DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master’s Thesis in Informatics: Games Engineering

Distributed Python Computation in Mixed Reality Environments

Maximilian Schmidt

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics: Games Engineering

Distributed Python Computation in Mixed Reality Environments

Verteilte Python Anwendungen in Mixed Reality Umgebungen

Author:	Maximilian Schmidt
Supervisor:	Prof. Gudrun Klinker, Ph.D.
Advisor:	Sandro Weber, M.Sc.
Submission Date:	August 10, 2022

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, August 10, 2022 Maximilian Schmidt

Abstract

To investigate research questions overarching the domains of augmented, virtual and mixed reality as well as IoT- and web-based technologies with respect to Human-Computer Interaction tasks, the Ubi-Interact [1] framework was developed at TU Munich.

This thesis deals with developing a Python software suite to use this framework, in a way that is tailored for use by researchers and students alike, no matter their domain specific backgrounds. It also showcases the advantages of the setup by providing a modular solution for common Optical Character Recognition tasks as a distributed computation module for Ubi-Interact, using the *Tesseract* [2] engine combined with different image processing algorithms to achieve a robust and flexible solution with sufficient performance to be used in real time applications.

It is shown that Ubi-Interact excels at breaking down problems (or solutions) to different complexities, and allows flexible workflows – from rapid prototyping to performance oriented low level development – which is specifically highlighted by the features of the Python suite.

Contents

Abstract **iii**

1 Introduction **1**
1.1 Historical Classification . . . 1
1.2 Robot Operating System (ROS) . . . 2
1.2.1 Design Goals . . . 2
1.2.2 Messages . . . 3
1.3 Message Oriented Middleware (MOM) – MQTT, DDS . . . 3
1.3.1 Middleware Abstraction in ROS . . . 4
1.3.2 MOM Solution without Abstraction . . . 5
1.3.3 Which MOM? . . . 6
1.4 MOOS . . . 7
1.4.1 Design . . . 8
1.4.2 Message Data . . . 8
1.5 LCM . . . 9
1.6 Ubi-Interact . . . 9
1.6.1 Message Handling . . . 10
1.6.2 Middleware Features . . . 11
1.6.3 Ecosystem . . . 12
1.6.4 Data Transformation . . . 12
1.6.5 Use Case . . . 13
1.7 Thesis Motivation . . . 14

2 Ubi-Interact Python Packages **15**
2.1 Protobuffer Package . . . 15
2.1.1 Readability . . . 15
2.1.2 Extensibility . . . 16
2.1.3 Idiomatic Python Use . . . 21
2.1.4 Type Hints . . . 23
2.1.5 Namespaces and Package Structure . . . 24
2.1.6 Ubi-Interact . . . 24
2.2 Problems and Requirements . . . 31
2.2.1 Middleware Protocol . . . 33
2.2.2 Processing Modules . . . 35
2.2.3 Python Language Features . . . 37

2.3 Design 44
2.3.1 “Protocol” concept 44
2.3.2 Client 47
2.3.3 Processing Modules 50
2.3.4 Node 50
2.3.5 CLI 56
2.3.6 Implementation Details 56
2.4 OCR Module 56
2.4.1 OCR in Mixed Reality 56
2.4.2 Involved Technology 57
2.4.3 Automatic Module Discovery 58
2.4.4 Portability 59

3 Evaluation **61**

4 Conclusion **77**
4.1 Summary 77
4.2 Future Work 78

Acronyms **79**

List of Code Examples **81**

List of Figures **83**

List of Tables **85**

Bibliography **87**

1 Introduction

The motivation for this thesis and the motivation to develop a tool like Ubi-Interact are – of course – closely related. Therefore, if one recognizes the use cases of the framework, the need to extend it into the Python world will arise naturally.

The following discussion will highlight the specifics of Ubi-Interact. By technical comparison with other existing tools that seem to address similar issues, one can make an effort to identify the respective use cases. Nonetheless, not only the identified use cases but also the overall goal for a framework or tool should be taken into account when deciding to use it, since they will guide the further development and improvements – as well as the support – one might expect for each tool.

Ubi-Interact is motivated by the need to have an open and extendable networking ecosystem with sufficient performance to support Human-Computer Interaction (HCI) tasks. It needs to interface with game engines like Unity that are prominently used to develop mixed and virtual reality applications, as well as other existing infrastructure [1]. Ubi-Interact aims to provide an alternative to existing solutions (e.g ROS) in this regard [1], as detailed in section 1.6. For more details on Ubi-Interact design decisions refer to Weber *et al.* [1].

1.1 Historical Classification

Some of the systems discussed in chapter 1 have fallen out of favor since their initial development, the robotics middleware landscape – especially in academics – is virtually dominated by the Robot Operating System (ROS). A comparison of seven different "communication packages" was published in Moore *et al.* [3] at Sep. 2, 2009 – ROS is the only package that is still in active development. Section 1.4 still discusses the Mission Oriented Operating Suite (MOOS) and section 1.5 focuses on the Lightweight Communications and Marshalling (LCM) toolkit, since those packages are the most up-to-date packages that were used prior to ROS (see table 1.1 for latest releases of each package analysed by Moore *et al.*)

Package	Latest Release		Platforms
LCM	August 31, 2018	`v1.4.0` [4]	C, Java, Python, MATLAB
IPC (Carmen)	November 3, 2014	`v3.9.1` [5]	C, Java, Python (since last version)
JAUS	March 12, 2014	`v5.141203` [6]	C
MOOS	June 14, 2018	`v10.4.0` [7]	C++
Player/Stage	April 27, 2017	`4.3` [8]	C, C++, Java, Tcl, Python
Robotics Studio	March 8, 2012	`4.0` [9]	.NET
ROS	active		C++, Python[1]

Note: Adapted from Figure 1, D. Moore *et al.*, "Lightweight Communications and Marshalling for Low-Latency Interprocess Communication," Sep. 2, 2009. [Online]. Available: `https://dspace.mit.edu/handle/1721.1/46708` (visited on 05/25/2022)
[1] community libraries for JVM (Android), Objective C (iOS), C#, Swift, Node.js, Ada, .NET Core, UWP and C# and Rust

Table 1.1: Comparison of (historically) used communication packages [3]

1.2 Robot Operating System (ROS)

1.2.1 Design Goals

The ROS SDK [10] provides middleware components similar to Ubi-Interact. The first association and expectation one might have with respect to Ubi-Interact is that it behaves just like ROS. Since ROS is a mature, widely used and generally well received framework with a big community, a discussion about why one would want to develop something similar "from scratch" instead of appreciating the de-facto standard in robotics development, is needed. Nonetheless most of the ROS design goals and philosophy is applicable to Ubi-Interact.

The goals of ROS can be summarized as being [1]

- Peer-to-peer
- Multi-lingual
- Tool-based
- Thin
- Free and Open-Source

To make ROS *thin*, it provides "modules" – i.e. modular software packages – which employ the principle of component-based systems [12] to make the code reusable and modular in design. Due to the current complexity and scope of ROS, packages need to use a ROS specific build system to correctly integrate into the framework. Technically the current build system `catkin` is not dependent on ROS, and was in fact developed to

[1] Refer to Quigley *et al.* [11] for a more in-depth discussion of the ROS philosophy

reduce the inherent complexity of generic/multi-purpose build systems like CMake to be usable by ROS developers without a software engineering background and to remedy the problems of the old build system `rosbuild` [13]. Notably, the developers of ROS recognized the existing individual preferences for certain languages, as a result of technical or cultural considerations [11]. These preferences can also extend to the eco-system around the languages, including build tools: For example `catkin`'s complexity can be perceived differently, depending on the software background of the user – while it might not seem complex or unusual to developer with a C^{++} background and CMake experience, most interpreted languages like Python or JavaScript package and build their software differently. While familiarizing oneself with a build tool can be expected from most developers, there is certainly an argument to be made in favor of supporting the use of whichever tools they are most experienced in.

1.2.2 Messages

ROS uses custom message generators (as part of the client libraries) which convert message definition files in Interface Description Language (IDL) syntax to source code and have to be invoked from the build scripts. The "gory details [emphasis added]" [14] are mostly hidden from users without a software background: in general, ROS uses the DDS middleware for its communication implementation (which in turn uses IDL messages) but provides an abstraction for the implementation [15]. For ROS 2 a middleware implementation using component libraries like ZeroMQand Protocol Buffers was considered, but an implementation "from scratch" was rejected in favor of the existing `DDS` middleware [15]. Contrarily, Ubi-Interact chooses the alternate route and implements its communication via ZeroMQ (or simple TCP via WebSockets) and protocol buffers, the implications are discussed in section 1.6.

1.3 Message Oriented Middleware (MOM) – MQTT, DDS

Message Oriented Middleware (MOM) protocols and packages are ubiquitous in the Internet of Things (IoT) context – participants need an open network interface to implement the communication, which can vary from a simple RESTful API to complex messaging with QoS promises. This section focusses on DDS and MQTT since both use a publish-subscribe messaging pattern that is suitable for a distributed system with many heterogeneous participants that join and leave independently. DDS represents the Human-Computer Interaction and robotics domain, as it is used to implement the ROS middleware features (see section 1.2), whereas Message Queuing Telemetry Transport (MQTT) is chosen due to its more general IoT/automation use case.

Design Goal	Related Features	Advantages	Disadvantages
Peer-to-peer	`Topic` protocol for broadcasted communication `Service` protocol for synchronous transactions `DDS` middleware layer	Topics and Services are implemented on top of DDS which e.g. already implements a lot of Quality of Service (QoS) features	additional complexity by converting between concepts, e.g. compare *Topic and Service name mapping to DDS* [16]
Multi-lingual	`XML-RPC` for peer-to-peer negotiation	usable in many languages	could use simple REST [17, 18]
	custom message format based on IDL	integrated into build system abstraction from middleware	dependency on ROS
Tool-based	microkernel design	–	–
Thin	build system	consistency	complexity dependency on ROS
	modules	reusable code	–

Table 1.2: ROS Feature Analysis

1.3.1 Middleware Abstraction in ROS

The decision to provide an abstraction to the DDS implementation for ROS has several reasons as explained in *ROS on DDS* [15], mainly these are

- the existence of multiple implementations of the DDS standard
- the inherent complexity of the DDS specification and API

Currently, there are proprietary and open-source implementations of the DDS standard. As mentioned in section 1.2 ROS aims to be open-source, which also applies to Ubi-Interact [1, §2 Motivation]. The effort to stay open-source is related to academic use – which was the biggest use case for ROS systems when it was first developed and is currently the major use case for Ubi-Interact. There are several generally feature-complete open-source implementations of DDS (see table 1.3) and some commercial implementations (which are not considered for academic use).

Name	Development	License	Python Support
OpenDDS	active	permissive[1]	work in progress [19]
Eclipse Cyclone DDS	active	copyleft[2]	Python binding [20]
FastDDS	active	permissive[3]	Python binding [21]
Open Splice	moved to Cyclone DDS	permissive[4]	discontinued [22]

Table 1.3: DDS open-source implementations

ROS provides the middleware abstractions to not be dependent on a specific implementation (the API changes from implementation to implementation) and currently uses `Cyclone DDS` as the default middleware (as of ROS "Galactic", previously `FastDDS` was the default – interestingly the `FastDDS` documentation still claims that this is the case as of the time of writing [24]). This change in default implementation is only possible because of the additional abstraction layer and is bound to happen from time to time as different implementations develop different features.

In addition to the fact that implementation details need to be abstracted, the drawback of using an end-to-end middleware solution is "that ROS must work within that existing design" [15] i.e. adopting an existing solution means adopting the philosophy and concepts of that solution – for example, DDS associates topics with data types *and QoS specifications*. That means that in ROS 2 there is now the possibility for incompatible QoS profiles for subscribers/publishers, while historically – in ROS 1 – any publisher and subscriber with the same message type on the same topic would be connected [25]. Even for concepts which exist one-to-one in ROS and DDS – e.g. topic names – the developers have to deal with possible hidden complexity – for example, allowed naming schemes for ROS topics and DDS topics differ historically, so translating the topic names has to be done carefully – see disadvantages of the topic concept in table 1.2.

1.3.2 MOM Solution without Abstraction

The reasons for ROS to provide an abstraction layer for the middleware implementation show a general trade-off that has to be considered when developing a tool that should integrate into an existing (IT) system. In the case of HCI tasks, these systems consist of the physical devices used for the interaction like IoT gadgets, smartphones, virtual reality devices or web browsers, their relations to each other, the user(s), or to put it simply: "the network of people and technological artifacts involved in the work" [26, p.3]. These "artifacts" will have specific, individual technical features and limitations.

[1] uses custom license with "generous license terms similar to ACE, TAO and MPC" [23]
[2] uses Eclipse Public License v. 2.0
[3] uses Apache License v. 2.0
[4] uses Apache License v. 2.0

Weber *et al.* [1] identify the need for the Ubi-Interact framework to integrate specific technology (or replace it) which is associated with the *artifacts* one is dealing with in the context of HCI tasks in mixed and virtual reality 3D environments – like game engines (e.g. `Unity`), physics engines, ROS for "full-body virtual re-embodiment avatars" [27], or the specific data communication interfaces of IoT devices.

Developers have to deal with the trade-off that – as a matter of principle – if you use an existing tool, you get the features of that tool "for free", but you pay to *adapt them to your domain*. This trade-off is the reason a Message Oriented Middleware without additional domain specific abstractions does not scale well with respect to code reusability. Application developers need to have in-depth knowledge of the MOM to write *adapters* for specific use cases, if no abstraction is provided by the framework. Increasing code reusability is one of the main features of ROS and Ubi-Interact alike, which is hard to achieve if applications in various domains with diverse requirements and limitations need to be implemented relative to a fixed MOM (precisely when features or concepts are not already present in the MOM implementation to begin with). In the end, a good abstraction is not only a necessary decision due to the fact that one tries to provide *integration* and *scalability*, it is in itself a feature.

1.3.3 Which MOM?

If one chooses to use an existing MOM to implement the communication in a distributed system, due to the cost associated with *adaption* in section 1.3.2, it would be ideal to use a solution which is "isomorphic" to the communication concepts used in the developed framework. This is trivial if one allows the communication paradigms to be fully dictated by the MOM solution, which is – as discussed in section 1.3.2 – difficult or impossible when *integration* into an existing system is desired. Refer to table 1.4 for an overview of strengths and weaknesses of MOM solutions and their categorisation in terms of importance for virtual or mixed reality HCI – for more details on the feature grading refer to Aures and Lübben [28]. Notably, MQTT does provide fewer options in terms of data modeling, but is considered simpler to use [28, §5.10], which is a consequence of MQTT being a very lightweight protocol as implied by the "protocol overhead" in table 1.4. In turn, this means that to provide the missing data modeling capabilities which are desired in the HCI context, a framework which relies on MQTT needs to implement those on top of the communication protocol – in which case it could arguably be simpler not to use MQTT in the first place and implement the "middleware" layer exactly to the desired specifications.

While both MOMs in question use a publish-subscribe pattern, the Data Distribution Service relies on *peer-to-peer* communication whereas the Message Queuing Telemetry Transport uses a *broker* setup where all communication is addressed at the broker application responsible for distributing the messages to subscribers. Arguably, the latter is preferable for the use in mixed reality HCI tasks involving IoT gadgets, since it creates

a topology where the broker can naturally provide additional data manipulation to the connected nodes. According to Weber *et al.* if "[...] for example an IoT device already provides access through open network interfaces like [sic] RESTful API, a central modular process can be established as a communication and status manager for this device – again exposing its capabilities to the wider system" [1, §3 Goals]. On the one hand, it can also be argued that a star-shaped network topology remains simpler, no matter the number of participants, and "[...] the clients operate independently with interconnections. This prevents rogue clients (badly written or hung) from directly interfering with other clients" [29, §1.1 Topology], but setting up communication this way could introduce "bottle-necks" at the broker on the other hand.

Measure	DDS	MQTT	3D VR/AR HCI Importance
data integrity	++	-	*Security*
authentication	++	+	Importance depends on data / legal context,
access control	++	-	*arguably less important* in research than in
encryption	++	-	production [30, p. 16]
data gnostic	++	-	*Data Modeling*
data centric	++	-	*Important* – data in 3D VR / AR context typically
serialization	++	-	is complex and heterogeneous
protocol overhead	-	+++	
QoS	+++	+	
simplicity of use	-	++	*Practical usability*
real world testing	++	+	*Important* – especially in research, since it affects
monitoring & RTM	+++	+	resources spent on application development [28]

Note: adapted from Table 1, p. 3, G. Aures and C. Lübben, "DDS vs. MQTT vs. VSL for IoT," *Network*, vol. 1, 2019

Table 1.4: Overview of MOM solutions [28]

1.4 MOOS

An interesting solution because of the simplicity of the design, the Mission Oriented Operating Suite (MOOS) – a "Light, Fast, Cross Platform Middleware for Robots" implemented in C++, with Python bindings available [31] – is not actively developed anymore (latest version `v10.4.0` released June 14, 2018 [7]). It has a "maritime heritage", but as of version `10.0.0` the domain independent communication tools were cleanly split from the application code that was used in maritime autonomy contexts [32] and made available as the `core-moos`[7] library.

1.4.1 Design

The communication scheme that MOOS used prior to version `10.0.0` is very simple. Instead of an asynchronous publish-subscribe pattern, the communication between MOOS clients and the broker (called the `MOOSDB`, since it simply acts like a storage or "mailbox" for messages) happens synchronously at a defined rate that can be specified for each client. Every time the client communicates with the broker, it sends all messages in its *outbox* – wrapped into a single packet or "super message" – and the broker replies with a packet containing notifications for the clients subscriptions which are placed in the client's *inbox*. Retrieving information from the *inbox* or putting messages into the *outbox* can happen asynchronously, but for the actual communication over the TCP/IP connection a "one packet sent, one packet received" policy is enforced [29, §5.1]. With version `10.0.0` asynchronous communication was added, as well as "wildcard" subscriptions.

MOOS components that implement different functionality or components run in separate processes[1] with one MOOS client instance per process and don't need to know about each other[2], they can however be grouped into "communities" (a group of related processes that handle inter-process-communication via their own `MOOSDB`), and a *bridge* application can handle data sharing between multiple communities or `MOOSDB` instances [34]. This allows developers to use more complex client topologies and UDP connections (for data sharing), for example when using unreliable wireless connections.

1.4.2 Message Data

MOOS does not concern itself with data marshaling, instead messages can only contain data as floating point numbers or strings – which then will need to be parsed by the specific application. Similar to the use of single processes instead of threads, this can be partly attributed to the academic use of MOOS – Newman [29, §3] advocates the use of string data for anything non-scalar to

- make the data and log files human readable
- make all data the same type
- make it easy to replay a log file for developing and debugging
- allow the data "schema" (i.e. the contents and internal order of string data) to change without crashing clients[3]

[1] Each MOOS client uses multiple threads to handle the communication
[2] There are multiple components available to use, see *essential-moos* [33]
[3] of course they would not be able to understand the changed data

which addresses similar issues to the threading decision which was made on account of "stability" and "the basis of swift and pain-free development by several programmers with diverse backgrounds", referring to programming guidelines and styles that Newman considers to not be necessarily native to all software developers "especially in an *academic environment* [emphasis added]"[29, §4]. These concepts and their justifications MOOS introduces for simplicity might prove valuable for similar software designs or turn out to be a mere product of "simpler times".

1.5 LCM

The Lightweight Communications and Marshalling (LCM) libraries and tools were also developed "especially for real-time robotics applications" [35], similarly to ROS. At the time, they were the only solution that handled type-safe marshalling (i.e. encoding and decoding data) of data for multiple languages [3, Fig. 1]. Here "type-safe" refers to the handling of "endianess" of the binary messages shared across components. At the time of LCM's development, ROS did only support little-endian systems, and did not have the community support for client libraries in languages other than C^{++} and Python that it has today. While the discussions of shortcomings in software packages used in 2009 is not contributing much to the design of Ubi-Interact, Moore *et al.* [3] identify the need for automatic marshaling for multiple languages. As seen in section 1.2 and table 1.2 this need was also identified as a key component in the design of ROS, and has been addressed with different solutions over the course of its development – currently it is handled by the DDS middleware, as discussed in section 1.3.1. Both LCM and ROS chose to use their own type specification languages – based on IDL for ROS and the XDR [36] standard for LCM – at a time where there were de-facto no better alternatives. For modern software like Ubi-Interact the use of a custom IDL is neither practical nor necessary: The need for cross-platform serialization for major players in web-based technologies like Google or Facebook led to the development of several widely used interface description languages like Protocol Buffers [37] or FlatBuffers [38] which support all features that were deemed relevant 10 years ago (e.g. handling of endianess), as well as practically all popular languages.[1]

1.6 Ubi-Interact

There are two ways to approach the design of Ubi-Interact. One way is to identify the goals and evaluate the solutions, similar to the way design goals for ROS were evaluated in table 1.2. A different way is to analyse the solutions – in the context of the previous discussions of this chapter – and develop a use case that justifies them. Then the evalu-

[1] the IDL used by ROS is currently depending on the choice of middleware solution. Not using popular serialization libraries is a trade-off made in order to gain features and convenience by not implementing the middleware from scratch, see section 1.3.1

ation of the design will be related to the applicability of the constructed use case and its congruency with the use case that Ubi-Interact aims for. While the first way is natural during the development of the system, for a developer who joins the project at a later stage, the design will be primarily "perceived". The features presented by Weber *et al.* in section 5 of "Ubi-Interact" [1] are mostly still relevant for the current iteration of the framework. They will be categorised as *Message Handling*, *Middleware Features*, *Ecosystem* and *Data Transformation* and discussed in sections 1.6.1 to 1.6.4.

1.6.1 Message Handling

Ubi-Interact uses protocol buffers to handle message serialization in a portable, cross-platform, cross-language manner. Protocol buffers address all issues that were identified by Moore *et al.* [3] and Newman [29]. One of the most practical features of the protocol buffer framework is the possibility to develop plugins for the `protoc` compiler, which is responsible for compiling schema files to platform and language dependant application code. A compiler plugin itself acts on specific protocol buffer messages with easily accessible API [39]. Therefore, a multitude of third party plugins have been developed to alleviate issues and implement features for the official language plugins – compiling schema files to documentation, to native code which can be used more idiomatically[1], to supplemental files that allow optional static type checking in dynamically typed languages, and much more.

Readability for humans is supported by easily readable schema files, plugins to generate documentation for those files, and support for conversion to and from human readable representations like JSON out of the box.

