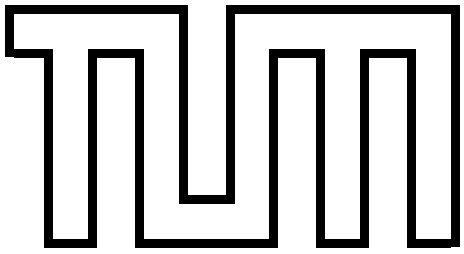


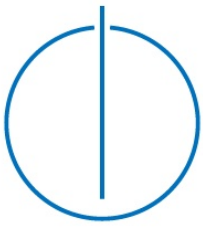
Technical University of Munich

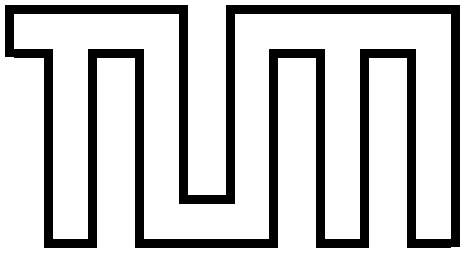
Department of Informatics

Bachelor's Thesis in Information Systems

Networked Dynamic Execution Environment for Microcontrollers

Fiona Guerin





Technical University of Munich

Department of Informatics

Bachelor's Thesis in Information Systems

Networked Dynamic Execution Environment for Microcontrollers

Vernetzte Dynamische Laufzeitumgebung für Mikrocontroller

Author: Fiona Guerin

Supervisor: Prof. Dr.-Ing. Jörg Ott

Advisor: M.Sc. Teemu Henrikki Kärkkäinen

Submission: 15.09.2018

I assure the single handed composition of this bachelor's thesis only supported by declared resources.

Munich, 15.09.2018

(Fiona Guerin)

Abstract

Applications in a programmable space compose functionalities from a set of heterogeneous microcontrollers. The latter must hence offer dynamic services consumable in generic contexts. Most services are orchestrated between domain-specific components and in static environments. We therefore present a networked dynamic execution environment that makes a microcontroller a dynamic service provider. It receives and performs scripts written in an extension language. The scripts initiate microcontroller requests to sensors, to actuators, and to data management systems. Our thesis proposes a new design of both services and programmable spaces.

Contents

Introduction

A programmable space builds on service orchestration. We define a programmable space as a set of devices such as computers, smart phones, and microcontrollers. A service is a well defined functionality that devices in a programmable space can offer to or consume from each other. A programmable space composes an application from multiple services. For example, a GPS system receives a signal from a satellite to determine its current position. It then publishes this information to location-based services. An application dedicated to petrol station comparison may for instance combine the vehicle positioning with the tank level and then navigate the car to the cheapest possible petrol station. See figure 1.1 for illustration.

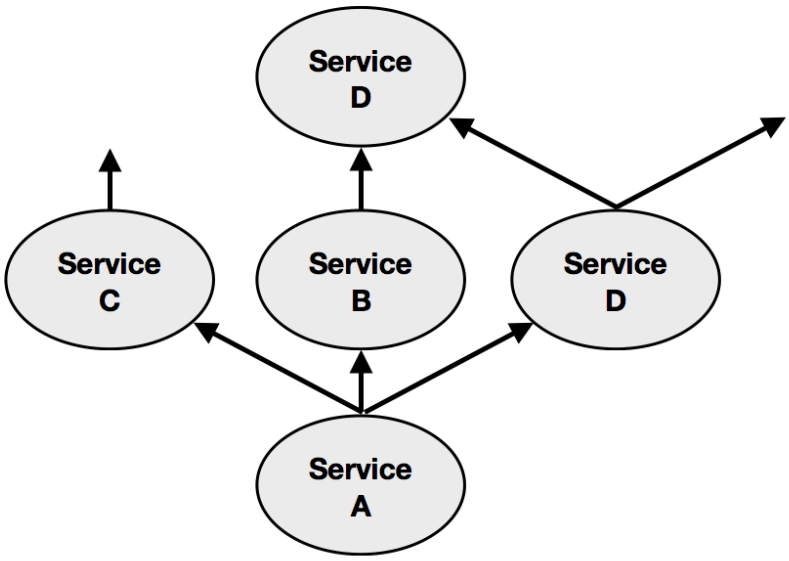


Figure 1.1: Service orchestration

1.1 Scope and Goals

We design the services in a programmable space dynamically, i.e. adaptable at runtime. First, the producer of a dynamic service declares which tasks it can perform. Taking this scope into account, a service consumer then defines a control flow that the service provider is to carry out. For example, an application may require a service provider to read from a thermostat, return its result and display the result. Instead of sending a sequence of requests, the application composes a script that the service producer executes in its entirety. It may thereby nest functions or it may convert a recording into another unit.

Common programmable spaces are based on static services. The control flow of a static service is defined at runtime and thus the output of such a function is influenced purely by its input. Service consumers do not attempt to define the internal processes of a static service provider, but interact through data transfer. In the Internet of Things, for example, an application moves data from its origin to a cloud, a fog, or the edge of a network to have it processed. Instead, dynamic services compute information in the system in which it is generated. See figure 1.2 for illustration.

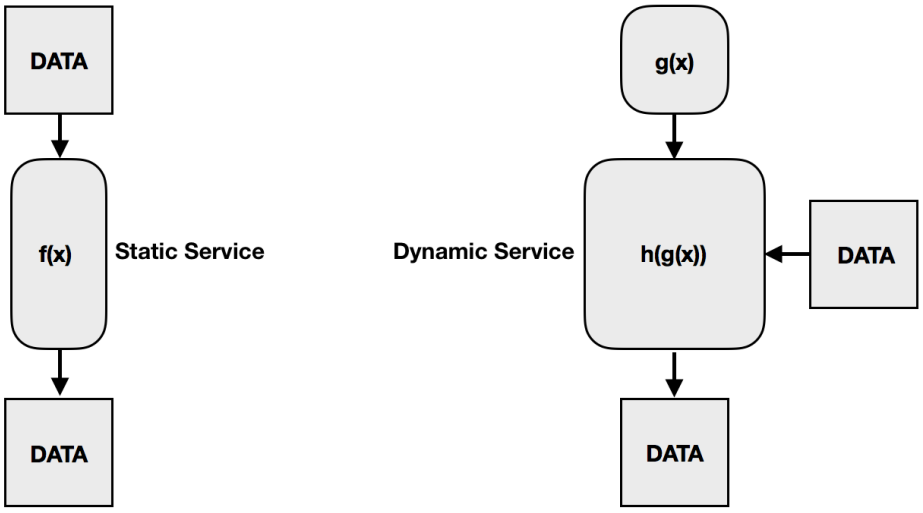


Figure 1.2: Sending functions to data

Our bachelor thesis introduces a networked dynamic execution environment for microcontrollers. A microcontroller is a small computer device to which we connect sensors and actuators. The networked dynamic execution environment runs on the microcontroller and offers dynamic services, i.e. it executes scripts from other applications. This allows the microcontroller to read from sensors and output to actuators both on demand (pull principle) and periodically (push principle). The runtime environment receives scripts, interprets them, executes them for all available hardware and responds to its requesting client. See figure 1.3 for illustration.

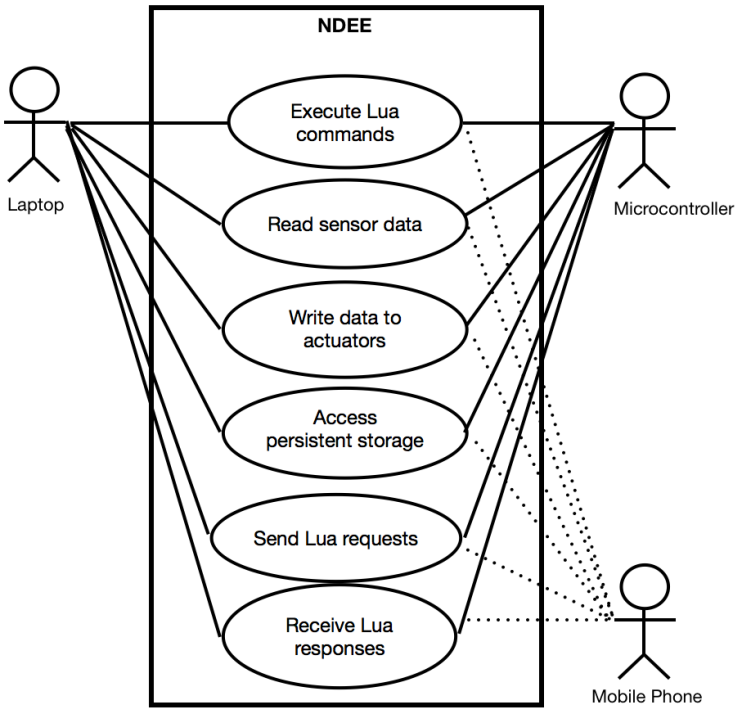


Figure 1.3: Use Cases

Our networked dynamic execution environment redefines both the concepts of services and of programmable spaces. Service providers now dynamically create their control flows and they interpret extension languages that they are not native to. We integrate the software into an embedded system with connected sensors, actuators and data management. Programmable spaces enable applications from any context to compose any services [7] via individual tailoring. For this, instead of passing data to functions, the applications send functions to data and actuators.

1.2 Thesis Structure

Our bachelor thesis is structured as follows: The chapter Background describes theoretical concepts behind our development, the chapter Design documents conceptual decisions on which the details of the chapter Implementation are based. We evaluate our results in the chapter Evaluation. A final Conclusion summarizes our work and suggests future research.

Background

We base our networked dynamic execution environment on state-of-the-art theory. Because it runs on embedded hardware, we learn about microprocessors, systems on a chip, and microcontrollers. We outline attachable sensors and actuators as well as virtual machines that abstract from the embedded hardware. Because the execution environment communicates over a network, we describe the networking standards Wi-Fi, Bluetooth, Zigbee, and LoRa. Because Lua scripts define the control flow of our application, we write about Lua as an extensible extension language to C.

2.1 Embedded Hardware

Embedded systems build an Internet of Things. Applications of the latter are environmental monitoring, autonomous driving, and human health control. [9] The former combine computer software and hardware, with the hardware board integrating sensors, actuators and data management. An Internet of Things designs an embedded system for data exchange and service exchange. For example, a cardiac pacemaker records a patient's heart rate and stimulates an appropriate heartbeat. The implant transmits the data to a doctor. [16]

The selection of the hardware board as well as the connected sensors and actuators depends on the system requirements.

2.1.1 Hardware Boards

Microprocessors serve as common components in microcontrollers and single board computers. A microcontroller integrates the processing unit, the control unit and the

memory unit of a classic processor on a single chip. It thus fulfills the usual functions of a processor, in particular the processing of elementary commands and interaction with peripheral components via data, address and control pins. [12] However, microprocessors cannot operate autonomously, i.e. without connection to external elements such as memories.

Systems on a chip run a program that is designed for user interaction. A system on a chip integrates multiple chip functionality into a single chip. Its included central processing unit, random access memory, and memory management unit allow for a complex operating system and graphical user interfaces. [26] For example, a user can surf the internet, receive e-mails, and simultaneously edit documents on such a system.

Microcontrollers run a program that is designed for a specific task. A microcontroller is a microprocessor that contains a memory and a small number of pins for input and output. It executes a program on its own. [12] [26] Due to the absence of a memory management unit within the microcontroller, such a program has a narrowly defined purpose and it includes little user interaction. For example, a microcontroller cannot run a Linux operating system, but it can monitor the room climate.

A system on a chip optimizes the flexibility of its hosted program, while microcontrollers are cheaper, smaller, and more energy efficient.

2.1.2 ESP32

The ESP32 is a market leading microcontroller. It has been developed by the Chinese company Espressif Systems and it is manufactured by Taiwan Semiconductor Manufacturing Company. The microcontroller can connect to personal, local, and wide area networks because it includes both a Wi-Fi and Bluetooth chip. Its Tensilica Xtensa LX6 microprocessor performs concurrent calculations because it is divided into two independent cores. [1] Espressif Systems have designed the ESP32 to withstand wide temperature ranges, be affordable, small, and power-efficient [10]. See figure 2.1 for illustration.

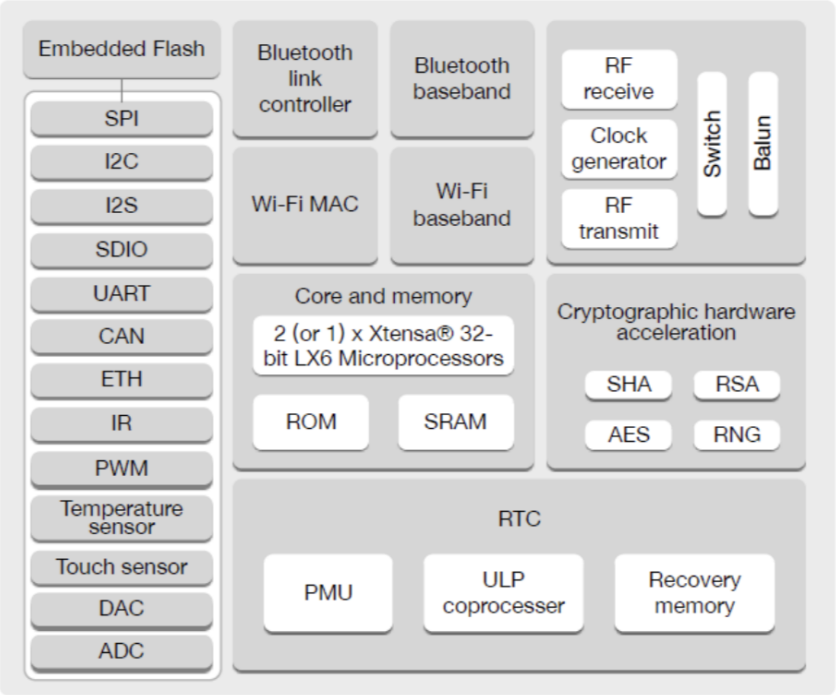


Figure 2.1: ESP32 [1]

The IoT Development Framework for ESP32 (ESP-IDF) converts all technical features of the ESP32 hardware into well-defined software interfaces. These interfaces make all higher software layers independent of the underlying microcontroller details and enable access via the C programming language. [1] For instance, the ESP-IDF provides methods for configuration, input from, and output to GPIO pins. A built-in WiFi module allows programmers to set up the ESP32 system for network communication, to select a WiFi connectivity configuration, and to interact among peers. An Analog to Digital Converter (ADC) component can attenuate analog signals and discretize values from continuous domains. The real-time operating system FreeRTOS provides basic kernel services such as scheduling and it is designed to respond instantly to external events and requests. [1]

2.1.3 Sensors and Actuators

Sensors convert ambient events into electrical signals. A sensor measures a specific parameter such as temperature, air pressure or geographical location. It reacts physically and/or chemically to a change in its measured variable. For example, the metal wire

of an NTC thermistor (Negative Temperature Coefficient) contains metal oxides. As the resistance of these semiconductor materials increases with falling temperature, the current I over the metal wire of the thermistor always corresponds to the ambient temperature T [19]. See figure 2.2 for illustration.

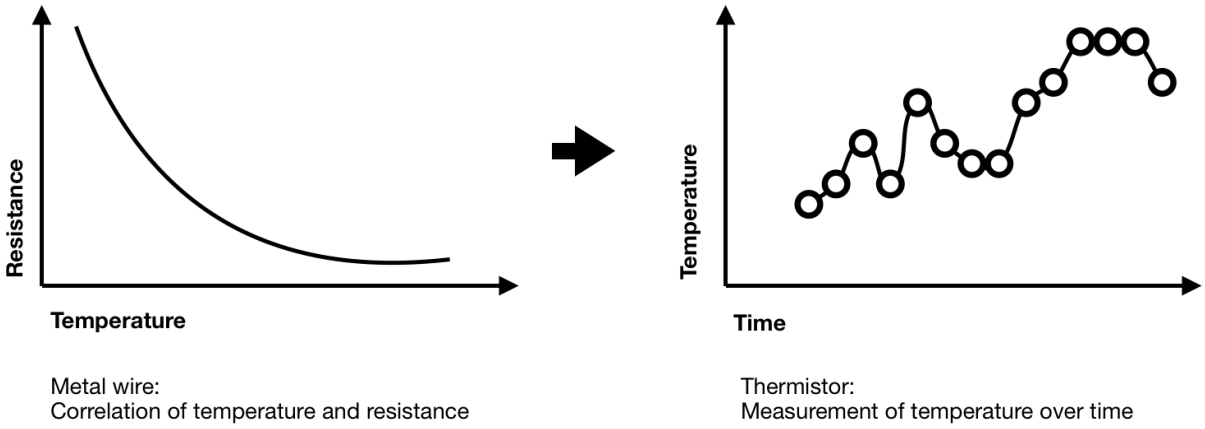


Figure 2.2: NTC thermistor

Actuators convert electrical signals into ambient events. An actuator, such as a motor, an LCD, or an OLED display, outputs data. It reacts physically and/or chemically to an electrical signal. For example, an OLED display internally consists of a large number of molecules. When a positive and a negative charge arrive at the molecule, they bring the molecule into an excited state. The molecule relaxes by dropping the negative into the positive, which causes it to emit a photon. [4] The OLED uses numerous of these molecules to create a colored image.

