hochschule mannheim

Conception, Implementation and Postmortem Documentation of a Modular Proxy Application for Testing Internet of Things Applications

Moritz Laurin Thomas

Master Thesis

for the acquisition of the academic degree Master of Science (M.Sc.)

Course of Studies: Computer Science

Department of Computer Science

University of Applied Sciences Mannheim

31.05.2021

Tutors

Prof. Dr. Thomas Specht, Hochschule Mannheim

Pierre-Alain Mouy, M.Sc., NVISO GmbH

Thomas, Moritz Laurin:
Conception, Implementation and Postmortem Documentation of a Modular Proxy Application for Testing Internet of Things Applications / Moritz Laurin Thomas. –
Master Thesis, Mannheim: University of Applied Sciences Mannheim, 2020. 67 pages.

Thomas, Moritz Laurin:
Konzeption, Implementierung und Post-mortem-Analyse eines modularen Proxys zum Testen von Anwendungen im Internet der Dinge / Moritz Laurin Thomas. –
Master-Thesis, Mannheim: Hochschule Mannheim, 2020. 67 Seiten.

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ich bin damit einverstanden, dass meine Arbeit veröffentlicht wird, d. h. dass die Arbeit elektronisch gespeichert, in andere Formate konvertiert, auf den Servern der Hochschule Mannheim öffentlich zugänglich gemacht und über das Internet verbreitet werden darf.

Mannheim, 31.05.2021

Moritz Laurin Thomas

Abstract

Conception, Implementation and Postmortem Documentation of a Modular Proxy Application for Testing Internet of Things Applications

TBD

Konzeption, Implementierung und Post-mortem-Analyse eines modularen Proxys zum Testen von Anwendungen im Internet der Dinge

TBD

Acknowledgements

TBD

Contents

Chapter 1

Introduction

This chapter will introduce the underlying motivation of this thesis. Then, it will give an overview of this thesis' purpose and structure. Lastly, this chapter will show the process that the work on this thesis went through, explaining the scientific methods applied and software engineering practices used.

1.1. Motivation

Today scientific and industrial parties work on connecting physical entities such as machines, buildings and even humans to the internet by equipping them with digital sensors and actuators, referred to as "Internet of things (IoT)". While this progression promises many positive effects, such as simplifying tasks in our personal day-to-day life ("Smart Home" applications), monitoring our personal health ("eHealth") and increasing efficiency and safety of industrial plants ("Industrial internet of things (IIoT)", also referred to as "Industry 4.0"), it also yields the risk of introducing new attack-vectors to parts of our environment: "smart" devices used at home or at other sensitive places may implement weak security implementations or faulty security design, resulting in private and personal data being available to parties interested in violating the privacy of one's home (e.g. vaccum robots leaking information about the interior design of homes[10]) or conducting industrial espionage which is an acute threat [2, p. 14].
The diversity of both deployed smart devices and the internet services those devices are connected to lead to the need and use of ever-increasing complex technologies used for communication, data storage and access management, further adding to po-

tential attack-vectors of connected devices and distributed applications [5, p. 119]. This complexity and the sheer number of connected devices is actively being exploited by attackers today and the number of attacks on IoT devices is increasing [4].
There are security guidelines, best practices and innovative approaches for developing secure smart applications [5, p. 120][7, p.326-328], however testing such applications proves to be cumbersome: intercepting, dissecting, inspecting and manipulating the communication in these applications requires working on various abstraction layers. In order to evaluate the security of such applications, penetration testers often spend a considerable amount of time dissecting applications and setting up a test-environment.
The goal of this thesis is to conceptualize, implement and evaluate a modular proxy application that supports evaluation of the security implementations of IoT applications.

1.2. Purpose and Structure of the Thesis

This thesis is separated into eight chapters: chapter (2) will give an overview of and discuss related and previous work. After that, relevant fundamentals about computer networks, IoT applications and information security will be covered in chapter 3.
The chapters 4 to 7 describe the research and development process of the IoT proxy application in chronological order: the problem space of the application is shown and dissected in chapter 4, yielding essential insights into potential challenges and technical requirements. Building upon these, the conceptual design of the IoT proxy application is proposed in chapter 5. This included the process of collecting, documenting and analysing of software requirements, describing the application's work context and designing a software architecture that complies with the aforementioned requirements. Subsequently, chapter 6 involves a prototypical implementation of the aforementioned software concept, focusing on the goals and constraints of the implementation, the tools and frameworks used and the implementation of core components of the application. The resulting implementation and the project itself are then analysed in a postmortem documentation, pointing out the reasons why and how the project ultimately failed.

The thesis ends with a summary of all results produced and conclusions drawn from the work on this thesis.

Chapter 2

Related Work

This chapter will discuss related and previous work on topics in this thesis' context. This includes network analysis in general (and IoT in particular), homogenization and unification of various IoT related technologies and performance of security evaluations of these technologies.

2.1. Computer Network Analysis in General

TBD: Polymorph [8]

2.2. Homogenization of the IoT Landscape

TBD: IoT proxy for homogenization [6]

2.3. IoT Security Analysis

As part of their master's thesis, Bellemans conducted a study in 2020 that evaluated the security and privacy implementations of fifteen "*smart*" devices from a wide price range available on the market at the time. They performed automated analyses and requested data access from manufacturers [3]. The thesis showed that the devices made use of a variety of both standardized and proprietary transport and application protocols. It also found severe flaws in the devices' compliance to General

Data Protection Regulation (GDPR): about a third of the devices' manufacturers did not reply to GDPR requests at all, however Bellemans noted that the COVID-19 pandemic may have had an impact on their data access requests. The thesis suggests that the introduction of a quality label that guarantees appropriate implementation of security and privacy aspects could prove beneficial for customers.

In 2017, Apthorpe et al. presented a three stage strategy to examine metadata of network traffic of four smart devices [1]. By monitoring the devices' traffic, they showed that even though the communication between the devices and their corresponding internet servers were encrypted, passive observers could deduce information about users' behaviour by identification of the destination server and analysis of the rate of traffic being sent. A noteworthy aspect of their work is that they performed this analysis from an Internet Service Provider (ISP)'s point of view, exclusively examining metadata of the communication that took place. The strategy described in the paper consists of the following (greatly simplified) steps:

1. Identifying communication streams of individual devices (e.g. by examining the TCP packets' destination IPs).
2. Associating the streams with specific device models (e.g. by performing reverse-look ups of the aforementioned IPs).
3. Analysing traffic rates (presuming that traffic is generated upon taking measures).

TBD: Add simple process diagram

Apthorpe et al. conclude that their strategy works well on inferring behaviour from regular internet traffic of smart devices, however they assume that shaping traffic or making use of proxies (that effectively mask the destination IPs) could be effective counter-measures. It is safe to assume that regular smart home setups do not make use of proxies or traffic shaping though, thus being vulnerable to this kind of attack.

TBD: NVISO Labs: Théo Rigas, IOXY [9]

Chapter 3

Theoretical Background

This chapter provides an overview of the technologies and concepts referred to in subsequent chapters. Starting with section 3.1, essential concepts of computer communication in networks will be presented and examined, covering the concept of network layers, intercepting of communication between two parties and analysis of transferred data. Building upon these fundamentals, section 3.2 introduces the fields of use of IoT applications, common architectures used today to implement them and popular protocols they make use of. Lastly, it will discuss security considerations important to IoT applications. After that, section 3.3 will provide insights into relevant concepts and the practices used and applied in information security. It covers key concepts and legal considerations, integration of information security in software development and common practices and methods involved.

3.1. Computer Networks

3.1.1. Network Layers

TBD Transmission Control Protocol (TCP)

3.1.2. Proxying Network Traffic

TBD; planned:

1. Definition; Working Principle
2. Use Cases

3. Abuse Cases

3.1.3. Deep Packet Inspection

TBD

3.2. (Industrial) Internet of Things

3.2.1. Fields of Use

3.2.2. Application Architectures

3.2.3. Common Protocols

Building up on pre-existing network infrastructure and in order to meet requirements specific to individual fields of use and use-case scenarios, the landscape of IoT attends with a great variety of *communication protocols* (further used to refer to both transport and application protocols). This section will provide a brief overview of the working principles, use cases and history of some protocols commonly used in IoT and IIoT applications today.

Hypertext Transfer Protocol (HTTP) *TBD*

WebSockets (WS) *TBD*

message queuing telemetry transport (MQTT) *TBD* Amazon Web Services (AWS) IoT

Modbus TCP *TBD*

Profibus/Profinet *TBD*

OPC Unified Architecture (OPC U/A) *TBD*

3.2.4. Security Considerations

3.3. Information Security

TBD

3.3.1. Key Concepts

3.3.2. Legal Background

Compliance

Data Protection

3.3.3. Integration in Software Development

Traditional Approaches

Modern Approaches

3.3.4. Methodology

Risk Management

Incident Response

Reverse Engineering

(Physical) Penetration Testing

Source Code Audits

Application Configuration

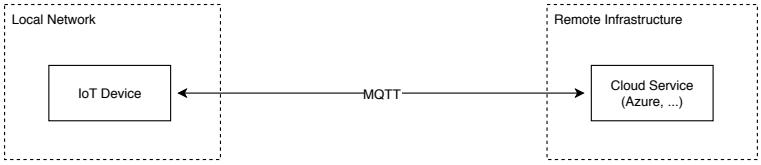
Chapter 4

Understanding the Problem Space

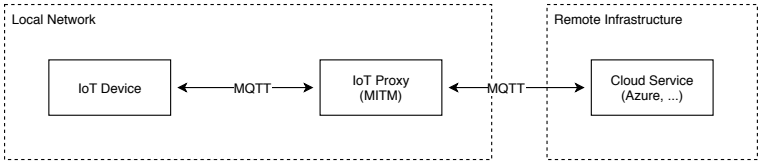
In order to provide a satisfying solution to the problem at hand, the problem itself and the environment it occurs in must be researched. This chapter aims to explore and examine the problem space, resulting in a set of artefacts (namely a domain model and a set of requirements) that aid in understanding the context and designing an appropriate solution. First, a prototypical network proxy is designed and implemented in section 4.1 to get an understanding of the problems and challenges involved in designing, implementing and using such software. Based on these experiences, interviews with experts in penetration testing are conducted and evaluated in section 4.2 to get a proper understanding of their everyday work and resulting problems. Lastly, existing software that aims to intercept communication for various scenarios and technologies is examined in section 4.3, compared to each other and their usefulness for the problem-specific scenarios is assessed.

4.1. Prototypical Implementation

The prototype was designed to be used in three different scenarios, each taking place in a different context. The goal of this section was to implement a prototype that could be used as a proxy to intercept communication between an IoT device and its cloud service as shown in figure 4.1. It was developed incrementally so individual components could be derived from requirements, designed, implemented and evaluated in fixed sprints.



(a) Regular communication between an IoT device and a cloud service.



(b) Communication intercepted by a man-in-the-middle (MITM) proxy.

Figure 4.1.: Installing a MITM proxy to intercept network communication for penetration testing.

4.1.1. Example Scenarios

The following scenarios describe hypothetical configurations of IoT/IIoT devices that should be tested with the prototype and vary in both technical and logical complexity as well as in closeness to reality:

Scenario #1: Legacy industrial control system (ICS) Application In this IIoT scenario, a human-machine interface (HMI) (*Siemens KTP400 Basic*) sends commands to and receives data from a programmable logic controller (PLC) (*Siemens S7-1200*) using Modbus TCP. The PLC continually counts up a value up to 100 and begins anew at zero while the HMI displays the current value and provides a button that, upon being pressed by a user, resets the current value to zero.
In this scenario, attackers could perform a variety of attacks on the system by intercepting and manipulating network traffic, for example:

- By dropping messages sent from the PLC to the HMI, the application may appear unresponsive as new data is not displayed on the HMI. In production environments, this could lead to dangerous situations as sensor readings that indicate harmful environmental conditions would not be presented to supervising personnel.
- When dropping messages sent from the HMI to the PLC, control commands can be suppressed. This attack can result in catastrophic situations when emergency shutdowns issued by supervising personnel are not registered by the affected machines.

This scenario is of an academic nature and does not depict a realistic ICS process, but focusses on the use of a legacy transport protocol. Due to the rather simple nature of the Modbus TCP protocol, intercepting and manipulating communication is expected to be trivial.

Scenario #2: IoT Cloud Application This IoT smart home scenario utilizes two local IoT devices that are integrated into a cloud environment such as the AWS IoT platform: a thermometer and an air conditioner (A/C) unit. Both devices connect to the cloud platform, authorize themselves at a Representational State Transfer (REST) interface via HTTP and upgrade their HTTP-connection to WS streams upon successful authorization. They eventually communicate to a remote MQTT broker by tunnelling MQTT packets over the WS stream. At this stage, the thermometer publishes temperature readings to an MQTT topic while the A/C unit subscribes to the same topic and adjusts its operation depending on the incoming temperature readings.
This distributed communication setup introduces a set of possible attacks that could be performed when attackers *impersonated* client-devices or the remote server:

- Impersonating the thermometer, attackers could send incorrect temperature data and effectively control the A/C unit. When sending low temperature readings while the environment temperature is high, the A/C unit would stop running. Conversely, when high temperature readings are sent while the environment temperature is low, the A/C unit would run, and thus further cool down the environment.
- Attackers that impersonate the remote server could drop or manipulate incoming publish packets, thus altering whether and/or what information is relayed other connected devices. For example, temperature readings that indicate a high environment temperature that would lead to the A/C unit to be powered up could be rewritten in such a way that the transmitted temperature value is considered to indicate a low environment temperature, thus preventing the A/C unit from running automatically.

This scenario makes use of three communication protocols, uses these protocols dependent on the state of authentication and even tunnels one protocol through another one. Therefore the proxy application has to implement a state-machine (as seen in figure 4.2) and testing communication in this scenario is expected to be more complex than the first one. Also, since this scenario makes use of the AWS

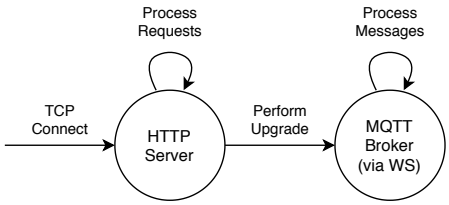


Figure 4.2.: State machine of AWS IoT communication

IoT infrastructure, special care must be taken to ensure that authentication is properly relayed to the cloud servers.

Scenario #3: Water Treatment Plant Similar to scenario #2, this scenario makes use of MQTT for transporting messages. However, the scenario takes place in an ICS context of critical infrastructure.

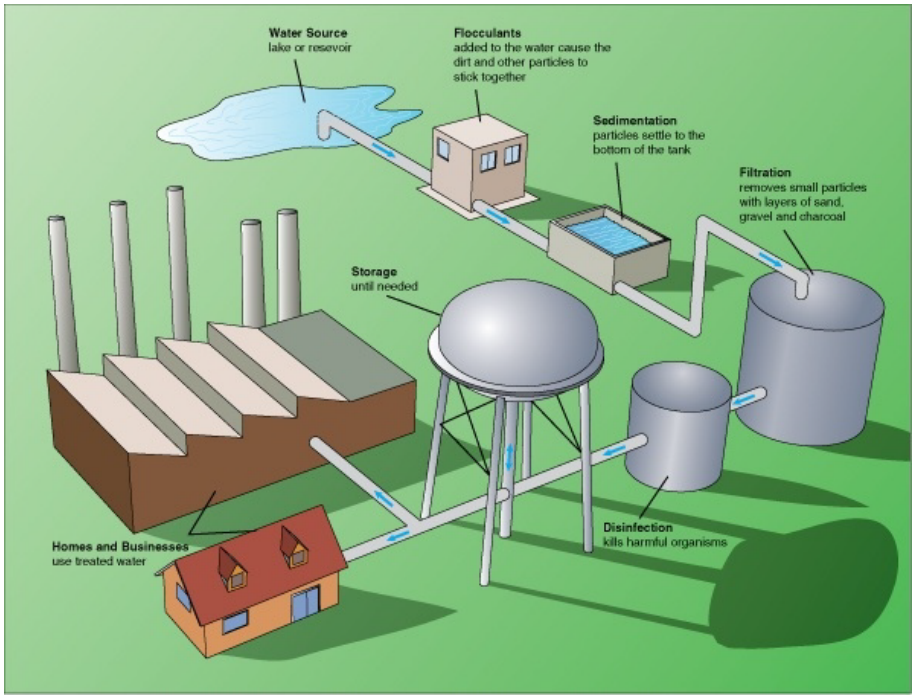


Figure 4.3.: Illustration of a typical drinking water treatment process. (by the CK-12 Foundation)[1]

As shown in figure 4.3, there are multiple steps involved in treating water for drinking. The scenario represents these steps as separate stations (“source”, “flocculants”, “sedimentation”, “filtration” and “storage”) that act as MQTT clients. Each station receives water into an input tank, processes water from its input tank in a specified rate and flushes processed water into an output tank. Similar to how

[1] https://en.wikipedia.org/wiki/File:Illustration_of_a_typical_drinking_water_treatment_process.png

threads can suffer from starvation in a multithreading environment, these stations can either "run dry" when their input tank is empty or overflow when either tank is filled beyond their capacity. In this scenario, stations will only process water from their input tanks if their output tank provides sufficient available capacity.
Similar to scenario #2, attackers could perform a series of attacks in this scenario:

- Without intercepting any communication, attackers could cyclically overwrite water levels of either input and output tanks to stop stations and bring processing to a halt. For example, when attackers overwrite the "storage" station's input tank level to indicate exhausted capacities, the "disinfection" station would fill its output tank and eventually stop processing water. This would cause the "disinfection" station's input tank to fill up and lead to the "filtration" station's output tank to fill up. Ultimately, the water treatment plant would halt.
- When any station publishes data about its tanks' levels indicating full or empty capacities, attackers could intercept those messages and change them to either indicate the opposite (change tank levels indicating *full* capacities to levels indicating *empty* capacity) or some arbitrary level information. This could lead to either pumping water from empty tanks, potentially damaging the pumps, or to overfilling tanks, leading to leaking excess water and potentially damaging further equipment.
- Attackers that intercept messages between the stations and the broker can collect and analyse them and try to draw conclusions about the use and activity of the system. This may allow attackers to identify patterns that show when the plant is operating at high capacities, maximizing the effect of attacks against the plant.