Debugging or "replaying" messages for applications is easy, since the messages can be displayed, stored and even manipulated in readable form (JSON) and feeding them into the application then becomes a matter of knowing the respective message types. This also applies to developing compiler plugins: The plugin can be implemented in any language, and will read a defined protocol buffer message from standard input during execution, and then write a message of a predefined type to standard output. This can be mocked in a few lines of code to develop the plugin independently from the use in a `protoc` call, which greatly simplifies debugging and introspection during development.

[1]the types produced by the official plugins all use the same API which is similar to the C^{++}API (e.g. in naming conventions) and does not lend itself particularly well to produce idiomatic code in e.g. Python or JavaScript

Since the protocol buffer specification encourages users to make fields of the message schema *optional* (although *required* fields are supported) updating or changing the schema does not automatically crash clients that don't know about the updated schema, instead this agreement on (compound) data types between processes can help to correctly differentiate message compatibility. This rather finely grained control over the agreed structure of exchanged data can be a big factor in developing correct, distributed and robust applications.

On top of the issues which were discussed in section 1.4, Moore *et al.* [3] identify some requirements for marshaling tools – like handling of byte order, which was discussed in section 1.5, or support for language specific features like the use of namespaces to prevent type names from clashing [3, § 3.1.3] which are also met by protocol buffers[1].

While disagreements on the structure of the data can happen – and don't always equate to incompatibilities – the platform and language specific code for a protocol buffer schema generally should be up-to-date across applications. Since one is dealing with multiple libraries – maybe even multiple libraries for a single target language, e.g. in the case of optional files of a static type checker – one is also dealing with multiple eco systems and tools to model software dependencies. It is therefore beneficial to invest into smooth packaging and distribution of the protocol buffer libraries.

1.6.2 Middleware Features

Akin to ROS, Ubi-Interact offers two communication patterns. Communication using a publish-subscribe paradigm is offered over an asynchronous bi-directional channel between client and the message broker – Ubi-Interact does not use peer-to-peer communication. For request-reply interactions this communication would not be suitable, nonetheless these interactions are often required [40] therefore an alternative synchronous channel is used. The asynchronous channel can be implemented via WebSockets or ZeroMQ router-dealer-sockets – Ubi-Interact currently foregoes the use of existing middleware solutions (like MQTT, ZeroMQ's publish-subscribe sockets or DDS) to

1. simplify the technology stack
2. simplify the network topology (compare section 1.3.3)
3. facilitate preferable – or at least equivalent – handling of web browsers and web technologies

The synchronous channel can be implemented via `HTTP(S)` requests or `ZeroMQ`'s request-reply sockets. There is no apparent reason to use a specific RPC protocol over this channel, thus the messages can be either transmitted in binary form (compare section 1.6.1) or as JSON – the amount of transmitted data is typically not an issue for infrequent request-reply communication.

[1] at least in theory – whether the generated code respects the namespace declaration of the schema files is an issue for chapter 2

The fact that the broker offers multiple communication channels for each client allows Ubi-Interact to bridge the gap between different technology domains. For example mixed reality applications for the `Microsoft HoloLens` are typically developed using Unity's integration for Microsoft's Universal Windows Platform (UWP) framework, but Unity's `.NET` flavour is not fully compatible with the `.NET` for UWP specification [41] – also UWP can't make use of new `.NET` versions[1]. This introduces a technical limitation on serialization [41], making it impossible to use the standard C# implementation of the protocol buffer package for JSON (de-)serialization in this very specific context. Developers that are experienced in this domain know that they are encouraged to use different serialization [41] e.g. to/from binary data. Instead of forcing developers in every domain to use a binary encoding for communication with the broker – due to technical limitations in a single domain – they can choose whichever encoding is more practical for their use case, hence integrating web and IoT applications with the mixed and virtual reality domain on equal grounds.

1.6.3 Ecosystem

Ubi-Interact packages are developed natively for every targeted platform or language. There is no "cross-compilation" support, and the different implementations have different features. Conversely though, Ubi-Interact makes heavy use of the cross-platform protocol buffer specifications to encode internal state and other parts of the system "worth communicating" [1, §5.1], this includes specification of system behavior via *Processing Modules* [1, §5.5] – see section 1.6.4. Because of the distributed nature of these modules, they should – if the platform and environment enables it – be readily available using the surrounding eco-system (e.g. `NodeJS` / `npm` or other language dependent packaging systems).

Processing Modules are often used to integrate external libraries like ROS, TensorFlow or OpenCV with specific dependencies on the environment [1, §5.5] into a larger distributed application – therefore it is not always possible or desirable to have an existing implementation of a given module on all possible platforms. To achieve code reuse and modularity though, the modules need to be *easily discoverable* – best-case in a platform independent manner – to inform developers about (pre-)existing implementations and functionality.

1.6.4 Data Transformation

"Data Modeling" is important in the mixed or virtual reality HCI context, since data can become very complex and heterogeneous (compare table 1.4) data access and discovery at application level should be transparent [28]. Topics in Ubi-Interact are identified by a topic string, but through the concept of *Devices* and *Components* which are encodable

[1] to support `.NET` 5/6, apps need to be developed with `WinUI` 3 [42]

via corresponding protocol buffer messages – therefore known to, and agreed upon by, all participants – topic strings can be associated with additional meta information like the exchanged message format, arbitrary tags, the corresponding device (which is not necessarily a physical device but simply a "meta" structure to group several components logically) and more [1, §5.4]. Efforts are made to make inference and searching of these components easier, but these concepts will not be enforced on participants – it is possible to subscribe and publish to a topic without dealing with meta information.

Additionally, Ubi-Interact uses *Processing Modules* to "provide system behavior in a decoupled, I/O device agnostic, modular, reusable and shareable fashion" [1, §5.5]. They can e.g. provide distributed topic data manipulations, processing or analysis – typically when instantiated in dedicated *processing nodes* – or implement a reusable communication endpoint to facilitate communication between Ubi-Interact and a (physical) device, application or infrastructure which comes with – or needs to use – different interfaces or (network) API than Ubi-Interact.[1] This feature is rather powerful and definitely part of the feature set that should be present in the Ubi-Interact implementation of every target platform – to be precise it gets "exponentially" more powerful the more possible platforms are targeted, since each new platform can make use of its unique capabilities to allow new behavior to be shared with all existing applications.

1.6.5 Use Case

The discussion in sections 1.6.1 to 1.6.4 shows that Ubi-Interact is suitable for academic research in HCI contexts, since it aims for simplicity where possible and makes minimal assumptions on the participating devices or clients. Advanced concepts like *Processing Modules* or *Components* are not enforced on the user, which allows developers to prototype quickly and enhance their applications as they get more experienced with the features, while basic features are similar enough to popular frameworks like ROS to be picked up with minimal experience. Since Ubi-Interact targets different languages and platforms, developers can "stay in their comfort zone" and still contribute meaningfully by sharing their work as a module while exploiting the unique capabilities of each platform.

This use case is currently also the one where Ubi-Interact is used in practice for research in serious games [43, 44], augmented reality applications [45] and "superhuman sports" [46]. Efforts are also made to research its application for physical (re-)embodiment in VR [27]. It is apparent that Ubi-Interact does not compete with ROS in the robotics domain. Instead it aims to interface and bridge the gaps to allow students and researchers to develop novel applications that are not constrained to a single domain.

[1] for details refer to Weber *et al.* [1, §5.5]

1.7 Thesis Motivation

Extending Ubi-Interact into the Python world contributes meaningfully to the overall "eco-system" since Python is a very popular programming language [47, 48] – especially in academics, since it is easy to teach[1] and use. It provides powerful packages for computer vision tasks and "data science" in particular and the ubiquitous "Python Notebooks" are a great tool to (collaboratively) implement and learn a variety of concepts in typical computer science curriculums.

Consequently, development of the Python package was not done with a specific application in mind, instead it focuses on being *educational*, *idiomatic*, *easy to maintain* and last but not least *well documented* – to be of use for students and researchers alike.
Nonetheless, an example processing module was implemented, which integrates different Optical Character Recognition (OCR) tools to provide real-time text recognition for a stream of image data e.g. from a camera device. The integration of Python applications into the Ubi-Interact communication and middleware layer is already used in the context of the "superhuman sports" project by Eichhorn *et al.* [46].

The *Ubi-Interact Python Node* [49] implements only the client and processing node capabilities of the Ubi-Interact protocol since there is currently no need to re-implement the broker node.

[1] it's a little less easy to teach correctly, but that's not necessarily the focus in academics <\rant>

2 Ubi-Interact Python Packages

This chapter focuses on the implementation goals and design decisions that define the *Ubi-Interact Python Node*. When technical implementation details become relevant to justify the design, they will be illustrated by short code examples. Section 2.3 also briefly showcases parts of the Python API, highlighting where it adheres to requirements that were identified in section 2.2.

2.1 Protobuffer Package

As already mentioned in section 1.6.1 the official protocol buffer compiler plugin for Python produces code that is definitely improvable when it comes to

1. readability
2. extensibility
3. use of Python idioms and language features
4. use of *type hints*
5. namespaces and package structure

which will be discussed in sections 2.1.1 to 2.1.5.

2.1.1 Readability

Discussing software readability measures in-depth is out of scope for this section, since they are themselves an active research topic as they are a widely accepted proxy for code quality. Most code readability measures mainly rely on structural metrics [50], but some also combine those with textural metrics which analyze the "source code lexicon" i.e. the lexical tokens used in code snippets [51]. While the code generated by the default Python `protoc` plugin is functional and syntactically correct – which implies a certain code structure solely because Python syntax cares about *indentation* (a common structural metric) – it scores poorly in terms of readability metrics since it is always "optimized for code size" [52] and therefore uses minimal classes and reflections to build the API when the modules are imported.

As you can see in example 2.1.2 the generated code does not reflect the field layout of the schema file in any way and it is not visible which names will be present in the global namespace after importing the module or which API methods are present. All this information will have to be inferred from the protocol buffer API documentation and additional custom documentation for the schema files. Although comments are possible in schema files, they are not used in the generated Python module and the information is lost on the user.

To make things worse, the API of the generated types uses capitalized function names like `SerializeToString` while the Python style guide clearly recommends function names to be “lowercase, with words separated by underscores as necessary to improve readability” [53] and reserves “CapWord” naming for classes and type variables. Even “mixedCase is allowed only in contexts where that’s already the prevailing style (e.g. threading.py) to retain backwards compatibility” [53]. This means code that uses the types generated with the default Python `protoc` plugin will have clashing naming conventions and therefore be less readable and idiomatic.

2.1.2 Extensibility

Types generated by the default plugin are (by design) not transparent enough to be easily extensible with additional functionality. The actual metaclass[1] that builds the message classes is not supposed to be extended by “outside clients” [54] – the details of the construction of new message classes are hidden inside of stateless construction helpers, which are not methods on the metaclass, to make it even more clear that they are “not really using any state there and to keep clients from thinking that they have direct access to these construction helpers” [54]. Of course this is all by design, the documentation on generated Python code warns users that the “generated classes are not designed for subclassing and may lead to ‘fragile base class’ problems. Besides, implementation inheritance is bad design.” [52] Notwithstanding this claim by the protocol buffer developers, using the messages in a intuitive, extensible and last but not least “pythonic” way merits some further discussion.

First, one should clarify what the actual public API of a protocol buffer message should allow:

- Assigning to public message fields
- Serializing the message to bytes
- Deserializing bytes to message objects
- Converting messages back and forth between different non-byte representations like JSON or Python dictionaries

[1] `google.protobuf.internal.python_message.GeneratedProtocolMessageType`, see [54]

```
syntax = "proto3";
package my_package.dataStructure;

message Color { // 4 Channel color using r,g,b and alpha channel
    double r = 1;
    double g = 2;
    double b = 3;
    double a = 4;
}
```

2.1.1: Protocol buffer schema defining a color – `color.proto`

```
# -*- coding: utf-8 -*-
# Generated by the protocol buffer compiler.  DO NOT EDIT!
# source: color.proto
"""Generated protocol buffer code."""
from google.protobuf.internal import builder as _builder
from google.protobuf import descriptor as _descriptor
from google.protobuf import descriptor_pool as _descriptor_pool
from google.protobuf import symbol_database as _symbol_database
# @@protoc_insertion_point(imports)

_sym_db = _symbol_database.Default()




DESCRIPTOR = _descriptor_pool.Default().AddSerializedFile(b'\n\x0b\x63olor.
    proto\x12\x18my_package.dataStructure\"3\n\x05\x43olor\x12\t\n\x01r\x18\x01
     \x01(\x01\x12\t\n\x01g\x18\x02 \x01(\x01\x12\t\n\x01\x62\x18\x03 \x01(\x01\
    x12\t\n\x01\x61\x18\x04 \x01(\x01\x62\x06proto3')

_builder.BuildMessageAndEnumDescriptors(DESCRIPTOR, globals())
_builder.BuildTopDescriptorsAndMessages(DESCRIPTOR, 'color_pb2', globals())
if _descriptor._USE_C_DESCRIPTORS == False:

  DESCRIPTOR._options = None
  _COLOR._serialized_start=41
  _COLOR._serialized_end=92
# @@protoc_insertion_point(module_scope)
```

2.1.2: Python module compiled from `color.proto` using default plugin

- Inspecting the message structure[1] to e.g. find out which field of a `oneof` group is set, or which field names are used – this makes code more "data gnostic"

Conversion to and from `JSON` is supported by the `google.protobuf.json_format` module, while the other features are supported out of the box by the generated message types.

The following considerations also influence our requirements for a good protocol buffer implementation (in no particular order and initially without any valuation)

- Python supports multiple inheritance – not just as an afterthought or "by accident", but as a major language feature.
- Python is dynamically typed and makes use of "duck typing", this means formally meeting type or interface specifications by inheritance should be rarely needed.
- Python type hints, generics, abstract base classes and *Protocols*[2] can be used to implement *co-variant* and *contra-variant* "type contracts".
- Inexperienced OOP developers are often familiar with some sort of inheritance, mostly to model a "is a" relationship when reasoning about types – but sometimes also to reuse code.
- If one chooses to use inheritance as part of the public API in any way, volatility of the base class implementations will likely introduce the aforementioned "fragile base class" problems.
- Users will likely port code from other Ubi-Interact target platforms like JavaScript and – depending on their Python experience – try to emulate patterns they observe in this code in their Python implementations.

We also observe how the public API for a message is modeled in the JavaScript implementation of the protocol buffer framework used by Ubi-Interact: It allows the objects to have arbitrary public attributes but simply ignores any that are not part of the message schema when (de-)serializing. This can lead to patterns that (ab-)use this fact by inheriting from message classes, to create types that encode different state and behavior but share a representation – at some point the JavaScript implementation used an implementation like this for *Processing Modules* which represent arbitrary code for a specific purpose but need to be serializable in a consistent manner to communicate that a client offers or requests such a module, see section 1.6.4.

[1] also known as *reflection*
[2] also known as "static duck typing", see Ivan Levkivskyi *et al.* [55]

Use Case: Errors as Exceptions

Python makes use of exceptions as flow control structures – different exception types can be used in *except* statements to handle specific kinds of exception objects that are *raised* in the corresponding *try* block. Libraries define their own exception types by inheriting from existing exception types e.g. from the standard library. The Ubi-Interact framework defines a schema for `Error` messages which can be shared between clients and the broker to inform participants about failures in the system [1, §5.3]. If a Python client receives such a message, it maybe needs to change the program flow by triggering some exception handling.

Therefore the Python package should supply a custom `Exception` type (preferably multiple types for different kinds of errors) that can be in some way (de-)serialized as an `Error` message.

Use Case: Processing Modules

Users of the Ubi-Interact framework can use *processing modules* (compare section 1.6.4) to implement distributed computations. These modules are defined in terms of different protocol buffer messages to model "desired inputs and expected outputs" and the processing mode [1, §5.5]. The Python implementation should define some API to run user-defined Python processing modules and developing new modules should be easy and fast. Using an API similar to the JavaScript implementation where modules define a public interface of lifecycle callbacks that can be implemented in user defined modules is preferable.

Therefore the Python package should supply an interface that users can implement to create objects with the appropriate processing module callbacks, which are (de-)serializable as Ubi-Interact `ProcessingModule` messages.

Design for Extensibility

One can support both use cases with wrappers around protocol buffer messages since extending the generated types should be avoided. We will discuss some example designs for the *Exception* use case.

The simplified design seen in fig. 2.1 hides the protocol buffer specifications and only provides a public API to convert back and forth between types, basically designing the specifications as immutable once the exception object has been created. This might be possible for exceptions which *are* typically treated as immutable objects, but it will not generalize to the *Processing Module* use case since the specifications of a module might be updated during its lifetime. In this case the public interface needs to mirror the `Message` interface of the protocol buffer package, to allow merging and updating the specifications as needed.

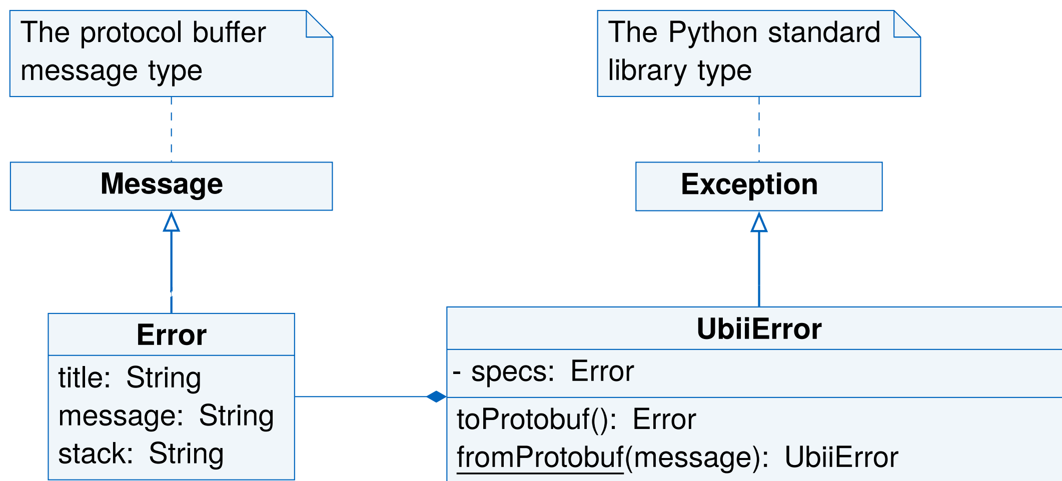

Figure 2.1: Design of protocol buffer handling hiding specifications

The simplified design seen in fig. 2.2 models the protocol buffer specifications as a public data member, while still keeping the factory class method to initialize the objects. This makes the specifications mutable by design, and one could use the Python `property` decorator to manage the specification access[1].

The binary encoded channel for `HTTP(S)` requests was added for the reasons concerning the interaction between the `C#` implementation and the `Microsoft HoloLens` described in section 1.6.2, late during development of the Python package. Since it is harder to debug transmitted messages which are not human readable, the Python package did not switch to binary encoding. Consequently, as discussed in section 1.6.2, the implementation also needs a way to serialize and (de-)serialize the messages to `JSON` for synchronous communication with the broker.
Since `JSON` encoding is not part of the basic API of a generated protocol buffer message and instead supported by the `google.protobuf.json_format` module as mentioned previously, the design in fig. 2.3 is an improvement over the design from fig. 2.2, since it bundles all needed functionality in an interface that is generalized for all use cases where custom types need to have a protocol buffer message specification. This interface could even be implemented using Python's support for *generics*[2], to not loose the information about the specific `Message` type that is wrapped. Note that an OOP *interface* is modeled through abstract base classes and multiple inheritance in Python, and

[1] there are no truly private members in Python, so it's not a good idea to jump through hoops to get the "most private member possible", but properties are typically a good compromise to keep users from accidentally using an attribute they are not supposed to

[2] generic classes are a feature of the typing support introduced with *PEP 484 – Type Hints* [56]

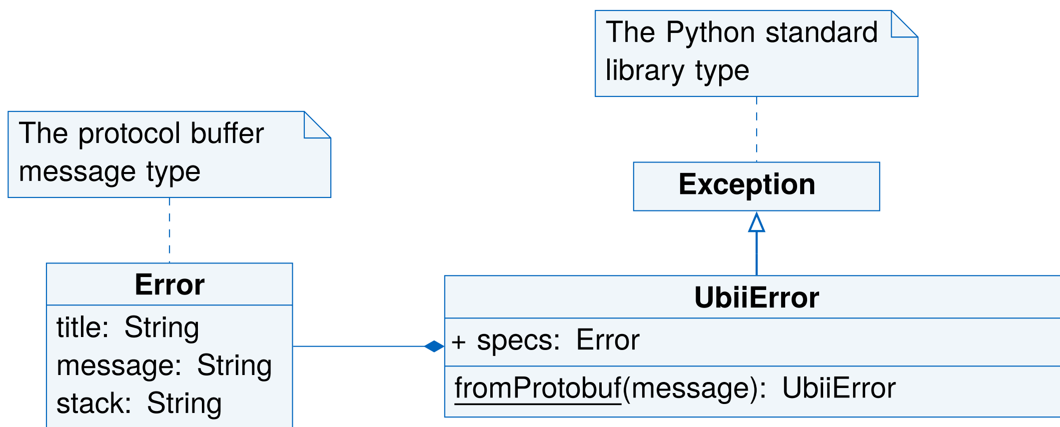

FIGURE 2.2: Design of protocol buffer handling with public specifications

– practicality aside – one could even go as far as declaring all members of the abstract `ProtoSpecs` type in fig. 2.3 as abstract (`specs` would the become an abstract property) to get a "pure" interface that can be implemented in a concrete type for some specific protocol buffer implementation.

Although defining a custom interface to serialize and deserialize messages and implement it against a specific protocol buffer package or API would allow to switch implementations later, it introduces additional complexity which needs to be documented. Fixing the protocol buffer implementation seems to be the more practical approach, since the documentation doesn't need to be duplicated, adjusted or otherwise maintained – which is especially important in our academic setting.

2.1.3 Idiomatic Python Use

Additional issues are present in the default protocol buffer package which make idiomatic Python code harder to write:

- It defines a message interface with method `SerializeToString` which converts the message object to bytes. This could be implemented using the `__bytes__` special method in pythonic code as explained in *Data model — Python 3 documentation* [57].
- Direct assignment to embedded message fields is not possible. Instead, assigning a value to any field within the child message implies setting the message field in the parent – if a nested message should be set from another message, one can use `CopyFrom` to copy all nested fields [52].
- Direct assignment to repeated message fields has the same issue, which further convolutes assignments.