Actuators connect *the information processing part of an electronic control system with a technical or non-technical, e.g. biological process* [18]. For this, DIN defines a procedure and supports the actuator with a plant, a sensor, and a controller: An embedded system continuously adapts its outputs to a reference value. The plant directly manipulates the output of a system and a sensor then checks the conformity of this output with the reference value. If the sensor detects a deviation, the output of the system must be readjusted. To do so, the controller determines how the plant must reregulate the system. [18] For example, a car driver may wish for a specific interior temperature for his vehicle. Based on the driver's input and the current ambient temperature, an actuator decides how the vehicle's cooling system is activated. See figure 2.4 for illustration.

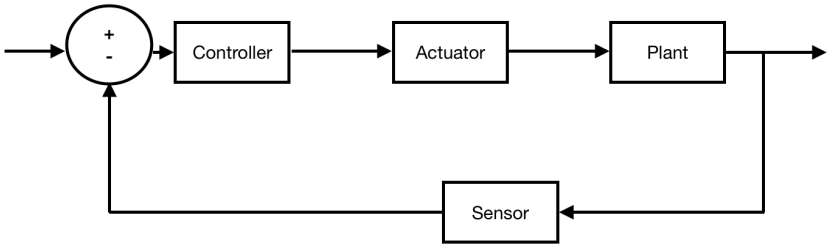


Figure 2.3: Control loop (adapted from [18])

2.2 Lua

An Internet of Things often requires the use of an extensible extension language: Microcontrollers as the basic components in an Internet of Things typically perform one defined function, such as the measurement of elementary environmental data. User applications are thereby often a combination of the functionalities of several heterogeneous microcontrollers. Therefore, a microcontroller must be configurable for higher applications. This corresponds either to the setting of preferences or even to the expansion of the microcontroller's capabilities.[23] To achieve the latter, applications send scripts to a microcontroller that augment its functionality. Since the scripts extend the proprietary programming language of the microcontroller, they are written in languages called extension languages. Extension languages must

- be simple because the scripts may be written by non-programmers;
- be small because the scripts must not be costly to add to an existing programming environment;
- have good information description facilities to handle measurement data from the microcontroller; and
- be extensible because the concrete execution of a script may depend on the characteristics of its underlying microcontroller. [23]

The scripting language Lua has been designed as an extensible extension language. Lua combines features of a data description language with procedural elements such as coroutines and conditions, while meeting the above requirements of being small, simple and extensible. For example, Lua represents its data elements mainly as tables. [23] A

table is an associative array that can be created and manipulated at runtime and can also refer to other tables. Due to Lua's leanness, the language interpreter consists of just 500 lines of code.

In its internal implementation Lua has a good interworking with programming language C: The interpreter of the scripting language Lua is a lean program written in C. It receives scripts and files via a user interface, which it then passes on to a Lua library. The Lua library, also a C program, takes over the actual execution. [14] The Lua code is first compiled into a bytecode that is then executed on a virtual machine. [22] The Lua interpreter can then return the generated output via the user interface. The bytecode represents Lua constructs as elements of the programming language C. For example, Lua values correspond to a C struct that contains the data type of the value and the actual value [22]. See figure 2.4 for illustration.

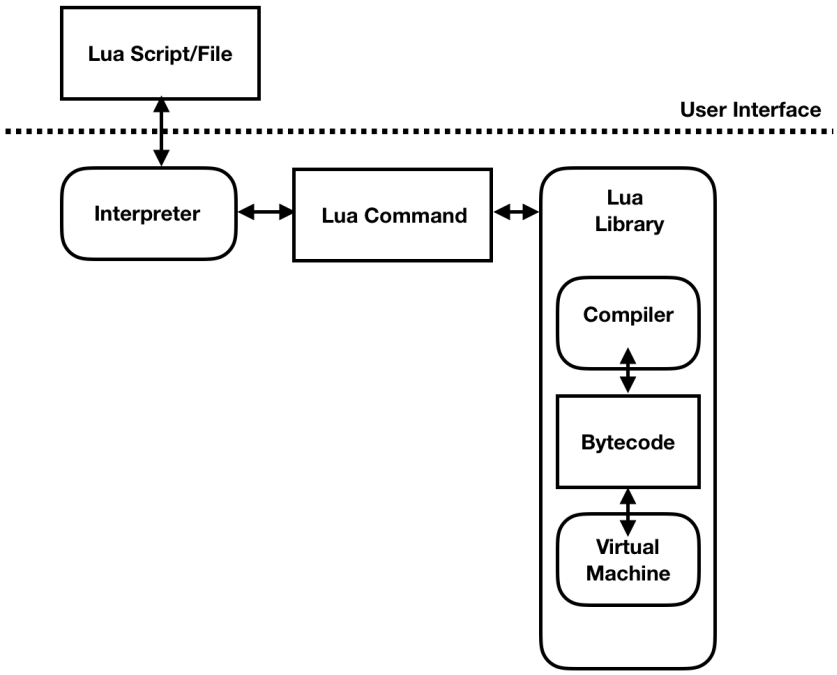


Figure 2.4: Lua: Compilation and Execution

2.3 Microcontroller Networks

Microcontroller communication forms the basis for an Internet of Things. Several microcontrollers are connected either via a cable or wirelessly. Examples of transmission media are infrared waves and radio waves, which in turn operate standards such as Wi-Fi and Bluetooth. Each standard optimizes different network properties based on its individual attributes.

2.3.1 Network Standards

Wi-Fi, Bluetooth, and Zigbee are wireless communication standards as specified by the Institute of Electrical and Electronics Engineers (IEEE). They operate on the same frequency band of 2.4 GHz [5]: When devices interact wirelessly, they communicate via electromagnetic waves. Each wave can be characterized by its wavelength and the frequency at which it occurs. Wi-Fi and Bluetooth are based on radio waves ranging from 3 kHz to 300 GHz. The radio waves are in turn segmented into bands of different frequencies, which enables the coexistence of different wireless applications. Both standards operate on the 2.4 GHz ultra-high frequency band [24] [25] [11]. See figure 2.5 for illustration.

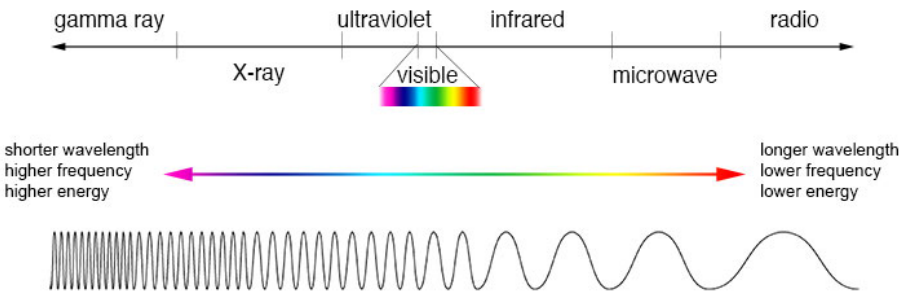


Figure 2.5: Electromagnetic spectrum [2]

LoRa is a wireless communication technology owned by Semtech. It operates on MHz-radio frequency bands which, for example, range from 433 MHz – 434 MHz in Germany and from 863 MHz – 870 MHz in Europe [20].

Wi-Fi is a technical standard for local area networks and for wide area networks. Wi-Fi is characterized by its high range of 50 — 100 m, its high data rate of 54 Mbps, and its

high power consumption [25] [11] [5]. Wi-Fi connects almost any number of devices either in a local network using ethernet or in a global network via routers.

Bluetooth is a technical standard for personal area networks. Bluetooth is characterized by its low range of 10 — 100 m, its low data rate of 1 Mbps, and its medium power consumption [25] [11] [5] [20]. Bluetooth connects 2 — 8 local devices in a private network.

Zigbee is a technical standard for personal area networks. Zigbee is characterized by its low range of 10 — 100 m, its low data rate of 25 Mbps, and its low power consumption. It optimizes a private network for light-weight and energy-efficient data transfer [5].

LoRa (Long Range) is a technical standard for long range wide area networks. LoRa is characterized by its high reach of 2 — 40 km, its low data rate of 1 Mbps, and its low power consumption [20]. It optimizes on the coverage of a wide area network.

Wi-Fi, Bluetooth, Zigbee, and LoRa differ in their spread spectrum technology. Using the latter, we define the spread of a signal over its frequency domain and we aim to transfer more signals securely and without interference. While Bluetooth depends on frequency hopping spread spectrum technique, Wi-Fi and Zigbee use direct sequence spread spectrum technique, and LoRa is based on chirp spread spectrum technique. In frequency hopping, a signal regularly changes — it hops between — multiple frequencies. Direct sequence spreading exclusive-or links a signal to a pseudo numerical number and hence increases the signal bandwidth. Chirp spread spectrum technology builds on signals, so called chirps, whose frequency increases or decreases continuously over time [8]. See table 2.1 for illustration.

	Wi-Fi	Bluetooth	Zigbee	LoRa
Range	High	Low	Low	High
Data rate	High	Low	Low	Low
Power consumption	High	Medium	Low	Low
Frequency band	2.4 GHz	2.4 GHz	2.4 GHz	MHz
Spread spectrum	Direct sequence	Frequency hopping	Frequency hopping	Chirp

Table 2.1: Network standards

The Bluetooth specification can be divided into the two technologies classic Bluetooth and Bluetooth Low Energy, which are optimized for different application contexts. Wi-Fi enabled devices can function as both stations and access points. Accordingly, different types of Wi-Fi networks can be formed within the Wi-Fi standard - including, for example, ad-hoc networks and infrastructure networks.

2.3.2 Classic Bluetooth and Bluetooth Low Energy

While classic Bluetooth serves bidirectional communication, the main use case of Bluetooth Low Energy is unidirectional beaconing. Classic Bluetooth networks are scatternets, i.e. they connect two devices one-to-one. Bluetooth Low Energy networks are stars, i.e. they connect one master node to multiple slave nodes [13]. Hence, in contrast to classic Bluetooth, Bluetooth Low Energy declares one device as a data requester and all other devices as its respondents.

2.3.3 Wi-Fi Stations and Wi-Fi Access Points

A station can connect to a Wi-Fi network. Thus, all devices count as stations that have a so-called Network Interface Controller, i.e. a Wi-Fi adapter. Accordingly, laptops, smartphones and some microcontrollers can serve as stations.

An access point either improves the connectivity of an existing network or it sets up its own network in the role of a master. For this purpose, there are various modes in which the access point can be configured. If the access point represents the master of a network, it does so in AP mode, in which several clients connect to the access point and cooperate via the resulting network. In repeater mode, the access point is connected to a local Ethernet network. By amplifying incoming signals for the Ethernet network, the access point increases the range of the corresponding network. In addition, two or more access points can collaborate in their tasks if they operate in wireless client or wireless bridge mode.

2.3.4 Wi-Fi Ad-hoc Networks and Wi-Fi Infrastructure Networks

An ad-hoc network connects two devices directly to each other. This means that such a network consists of exactly two nodes and their point-to-point connection. Such a node represents a Wi-Fi-enabled device that can serve its communication partner in both station and access point mode [21]. A short example: User A may send a photo via his mobile phone to a user B's mobile phone. Mobile phone A thereby functions as the access point and mobile phone B as the station.

An Infrastructure network connects a server with several clients. The server is explicitly configured as an access point and the clients as stations. This results in a set of linked

devices. This quantity is also known as a basic service set (BSS) and is identified using a BSSID. The access point sends out radio signals, so-called beacons. Devices in the vicinity receive these beacons and can use them to connect to the access point. Several basic service sets can be combined to form an extended service set (ESS) [21]. To join such an extended service set, a new station connects to exactly the access point within the extended service set with the best possible connection quality.

Design

In a programmable space, applications orchestrate microcontroller services. A microcontroller can record information about its environment via its attached sensors and it can output data to its environment via its attached actuators. The device thereby has one elementary purpose. For example, a microcontroller A may measure its ambient temperature, a microcontroller B may display numbers on an actuator, and a microcontroller C may detect infrared signals. Applications in the programmable space have a more complex purpose. An application 1 may predict the weather and an application 2 may navigate a car. Such applications compose their high-level operations from microcontroller functions. See figure 3.1 for illustration.

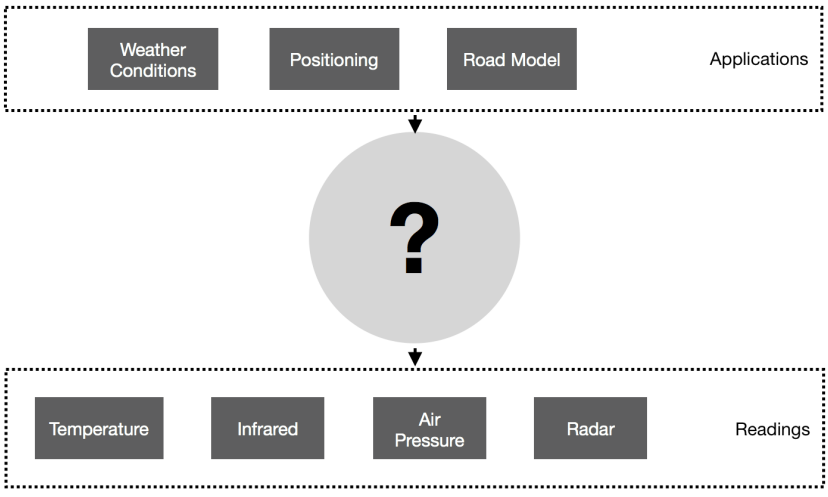


Figure 3.1: Service orchestration

A central environment model mediates between applications and microcontrollers. Since

the latter can be unreliable, heterogeneous, and/or proprietary, the central environment model serves as an abstraction. It internally defines elementary data types such as temperature, humidity and position for which it must hold values. The central environment model then prompts the underlying microcontrollers to capture the requested data, process it as desired and return it. For example, the model may require a reading of the current air pressure measured in bar units: First, it connects to all microcontrollers and learns which of them are able to determine air pressure. Secondly, it selects a suitable microcontroller, which in this case senses the air pressure in the unit of Pascal. Accordingly, the central environment model provides a script that converts pressure measurements from Pascal to pressure measurements in bar. Finally, it sends the script to the chosen microcontroller. The microcontroller reads out the air pressure locally, carries out the translation to bar, and retransfers its result to the central environment model. Thus it now has a current pressure value in the unit bar. The central environmental model stores elementary data. Complex applications in the above layer consume this information. See figure 3.2 for illustration.

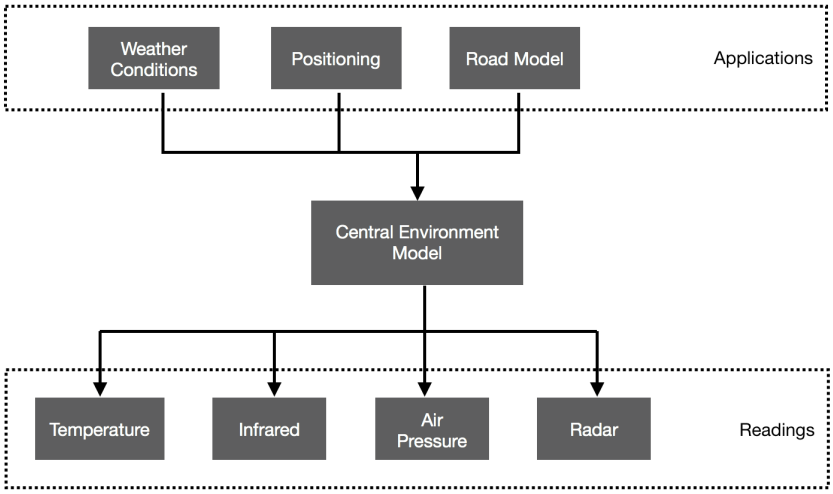


Figure 3.2: Central Environment Model

We design an embedded system as a provider of dynamic services. An application sends it a script via its network connection, an execution environment interprets and carries out the script at runtime, and the embedded system responds via the respective network connection. The execution environment accesses drivers of attached sensors, actuators, and data management. For example, a local backend may make the embedded system read from a barometer, display the measurement on an LCD, put it in a local database,

and send it to the cloud. See figure 3.3 for illustration.

