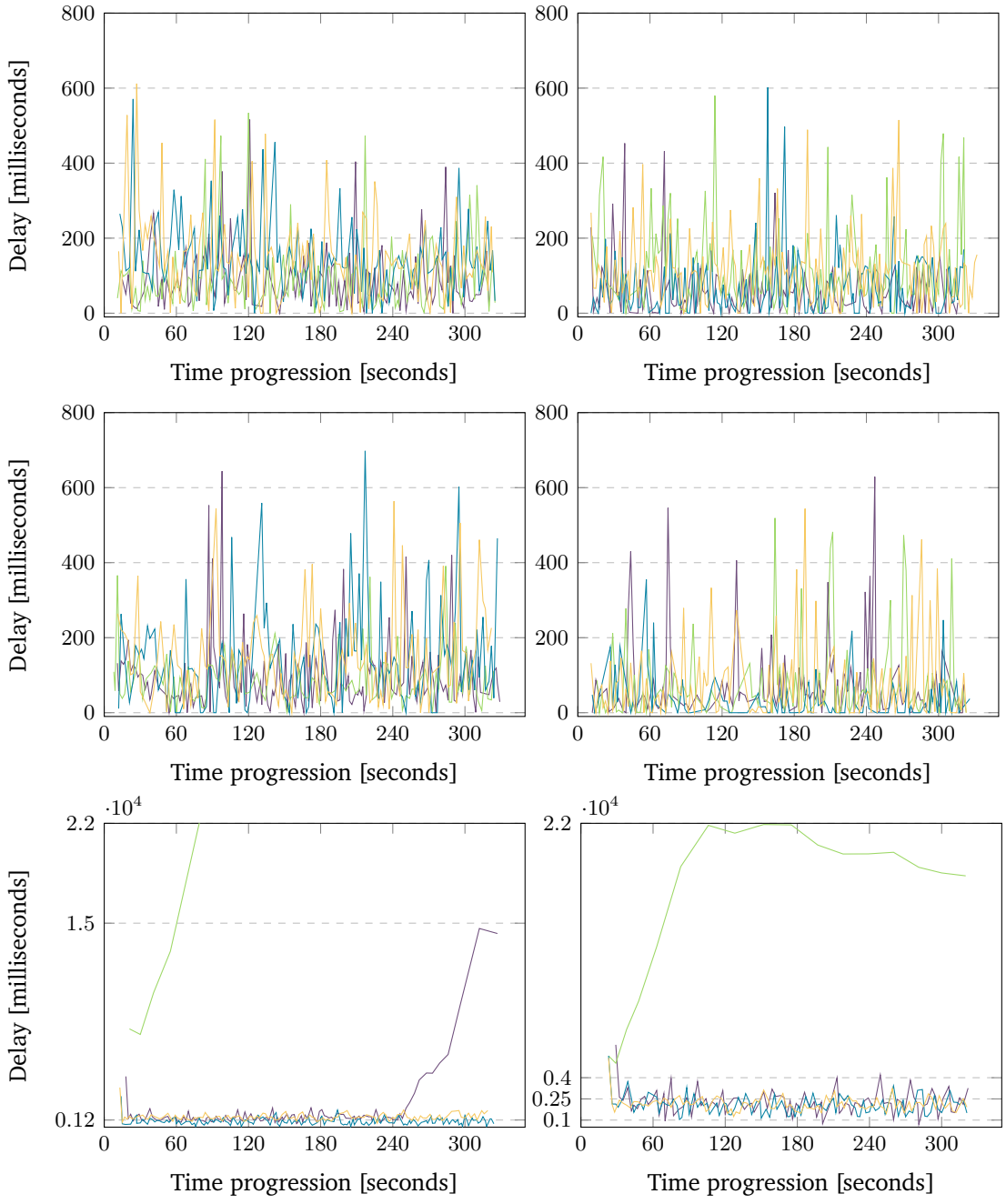

Figure A.2.: Delay measurements with the fixed sending interval left and variable sending interval right. Assignments from top to bottom: 1,2,3.