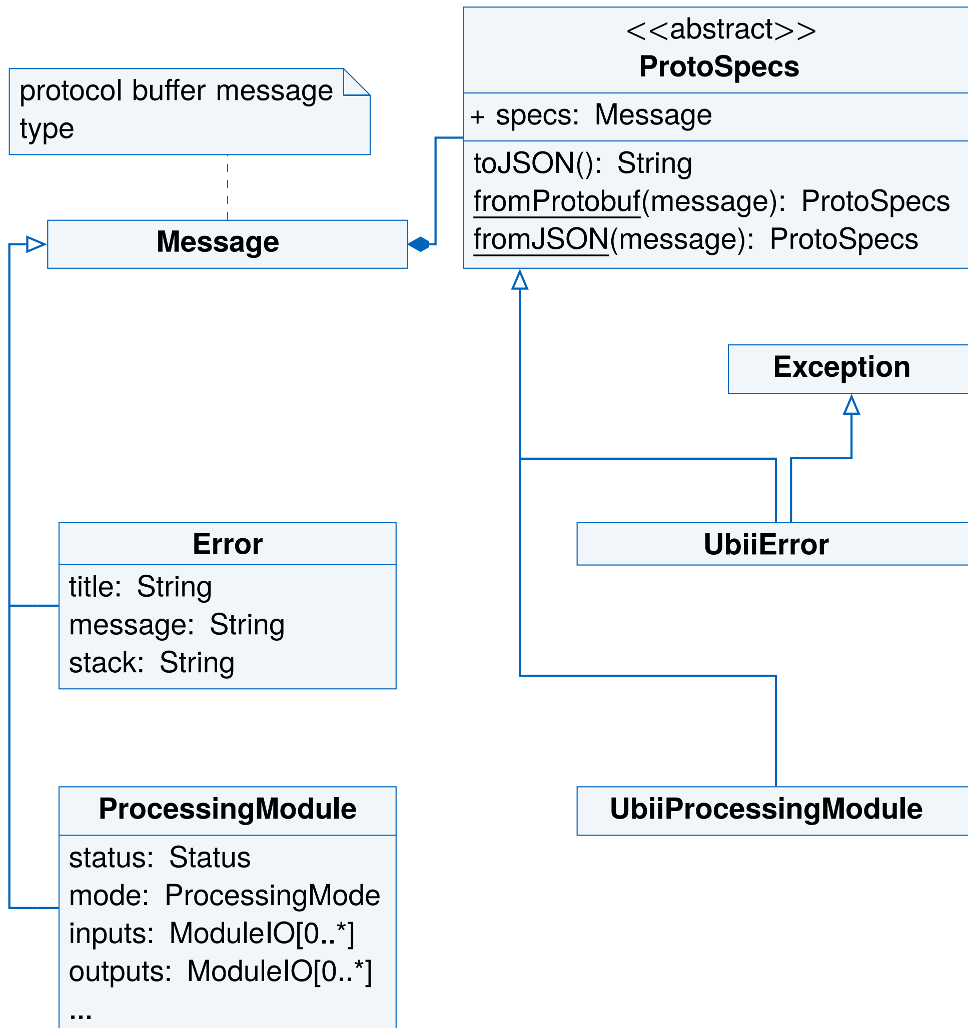

FIGURE 2.3: Design of protocol buffer handling using abstract specification base class

The problems with composite field assignments can't really be solved easily since they relate to memory management details in the protocol buffer runtime. In theory they could be made implicit – of course this could lead to situations where it is not obvious that message copies are made internally which then need to be addressed.

What should be expected though, is that the types of the `google.proto` package conform to Python standards, for example a `RepeatableCompositeFieldContainer` which is used for access to fields that are declared as `repeated` in the schema file and contain composite types (e.g. another message) claimed to implement the `MutableSequence` interface defined by Python's `collections.abc` module. In fact, it did only implement the full interface since version `3.20.0` of the official package – before, it was missing support for operators[1]. With version `4.21.0` though (following `3.20.1`) the Python protocol buffer implementation switched to a new protocol buffer runtime – μpb [58] – and once again does not implement the correct Python interfaces. The wrapped C types provide some functionality from a `MutableSequence`, like extending or appending with corresponding methods. However they don't adhere to the Python data model and – like previous versions – don't implement the `__add__` or `__iadd__` methods which *are* required by the `MutableSequence` interface. In practice, this means that assignments of the form

```
message.repeated_fld += [additional_content]
```

or

```
message_a.repeated_fld = message_b.repeated_fld + [additional_content]
```

will raise errors, although they are expected to work for a type which claims to conform to the `MutableSequence` interface.

2.1.4 Type Hints

Python type hints are not new anymore[2] and allow static type checkers to help developers write code that is less likely to fail at runtime. The default protocol buffer package does support type hints since version `3.20.0` which was not available when development on the Python node started. Before the functionality was integrated into the default compiler plugin, a third party plugin could be used to generate *type stubs* from schema files [59]. Type stubs are Python modules that type checkers can use to infer type information for libraries that don't use type hints. They need to follow special naming conventions to be usable [60].

[1] operators are supported by implementing specific "dunder" methods in Python, e.g. objects which implement an `__add__` method support addition with the `+` operator

[2] introduced in *PEP 484 – Type Hints* [56]

2.1.5 Namespaces and Package Structure

Protocol buffer schema files allow users to define *packages* to prevent name clashes and structure the defined types [37]. These packages are ignored for the generated Python modules though – instead the modules are organized according to the file system layout of the schema files used as compiler input [52].

Use Case: Topic Data Types

Ubi-Interact clients publish data in topics as `TopicDataRecord` messages. This type has a *oneof* group defining all possible payload types the record could carry[1]. To allow clients to associate searchable meta data with specific topics they can register *components* (which are serializable with a specific protocol buffer message), this would allow a client to e.g. subscribe to all topics with specific tags, clients, devices or – last but not least – *data types*. Of course the information of the expected data field inside the payload *oneof* of all records in a specific topic needs to be encoded then and the most practical way is to use the unique name of the protocol buffer message[2]. Without special care during development of the Python package, this information would not be sufficient to e.g. import the correct Python type for the message, since the *package* structure from the schema declarations is present in the unique name that is passed, but not necessarily in the import path for the corresponding Python module.

In fact, the Python modules generated by the default plugin are mapped one-to-one to input schema files: The output is a loose collection of modules and not even organized into packages in the first place[3] – the default protocol buffer package does not care at all how the message definitions are made available to the Python runtime.

2.1.6 Ubi-Interact

All things considered, the default protocol buffer package and plugin weren't a suitable solution for the Ubi-Interact use case where performance is not necessarily as important as simplicity. There are two useful third party plugins available to generate different Python code as protocol buffer implementation, as well as one package to generate type stubs which are compared in table 2.1.

[1] using this setup makes keeping track of topic data types unnecessary since the code is able to inspect the type of the *oneof* group through the reflection features of protocol buffers, if necessary

[2] at least for user defined types – primitive types need special treatment

[3] the plugin does not generate the appropriate `__init__.py` files to make the import mechanism recognize the folders as packages

Instead of the default implementation the Python protocol buffer distribution `ubii-message-formats` [64] for Ubi-Interact is using `proto-plus` [63] a Google package which provides wrappers around protocol buffer message types, but with the following adjustments:

1. custom compiler plugin available as `codestare-proto-plus` [65] to compile schema files to `proto-plus` code – including reStructuredText docstrings
2. adjusted metaclass and helper methods to make wrappers easily extendable
3. custom build tool plugin to automate compilation and pre-process schema files

Compiler plugin

The `proto-plus` package does not provide a plugin for the `protoc` compiler and instead defines message wrappers directly in Python code. There probably is a compiler used for Google projects that give the user the option to use `proto-plus` code – like the Google Ads API [66] – since those projects need to support existing message formats but it is no publicly available. Installing the `codestare-proto-plus` [65] module makes its plugin capabilities available for a compiler running in the same environment. It has been inspired by the `mypy-protobuf` package and corresponding compiler plugin but implements some additional features and currently lacks the support for RPC definitions in the schema files[1]. The additional features include support for docstrings which will be converted to appropriate reStructuredText (see example 2.2) and additional parameters that can be supplied to the plugin to automatically generate appropriate `__init__.py` files inside the generated directory structure to build packages instead of a loose collection of modules.

Through the use of `__init__.py` files, the module can support importing of message (wrapper) types according to their schema names: The plugin builds one module per schema input file, like the default plugin, but creates `__init__.py` files that `import` all types of the generated modules in the directory to make them in turn importable without knowing exactly which module specifies them. Since the generation of `__init__.py` files needs information about all types that need to be imported – which can only be deduced from the input schema files – this feature is limited to use cases where all necessary schema files are available (this is the exact same limitation the `better-proto` plugin has invariably, compare table 2.1), but can be turned off if the schema files should be compiled incrementally or in parallel. Through the use of the `proto-plus` "package" feature – which is sadly not very well documented currently – the `proto-plus` code can specify the message pool for the messages that are build internally. Messages with the same "package" definitions will be added to the same pool, which allows to incrementally build a mutual pool from multiple modules. This could for example be used to supply the Ubi-Interact messages in multiple packages, in order to allow clients

[1] this protocol buffer feature is not needed for Ubi-Interact

to choose which ones they need and only use the minimal message set to speed up (de-)serialization. This feature is also supported by the plugin through the optional "package" parameter, and documented in the `ubii-message-formats` documentation (for the lack of documentation by the `proto-plus` developers).

Notably, this does *not* fix the issues of the Python package structure mirroring the directory structure of input schema files since the plugin should be able to compile schema files to modules in a one-to-one manner to allow incremental builds. The only reasonable choice is then to mirror the structure – and since the directory structure of a package and its subpackages define the import path used by the Python import mechanism, it is impossible to respect the schemas "package" definitions for the generated Python package. The module level attribute one can see in l. 7 of example 2.2 is used for the aforementioned `proto-plus` "package" feature and uses the package `"my_package"` since it was explicitly passed to the plugin during compilation. The generated Python module would be named `color_plus.py` – since the input file is named `color.proto` – and an import statement to import the `Color` type would look like

```
from color_plus import Color
```

although the schema file defines the "package" as `my_package.dataStructure` (compare example 2.1.1), and the corresponding type would be imported in *schema* files as `my_package.dataStructure.Color` regardless of the directory layout.

Build Tools

To solve the problem of schema file directory structure, a tool was developed that is able to "fix" the directory structure of schema files by copying them to a tree that mirrors the package definitions inside the provided schema files (so a schema file that declares `package foo.bar;` would end up inside a `foo/bar/` directory). A CLI interface and a plugin for the `setuptools` build backend are available to integrate this process into the Python build process if necessary. For our use case the generated files should make up a subpackage of our protocol buffer package so that it can be replaced by updated versions whenever the schema changes. The package will also provide additional functionality as a separate `util` subpackage. This subpackage is not shown in fig. 2.4, and users don't need to be concerned with it, as all message wrappers are importable from the main `ubii.proto` package. This is specifically documented so that the `ubii-message-formats` package maintainer has more control over what users get when they want to import a message wrapper: in theory it would be able to switch to an implementation that does not use `proto-plus` (as long as the API is compatible) or provide several different implementations in parallel. Importing from the main package would then reference a certain default implementation, but if the Ubi-Interact framework e.g. decides to support *flatbuffers* the generated code could be distributed as an additional subpackage without problems).

Example 2.2: Python module compiled from `color.proto` using custom plugin – compare example 2.1.1

```python
"""
@generated by codestare-proto-plus.  Do not edit manually!
"""
import proto
import proto.message

__protobuf__ = proto.module(
    package="my_package",
    manifest={
        "Color",
    }
)


class Color(proto.message.Message):
    """
    4 Channel color using r,g,b and alpha channel

    Attributes:
        r (proto.fields.Field): :obj:`~proto.fields.Field` of type
            :obj:`~proto.primitives.ProtoType.DOUBLE`
        g (proto.fields.Field): :obj:`~proto.fields.Field` of type
            :obj:`~proto.primitives.ProtoType.DOUBLE`
        b (proto.fields.Field): :obj:`~proto.fields.Field` of type
            :obj:`~proto.primitives.ProtoType.DOUBLE`
        a (proto.fields.Field): :obj:`~proto.fields.Field` of type
            :obj:`~proto.primitives.ProtoType.DOUBLE`
    """

    r = proto.Field(
        proto.DOUBLE,
        number=1,
    )
    g = proto.Field(
        proto.DOUBLE,
        number=2,
    )
    b = proto.Field(
        proto.DOUBLE,
        number=3,
    )
    a = proto.Field(
        proto.DOUBLE,
        number=4,
    )
```

Package	Main Features	Advantages	Disadvantages
`google.protobuf` [37]	Compiler plugin to generate Python modules for schema files Interface for expected message API (compare section 2.1.2)	Simple in the sense that no third party packages are needed Designed in conjunction with other default implementations for different platforms for consistent API Well documented	Inconsistent naming No type hints when development of Ubi-Interact Python node started Needs custom wrappers for extensibility and easier JSON support Does not respect package declarations
`mypy-protobuf` [59]	Type stubs for default package	Makes default API better usable	– [1]
`betterproto` [61]	Reimplementation of protocol buffer framework from scratch Custom compiler plugin to generate Python *packages* Focus on idiomatic Python patterns	Native type checking Readable Python modules (messages are generated as simple *data classes*) Relative imports possible, declared packages are respected Python naming conventions are respected	Implementation from scratch leads to multiple open bugs (18 at the time of writing [62]) Multiple schema files are compiled into a single module i.e. all schema files need to be available during each compilation or the package breaks – new messages can't be easily added No fallback to original implementation possible i.e. critical bugs need to be fixed or package is unusable
`proto-plus` [63]	"Idiomatic" protocol buffer wrappers developed and used by Google Messages can be defined as simple Python classes for readability, package builds original descriptors under the hood and wraps them for better API Advanced marshaling features allow to specify rules for automatic marshaling of data send over the wire	Uses `google.protobuf` under the hood to build same types Features are available via a custom class that message wrappers inherit (similar to the design proposed in fig. 2.3) Classes that represent the messages are readable[3] API is more "pythonic" than standard package but one can always fall back to the default	No compiler plugin available Slightly slower than default implementation due to marshaling overhead Wrapper classes are not designed to be extendable per se[2] Limited typing support

[1] Generating the stubs should be part of the build process for the Ubi-Interact protocol buffer package which implies some development effort
[2] This can be fixed by refining the metaclass [3] Compare example 2.1.2 and example 2.2 to see the difference

Table 2.1: Comparison of protocol buffer Python packages

Additional Features

For use in Ubi-Interact, the generated message wrappers will be accompanied by a custom JSON encoder (implementing the interface of the `JSONEncoder` class in the `json` module of the standard library) which will be able to handle de- and encoding of the HTTP(S) traffic used for the synchronous "service" connection[1] (compare section 1.6.2) as well as a custom metaclass extending the metaclass used by the `proto-plus` package to build the message wrappers.

Figure 2.4 shows the basic layout of the `ubii.proto` package. The `ubii` namespace is used to group all Python modules related to Ubi-Interact, the framework and node implementation from section 2.3 are distributed as `ubii.framework` and `ubii.node` respectively. The `ubii.proto.util` module provides the JSONencoder and the extended metaclass which can be used to extend message wrappers, illustrated in example 2.3 where a class `CustomComponent` which acts like a wrapper around a Component protocol buffer message (since it inherits from `ubii.proto.Component`, the wrapper generated from schema files) is defined.
When defining a new proto-plus message wrapper, a *new* protocol buffer descriptor will be built under the hood – unless the `__protobuf__` attribute is defined in the module. Assigning the `__protobuf__` attribute of a module to the `ubii.proto.__protobuf__` attribute informs the metaclass mechanism where to look for existing message descriptors [67] for all wrappers built in the new module. This behavior is not very well documented in the `proto-plus` module (currently only in source code), and is a likely source of bugs since not setting the `__protobuf__` attribute will produce wrappers that serialize and deserialize the messages equivalently, but can't be used completely interchangeably. Therefore the documentation of the `ubii-message-formats` package covers this in more detail [68].

Since the custom metaclass also inherits the `abc.ABCMeta` class, it is able to build abstract classes. This mechanism now allows the Python framework to define abstract base classes that already implement the wrapping of a specific protocol buffer message – as a well defined interface for user types that need to be serializable in a certain way (recall that Python supports multiple inheritance instead of interfaces). For example, users can implement processing modules by inheriting from an abstract base class provided by the `ubii.framework` package, which is itself inheriting from the `ubii.proto.ProcessingModule` wrapper class (but built with the custom metaclass instead of the default metaclass of the `proto.message.Message` type). Defining the behavior of the processing module is then a matter of overwriting the same callback methods used for the JavaScript implementation, and marshaling as well as handling of the actual processing module objects in the Python framework comes "for free".

[1] the `proto-plus` wrappers also provide an easier interface for the JSON handling of the `google.protobuf.json_format`, module out of the box

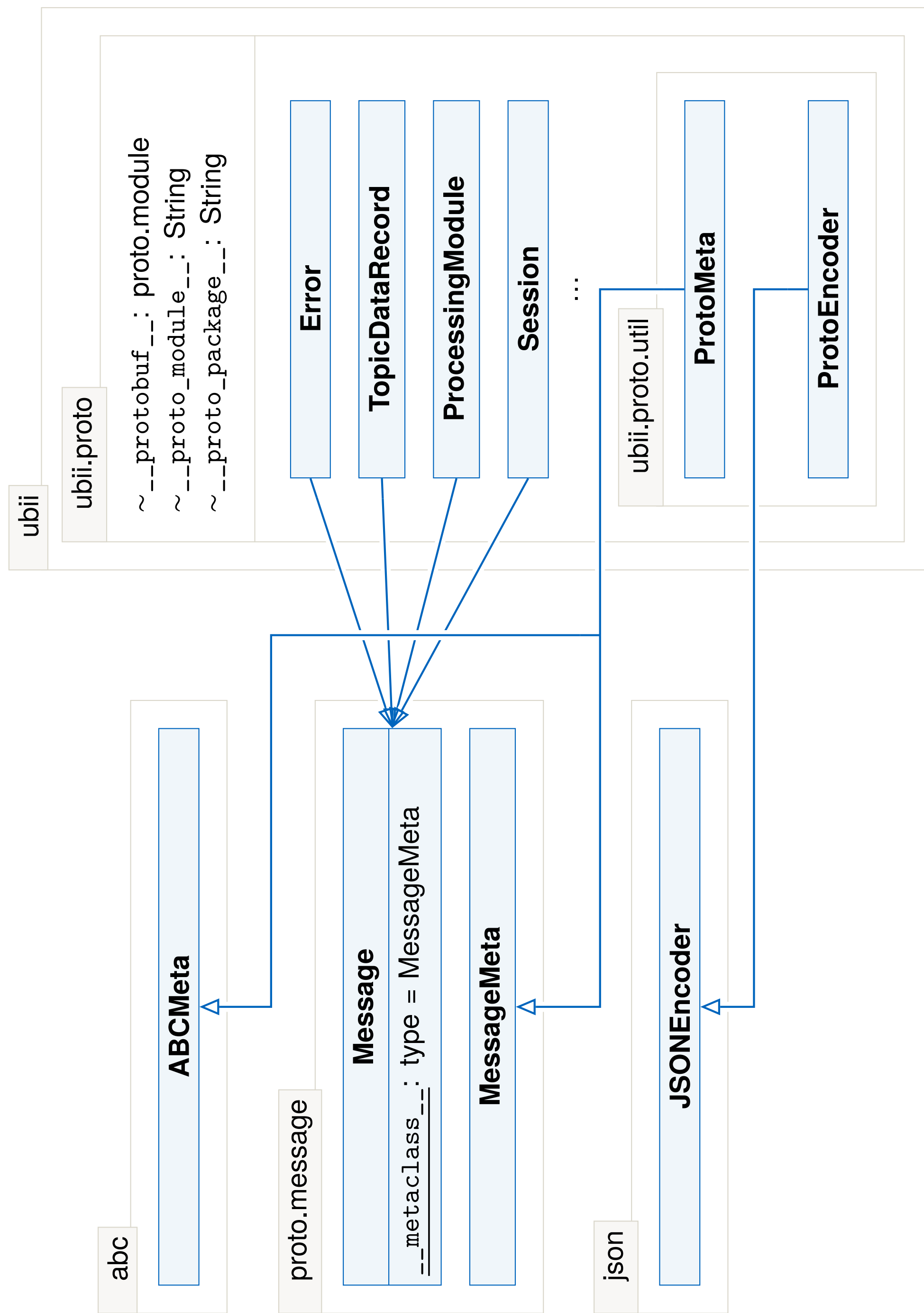

FIGURE 2.4: ubii-message-formats package

Example 2.3: Use of custom metaclass to extend a protocol buffer wrapper

```
import ubii.proto as ub

__protobuf__ = ub.__protobuf__

class CustomComponent(ub.Component, metaclass=ub.ProtoMeta):
    """
    This custom wrapper wraps the same message as ubii.proto.Component
    but can do something fancy
    """
    def __init__(mapping, *, fancy, **kwargs):
        super().__init__(mapping, **kwargs)
        self._fancy = fancy

    def fancy_method(self):
        return self._fancy
```

Type Hints

Type stubs generated for the `proto-plus` module have been generated and updated with some generics to improve typing support, more type hints can be added through the compiler plugin or the stubs in the future.
The stubs are distributed as `generic-proto-plus-stubs` [69].

2.2 Problems and Requirements

To respect the Separation of Concerns (SoC) principle, the framework needs to make sure that the interaction between behavior and representation of objects that are mappable to protocol buffer messages is designed in a flexible way. For example, the protocol buffer message shown in example 2.4 represents a client node. Some parts of that representation are a result of communication with the broker – for example a client does not have an `id` from the start, it will get it by using the *registration* service.
This means that how the local client representation is supposed to be kept up-to-date depends on implementation details: How does the client node know how it can communicate with the broker node in the first place? How does it know the message formats used in the *registration* communication? In a sense the answers to these questions describe the Ubi-Interact "middleware protocol" used for communication between the specific client and broker. It would be bad design if client code which only wants to deal with high level features like subscribing and publishing would need to change because the intricacies of the low level client broker communication change at some point. On the other hand, client code should be able to adapt the "middleware protocol" as easily

as all other parts of the node implementation, for example for nodes that should perform a special task – like running *processing modules*.
While all clients need to get an `id` at some point, the specifics of the way it is received should only concern the code that deals with the implementation of the "middleware protocol".