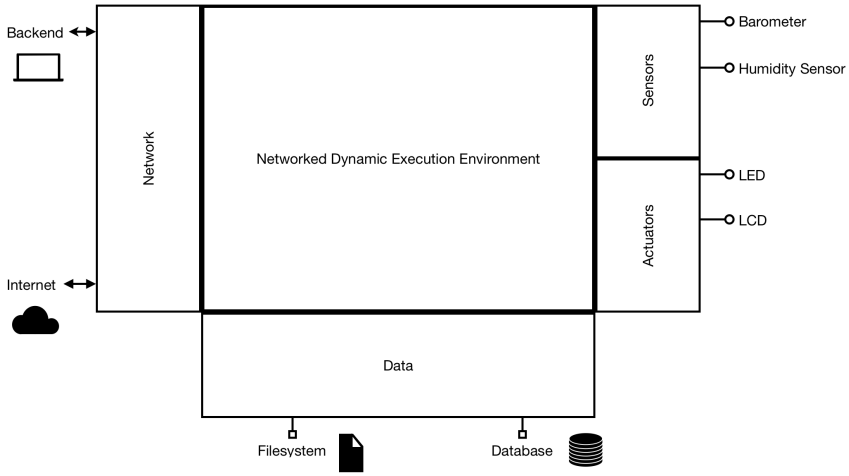


Figure 3.3: Components of a dynamic service provider

3.1 Hardware

We select an ESP32 microcontroller as the hardware platform of the dynamic service provider. The application serves as an execution environment, i.e. has a narrowly defined purpose. When we base the dynamic service provider on a microcontroller, we can additionally meet strict requirements for cost, size and power consumption.

3.2 Network

We configure a dynamic service provider as a Wi-Fi access point. Using Wi-Fi allows for more concurrent client connections than Bluetooth. In some application contexts a connected client rarely communicates with the dynamic service provider. For example, when recording the room climate, a client may receive a temperature measurement once a day. If only a few clients request a service from the system, it is often idle. The use of Wi-Fi is therefore in line with an intelligent use of resources. Since the service producer functions in the role of a server, we configure it not as a station, but as an access point. Clients can hence contact the dynamic service provider directly and they are also independent of the availability and reliability of external WLAN signals.

3.3 Lua

Service orchestrations from a dynamic service provider can be defined in the script language Lua. Lua can be embedded into other programming languages, especially into C. Lua has a simple and powerful C API for this. Lua scripts can be easily created, efficiently compiled and executed, as well as ported to other platforms without changes. Lua uses a very small number of language constructs, has a small core, but can be supplemented by libraries at any time. [22]

3.4 Execution Environment

The execution environment orchestrates dynamic services from static services. It consists of a static service provider, a dynamic service provider and a coordinator. The coordinator receives a script from the network, executes the script in the dynamic service provider, and returns the results over the network. The dynamic service provider executes the function sequence in the script by consuming each function as a static service from a static service provider. The static service provider groups its services by their purpose into a native Lua environment, a pull environment, and a push environment. See figure 3.4 for illustration.

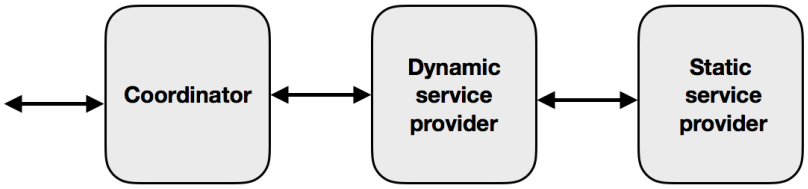


Figure 3.4: Execution Environment

3.5 Pull and Push

A pull environment provides services on demand: When a client requests the pull environment to read from a sensor or to output to an actuator, the pull environment

performs the required task once and returns the result immediately. See figure 3.5 for illustration.

A push environment periodically performs services. It reads from sensors at defined intervals, it outputs data to actuators at defined intervals, and it sends information to external consumers at defined intervals. A client can initiate such a service, register for it and terminate it. For example, a push environment may record the temperature hourly. A new client can then register for an hourly temperature consumption. See figure 3.5 for illustration.

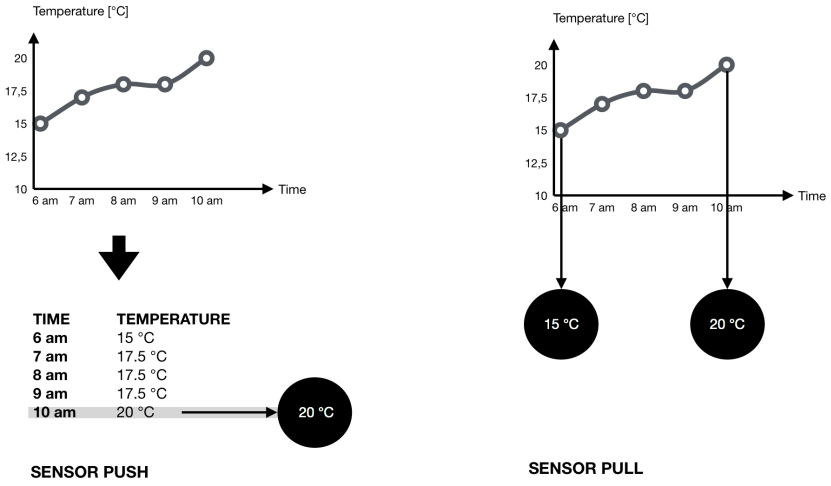


Figure 3.5: Pull and push

We select pushing data over pulling it if clients query the data more often than it is published. This applies to information that is regularly consumed, simultaneously consumed and/or expensive to produce. For example, a sensor reading may require 10 seconds of processor time, the reading may be published every 50 seconds, and external entities may request the reading every 5 seconds. While pushing uses 20% of the processor power, pulling requires 200% and would overload the system. See table 3.1 for illustration.

	PUSH	PULL
Frequency of consumption	Periodic, frequent measurements	Rare, one-off measurements
Number of consumers	Multiple consumers	Few, one consumer
Cost of measurement	Expensive to process and store	Cheap to process and store

Table 3.1: Trade-off between pulling and pushing

A client of our system chooses how the execution environment generates data. An external can pull information, it can request pushing information, and it can define both the frequency and the duration of the push. For example, an application that requires one-time information about air pressure selects pulling. By pushing, another application may receive eight hourly temperature measurements.

3.6 Lua Scripts

A client can call functions that are native to Lua. The Lua virtual machine compiles native functions without external libraries. When a Lua function requires an input, it can compose a function that produces an output.

A client can call functions defined in the pull environment. Via such functions, an application requests the networked dynamic execution system to read from a sensor or output to an actuator on demand. A script can also nest a sensor function into an actuator function. For example, a function that measures temperature may be embedded into a function that displays the respective temperature recording on an LCD.

A client can call functions defined in the push environment. Via such functions, an application requests the networked dynamic execution system to periodically read from a sensor, to periodically pass recordings to an actuator, or to periodically send the client such recordings. A script can nest a sensor function into an actuator function and it can also nest a sensor function into a client function. For example, a function that pushes air pressure in a storage may be embedded into a function that pops the recording from the storage and displays it on an LCD. See figure 3.6 for illustration.

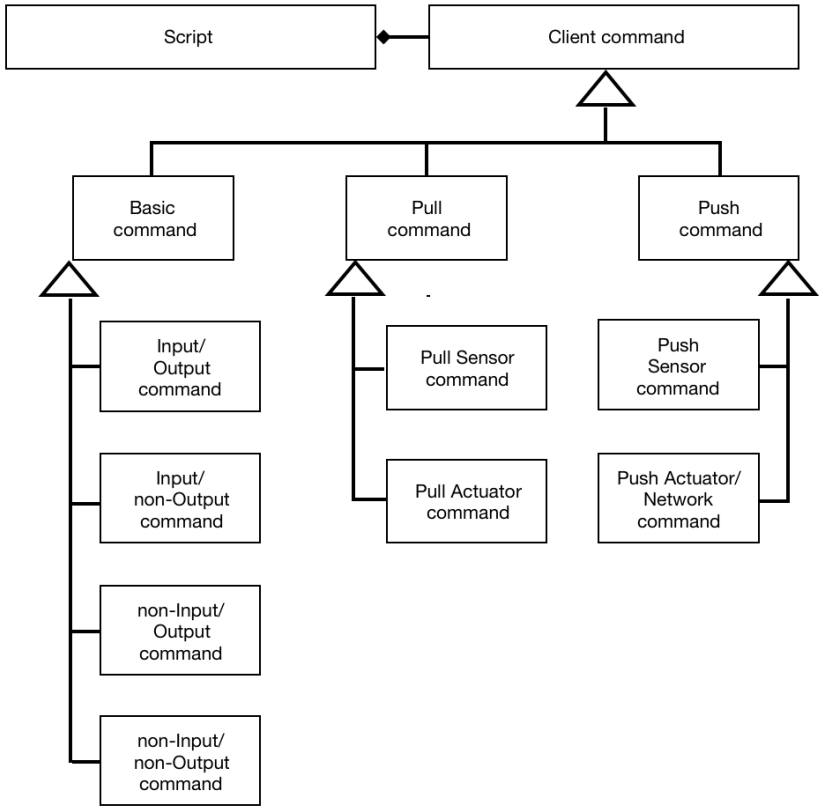


Figure 3.6: Lua script

Implementation

The following chapter documents how we implement the networked dynamic execution environment. We examine the system as a black box, i.e. from an external perspective, and as a white box, i.e. from an internal perspective.

4.1 Black Box Behavior

Every function in a client script specifies its targeted service and the arguments to pass to the service. If a client wants to consume a service from the native Lua environment, the client calls a native Lua function. If a client wants to consume a service from the pull environment and the client demands a sensor reading, it declares the sensor, the addressed measurand, and the id of its request. If a client wants to consume a service from the pull environment and the client demands output to an actuator, it declares the actuator, the output value, and the id of its request. If a client wants to consume a service from the push environment and the client requests periodic production, it specifies its targeted measurand as well as the frequency of measurement, the duration of measurement, and the id of the request. If a client wants to consume a service from the push environment and the client requests periodic consumption, it selects itself or a defined actuator as the consumer as well as the frequency of the measurement, the duration of the measurement, and an id of its request. See figure 4.1 for illustration.

Native Lua environment	Push environment	Pull environment
4 + 5	Push temperature ID: 1 Frequency: 200 Duration: -1	Pull pressure ID: -9
Print („Hello World“)	Push to OLED ID: 2 Frequency: 3008 Duration: 99 From stack: 1	Output to LCD ID: 2 Value: 70 °C
80	Push to client ID: 1 Frequency: 10 Duration: -1 From stack: 4	

Table 4.1: Example functions

As a response, the client receives a status message. In case of a failure in the script execution, the status message notifies the client about the source of fault in the script. For example, error code 6 indicates a function argument out of scope. Otherwise, the message acknowledges correct execution and optionally contains returned data. An OK confirms that a value has been output to an actuator on demand or that the push environment has initiated its periodic push to a client or actuator. A STACK confirms the periodic sensor recording in the push environment and specifies the id of the respective storage stack. Pulling from a sensor returns a VALUE. VALUEs contain sensor recordings. See table 4.2 for illustration.

Status message	Status code
OK	0
VALUE	1
STACK	2
ERROR: Script not compilable	3
ERROR: Sensor/Actuator failed	4
ERROR: Bad argument declaration	5
ERROR: Argument out of scope	6
ERROR: Missing return statement	7
ERROR: Requested stack does not exist	8

Table 4.2: Status messages

The dynamic service provider adds metadata to elementary sensor data. A sensor measures the value of a physical quantity, for example a temperature sensor may measure the value '20.5'. However, such a value can only be processed sensibly if it is complemented by its context. In particular, the measurand, the sensor and the corresponding measuring unit must be known. In addition, the data type of the measured value is required for further machine processing. Each sample should also be supplemented with a time stamp - especially if the associated measurements are repeated, for example in a Push environment. This time stamp represents the current Unix time, i.e. the number of milliseconds since January 1, 1970. An object corresponding to the above example may therefore be 'Temperature, Temperature sensor no. 0, °C, 1535034175505, Float, 20.5'. See table 4.3 for illustration.

Measurand	Device	Unit	Time Stamp	Type	Value

Table 4.3: Sensor object

Such a sensor object is returned to the client as a specific bit code: The networked dynamic execution environment defines a protocol with the help of which it maps each field value of a sensor object uniquely to a bit code of defined length. For example, the protocol

specifies the measurand temperature as eight zero bits (0000 0000). The numerical value of a measurement is converted into a sequence of 4 bytes using methods that are common for the Integer32 or Float32 formats. A Unix timestamp represents an integer 64 data type and is therefore described by 8 bytes. The complete protocol is shown below. See tables 4.4, 4.5, and 4.6 for illustration.

Measurand	Device	Unit	Time Stamp	Type	Value
1 B	1 B	1 B	8 B	1 B	4 B

Table 4.4: Decoded sensor object

Measurand	**Bit representation**	**Unit**	**Bit representation**	**Device**	**Bit representation**
Temperature	0000_0000	°C	0000_0000	TempSens0	0000_0000
		… (°F, etc.)	[0000_0001; 1111_1111]	… (TempN, etc.)	[0000_0001; 1111_1111]
CO_2	0000_0001	ppm	0000_0000	CO_2Sens0	0000_0000
		… (%, etc.)	[0000_0001; 1111_1111]	… (CO_2N, etc.)	[0000_0001; 1111_1111]
Humidity	0000_0010	g/m³	0000_0000	HumSens0	0000_0000
		… (%, etc.)	[0000_0001; 1111_1111]	… (HumN, etc.)	[0000_0001; 1111_1111]
Air pressure	0000_0011	bar	0000_0000	AirSens0	0000_0000
		… (Pa, etc.)	[0000_0001; 1111_1111]	… (AirN, etc.)	[0000_0001; 1111_1111]
… (O_2, etc.)	[0000_0100; 1111_1111]				

Table 4.5: Protocol

Type	**Bit representation**
Integer	0000_0000
Float	0000_0001
… (Short, etc.)	[0000_0010; 1111_1111]

Table 4.6: Protocol

4.2 White Box Behavior

A dynamic service provider always runs multiple modules in parallel: A network module cooperates with one or more clients. Through it, a client can connect to the system at any time. Clients concurrently send the system Lua scripts which the network module inserts into a script queue. A coordinator component serves as the runtime. It waits for a new script to become available in the script queue and then calls its embedded Lua interpreter to execute the script. For this, the Lua interpreter may consume services from the native Lua environment, the push environment, and/or the pull environment. The respective environment returns its result to the Lua interpreter and the Lua interpreter forwards it to the coordinator component. The coordinator sends the result to the requesting client via the network module. See figure 4.1 for illustration.