This scenario greatly simplifies drinking water treatment by reducing the process to the producer-consumer problem known from multithreading. A more realistic representation of drinking water treatment plants would take further details, such as chemicals used in disinfection, into account.
This scenario involves only MQTT as a transport protocol but requires six MQTT clients to run simultaneously.

Derived Use-Cases

Summarizing the scenarios detailed above, a number of high-level use-cases can be derived from them (shown in 4.4). The actors are the *attacker* that intends to interact with the system in a potentially malicious way and the *server* and *client* that use the system for transportation of messages. The following use-cases are found in the aforementioned scenarios:

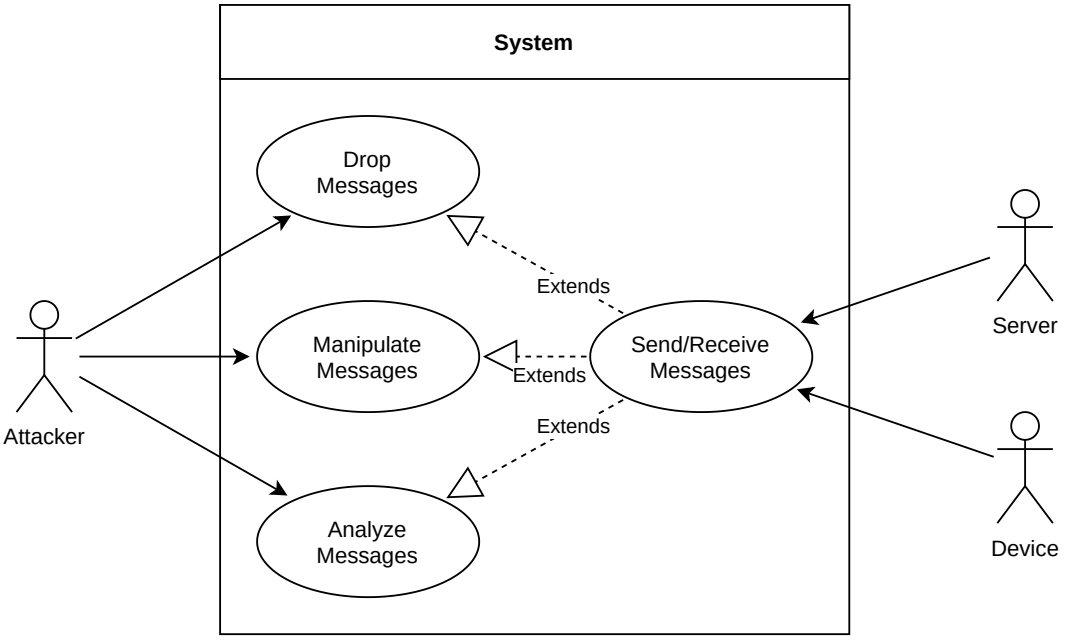


Figure 4.4.: High-level use-cases of a proxy in a generic IoT/ICS environment.

- **Send/Receive Messages:** The server and client send and receive messages to communicate with each other. This interaction does not require interaction with the attacker.
- **Drop Messages:** The attacker discards incoming or outgoing messages by not relaying them to the intended target. This can cause loss of control in the application that this communication takes place in.
- **Manipulate Messages:** Incoming or outgoing messages can be changed by an attacker, altering various properties such as Quality of Service (QoS) (for MQTT messages), host (for HTTP requests) or the payload of a message (e.g. the content of an HTTP response).
- **Analyse Messages:** Attackers can collect and analyse passively without altering them, allowing them to deduce information about the behaviour of the affected system and potentially its user(s).

4.1.2. Requirements

To be able to operate in all of the aforementioned scenarios, the prototype had to implement a set of functional requirements:

F1 **Protocols:** The software must implement parsing/crafting messages/packets of the communication protocols: HTTP, WS, MQTT and Modbus TCP.
Fit criterion: The software must implement and support the HTTP, WS and MQTT protocols so messages of those protocols can be further processed by the software.

F2 **Network Stacks:** The software must be able to parse protocols that are tunnelled through other protocols ("*stacked*"). It must provide an interface to the user where they can specify which communication protocols are used and whether and how they are stacked (further referred to as *network stack*).
Fit criterion: The software processes a configuration file that lets users specify which protocols to be used and whether/how they are stacked.

F3 **State-Machines:** The software must be able to switch network stacks and scripts for processing dependent on configurable *states* and *transitions* between them. It must provide an interface for the user to specify when to switch to using another network stack, represented using finite-state machines (FSMs) and rule sets for transmission between states.
Fit criterion: The software processes a configuration file that lets users specify when to switch between network stacks.

F4 **Integration:** The software shall provide interfaces for integration of third-party software.
Fit criterion: The software implements interfaces that allow sending packets to other applications such as "Burp Suite".

F5 **Scripting:** The software shall provide scripting capabilities for automated manipulation and discarding of messages.
Fit criterion: Users can define scripts that are executed on messages.

F6 **Logging:** The software shall provide means for collecting and saving messages for future analysis.
Fit criterion: The software saves messages to a MySQL database.

The following non-functional requirements were defined:

N1 Platform Compatibility: In order to support a broad spectrum of target platforms, the software shall be implemented platform-independently.

N2 Reusability: The software shall be reusable so it can be used in future tests that may feature new configurations of network stacks.

N3 Open Source: The software shall be available as open source software so programmers and members of the IT community may contribute to improving it.

Due to this implementation serving as a prototype and being of an academic nature, no specific constraints were defined. It was to be developed strictly ignoring aspects of usability and stability as it should not be used in production environments but in laboratories exclusively.

4.1.3. Design

The prototype was designed to be fit for use in the second scenario as, regarding network communication, it was more complex than the other ones. Specifically, the second scenario demanded the implementation of a network stack and a state machine to switch between states. Parsing protocols that were tunnelled through other protocols appeared to be a potentially challenging requirement which is why the focus on the design and implementation of this prototype was on the underlying management, processing and relaying of messages. In order to tackle it, a variation of the *pipeline* (sometimes referred to *pipes and filters*) design pattern was used (as shown in figure 4.5). It was designed to be used as follows:

Messages originate from a listener, for example messages with raw byte payloads are received from a TCP socket. These messages are sent to an initial *pipe* to be processed *down*.

Pipes are bi-directional routers that represent processing-steps of pipelines and perform the following actions on messages that are processed through a pipeline:

1. Pipes use optional *encoders* to disassemble/de-serialize messages when processing them *down* the pipeline and re-assemble/serialize them when they process messages *up* the pipeline.
2. Pipes can use “filters” to perform operations on messages such as replacing header values or manipulating payloads.

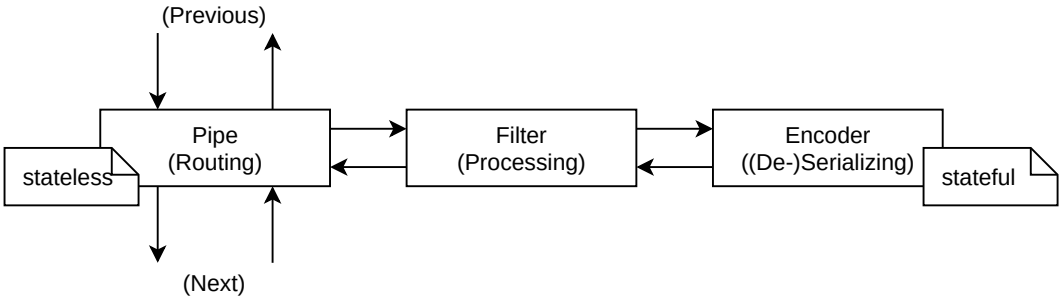


Figure 4.5.: The variation of the "pipes and filters" design pattern used in the prototype.

3. They forward messages to the next pipe in its pipeline when processing messages down or to the previous pipe when processing messages back up.

There are extensions to basic pipes such as:

- *EndPipes* are appended to the end of a pipeline and reverse the message processing direction so messages that were processed down are sent back up the pipeline to be processed up.
- *ProcessingPipes* mandate encoders and filters to be used. These pipes are used to indicate that messages are not not only routed but also processed and encoded or decoded.
- *IntegrationPipes* allow integration of other software into the pipeline. For example, penetration testing software such as Burp Suite could be integrated.

An exemplaric configuration of the pipeline design pattern envisioned for this prototype for use in the AWS IoT scenario is shown in figures A.1 and A.2. These diagrams visualize how messages are processed *down* and back *up*.
Figure A.1 shows the first state of the AWS IoT scenario that processes HTTP communication. It features a TCP server socket that accepts incoming connection requests from an IoT device and a client socket that is connected to the AWS cloud. Since the communication to the AWS cloud is Transport Layer Security (TLS)-encrypted, it is first decrypted by a filter and then processed by a HTTP filter. Then, the parsed HTTP requests and responses are sent to external tools (e.g. BurpSuite). Once the end of the pipeline is reached, the messages are sent back up the pipeline, being encoded back into a form usable for the IoT device or cloud server.
Once the prototype detects that the state must be changed to processing MQTT over WS communication, a different network stack is initialized and used, as shown in figure A.2. In this state, TLS-encryption is decrypted and passed into a WS fil-

ter that (de-)serializes WS packets. The payload of data frames is then forwarded to an MQTT layer. In this specific configuration show in figure A.2, the payload of MQTT messages is (de-)serialized as JavaScript object notation (JSON) before being sent to external tools by the integration pipe.

4.1.4. Testing

To test the prototype, a simple testbed was designed and implemented to realize scenario #3 (discussed in section 4.1.1). It consisted of two Debian 10 machines that acted as a MQTT broker and clients and a Kali Linux machine that ran the prototype and provided tools such as Wireshark that could be used for debugging and monitoring network traffic. All machines were connected to a single network (shown in figure 4.6) and were assigned static Internet Protocol (IP) addresses. While this setup allowed for more sophisticated MITM mechanisms such as Address Resolution Protocol (ARP) spoofing, the decision was made to directly connect the MQTT clients to the *kali* machine to reduce complexity and accelerate and simplify testing. Separate machines were used for the MQTT broker and clients so that actual MITM attacks could be performed if the need to arose. Also, running the broker on a separate machine simplified debugging as network traffic could be attributed to broker or clients easier by examining the packets' source and destination IPs.

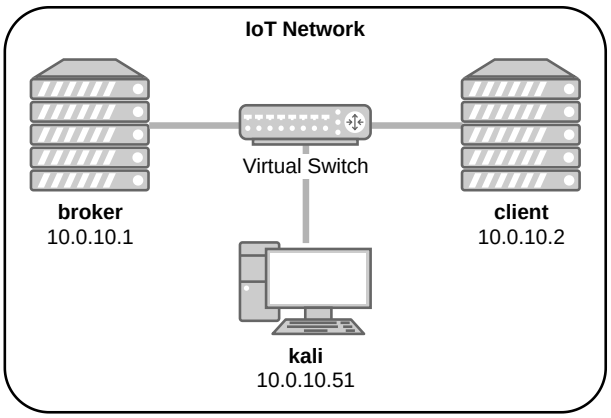


Figure 4.6.: A network diagram of the testbed that was used for testing the prototype.

The MQTT broker software used on the *broker* machine was Eclipse Mosquitto[2] 1.5.7 and had the WS transport enabled to allow for clients to connect via WS. The

[2] https://mosquitto.org/

MQTT clients running on the *client* machine were implemented in Python using the Eclipse Paho library for Python (paho-mqtt[3]).

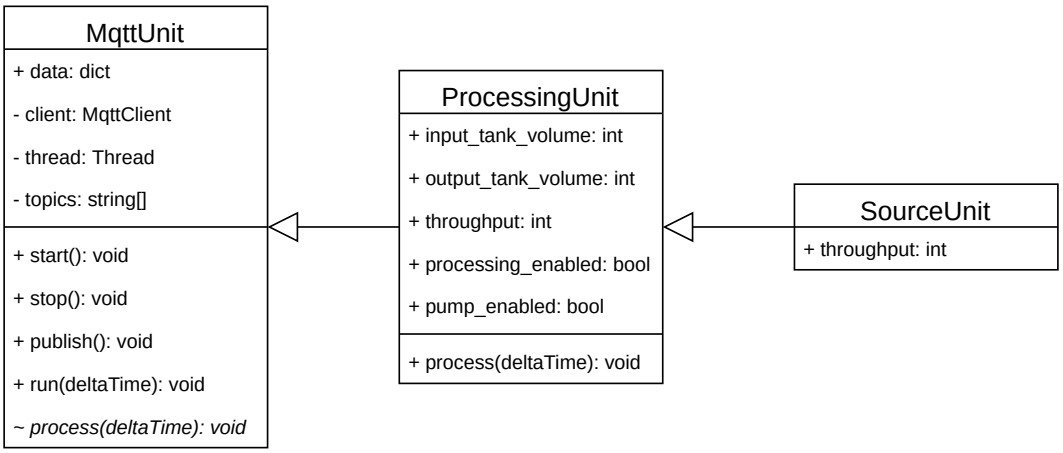


Figure 4.7.: The "ProcessingUnit" data-structures represent individual stations of the simplified water treatment plant.

The water treatment scenario required water treatment stations to be simulated individually as separate MQTT clients, which was done by representing them as "ProcessingUnits" in the Python implementation of the testbed. As can be seen in figure 4.7, ProcessingUnits held individual *MqttClient* instances running in separate threads, were subscribed to the topics of relevant other units such as their direct predecessors and successors and were capable of publishing their current state. Their *process* method would be called cyclically and allow for units to calculate their intake, throughput and output.

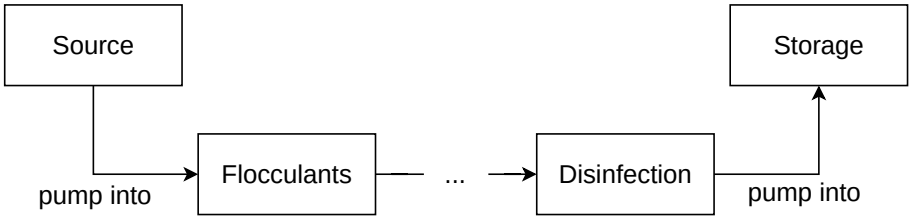


Figure 4.8.: Chaining of the water treatment units, originating from a water source and eventually leading to a storage at the end of the processing pipeline.

These units were then "chained" up (shown in figure 4.8) in the order in which they were presented in the scenario by specifying their direct predecessor and successor units: potentially contaminated water would be pumped out of the *source*, processed by a series of stations and eventually flushed into the *storage*. The *source* was an instance of the "SourceUnit" that featured a throughput calculated by a sine-wave

[3] https://pypi.org/project/paho-mqtt/

function that used the elapsed time since program startup as input parameter. Also, in order to keep the program running infinitely without either the *source* "running dry" due to its input tank emptying or the *storage* overfilling, the *storage's* output was programmed to feed back into the *source's* input tank (as can be seen in figure 4.9). While this was not a realistic approach, it kept the program's design simple and allowed for continuous testing and did not impact the MQTT communication.