The API also needs to specify interfaces to execute common tasks – at least for performing service communication, subscribing to topics and publishing data – which should be accessible via an object that conceptualizes our client node. Treating the client node as the interface between the user and the Ubi-Interact framework is the standard in all currently existing target implementations. These interfaces need to be implemented against a specific "middleware protocol" though – see section 2.2.1.

Example 2.4: Protocol buffer message schema defining a Ubi-Interact client – `client.proto`

```
message Client {
    enum State {
        ACTIVE = 0;
        INACTIVE = 1;
        UNAVAILABLE = 2;
    }

    string id = 1;
    string name = 2;
    repeated ubii.devices.Device devices = 3;
    repeated string tags = 4;
    string description = 5;
    repeated ubii.processing.ProcessingModule processing_modules = 6;
    bool is_dedicated_processing_node = 7;
    string host_ip = 8;
    string metadata_json = 9;
    State state = 10;
    float latency = 11;
}
```

2.2.1 Middleware Protocol

To illustrate the design problem that needs to be solved, the sequence diagram in fig. 2.5 shows the communication between a client node and the broker until the clients minimal functionality can be guaranteed, i.e. it can

- use *services*
- *subscribe* to topics
- *publish* topic data

The basic communication consists of the two "Setup" steps in fig. 2.5. The "Processing" step shows the communication that is needed for nodes that want to run their own processing modules.

The "Async Setup" step can only happen *after* the "Sync Setup" step, since the client needs to have a unique *id* to use the asynchronous topic communication with the broker, therefore it needs to be able to use the *registration* service which will register the client node at the broker and return the unique *id* in its reply message (the updated `Client` message if the registration was successful).

The client node does not need to know how to use each *service* a priori, instead it can "ask" the broker – this happens in the "Sync Setup" step. The only communication that needs to be known a priori is the *server configuration* service. How brokers advertise this path is not part of the protocol – the standard for current nodes is to simply provide the broker nodes IP[1] and the (topic) path for the configuration service[2] when initializing the client.

After the *server configuration* service has been used to retrieve the brokers configuration (as a `Server` message), the client needs to use the contained *constants* – definitions, specific to this very broker, for the (topic) paths of possibly available services and other special topics[3] as well as data types. With this information, the client node is able to retrieve the *list of (advertised) services* and reevaluate or update its knowledge of synchronous communication – which it just learned via the *constants* – one final[4] time.

After the client "knows" how to use the synchronous service communication, has been registered and has established its dedicated asynchronous topic data connection it should be able to perform the basic publish and subscribe tasks. Requesting a subscription is possible via *service* communication, publishing and receiving messages happens asynchronously by sending/receiving `TopicData` protocol buffer messages for specific topics via the dedicated bi-directional *topic connection*.

[1] via configuration files, command line parameters and/or environment variables
[2] relying on the broker node documentation e.g. *Requests · SandroWeber/ubi-interact Wiki* [70]
[3] relevant for nodes that want to do more than just simple publish-subscribe communication
[4] in theory the services could change later – e.g. depending on the broker state – and would need to be updated

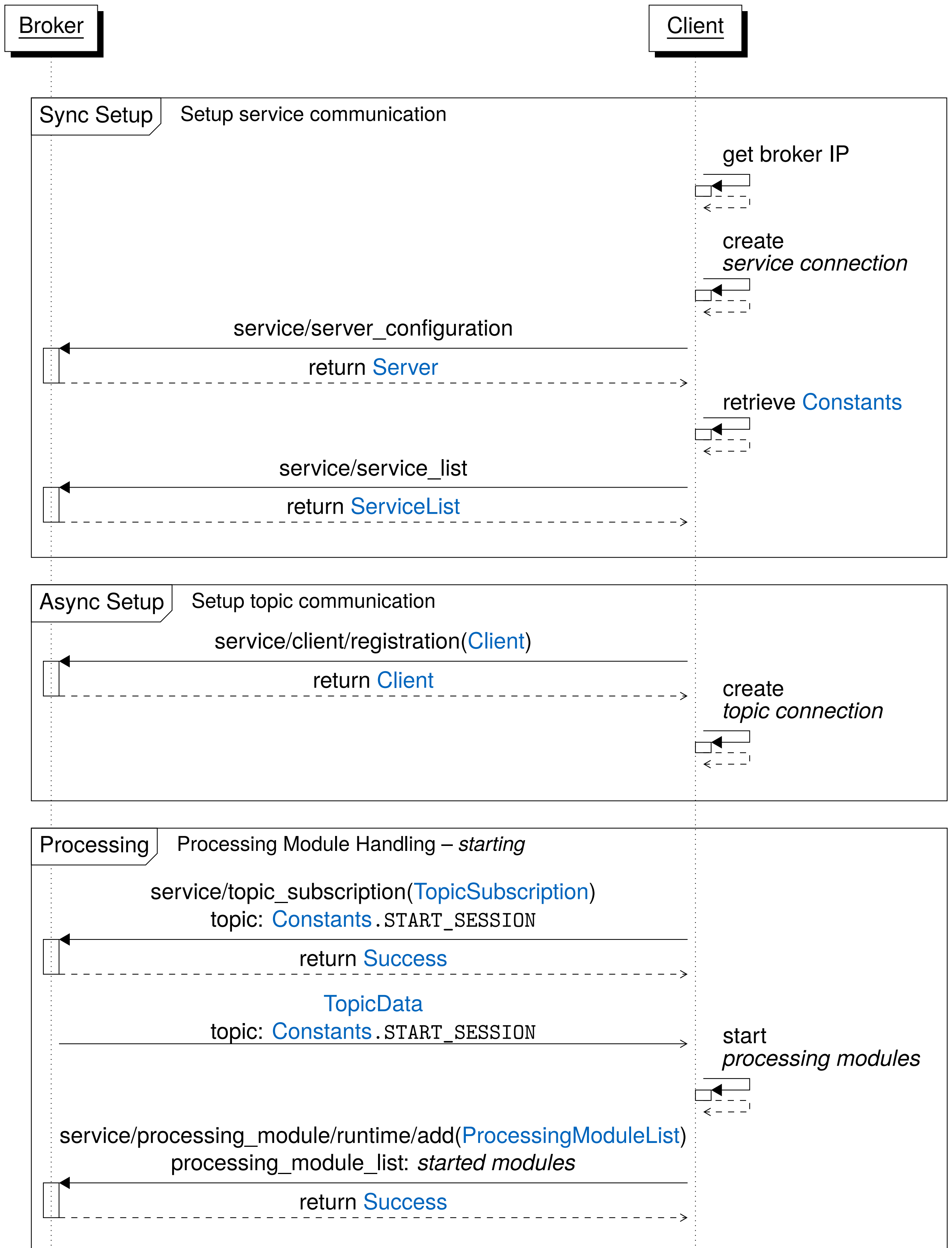

FIGURE 2.5: Ubi-Interact "middleware protocol"
Names in blue are protocol buffer messages
See *Requests · SandroWeber/ubi-interact Wiki* [70]

The following (competing) requirements and problems arise for the interface design:

1. What functionality the client exposes *should* be independent from its representation as a protocol buffer message which could e.g. become subject to change.
2. What functionality the client exposes *should* be independent from the "middleware protocol" which could e.g. also be subject to change.
3. How the functionality is implemented *is* inevitably strongly coupled to the "middleware protocol" implementation.
4. It *should* be clearly conceptualized – if possible in code – when a client is considered "usable", i.e. when a feature becomes available as part of the "middleware protocol".
5. Code using the interface to subscribe and publish, make service calls or use other features implemented on top of the "middleware protocol" *should* not have to deal with the protocol itself, it should be able to "start with a usable client".
6. The representation of the client needs to be kept up-to-date during the different stages of the "middleware protocol".
7. The interface *should* be "typed" so that type checkers or an IDE can provide additional support to use it correctly.

2.2.2 Processing Modules

Running *processing modules* is a very desirable feature for a Ubi-Interact node implementation targeting a previously unsupported environment. The node implementation needs to support a more complex "middleware protocol"[1] as seen in fig. 2.5. The information about the processing modules which the client is able to run needs to be available as part of the `Client` message sent during the *registration* in the "Async Setup" step (see the schema in example 2.4). After the registration it needs to subscribe to a specific topic (the exact topic-path used by the broker has been communicated during the "Sync Setup" step) to handle messages about new *sessions* (the *Session* concept will not be presented here, for more information refer to Weber *et al.* [1, § 5.5]). After the client node identifies that a *processing module* requested in a *session* can be provided by itself, it needs to start the processing, wire up inputs and outputs as specified, and inform the broker about the started modules.

In addition *processing modules* are prime examples of objects that have complex behavior which interacts with the representation as a protocol buffer message. The schema in

[1] for brevity the communication which is needed to stop *processing modules* is not shown

Example 2.5: Protocol buffer schema defining a Ubi-Interact processing module – processingModule.proto

```
message ProcessingModule {

    enum Status {
        INITIALIZED = 0;
        CREATED = 1;
        PROCESSING = 2;
        HALTED = 3;
        DESTROYED = 4;
    }

    enum Language {
        CPP = 0;
        PY = 1;
        JS = 2;
        CS = 3;
        JAVA = 4;
    }

    string id = 1;
    string name = 2;
    repeated string authors = 3;
    repeated string tags = 4;
    string description = 5;
    string node_id = 6;
    string session_id = 7;

    Status status = 8;
    ProcessingMode processing_mode = 9;
    repeated ModuleIO inputs = 10;
    repeated ModuleIO outputs = 11;
    Language language = 12;

    string on_processing_stringified = 13;
    string on_created_stringified = 14;
    string on_halted_stringified = 15;
    string on_destroyed_stringified = 16;
}
```

example 2.5, for example, shows that a module has a `status` which represents the processing state (the associated state machine is shown in Weber *et al.* [1, fig. 3]). However the processing behavior is implemented, it needs to have an associated representation that is up-to-date, just like a *Client* needs to be updated during the different stages of the associated "middleware protocol".

2.2.3 Python Language Features

To discuss the design for these requirements, a short overview over some language features of Python is justified.

Multiple Inheritance

Many arguments can be made against – but also in favor of – multiple inheritance in Python the main takeaway from the commonly referenced articles on the matter *Python's Super Considered Harmful* [71] and *Python's super() considered super!* [72] is that *composition over inheritance* [73] is a useful principle/guideline – also for Python code – but *when it's done right* there is nothing "scary" or "harmful" in using multiple inheritance in Python. Many mistakes Python developers make often come down to simply using features like they were used to – before migrating to Python – instead of in the intended way. Multiple inheritance is no different, it relies on the correct use of Python's `super()` callable to delegate method calls. Saying that "one big problem with 'super' is that it sounds like it will cause the superclass's copy of the method to be called. This is simply not the case, it causes the next method in the MRO[1]to be called" [71] is like saying that English is a badly designed language because Germans tend to confuse the meaning of "to become" and "to get".

Dependency Injection

So that two objects, one which provides some functionality (referred to as *service*) and another which wants to use that functionality (referred to as a *client*), respect the SoC principle, the design needs to make sure that the client does not need to know how to construct the service. As a concrete example, the Ubi-Interact client node might be the *client* in this context, and the *service* is an object that somehow implements a topic connection. The node wants to use that connection to send and receive topic data, but it would be desirable if the actual connection implementation could be changed later[2], without affecting the client. One way to design this (especially in statically typed languages,

[1] the Method Resolution Order (MRO) is an attribute of Python types that defines in which order parents and siblings are searched when a call should be delegated

[2] e.g. for testing it would be nice to use a mocked connection which "sends" some test data

where there is more cost – e.g. additional compilation – associated with "dynamically" changing parts of the code) is commonly referred to as *dependency injection* or *inversion of control*, basically the client is provided with the service by some external code – the *injector* – which it is not aware of [74].

Example 2.6: Naive dependency injection
Obvious flaws discussed in section 2.2.3

```python
class Connection:
    """A service"""
    def __init__(server):
        self.server = server
        self.is_open = False
    def open(self):
        print(f'Opening connection to {self.server}')
        self.is_open = True
    def close(self):
        print(f'Closing connection to {self.server}')
        self.is_open = False
    def send(self, message):
        if self.is_open:
            print(f"Sending {message} to {self.server}")

class Client:
    """A client"""
    def publish(self, connection, message):
        connection.send(message)
```

```
>>> client = Client()
>>> connection = Connection('test server')
>>> connection.open()
    Opening connection to test server
>>> client.publish(connection, "Foo")
    Sending Foo to test server
>>> connection.close()
    Closing connection to test server
```

On the one hand Python code in the standard library often uses dependency injection, on the other hand *dedicated frameworks* are used very infrequently. But why is that the case?
The naive implementation in example 2.6 explicitly injects the dependency on the service in the functionality of the client. This has some obvious drawbacks:

1. Client code needs a connection object whenever it wants to make a `publish` call.
2. A connection with some special interface is passed around, but the `publish` method only depends on the `send` functionality.

It seems impossible to "hide" this complexity without creating a tightly coupled client and service, on first glance. For example, although it looks like the client does not *create* the connection in example 2.7, in fact it uses the special `open` interface and becomes tightly coupled to the connection implementation. This actually is the anti-pattern *inversion of control* aims to avoid.

Example 2.7: Bad dependency injection
anti-pattern which leads to tight coupling

```
class Connection:
    ... // same as before

class Client:
    def __init__(connection):
        self.connection = connection
        self.connection.open()

    def publish(self, message):
        self.connection.send(message)

    def close(self):
        self.connection.close()

connection = Connection('test server')
client = Client(connection)
client.publish("Foo")
client.close()
```

In Python "everything is an object"[75] – this allows developers to design the same scenario that was just discussed like shown in example 2.8. Client code still needs a usable connection, which is now just any callable that can send the message, whenever it wants to use the publish functionality. There is no need for special dependency injection frameworks, because the nominal "type" of the callable that is used to send the message does not matter in a dynamically and "duck" typed language like Python. Python allows to use callables with state, consequently one can cleanly separate the call *arguments* and *context*. In example 2.8 `connect` instances are callable, but can also be used as *context managers* (this provides a much cleaner interface for IO operations). This is not part of the modeled dependency, though – the client does not need to know if he is using a complex IO connection or a mocked test connection. Python even allows to specify the expected call signatures with generics, type hints or *static duck typing*[55] – i.e the `publish` functionality in the client can specify exactly how the provided duck should "quack".

Example 2.8: Python dependency injection pattern
No special interface required – compare with example 2.6

```
class connect:
    def __init__(self, server):
        self.server = server
        self.is_open = False
    def __enter__(self):
        print(f'Opening connection to {self.server}')
        self.is_open = True
        return self
    def __exit__(self, *exc_info):
        print(f'Closing connection to {self.server}')
        self.is_open = False
    def __call__(self, message):
        if self.is_open:
            print(f"Sending {message} to {self.server}")

class Client:
    def publish(self, send, message):
        send(message)
```

```
>>> client = Client()
>>> with connect('test server') as connection:
...     client.publish(connection, "Foo")
    Opening connection to test server
    Sending Foo to test server
    Closing connection to test server
>>> client.publish(print, "Bar")
    Bar
```

To take this to the extreme, the `publish` functionality itself could become an instance of a special callable type, like in example 2.9. The `MessageConsumer` type models the exact call signature that is expected (i.e. one positional argument named `message` which is a *proto-plus* wrapper, no return value), which means an IDE will be able to deduce that a callable with appropriate signature is accessible via the clients `publish` attribute. This code is as flexible as possible, it would even be possible to change the `publish` implementation of one specific client instance at runtime. The connection is provided like in example 2.8, except that instead of making connect a class that implements the *context-manager* "interface", the code in example 2.9 creates an equivalent context manager[1] in half the lines of code using the `contextlib` module from the Python standard library – which again shows the immense flexibility of this design.

[1] except that since we are making promises about types by using *type hints* the returned callable can reasonably use the special *proto-plus* API to serialize the arguments it receives

Example 2.9: Python dependency injection with static duck typing
Application of the pattern from example 2.8 everywhere

```python
from typing import Protocol
from proto.message import Message
from ubii.proto import Error, Success
from contextlib import contextmanager

class MessageConsumer(Protocol):
    def __call__(self, message: Message) -> None: ...

class Client:
    class Publish:
        transport: MessageConsumer | None
        def __init__(self):
            self.transport = None

        def __call__(self, message: Message):
            if self.transport is not None:
                self.transport(message)

    def __init__(self):
        self.publish: MessageConsumer = self.Publish()

@contextmanager
def connect(server) -> MessageConsumer:
    def send(message: Message) -> None:
        payload = type(message).serialize(message)  # we promised the right type!
        print(f'Sending {payload} to {server}')

    print(f'Opening connection to {server}')
    yield send
    print(f'Closing connection to {server}')
```

```
>>> client = Client()
>>> with connect('test server') as connection:
...     client.publish.transport = connection
...     client.publish(Error(title='Foo'))
...     client.publish(Success(title='Bar'))
    Opening connection to test server
    Sending b'\n\x03Foo' to test server
    Sending b'\n\x03Bar' to test server
    Closing connection to test server
```

To allow easier "functional" design, some utility callables which extend the `functools` module from the standard library can be found in the `ubii.framework.util.functools` module [76].

Asyncio

To write modern asynchronous code in Python, the standard library provides developers with the `asyncio` framework. Users can declare coroutines with the `async` keyword, which can be scheduled to be executed when they are *await*-ed or run concurrently as *tasks*. The internal handling of coroutines in the `asyncio` *event loop* does not need to be discussed in detail, but it is important to note that "special" callables like they are used in example 2.8 are not simply convertible to a coroutine by adding the `async` keyword in the right place. Normally, coroutines are created by declaring a function like

```
1  async def foo():
2      return 'foo'
```

which in this case creates a function that returns a coroutine that can be later awaited to yield "foo".

The coroutines implement a special interface consisting of four methods: `__await__`, `send`, `throw` and `close` [77]. Building complex coroutines out of smaller sub-coroutines can be done analogously to building complex callables by creating a special class that implements the interface. On one hand, the "functional" design has a lot of benefits, as was shown in the previous section, and creating special classes to represent complex coroutines – instead of creating them with `async def` functions – could be used frequently. On the other hand, the `async def` declaration is easier to read and implement. For this reason the Ubi-Interact Python framework uses a coroutine wrapper class to implement complex coroutines in the same style as complex callables, detailed documentation and examples are available[78]. Example 2.10 illustrates the usage by defining a small `processing_steps` wrapper around some coroutine which is broken down into smaller pieces of functionality. A similar result could be achieved by using nested functions, but this design allows to make the state of the coroutine (in our example the `steps` it performs) public and mutable – similar designs are used throughout the codebase of the Ubi-Interact Python framework to model the functionality of some objects as mutable and allow code in different places to make small adjustments to this functionality. Since instances of `processing_steps` *are* considered coroutines, they can run in background tasks without an issue.

Example 2.10: Usage of custom asyncio coroutine wrapper

```python
import asyncio
from codestare.async_utils import CoroutineWrapper

class processing_steps(CoroutineWrapper):
    def __init__(self, values, *, steps=None):
        self.values = values
        self.steps = list(steps) or []
        super().__init__(coroutine=self.work())

    def process(self, value):
        for step in self.steps:
            value = step(value)
        return value

    async def work(self):
        async for value in self.values:
            self.process(value)

async def main():
    processing = processing_steps(range(3), steps=[print])
    # or maybe more processing ?
    processing.steps = [lambda x: x + 1] + processing.steps
    await asyncio.create_task(processing)
```

```
>>> asyncio.run(main())
    1
    2
    3
    4
```

All utility tools for async development are published in a separate distribution, `codestare-async-utils` [79], which is documented as part of the Ubi-Interact Python node documentation. The `codestare.async_utils.wrapper` module provides the coroutine wrapper, while the `codestare.async_utils.nursery` module deals with handling of asyncio background tasks: Tasks can be managed by a `TaskNursery` (multiple nurseries can be instantiated for one asyncio event loop), which sets up the right callbacks to catch exceptions and handle system signals (on Windows systems only `SIGBREAK` and `SIGINT` are usable to catch keyboard interrupts). Task nurseries implement the `contextlib.AsyncExitStack` [80] interface which basically allows them to perform arbitrary async teardown code when exceptions occur in managed tasks[1].

[1] for more information refer to *codestare.async_utils.nursery module — ubii-node-python documentation* [81] and *contextlib — AsyncExitStack — Python 3 Documentation* [80]

2.3 Design

The `ubii-node-python` distribution entails three Python packages for the `ubii` namespace:

- `ubii.node` – Ubi-Interact client node implementation in Pythonand `pytest` plugin
- `ubii.cli` – CLI for the Ubi-Interact node
- `ubii.framework` – base framework that the node implementation is built on

The design of the packages is discussed here, while section 2.2 in general and specifically the discussion of language features from section 2.2.3 deal with the particularities that necessitate specific decisions.

2.3.1 "Protocol" concept

The word "protocol" is used throughout the discussions in section 2.2 to describe a series of predefined stages or steps as building blocks for complex behavior of an object – namely the "middleware protocol", which defines how a client communicates with a broker, and the state machine of a *processing module*.

As discussed in section 2.2.3, implementation of complex behavior can be broken down using a combination of language features and special utility classes. As a result, the Ubi-Interact Python framework uses the concept of a "protocol" to describe object behavior which is defined as a state machine. This makes the node's communication with the broker a *protocol* as well as the processing behavior of a processing module – and allows to cleanly separate the behavior for the respective objects from their representation as a protocol buffer message and other public interfaces.
A *protocol* defines the states of the state machine it models and callbacks for possible state transitions (callables which return coroutines or other awaitables, e.g. by using `async def`). Associated with a *protocol* is a special coroutine which handles running an instance of a concrete *protocol* implementation. It can run in a background task which is by default managed by the *protocol* itself and can be started and stopped using the corresponding `start` and `stop` operations.

The `context` of a *protocol* is mutable state that is passed as input to all defined callbacks (by default it is just a namespace, but concrete implementations can overwrite the `context` property to e.g. return a data class which can use type hints so that the callbacks know which elements they can expect in the state they get as input).

A concrete implementation of a *protocol* can be found in the `processing` module of the framework, and a further specialisation – but still abstract – which gives more structure to the expected client broker communication can be found in the `client` module of the framework – a concrete client protocol implementation is available from the `ubii.node.protocol` module. Refer to the diagram in fig. 2.6 for an overview of the relationships, and to example 2.11 for a minimal implementation.