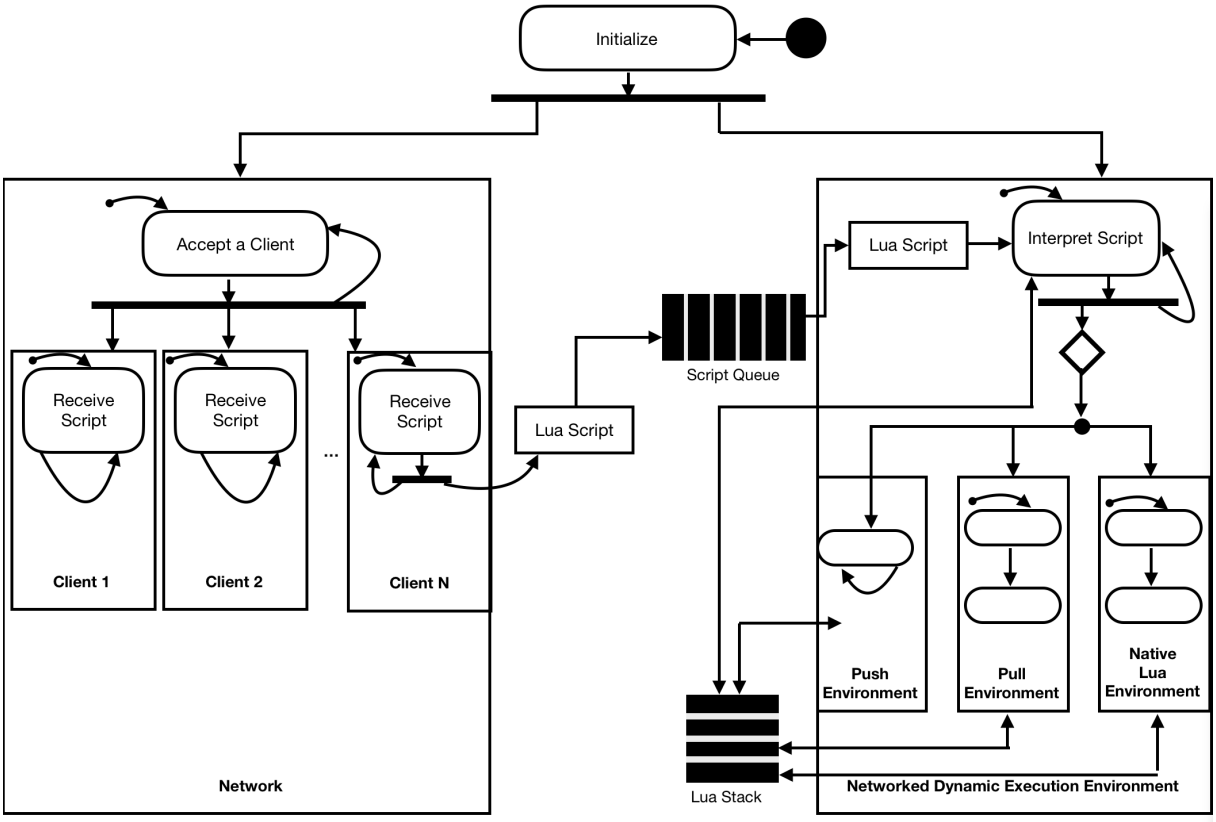
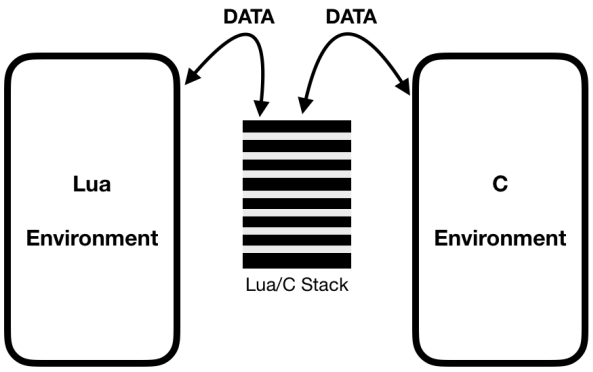


Figure 4.1: Core of the networked dynamic execution environment

4.2.1 Lua/C API

The Lua/C-API makes Lua an extensible extension language to C. The API is a stack and it stores both functions and variables. When C embeds a Lua script, C pushes the values and functions from the script onto the stack. The Lua virtual machine pops the instructions from the stack, carries them out, and pushes the results back on the stack. The Lua environment returns a status bit to the C environment. If the Lua virtual machine could run without failure, the C environment eventually pops its outputs from the stack. Lua embeds C by linking Lua functions to C functions that are defined in a C library. Analogous to above, both environments exchange data and commands via the API. When a C function terminates, it returns to Lua the number of its pushed arguments on the stack.

Figure 4.2: Behavior of the Lua/C-API



4.2.2 Sensor Software

The sensor software reads in the electronic signals of a sensor, quantizes the data, and calibrates it. Depending on the type of sensor, it reads in its electrical signals via a specific medium. These can be GPIO pins, which the sensor software declares as input pins, or an appropriately configured I2C bus. To quantize an electrical signal, the sensor software uses an Analog-to-Digital Converter (ADC), which maps the measured voltage to a defined number of bits. In addition to the width, i.e. the number of bits, the sensor software also specifies how much the measured voltage should be amplified. An example is a measurement of the gas concentration: The sensor software reads the electronic signal, maps the measured value to 12 bits and does not amplify it (i.e. with a factor of 0dB).

This resulting bit sequence represents a specific decimal value. The sensor software can convert this decimal value by calibration into the corresponding value of the measured physical quantity, for example into a temperature value. See figure 4.3 for illustration.

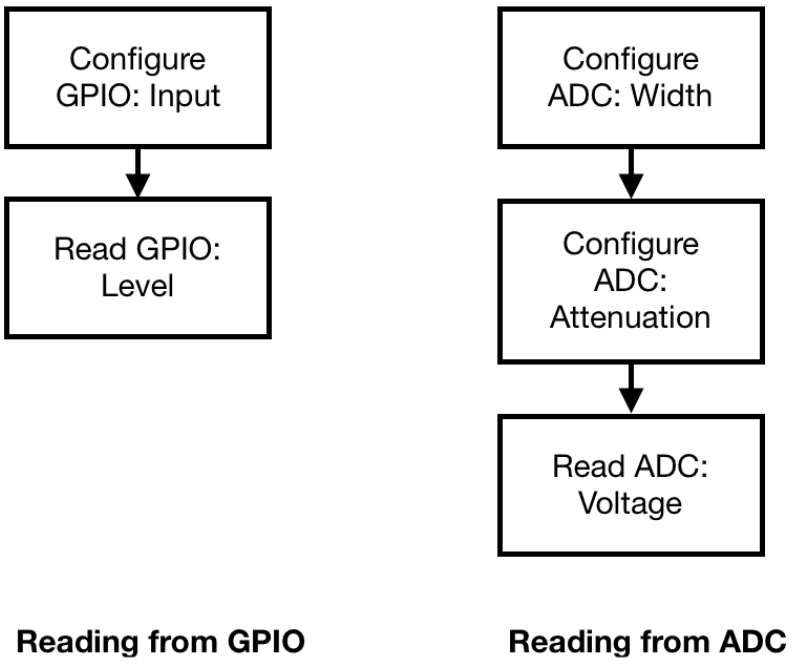


Figure 4.3: Sensor Software

4.3 Pull Environment

When Lua calls a function in the pull environment, the function reads from its assigned sensor or it outputs a value to its assigned actuator and then returns the recording back to Lua.

4.4 Push Environment

A push environment is a data management system and it contains its own information producers and information consumers. An information producer reads measurements from

a sensor and stores its recordings in the push environment. An information consumer takes data from the push environment and hands it to an external component. The intermediary between data producers and data consumers is a dedicated data structure, such as a list, a stack, or a map. Our networked dynamic execution environment comprises multiple stacks. On these stacks we hold environmental information such as about the temperature or about the air pressure. The measurements are periodically generated by dedicated tasks and they are periodically consumed by dedicated tasks. For example, a producing task A may hourly push a temperature recording onto the stack and a consuming task B may pop a temperature recording every two hours. After it has consumed an element from a stack, a task either hands this element to an actuator or it sends the element to an external client. See figure 4.5 for illustration.

A push environment couples reactive systems. A reactive system is an infinite program which reacts to inputs from its environment and/or generates outputs to its environment. Two reactive systems are coupled via a common resource. See figure 4.4 for illustration.

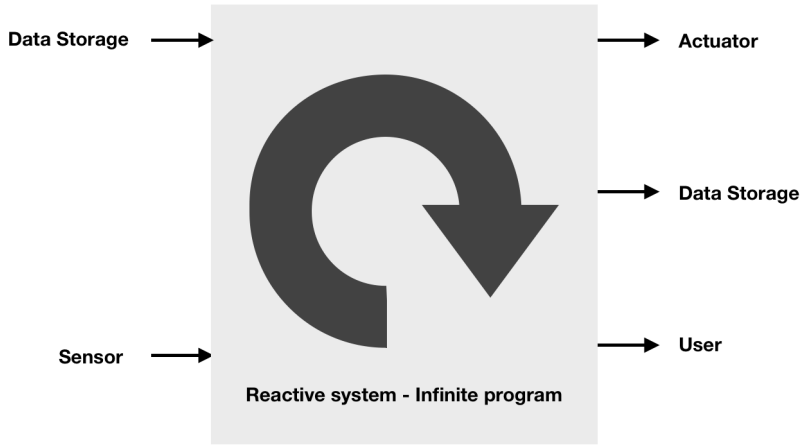


Figure 4.4: Reactive system

We will gain deep understanding of our push environment by learning about its reactive systems and by learning about how they are connected.

4.4.1 Tasks

Our reactive systems run as tasks. A task is an infinite program which operates concurrently to other tasks. It receives data from a sensor or a stack and it transfers data to an actuator, an external client, or a stack. Two tasks exchange information via the stack. Our push environment contains the three types *sensor tasks*, *actuator tasks*, and *client tasks*. See figure 4.4 for illustration.

- A sensor task produces sensor data. It periodically reads from a defined sensor and pushes the reading onto a defined stack. Examples for sensor tasks may store temperature recordings or they may store air pressure recordings on the stack.
- An actuator task consumes data for an actuator. It periodically pops an element from a defined stack and operates a defined actuator based on the element. Examples for actuator tasks may display information on an OLED or they may flash an LED.
- A remote client task consumes data for an external client. It periodically pops an element from a defined stack and sends it to a defined client in the network. Examples for client tasks may transfer information to a mobile phone or laptop.

See figure 4.5 for illustration.

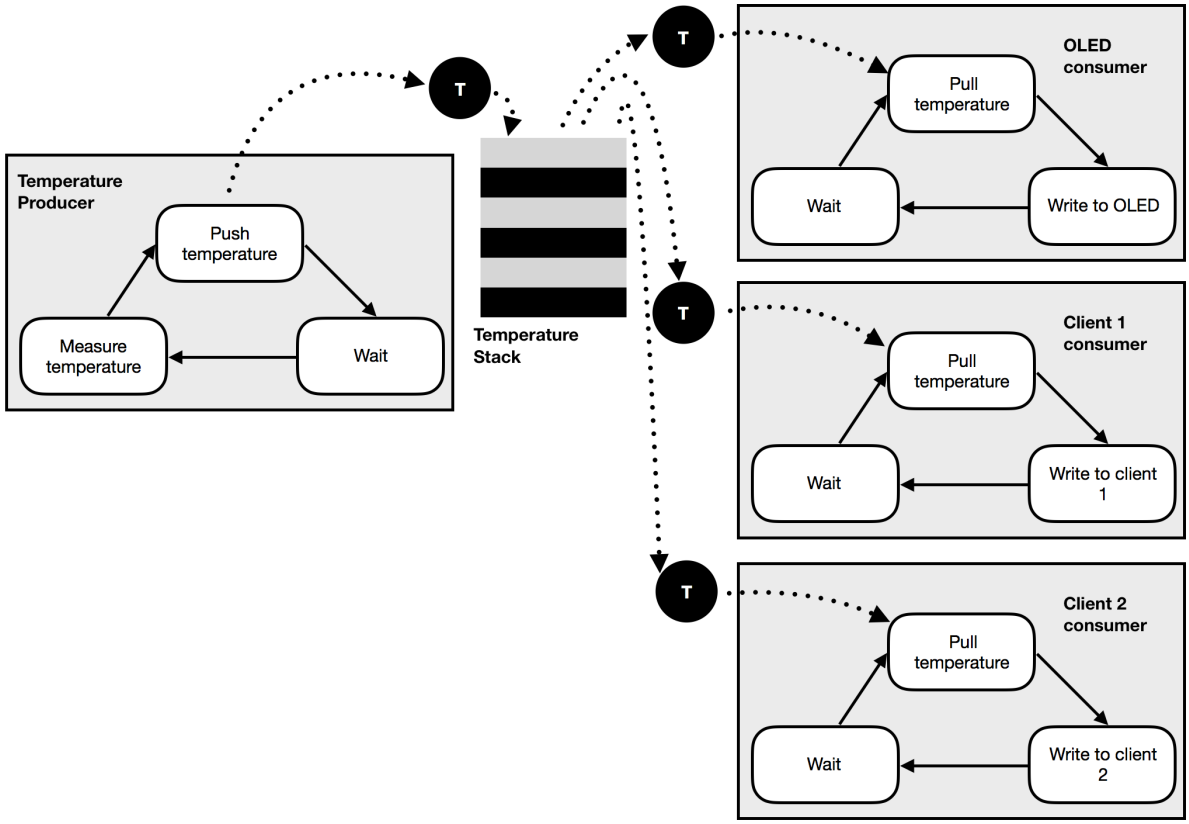


Figure 4.5: Push environment

On the basis of this classification we define a task as a producer or as a consumer. A producer is a sensor task that generates data. A consumer is an actuator task or a client task that uses the data. See figure 4.6 for illustration.

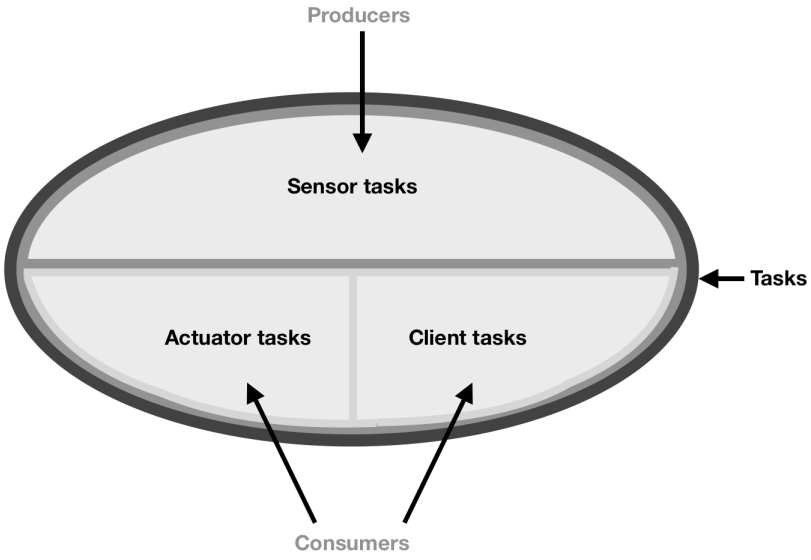


Figure 4.6: Tasks

The push environment assigns a hardware function to each task. Using the hardware function a sensor task can read from a sensor, an actuator task can forward an object to an actuator, and a client task can send information to a client. A task stores a pointer to its hardware function which it can call at any time. For example, a task may produce temperature data. To do this, it accesses a pointer to a sensor function which reads from the temperature sensor, appends protocol information, and returns it to the producer task.

Each task has a frequency and a duration. Since the task is a loop, it has a fixed number of iterations and it is repeated at fixed intervals. The frequency, a positive integer, indicates the latter in milliseconds. The duration defines the former, whereby a negative duration represents an infinite loop. For example, if we set the frequency to 500 and the duration to 10, the task will repeat 10 times and wait 500 ms between each iteration. If we set the frequency to 2000 and the duration to -1, the task will run forever and wait 2 s between each iteration. See table 4.7 for illustration.

Duration	Negative (-)	Zero (0)	Positive (+)
Meaning	Infinite loop	Terminating loop	Loop with x iterations

Table 4.7: Duration parameter

The configuration of a task regulates its control flow. When the push environment initializes a task, it assigns the parameters *frequency*, *duration*, *stack*, and *hardware function* to the task. The task runs in the specified frequency and over the specified duration. A producer task receives an element from its hardware function and pushes it onto its declared stack. A consumer task pops an object from its declared stack and hands it to its hardware function. See figures 4.5, 4.9, and 4.8 for illustration.

A task is scheduled by both itself and the operating system. For this, a task always assumes one of the four states *ready*, *running*, *waiting*, and *suspended*. When a task is *ready*, the operating system grants it to run. When a running task terminates, it becomes *suspended*. A task puts itself into *waiting* state in order to pause until an external event occurs. For example, a *running* task can continue performance after a period of 3 seconds. To do this, it sets an external timer to 3 seconds, changes to the *waiting* state and is interrupted by the timer after the time span has elapsed. See figure 4.7 for illustration.

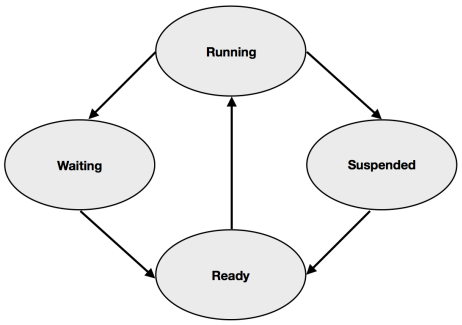


Figure 4.7: Task scheduling

We design a task as a loop according to an algorithm:

1. Input: A producer task calls a sensor function and is returned a protocol-compliant reading. A consumer task pops an element from its stack.