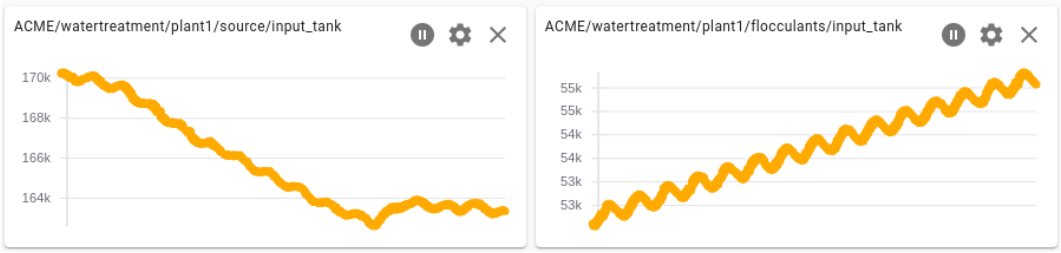


Figure 4.9.: Screenshot of the application "MQTT Explorer" that was used to inspect and visualize the state of the water treatment plant. The left graph shows how the *source's* input tank steadily emptied until it was filled by the *storage's* output tank. The right graph shows how the *flocculant* unit's input tank slowly filled up.

4.1.5. Implementation

The prototype was partially implemented over the course of ???weekly sprints after which work on the prototype was halted. It was written in TypeScript due to the language's flexibility and static typing. It allowed to precisely specify interfaces and its runtime (NodeJS) would allow it to make use of asynchronous programming, which would benefit this prototype. The rough design worked out in section 4.1.3 was specified in greater detail so individual classes could be derived and implemented.

Pipes and Filters As shown in figure 4.10 the pipeline design pattern was altered in such a way that the basic *IPipe* interface was implemented by the *BasePipe* class that held a reference to a single *IFilter* but lacked a reference to an *IEncoder*. Filters would hold a reference to encoders because encoders were used directly by filters for (de-)serialization prior to any other processing (such as executing scripts). Thus, encoders would not exist without filters, resulting in a composition relationship between the two. As indicated by their prefix *IFilters* and *IEncoders* were only interfaces that set a behaviour for their specific uses: encoders would implement (de-)serialization of specific protocols while filters added logic to processing

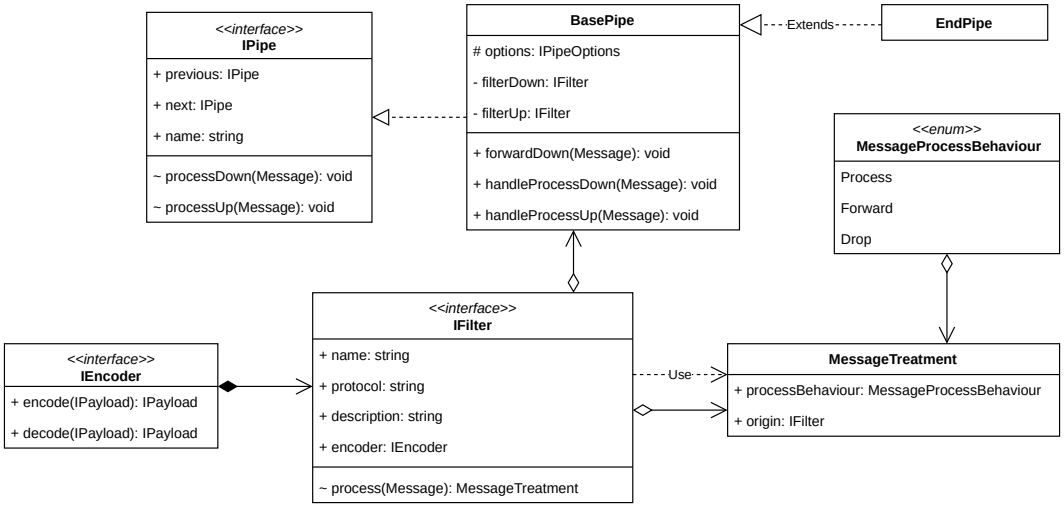


Figure 4.10.: The classes and interfaces used to implement pipelines in the TypeScript prototype.

(de-)serialized packets, such as executing scripts. The prototype implemented encoders for HTTP and MQTT as well as a *BaseFilter* that did not add any logic to processing but allowed to test the encoder implementation. Also, specific *NopFilter* and *NopEncoder* ("Nop" meaning "no operation") classes were implemented that did not implement any logic. This was used to test sending messages down and up the pipeline without processing them at all. The MQTT encoder used the "mqtt-packet"[4] library which offered a comparatively simple API for serializing (*generate(Packet)*) and de-serializing (*parser.parse(Buffer)*). However, lacking a library that offered a similar low-level and simple API for HTTP (de-)serialization, a custom encoder for these tasks was implemented. Due to HTTP being a comparatively simple and text-based protocol, all that needed to be done for de-serialization was parsing the HTTP headers (separated by new-lines) and, depending on whether or not the "Content-Length" header was present, extracting the HTTP body.

Messages and Payloads The pipeline system implemented routing and processing of messages, however this required a concept of what messages are and how they convey the information required to perform meaningful and useful operations on network communication. Figure 4.11 shows the classes and interfaces that defined the messages and payloads types. *Messages* hold basic information such as a unique identifier, the communication protocol they were sent through and metadata that was used to store header information in. The *IPayload* interface allowed for implementation and use of various payload formats such as raw binary information

[4] https://github.com/mqttjs/mqtt-packet, commit 4b6278d890e0c2fca01da62c5f9b63e05f5fd899

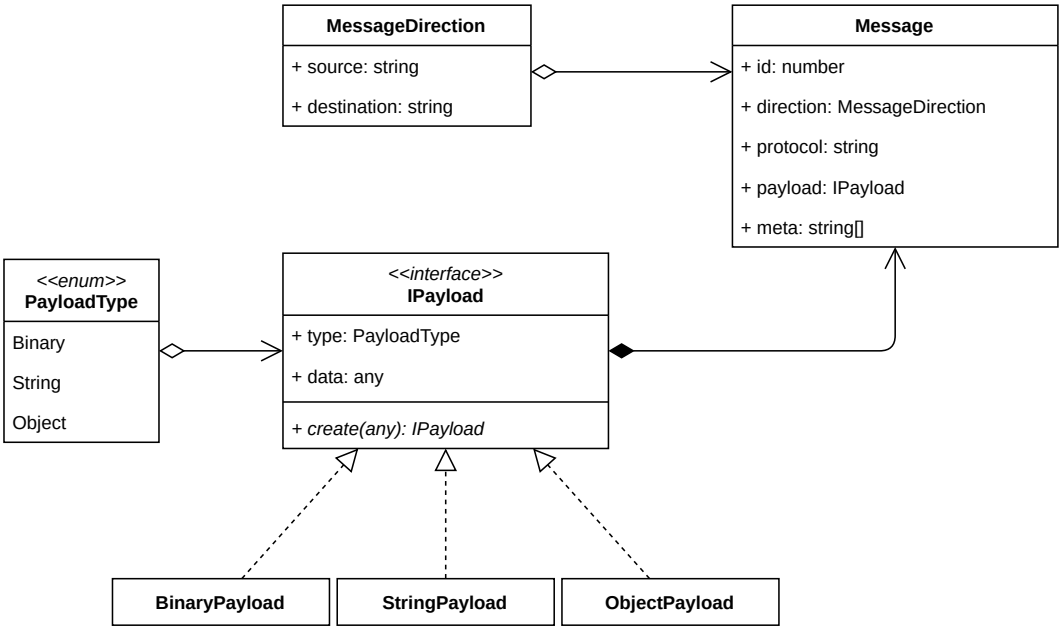


Figure 4.11.: ?

(e.g. MQTT message bodies), string contents (e.g. HTTP response bodies when the *Content-Type* header indicated *text* data) or JavaScript objects such as dictionaries that could hold arbitrary data for cases where there was no meaningful way to extract payloads from messages. The *MessageDirection* structure was used to relay the message to the correct socket after it was processed by the pipeline.

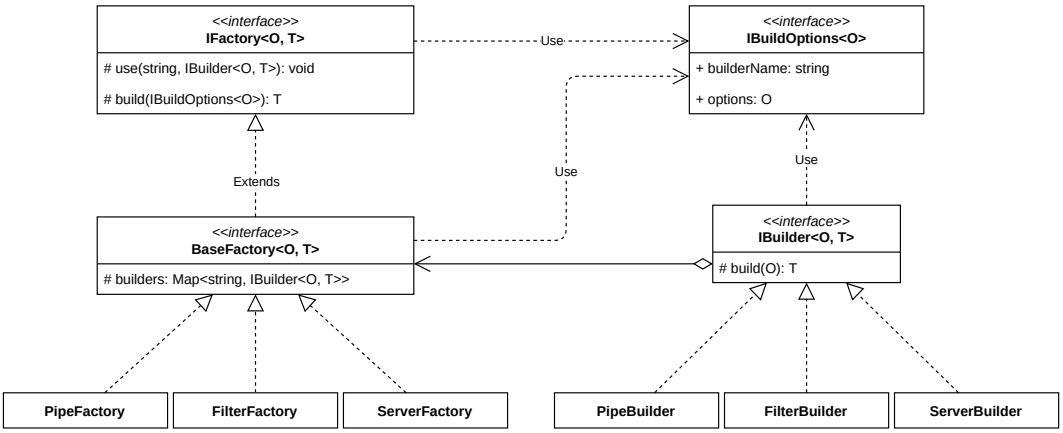


Figure 4.12.: ?

Factories and Builders The requirement "F2 Network Stacks" implied a way of initializing various objects that represent pipelines, sockets and FSMs, dependent on configuration files loaded at runtime. As shown in figure 4.12, the factory design

pattern was used to provide an easy way to initialize pipes, filters, encoders and sockets while passing them metadata used for object creation. An *IFactory* interface exposed simple methods for registering *IBuilders* and building objects. The generic type parameters *O* and *T* were placeholders for type specific options (e.g. *IPipeOptions* and *IFilterOptions*) and the type of the created objects (e.g. *IPipe* and *IFilter*), respectively. The options types would contain information that was used for creating individual instances, such as a pipe's name or a server socket's address to listen on. The *BaseFactory* class implemented the *IFactory* interface and held an internal hash-map that was used to register *IBuilders* by name. Lastly, the *IBuilder* interface provided a method for initializing objects with the given options, providing default values for constructor parameters. There were static instances of factories and builders of pipes, filters and servers. For instance, the global server-factory *SERVER_FACTORY* used the global TCP server builder instance *TCP_SERVER_BUILDER*.

4.1.6. Insights Gained

The following insights were gained through the prototypical implementation. Some resulted in questions relevant for the expert interviews that were to be held:

- Due to the maximum transmission unit (MTU), large messages were broken into chunks that were transferred sequentially. This required the proxy to work on streams of incoming data and reassemble messages from said chunks. While individual MQTT messages would often be short enough to be transmitted in a single TCP packet, other communication protocols such as HTTP could yield messages that were several hundred kilobytes or more in size (e.g. when downloading images). This also required the encoders to be stateful as they had to load data of incoming packets into individual buffers until they could parse complete messages, introducing the need to initialize one pipelines per device connected to the proxy application.
- Supporting multiple client devices was non-trivial as communication between clients and servers was not necessarily connection-oriented (e.g. HTTP) and individual client devices could not be detected reliably as the TCP server socket only had information about its peer's IP addresses that, in a real scenario, could be hidden behind another device that implemented IP masquerad-

ing (e.g. Network Address Translation (NAT)).
Q: Do penetration testers need to test multiple devices at the same time?

- In some cases, e.g. with WS data-frames, extending a message's payload resulted in its payload being split into multiple messages. This indirectly created new messages that, depending on the exact protocol used, needed to use generated values (such as an unique identifier) or context-specific information (e.g. authentication tokens used in HTTP headers). Also, some libraries would generate those values themselves and not define ways to specify those manually.
  Q: Do penetration testers require exact control over the implementation of protocols?
- Manipulating messages, automatically via scripting or by hand using third-party integrations (e.g. to *Burp Suite*), could introduce latency to the communication.
  Q: Are there strict timing requirements during penetration tests?
- Many libraries offered high-level functions to the programmer while avoiding exposure of low-level functionalities like crafting or parsing messages. Exposing such functionalities would require dissecting and altering libraries on a source-code level.

4.2. Interviewing Experts for Insights

Interviews may be an efficient way to get an expert's opinion on something they are proficient in. Thus, expert interviews were conducted to let security researchers give insight into their everyday work and the challenges they face when working with IoT and IIoT applications. The information and insights gathered in these interviews were then used to model a persona, various work scenarios and use-cases that as a whole aim to represent their work.

4.2.1. Interview Guideline

An interview guideline (shown in *TBD*) was created to keep focus on key points during interviews so that interviewees would not stray too far from the relevant

points. The guideline also served as a checklist so the interviewer could make sure that all questions and points that should be covered initially, were in fact covered by the end of the interviews. It was composed of three sections:

1. Experiences with IoT The answers to these questions would give insights into what kind of applications the security researchers had worked on in the past. Answers to question *1.1.* were of particular interest as they might represent what technologies were being examined by security researchers and may be popular in today's applications.

2. Processes in Everyday Life This section aimed to cover questions about the processes and tasks security researchers perform during penetration tests of IoT applications in their everyday life. Ideally, answers to those questions would show the approaches taken and challenges faced during their work, uncovering potential needs and underlying motivation.

3. The Future of IoT This section had security researchers assess what the future of IoT may be like from their point of view. This required the interviewees to make a critical assessment of the status quo.

4.2.2. Conducting Interviews

Interviews were conducted with six *NVISO* employees (Patrick Eisenschmidt, Cédric Bassem, Théo Rigas, Oliver Nettinger, Pierre-Alain Mouy, Jonah Bellemans) that all had worked on security assignments on IoT or IIoT applications in the past. There is considerable variety in

- the experience they had in working on security assignments in general: all interviewees had a strong background in cyber security that reached back multiple years except one who was a working student at *NVISO Labs* (Bellemans).
- the experience they have had in working on IoT/IIoT applications: two interviewees worked on assessing IoT/IIoT applications only occasionally (Eisenschmidt, Mouy), one was part of a car manufacturer's automotive security

team in the past (Nettinger) and two were part of *NVISO Labs* and worked with smart devices on a regular basis (Bassem, Rigas).

- the focus of their everyday work: two interviewees were *NVISO* chief executives and switched to working on management tasks rather than security assessments (Nettinger, Mouy), one was a working student finishing their master's thesis with a focus on legal aspects of IoT devices (Bellemans) and the remaining three worked on security assessments in a variety of fields (Eisenschmidt, Bassem, Rigas).

The duration of the interviews varied from 45 minutes to two hours depending on the amount and level of detail of information provided by the interviewees and the number of times that the interviewer had to ask further questions.

Due to the COVID-19 pandemic, interviews were conducted remotely over Microsoft Teams and recorded for later review and analysis. All interviews were conducted successfully, however some problems were had: due to unstable internet connections interviews were sometimes interrupted for up to 30 seconds, low bandwidth and low microphone quality sometimes made making out specific words and phrases very hard.

4.2.3. Interview Analysis

The answers interviewees gave to the various questions present in the interview guide varied greatly in detail. The following paragraphs attempt to summarize the essential statements interviewees made, sorted by the sections of the interview guide and ending with conclusions drawn from the interviews.

1. Experiences with IoT Asked about the technologies they encountered in their work, most interviewees stated that MQTT (5/6) and HTTP (6/6) were widely used in the smart applications they assessed. For IoT devices they found that Espressif microcontrollers such as the ESP32 and ESP8266 were used (2/6). Especially in cheap devices they found that custom protocols and infrastructure were used (2/6), whereas high-end devices usually used MQTT and HTTP and worked with well-known cloud infrastructures such as AWS, Microsoft Azure or Google Cloud Platform. Most interviewees worked on Smart Home products (4/6) with one notable exception being Nettinger who worked on Smart Cars.

Usually, there were no technical constraints for the interviewees when performing security assessments. There were some non-technical constraints such as working from a black-box perspective rather than working from a white-box perspective that would allow evaluating more security aspects of a system in less time (Eisenschmidt, Bassem, Rigas). Depending on the client and the exact application that was to be tested, interviewees said that they made use of either mobile lab environments (2/6) or stationary lab environments (3/6). Also, interviewees stated that they usually assessed devices and applications individually.