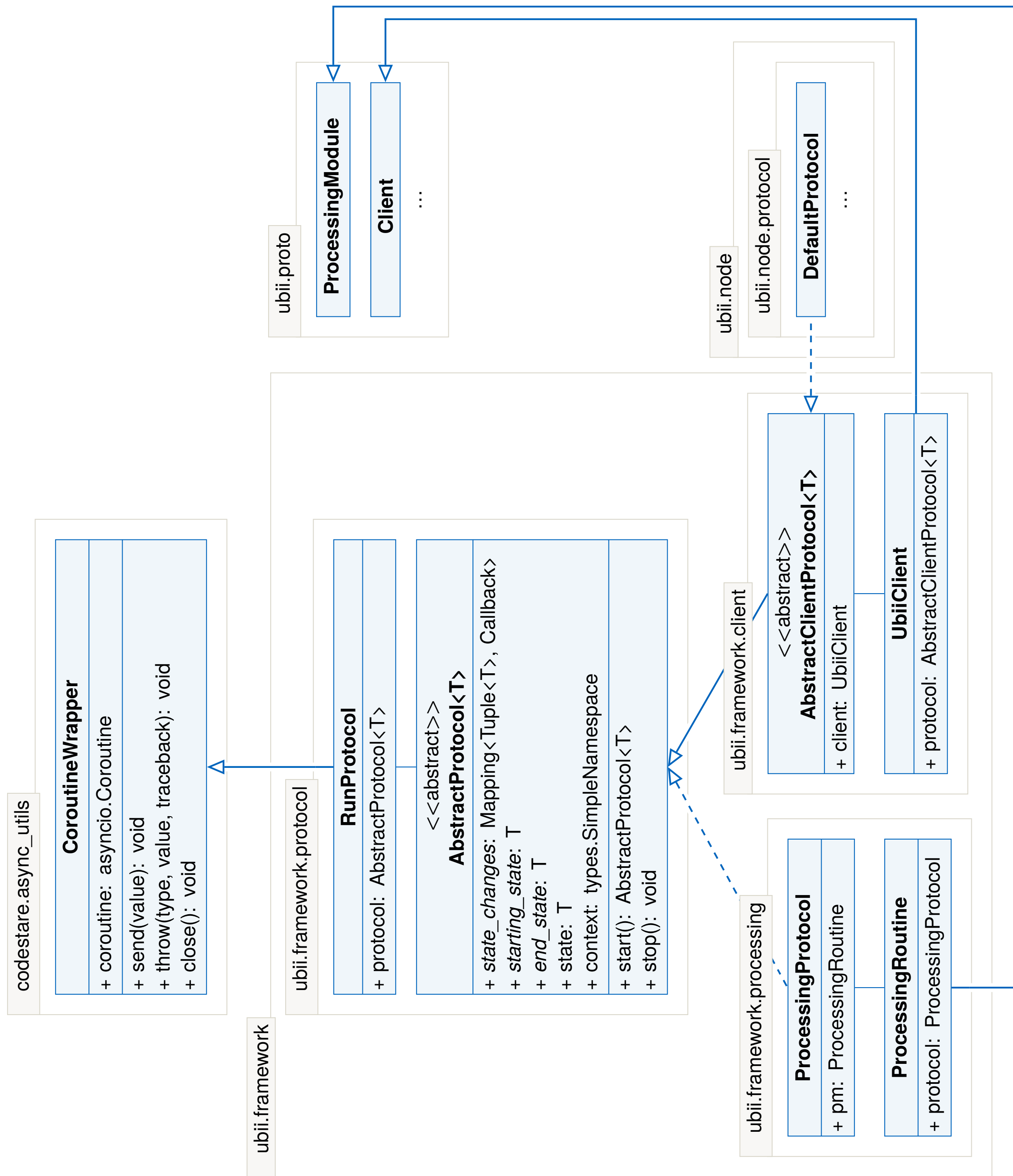

FIGURE 2.6: Design of object behavior as state machine instances – some attributes and dependencies are omitted for brevity[a]

[a]e.g. attributes of protocol buffer wrappers

Example 2.11: Minimal *protocol* usable with Ubi-Interact Python framework

```python
import enum
import asyncio
from ubii.framework.protocol import AbstractProtocol

class TestProtocol(AbstractProtocol):
    class TestStates(enum.IntFlag):
        START = enum.auto()
        RUNNING = enum.auto()
        END = enum.auto()
        ANY = START | RUNNING | END

    async def on_start(self, context):
        print(f"starting with context:\n-> {context}")
        await asyncio.sleep(2) # simulating some setup IO ...
        await self.state.set(self.TestStates.RUNNING)

    async def on_run(self, context):
        print(f"running with context:\n-> {context}")

    async def on_stop(self, context):
        print(f"stopping with context:\n-> {context}")

    starting_state = TestStates.START
    end_state = TestStates.END

    state_changes = {
        (None, TestStates.START): on_start,
        (TestStates.START, TestStates.RUNNING): on_run,
        (TestStates.ANY, TestStates.END): on_stop,
    }


async def main():
    protocol = TestProtocol()
    async with protocol as started_protocol:
        state = await started_protocol.state.get(
            predicate=lambda s: s != started_protocol.TestStates.START
        )
        print(f"Now in state {state!r}")
```

```
>>> asyncio.run(main())
starting with context:
-> namespace(state_change=(None, <TestStates.START: 1>))
running with context:
-> namespace(state_change=(<TestStates.START: 1>, <TestStates.RUNNING: 2>))
Now in state <TestStates.RUNNING: 2>
stopping with context:
-> namespace(state_change=(<TestStates.RUNNING: 2>, <TestStates.END: 4>))
```

2.3.2 Client

The design of client nodes shouldn't involve lots of `UbiiClient` subclasses for different client behaviors, instead – as just discussed – the runtime behavior, the client *protocol*, should be part of the public interface. In fact, a client node is simply defined by its representation as a `ubii.client.Client` protocol buffer message, its behavior in terms of the used client *protocol* and the public interface it exposes to users to execute certain tasks in the Ubi-Interact environment, like e.g. subscribing and publishing. The representation as a protocol buffer message is "free" if the node inherits the functionality of the corresponding message wrapper, which is possible by means of a custom meta class that was introduced in section 2.1.6.

The interface that the client offers should make use of type hints, as already mentioned in section 2.2 they are a very useful feature in modern Python code. Even if the interface is dynamically implemented by the client *protocol*, which will be the case since the possible implementations of features – like subscribing to topics and publishing data – are dependent on the concrete client *protocol* that is used to communicate with the broker, the type hints will help end users to write application code that is agnostic to the *protocol* internals and concrete implementation of the interfaces. Also, since the interfaces are partly asynchronous, the node instances need to be integrated into async code in a way that application coroutines can wait for them to be usable – without dealing with the internals of the nodes "client protocol"[1] on which they depend.

"Behavior" concept

To implement a set of *typed*, *dynamic* attributes of node instances that become available at some point during the *protocol*, the `UbiiClient` makes use of "data classes" [82]. Small sets of related interfaces can be defined like shown in example 2.12, the Ubi-Interact Python framework refers to these sets of interfaces as *behaviors*, while the runtime behavior is referred to as the client *protocol*. The dataclasses are not limited to using callables, a behavior can be modeled via a "standard" attribute as well. For example the support for service calls in example 2.12 is modeled as access to a `sercice_map` attribute. If the callable interfaces are defined with Python's static duck typing support, the type checker or IDE can even deduce the correct argument names – see fig. 2.7.

Client instances receive two sets of *behaviors* on initialization, the *required behaviors* and the *optional behaviors*, by default required behaviors include only access to service calls, subscription handling and publishing of data while optional behaviors include registering of devices, handling of sessions, handling of processing modules and de-registering/re-registering the client.
Default behaviors are defined in the `ubii.framework.client` module, but client instances could choose to support completely different sets of behaviors. The client then

[1] refer to fig. 2.5 once again for a schematic overview

Example 2.12: Client interface definition using dataclasses and static duck typing[a]

```python
from dataclasses import dataclass
from typing import Tuple, Awaitable, Protocol as DuckTypingProtocol
from ubii.framework.topics import Topic
from ubii.framework.services import ServiceMap
from ubii.proto import TopicDataRecord

class subscribe_call(DuckTypingProtocol):
    def __call__(self, *pattern: str) -> Awaitable[Tuple[Topic, ...]]:
        """
        This defines the exact signature we expect for the subscribe interface
        """

class publish_call(DuckTypingProtocol):
    def __call__(self, *records: TopicDataRecord | dict) -> Awaitable[None]:
        """
        This defines the exact signature we expect for the publish interface
        """

@dataclass(init=True, repr=True, eq=True)
class Subscriptions:
    """
    Behavior to subscribe to topics, defines two methods
    """
    subscribe_regex: subscribe_call | None = None
    subscribe_topic: subscribe_call | None = None

@dataclass(init=True, repr=True, eq=True)
class Publish:
    """
    Behavior to publish `TopicDataRecord` messages.
    """
    publish: publish_call | None = None
    """
    await to publish topic data
    """

@dataclass(init=True, repr=True, eq=True)
class Services:
    """
    Behavior to make service calls (accessed via the service map)
    """
    service_map: ServiceMap | None = None
```

[a] The `Protocol` type is used for Python's static duck typing. (It is not related to the *protocol* concept discussed in section 2.3.1 and imported with an alias to avoid confusion.) Imports in the example are intentionally explicit to reduce noise in the actual code.

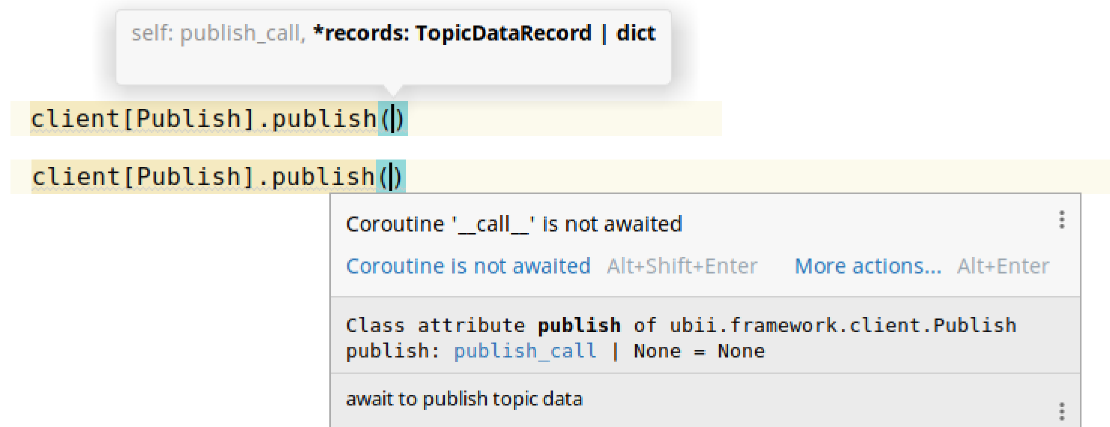

Figure 2.7: IDE hints with statically duck typed *behavior*

defines a specific interface to access these behaviors and makes use of asyncio synchronization primitives to notify waiting coroutines when behaviors become available. Users can implicitly wait for all *required* behaviors to be implemented by awaiting the client instance or await specific behaviors individually. The interface is shown in example 2.13 – assume the commands are executed in the experimental asyncio REPL[1].

Implementing a behavior is simply done by assigning to all fields of the corresponding dataclass. This may happen automatically as part of the *protocol* (e.g. all required behaviors should be *implemented* at some point, so users can just `await` the client), or require specific actions from the user, in which case all code should use the possibility to check the implementation status to handle the case of unsupported behaviors.

Figure 2.7 shows how an IDE can help users to correctly use the behaviors: with a `Publish` behavior defined exactly like in example 2.12, the IDE can e.g. show the argument names and notify the user that the call should probably be used in an `await` statement.

The combination of these features and Python's possibility to use sensible defaults for all dynamic attributes makes the overall interface as flexible as possible but if necessary also strict enough to guide the end user.

For detailed documentation refer to the documentation of the `ubii` namespace package [83] and the `ubii.framework.client` module [84] in particular.

[1]available since Python 3.8 – currently undocumented. Allows to use `await` statements directly in the REPL

2.3.3 Processing Modules

Processing Modules are implemented based on the *protocol* concept. As shown in fig. 2.6 the `ProcessingRoutine`[1] type implements the marshaling with respect to `ubii.processing.ProcessingModule` protocol buffer messages by inheriting the wrapper behavior from the `ubii.proto` module wrapper type.
Additionally, it implements runtime behavior through the public `protocol` attribute referencing a `ProcessingProtocol` instance. Processing module protocols deal with setting up the necessary `asyncio` primitives to handle their complex runtime behavior. A special `Scheduler` coroutine (using the coroutine wrapper discussed in section 2.2.3) handles the effects of the modules *processing mode* on its behavior – for more information about processing modes refer to [1, § 5.5].
The complex "behind the scenes" behavior of the module is separated from the public callbacks of the `ProcessingRoutine` type, which mirror the callbacks used in JavaScript code.
To allow easier portability between callback based JavaScript code and async Python code, the public callbacks of the `ProcessingRoutine` are implemented as normal callables that don't make use of the asyncio framework.
They are integrated into the async Python code by the `Scheduler` and corresponding async methods in the `ProcessingProtocol`. The protocol implementation supports handling of Topic Muxers and Demuxers and hot-swapping inputs and outputs of running modules by making clever use of custom "stateful" callables (discussed in section 2.2.3).

For detailed documentation refer to the documentation of the `ubii` namespace package [83] and the `ubii.framework.processing` module [85] in particular, as well as the tutorial on processing modules [86].

2.3.4 Node

The `ubii.node` package deals with implementing a suitable client *protocol* and provides the end user with a working Ubi-Interact node instance. Two versions of client protocols have been developed, also to provide an example how to extend the existing protocols. The `LegacyProtocol` deals with the minimum of possible communication, and can also instantiate and run processing modules which don't rely on client state for initialization, i.e. they are fully describable solely by the corresponding protocol buffer message at the time of client initialization.

It is not unusual though that processing modules rely on information exchanged as part of the client-broker communication for their own initialization. As shown in fig. 2.5 and the corresponding part in section 2.2 the client node itself starts without knowledge about the details of the client-broker communication, particularly without exact knowledge of the message types the broker uses. The brokers data type definitions are passed as part of the `ubii.server.Server` message which indicates a successful "synchronous

[1] the type is not called "processing module" to distinguish that it also has behavior

setup".
Processing modules that want to define message formats for their inputs and outputs "dynamically" (i.e. use the data type definitions of the specific broker handling their data) therefore need to be initialized *late* during the client protocol execution, specifically after the client has received the brokers data type definitions. Processing modules need to be instantiated before the client registers itself though, since the processing module representations need to be included in the correct client representation that is sent to the broker for registration (refer to example 2.4 for details on the client message specification).

Since the constants that contain these definitions are a protocol buffer message, i.e. their structure is known, the modules can be implemented against abstract constants (e.g. using a data type like `constants.MSG_TYPES.DATASTRUCTURE_IMAGE` for a not yet specified `constants` message they can access during initialization).
Handling processing modules with a more complex initialization is a new client *behavior* which is supported by a new client protocol.

To *implement* the behavior – using the terminology from section 2.3.2 – the user can provide a mapping from module names to module factories (i.e. callables returning `ProcessingRoutine` instances). If this is the case, the new protocol will use those callables to create the modules before the client is registered, so that they are correctly contained in the client message which is sent to the broker when the client registers itself.

The creation of clients with the right set of behaviors and protocols is handled by the `ubii.node.connect` module which defines a special callable, used as an interface to create functional Ubi-Interact nodes. Example 2.15 deals with possible uses of this object which can be used as a *callable*, an *awaitable*, an *async context manager* and a normal *context manager*, similar to a `UbiiClient` or a protocol, to support use-cases of different complexity – from using a Python node to simply publish and subscribe, to cases where a specific client setup is necessary.

For detailed documentation refer to the documentation of the `ubii` namespace package [83] and the `ubii.node` package [87] in particular.

Example 2.13: Client interface implementing *behavior* with Test-Protocol from example 2.11

```
@dataclass(init=True, repr=True, eq=True)
class FooBehavior:
    foo: Callable[[str], None] | None = None

@dataclass(init=True, repr=True, eq=True)
class BarBehavior:
    bar: str | None = None
```

- Two very simple behaviors: one defines a callable `foo` that takes a string as argument and returns nothing, the other defines an attribute `bar` which is just a string.

```
class TestProtocol(AbstractProtocol):
    ...

class FakeClientProtocol(TestProtocol):
    TestStates = TestProtocol.TestStates

    def __init__(self):
        self.client: UbiiClient | None = None

    async def on_run(self, context):
        self.client[FooBehavior].foo = print

    state_changes = {
        **TestProtocol.state_changes,
        (TestStates.START, TestStates.RUNNING): on_run
    }
```

- Create an extension of the `TestProtocol` defined in example 2.11 – note how the behavior of a client is accessed with `[]` access.
- The protocol *implements* the behavior simply by assigning to the attribute.
- A behavior is automatically considered *implemented* when all attributes have been assigned. The `state_changes` have to be updated as well to use the new callback.
- Protocols should pass the relevant data between steps as part of the context if possible – for simplicity the code above simply uses an instance attribute to access the client instead.

Example 2.13: Client interface implementing behavior with Test-Protocol
– continued from p. 52

```
24 protocol = FakeClientProtocol()
25 client = UbiiClient(
26     name='Test Client',  # wrapped `ubii.client.Client` message field
27     required_behaviors=(FooBehavior,),
28     optional_behaviors=(BarBehavior,),
29     protocol=protocol
30 )
31 protocol.client = client
```

- The developer has full control over which behaviors the client can implement, and how the protocol achieves this.
- The developer of a node needs to make sure that the client and its protocol are connected appropriately – e.g. the `FakeClientProtocol` needs to have its `client` field assigned.

```
>>> await client
>>> assert client.implements(FooBehavior)
```

- To implicitly wait for all required behaviors the user can simply `await` the client instance.
- Awaiting the client implicitly starts the protocol if it hasn't been started.
- After the client has successfully been *awaited*, all required behaviors will be implemented.

```
>>> async with client as started_client:
...     assert started_client.implements(FooBehavior)
```

- The client instance could also be used as an async context manager, just like a protocol, in fact using the client this way simply wraps the protocol context manager and stops the protocol when the context is exited.

```
>>> await client.implements(FooBehavior, BarBehavior)
```

- The `implements` interface can also be used in `await` expressions, to explicitly wait for the client behaviors to be implemented.

Example 2.14: Client interface implementing behavior with Test-Protocol
– continued from p. 53

```
32 async def use_behaviors():
33     await client.implements(FooBehavior, BarBehavior)
34     client[FooBehavior].foo(
35         f"Using behaviors -> {client[BarBehavior].bar}"
36     )
37 async def implement_bar():
38     await client[BarBehavior].bar = 'Bar'
39 async def main():
40     async with client as running_client:
41         assert not running_client.implements(BarBehavior)
42         await asyncio.gather(use_behaviors(), implement_bar())
```

```
>>> asyncio.run(main())
    starting with context:
    -> namespace(state_change=(None, <TestStates.START: 1>))
    Using behaviors -> Bar
    stopping with context:
    -> namespace(state_change=(<TestStates.RUNNING: 2>, <TestStates.END: 4>))
```

- The example code starts two tasks in the background: one implements the optional `BarBehavior`, the other uses it – together with the required `FooBehavior`.
- Code can use the behaviors without knowing when they are implemented by simply awaiting their implementation.

Example 2.15: Interface to instantiate Ubi-Interact node

```
>>> from ubii.node import connect_client
>>> client = await connect_client('http://localhost:8102/services/json')
>>> print(client.protocol.state)
    <States.CONNECTED: 8>
```

- The connect interface can be used in an await expression.
- This creates a client with protocol according to the connect call.
- The interface takes optional arguments to instantiate specific clients with specific protocols.
- Using the interface this way awaits the client, i.e. the node is implicitly started.
- Stopping the client is not handled implicitly.

```
>>> with connect_client() as client:
...     client.is_dedicated_processing_node = True
...     await client
```

- The connect interface can be used as a normal context manager in a with expression.
- This creates a client with protocol when it is entered and tries to stop the client protocol when it is exited.
- This can be used if changes to the client need to be done before it is started, e.g. if processing modules need to be assigned.
- Starting the client is not handled implicitly.

```
>>> async with connect_client() as running_client:
...     print(running_client.protocol.state)
    <States.CONNECTED: 8>
```

- The connect interface can be used as an async context manager in an async with expression.
- This creates a client with protocol when it is entered, awaits it and stops the protocol when it is exited.
- Starting and stopping the client is handled implicitly.

2.3.5 CLI

A CLI for a client node – basically a minimal example for a script using the node implementation from `ubii.node` – is available from the `ubii.cli` package, and as an automatically installed console script entry point for the `ubii-python-node` distribution[1]. The CLI also auto-discovers installed Python processing modules – if developers of processing modules make them available through an *entry point* in a specific group (currently `ubii.processing_modules`, documented as part of the `cli` package [89]). An example of an auto-discoverable processing module is described in section 2.3.6. A minimal client without advanced module handling is implemented in *Getting started — Example Client — ubii-node-python documentation* [90].

2.3.6 Implementation Details

The default connections of the Python node are implemented using the `aiohttp` library, which supports WebSockets and HTTP(S) requests out of the box and uses Python's `asyncio` framework. A client protocol using `aiohttp` primitives as basis for service and topic connections is implemented in the `ubii.node.protocol` module, and used by default for clients created by means of the `connect` interface introduced in section 2.3.4. For all packages, documentation is automatically generated using the combination of `sphinx` and *Read the Docs* [91].

2.4 OCR Module

To show the uses of the Python node, a *processing module* was implemented to perform OCR tasks in a Ubi-Interact context. It is available as `ubii-processing-module-ocr` on PyPi.