2. Output: A producer task pushes its reading onto its stack. A consumer task calls its consumer function and passes it its data element.
3. Wait: The task is paused as long as the frequency parameter specifies. For example, if the frequency parameter is set to one hour, the task will wait one hour.
4. Refresh: The task checks whether a client has increased the frequency and/or duration parameter and updates them if necessary.
5. Iteration: Based on its remaining number of iterations, the task terminates or restarts the task step 1.

See figures 4.8 and 4.9 for illustration.

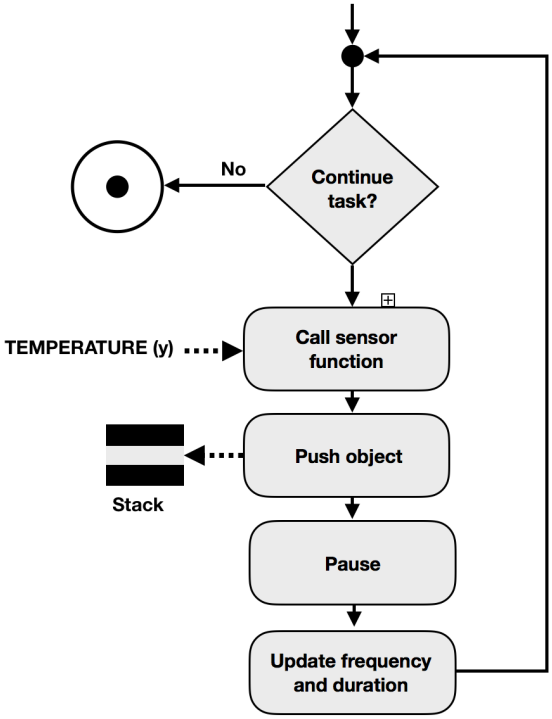


Figure 4.8: Producer Task

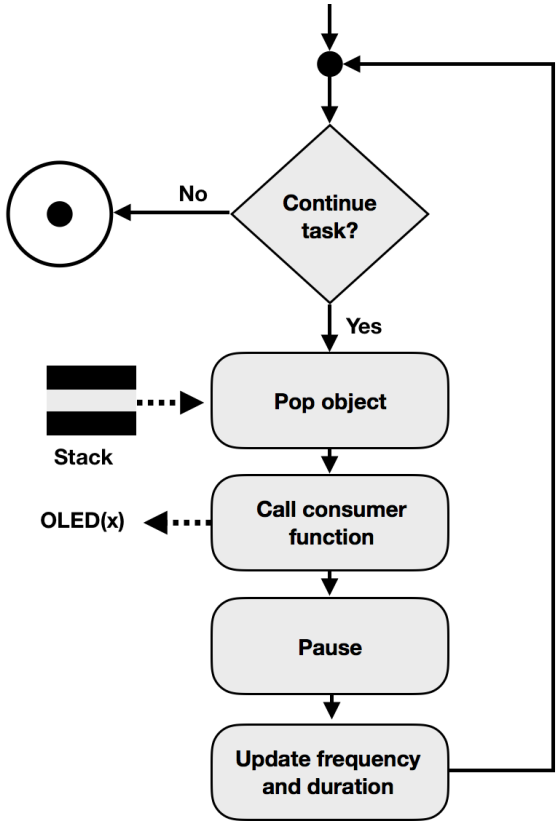


Figure 4.9: Consumer Task

4.4.2 Stacks

Our stacks support LIFO access. LIFO is an anagram for Last In First Out. This means that a sensor task always pushes and an actuator or client task always pops an element to and from the top of the stack. For example, if the sensor task pushes rising temperature values, the recordings on the stack will thus increase from bottom to top. See figure 4.5 for illustration.

Every stack serves one measurand. This means that a stack is to contain exactly one measurand type and two measurands of the same type are to be stored on the same stack. Accordingly, our push environment may dedicate one stack to temperature, it may dedicate a second stack to air pressure, and it may assign a third stack to humidity. If multiple sensors measure the same parameter, the readings are pushed onto the same

stack. A client or actuator consumes the parameter from exactly the same stack. See figure 4.5 for illustration.

4.4.3 Clients

The push environment is regulated by clients. A client is a hardware component such as a laptop or a smart phone. It is connected to the microcontroller via a socket, and the push environment identifies every client via a unique id. A client initiates, manipulates, or terminates a production process or a consumption process in the push environment by sending Lua scripts. For example, a client may request the push environment to periodically record the air pressure, then increase the frequency of the recording, and eventually stop the task. See figure 4.10 for illustration.

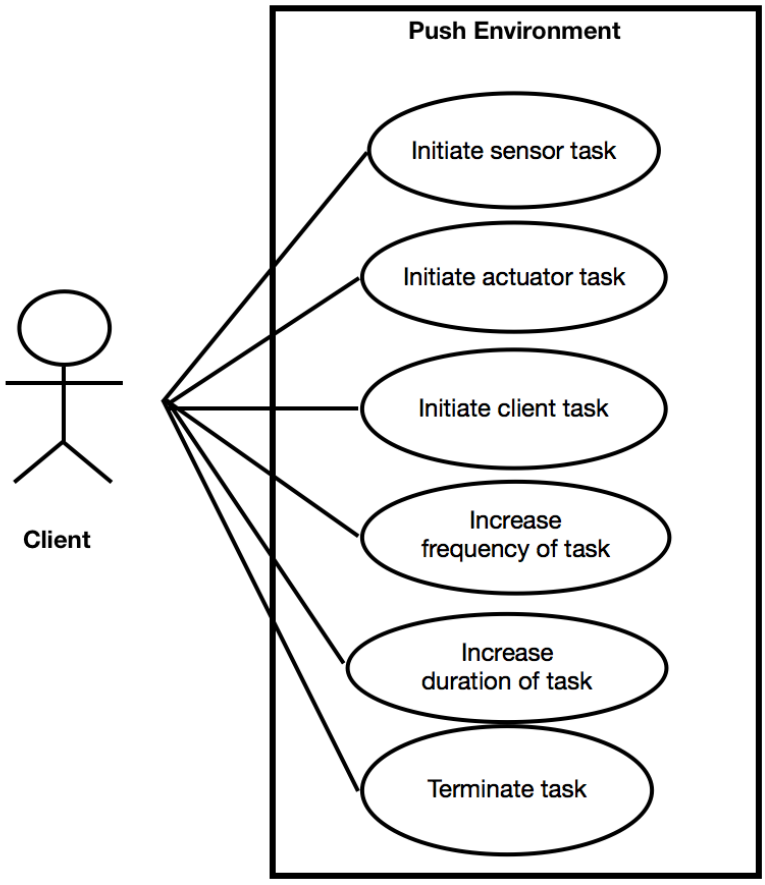


Figure 4.10: Client use cases

4.4.4 Relationship of tasks, stacks, clients

The push environment is designed according to a well-defined set of rules:

1. Each type of measurand is recorded by a dedicated task and stored on a dedicated stack. Therefore, if a client wants to generate a data collection task for a specific measurand, there are two possibilities: Either a task already exists for the given measurand or no such task exists. In the latter case, the task and stack are generated anew. In the first case, however, these already exist. Therefore, no further task is created, but rather the client registers with the existing task. By registering for a task, a client requests the push environment to (further) perform the task for it.

2. If measurements of a specific stack are output to a specific actuator, this activity is assigned to exactly one task. The periodic display of the air pressure on an OLED display is thus performed by exactly one task, for which several clients can register. The periodic indication of the air pressure by an LED display is assigned to another task, for which again several clients can register.

3. If measurements of a specific stack are sent to a specific client, this activity is assigned to exactly one task. The periodic transfer of a temperature measurement to a client 1 is thus performed by exactly one task A, while the periodic transfer of a temperature measurement to a client 2 is performed by exactly one other task B, and the periodic transfer of a pressure measurement to the client 2 is again performed by exactly one other task C.

4. When initiating or manipulating a task, a client declares its minimum runtime: The duration of the task then equals the maximum of all durations externally requested for the task. For example, if three clients demand a task to endure for seven, three, and five iterations, the runtime of the task equals seven. If a client now requires the task to run infinitely, the new duration of the task is infinity. The constraint optimizes the performance of our push environment.

5. When initiating or manipulating a task, a client declares its minimum frequency: The frequency of the task then equals the maximum of all frequencies externally requested for the task. For example, if three clients demand the task to repeat itself every minute, every hour, and every two hours, the frequency of the task equals one reading per minute. If a client now requests an hourly recording, the task remains to restart every minute. The constraint optimizes the performance of our push environment.

6. A task is always associated with at least one client. If an external logs off from a

task and the task then serves no client, the push environment removes the task.

7. There are no stacks without both producing and consuming tasks. Allowed stacks have exactly one producer and at the same time none, one or many consumers or they have no producer and at least one consumer. Stacks with neither a producer nor a consumer are removed from the push environment to optimize memory utilization.

See table 4.8 for illustration.

	No producer	One producer	Multiple producers
No consumer	Remove stack.	✓	Only one producer allowed.
One consumer	✓	✓	Only one producer allowed.
Multiple consumers	✓	✓	Only one producer allowed.

Table 4.8: Producers, consumers, and a stack

See figure 4.11 for illustration.

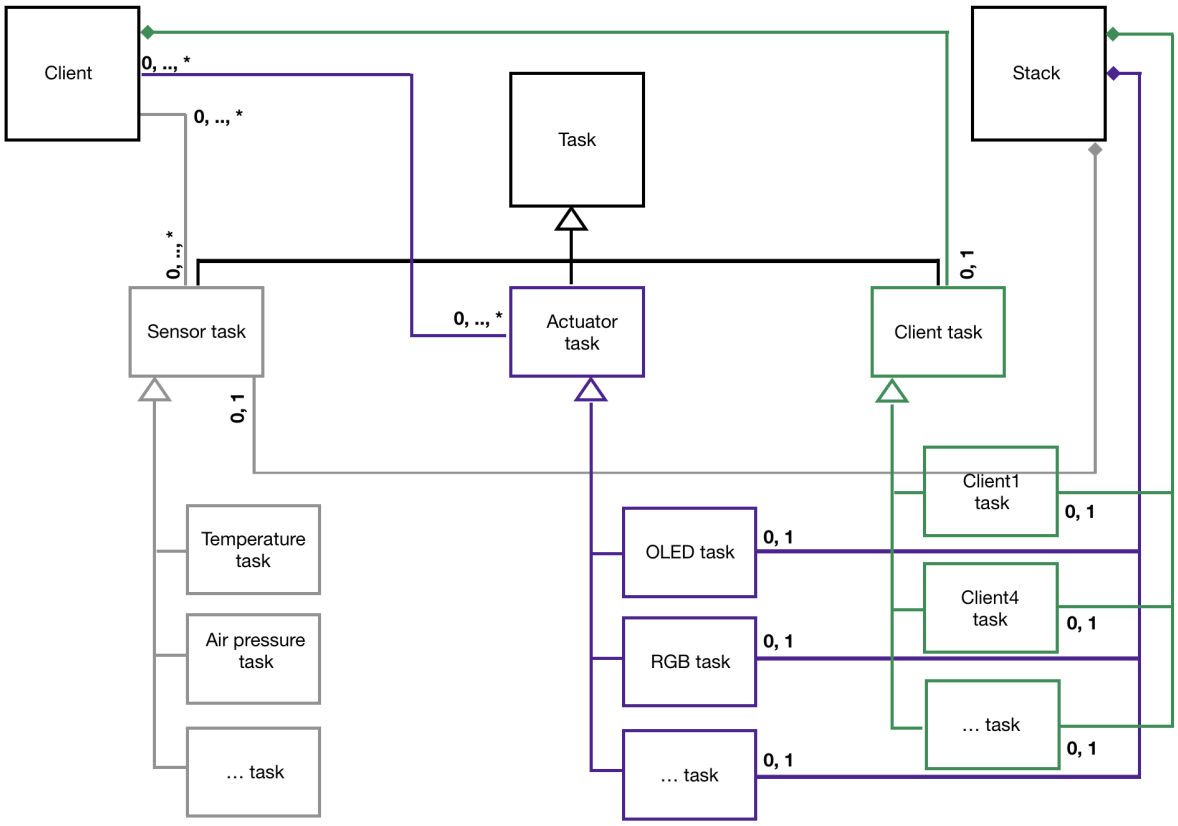


Figure 4.11: Class diagram for the push environment

4.4.5 Management of the task model

We have defined the relationship between clients, stacks, and tasks, distinguishing between producer, actuator consumer, and client consumer tasks. A relational database stores this data. The data model maps each task to its frequency, duration, and stack. Producer tasks are additionally assigned their measurand and their clients. Actuator consumers are instead assigned to their actuator and their clients. A client consumer is assigned to one registered client. All tasks are identified by their taskId, producers also by their stack and by their measurand. A (stack, actuator) pair defines a consumer task for actuators, while a (stack, client) pair identifies a consumer task for clients. See figures 4.12 and 4.9 for illustration.

The database is robust against anomalies. If a database contains redundant information,

this information takes up unnecessarily much storage space and its inconsistent modification can lead to anomalous processing. To prevent this, our database is in the fifth normal form. See figure 4.12 and table 4.9 for illustration.

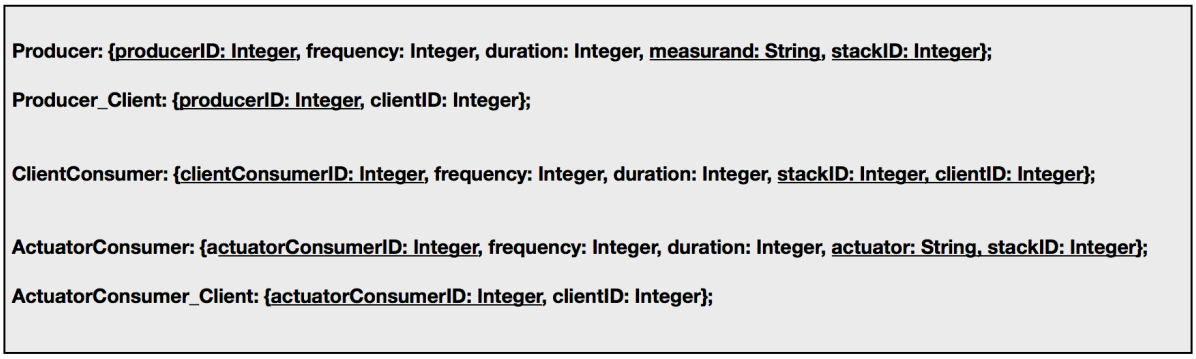


Figure 4.12: Database schema

PRODUCER				
producerID	**frequency**	**duration**	**measurand**	**stackID**
0	1000	80	“TEMPERATURE“	0
1	200	-1	“AIRPRESSURE“	1

PRODUCER_CLIENT	
producerID	**clientID**
0	8
0	3
1	8

ClientConsumer				
clientConsumerID	frequency	duration	stackID	clientID
2	2000	-1	0	8
4	400	-1	0	7
5	600000	77	1	8
8	90000	-1	0	3

ActuatorConsumer				
actuatorConsumerID	**frequency**	**duration**	**actuator**	**stackID**
3	200	-1	“OLED“	0
6	400000000	-1	“OLED“	1
7	800	21	“LED“	0

ActuatorConsumer_Client	
actuatorConsumerID	**clientID**
3	3
6	3
6	8
7	3

Table 4.9: Data model

The database schema is managed by a task manager. It serves as the storage of the information contained in our model. If a process in the push environment produces new entities and/or relations, these must be registered with the task manager. For a defined key entity, an external component can thus query all associated values. Example: A client initiates the creation of a new task in the push environment. As a result, the task manager registers the client and task as new entities as well as their relationship. In this way, the task manager can then identify all tasks of this client and can subsequently make this information available for a requesting application.

4.4.6 Status Messages

The push environment informs users about the results of its internal procedures. For this purpose, it sends status reports. We distinguish between error messages and confirmation messages.

Typical errors in a client's script can also affect the push environment. For example, the parameters of a method may be incomplete, of an unsupported type, or inherently incorrect. In addition, a client may wish to consume data from a stack whose ID does not appear in the Task Manager's data model.

If such errors do not occur, the push environment informs its client of the correct execution. This means that the instruction specified in its script has been understood and the appropriate cooperation of a task with a stack has been initiated. Thus the new client of a consuming task receives the message OK. The client of a producing task instead receives the message STACK, which contains the ID of the associated stack.