2. Processes in Everyday Life Regarding the goals of their assessments, interviewees would take on one of two approaches: The first was penetration testing (Eisenschmidt, Bassem, Rigas), aiming to evaluate as many components of a system as they could during their assessment. The second was red-teaming (Nettinger, Mouy, Bassem, Rigas) which aimed to get some level of access, preferably privileged, to a device or server in order to take influence on the application's logic or exfiltrate data. The scopes of their assessments was usually defined by the client and could include testing of devices, applications and firmware or performing source-code and cloud configuration reviews (Eisenschmidt).

The high-level tasks carried out during assessments would generally be the same across assessments: first, interviewees would inspect applications passively from a black-box perspective without interacting with them. This could incorporate looking for hardware interfaces on a device (Bassem, Rigas), looking for open network ports (Bellemans), reverse engineering Android applications and inspecting certain artefacts as manifest files (Eisenschmidt) and monitoring applications' network traffic (Bellemans). Nettinger stated that when working with cars, fuzzing was a task often carried out against bus protocol implementations because the devices implementing those protocols were often supplied by third parties and source-code was usually not available.

The tools used by the interviewees were mostly dependent on the technologies and protocols they worked with, such as Burp Suite for examining HTTP communication (Eisenschmidt, Bassem, Rigas, Nettinger, Mouy, Bellemans). However, some general tools were used for information gathering (nmap and nessus), monitoring (Wireshark) and networking (socat, mitmproxy). Bassem, Rigas and Mouy stated that they would occasionally implement their own tools or scripts when they found that there either were no tools available that suited their needs or those tools would

not work. According to Rigas, tools were highly specific to custom setups and preparing them up for use could be more challenging than actually using them. Bassem, Rigas and Bellemans stated that tools were often immature. Speaking of their automated tests performed on smart devices Bellemans criticized that automated tools often yielded inaccurate or incorrect results such as nmap reporting a game-server running on a smart lightbulb. Also, when manipulating communication of applications, interviewees generally were not interested in manipulating metadata such as headers but focused on the messages' payloads.

3. The Future of IoT When asked about the current challenges the interviewees were facing working on IoT assessments, they gave very individual answers: Eisenschmidt expressed concerns about data protection and cloud environments being a rather new technology that requires engineers to securely configure them. Mouy and Eisenschmidt stated that protocols and frameworks became increasingly complex and more and more devices interacted with each other, adding complexity to the security assessments. Also, there were a lot of custom protocols and frameworks that lacked proper tooling and were time-consuming to asses (Bellemans, Mouy). When working on IoT assignments, clients often had a traditional view on the assignments and occasionally wanted the testers to perform black-box tests only although additional white-box tests would potentially help covering more components and internals of applications (Bassem, Rigas).

Half of the interviewees stated that cloud computing will be more important and present in the future (Eisenschmidt, Nettinger, Mouy). They expect continued use of the comparatively old but proven HTTP (Bassem, Rigas) and the well-accepted MQTT (Bassem, Rigas, Nettinger). Regarding software development, they expect manufacturers of smart systems to involve IT security more into their development process (Bassem, Rigas, Nettinger) as well as use standardized frameworks (Mouy, Bassem, Rigas). However, they also stated concerns about the growing complexity of frameworks and the uncertainty of which frameworks will eventually gain wide acceptance (Bassem, Rigas). Regarding autonomous driving, Nettinger noted that current discussions about legal topics (such as the question about which party is to assume liability in case of accidents) will likely not come to an end anytime soon. Concerned about security and safety aspects of future IoT applications, Bellemans expressed the need for smart applications to be labelled or certified and they referred

to the European cybersecurity certification framework that is being worked on by the European Union Agency for Cybersecurity (ENISA).

Conclusions The interviews yielded a set of both very interesting and relevant insights into the interviewees' work and fields of expertise. The following insights served as a guide for further development of the proxy application:

- Smart devices often communicated via HTTP and MQTT. While the tools for security assessments with HTTP were very mature, there was a perceived lack of tools for MQTT.
- Often times, smart applications made use of proprietary protocols and infrastructure. While this was a fact the interviewees expect to be of less significance in the future, it still was of greater significance then.
- Penetration testers usually did not intend to test the protocol implementations used by applications but the contents transmitted over these protocols.
- Tools for working with specific protocols were often very immature and both installation and usage involved a lot of work.

These insights were translated into the following, new requirements:

N4 **Extensibility:** To allow for future implementation of further communication protocols the software shall be implemented in a modular fashion.

N5 **Deployment:** To allow the proxy application to be installed and used in a repeatable and reliable way, the proxy application shall be distributed using a deployment system.

4.3. Analysis of Existing Software

Wireshark more than 3,400,000 lines of C code[5]*TBD*

MITMf *TBD*

Ettercap *TBD*

[5]This number was returned by the *cloc* utility run on commit *3a8111e1c2adcdc0603993c6ed5d20a40f162125* from Aug. 4th 2020 of Wireshark's Github mirror.

bettercap *TBD*

mitmproxy *TBD*

mProxy *TBD*

IOXY *TBD*

scapy *TBD*

TBD; planned: paragraph about each program including a general description, uses, capabilities and usefulness

Name	*Latest Release*	*Implemented in*	*Supported Protocols*	*R*	*W*	*D*
Wireshark	2020-07-01	C	Various	F	N	N
MITMf	2015-08-28	Python	Various	?	F	F
Ettercap	2019-07-01	C	Various	F	F	F
bettercap	2020-03-13	Go	Various	F	F	F
mitmproxy	2020-03-13	Python	HTTP/S, WS	P	P	P
mProxy	Pre-Releases only	Go	MQTT	?	F	-
IOXY	Source only	Go	MQTT	F	F	F

Table 4.1.: Comparison of existing software where R, W and D describe read, write and deletion capabilities, respectively. F, N and P indicate full, no or partial functionality, respectively.

Chapter 5

Conceptual Design

Building on the software design of the first prototype presented in section 4.1.3 and the insights gained in section 4.1.6, two design concepts were worked out. The following sections will detail components and principles of both concepts.

5.1. Design #1: Monolithic Proxy Application

This design concept is based on the general ideas presented in section 4.1.3 (e.g. state-machines, network stacks and pipes) and employs a basic architecture shown in figure 5.1.

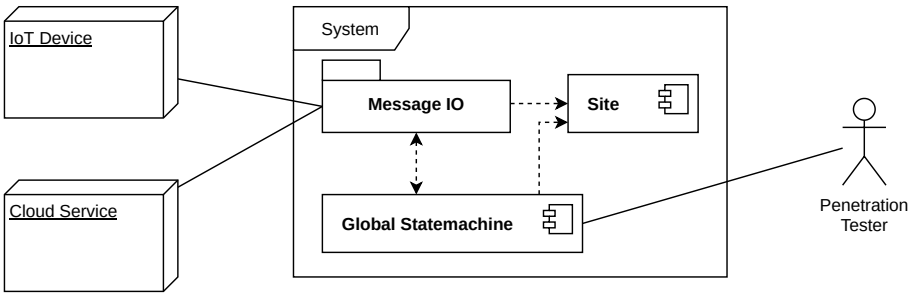


Figure 5.1.: High-level component diagram of the proxy application concept

High-level Overview As discussed in the previous chapter, the requirement "F2 Network Stacks" introduces the need for dynamically initialized objects which in this concept is implemented by making use of the abstract factory pattern in the "Site" component. This component allows for registering *Factories* that are used to initialize objects. Similar to the implementation in the first prototype, factories

initialize objects using metadata supplied from a configuration file.
Communication with other systems is encapsulated into the “Message IO”-package shown in figure 5.2. Applications that are tested by penetration testers are connected to sockets provided by the “Gateway” component and temporarily stored in a message queue to be processed by the network stacks organized by the “Global Statemachine”. Similar to the “Server” interface used in the first prototype, gateways provide means of communicating with external systems and receiving and sending messages. They are highly abstract and meant to be used for implementing interfaces for any kind of communication protocols and technologies, such as IP-based TCP and User Datagram Protocol (UDP) communication but also other protocols such as USB, Bluetooth, ZigBee or KNX. It should be noted that the static

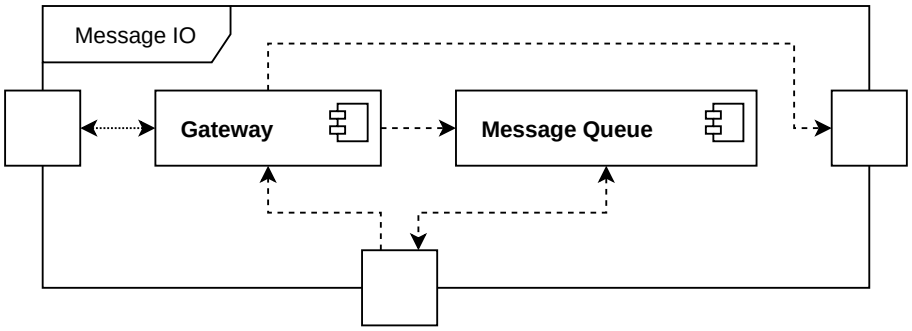


Figure 5.2.: The “Message IO”-package

view of the design is rather simple due to its dynamic runtime behaviour: many instances and relationships are only instantiated at runtime and not pre-determined. An schematic representation of the dynamic structure and interweaving of state-machines, network stacks and pipes (in this concept called a “pipeline”) is shown in figure A.5. This figure highlights a series of active state-machines and network stacks which together constitute the active pipeline.
Figure A.3 illustrates the recursive nature of this concept processing (dequeued) messages:

1. A state-machine F (initialized with the global state-machine instance) relays messages M through its active state S's network stack instance N.
2. In N, all of its pipes P process M until the end of N is reached (P does not hold a reference to a succeeding pipe instance). If F holds a reference to a succeeding FSM, F is set to this reference and the processes continues from step 1.

3. If N does not hold a reference to a nested FSM, the end of the network stack is reached and the direction of traversing the network stack is reversed.
4. P is set to N's last pipe instance and M is processed by P until the start of N is reached (i.e. P does not hold a reference to a preceding pipe instance). If F holds a reference to a preceding FSM instance, F is set to this reference, N is set to F's network stack reference and the process continues from step 4.
5. If N does not hold a reference to a preceding FSM, the beginning of the whole pipeline is reached and F is the global state-machine.

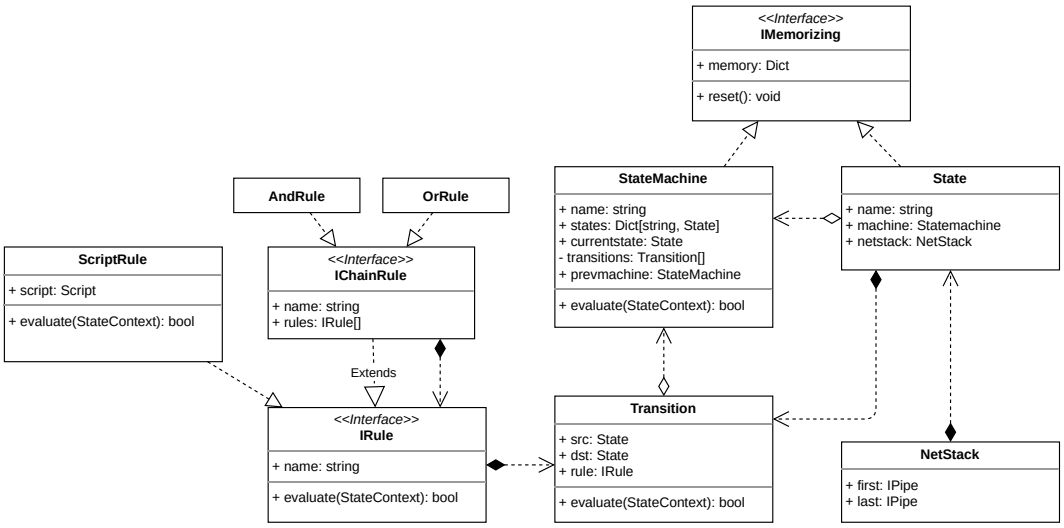


Figure 5.3.: ?

State-Machines The classes related to the state-machine component are shown in figure 5.3: *StateMachines* hold a set of *States* and *Transitions*. In order to change states, state-machines evaluate a context by checking each of their transitions for whether their conditions for transition are met or not. This context is an aggregation of the *memory* of each state-machine and their active states in the active pipeline. Transitions are defined by a source state, destination state and an *IRule* that evaluates a given context. Rules can be concatenated with logical AND or OR operators and are designed to be scripts that operate on the given context. This allows the creation of nested rules such as the following one:

$$\mathbf{changeToWS}(c) = \mathbf{AND}(\mathbf{clientUpgrade}(c), \mathbf{serverUpgrade}(c))$$

In this example, a transition with the above rule would evaluate to $true$ and trigger a state transition in a state-machine when the aggregated memory c of all state-machines and their active states of the active pipeline indicated that an HTTP request was detected that requested an upgrade to the WS protocol (for instance, $clientUpgrade$ would look for an entry $clientUpgradeRequested$ in c and evaluate its contents) and that an HTTP response was detected that confirmed the upgrade request. This would allow a state-machine to detect upgrades of HTTP communication to the WS protocol.
States hold a *NetStack* which in turn encapsulate a series of connected pipes, holding references to this series' first and last elements.

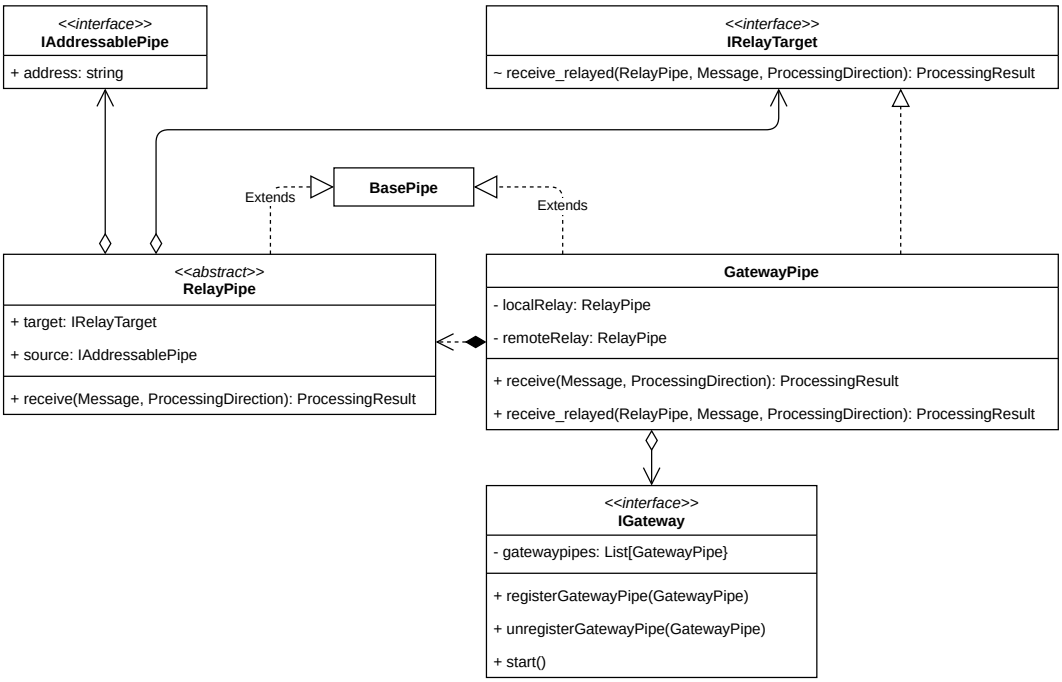


Figure 5.4.: ?

Gateway The gateway component is defined by the "IGateway" interface shown in figure 5.4. It is designed to be run as a service in a separate thread that interacts with communication interfaces on machines (i.e. bluetooth dongles or ethernet interfaces). During operation it accepts incoming connections C_I and creates its own respective outgoing connections C_O to remote servers. Pairs of connections C_I and C_O are held in technology-specific pipe implementations and encapsulated in individual "RelayPipe" instances. Those RelayPipe instances are assigned to "GatewayPipe" instances. Improving on the first prototype's design, the GatewayPipe

acts as a multiplexing pipe that accepts messages originating from the two encapsulating "RelayPipes" that act as two communication ports (e.g. the client device and the cloud server of scenario #2 described in section 4.1.1) that hold information about the address of their communication peers in their address field (e.g. an IP address of the remote peer). For instance, two TCP client sockets can be handled by two "TcpPipes" (that inherit from the RelayPipe class), allowing TCP packets to be routed into the pipeline via a GatewayPipe. Once messages are processed and sent back up the pipeline to a GatewayPipe, the GatewayPipe can find the correct RelayPipe to relay the message to by comparing their addresses with the message's "MessageDirection" information.