2.4.1 OCR in Mixed Reality

There is a multitude of applications for OCR in augmented and virtual reality – from extracting metadata for books in a library [92] to interactive text books which help people with learning disabilities [93], augmenting physical documents with searching capabilities for academic reading tasks [94] or translating text [95, 96]. In all cases the tasks are rather involved and rely on existing specialized technologies for the different sub problems – like viewpoint tracking and interaction with virtual objects, handling possible input methods and gesture recognition as well as relevant output methods (like speech synthesis or "augmenting" text onto objects) and, last but not least, image processing for text detection and OCR. Depending on the context and technology stack, the sub-tasks are solved locally (using a e.g. ARToolkit [97] or the tools provided by the

[1] refer to *Getting started — CLI — ubii-node-python documentation* [88] for documentation

Microsoft HoloLens™) or remotely using a Software as a Service (SaaS) approach [92, 94]. In the case where one can use Ubi-Interact for applications like these, a *processing module* would be the right tool to handle *Text Detection* and/or Optical Character Recognition (OCR), so that multiple platforms could integrate and use this functionality. In addition to the advantage that no possibly sensitive data has to be sent to some remote web service, and the associated delay which would rule out the use in real time applications, Ubi-Interact provides all the communication and middleware tools out of the box, so developing a processing module should be a comparatively easy task in relation to e.g. implementing a dedicated web server to avoid privacy and latency issues.

2.4.2 Involved Technology

Out of the discussed publications Li *et al.* [94] choose to handle OCR by using the Microsoft Azure API [98] while all other publications choose to use the Tesseract OCR engine [2].

Tesseract OCR Engine

The Tesseract engine is open source, which makes it specifically interesting for research applications. It is also very accurate and can process a wide variety of image formats via the Leptonica Image Processing Library. It is historically one of the most accurate OCR engines (compare results of the 1995 UNLV Accuracy test for early Tesseract versions [99]), and has since been adopted and improved extensively by Google. The implementation uses the `tesserocr` [100] Python wrapper which performs better than the default Python bindings because it allows to load and reuse a Tesseract instance for multiple API calls.

OpenCV

Tesseract's performance – especially in scenarios where the OCR has to be done in "natural" scenes – can benefit from preprocessing the image data, e.g to detect the text bounding boxes (for example used by Fragoso *et al.* [95]). Despite constant improvements of the text detection capabilities built into Tesseract, the significance of preprocessing for our use case is shown in chapter 3. In the future this functionality could become a dedicated processing module, for now the `ubii-processing-module-ocr` distribution provides three different processing modules instead

`TesseractOCR_PURE`
: only uses the capabilities of the Tesseract library to detect text bounding boxes and extract contents

`TesseractOCR_MSER`
: performs the Maximally Stable Extremal Region (MSER) algorithm [101] to detect character bounding boxes before using Tesseract to do the OCR

`TesseractOCR_EAST`
: uses the Efficient and Accurate Scene Text Detector (EAST) pipeline proposed by Zhou *et al.* [102] to detect text bounding boxes, then uses Tesseract for the OCR

The preprocessing is implemented with the help of the OpenCV [103] image processing library, which supports the MSER algorithm out of the box[1] and the EAST pipeline through its capability to load Convolutional Neural Network (CNN) files.

Numpy

To get the last part of the EAST pipeline reasonably fast, the prediction step was implemented using features of the Python library `numpy`, as naively porting the C code from the corresponding OpenCV example [104] will lead to bad performance[2]. A naive implementation was up to an order of magnitude slower – depending on the hardware and image size – than an optimized implementation using `numpy`'s "matrix computation" features for the `predict` and `decode` steps of the EAST pipeline.

2.4.3 Automatic Module Discovery

Section 2.3.5 mentions that the CLI script for the Ubi-Interact Python node automatically discovers installed *processing modules*, and loads them. This functionality is based on *entry points*, a feature which a Python distribution can use to "advertise components it provides to be discovered by other code" [105].
The `ubii-processing-module-ocr` distribution advertises its three implementations of the `ubii.framework.processing.ProcessingRoutine` type as entry points in a specific group (`ubii.processing_modules`) so the CLI implementation can find them if they are installed. New processing modules should also implement this functionality so that they can be automatically loaded in the default Ubi-Interact Python client if they are installed. Developers can consult the documentation of their build backend of choice (`ubii-processing-module-ocr` uses `setuptools`) on how to define entry points.

[1] because we're extracting bounding boxes – no pun intended
[2] naively porting the code is actually how all the "online tutorials" do it – because people are not supposed to understand `numpy`, supposedly? Anyways, Python wouldn't be the same without bad code that's repeatedly copy pasted all over the internet <\rant>

2.4.4 Portability

During development of the processing module, `tesserocr` on Windows only supported Python version 3.6 and 3.7, support for newer versions was added later [106]. Therefore – to allow users to run the processing module on a Python node running on a Windows platform, historically – both the processing module and the Ubi-Interact Python node implementation support Python version 3.7 and upwards. The `tesserocr` package has to be installed manually on Windows machines though, it is not available on PyPi.

3 Evaluation

This chapter deals with evaluating the performance of the newly developed Ubi-Interact Python client node running the OCR module introduced in section 2.4.

Ubi-Interact Python *processing modules* which run in "frequency" mode, instead of processing whenever they receive input (which would correspond to the "trigger on input" mode) or whenever the broker schedules the processing (corresponding to "lockstep" mode), will trigger the processing depending on the chosen frequency value.[2]

The performance of "frequency" modules can be evaluated by the scheduler, in terms of the actual interval between processing steps compared to the planned interval. If a processing pass takes longer than the interval resulting from a specific frequency (e.g. for a frequency of $f = 10\frac{1}{s}$ the corresponding interval between processing steps would equate to $\Delta t = 0.1s$ accordingly) the relative error between actual processing interval and the corresponding frequency-interval can be used as a performance metric. The default *scheduler* which is used by the processing protocol to handle the correct processing execution – see section 2.3.3 – exposes the last 30 time intervals between computations, as well as the last 30 execution times, and takes two optional arguments – `schedule_perf_metric` and `exec_perf_metric` – which need to be callables that can compute the performance from a given scheduler instance, e.g. by computing the error of execution times or scheduling times. The default performance metric, based on the relative errors between processing intervals, is shown in eq. (3.1).

$$
\begin{aligned}
\bar{t} &= avg(\texttt{scheduler.exec_delta_times}) \\
error_t &= \frac{\bar{t} - \texttt{scheduler.delay}}{\texttt{scheduler.delay}} \\
PM_{perf} &= \begin{cases} 1 & \bar{t} < \texttt{scheduler.delay} \\ 1 - error_t & \text{otherwise} \end{cases}
\end{aligned}
\tag{3.1}
$$

[2] In addition to the aforementioned main processing modes which are described in greater detail by Weber *et al.* [1, § 5.5], newer versions of the Ubi-Interact framework also support a "free" processing mode which lets the client node decide independently when to process.

The `scheduler.delay` variable in eq. (3.1) contains the computed delay or interval between processing steps, corresponding to the frequency value set for the module's processing mode. The `scheduler.exec_delta_times` variable contains the last 30 recorded execution times. As one can see from eq. (3.1) PM_{perf} can get negative if the relative error becomes larger than 1 so it is not a norm in the mathematical sense – for practical purposes this is not relevant though.

While this feature is useful to inspect a modules performance during runtime, more detailed statistics have been computed, for the purpose of this thesis. A test module was used to simply record all execution times and scheduling times for different processing frequencies over a $10s$ time period, which were then evaluated.

The statistics in table 3.1 are computed from the measured scheduling times – or rather the equivalent frequencies that are implied by those timings – to evaluate the timing accuracy of the code. Although it is in theory possible to perform around 4000 processing passes per second (as long as they don't actually perform an expensive computation), the `asyncio` framework is limited by the clock resolution of the used event loop, but even more so by the system calls that are used to notify waiting tasks. For example on Linux `await asyncio.sleep(...)` uses two `epoll_wait` system calls where for values smaller than 1 ms `epoll_wait` requests a 1 ms timeout nonetheless. For values smaller than 15 μs `epoll_wait` requests a 0 ms timeout, although the two syscalls take 8 μs each so the total time waited is still at least 16 μs[1]. To alleviate this issue, the *scheduler* has an attribute `timing_thresholds` which can take a tuple of "minimum" timing delays. The default protocol sets this value to `(0.001, 0.00015)` – only on Linux machines when the used loop is the default `asyncio` loop – which the scheduler uses to adjust its timings which results in a much more accurate mean scheduling frequency, as can be seen in table 3.1: The user can adjust this behavior (or opt out of it) by changing the `timing_thresholds` attribute, an empty value suggests no adjustments which will make the actual scheduling intervals at least as big as the `delay` associated with the set

	10 *hz*	10 *hz* *no adjust.*	60 *hz*	60 *hz* *no adjust.*	120 *hz*	120 *hz* *no adjust.*	200 *hz*	200 *hz* *no adjust.*
count	104	103	615	580	1,257	1,120	2,120	1,748
mean	9.977	9.876	58.7	55.34	120.4	107.1	203.9	167.7
std	0.026	0.029	0.938	1.219	6.163	4.876	15.868	9.866
min	9.825	9.747	55.05	35.33	74.24	69.85	133.6	55.22
50%	9.977	9.877	58.62	55.19	122	108.4	207.4	168.5
max	10.06	9.927	63.07	59	135.3	117.5	236.4	189.3

Table 3.1: Timing statistics for test modules with different frequencies

[1] to verify this, run a program that waits for the specified time and then executes a visible syscall under `strace`. Every `sleep` produces 2 `epoll_wait` calls, where the first waits for approximately the specified time, but always at least 1 ms until the waiting time goes below 15 μs.

A) Text Render B) High Quality Photo[1] C) Low Quality Webcam Image

FIGURE 3.1: Images used for evaluation tests – original images resized to 1200 × 900 pixels

[1] by Michael Sander - Self-photographed, CC BY-SA 3.0, `https://commons.wikimedia.org/w/index.php?curid=658534`

frequency. Refer to table 3.1 for comparison – one can see that the adjustments become more relevant for higher frequency modules and are solving the issue of systematic timing errors but result in less predictable timings. Note that the relative standard deviation of a 200 *hz* frequency module is $15.87/203.9 \approx 0.078$ so around 8% of the mean value, which means the timings are still tightly clustered around the mean albeit less so than for the non-adjusted module (which has a relative standard deviation of $9.9/167.7 \approx 0.059$ so around 6%). One can also see that for the adjusted timings, the 50th percentile or median is close to the target frequency, which indicates that there are approximately as many timings that were faster than the target frequency, as there were slower timings. The code used to measure these timings is available as a `pytest` test case in the `ubii-node-python` distribution.

Evaluation of the OCR performance is also implemented as a `pytest` test case available as part of the `ubii-processing-module-ocr` distribution. Pytest test cases allow for easy parametrization of the modules and testing different input images. For comparable results, all images have the same dimensions (1200 × 900 pixels) but different "quality": one image is simply rendered text on white background to get a baseline performance, one image is a high quality photograph of a sign, and the last one is a webcam image which shows the text from the baseline example in a real world scene, compare fig. 3.1. For each evaluation, statistics of the *execution times* have been included. Note that these are not comparable to the statistics of the scheduling times shown in table 3.1, but are simply a more detailed measurement compared to the normal *performance metric* from eq. (3.1). By measuring the execution times it is possible to evaluate the possibility for real time application of the processing modules.

All measurements were done with an Intel® Core™ i5-8265U CPU with 1.60GHz and 8GB of DDR4 RAM clocked at 2667 MHz. Modules ran for 20s or until they processed 50 images, whichever happened first.

`TesseractOCR_PURE`

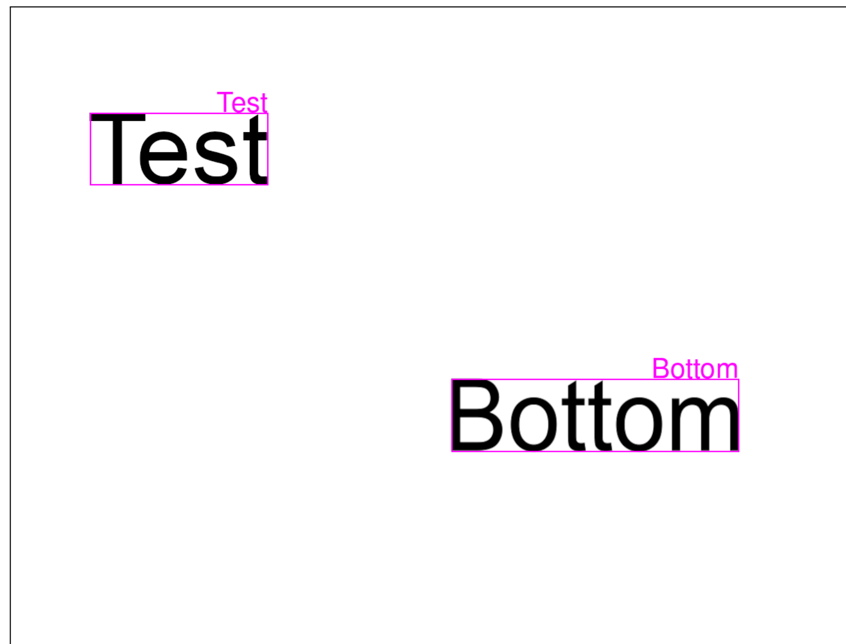

FIGURE 3.2: Pure Tesseract processing on baseline image

count	$\bar{t}$ [s]	relative std	median [s]
50	0.050	0.027	0.050

Without preprocessing the pure Tesseract module is able to detect the test text without issues. It also has the best execution time, corresponding to $\approx$ 20 hertz – according to the measured mean execution time of 0.050s. The default settings for the module instantiate the API in “default” OCR engine mode – one can choose between a “tesseract only” implementation for the character recognition, a Long short-term memory (LSTM) network implementation, or a combination of both methods (for more information refer to the `--oem` option in the Tesseract CLI documentation [107]) – and “automatic” page segmentation mode (see the `--psm` option in the Tesseract CLI documentation [107]), and considers text with a minimum confidence of 70. If these settings are changed for certain parts of the evaluation, it will be noted.

`TesseractOCR_EAST`

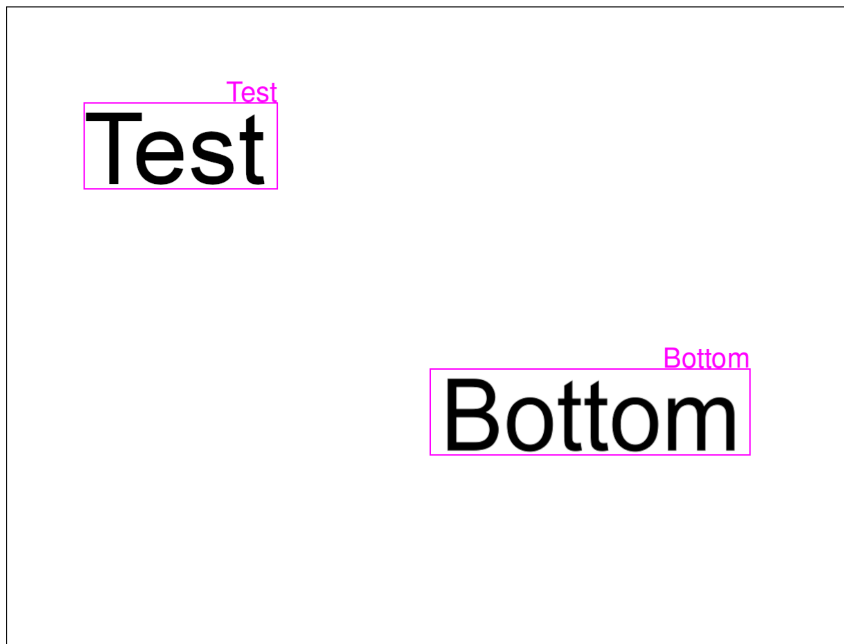

FIGURE 3.3: EAST preprocessing on baseline image

count	$\bar{t}$ [s]	relative std	median [s]
50	0.135	0.234	0.122

EAST preprocessing detects the text without issues, it can be extracted with results that match the pure Tesseract modules performance. Notably, the execution time is longer (more than double the execution time of the non-preprocessing module) but processing still happens at around 7 processing passes per second, which should be considered usable in real time applications.

`TesseractOCR_MSER`

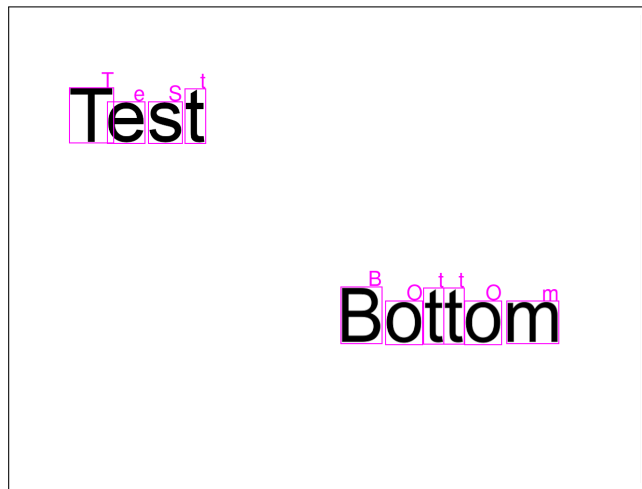

Figure 3.4: MSER preprocessing on baseline image

count	$\bar{t}$ [s]	relative std	median [s]
50	0.074	0.038	0.073

This module instantiates the Tesseract API in "single character" page segmentation mode. The MSER algorithm is able to detect the bounding boxes, Tesseract detects the characters. The default settings use a `padding` of 2 px around detected boxes. Without padding, the regions are sometimes too small and Tesseract fails to correctly extract the character. E.g. the "T" in fig. 3.5 is extracted as "1" and the "m" can't be extracted with enough confidence. If the padding is too large, extraction fails because bounding boxes contain artifacts from neighboring characters.

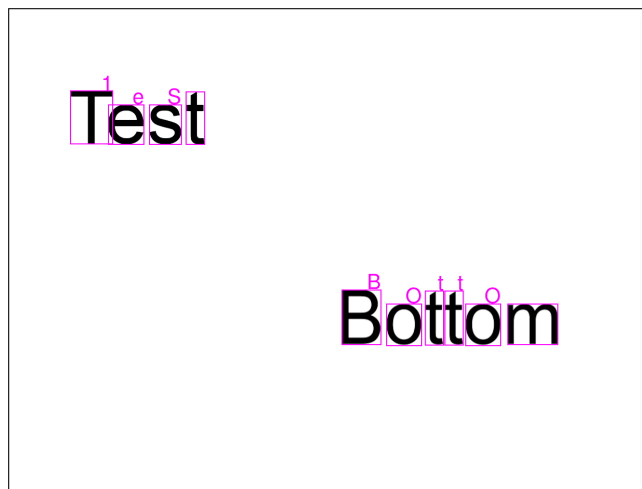

Figure 3.5: MSER preprocessing without padding on baseline image

`TesseractOCR_MSER`

FIGURE 3.6: MSER preprocessing on sign phorograph

count	$\bar{t}$ [s]	relative std	median [s]
20	0.941	0.081	0.910

Using the high quality photograph, the limitations of the MSER module become visible. This is not the right use case, there is too much noise in the image, and a lot of ”extremal” regions are detected. Due to the large number of regions of interest, the processing takes nearly a second. Note that it is also possible to use the colored image as input, but since performance is almost always better with a grayscale input the MSER module applies the algorithm to a grayscale version of the input image by default (the corresponding module parameter is the boolean value `mser_grayscale`, also using any of the possible “color only” options for the `mser_args` parameter will automatically use a colored input image – for more information refer to the documentation of the OpenCV MSER implementation [108]).

`TesseractOCR_MSER`

FIGURE 3.7: MSER preprocessing parameter adjustments on sign photograph

count	$\bar{t}$ [s]	relative std	median [s]
50	0.245	0.202	0.226

Suppressing bounding boxes where no text has been found is possible in all modules, also the `mser_args` parameter was used here to set the maximum considered variation for the MSER algorithm to 0.02 which prunes bounding boxes with similar sized children, which at least reduces the processing time to $\approx 0.25\,s$, although the results are still questionable. The MSER module is clearly not well suited for this scene and not very robust with respect to the input image.

`TesseractOCR_PURE`

FIGURE 3.8: Pure Tesseract processing on sign photograph with confidence values

count	$\bar{t}$ [s]	relative std	median [s]
50	0.137	0.023	0.137

Processing in "automatic" page segmentation mode correctly finds the bounding boxes. For illustration fig. 3.8 includes the confidence values for extracted text – as seen the confidence for the middle text is very low. There is also a "false positive" text box, below the sign – discarding boxes without recognized text would get rid of it. Using the "sparse" segmentation mode was considered as an alternative, but did produce worse results, i.e. additional "false positive" regions of interest, some with more text confidence than the middle line of the sign.

FIGURE 3.9: Pure Tesseract processing on sign photograph in "sparse" segmentation mode

`TesseractOCR_EAST`

Figure 3.10: EAST preprocessing on sign photograph

count	$\bar{t}$ [s]	relative std	median [s]
50	0.199	0.072	0.199

For the settings used in the baseline example, the bounding boxes are somewhat inaccurately detected by the EAST preprocessing, which seems to hinder the successful text extraction.
Actually though, the EAST module does not only detect bounding boxes, it also performs *non-maximum suppression* on the detected boxes – by default the threshold to discard boxes is 0.5 i.e. boxes which overlap by at least 50%. If one increases this value, more of the detected bounding boxes become visible as can be seen in fig. 3.11, and it becomes obvious that in the case of the boxes for the word "Universitätsstadt" the one with the highest confidence score (which "survives" the *non-maximum suppression* and ends up in fig. 3.10) is not the one which is best suited for the Tesseract task.

Instead of choosing bounding boxes by *non-maximum suppression*, the module implements an alternate scheme to *merge* overlapping bounding boxes which has proven to be more robust than searching for the right threshold parameters, but does not generate results with equal accuracy.

`TesseractOCR_EAST`

FIGURE 3.11: EAST preprocessing on sign photograph, decreased non-maximum suppression

count	$\bar{t}$ [s]	relative std	median [s]
50	0.237	0.056	0.234

For a threshold of 0.7 in the *non-maximum suppression* stage of the EAST module, and an OCR confidence of 50 (which is just enough to not consider the bad result from fig. 3.10), one can see that several "candidates" for each line were detected, and each group contains a box where the text can be extracted with good confidence, but the EAST algorithm is sometimes more confident in ill-aligned boxes, which survive strict *non-maximum suppression* instead of the boxes which are well suited for the Tesseract task. This means good results can be achieved by tweaking the suppression threshold in a way that more boxes survive, but keeping the OCR threshold high enough that "bad" results get discarded, compare fig. 3.12.