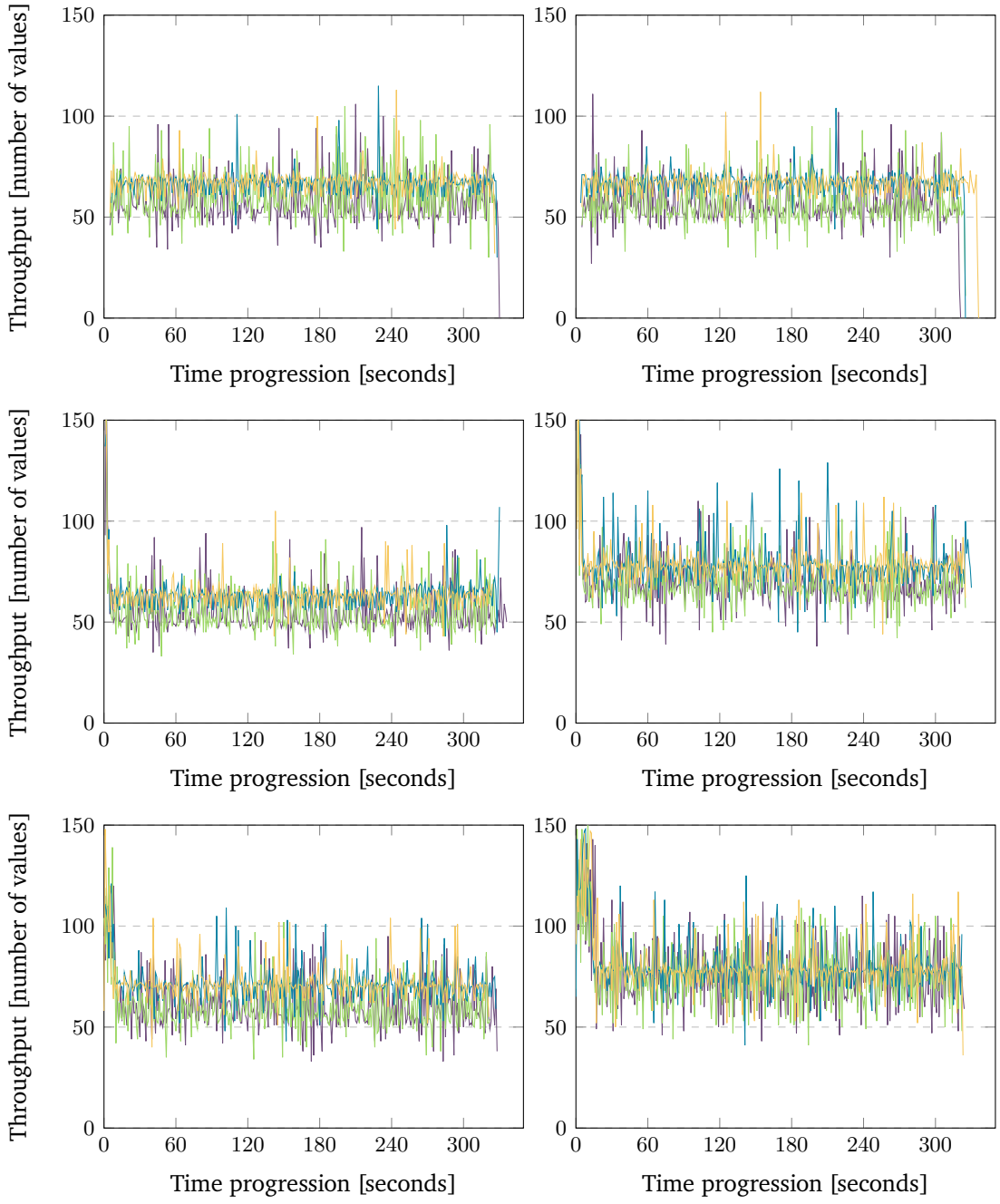

Figure A.3.: Button throughput measurements with the fixed sending interval left and variable sending interval right. Assignments from top to bottom: 1,2,3.