4.4.7 Control Flow

An algorithm defines the control flow in the push environment.

1. Initialization: The Lua interpreter pushes function arguments on the Lua/C stack and calls a C function.
2. Arguments: The C function pops all arguments from the Lua/C stack and verifies them. If they are invalid, continue with step 6, otherwise with step 3.
3. Stack: The system checks if a matching stack exists.
   (a) Stack not available: Consumption from an absent stack is a fault. Continue with step 6. For production the stack is created anew. Proceed to step 4.

   (b) Stack available: If the stack already exists, the push environment checks whether the stack is already assigned to the defined task.

      i. Task not available: Continue with step 4.

      ii. Task available: The push environment sets the frequency of the task to the maximum of the current and the demanded frequency. It sets the duration to the maximum of the current and requested duration. Continue with step 5.

4. Task: A new task is created. This task meets the functional requirements of the client and runs with the desired minimum frequency and duration. The new task and the relationship to its stack are registered with the task manager.

5. Registry: The assignment between the task and the client is logged on to the Task Manager. The Task Manager takes into account that a client can only register once for a task and adjusts its data model accordingly.

6. Status: The push environment pushes a status message to the Lua/C stack. This is an OK for a consuming task, a STACK for a producing task, and an ERROR for a failure.

7. Return: The C function indicates its termination to the Lua interpreter.

See figure 4.13 for illustration

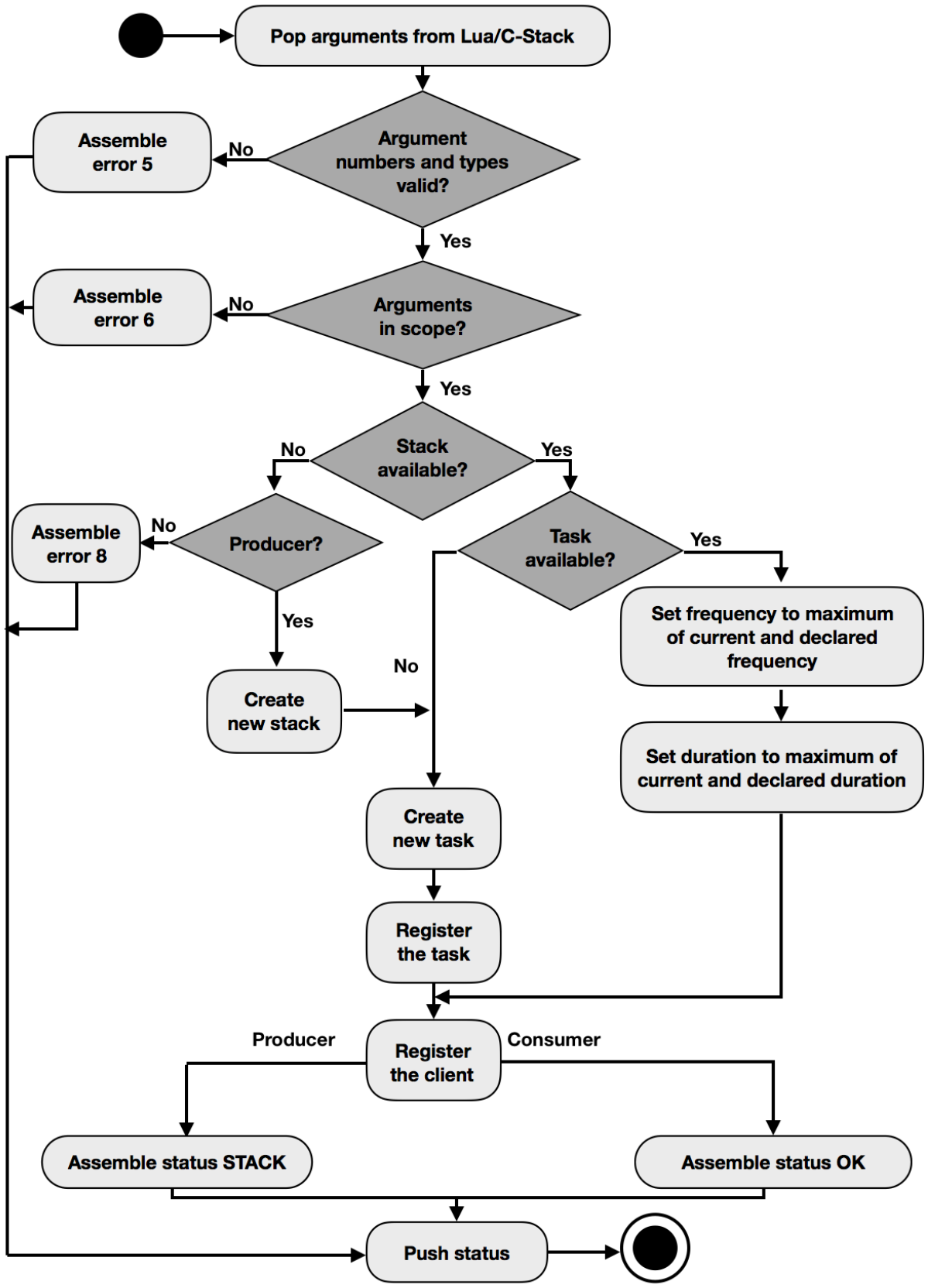


Figure 4.13: Control Flow

4.4.8 Functional Composition

Consumer functions can be composed from producer functions: Because every producer function returns a STACK object and because every consumer function requires a stack object, such a production can be nested into a consumption. For example, the function *oled(2, 300, 20, (temperature(1, 300, 20))* would cause a periodic measurement of the ambient temperature and its subsequent display on an OLED. The external client would be informed accordingly with an OK message if the measurement was carried out correctly. See figure 4.14 for illustration.

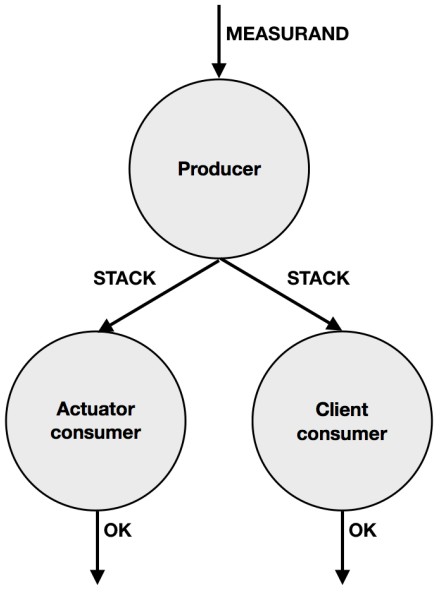


Figure 4.14: Functional composition

Evaluation

A software test is to prevent accidents with potential financial losses, physical injury and fatal consequences: Every software contains both errors and trade-offs between non-functional properties. Thus, when evaluating such a system, there will always be test cases that the system can fulfill, test cases that show strengths of a system, and test cases for which the system has not yet been adequately developed. A failed test or a recognized weakness serves as a central starting point for understanding and reworking an application or even for in-depth research. In this way, software tests help to continuously improve the quality of a program. An exemplary system whose software must minimize its number of defects and optimize performance is the airbag of an automobile. One test case here would check whether the software identifies dangerous deceleration, while another test case questions whether the air bag opens fast enough.

5.1 Hardware Design

We construct a hardware platform suitable for the evaluation of the networked dynamic execution environment. An ESP32 microcontroller is connected to a display and a barometer. In particular we use an ESP-WROOM-32, an OLED I2C display and the BMP180 from SunFounder. Each output pin of the display and the barometer is wired to a specific pin of the microcontroller. See table 5.1 for illustration.

BMP180 & OLED: Pin	ESP32: Pin
VCC	3.3 V
GND	GND
SDA	GPIO18
SCL	GPIO19

Table 5.1: Pin map

The resulting hardware platform becomes a runtime environment when we load the corresponding application code onto the hardware. To do this, we connect the computer on which the software is developed to the ESP32 microcontroller via a USB to micro USB cable. Using the IoT Development Framework for ESP32, the networked dynamic execution environment can be compiled and then flashed to the microcontroller.

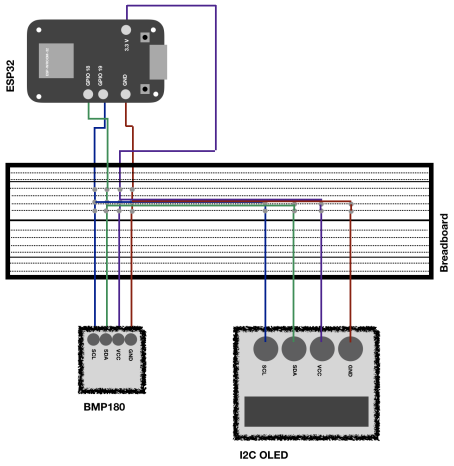


Figure 5.1: Hardware architecture

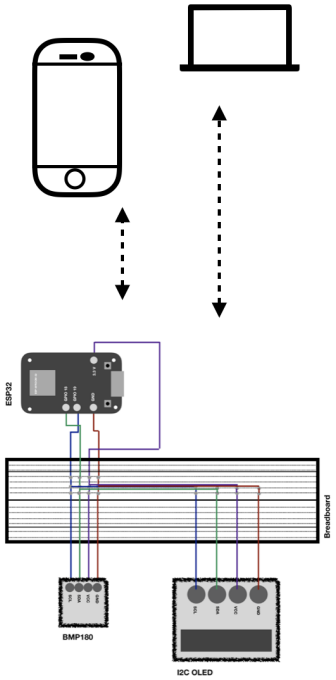


Figure 5.2: Architecture for evaluation

Because the system is designed for configuration by external actors, it takes several clients to perform a test. We enable a laptop and a smartphone to connect to the networked dynamic execution environment via the Transmission Control Protocol. By executing Javascript applications, clients can bind to the ESP32 via the IP address, exchange Lua scripts with it and finally disconnect again. See figures 5.1 and 5.2 for illustration.

5.2 Functional Evaluation

A functional software test verifies and validates the system functionality, i.e. the conformity of the application to its requirements. A verification tests a single component, the integration of several components, and the entire software system against their respective specification in the conceptual and detailed design. A validation then questions whether the system also meets the actual user needs.

5.2.1 Test Cases

The functional test comprises the following sequence of steps:

1. Setup: We load our application onto the ESP32 and we connect it to 3.3V.
2. Connection: We connect both a laptop and a smart phone to the ESP32 access point. For this, we select the SSID “ESP32“ and enter the demanded password.
3. Registration: The clients log in to the networked dynamic execution environment. The ESP32 has the IP address 192.168.1.1 and accepts clients on port 3000.
4. Communication: We document and perform tests on permutations of Lua commands. See test cases below.
5. Closure: The client logs off from the networked dynamic execution environment and we disconnect it from ESP32.

See table 5.2 for the test case specification.

ID	Objective	Command	Expected Result	Received Result	Status
1	Setup	make flash monitor	Availability of ESP32 as access point	ESP32 is available as access point.	Passed
2	Connection		Wi-Fi connection to ESP32.	Both clients are connected to ESP32.	Passed
3	Registration	const client = net.createConnection(3000,"192.168.1 .1", () => { console.log(,connected!’); });	connected!	connected!	Passed
4	Closure	client.on('end', () => { console.log(,disconnected!'); });	disconnected!	disconnected!	Passed

Table 5.2: Setup: Test cases

With regards to test step 4, the communication of our push environment is structured according to its use cases. We assemble permutations of Lua commands, we have them interpreted by our runtime, and we verify loggings and status messages. The commands are either defined in native Lua, in our pull environment, or in our push environment. We

test native Lua functions both with and without an input as well as with and without an output. We pull from sensors and pass values to actuators. We make sensors periodically push recordings and we have these recordings periodically output to actuators and clients. The runtime interprets the documented operations both in isolation and in composition. See figures 5.3 and 5.4 for illustration.

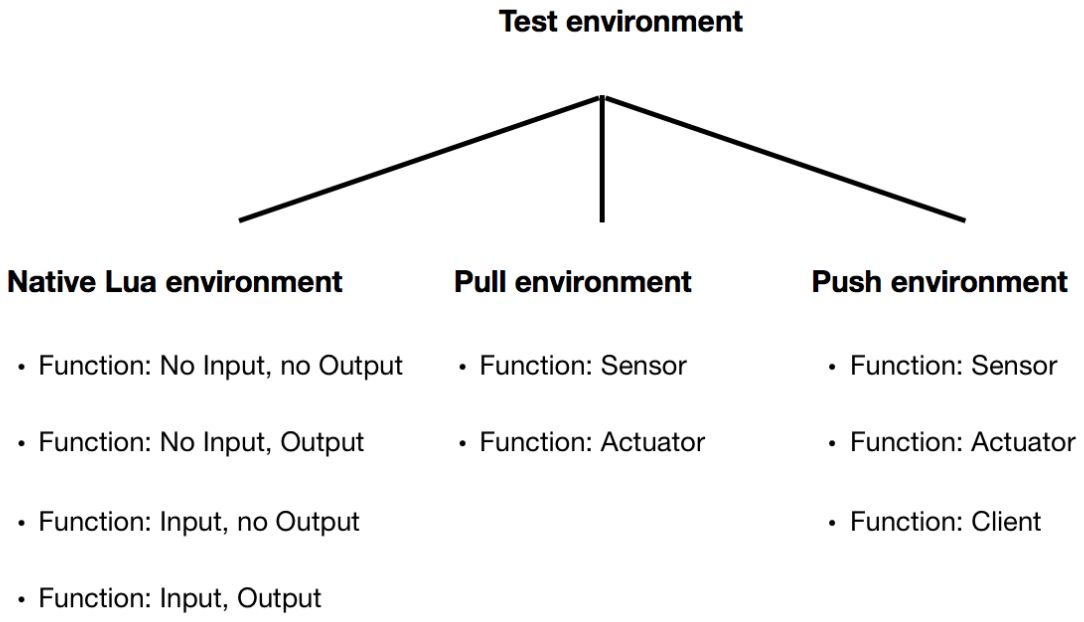


Figure 5.3: Classification of test cases

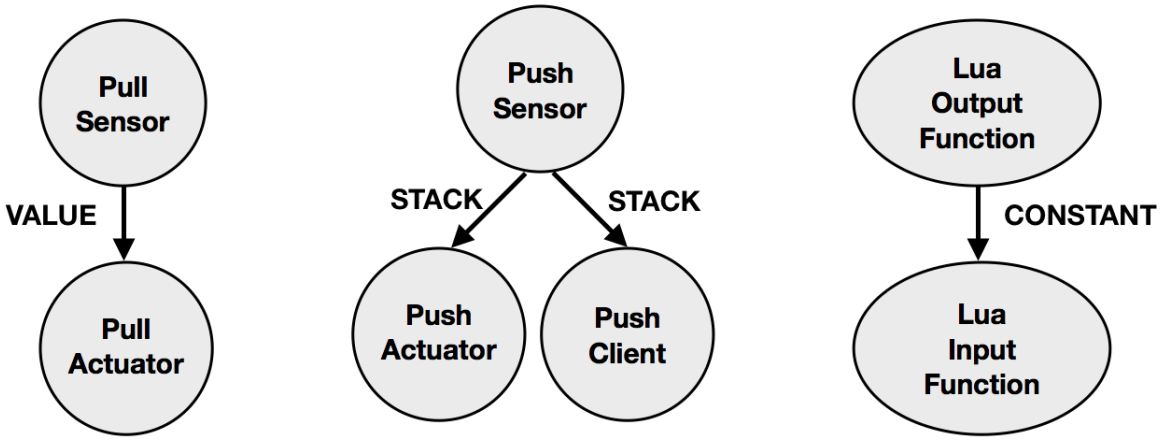


Figure 5.4: Functional composition

To efficiently evaluate each test environment, we verify the system against equivalence classes. If we view our networked dynamic execution environment as a black box, a particular category of inputs must always create a clearly defined category of outputs. We group such corresponding categories into equivalence classes, select an element from each input class, have it converted into an output by our runtime, and verify the output's conformity with its respective equivalence class. For example, if a client sends a request with a positive identifier, we expect the returned bit string to contain an identifier field with a leading 0; if a client sends a request with a negative identifier, we expect the returned bit string to contain an identifier field with a leading 1. If the request identifier is zero, we require it to be decoded as the bitstring 0000 0000. Therefore, we select identifiers 3, 0, and -4 for the equivalence test.