`TesseractOCR_EAST`

FIGURE 3.12: EAST preprocessing on sign photograph, decreased non-maximum suppression, decreased min confidence

count	$\bar{t}$ [s]	relative std	median [s]
50	0.214	0.120	0.208

For a threshold of 0.7 in the *non-maximum suppression* stage of the EAST module, and an OCR confidence ≥ 50 the “false positive” word “|lnivercitatectadt” from the ill-aligned box which survived in fig. 3.10 when using a smaller suppression threshold, can be discarded due to insufficient OCR confidence. One can see that although the preprocessing increases the execution time – decreasing the execution frequency from $\approx 6\,hz$ without preprocessing in fig. 3.8 to $\approx 4\,hz$ – it is still useable in real time applications and can produce very good results for images like the example photograph. Tweaking parameters of course requires some knowledge of the input data and testing, but a slightly smaller OCR confidence as in the baseline text render and a slightly bigger threshold for the *non-maximum suppression* – or using the *merge* scheme instead – should produce reasonable results in most cases. Note that more possible candidates result in more OCR passes and slightly slower execution compared to stricter bounding box selection.

`TesseractOCR_PURE`

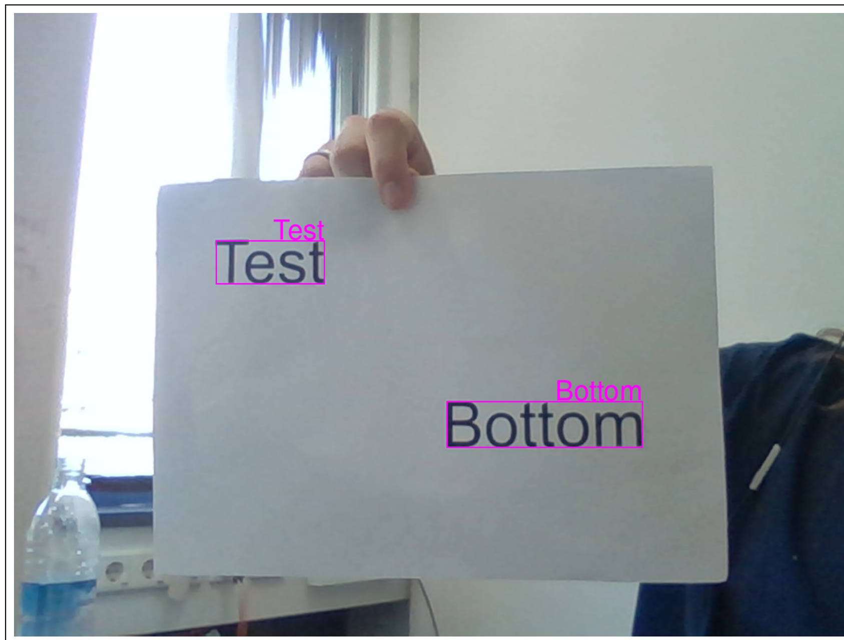

FIGURE 3.13: Pure Tesseract processing in "sparse" page segmentation mode on webcam image

count	$\bar{t}$ [s]	relative std	median [s]
50	0.067	0.038	0.067

Using the settings for the baseline case – specifically automatic page segmentation mode – does not find a single bounding box to even extract text in the first place!

This is a case for "sparse" page segmentation which produces very good results as can be seen in fig. 3.13, while the automatic page segmentation mode fails to detect any bounding boxes. Execution times are very small, as always for the pure Tesseract module, and allow execution frequencies of $\approx 15\,hz$. Notably, knowledge about the input image and testing was needed for the right choice of page segmentation mode – as mentioned, "automatic" mode does fail for a low quality input like this.

`TesseractOCR_EAST`

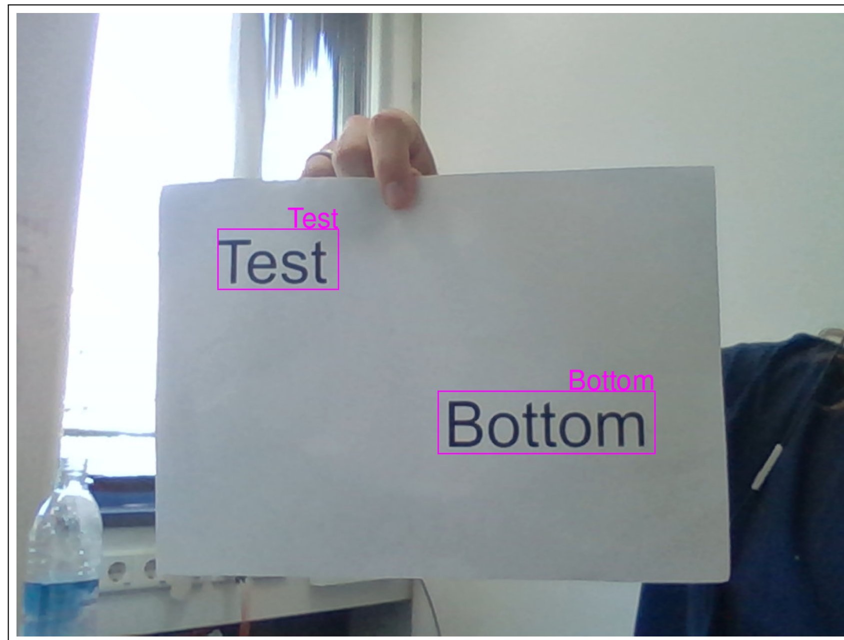

FIGURE 3.14: EAST preprocessing on webcam image

count	$\bar{t}$ [s]	relative std	median [s]
50	0.134	0.094	0.130

EAST preprocessing with default settings from the baseline example finds and extracts text even in this low quality image without issue. This is the big advantage of the preprocessing module compared to the pure Tesseract module – it is much more robust to different kinds of input images. While for the pure Tesseract module the choice of page segmentation made the difference between nearly perfect performance and failure, the EAST preprocessing module hides the difference from the user. This added robustness comes with the cost of increased processing time (like in the baseline case around two times the processing time for the pure Tesseract module) but it is still fast enough for real time applications with execution frequencies of $\approx 7\,hz$ just like in the baseline case.

`TesseractOCR_MSER`

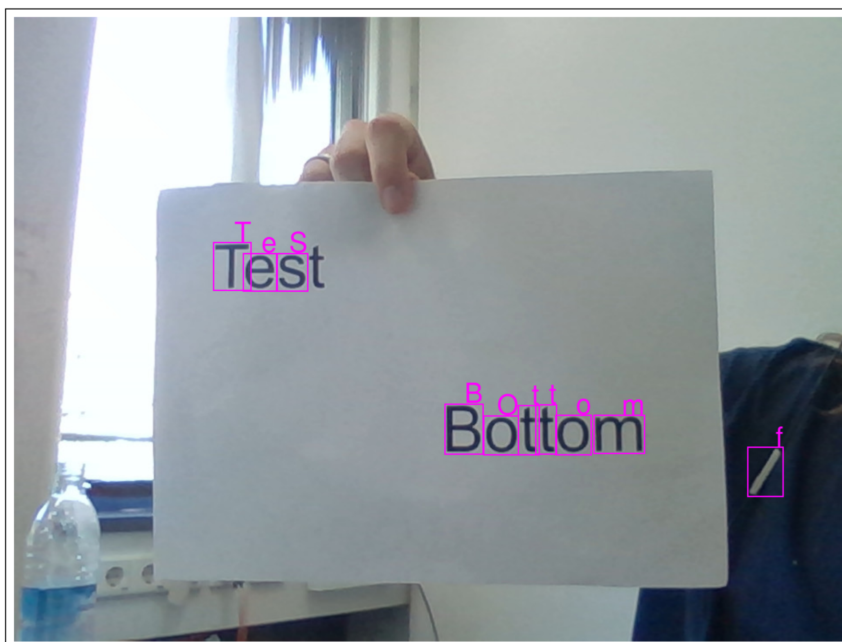

FIGURE 3.15: MSER preprocessing on webcam image

count	$\bar{t}$ [s]	relative std	median [s]
50	0.148	0.111	0.144

Due to the sparse text and higher contrast in the scene – compared to the photograph – one can even achieve acceptable results with the MSER module with some adjustments. The biggest difference compared to the baseline case is the use of a character whitelist. Variables for the Tesseract API can be passed to the module via the `api_variables` argument, in this test case the `tessedit_char_whitelist` variable of the API instance is set to include only alphabetic characters. With some tweaking of the OCR confidence threshold (a value of 60 was used here), the results are acceptable. Since the MSER algorithm does not detect characters, but just regions, and Tesseract in "single character" segmentation mode assumes one character per input region, one still tends to get false positives (e.g the "f" in fig. 3.15) while characters which are too close together will not be extracted with high confidence due to the artifacts that were already mentioned as problems for the baseline case. Execution times are reasonable, $\approx 7\,hz$ execution frequencies are possible. As we have seen in the photograph example, execution times of the MSER module are correlated with the number of extremal regions found in the image, so for any input with a manageable amount of regions of interest the execution times should be usable for real time applications.

Note that it would be possible to adjust the default settings for the EAST preprocessing module in a way that it would produce the required results on all tested images without tuning. Specifically, a *non-maximum suppression* threshold of 0.7, an OCR confidence threshold of 60 and choosing to discard all bounding boxes where no text is extracted would work in all cases. The defaults use different values nonetheless, since choosing an OCR confidence threshold of 60 is just as arbitrary as choosing a value of 70 (which is the current default), and a strict *non-maximum suppression* with a threshold of 0.5 is the default for the OpenCV implementation of the algorithm.

4 Conclusion

4.1 Summary

A Python software suite for Ubi-Interact was developed, which features a clean API for the end user as well as a flexible framework for (future) developers and maintainers as well as a `pytest` plugin (the `ubii.node.pytest` module available as part of the `ubii-python-node` distribution), a feature complete Ubi-Interact client node with *processing module* support (`ubii.node` package), a custom protocol buffer package with improved usability (`ubii.proto` package), and a *processing module* able to perform multiple real time OCR tasks due to its flexible design (`ubii.processing_modules.ocr` package available as `ubii-processing-module-ocr` distribution), and last but not least extensive, complete, automatically updated documentation for all relevant parts (and beyond).

These software modules will be (and have been) used to bring the cross-platform, cross-language communication and middleware features of Ubi-Interact into the world of Python – for researchers and students which aim to develop applications bridging the gap between Internet of Things, Human-Computer Interaction and mixed reality domains like the "Catching the Drone - A Tangible Augmented Reality Game in Superhuman Sports" [46] project. Conversely it offers the unique capabilities of Python to the existing Ubi-Interact infrastructure – for example easy to use image processing tools, or an alternative test framework to aid in developing nodes in other languages.

Analyzing the use cases of Ubi-Interact – also in comparison to existing solutions (for related problem statements) like the Robot Operating System (ROS) or other historically and currently used middleware solutions – showed the importance of a multi-layered design which could present the software in different levels of complexity to facilitate its use by researchers and students with different backgrounds and experience in an academic setting. This was achieved by the design concepts of *protocols* and *behaviors* (see section 2.3) which are used throughout the Ubi-Interact Python framework to break down and simplify complex runtime behaviors in a context where respecting the Separation of Concerns (SoC) principle is of utmost importance as objects and concepts are shared across the system boundaries by multiple heterogeneous nodes in a distributed network.

4.2 Future Work

Ubi-Interact is still an evolving software, and as new features – like for example advanced inference and search on topic metadata or new *modes* for processing modules – are implemented, they should be added to the Python node at some point. Also it would be reasonable to split the developed *processing module* which is able to perform two separate tasks – namely recognizing regions of interest in an image, and extracting text from those regions – into two separate processing modules to increase modularity. As the Python support for protocol buffers changes, the corresponding package and its use in the Python node need to be reevaluated, since keeping protocol buffer support up-to-date could further improve the performance of the framework. Support for the *μpb* [58] runtime in the *Proto Plus for Python* [63] package is especially interesting in that regard.

Given the now available OCR capabilities, a Ubi-Interact powered system could be used for a wide range of HCI tasks. For more complex text processing, e.g. when dealing with mixed reality document annotations (a common research problem [94, 96, 109, 110]) separating the preprocessing and actual OCR functionality would allow to use domain specific models to obtain the structure (i.e. regions of interest) from the image. In the case of document annotation, the publication of the `PubLayNet` [111] dataset has led to a surge in performance of machine learning models dealing with extracting structural features from (digital) documents. Deep neural networks pre-trained on `PubLayNet` – a dataset of scientific documents – can also be repurposed for different domains. This approach of "transfer learning" is made possible due to the large size of the dataset, which allows to learn "general" features that can be applicable to several domains. Zhong *et al.* [111] show that models pre-trained on `PubLayNet` and adapted for documents of private health insurance providers (which are not immediately similar to the scientific documents used to train the base model) can "substantially outperform" models that are for example pre-trained with more generic datasets (COCO [112] and ImageNet [113]). Using a domain specific model trained on `PubLayNet`(R. Wang *et al.* [114] propose a spacial graph convolutional network to detect text bounding boxes, for example) would certainly outperform the EAST model (which was developed prior to the publication of `PubLayNet`, and is therefore trained on "standard datasets"[1]) used in conjunction with the current OCR processing in Ubi-Interact, when it comes to extracting document structure.

Since the parameters and objects involved in the algorithms used for the OCR task are made available and mutable at runtime by design, a context aware module could be integrated into a feedback loop to allow inference of the best parametrization via (user supplied) context information.

Hopefully the Ubi-Interact Python packages will help to develop more interesting applications, by giving users of Ubi-Interact yet another platform to work with.

[1] including ICDAR 2015, COCO-Text and MSRA-TD500

Acronyms

API Application Programming Interface. 3–5, 10, 15, 16, 18–21, 26, 28, 32, 40, 64, 66, 75, 77

CLI Command Line Interface. 44, 56, 58, 64

CNN Convolutional Neural Network. 58

DDS Data Distribution Service. 3–6, 9, 11, 85

EAST Efficient and Accurate Scene Text Detector. 58

HCI Human-Computer Interaction. iii, 1, 3, 5, 6, 12, 13, 77, 78

IDL Interface Description Language. 3, 9

IoT Internet of Things. iii, 3, 5–7, 12, 77

LCM Lightweight Communications and Marshalling. 1, 9

LSTM Long short-term memory. 64

MOM Message Oriented Middleware. 3, 6, 7, 85

MOOS Mission Oriented Operating Suite. 1, 7–9

MQTT Message Queuing Telemetry Transport. 3, 6, 11

MRO Method Resolution Order. 37

MSER Maximally Stable Extremal Region. 58

OCR Optical Character Recognition. iii, 14, 56–58, 61, 63, 64, 71, 72, 75–78

OOP Object Oriented Programming. 18, 20

QoS Quality of Service. 3–5

REPL Read Evaluate Print Loop. 49

ROS Robot Operating System. 1–6, 9, 11–13, 77

RPC Remote Procedure Call. 11, 25

SaaS Software as a Service. 57

SoC Separation of Concerns. 31, 37, 77

TCP Transmission Control Protocol. 8

UDP User Datagram Protocol. 8

UWP Universal Windows Platform. 12

XDR External Data Representation. 9

List of Code Examples

2.1 Default Python module generated from simple schema file 17
2.1.1 Protocol buffer schema – `color.proto` 17
2.1.2 Python module compiled from `color.proto` using default plugin 17
2.2 Python module compiled from `color.proto` using custom plugin 27
2.3 Use of custom metaclass to extend a protocol buffer wrapper 31
2.4 Protocol buffer schema – `client.proto` 32
2.5 Protocol buffer schema – `ProcessingModule.proto` 36
2.6 Naive dependency injection 38
2.7 Bad dependency injection 39
2.8 Python dependency injection pattern 40
2.9 Python dependency injection with static duck typing 41
2.10 Usage of custom asyncio coroutine wrapper 43
2.11 Minimal *protocol* usable with Ubi-Interact Python framework 46
2.12 Client interface definition using dataclasses and static duck typing . . . 48
2.13 Usage of public Ubi-Interact Python node interface – part 1 52
2.13 Usage of public Ubi-Interact Python node interface – part 2 53
2.14 Usage of public Ubi-Interact Python node interface – part 3 54
2.15 Interface to instantiate Ubi-Interact node 55

List of Figures

2.1 Design of protocol buffer handling hiding specifications 20
2.2 Design of protocol buffer handling with public specifications 21
2.3 Design of protocol buffer handling using abstract specification base class 22
2.4 ubii-message-formats package 30
2.5 Ubi-Interact "middleware protocol" 34
2.6 Design of object behavior as state-machine instances 45
2.7 IDE hints with statically duck typed *behavior* 49

3.1 Images used for evaluation tests 63
3.2 Pure Tesseract processing on baseline image 64
3.3 EAST preprocessing on baseline image 65
3.4 MSER preprocessing on baseline image 66
3.5 MSER preprocessing without padding on baseline image 66
3.6 MSER preprocessing on sign phorograph 67
3.7 MSER preprocessing parameter adjustments on sign photograph 68
3.8 Pure Tesseract processing on sign photograph with confidence values . . 69
3.9 Pure Tesseract processing on sign photograph in "sparse" segmentation mode 69
3.10 EAST preprocessing on sign photograph 70
3.11 EAST preprocessing on sign photograph, decreased non-maximum suppression 71
3.12 EAST preprocessing on sign photograph, decreased non-maximum suppression, decreased min confidence 72
3.13 Pure Tesseract processing in "sparse" page segmentation mode on webcam image 73
3.14 EAST preprocessing on webcam image 74
3.15 MSER preprocessing on webcam image 75

List of Tables

1.1 Comparison of (historically) used communication packages [3] 2
1.2 ROS Feature Analysis 4
1.3 DDS open-source implementations 5
1.4 Overview of MOM solutions [28] 7

2.1 Comparison of protocol buffer Python packages 28

3.1 Timing statistics for test modules with different frequencies 62

Bibliography

[1] S. Weber, D. Dyrda, M. Ludwig, and G. Klinker, "Ubi-Interact," in *MobiQuitous 2020 - 17th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, ser. MobiQuitous '20, New York, NY, USA: Association for Computing Machinery, Dec. 7, 2020, pp. 291–300, ISBN: 978-1-4503-8840-5. DOI: 10.1145/3448891.3448924. [Online]. Available: https://doi.org/10.1145/3448891.3448924 (visited on 05/18/2022).

[2] *Tesseract OCR*, tesseract-ocr, Jul. 2, 2022. [Online]. Available: https://github.com/tesseract-ocr/tesseract (visited on 07/02/2022).

[3] D. Moore, E. Olson, and A. Huang, "Lightweight Communications and Marshalling for Low-Latency Interprocess Communication," Sep. 2, 2009. [Online]. Available: https://dspace.mit.edu/handle/1721.1/46708 (visited on 05/25/2022).

[4] *Lightweight Communications and Marshalling (LCM)*, lcm-proj, May 23, 2022. [Online]. Available: https://github.com/lcm-proj/lcm (visited on 05/25/2022).

[5] Reid Simmons. "Inter Process Communication (IPC)." (Nov. 4, 2014), [Online]. Available: http://www.cs.cmu.edu/~ipc/ (visited on 05/25/2022).

[6] "ACTIVE-IST - Open Source Tools." (2011), [Online]. Available: http://active-ist.sourceforge.net/index.php (visited on 05/25/2022).

[7] *Core-moos*, themoos, May 4, 2022. [Online]. Available: https://github.com/themoos/core-moos (visited on 05/24/2022).

[8] R. Vaughan, *The Stage Simulator*, May 16, 2022. [Online]. Available: https://github.com/rtv/Stage (visited on 05/26/2022).

[9] "Announcing Microsoft Robotics Developer Studio 4 Beta - Microsoft Robotics Blog - Site Home - MSDN Blogs." (Sep. 23, 2011), [Online]. Available: https://web.archive.org/web/20110923175247/http://blogs.msdn.com/b/msroboticsstudio/archive/2011/09/17/announcing-microsoft-robotics-developer-studio-4-beta.aspx (visited on 05/26/2022).

[10] Open Robotics. "ROS: Landing Page." (2021), [Online]. Available: https://www.ros.org/ (visited on 05/07/2022).

[11] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng, *et al.*, "ROS: An open-source robot operating system," in *ICRA workshop on open source software*, Kobe, Japan, vol. 3, 2009, p. 5.

[12] A. Brooks, T. Kaupp, A. Makarenko, S. Williams, and A. Oreback, "Towards component-based robotics," in *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Aug. 2005, pp. 163–168. DOI: 10.1109/IROS.2005.1545523.

[13] Open Robotics. "Catkin/conceptual_overview - ROS Wiki." (Mar. 26, 2020), [Online]. Available: http://wiki.ros.org/catkin/conceptual_overview (visited on 05/09/2022).

[14] Open Robotics. "Messages - ROS Wiki." (2019), [Online]. Available: http://wiki.ros.org/Messages (visited on 05/15/2022).

[15] Open Robotics. "ROS on DDS," ROS on DDS. (2021), [Online]. Available: http://design.ros2.org/articles/ros_on_dds.html (visited on 05/17/2022).

[16] "Topic and Service name mapping to DDS." (Jun. 2018), [Online]. Available: https://design.ros2.org/articles/topic_and_service_names.html (visited on 07/27/2022).

[17] J. Canady. "If you have REST, why XML-RPC?" (May 11, 2013), [Online]. Available: https://web.archive.org/web/20130511053512/http://joncanady.com/blog/2010/01/14/if-you-have-rest-why-xml-rpc/ (visited on 05/15/2022).

[18] "XmlRpcDiscussion - Atom Wiki." (2003), [Online]. Available: http://www.intertwingly.net/wiki/pie/XmlRpcDiscussion?action=show&redirect=DontUseXmlRpc (visited on 05/15/2022).

[19] *PyOpenDDS*, OCI Labs - Object Computing, Inc., Apr. 13, 2022. [Online]. Available: https://github.com/oci-labs/pyopendds (visited on 05/22/2022).

[20] *Python binding for Eclipse Cyclone DDS*, Eclipse Cyclone DDS™, May 12, 2022. [Online]. Available: https://github.com/eclipse-cyclonedds/cyclonedds-python (visited on 05/21/2022).

[21] *Python binding for Fast DDS*, eProsima, May 10, 2022. [Online]. Available: https://github.com/eProsima/Fast-DDS-python (visited on 05/22/2022).

[22] N. Wang and S. Shasharina, "Data Distribution Service for Python Applications," p. 21, Nov. 9, 2015. [Online]. Available: https://www.omg.org/news/meetings/tc/dc-13/special-events/dds-pdfs/S8-Wang.pdf (visited on 05/21/2022).