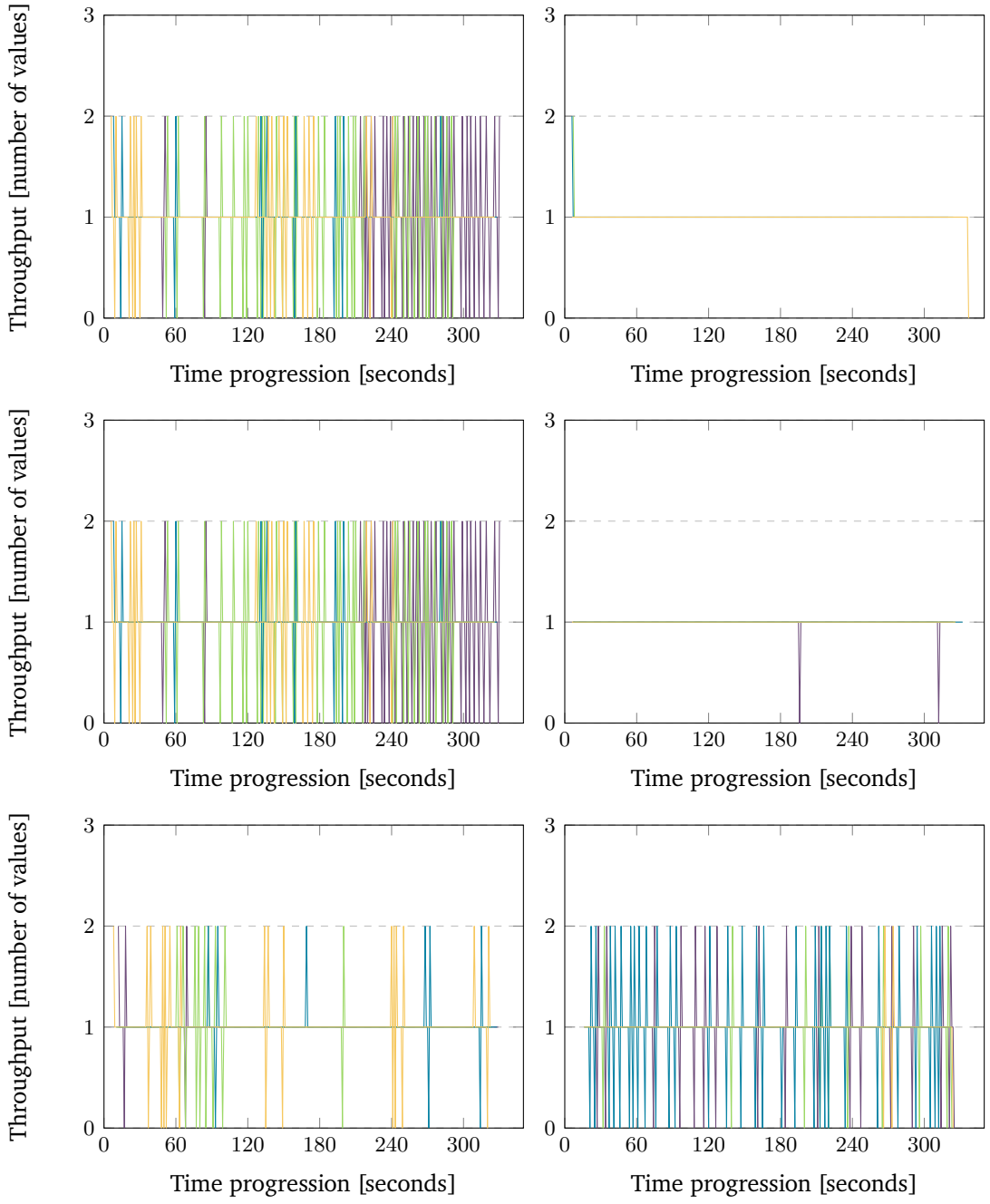

Figure A.4.: CO2 throughput measurements with the fixed sending interval left and variable sending interval right. Assignments from top to bottom: 1,2,3.