Pipes Building upon the approach of routing and processing messages via pipes discussed in section 4.1.3, this design concept addresses some inconsistencies of the former design and adds needed flexibility. As shown in figure A.4, the "IPipe" interface persisted and is extended by the "ITrackablePipe" interface that adds a unique identifier to pipes. This enables the application to easily locate pipes by looking up their identifiers in the "PipeDirectory", allowing to interact with and inject messages into individual pipes directly. A "BasePipe" implements the ITrackablePipe interface as well as simple routing logic for forwarding messages up and down pipelines. However, only "ProcessingPipes" actually perform any kind of operations on messages directly: they can employ "IEncoders" for (de-)serialization and "IProcessors" for transformation of messages. Contrary to the design concept of the first prototype, IEncoders need to specify which data formats they support as source and target encodings. This allows the implementation of multiple IEncoders for the same protocol that work with different source or target data formats. For example, some IEncoder may only provide decoding functionality for raw binary data into HTTP messages with raw binary bodies while another implementation provides functionality to encode strings into HTTP message bodies. In the first prototype's design, the very concept of filters was only vaguely described and lacked a clear and concise interface. This issue is resolved in this next iteration of the design concept:

- Filters are renamed to "IProcessors" (conveying the purpose and meaning of the interface in its name)
- IProcessors specify a "ProcessingDirection" that determines whether messages shall be processed on their way *down* or *up* a pipeline or in any direction, effectively granting control over applying transformations on messages.

This can be helpful when transformations shall only be applied in one direction or maybe only once in a pipeline, like replacing the contents of the body of an HTTP message.

- An IProcessor can apply logic to messages in its *process* method that also receives the pipeline's context. The returned "ProcessingResult" indicates success or failure of the operation or whether the IProcessor requests dropping a message or sending it back up the pipeline.

While there are many opportunities for specific implementations of the IProcessor interface, one general implementation is envisioned by the design concept: a simple "ScriptProcessor" allows penetration testers to supply scripts that are executed at runtime and allow transformation of messages. This directly fulfils the requirement ""F5 Scripting"".

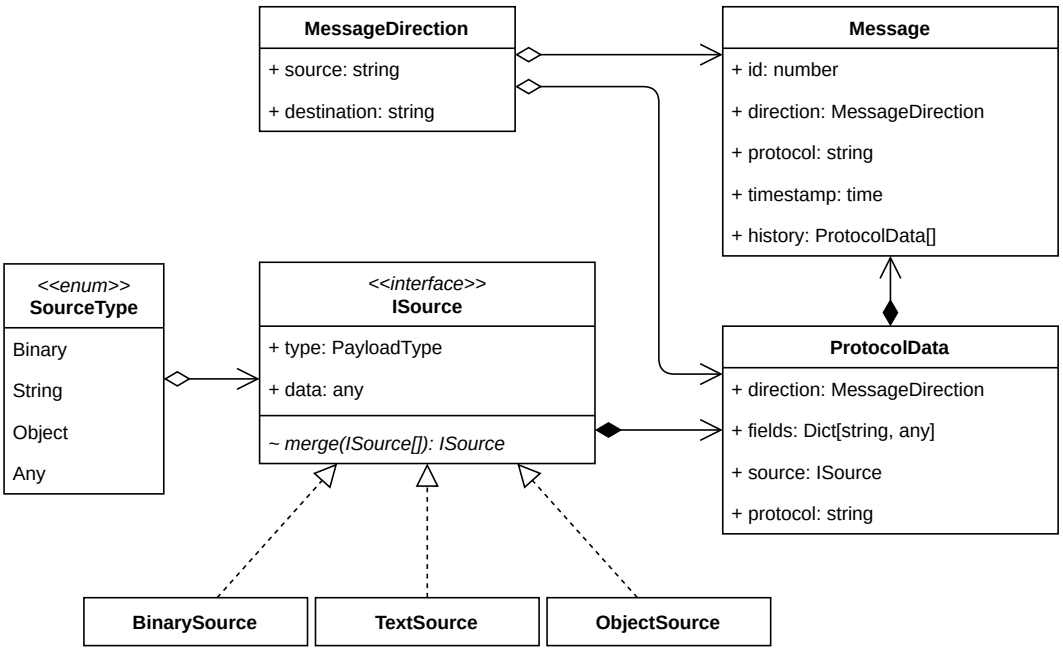


Figure 5.5.: ?

Messages Compared to the first prototype's design, the data-structures that represent messages are mostly unchanged (as shown in figure 5.5). However, during implementation and testing of the first prototype it became apparent that in some cases "historical" information about messages was required. This iteration of the design adds a list of "ProtocolData" instances to messages that specify information about the protocol, message headers and the payload ("ISource"). IProcessors and

IEncoders append newly transformed or (de-)serialized ProtocolData instances to messages. So over time, a message contains records of all those operations performed on it. This information can be useful in a number of cases like serialization: when a message is deserialized (e.g. the payload of a WS message is extracted) on its way down a pipeline, important information about the formerly encapsulating protocol is lost (such as the WS frame's flags). When a message is serialized on its way back up a pipeline, an IEncoder would have to generate this information or try and deduce it from the message, which is not always possible. However, since it can access the messages' history and former ProtocolData, it can read the original information and use this for serialization.

5.2. Design #2: Distributed Proxy Services

The design shown in the previous section was an iteration of the design worked out for the first prototype in section 4.1.3 and addressed some fundamental, architectural flaws and aimed for better flexibility and more meaningful interface definitions. However, it did not address other problems that were encountered during the implementation of the first prototype: constraints in platform, framework and programming language compatibility and flexibility. As a consequence of these constraints, the proxy application needed to be developed as a monolithic application. Therefore, each extension, like additional IEncoders that added support for new protocols, was required to be implemented in the same programming language and run on the same platform and as part of the same process as the proxy application. This also effectively limited the available selection of libraries. Another potential problem of the former design concept was the tight coupling of pipes and the deeply nested structure and hierarchy of state-machines, pipes and network stacks. While this architecture allowed to implement routing messages through composition (*by design*), it greatly added to the runtime complexity and made debugging the application significantly harder.

Overview Another iteration of the design (shown in figure 5.6) was made to address these issues. While the "Message IO" and "Site" components are left unchanged, the global state-machine is replaced by the "Registry" and "PipelineRepository" components that allow for de-centralized and more controlled processing of messages.

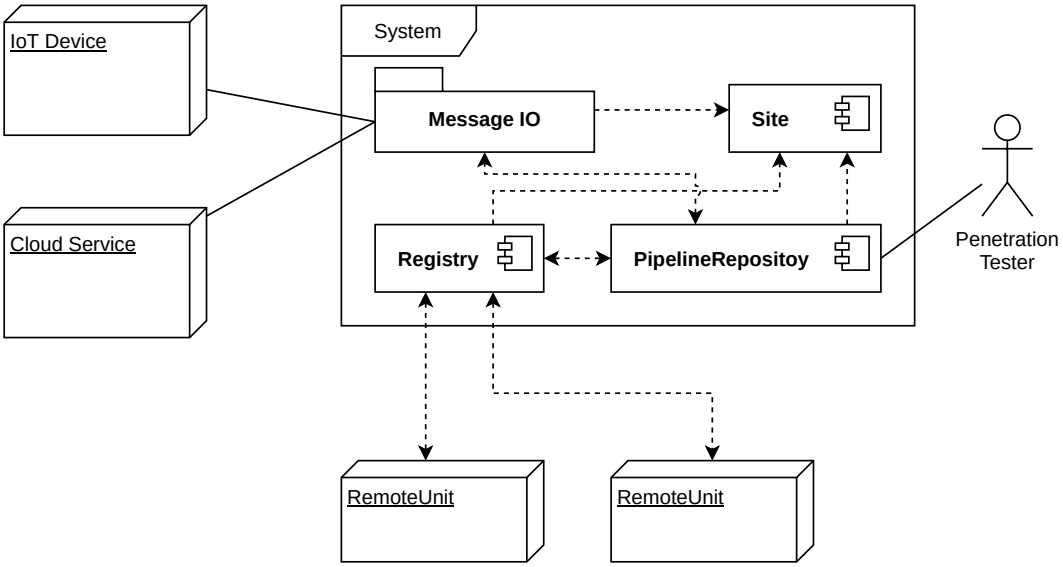


Figure 5.6.: ?

Registry and Units To implement de-centralized processing of messages, a central registry is required to register remote processing units (shown in figure 5.7). The central registry is represented by the "IRegistry" interface that allows remote units to register themselves and allows the PipelineRepository to request sessions to units that implement requested features. Remote units can be remote machines that implement the "IPort" interface that provides a list of "IUnit" instances. An IUnit implements one or more "Features", such as specific (de-)serialization or other processing, and effectively provides IEncoder and IProcessor functionalities. This transforms formerly direct calls to IEncoders and IProcessors to remote procedure calls (RPCs). Since IEncoders and IProcessors can be stateful, IUnits initialize them in "ISessions" for each requested feature. The IRegistry and IPort interfaces are explicitly kept rather simple and unspecific to the exact means of communication between them so that they can be implemented in various ways, making use of various inter-process communication (IPC) techniques.
Also, to transmit data between the proxy application and its remote units, this data needs to be (de-)serialized and a format for serialization has to be chosen.

PipelineRepository The PipelineRepository component holds information about all configured pipelines (that is a flattened representation of the hierarchically configured state-machines and network stacks) and their contexts. This allows to remove the pipes from the software architecture, providing better traceability of mes-

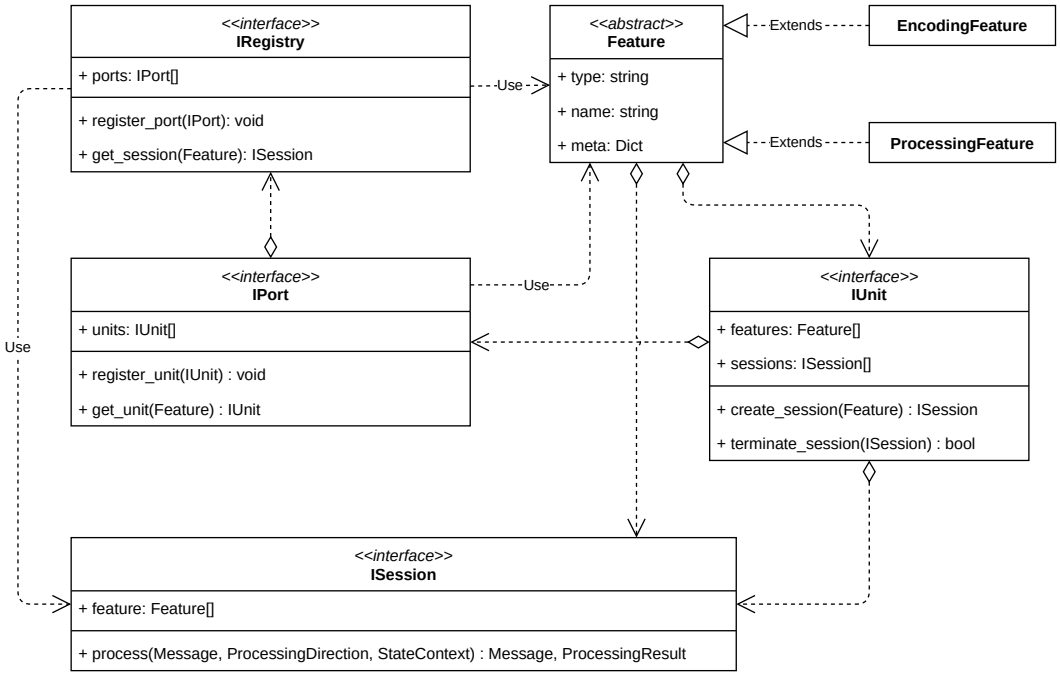


Figure 5.7.: ?

sages throughout the system and makes debugging the high-level application logic more accessible. Also, organizing network stacks and state-machines in one central place encourages creation of means to interface with these mechanisms such as REST-application programming interfaces (APIs) that let penetration testers inspect the message queue and ongoing processes.

State of the Design Concept This design concept promises to solve severe issues of the previous design iteration shown in section 5.1 and already defines some very high-level components. However, due to time constraints some components' designs were not finished and require further work on specifics. For this, certain questions need to be answered and translated into the design:

- **PipelineRepository:** How exactly is the hierarchy of state-machines and network stacks flattened? How is this flattened hierarchy represented in data-structures? How are instances of individual state-machines and network stacks initialized and organized for individual gateway-connections?
- **ISession:** How is information relevant to IEncoder and IProcessor instances (such as ScriptProcessors' Script instances) passed to remote units?

- **IRegistry/IPort:** How exactly are RPCs performed? Are there mature and appropriate frameworks that provide RPCs implementations?

5.3. Comparison of Both Designs

Both design concepts discussed in the previous sections, the monolithic and the distributed concept, promise to solve specific problems. The following paragraphs compare both concepts on the base of a set of core design aspects:

Software architecture The monolithic concept suggests a centralized and self-contained design that combines the high-level business logic of a proxy-application with low-level tasks such as (de-)serialization of various protocols. It is designed to be run on a single machine.
In contrast to this, the distributed concept separates the high-level business logic (like routing messages) and low-level tasks as part of a client-server model: the high-level logic is implemented in the central proxy server while low-level tasks are isolated into separate remote services. These services can either be run on the same machine the proxy server runs on or on external machines. Through dynamic creation and registration of remote service instances, this concept also implements scalability. Since there is no restrictions to the programming languages, platforms or frameworks used by remote services, the concept also embraces platform compatibility.

Complexity The reliance on deeply nested data-structures has a high impact on the complexity of the monolithic concept at runtime. This makes debugging an implementation of this concept significantly harder and more time-consuming. However, adding new extensions to this concept becomes a comparatively easier task as integration of such new extensions only takes place on a source-code level.
As opposed to this, the distributed approach simplifies the high-level tasks such as routing messages by introducing components that allow traceability and thus establish transparency. The offloading of protocol implementations into logical units that are accessed via IPC however contribute to a more challenging deployment of the application. Also, due to their distributed nature, debugging these remote units can

introduce further problems (e.g. connection losses and high latencies) and requires a more sophisticated and complex testing environment than the monolithic concept.

Maturity Some of the core components of the monolithic concept were already tested and proven by the first prototype. For instance, linked pipes proved to be an effective means to route and process messages up and down a processing pipeline. However, due to time-constraints, other core concepts such as the state-machines could not be tested. Regarding completeness, it is noteworthy that this concept's interfaces are well-defined.
Contrary to this, the distributed concept is not finished and requires further work to clear up a number of essential questions before it can be completed. Also, the previously proven effective idea of using pipes for routing and processing is removed in the distributed approach. This denies the approach any effectiveness acquired through previous design iterations.

Chapter 6

Implementing the Modular Proxy Application

This chapter covers an exemplaric implementation of the concept that was worked out in chapter 5, starting with formally describing the goals and constraints of this implementation in section 6.1. Afterwards, an overview and comparison of available and suitable tools for the task is performed in section 6.2. The chapter concludes with details about the implementation of individual components in section 6.3, describing how specific challenges were overcome.

6.1. Goals and Constraints

The goal of this thesis' implementation was to implement the "Monolithic Proxy Application" design concept described in section 5.1 to a maturity level that allowed to test its usefulness and effectiveness in the testbed described in section 4.1.4 that aimed to represent scenario #2 discussed in section 4.1.1. Thus, a focus was set on implementing a vertical prototype that featured important core components (such as factories, state-machines and network stacks) and a set of exemplaric protocol implementations (HTTP, WS and MQTT). Similar to the first prototype discussed in section 4.1, this prototype was a proof-of-concept implementation and neglected quality attributes such as usability and performance.
The prototype had the working title "net-riot", which indicated that this was a networking tool and was to be used in the IoT context.

6.2. Tool Selection

To choose the tools for implementing the design concept, a list of requirements to tools was inferred from the software requirements discussed and expert interviews shown in chapter 4:

F1 Scripting: The tool must provide scripting capabilities that allow penetration testers to execute complex scripted operations on messages.

F2 Libraries: In order to avoid custom implementation of the HTTP, WS and MQTT protocols, the tool must provide a rich set of libraries that can be used to work with said protocols.

F3 Deployment: To allow the prototype to be installed in an uncomplicated way, the tool must provide or support mechanisms that simplify deployment, such as code compilation and static linking of dependencies or containerization.

F4 Accessibility: The tool must be powerful and complex enough to solve the software requirements and implement the design concept, but it must also feature a "barrier or entry" that is low enough so extending the application is feasible for open source developers.