[23] *OpenDDS*, Object Computing, Inc., May 20, 2022. [Online]. Available: https://github.com/objectcomputing/OpenDDS (visited on 05/21/2022).

[24] eProsima. "DDS API — Fast DDS 2.6.0 documentation." (Feb. 14, 2022), [Online]. Available: https://fast-dds.docs.eprosima.com/en/latest/ (visited on 05/22/2022).

[25] Open Robotics. "About Quality of Service settings — ROS 2 Documentation: Rolling documentation." (May 11, 2021), [Online]. Available: http://docs.ros.org/en/rolling/Concepts/About-Quality-of-Service-Settings.html?highlight=topic (visited on 05/22/2022).

[26] P. C. Wright, R. E. Fields, and M. D. Harrison, "Analyzing Human-Computer Interaction as Distributed Cognition: The Resources Model," *Human–Computer Interaction*, vol. 15, no. 1, pp. 1–41, Mar. 2000, ISSN: 0737-0024, 1532-7051. DOI: 10.1207/S15327051HCI1501_01. [Online]. Available: https://www.tandfonline.com/doi/full/10.1207/S15327051HCI1501_01 (visited on 05/22/2022).

[27] S. Weber and G. Klinker, "VR Re-Embodiment in the Neurorobotics Platform," 2019. DOI: 10.18420/MUC2019-WS-585. [Online]. Available: http://dl.gi.de/handle/20.500.12116/25215 (visited on 05/23/2022).

[28] G. Aures and C. Lübben, "DDS vs. MQTT vs. VSL for IoT," *Network*, vol. 1, 2019.

[29] P. Newman, "Under the Hood of the MOOS Communications API," p. 7, Mar. 17, 2009.

[30] J. Recor, M. Luker, R. Petersen, *et al.*, "Organizing for improved security," 2003.

[31] *Python-moos*, themoos, Jul. 7, 2021. [Online]. Available: https://github.com/themoos/python-moos (visited on 05/24/2022).

[32] P. Newman, "A Guide to using MOOS-V10 Communications," p. 25, Jul. 2, 2013.

[33] *Essential-moos*, themoos, Jul. 8, 2021. [Online]. Available: https://github.com/themoos/essential-moos (visited on 05/25/2022).

[34] P. Newman, "Bridging Communities with pMOOSBridge," p. 6, Jun. 21, 2009.

[35] A. S. Huang, E. Olson, and D. C. Moore, "LCM: Lightweight Communications and Marshalling," in *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Oct. 2010, pp. 4057–4062. DOI: 10.1109/IROS.2010.5649358.

[36] M. Eisler, "XDR: External Data Representation Standard," Internet Engineering Task Force, Request for Comments RFC 4506, May 2006, 27 pp. DOI: 10.17487/RFC4506. [Online]. Available: https://datatracker.ietf.org/doc/rfc4506/ (visited on 05/26/2022).

[37] *Protocol Buffers - Google's data interchange format*, Protocol Buffers, May 26, 2022. [Online]. Available: https://github.com/protocolbuffers/protobuf (visited on 05/26/2022).

[38] *FlatBuffers*, Google, May 26, 2022. [Online]. Available: https://github.com/google/flatbuffers (visited on 05/26/2022).

[39] "Plugin.pb.h | Protocol Buffers," Google Developers. (May 18, 2021), [Online]. Available: https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.compiler.plugin.pb (visited on 05/27/2022).

[40] Open Robotics. "Services - ROS Wiki." (2019), [Online]. Available: http://wiki.ros.org/Services (visited on 05/15/2022).

[41] S. Whims. "Missing .NET APIs in Unity and UWP - UWP applications." (Jun. 23, 2022), [Online]. Available: https://docs.microsoft.com/en-us/windows/uwp/gaming/missing-dot-net-apis-in-unity-and-uwp (visited on 05/31/2022).

[42] "Developing for Windows with the Windows App SDK · Discussion #1615 · microsoft/WindowsAppSDK," GitHub. (Oct. 19, 2021), [Online]. Available: https://github.com/microsoft/WindowsAppSDK/discussions/1615 (visited on 05/31/2022).

[43] D. A. Plecher, C. Eichhorn, A. Köhler, and G. Klinker, "Oppidum - A Serious-AR-Game About Celtic Life and History," in *Games and Learning Alliance*, A. Liapis, G. N. Yannakakis, M. Gentile, and M. Ninaus, Eds., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2019, pp. 550–559, ISBN: 978-3-030-34350-7. DOI: 10.1007/978-3-030-34350-7_53.

[44] D. A. Plecher, A. Ulschmid, T. Kaiser, and G. Klinker, *Projective Augmented Reality in a Museum: Development and Evaluation of an Interactive Application*. The Eurographics Association, 2020, ISBN: 978-3-03868-111-3. DOI: 10.2312/egve.20201258. [Online]. Available: https://diglib.eg.org:443/xmlui/handle/10.2312/egve20201258 (visited on 06/01/2022).

[45] D. Plecher, M. Ludl, and G. Klinker, "Designing an AR-Escape-Room with Competitive and Cooperative Mode," 2020. DOI: 10.18420/VRAR2020_30. [Online]. Available: http://dl.gi.de/handle/20.500.12116/33433 (visited on 06/01/2022).

[46] C. Eichhorn, A. Jadid, D. A. Plecher, S. Weber, G. Klinker, and Y. Itoh, "Catching the Drone - A Tangible Augmented Reality Game in Superhuman Sports," in *2020 IEEE International Symposium on Mixed and Augmented Reality Adjunct (ISMAR-Adjunct)*, Nov. 2020, pp. 24–29. DOI: 10.1109/ISMAR-Adjunct51615.2020.00022.

[47] T. F. Bissyandé, F. Thung, D. Lo, L. Jiang, and L. Réveillère, "Popularity, Interoperability, and Impact of Programming Languages in 100,000 Open Source Projects," in *2013 IEEE 37th Annual Computer Software and Applications Conference*, Jul. 2013, pp. 303–312. DOI: 10.1109/COMPSAC.2013.55.

[48] K. Srinath, "Python–the fastest growing programming language," *International Research Journal of Engineering and Technology (IRJET)*, vol. 4, no. 12, pp. 354–357, 2017.

[49] Maximilian Schmidt, *Ubi-Interact Python Node*, Jan. 29, 2022. [Online]. Available: https://github.com/SandroWeber/ubii-node-python (visited on 06/01/2022).

[50] J. Dorn, "A general software readability model," *MCS Thesis*, vol. 5, pp. 11–14, 2012. [Online]. Available: http://www.cs.virginia.edu/weimer/students/dorn-mcs-paper.pdf.

[51] S. Scalabrino, M. Linares-Vásquez, R. Oliveto, and D. Poshyvanyk, "A comprehensive model for code readability," *Journal of Software: Evolution and Process*, vol. 30, no. 6, e1958, 2018, e1958 smr.1958, ISSN: 2047-7481. DOI: 10.1002/smr.1958. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.1958 (visited on 06/01/2022).

[52] "Python Generated Code | Protocol Buffers," Google Developers. (Jun. 2, 2022), [Online]. Available: https://developers.google.com/protocol-buffers/docs/reference/python-generated (visited on 06/03/2022).

[53] G. van Rossum, Barry Warsaw, and Nick Coghlan. "PEP 8 – Style Guide for Python Code." (Jul. 5, 2001), [Online]. Available: https://peps.python.org/pep-0008/ (visited on 06/02/2022).

[54] *Protocol Buffers - google.protobuf.internal.python_message*, Protocol Buffers, Jun. 2, 2022. [Online]. Available: https://github.com/protocolbuffers/protobuf/blob/67f46d249565b5002795c164625eb237437c966a/python/google/protobuf/internal/python_message.py (visited on 06/02/2022).

[55] Ivan Levkivskyi, Jukka Lehtosalo, and Lukasz Langa. "PEP 544 – Protocols: Structural subtyping (static duck typing)." (Mar. 5, 2017), [Online]. Available: https://peps.python.org/pep-0544/ (visited on 06/07/2022).

[56] G. van Rossum, Jukka Lehtosalo, and Lukasz Langa. "PEP 484 – Type Hints." (Sep. 29, 2014), [Online]. Available: https://peps.python.org/pep-0484/#generics (visited on 06/11/2022).

[57] "Data model — Python 3 documentation." (2022), [Online]. Available: https://docs.python.org/3/reference/datamodel.html (visited on 06/11/2022).

[58] *μpb: Small, fast C protos*, Protocol Buffers, Jun. 30, 2022. [Online]. Available: https://github.com/protocolbuffers/upb (visited on 06/30/2022).

[59] N. Koorapati, *Nipunn1313/mypy-protobuf*, Jun. 8, 2022. [Online]. Available: https://github.com/nipunn1313/mypy-protobuf (visited on 06/11/2022).

[60] Ethan Smith. "PEP 561 – Distributing and Packaging Type Information." (Sep. 9, 2017), [Online]. Available: https://peps.python.org/pep-0561/ (visited on 06/11/2022).

[61] D. G. Taylor, *Better Protobuf / gRPC Support for Python*, Jun. 13, 2022. [Online]. Available: https://github.com/danielgtaylor/python-betterproto (visited on 06/13/2022).

[62] D. G. Taylor, *Issues · danielgtaylor/python-betterproto*, Jun. 13, 2022. [Online]. Available: https://github.com/danielgtaylor/python-betterproto/issues?q=is%3Aopen+is%3Aissue+label%3Abug (visited on 06/13/2022).

[63] *Proto Plus for Python*, Google APIs, Jun. 13, 2022. [Online]. Available: https://github.com/googleapis/proto-plus-python (visited on 06/13/2022).

[64] M. Schmidt, *Ubii-message-formats : "Python Code for Ubi Interact protobuf messages"*, 2022. [Online]. Available: https://github.com/saggitar/ubii-msg-formats.git (visited on 06/14/2022).

[65] M. Schmidt, *Codestare-proto-plus : "Protoc plugin to compile proto plus python classes"*, 2022. [Online]. Available: https://github.com/saggitar/proto-plus-plugin.git (visited on 06/14/2022).

[66] "Protobuf Messages | Google Ads API," Google Developers. (Jun. 29, 2022), [Online]. Available: https://developers.google.com/google-ads/api/docs/client-libs/python/protobuf-messages (visited on 06/14/2022).

[67] "Google.protobuf.descriptor_pool — Protocol Buffers 3.17.0 documentation." (2008), [Online]. Available: `https://googleapis.dev/python/protobuf/latest/google/protobuf/descriptor_pool.html#module-google.protobuf.descriptor_pool` (visited on 06/16/2022).

[68] M. Schmidt. "Getting started — ubii-message-formats documentation." (2022), [Online]. Available: `https://ubii-msg-formats.readthedocs.io/en/feature-python/getting-started.html` (visited on 06/16/2022).

[69] M. Schmidt, *Generic-proto-plus-stubs: Manually updated type stubs for proto-plus with some generics*, 2022. [Online]. Available: `https://github.com/saggitar/generic-proto-plus-stubs` (visited on 06/17/2022).

[70] "Requests · SandroWeber/ubi-interact Wiki," Requests · SandroWeber/ubi-interact Wiki. (2021), [Online]. Available: `https://github.com/SandroWeber/ubi-interact/wiki/Requests`.

[71] James Knight. "Python's Super Considered Harmful." (2021), [Online]. Available: `https://fuhm.net/super-harmful/` (visited on 06/22/2022).

[72] R. Hettinger. "Python's super() considered super!" Deep Thoughts by Raymond Hettinger. (May 26, 2011), [Online]. Available: `https://rhettinger.wordpress.com/2011/05/26/super-considered-super/` (visited on 06/22/2022).

[73] E. Gamma, R. Helm, R. Johnson, R. E. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Deutschland GmbH, 1995, 512 pp., ISBN: 978-3-8273-3043-7. Google Books: `tmNNfSkfTlcC`.

[74] "Hollywood Principle." (Aug. 27, 2013), [Online]. Available: `http://wiki.c2.com/?HollywoodPrinciple` (visited on 06/22/2022).

[75] "Objects, values and types — Python 2 Documentation." (Mar. 7, 2005), [Online]. Available: `https://web.archive.org/web/20050307224942/http://www.python.org/doc/current/ref/objects.html` (visited on 06/22/2022).

[76] M. Schmidt. "Ubii.framework.util.functools module — ubii-node-python documentation." (2022), [Online]. Available: `https://ubii-node-python.readthedocs.io/en/develop/api/ubii/ubii.framework.util.functools.html` (visited on 06/25/2022).

[77] "Data model — Coroutine Objects — Python 3 Documentation." (2022), [Online]. Available: `https://docs.python.org/3/reference/datamodel.html#coroutine-objects` (visited on 06/23/2022).

[78] M. Schmidt. "Codestare.async_utils.wrapper module — ubii-node-python documentation." (2022), [Online]. Available: https://ubii-node-python.readthedocs.io/en/develop/api/codestare/codestare.async_utils.wrapper.html (visited on 06/24/2022).

[79] M. Schmidt, *Codestare-async-utils: "Utility modules for async development"*, 2022. [Online]. Available: https://github.com/saggitar/.git (visited on 06/24/2022).

[80] "Contextlib — AsyncExitStack — Python 3 Documentation." (2022), [Online]. Available: https://docs.python.org/3/library/contextlib.html#contextlib.AsyncExitStack (visited on 06/25/2022).

[81] M. Schmidt. "Codestare.async_utils.nursery module — ubii-node-python documentation." (2022), [Online]. Available: https://ubii-node-python.readthedocs.io/en/develop/api/codestare/codestare.async_utils.nursery.html (visited on 06/25/2022).

[82] "Dataclasses — Data Classes — Python 3 documentation." (2022), [Online]. Available: https://docs.python.org/3/library/dataclasses.html (visited on 06/25/2022).

[83] M. Schmidt. "Ubii namespace — ubii-node-python documentation." (2022), [Online]. Available: https://ubii-node-python.readthedocs.io/en/develop/api/ubii/ubii.html (visited on 06/29/2022).

[84] M. Schmidt. "Ubii.framework.client module — ubii-node-python documentation." (2022), [Online]. Available: https://ubii-node-python.readthedocs.io/en/develop/api/ubii/ubii.framework.client.html (visited on 06/29/2022).

[85] M. Schmidt. "Ubii.framework.processing module — ubii-node-python documentation." (2022), [Online]. Available: https://ubii-node-python.readthedocs.io/en/develop/api/ubii/ubii.framework.processing.html (visited on 06/29/2022).

[86] "Processing Modules Tutorial — ubii-node-python documentation." (2022), [Online]. Available: https://ubii-node-python.readthedocs.io/en/develop/concepts/processing.html (visited on 06/29/2022).

[87] "Ubii.node package — ubii-node-python documentation." (2022), [Online]. Available: https://ubii-node-python.readthedocs.io/en/develop/api/ubii/ubii.node.html (visited on 06/29/2022).

[88] M. Schmidt. "Getting started — CLI — ubii-node-python documentation." (2022), [Online]. Available: `https://ubii-node-python.readthedocs.io/en/develop/getting-started.html#cli` (visited on 06/29/2022).

[89] M. Schmidt. "Ubii.cli.main module — ubii-node-python documentation." (2022), [Online]. Available: `https://ubii-node-python.readthedocs.io/en/develop/api/ubii/ubii.cli.main.html` (visited on 06/29/2022).

[90] M. Schmidt. "Getting started — Example Client — ubii-node-python documentation." (2022), [Online]. Available: `https://ubii-node-python.readthedocs.io/en/develop/getting-started.html#client-example` (visited on 06/29/2022).

[91] "Home | Read the Docs." (2022), [Online]. Available: `https://readthedocs.org/` (visited on 07/30/2022).

[92] A. A. G. Alex, S. Jegatha, J. G. Jayanthi, and S. A. Rabara, "SaaS Framework for Library Augmented Reality Application," in *2014 World Congress on Computing and Communication Technologies*, Feb. 2014, pp. 8–12. DOI: `10.1109/WCCCT.2014.58`.

[93] K. Vinumol, A. Chowdhury, R. Kambam, and V. Muralidharan, "Augmented Reality Based Interactive Text Book: An Assistive Technology for Students with Learning Disability," in *2013 XV Symposium on Virtual and Augmented Reality*, May 2013, pp. 232–235. DOI: `10.1109/SVR.2013.26`.

[94] Z. Li, M. Annett, K. Hinckley, K. Singh, and D. Wigdor, "HoloDoc: Enabling Mixed Reality Workspaces that Harness Physical and Digital Content," in *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, ser. CHI '19, New York, NY, USA: Association for Computing Machinery, May 2, 2019, pp. 1–14, ISBN: 978-1-4503-5970-2. DOI: `10.1145/3290605.3300917`. [Online]. Available: `https://doi.org/10.1145/3290605.3300917` (visited on 07/02/2022).

[95] V. Fragoso, S. Gauglitz, S. Zamora, J. Kleban, and M. Turk, "TranslatAR: A mobile augmented reality translator," in *2011 IEEE Workshop on Applications of Computer Vision (WACV)*, Jan. 2011, pp. 497–502. DOI: `10.1109/WACV.2011.5711545`.

[96] T. Toyama, D. Sonntag, A. Dengel, T. Matsuda, M. Iwamura, and K. Kise, "A mixed reality head-mounted text translation system using eye gaze input," in *Proceedings of the 19th international conference on Intelligent User Interfaces*, ser. IUI '14, New York, NY, USA: Association for Computing Machinery, Feb. 24, 2014, pp. 329–334, ISBN: 978-1-4503-2184-6. DOI: 10.1145/2557500.2557528. [Online]. Available: https://doi.org/10.1145/2557500.2557528 (visited on 07/02/2022).

[97] "Artoolkitx/artoolkitx: artoolkitX." (2020), [Online]. Available: https://github.com/artoolkitx/artoolkitx (visited on 07/02/2022).

[98] "Cloud Computing Services | Microsoft Azure." (2022), [Online]. Available: https://azure.microsoft.com/en-us/ (visited on 07/02/2022).

[99] "UNLV tests on Tesseract," tessdoc. (2020), [Online]. Available: https://tesseract-ocr.github.io/tessdoc/UNLV-Testing-of-Tesseract.html (visited on 07/02/2022).

[100] Fayez, *Tesserocr*, Jun. 29, 2022. [Online]. Available: https://github.com/sirfz/tesserocr (visited on 07/02/2022).

[101] J. Matas, O. Chum, M. Urban, and T. Pajdla, "Robust wide-baseline stereo from maximally stable extremal regions," *Image and Vision Computing*, British Machine Vision Computing 2002, vol. 22, no. 10, pp. 761–767, Sep. 1, 2004, ISSN: 0262-8856. DOI: 10.1016/j.imavis.2004.02.006. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0262885604000435 (visited on 07/02/2022).

[102] X. Zhou, C. Yao, H. Wen, Y. Wang, S. Zhou, W. He, and J. Liang, "EAST: An Efficient and Accurate Scene Text Detector," presented at the Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2017, pp. 5551–5560. [Online]. Available: https://openaccess.thecvf.com/content_cvpr_2017/html/Zhou_EAST_An_Efficient_CVPR_2017_paper.html (visited on 07/02/2022).

[103] *Opencv*, OpenCV, Jul. 2, 2022. [Online]. Available: https://github.com/opencv/opencv (visited on 07/02/2022).

[104] "OpenCV: Samples/dnn/text_detection.cpp." (Jul. 3, 2022), [Online]. Available: https://docs.opencv.org/4.x/db/da4/samples_2dnn_2text_detection_8cpp-example.html#_a4 (visited on 07/02/2022).

[105] "Entry points specification — Python Packaging User Guide." (Jul. 1, 2022), [Online]. Available: https://packaging.python.org/en/latest/specifications/entry-points/ (visited on 07/02/2022).

[106] "Releases · simonflueckiger/tesserocr-windows_build." (2017), [Online]. Available: https://github.com/simonflueckiger/tesserocr-windows_build/releases (visited on 07/03/2022).

[107] "Tesseract/tesseract.1.asc at main · tesseract-ocr/tesseract." (Sep. 13, 2021), [Online]. Available: https://github.com/tesseract-ocr/tesseract/blob/main/doc/tesseract.1.asc (visited on 07/26/2022).

[108] "OpenCV: Cv::MSER Class Reference." (Aug. 5, 2022), [Online]. Available: https://docs.opencv.org/3.4/d3/d28/classcv_1_1MSER.html (visited on 07/26/2022).

[109] J. Lin, G. Sun, T. Cui, J. Shen, D. Xu, G. Beydoun, P. Yu, D. Pritchard, L. Li, and S. Chen, "From ideal to reality: Segmentation, annotation, and recommendation, the vital trajectory of intelligent micro learning," *World Wide Web*, vol. 23, no. 3, pp. 1747–1767, May 1, 2020, ISSN: 1573-1413. DOI: 10.1007/s11280-019-00730-9. [Online]. Available: https://doi.org/10.1007/s11280-019-00730-9 (visited on 08/04/2022).

[110] J. Qian, Q. Sun, C. Wigington, H. L. Han, T. Sun, J. Healey, J. Tompkin, and J. Huang, "Dually Noted: Layout-Aware Annotations with Smartphone Augmented Reality," in *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, ser. CHI '22, New York, NY, USA: Association for Computing Machinery, Apr. 29, 2022, pp. 1–15, ISBN: 978-1-4503-9157-3. DOI: 10.1145/3491102.3502026. [Online]. Available: https://doi.org/10.1145/3491102.3502026 (visited on 08/04/2022).

[111] X. Zhong, J. Tang, and A. Jimeno Yepes, "PubLayNet: Largest Dataset Ever for Document Layout Analysis," in *2019 International Conference on Document Analysis and Recognition (ICDAR)*, Sep. 2019, pp. 1015–1022. DOI: 10.1109/ICDAR.2019.00166.

[112] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft COCO: Common Objects in Context," in *Computer Vision – ECCV 2014*, D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, Eds., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2014, pp. 740–755, ISBN: 978-3-319-10602-1. DOI: 10.1007/978-3-319-10602-1_48.

[113] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, Jun. 2009, pp. 248–255. DOI: 10.1109/CVPR.2009.5206848.

[114] R. Wang, Y. Fujii, and A. C. Popat, "Post-OCR Paragraph Recognition by Graph Convolutional Networks," presented at the Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision, 2022, pp. 493–502. [Online]. Available: `https://openaccess.thecvf.com/content/WACV2022/html/Wang_Post-OCR_Paragraph_Recognition_by_Graph_Convolutional_Networks_WACV_2022_paper.html` (visited on 08/04/2022).