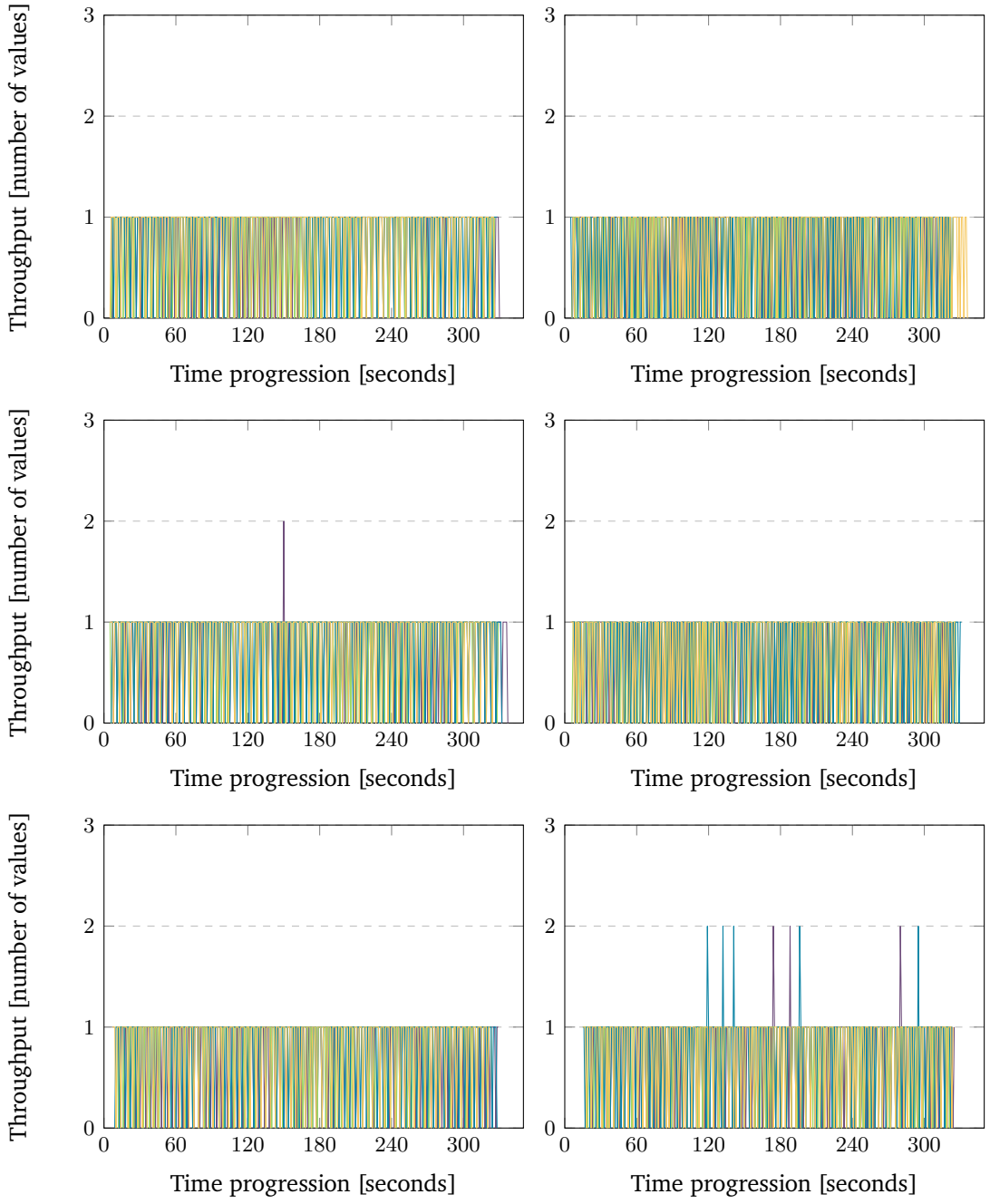

Figure A.5.: DHT11 throughput measurements with the fixed sending interval left and variable sending interval right. Assignments from top to bottom: 1,2,3.