5.2.1.1 Native Lua Environment

The runtime interprets functions native to Lua. Because we assume the correct behavior of the Lua Virtual Machine, we test our use of the Lua/C-API. Functions with and without both input and output are passed to the API and corresponding results are expected:

1. No input, no output: The runtime interprets an empty string.
2. No input, output: The runtime interprets a constant.
3. Input, no output: The runtime interprets a function that needs input and that produces no output.

4. Input, output: The runtime interprets a function that needs input and that produces output.
5. Composition: The runtime interprets a function that needs an input and produces an output. The output serves as the input of a wrapper function that produces an output.

See table 5.3 for the test case specification.

ID	Objective	Command	Expected Result	Received Result	Status
5	No input, no output	,'	Missing return	Missing return	Passed
6	No input, output	return 5	5	5	Passed
7	Input, no output	print ('Hello World')	ESP32: Hello World Missing return	ESP32: Hello World Missing return	Passed
8	Input, output	return 4+5	9	9	Passed
9	Composition	return 1+(9-3)	7	7	Passed

Table 5.3: Native Lua environment: Test cases

5.2.1.2 Pull Environment

The runtime interprets functions native to the pull environment. All functions have request identifiers from distinct domains. For a sensor, the Lua commands address different measurands. An actuator is made display both errors and sensor readings.

1. BMP180, pressure, positive ID: The runtime interprets an air pressure measurement. The request identifier is positive.
2. BMP180, pressure, ID zero: The runtime interprets an air pressure measurement. The request identifier is zero.
3. BMP180, temperature, negative ID: The runtime interprets a temperature measurement. The request identifier is negative.
4. OLED, value, positive ID: The runtime interprets the output of temperature object. The request identifier is positive.
5. OLED, value, negative ID: The runtime interprets the output of a temperature object. The request identifier is negative.

6. OLED, error, ID zero: The runtime interprets the output of error number 4. The request identifier is zero.
7. Composition: The runtime interprets the output of a temperature object. The temperature object is produced after the runtime has interpreted a temperature measurement.

See table 5.4 for the test case specification.

ID	Objective	Command	Expected result	Received result	Status
10	Pressure, positive ID	return bmp180.pressure(1)	Value	Value	Passed
11	Pressure, ID zero	return bmp180.pressure(0)	Value	Value	Passed
12	Temperature, negative ID	return bmp180.temperature(-5)	Value	Value	Passed

13	Value, positive ID	local answer = {};
		answer['status'] = 1;
		answer['requestid'] = 1;
		answer['measurand'] = ,TEMPERATURE';
		answer['device'] = 'BMP180'; answer['unit'] = 'DEGC'; answer['timestamp'] = 12345687;
		answer['type'] = 'FLOAT';
		answer['value'] = 1.2;
		return lcd.write(2, answer)
			OK	OK	Passed

14	Value, negative ID	local answer = {};
		answer['status'] = 1;
		answer['requestid'] = 1;
		answer['measurand'] = ,TEMPERATURE';
		answer['device'] = 'BMP180'; answer['unit'] = 'DEGC'; answer['timestamp'] = 12345687;
		answer['type'] = 'FLOAT';
		answer['value'] = 1.2;
		return lcd.write(-1, answer)
			OK	OK	Passed

15	Error, ID zero	local answer = {};
		answer['status'] = 4;
		answer['requestid'] = 1;
		answer['measurand'] = ,TEMPERATURE';
		answer['device'] = 'BMP180'; answer['unit'] = '';
		answer['timestamp'] = 12345687; answer['type'] = '';
		answer['value'] = 0;
		return lcd.write(0, answer)
			OK	OK	Passed

16	Composition	return lcd.write(1, bmp180.temperature(-9))	OK	OK	Passed

Table 5.4: Pull environment: Test cases

5.2.1.3 Push Environment

The runtime interprets functions native to the push environment. All functions select both request identifiers and task durations from distinct domains. In addition, sensor functions address different measurand types.

1. Temperature, positive ID, positive duration: The runtime initiates a periodic temperature measurement for x iterations. The request identifier is positive.
2. Temperature, ID zero, duration zero: The runtime initiates a periodic temperature measurement which is about to terminate. The request identifier is zero.
3. Pressure, negative ID, negative duration: The runtime initiates a periodic pressure measurement for an infinite time. The request identifier is negative.
4. OLED, positive ID, negative duration: The runtime initiates periodic output to an OLED for an infinite time. The request identifier is positive.
5. OLED, ID zero, positive duration: The runtime initiates periodic output to an OLED for y iterations. The request identifier is zero.
6. OLED, negative ID, duration zero: The runtime initiates periodic output to an OLED which is about to terminate. The request identifier is negative.
7. Client, positive ID, duration zero: The runtime initiates periodic transfer to a client and the transfer is about to terminate. The request identifier is positive.
8. Client, negative ID, positive duration: The runtime initiates periodic transfer to a client for z iterations. The request identifier is negative.
9. Client, ID zero, negative duration: The runtime initiates periodic transfer to a client for an infinite time. The request identifier is zero.

See table 5.5 for illustration.

ID	Objective	Command	Expected result	Received result	Status
17	Temperature, positive ID, positive duration	return dle.temperature(3, 400, 34)	Stack	Stack	Passed
18	Temperature, ID zero, duration zero	return dle.temperature(0, 1500, 0)	Stack	Stack	Passed
19	Pressure, negative ID, negative duration	return dle.pressure(-1, 900, -1)	Stack	Stack	Passed
20	OLED, positive ID, negative duration	return dle.lcd(241, 300, -2)	OK	OK	Passed
21	OLED, ID zero, positive duration	return dle.lcd(0, 100, 9)	OK	OK	Passed
22	OLED, negative ID, duration zero	return dle.lcd(-2, 2000, 0)	OK	OK	Passed
23	Client, positive ID, duration zero	return dle.net(8, 600000, 0)	OK	OK	Passed
24	Client, negative ID, positive duration	return dle.net(-8, 9000000, 9000)	OK	OK	Passed
25	Client, ID zero, negative duration	return dle.net(0, 800000, -1)	OK	OK	Passed

Table 5.5: Simple push environment: Test cases

We now assume the functional correctness of simple producers and consumers. On this basis, we verify the correlations within a functional sequence. Therefore, our runtime interprets the successive initialization, manipulation, and removal of a task.

1. Initiation: The runtime system initiates a temperature measurement at frequency f1 over the finite duration d1. Both f1 and d1 are positive integers.
2. Frequency decrease, duration decrease: The runtime system initiates a temperature measurement at frequency f2 over the finite duration d2. Both f2 and d2 are positive integers and smaller than f1 and d1.
3. No change of frequency, duration increase: The runtime system initiates a

temperature measurement at frequency f3 over the finite duration d3. Both f3 and d3 are positive integers. While f3 equals f2, d3 is greater than f1.

4. Frequency increase, no change of duration: The runtime system initiates a temperature measurement at frequency f4 over the finite duration d4. Both f3 and d3 are positive integers. While f4 is greater than f2, d4 equals d3.
5. Finite to infinite duration: The runtime system initiates a temperature measurement at frequency f5 over the infinite duration d5. While f5 is a positive integer, d5 is a negative integer.
6. Infinite to finite duration: The runtime system initiates a temperature measurement at frequency f6 over the finite duration d6. Both f6 and d6 are positive integers.
7. Infinite to infinite duration: The runtime system initiates a temperature measurement at frequency f7 over the infinite duration d7. While f7 is a positive integer, d7 is a negative integer.
8. Logoff: The runtime interprets the log off from the respective temperature recording.
9. Initiation: The runtime system initiates a temperature measurement at frequency f8 and duration f9.

See table 5.6 for illustration.

ID	Objective	Command	Expected log	Received log	Expected result	Received result	Status
26	Initiation	return dle.temperature(1, 1500, 80)	Frequency: 1500 Duration: 80	Frequency: 1500 Duration: 80	Stack	Stack	Passed
27	Frequency decrease, duration decrease	return dle.temperature(2, 1000, 10)	Frequency: 1000 Duration: 80	Frequency: 1000 Duration: 80	Stack	Stack	Passed
28	No change of frequency, duration increase	return dle.temperature(3, 1000, 100)	Frequency: 1000 Duration: 100	Frequency: 1000 Duration: 100	Stack	Stack	Passed
29	Frequency increase, no change of duration	return dle.temperature(4, 2000, 100)	Frequency: 1000 Duration: 100	Frequency: 1000 Duration: 100	Stack	Stack	Passed
30	Finite to infinite duration	return dle.temperature(5, 1000, -1)	Frequency: 1000 Duration: -1	Frequency: 1000 Duration: -1	Stack	Stack	Passed
31	Infinite to finite duration	return dle.temperature(6, 1000, 20)	Frequency: 1000 Duration: -1	Frequency: 1000 Duration: -1	Stack	Stack	Passed
32	Infinite to infinite duration	return dle.temperature(7, 1000, -1)	Frequency: 1000 Duration: -1	Frequency: 1000 Duration: -1	Stack	Stack	Passed
33	Logoff	return dle.off_temperature(8)			OK	OK	Passed
34	Initiation	return dle.temperature(9, 1500, -1)	Frequency: 1500 Duration: -1	Frequency: 1500 Duration: -1	Stack	Stack	Passed

Table 5.6: Correlated push environment: Test cases

We assume that sequences of sensor functions, actuator functions, and client functions are interpreted correctly. We now verify the concurrent registration of two clients for one stack. Two clients log on to a task one after the other, then one client logs off and on again, and finally both clients log off. We distinguish between production tasks and actuator tasks, which can be subscribed to by several clients, and client tasks, to which only one client is assigned.

1. OLED, registration 1: Client 1 registers for periodic consumption of pressure measurements for an OLED.
2. OLED, registration 2: Client 2 registers for periodic consumption of pressure measurements for an OLED.
3. OLED, logoff 1: Client 1 logs off from periodic consumption of pressure measurements for the OLED.
4. OLED, registration 1: Client 1 registers for periodic consumption of pressure measurements for an OLED.
5. OLED, logoff 1: Client 1 logs off from periodic consumption of pressure measurements for an OLED.
6. OLED, logoff 2: Client 2 logs off from periodic consumption of pressure measurements for an OLED.
7. Client 1, registration: Client 1 registers for periodic reception of temperature readings.
8. Client 2, registration: Client 2 registers for periodic reception of temperature readings.
9. Client 1, logoff: Client 1 logs off from periodic reception of temperature readings.
10. Client 1, registration: Client 1 registers for periodic reception of temperature readings.
11. Client 1, logoff: Client 1 logs off from periodic reception of temperature readings.
12. Client 2, logoff: Client 2 logs off from periodic reception of temperature readings.

See tables 5.7, 5.8, 5.9, 5.10, and 5.11 for illustration.

ID	Objective	Command	Expected log	Received log	Expected result	Received result	Status

35	OLED, registration 1	local stack = {};
		stack['status'] = 2;
		stack['requestid'] = 7;
		stack['measurand'] = '';
		stack['device'] = '';
		stack['unit'] = '';
		stack['timestamp'] = 12345687;
		stack['type'] = 'INTEGER';
		stack[,value'] = 0;
		return dle.lcd(1, 2000, -1, stack)
			Number of stacks = 1
			
			Stack 0:
			Number of tasks = 2
			
			Stack 0, Task 0:
			Number of clients = 1
			
			Stack 0, Task 1:
			Number of clients = 1
				Number of stacks = 1
				
				Stack 0:
				Number of tasks = 2
				
				Stack 0, Task 0:
				Number of clients = 1
				
				Stack 0, Task 1:
				Number of clients = 1
					OK	OK	Passed

36	OLED, registration 2	return dle.lcd(2, 1000, -1, stack)	Number of stacks = 1 Stack 0: Number of tasks = 2 Stack 0, Task 0: Number of clients = 1 Stack 0, Task 1: Number of clients = 2	Number of stacks = 1 Stack 0: Number of tasks = 2 Stack 0, Task 0: Number of clients = 1 Stack 0, Task 1: Number of clients = 2	OK	OK	Passed
37	OLED, logoff 1	return dle.off_lcd(3, stack)	Number of stacks = 1 Stack 0: Number of tasks = 2 Stack 0, Task 0: Number of clients = 1 Stack 0, Task 1: Number of clients = 1	Number of stacks = 1 Stack 0: Number of tasks = 2 Stack 0, Task 0: Number of clients = 1 Stack 0, Task 1: Number of clients = 1	OK	OK	Passed

Table 5.7: Multi-client push environment: Test cases

38	OLED, registration 1	return dle.lcd(4, 2000, -1, stack)	Number of stacks = 1 Stack 0: Number of tasks = 2 Stack 0, Task 0: Number of clients = 1 Stack 0, Task 1: Number of clients = 2	Number of stacks = 1 Stack 0: Number of tasks = 2 Stack 0, Task 0: Number of clients = 1 Stack 0, Task 1: Number of clients = 2	OK	OK	Passed
39	OLED, logoff 1	return dle.off_lcd(5, stack)	Number of stacks = 1 Stack 0: Number of tasks = 2 Stack 0, Task 0: Number of clients = 1 Stack 0, Task 1: Number of clients = 1	Number of stacks = 1 Stack 0: Number of tasks = 2 Stack 0, Task 0: Number of clients = 1 Stack 0, Task 1: Number of clients = 1	OK	OK	Passed
40	Client 1, registration	return dle.net(6, 500, -1, stack)	Number of stacks = 1 Stack 0: Number of tasks = 3 Stack 0, Task 0: Number of clients = 1 Stack 0, Task 1: Number of clients = 1 Stack 0, Task 2: Number of clients = 1	Number of stacks = 1 Stack 0: Number of tasks = 3 Stack 0, Task 0: Number of clients = 1 Stack 0, Task 1: Number of clients = 1 Stack 0, Task 2: Number of clients = 1	OK	OK	Passed

Table 5.8: Multi-client push environment: Test cases

41	Client 2, registration	return dle.net(7, 500, -1, stack)	Number of stacks = 1 Stack 0: Number of tasks = 4 Stack 0, Task 0: Number of clients = 1 Stack 0, Task 1: Number of clients = 1 Stack 0, Task 2: Number of clients = 1 Stack 0, Task 3: Number of clients = 1	Number of stacks = 1 Stack 0: Number of tasks = 4 Stack 0, Task 0: Number of clients = 1 Stack 0, Task 1: Number of clients = 1 Stack 0, Task 2: Number of clients = 1 Stack 0, Task 3: Number of clients = 1	OK	OK	Passed
42	Client 1, logoff	return dle.off_net(8, stack)	Number of stacks = 1 Stack 0: Number of tasks = 3 Stack 0, Task 0: Number of clients = 1 Stack 0, Task 1: Number of clients = 1 Stack 0, Task 2: Number of clients = 1	Number of stacks = 1 Stack 0: Number of tasks = 3 Stack 0, Task 0: Number of clients = 1 Stack 0, Task 1: Number of clients = 1 Stack 0, Task 2: Number of clients = 1	OK	OK	Passed

Table 5.9: Multi-client push environment: Test cases

43	Client 1, registration	return dle.net(9, 500, -1, stack)	Number of stacks = 1 Stack 0: Number of tasks = 4 Stack 0, Task 0: Number of clients = 1 Stack 0, Task 1: Number of clients = 1 Stack 0, Task 2: Number of clients = 1 Stack 0, Task 3: Number of clients = 1	Number of stacks = 1 Stack 0: Number of tasks = 4 Stack 0, Task 0: Number of clients = 1 Stack 0, Task 1: Number of clients = 1 Stack 0, Task 2: Number of clients = 1 Stack 0, Task 3: Number of clients = 1	OK	OK	Passed
44	Client 1, logoff	return dle.off_lcd(10, stack)	Number of stacks = 1 Stack 0: Number of tasks = 3 Stack 0, Task 0: Number of clients = 1 Stack 0, Task 1: Number of clients = 1 Stack 0, Task 2: Number of clients = 1	Number of stacks = 1 Stack 0: Number of tasks = 3 Stack 0, Task 0: Number of clients = 1 Stack 0, Task 1: Number of clients = 1 Stack 0, Task 2: Number of clients = 1	OK	OK	Passed

Table 5.10: Multi-client push environment: Test cases

45	Client 2, logoff	return dle.off_lcd(11, stack)	Number of stacks = 1 Stack 0: Number of tasks = 2 Stack 0, Task 0: Number of clients = 1 Stack 0, Task 1: Number of clients = 1	Number of stacks = 1 Stack 0: Number of tasks = 2 Stack 0, Task 0: Number of clients = 1 Stack 0, Task 1: Number of clients = 1	OK	OK	Passed
46	OLED, logoff 2	return dle.off_lcd(12, stack)	Number of stacks = 1 Stack 0: Number of tasks = 1 Stack 0, Task 0: Number of clients = 1	Number of stacks = 1 Stack 0: Number of tasks = 1 Stack 0, Task 0: Number of clients = 1	OK	OK	Passed
47	Composition: Temperature, OLED	return dle.lcd(14, 3000, -1, dle.temperature(13, 2000, -1))			OK	OK	Passed
48	Composition: Pressure, Client	dle.net(16, 500, -1, dle.pressure(15, 1000, 1))			OK	OK	Passed

Table 5.11: Multi-client push environment: Test cases

5.2.2 Summary

Because all test cases pass, it is assumed that the networked dynamic execution environment is functionally correct. We have assigned all permutations of Lua commands to an equivalence class. It is shown that an instruction from each equivalence class

is interpreted as expected. We have verified the native Lua environment, the pull environment, and the push environment against their respective use cases.