It was found that Python satisfied all of these requirements:

- Python is a free, open source and general purpose programming language that was first released in 1991 and is being continuously improved and updated.
- The built-in *exec*[1] function provides execution of arbitrary Python code at runtime. Although it is infamous for its security implications, it is very suitable for scripting.
- The Python Package Index (PyPI) is a public repository of more than 300.000[2] Python packages that can be installed using the *pip* command-line tool. There are numerous packages that implement the protocols WS, MQTT and HTTP.
- Python supports multiple ways to deploy projects, including packaging projects into executable files [3] and containerization[4].

[1] https://docs.python.org/3/library/functions.html#exec
[2] Based on PyPI's statistics: https://pypi.org/
[3] https://packaging.python.org/overview/#bringing-your-own-python-executable
[4] e.g. using Docker https://hub.docker.com/_/python/

- Its comparatively simple syntax and its design philosophy that values accessible code higher than performance[5] encourage readability and maintainability in Python projects. When implemented, this makes Python an accessible programming language. Also, its optional static typing allows to omit redundant type information for simple methods and to provide explicit type information for complex and shared pieces of code like algorithms and interfaces.

Git was used for version-control and Microsoft Visual Studio Code was the integrated development environment (IDE) used for implementation.

6.3. Individual Components

The following sections discuss especially challenging aspects of net-riot's implementation, which problems were encountered and how they were solved.

6.3.1. Gateways

The protocols used in scenario #2 (HTTP, WS and MQTT) are used on top of TCP. Therefore, net-riot implemented a TCP-Gateway that allowed it to create TCP server sockets to listen on for incoming connection requests. For incoming connections C_I, respective outgoing connections C_O were initialized and connected to a preconfigured remote server. For each of those connections, "TcpPipe" instances P_I and P_O were initialized that ran in separate threads and accepted incoming packets. The gateway then initialized "TcpGatewayPipe" instances with P_I and P_O that routed messages originating from the TcpPipes into the pipeline and from the pipeline to the correct TcpPipe instance.
Messages that originated from gateways were temporarily stored in a queue so that only a single message was processed in the pipeline at any given time. It was found that if multiple messages were processed simultaneously, the global state-machine could change states while a message was still being processed in a then inactive state, resulting in this state's network stack being left unconnected and unable to route the message back up the pipeline. For the same reason, net-riot only supported one single connection.

[5] Described in 19 aphorisms in "The Zen of Python" (https://www.python.org/dev/peps/pep-0020/)

6.3.2. Encoders

For each of the protocols used in scenario #2, net-riot implemented a separate IEncoder.

HttpEncoder For HTTP, no library was found that parsed HTTP requests or responses into low-level representations that did not discard essential information. However, since HTTP is a comparatively simple, text-based and stateless protocol, a custom IEncoder was implemented that parsed HTTP requests and responses from raw binary data and allowed to assemble requests and responses from processed messages. For assembly, the implementation used the HTTP header information contained in a message's "history" field that held the headers that were parsed when the message was first parsed by this encoder. One practical pitfall of the custom implementation was the "Content-Length" HTTP header that indicated the number of bytes contained in an HTTP request body or HTTP response body. Systems that parse HTTP messages (such as web-servers and browsers) use the value of the "Content-Length" header to read the indicated number of bytes from a TCP stream and associate it with the parsed headers. If the length of a message body is modified by a proxy, the "Content-Length" header indicates the wrong number of bytes to read: this will either result in reading too few bytes (thus, discarding information) or reading too many bytes rendering future requests malformed. To solve this issue, the custom encoder provided the configurable flag *recalculateContentLength* that, if set to true, dynamically calculated the value of the "Content-Length" header to reflect the actual length of bytes contained in the message's BinarySource.

WsEncoder The library used for WS implementation was "websockets"[6]. It offered methods for parsing (*framing.Frame.read*) and assembling (*framing.Frame* constructor) WS frames. These functions worked fine for regular WS frames. However, to save bandwidth, some WS client and server implementations make use of Per-Message Compression Extension (PMCE). The use of PMCE is indicated by the "rsv1" bit of WS headers being set to true. While the library implemented PMCE (*extensions.permessage_deflate.PerMessageDeflate*) and using the extension worked on the first messages of a WS connection, it would fail to correctly compress messages after it processed a number of messages, causing the remote WS clients and servers to terminate the connection. It was found that the PMCE im-

[6] https://github.com/aaugustin/websockets/, commit 6b5cbaf41cdbc9a2074e357ccc613ef25517dd32

plementation was stateful and not reset after being used. This issue was solved by initializing new *PerMessageDeflate* instances per WS frame assembly.

MqttEncoder For (de-)serialization of MQTT messages, the library "hbmqtt"[7] was used in net-riot. The library implemented individual classes for each MQTT message type and provided methods for parsing (*from_stream*) and assembly (*to_bytes*) of packets. However, the library did not provide any method that chose the correct class for (de-)serialization of a given binary buffer or processed message. Therefore, this functionality was implemented in net-riot: the *MqttEncoder* defined a dictionary that mapped MQTT message types to a tuple of classes (*Packet*, *VariableHeader* and *Payload*) provided by the library and attempted to parse the basic MQTT message header of a binary buffer. This header included the MQTT message type that was then used to look up the correct classes for parsing in the dictionary. When MQTT messages were re-assembled by the *MqttEncoder*, it extracted the message type from the parsed header (that was stored in the message's "history" field) and used this to look up the correct classes for serialization in the dictionary.

6.3.3. Scripting

In order to allow execution of scripts on messages, net-riot implemented the "ScriptProcessor" discussed in section 5.1. This IProcessor implementation used Python's built-in *exec* function for executing arbitrary Python code (shown in listing 6.1). The function allows the caller to define the available objects of the local and global scopes in the called code. This mechanism was used to pass the message instance, processing direction, context and "result" object to the script in the local scope. The *ProcessingResult* values were passed to the script in the global scope which allowed to script to assign one of these values to the "result" object in the local scope. After execution, the "result" value was evaluated and returned by the ScriptProcessor. This effectively enabled scripts to:

- Read and write fields and payloads of messages.
- Read and write the active pipeline's context and thus, its state-machines' memory. This could be used to trigger state transitions.

[7] https://github.com/beerfactory/hbmqtt/, commit 31165fb0e827925417f99a7b1f475a9d67e1c72f

- Control whether messages were processed further, being immediately sent back up the pipeline, being dropped or ignored.

```
class ScriptProcessor(IProcessor):
    def __init__(self, script: Script, inDirection: ProcessingDirection =
        ProcessingDirection.ANY):
        super().__init__(inDirection)
        self._script_ = script

    def process(self, msg: Message, direction: ProcessingDirection, context:
        StateContext) -> Tuple[Message, ProcessingResult]:
        _locals = {'msg': msg, 'direction': direction,
                   'res': ProcessingResult.SUCCESS,
                   'context': context}
        _globals = {
            'SUCCESS': ProcessingResult.SUCCESS,
            'ERROR': ProcessingResult.ERROR,
            'INCOMPLETE': ProcessingResult.INCOMPLETE,
            'DROP': ProcessingResult.DROP,
            'BACK': ProcessingResult.BACK,
        }
        try:
            exec(self._script_.body, _globals, _locals)
        except Exception as e:
            print(f'Script error: {str(e)}')
            return (_locals['msg'], ProcessingResult.ERROR)
        return (_locals['msg'], _locals['res'])
```

Listing 6.1: The ScriptProcessor implementation of net-riot using Python's built-in exec function.

For net-riot to support MQTT communication that was tunnelled via WS it had to detect upgrades of HTTP connections to WS. This was achieved by ScriptProcessor instances that executed scripts that examined the HTTP communication and set values in the memory of specific state-machines using the context that was supplied to them. Listing 6.2 shows the script that detected a server's HTTP response to an upgrade to WS: if there was an "Upgrade" header with the value "websockets" and if the status code of the response was 101, the key "serverUpgrade" of the nested state-machine "http" was set to $True$. If any of these checks failed, it was set to $False$. This script worked in conjunction with another script that performed similar checks to detect a client's request to upgrade the connection. The "http" state-machine used "ScriptRules" that evaluated the value of the "serverUpgrade" key and triggered a state transition if both a client's upgrade request and the server's upgrade response were detected.

```
def process(msg, context):
    _upgrade = msg.latest_data.fields.get('Upgrade')
    _status = msg.latest_data.fields.get('::Status')
    _upgradeToWs = _upgrade.lower() == 'websocket' if _upgrade else False
    _status101 = _status.find('101') != -1 if _status else False
```

```
    context.memory['http_to_ws']['http']['serverUpgrade'] = _upgradeToWs and
        _status101
    return SUCCESS


res = process(msg, context)
```

Listing 6.2: The script net-riot used to detect upgrades of HTTP connections to WS.

6.3.4. Configuration Parsing and Building

The requirement "F2 Network Stacks" mandated the capability to read a configuration file at runtime and initialize the therein specified hierarchy of state-machines and network stacks.

JSON Configuration and Schemas For formal specification and representation of state-machines and network configurations, net-riot made use of the JSON format. In over 600 lines of code, a JSON schema was defined that specified the structure and types of net-riot configuration files. JSON and JSON schema were chosen over alternatives such as XML and YAML due to their simple syntax, flexibility and powerful features (such as $definitions$).

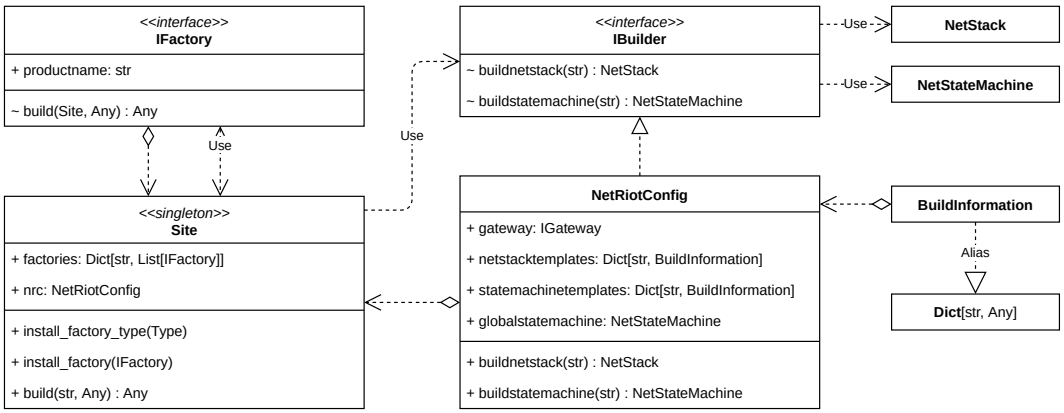


Figure 6.1.: ?

Factories, Builders and Templates The "Site" component was briefly discussed in section 5.1 and envisioned as an implementation of the abstract factory pattern. Figure 6.1 shows net-riot's implementation of the Site component and its associated interfaces and classes. The Site component holds a dictionary that maps product-

names to factories. In turn, factories can instantiate objects such as pipes, gateways and state-machines (each differentiated by distinct names). Listing 6.3 shows a simple factory implementation that is used to instantiate "ScriptRule" objects. However, prior to calling the constructor of the ScriptRule class, it needs to acquire a "Script" instance. It does so by requesting the Site to build a Script instance with the supplied information. In turn, the Site looks up the correct factory to use for instantiating the requested Script instance and calls it. Thus, factories can recursively request further objects to be built by the Site while they themselves provide the functionality to produce a single product each.

```
class ScriptRuleFactory(IFactory):
    def __init__(self) -> None:
        super().__init__(RULE_SCRIPT)

    def build(self, plant: Site, obj: Dict[str, Any]) -> Any:
        _scriptobj = obj['config']
        _name = obj['name']
        _script = plant.build(SCRIPT, _scriptobj)
        return ScriptRule(_name, _script)
```

Listing 6.3: A simple factory that requested the build of a "Script" instance to instantiate a "ScriptRule".

The "NetRiotConfig" class implements further logic to net-riot that allows to dynamically instantiate whole state-machines and network stacks. It does so by saving the configured state-machines and network stacks as *templates* and passing them to the Site to build. For instance, net-riot's configuration file may specify a state-machine by the name of "http_to_ws" that is specified to be used as a nested FSM of a state "entry" of the global state-machine. When the global state-machine is built and an instance of the nested FSM is requested, the Site component calls NetRiotConfig's *buildstatemachine* method and supplies the name "http_to_ws". In turn, NetRiotConfig calls the Site component's *build* function and supplies the template of the requested state-machine.
Also, net-riot used Python's "inspect" package to dynamically acquire all types of implemented factories and register them in the Site instance (shown in listing 6.4).

```
import inspect
import parsing
from py_linq import Enumerable

factory_types = Enumerable([m for _, m in inspect.getmembers(parsing)])\
    .where(lambda m: inspect.isclass(m) and not inspect.isabstract(m) and parsing.
        factory.IFactory in inspect.getmro(m))\
    .to_list()

for factory_type in factory_types:
```

```
parsing.SITE.install_factory_type(factory_type)
```

Listing 6.4: Dynamic registering of all factory implementations.

Chapter 7

Postmortem Documentation

This chapter attempts to identify and spell out the causes of the project failure. The project timeline will allow a quantitative overview of the project progression and show what parts of the project slowed down progress. Then, an overview of the qualitative aspects of the deliverables will discuss the maturity of the implementation and which parts reached a satisfactory level.

7.1. What Constitutes the Failure

TBD: What failed? Why was the implementation not successful?

7.2. Quantitative Overview: Time Management

Comparing the planned thesis schedule to the actual course it has taken, this section discusses how the intended plan was implemented and changed at certain places. Also, it will examine the causes of the delays during development.

7.2.1. Project Timeline

Table 7.1 shows the initially planned thesis schedule divided into four phases, laying out the course of the thesis over a span of 24 weeks.

Phase / Task	Duration
1. Preparation	4 weeks (16, 66%)
Literature Research	1 week
Expert Interviews	1 week
Testbed Configuration	2 weeks
2. Prototype	7 weeks (29, 16%)
Prototype Conception	2 weeks
Prototype Implementation	4 weeks
Expert Feedback	1 week
3. Release Candidate	7 weeks (29, 16%)
RC Conception	2 weeks
RC Implementation	4 weeks
Expert Feedback	1 week
4. Finalization	6 weeks (25%)
MQTT Case Study	2 weeks
Thesis Finalization	4 weeks
Total	24 *weeks*

Table 7.1.: Initially planned schedule for the thesis

1. Preparation The initial phase covered preparation tasks for further work on the thesis. Literature research on the topics covered and touched in this thesis was carried out. Related work on IoT and ICS security analysis (as discussed in chapter 2) was of special interest as those showed what approaches had been taken to assess security implementations. Also, a testbed (discussed in section 4.1.4) for running the proxy application was built. A decision was made against conducting expert interviews before implementing a first prototype on the assumption that practical experience with the subject matter would benefit the expert interviews. The fact that a number of important questions arose from work on the first prototype later proved this decision to be correct. Performing the literature research and building a testbed was completed within the intended schedule of three weeks.

2. Prototype In the second phase, the prototype discussed in section 4.1 was designed and implemented in weekly sprints. Preceding these sprints, a rough design of the prototype's architecture and runtime behaviour was worked out in one week that would serve as a base for further design refinement and implementation in the sprints. These sprints ran for eight weeks in total: the initial design turned out to be too oversimplified so that sprints aiming to design and implement specific components were conducted rather isolated from other components that still needed to be worked on. As a result, both the integration of individual components and their interaction would fail and require redesigns and time-consuming adjustments to their implementation. Also, neither was the prototype mature enough to be used as a proxy application, nor was the resulting design and implementation clean enough to suggest putting further effort into working on them. After these eight sprints, work on this prototype was stopped and the expert interviews discussed in section 4.2 were prepared and conducted.

3. Release Candidate The third phase was intended to yield a fully functional proxy application. This was initiated by switching the technology stack from TypeScript to Python and re-designing and re-implementing large parts of the first prototype. In order to avoid the same mistake of refining a vague design concept and spending time adjusting the design and implementation to make them work, two weeks were spent on a new design concept shown in section **??**. This concept did not only define single components (discussed in section 6.3) but also interfaces that specified how those components interacted with each other, aiming for clear sep-

Phase / Task	Duration
1. Preparation	3 weeks (12%)
Literature Research	1 week
Testbed Configuration	2 weeks
2. TypeScript Prototype	10 weeks (40%)
Prototype Conception	1 week
Prototype Implementation	8 weeks
Expert Interviews	1 week
3. Python Candidate	12 weeks (48%)
RC Conception	2 weeks
RC Implementation	10 weeks
Total	*25 weeks*

Table 7.2.: Actual schedule of the project

aration of components and high flexibility in implementation. Components of the prototype that were independent of the communication protocols used at runtime, such as NetStacks and FSMs, were implemented first over the span of four weeks. Then, implementations for supporting the HTTP, WS and MQTT protocols followed over a span of another six weeks. Work on this prototype was stopped after those ten weeks as the technical difficulties discussed in section 7.2.2 made estimations over the remaining time needed to finish the prototype both hard to make and rather unreliable.