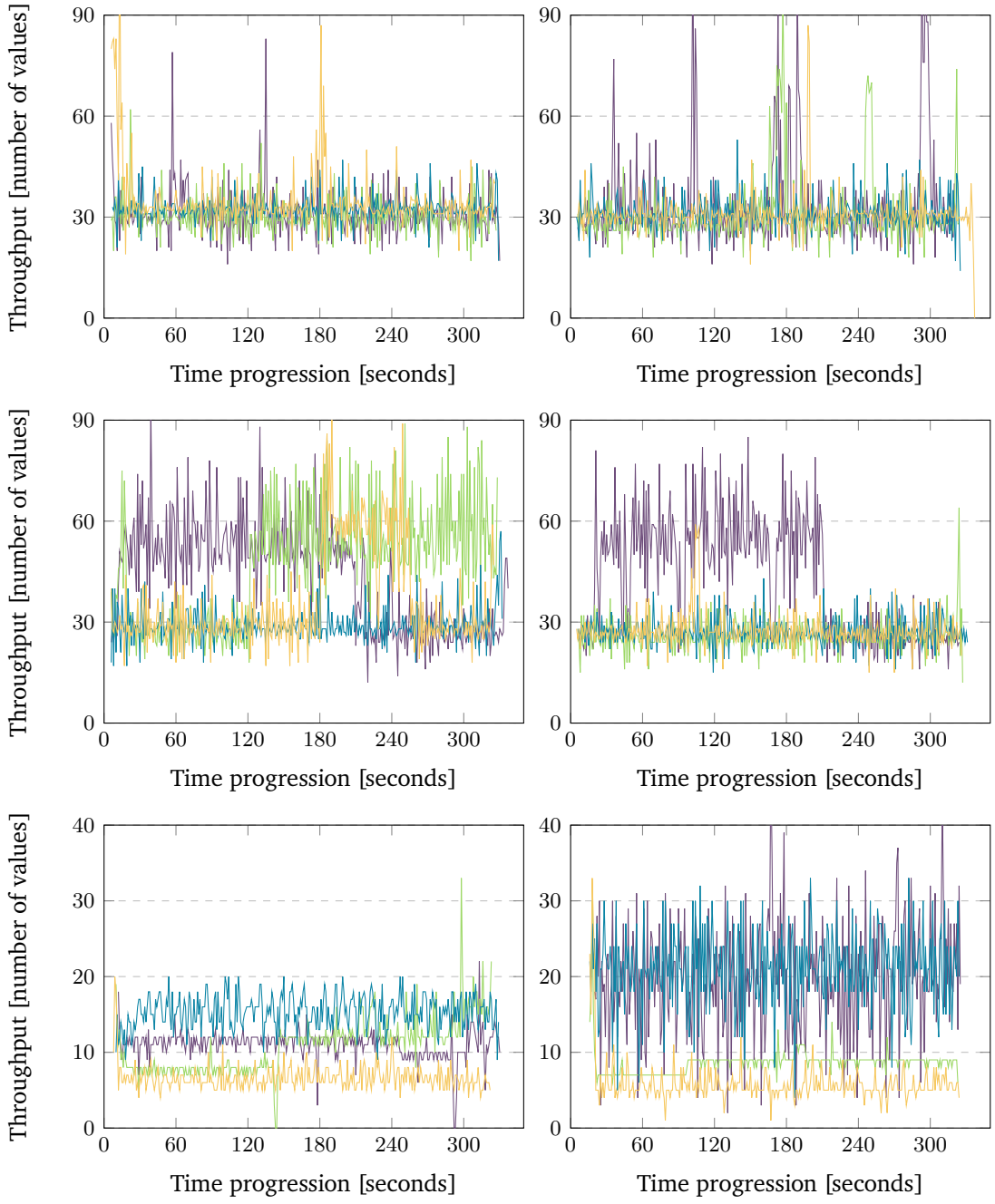

Figure A.6.: Ultrasonic throughput measurements with the fixed sending interval left and variable sending interval right. Assignments from top to bottom: 1,2,3.