5.3 Non-functional Evaluation

A non-functional software test evaluates non-functional system properties visible to the user. This comprises documentation, performance, security, and usability. The documentation includes both comments within the software and external documents. For example, a bachelor thesis serves documentation. A performance test reveals the speed of response under high load and the consumption of resources such as memory and processor power. Security tests question the confidentiality, availability, and integrity of an application. Usability is a property that is difficult to quantify. Therefore, we often evaluate user-oriented features - especially the simplicity, the learnability, and the memorability of an application - through prototypes and user feedback.

5.3.1 Usability Evaluation

We validate the comprehensibility of microcontroller functionality. When a client sends a script that the microcontroller cannot interpret, status messages allow the diagnosis and possible correction of the fault. We therefore assess whether an external is informed of failures in code compilation, measurement from a sensor, output to an actuator, functions with bad arguments, missing return statements, or requests for missing stacks.

1. Command not defined: The runtime interprets a command that is neither defined in native Lua, nor in the pull environment, nor in the push environment.
2. Module not defined: The runtime interprets a command that opens an undefined module.
3. Function not defined: The runtime interprets a function that is not defined in a given module.
4. Missing return statement: The runtime interprets a function that lacks a return statement.
5. Superfluous arguments: The runtime interprets a function with more arguments than specified.

6. Missing arguments: The runtime interprets a function with fewer arguments than specified.
7. Wrong argument type: The runtime interprets a function with a badly typed argument.
8. Argument out of scope: The runtime interprets a function with an argument out of scope.
9. Sensor failure: The runtime supports the sensor, but the sensor reading fails.
10. Actuator failure: The runtime supports the actuator, but the actuator output fails.
11. Absent stack: The runtime interprets a function that consumes from an absent stack.

See table 5.12 for illustration.

ID	Objective	Command	Expected result	Received result	Status
49	Command not defined	return §	ERROR 3	ERROR 3	Passed
50	Module not defined	return bzp176.pressure(1)	ERROR 3	ERROR 3	Passed
51	Function not defined	return bmp180.tmprtr(2)	ERROR 3	ERROR 3	Passed
52	Missing return statement	bmp180.pressure(3)	ERROR 7	ERROR 7	Passed
53	Superfluous arguments	return bmp180.temperature(4, 9)	ERROR 5	ERROR 5	Passed
54	Missing arguments	return dle.lcd(5, 400)	ERROR 5	ERROR 5	Passed
55	Wrong argument type	return dle.net(6, 1000, -1, 0)	ERROR 5	ERROR 5	Passed
56	Argument out of scope	return dle.temperature(7, -4000, -1)	ERROR 6	ERROR 6	Passed
57	Sensor failure	return bmp180.temperature(8)	ERROR 4	ERROR 4	Passed

58	Actuator failure	local value = {};
		value['status'] = 1;
		value['requestid'] = 1;
		value['measurand'] = ,TEMPERATURE';
		value['device'] = 'BMP180';
		value['unit'] = 'DEGC';
		value['timestamp'] = 12345687;
		value['type'] = 'FLOAT';
		value[,value'] = 1.2;
		
		return lcd.write(9, value)
			ERROR 4	ERROR 4	Passed

59	Absent stack	local stack = {};
		stack['status'] = 2;
		stack['requestid'] = 1;
		stack['measurand'] = '';
		stack['device'] = '';
		stack['unit'] = '';
		stack['timestamp'] = 12345687;
		stack['type'] = 'INTEGER';
		stack[,value'] = 77;
		
		return dle.lcd(10, 2000, 82, stack)
			ERROR 8	ERROR 8	Passed

Table 5.12: Usability evaluation

5.3.2 Performance Evaluation

We load test the microcontroller performance, i.e. we determine the response times of all critical transactions. A client applies both the pull mechanism to read from a sensor and to output to an actuator as well as the push mechanism to initiate or register itself for a producing task and a consuming task. It stops the time between sending and receiving a script.

1. Pull environment, sensor function: The runtime interprets a sensor function of the pull environment. The average time between sending the request and receiving the response is 40 ms.
2. Pull environment, actuator function: The runtime interprets an actuator function of the pull environment. The average time between sending the request and receiving the response is 20 ms.
3. Push environment, new stack: The runtime interprets a function that creates a new stack in the push environment. The average time between sending the request and receiving the response is 70 ms.
4. Push environment, new task: The runtime interprets a function that creates a new task for an existing stack in the push environment. The average time between sending the request and receiving the response is 30 ms.
5. Push environment, client registry: The runtime interprets a function that registers a client for an existing task. The average time between sending the request and receiving the response is 30 ms.

See table 5.13 for illustration.

ID	Objective	Command	Response times	Mean response time
60	Pull environment, sensor function	return bmp180.temperature(1)	39.325 ms 39.559 ms 38.833 ms 40.911 ms 38.840 ms	39.494 ms

61	Pull environment, actuator function	local value = {};
		value['status'] = 1;
		value['requestid'] = 1;
		value['measurand'] =
		,TEMPERATURE';
		value['device'] = 'BMP180';
		value['unit'] = 'DEGC';
		value['timestamp'] = 12345687;
		value['type'] = 'FLOAT';
		value[,value'] = 1.2;
		
		return lcd.write(2, value)
			20.472 ms
			19.105 ms
			20.363 ms
			20.503 ms
			19.180 ms
				19.925 ms

62	Push environment, new stack	local stack; stack = dle.pressure(3, 1000, -1)	61.972 ms 67.958 ms 69.138 ms 71.339 ms 69.282 ms	67.938 ms
63	Push environment, new task	return dle.lcd(4, 2000, -1, stack)	27.906 ms 27.272 ms 26.963 ms 27.700 ms 27.867 ms	27.542 ms
64	Push environment, client registry	return dle.pressure(5, 1000, -1)	29.859 ms 28.235 ms 27.451 ms 27.517 ms 28.074 ms	28.2272 ms

Table 5.13: Load test: Test cases

A human being perceives the networked dynamic execution environment to act instantly because it responds within 0.1 milliseconds. A latency of less than 10 seconds does not distract a user from his task, a latency of less than one second does not interrupt his thought flow, and a latency of less than one millisecond is not noticed.

We identify the creation of a new stack as a bottleneck in our execution environment. In both environments the functions react after about 30 ms. The deviations in the pull environment are the result of implementation details specific to the BMP180 and the LCD themselves. By initializing a new stack, the push environment more than doubles

the response time of the entire application. See figure 5.5 for illustration.

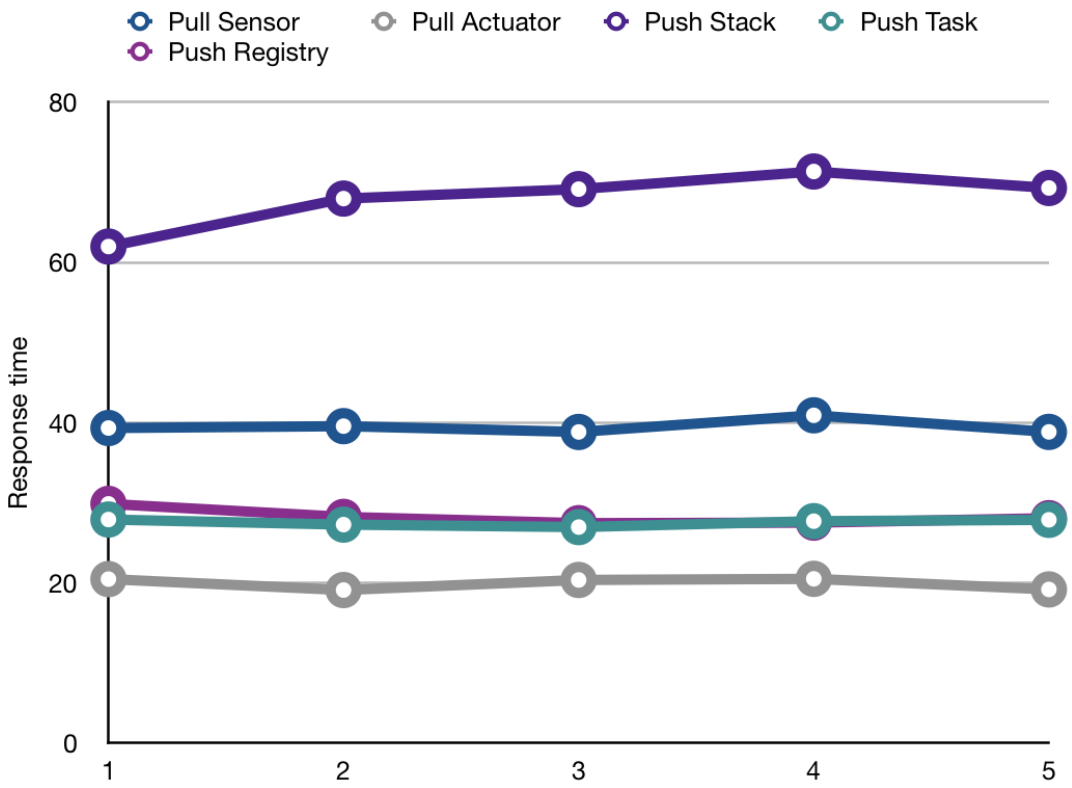


Figure 5.5: Load test: Diagram

5.3.3 Summary

The networked dynamic execution environment complies with non-functional standards. We assume that it is understandable and instantly available. Our runtime environment copes with errors and we perceive it to react immediately.

Conclusion

Via the networked dynamic execution environment a microcontroller provides dynamic services. The runtime system makes a microcontroller read from sensors and output to actuators whereby the data is either periodically recorded and consumed (push principle) or it is requested and displayed on demand (pull principle). In order to be configured by components non-native to the microcontroller, the execution environment interprets scripts in an extensible extension language. In such a programmable space, static service orchestration does not suffice. We apply services to a new usage context and we redesign their internal functionality.

Both, the security and the learnability of the networked dynamic execution environment can be improved. Security is understood as system availability, integrity, and confidentiality. The networked dynamic execution environment disregards the two latter by not preventing a potential attacker from reading and/or manipulating the exchanged functions and data. We protect our system with a password, but once a hacker has logged in, he can crash the system by overloading it. To do this, an attacker could initiate a large amount of concurrent tasks that are repeated at high frequency. Learnability is understood as the ease at which a new client learns how to consume dynamic services. For this, it needs to know what functions it can send to the networked dynamic execution environment. However, the current application does not inform externals about which sensors and actuators it has attached. Furthermore, when requesting available hardware, a user must look up the appropriate command in a manual.

Our networked dynamic execution environment shows both very good comprehensibility and performance. We define comprehensibility by an external's understanding of internal system procedures. For this, the application informs clients about the status of its execution, and in case of a failure it diagnoses the fault in the client script. We define system performance by the response time to an external request. The dynamic

service provider reacts within milliseconds and we thus perceive it to always answer instantaneously.

Our future research will focus on non-functional aspects of the networked dynamic execution environment. We will improve its learnability, rememberability, confidentiality, integrity, and memory utilization. For this, our runtime environment will beacon its hardware configuration, it will provide an API, it will be secured by the Secure Sockets Layer (SSL) protocol, and it will be equipped with a persistent data store.

List of Tables

List of Figures

Bibliography

[1] Y. Vagapov A. Maier A. Sharp. "Comparative Analysis and Practical Implementation of the ESP32 Microcontroller Module for the Internet of Things". In: *2017 Internet Technologies and Applications (ITA)* (Sept. 2017).

[2] National Aeronautics and Space Administration (NASA). *The Electromagnetic Spectrum*. URL: `https://imagine.gsfc.nasa.gov/science/toolbox/emspectrum1.html`.

[3] K. Ashton. "That 'Internet of Things' Thing". In: *RFID Journal* (2009).

[4] C. Prat B. Geffroy P. le Roy. "Organic light-emitting diode (OLED) technology: materials, devices and display technologies". In: *Polymer International* (2006).

[5] A. Chhabra B. Sidhu H. Singh. "Emerging Wireless Standards - WiFi, ZigBee and WiMAX". In: *International Journal of Electrical, Computer, Energetic, Electronic and Communication Engineering* 1.1 (2007).

[6] "Business models for the Internet of Things". In: *International Journal of Information Management* (July 2015).

[7] A. Joshi D. K. Mishra M. K. Nayak. *Information and Communication Technology for Sustainable Development*. Springer, Nov. 2017.

[8] R. Dixon. "Why Spread Spectrum?" In: *Communications Society* 13.4 (July 1975).

[9] S.A. Seshia E.A. Lee. *Introduction to Embedded Systems: A Cyber-Physical Systems Approach*. 2nd ed. MIT Press, 2017.

[10] *ESP32: A Different IoT Power and Performance*. URL: `https://www.espressif.com/en/products/hardware/esp32/overview`.

[11] "Future strategic plan analysis for integrating distributed renewable generation to smart grid through wireless sensor network: Malaysia prospect". In: *Renewable and Sustainable Energy Reviews Volume 53, January 2016, Pages 978-992 Renewable and Sustainable Energy Reviews* 53 (Jan. 2016), pp. 978–992.

[12] B. Weiss G. Gridling. *Introduction to Microcontrollers*. Vienna University of Technology, Feb. 2007.

[13] "How Low Energy is Bluetooth Low Energy? Comparative Measurements with ZigBee/802.15.4". In: *IEEE Wireless Communications and Networking Conference Workshops (WCNCW)* (2012).

[14] R. Ierusalimschy. *Programming in Lua*. 1st ed. Lua.org, Dec. 2003.

[15] *Internet of Things*. IoT European Research Cluster, Sept. 2009.

[16] H. Isselhorst. *IT-Grundschutz-Kataloge*. 15. Ergänzungslieferung. Bundesamt für Sicherheit in der Informationstechnik, Jan. 2016.

[17] R. Nair J.E. Smith. "The Architecture of Virtual Machines". In: *IEEE Computer Society* (2005).

[18] Hartmut Janocha. *ctuators: Basics and Applications*. Springer Science Business Media, 2013.

[19] D.Y. Bang K. Park. "Electrical properties of Ni–Mn–Co–(Fe) oxide thick-film NTC thermistors prepared by screen printing". In: *Journal of Materials Science: Materials in Electronics* 14 (Feb. 2003), pp. 81–87.

[20] "LoRaWAN - A Low Power WAN Protocol for Internet of Things: a Review and Opportunities". In: *International Multidisciplinary Conference on Computer and Energy Science*. July 2017.

[21] "Mixed-Mode WLAN: The Integration of Ad Hoc Mode with Wireless LAN Infrastructure". In: *Globecom*. 2003.

[22] W. Celes R. Ierusalimschy L.H. de Figueiredo. "The Implementation of Lua 5.0". In: *Journal of Universal Computer Science* 11(7) (July 2005), pp. 1159–1176.

[23] W.C. Filho R. Ierusalimschy L.H. De Figueiredo. "Lua-An Extensible Extension Language". In: *Software - Practice and Experience* 26(6) (June 1996).

[24] A. Saakian. *Radio Wave Propagation Fundamentals*. Artech House, Nov. 2017.

[25] J.M. Tjensvold. *Comparison of the IEEE 802.11, 802.15.1, 802.15.4 and 802.15.6 wireless standards*. Sept. 2007.

[26] R. Gouws W.A. Bisschoff I.N. Jiya. "Practical Considerations for Controller Selection in Residential Energy Management Systems: A Review". In: *International Journal of Applied Engineering Research* 13.10 (2018), pp. 7942–7949.