4. Finalization The final phase was intended to conduct a case study on how the proxy application would perform on scenario # 2 from section 4.1.1. Tests were made to run the proxy application in the testbed shown in section 4.1.4 which featured the same communication protocols that were used in scenario # 2. However, the proxy application failed to reliably transmit or encode the messages sent between the MQTT client and broker, thus resulting in a broken communication channel. The complex runtime behaviour and very time-consuming debugging of the proxy application (further elaborated on in section 7.2.2) lead to the decision to stop the project.

Table 7.2 shows the actual schedule of the thesis. As can be seen, 88% (22 weeks) of the time working on the thesis was spent designing and implementing the prototypes compared to a planned portion of roughly 60% (14 weeks).

7.2.2. Development Challenges

There was a series of development challenges that slowed down implementation of both prototypes considerably:

Complex runtime behaviour The combination of nested FSMs and pipelines lead to several problems during development. Even comparatively simple scenarios to use the proxy application in required a complete configuration file made of a global state machine and at least one netstack. This lead to a dynamic and long chain of references at runtime that made tracing back calls and attributing them to specific instances difficult.
Some problems such as a timing problem in the implementation of FSMs were very time consuming to debug: an FSM would change its state when any of its rules was evaluated successfully and indicated a state change. By design, all FSMs of an active netstack would evaluate their rules when a message entered or left any netstack. When a higher-level FSM (e.g. the global state-machine) changed its state *while* a message was still being processed in a lower-level FSM, the higher-level state-machine would change to another netstack, thus disconnect the lower-level state-machine. Eventually, the message would be processed back up and run into a pipe that had no upstream connection anymore, raising an exception and terminating the program. This particular error was discovered during the implementation and testing of the MQTT encoder, in a runtime setup that involved a global default state-machine with a default TCP netstack and a state-machine that handled HTTP to WS upgrades and processed MQTT messages utilizing network stacks for HTTP and WS/MQTT.
Other problems uncovered design flaws and required prompt changes to the software design or, in some cases, introduced new constraints to the project. One such example was discovered while testing the HTTP encoder implementation using Mozilla FireFox as an HTTP client. When browsing websites, the browser would open multiple connections to the target host to acquire multiple files at the

same time[1]. This required the proxy application to instantiate a new pipeline per incoming connection rather than reside on using a single pipeline. Also, this broke the design intention of pipes being connected to at most one preceding and one succeeding pipe as at some point, the pipelines needed to connect back to the global state-machine. However, when multiple pipelines connected back to a single FSM and the only objects pipes would connect to were other pipes, a multiplexing pipe needed to be implemented. This specific case required to make a decision for the proxy application to support multiple simultaneous connections or enforce the use of only one single connection. For a lab environment, enforcing the use of a single connection might work, however in real scenarios this constraint could potentially lead to the proxy application breaking applications at runtime. The decision was made to change the software design in a way that would allow the proxy to handle multiple connections, however the prototype would only support a single connection at a time.

Open source libraries Both prototypes made use of open source libraries that implemented various protocols and included serialization and de-serialization routines for handling protocol specific packets. However, such libraries appeared to be intended to be used for developing applications that used those protocols as a means for transporting data rather than directly parsing packets.
Usually, these libraries would offer an API that allowed to instantiate and operate clients and servers and bind callbacks to events. The implementations of packet serialization and de-serialization were often times hidden through encapsulation, missing typings or poorly documented. For instance, the JavaScript library "ws"[2] provided methods for serialization and de-serialization but lacked typings . Typings for this library were made available by the project "DefinitelyTyped", however those did not include the classes relevant for serialization and de-serialization ("Sender" and "Receiver")[3] . At the time of implementing the Python prototype, it used the library "websockets" that offered only an async de-serialization method ("framing.Frame.read"), requiring the use of asyncio which was circumvented by implementing a wrapper around it.
The Python prototype also used the "hbmqtt" library to (de-)serialize MQTT messages. The library used an object-oriented implementation for (de-)serializing MQTT

[1] For testing single HTTP connections, the key *network.http.max-connections-per-server* could be set to 1 in the *about:config* page.
[2] https://github.com/websockets/ws, version 7.0.0, commit 092a822a41eb22f6d6745c18bc29b9c40715680f
[3] https://github.com/DefinitelyTyped/DefinitelyTyped/, commit 4bf23527293b2943c7fc12585c21473905a564d7

messages where a class for each MQTT message type (e.g. *CONNECT*, *CONNACK*...) inherited from an abstract "MQTTPacket" superclass that defined a "to_bytes" method for serialization and an async "from_stream" method for de-serialization. Since the library did not implement a generic method that parsed a byte-buffer and returned the appropriate MQTT message object, this logic had to be implemented as part of the work in the prototype, requiring investigation of the (largely uncommented) source code of the library as its documentation did not cover these internal (de-)serialization methods but focused on high-level use of the API it implemented. From a software engineering point of view, omitting public interfaces to internal (de-)serialization methods and forcing specific programming patterns (such as async programming) are perfectly valid decisions in the context of single, individual modules. However, for those reasons, making use of the "heavy lifting" those libraries performed, was not trivial and came with workarounds and investigating the libraries' source code which in turn took up time during the implementation phases.

Then there were also instances of incomplete documentation: the Python library "websockets" implemented (de-)serialization of WS packets and also implemented the PMCEof the WS protocol. Calling the (de-)serialization methods of the "websockets" library and specifying the use of PMCE, the first incoming and outgoing messages would be compressed correctly, however following messages would be compressed incorrectly. This rendered the prototype useless as WS may use PMCE by default to reduce bandwidth. The library failed to raise exceptions or return error codes so from the prototype's runtime point of view it appeared to work just fine. After investigating the library's source code it was found that the instances implementing the extension were stateful. When supplying newly created instances of said extension implementation to the (de-)serialization methods, they worked as intended, compressing and decompressing any amount of WS packets. This could be due to a multitude of reasons including improper use of the PMCE instances or improper calling of the (de-)serialization methods. No documentation could be found about specifics on those specific topics, though.

For Python libraries, one reason why documentation was in some cases sparse, only documented high-level features and largely omitted in-code documentation (such as comments) might be the "pythonic" approach to writing Python code. "Pythonic" code values readability higher than performance and encourages writing self-explanatory code. While this way of programming may help to understand individual methods or even algorithms that use multiple methods, it does not by

itself aid in documentation of high-level concepts or complex interaction. Another reason for sparse documentation in open source libraries might be the developers' focus on implementing more features or improving the code-base instead of aiming for more complete documentation. Contrary to commercial products, there usually are no monetary incentives for developers of open source software to write documentation.

7.3. Qualitative: Deliverables

TBD: Which fit-criteria were met? What is the implementation currently capable of? Which requirements were not full-filled?

Chapter 8

Summary

This chapter provides a summary of the design concepts shown in chapter 5 and the implementation thereof in chapter 6.

8.1. Design Concepts

In chapter 5, the rough and vague design of the first prototype discussed in section 4.1.3 was further refined over the course of two iterations.
The first design concept presented in section 5.1 describes a monolithic proxy application that builds on the basic initial design which makes extensive use of the pipes and filters design pattern for message routing. At its core it sends and receives packets (*messages*) through gateways and (de-)serializes and transforms them through network stacks. It also features state-machines that allow implementation of complex logic of message (de-)serialization and transformation by allowing the binding of individual network stacks to states and triggering state transitions programmatically through the use of scripts. This results in a potentially deeply nested hierarchy of state-machines and network stacks.
The second design concept discussed in section 5.2 is an iteration of the monolithic design concept and describes a distributed proxy application that isolates (de-)serialization and transformation of messages from the internal proxy application logic. To decouple these low-level tasks from the high-level application logic of the proxy application, the concept introduces interfaces for remote units that provide access to specific features (i.e. (de-)serialization) and an interface for the central proxy application that allowed registering these remote units. Also, the nested hier-

archy of state-machines and network stacks is flattened and organized centrally in a repository in the central proxy application.
Both concepts feature distinct advantages and disadvantages and are compared in section 5.3.

8.2. Implementation

Chapter 6 shows an exemplaric implementation of the first design concept discussed in section 5.1 under the working title “net-riot”. The monolithic design concept was chosen for implementation due to its proven core ideas and its comparatively high maturity. The second example scenario presented in section 4.1.1 is used for reference because it features a nested communication stack and a corresponding testbed was already implemented in section 4.1.4. For this implementation, Python is used because of its flexibility, low barrier of entry and rich package ecosystem.
Since the reference scenario makes use of the HTTP, WS and MQTT protocols that used TCP as an underlying transport protocol, TCP gateways are implemented in net-riot as a MITM interface that external devices such as IoT devices and cloud server connect to. For HTTP (de-)serialization, net-riot implements a custom encoder while for WS and MQTT existing libraries are used.
For representation of stacked communication protocols (such as MQTT being transported via WS), network stacks and state-machines were implemented: network stacks bundle a series of connected pipes that perform operations on messages, such as (de-)serializing and manipulating messages. State-machines allow selecting which network stacks to actively use by binding them to individual states. State-machines regularly evaluate their context and check whether states should be changed dependent on their registered transitions’ rules. These ScriptRules execute scripts which can examine and manipulate the states’ and state-machines’ context information.
A central task left open for implementation by the design concept is the configuration of the proxy application for specific scenarios and the resulting dynamic instantiation and parametrisation of state-machines and network stacks. In net-riot, JSON files and schemas were used for configuration specification and validation. Also, a recursive variance of the abstract factory design pattern was implemented for dynamic instantiation of objects defined in the configuration files. Figure 8.1 shows the output of the “cloc” utility program executed on net-riot’s source code.

```
mo@ubuntu-vm:~/net-riot cloc.
      52 text files.
      52 unique files.
       3 files ignored.

github.com/AlDanial/cloc v 1.82  T=0.04 s (1200.8 files/s, 115447.7 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
Python                          42            768            196           2744
JSON                             5              0              0           1045
Markdown                         1              2              0             42
Bourne Shell                     2              0              0             10
-------------------------------------------------------------------------------
SUM:                            50            770            196           3841
-------------------------------------------------------------------------------
```

Figure 8.1.: The lines of code in net-riot calculated by the “cloc” utility program.

Chapter 9

Conclusion

TBD

9.1. Outlook

TBD: Discuss specific steps that can be taken to fully implement the concept.

- *UI based config editor*
- *Flat hierarchy!*

List of Abbreviations

A/C air conditioner
API application programming interface
ARP Address Resolution Protocol
AWS Amazon Web Services
ENISA European Union Agency for Cybersecurity
FSM finite-state machine
GDPR General Data Protection Regulation
HMI human-machine interface
HTTP Hypertext Transfer Protocol
ICS industrial control system
IDE integrated development environment
IIoT Industrial internet of things
IoT Internet of things
IP Internet Protocol
IPC inter-process communication
ISP Internet Service Provider
JSON JavaScript object notation
MITM man-in-the-middle
MQTT message queuing telemetry transport
MTU maximum transmission unit
NAT Network Address Translation
OPC U/A OPC Unified Architecture
PLC programmable logic controller
PyPI Python Package Index
PMCE Per-Message Compression Extension
QoS Quality of Service
REST Representational State Transfer
RPC remote procedure call
TCP Transmission Control Protocol
TLS Transport Layer Security
UDP User Datagram Protocol
WS WebSockets

List of Tables

List of Figures

Listings

Bibliography

[1] Noah Apthorpe, Dillon Reisman, and Nick Feamster. *A Smart Home is No Castle: Privacy Vulnerabilities of Encrypted IoT Traffic*. 2017. arXiv: 1705.06805 [cs.CR].

[2] Michael Bartsch et al. *Spionage, Sabotage und Datendiebstahl - Wirtschaftsschutz in der Industrie 2018*. Oct. 2018. URL: https://www.bitkom.org/sites/default/files/file/import/181008-Bitkom-Studie-Wirtschaftsschutz-2018-NEU.pdf.

[3] Jonah Bellemans. "The state of the market: A comparative study of IoT device security implementations". MA thesis. KU Leuven, 2020.

[4] Dan Demeter, Marco Preuss, and Yaroslav Shmelev. *IoT: a malware story*. Oct. 2019. URL: https://securelist.com/iot-a-malware-story/94451/ (visited on 02/17/2020).

[5] Jens Jäger et al. "Advanced Complexity Management Strategic Recommendations of Handling the "Industrie 4.0" Complexity for Small and Medium Enterprises". In: *Procedia CIRP*. Factories of the Future in the digital environment - Proceedings of the 49th CIRP Conference on Manufacturing Systems 57 (Jan. 2016), pp. 116–121. DOI: 10.1016/j.procir.2016.11.021.

[6] Wenquan Jin and DoHyeun Kim. "Development of Virtual Resource Based IoT Proxy for Bridging Heterogeneous Web Services in IoT Networks". In: *Sensors* 18 (May 2018), p. 1721. DOI: 10.3390/s18061721.

[7] Christian Lesjak et al. "Security in industrial IoT – quo vadis?" In: *e & i Elektrotechnik und Informationstechnik* 133.7 (Nov. 2016), pp. 324–329. DOI: 10.1007/s00502-016-0428-4. URL: https://doi.org/10.1007/s00502-016-0428-4.

[8] Santiago Hernández Ramos. *Polymorph: A Real-Time Network Packet Manipulation Framework*. Apr. 2018. URL: https://github.com/shramos/polymorph (visited on 02/17/2020).

[9] Théo Rigas. *IOXY - MQTT intercepting proxy*. July 2020. URL: https://blog.nviso.eu/2020/07/06 /introducing-ioxy-an-open-source-mqtt-intercepting-proxy/ (visited on 07/10/2020).

[10] Tilman Wittenhorst. *Ferngesteuert ins Smart Home: Saugroboter verrät Grundriss der Wohnung*. June 2019. URL: https://www.heise.de/security/meldung/Ferngesteuert-ins-Smart-Home-Saugroboter-verraet-Grundriss-der-Wohnung-4436657.html (visited on 02/17/2020).