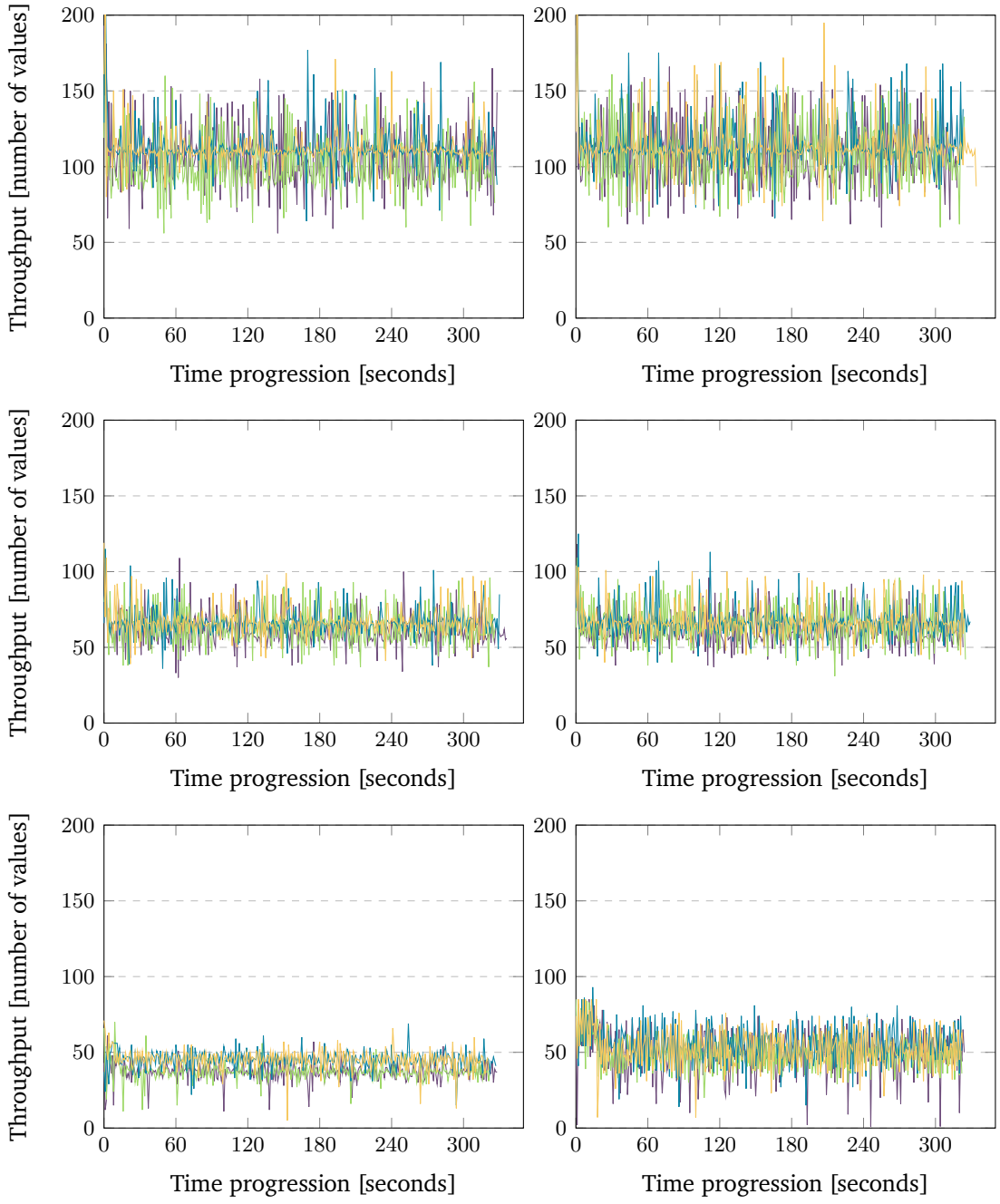

Figure A.7.: Rotary Encoder throughput measurements with the fixed sending interval left and variable sending interval right. Assignments from top to bottom: 1,2,3.

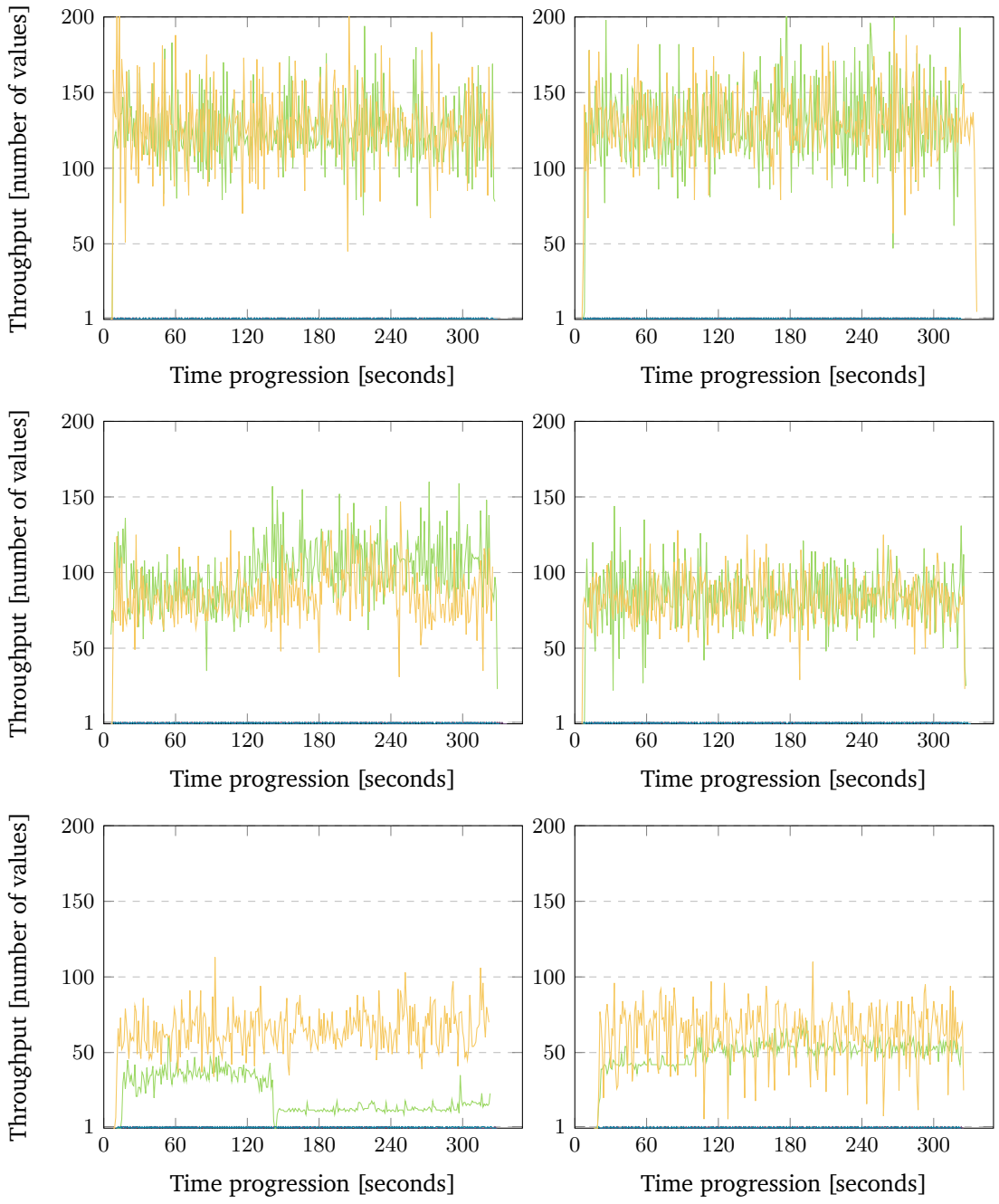

Figure A.8.: Total throughput measurements with the fixed sending interval left and variable sending interval right. Assignments from top to bottom: 1,2,3.