Appendix A

Diagrams

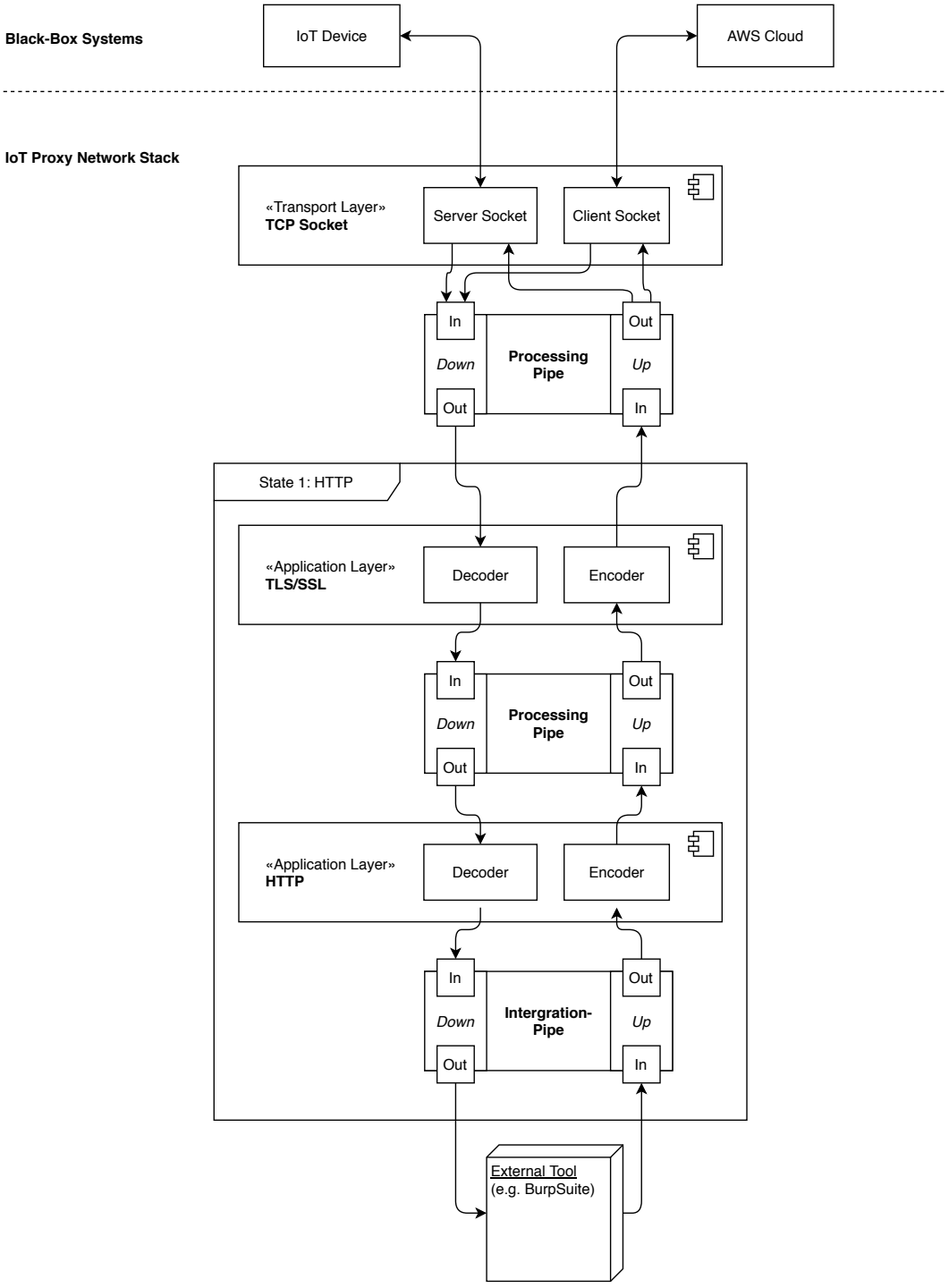


Figure A.1.: AWS IoT Scenario - State 1: HTTP Server

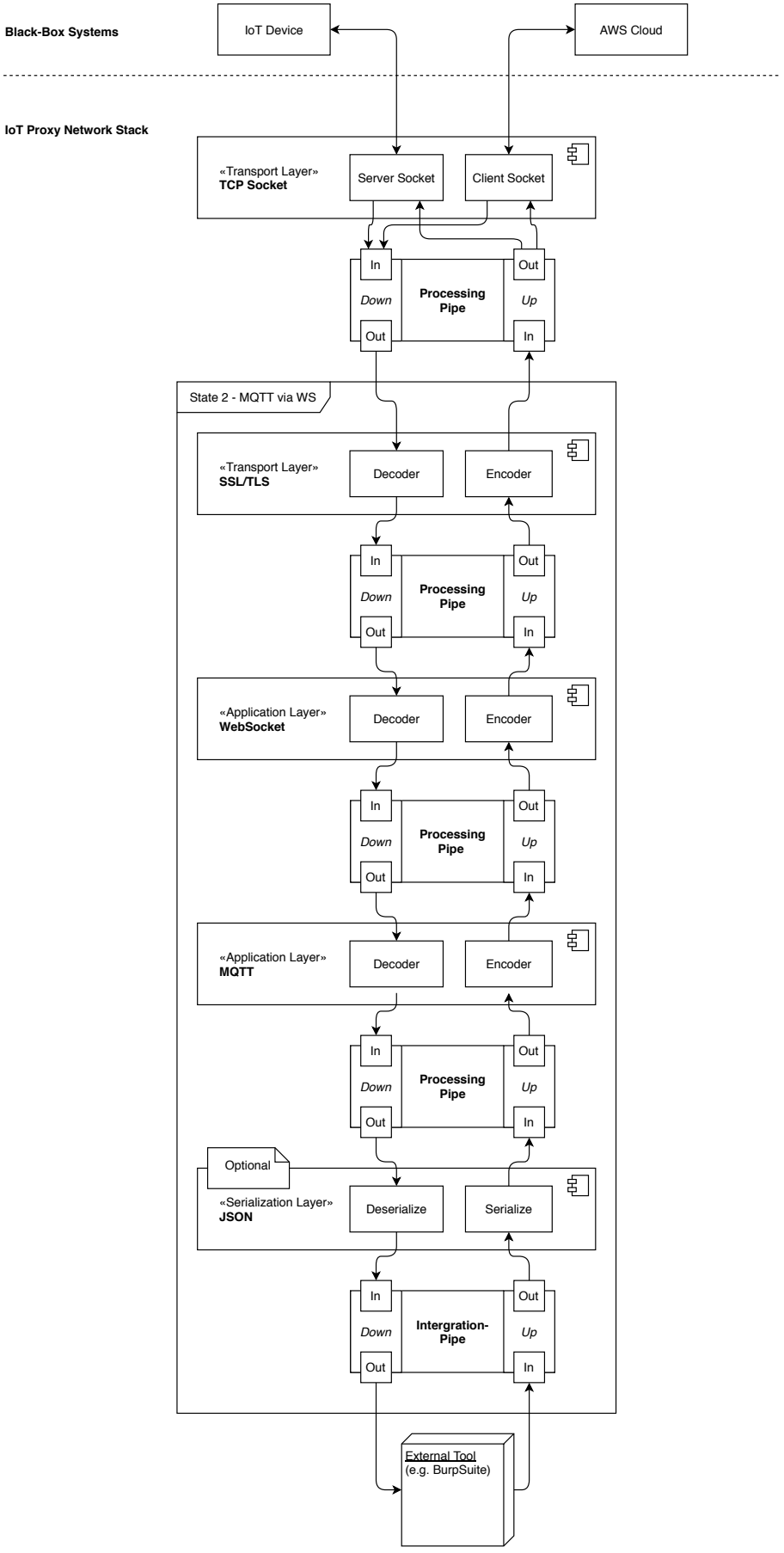


Figure A.2.: AWS IoT Scenario - State 2: MQTT via WS

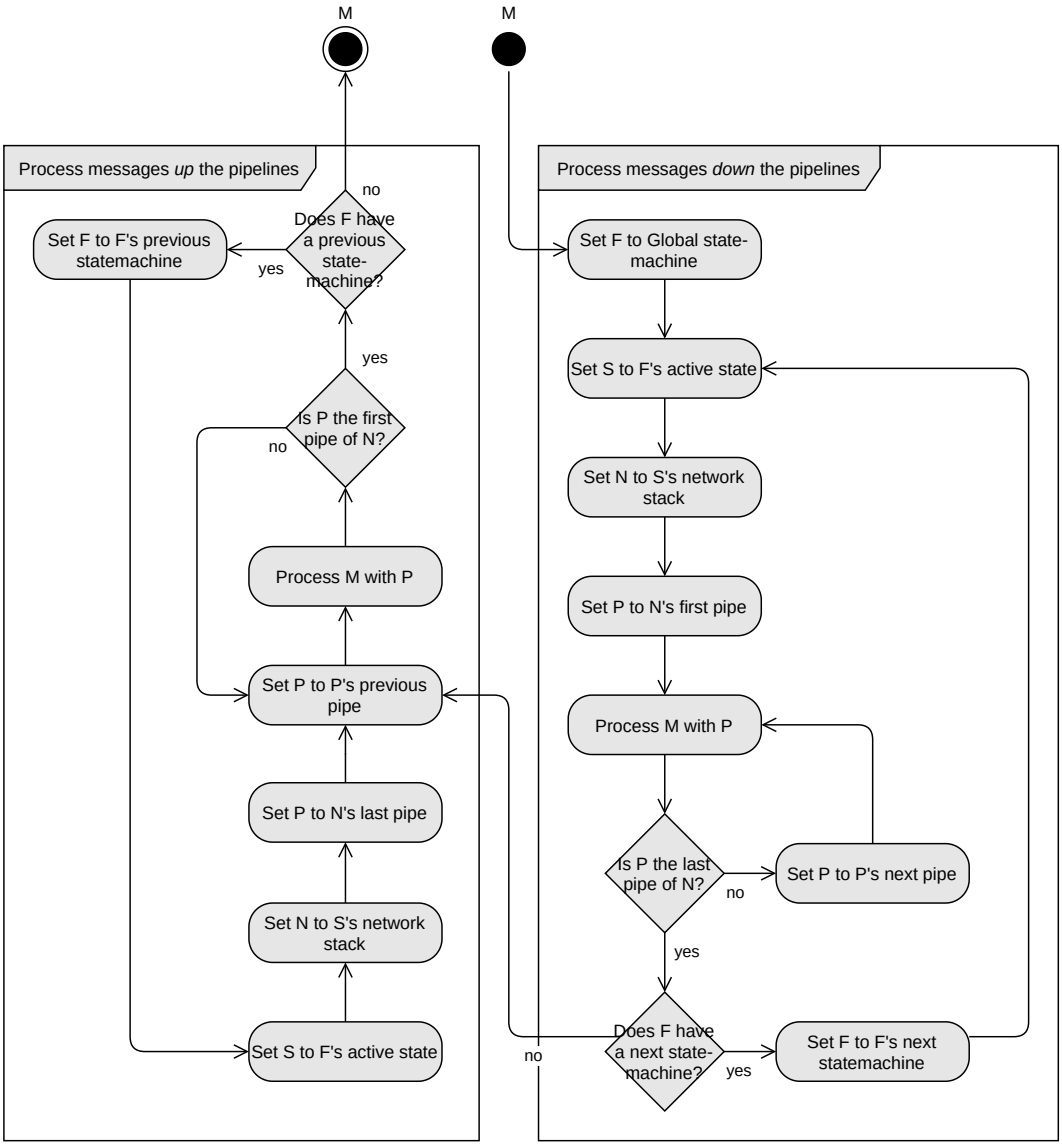


Figure A.3.: Message processing through an architecture of nested FSMs and network stacks

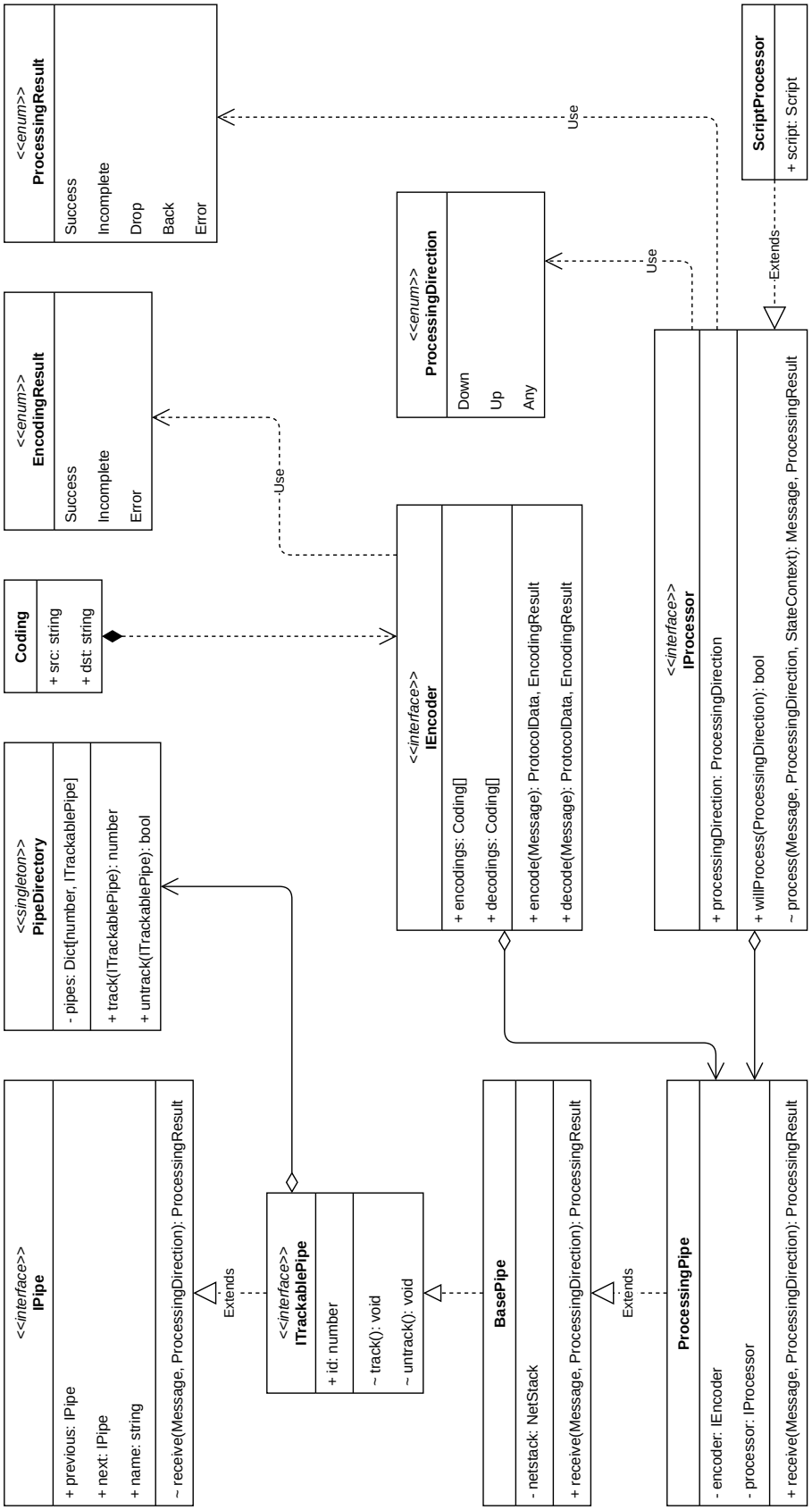

<<enum>>
ProcessingResult
Success
Incomplete
Drop
Back
Error
Use
ScriptProcessor
+ script: Script
<<enum>>
EncodingResult
Success
Incomplete
Error
Use
<<enum>>
ProcessingDirection
Down
Up
Any
Use
Extends
Coding
+ src: string
+ dst: string
<<interface>>
IEncoder
+ encodings: Coding[]
+ decodings: Coding[]
+ encode(Message): ProtocolData, EncodingResult
+ decode(Message): ProtocolData, EncodingResult
<<interface>>
IProcessor
+ processingDirection: ProcessingDirection
+ willProcess(ProcessingDirection): bool
~ process(Message, ProcessingDirection, StateContext): Message, ProcessingResult
<<singleton>>
PipeDirectory
- pipes: Dict[number, ITrackablePipe]
+ track(ITrackablePipe): number
+ untrack(ITrackablePipe): bool
<<interface>>
IPipe
+ previous: IPipe
+ next: IPipe
+ name: string
~ receive(Message, ProcessingDirection): ProcessingResult
Extends
<<interface>>
ITrackablePipe
+ id: number
~ track(): void
~ untrack(): void
BasePipe
- netstack: NetStack
+ receive(Message, ProcessingDirection): ProcessingResult
Extends
ProcessingPipe
- encoder: IEncoder
- processor: IProcessor
+ receive(Message, ProcessingDirection): ProcessingResult

Figure A.4.: ?

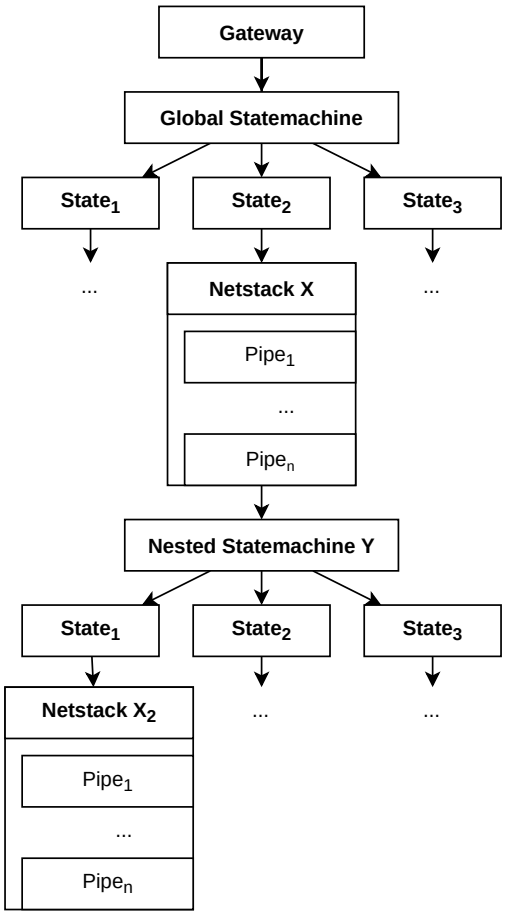

Gateway
Global Statemachine
State1
State2
State3
...
Netstack X
...
Pipe1
...
Pipen
Nested Statemachine Y
State1
State2
State3
Netstack X2
...
...
Pipe1
...
Pipen

Figure A.5.: ?