Technical University of Berlin

Faculty V - Mechanical Engineering and Transport Systems
Department of Machine Tools and Factory Management
Divison of Industrial Automation

Pascalstr. 8-9
10587 Berlin
https://www.iat.tu-berlin.de

Master's Thesis

Automatic Generation of Object Detection Services with Varying Detection Methods and Interfaces

Nikolas Keuck
388015
nikolas.keuck@gmail.com
Major: Computational Engineering Sciences

August 22, 2019

Referee: Prof. Dr.-Ing. Jörg Krüger
Tutor: Dipl.-Ing. Martin Rudorfer

Acknowledgments

First of all, I would like to thank my supervisor Martin Rudorfer for his constant flow of helpful ideas, high responsiveness and willingness to take time to support me.. It sums up to 15 hours of face to face meetings, 1.5 hours of video calls, 42 e-mails and unknown time for proofreading.

To my colleagues, namely Wilma, Marian and Nils: Thank you for being so co-operative. You made working full time and writing a master's thesis possible!

Last but not least, I would like to thank my family and friends who accompanied me over the years of my studies:

Thank you!

Hereby, I declare that I wrote this thesis myself with the help of no more than the mentioned literature and auxiliary means.

Berlin, August 22nd, 2019

.....

(Signature)

Abstract

Contemporary manufacturing relies on monolithic object detection (OD) software architecture to achieve high-performance automation. As production machines typically have a life span of over 20 years, object detection aligns to this rigid pattern. Typically, necessary OD hardware is delivered along with the software. In the last decade, OD research leaped forward while product life cycles shortened concurrently. Thus, it is desirable to get more frequent updates. Subsequently, the goals of this thesis are eliminating hardware and saving maintenance costs while simultaneously keeping up to date with state of the art OD methods. This thesis introduces *Recipe Generator*, a framework based on service-oriented architectures. It trains OD services stored in a Docker hub using a computer-aided design file. Then, it combines pairs of trained OD services and camera image acquisition services to recipes. Recipes are transferred, prepared and executed via an open platform communication unified architecture (OPC UA) vision server for easy manufacturing integration. The services are dockerized and communicate via Google remote procedure calls or representational state transfer. The main advantages of this approach are the flexible reuse of existing OD methods and the ease to add new ones in a preferred language and platform. There are two main challenges. The first is to increase the performance, reliability and availability of the framework by, e.g., time-sensitive networking. The second is to enhance the framework with a catalogue for "plug-and-play" services.

Zusammenfassung

Die moderne Fertigung setzt auf monolithische Softwarearchitekturen in der Objekterkennung (OE), um eine leistungsstarke Automatisierung zu erreichen. Da Produktionsmaschinen in der Regel eine Lebensdauer von über 20 Jahren haben, richtet sich die Objekterkennung nach diesem starren Muster. In der Regel wird die erforderliche OE-Hardware zusammen mit der Software geliefert. In den letzten zehn Jahren hat die OE-Forschung einen Sprung nach vorne gemacht, während sich gleichzeitig die Produktlebenszyklen verkürzt haben. Daher ist es wünschenswert, häufigere Aktualisierungen zu erhalten. Ziel dieser Arbeit ist es, Hardware zu eliminieren, Wartungskosten zu senken und gleichzeitig auf dem neuesten Stand der OE-Methoden zu bleiben. Diese Arbeit stellt *Recepy Generator* vor, eine Rahmenstruktur (Framework), das auf Dienste-orientierten Architekturen basiert. Es trainiert OE-Services, die in einem Docker-Hub gespeichert sind, mit einer computergestützten Designdatei (CAD). Anschließend werden Paare von trainierten OE-Diensten und Bilderfassungsdiensten zu Rezepten kombiniert. Die Rezepte werden über einen OPC UA Vision-Server in die Fertigung übertragen und dort vorbereitet und ausgeführt. Die Dienste sind dockerisiert und kommunizieren über Googles Fernprozeduraufrufe (gRPC) oder REST. Die Hauptvorteile dieses Ansatzes sind die flexible Wiederverwendung vorhandener OE-Methoden und das einfache Hinzufügen neuer Methoden in einer bevorzugten Sprache und Plattform. Es gibt zwei Hauptherausforderungen. Die erste besteht darin, die Leistung, Zuverlässigkeit und Verfügbarkeit des Frameworks durch z.B. zeitsensitive Vernetzung (TSN) zu erhöhen. Die zweite besteht darin, das Framework mit einem Katalog für sofort betriebsbereite Dienste zu erweitern.

Contents

List of Figures

List of Tables

List of Acronyms

API	Application Programming Interface
ATAM	Architecture Tradeoff Analysis Method
CAD	Computer Aided Design
CAP	Consistency, Availability, Partition Tolerance
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
CRUD	Create, Retrieve, Update, Delete
DCOM	Distributed Component Object Model
EMVA	European Machine Vision Association
ESB	Enterprise Service Bus
GCV	Google Cloud Vision
GraphQL	Graph Query Language
gRPC	Google Remote Procedure Call
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
ISO	International Organization for Standardization
IT	Information Technology
JPG	Joint Photographic (Experts) Group
JSON	JavaScript Object Notation
M2M	Machine to Machine
MQTT	Message Queuing Telemetry Transport
MT	Messaging Technology
OD	Object Detection
ODS	OD Service
ODM	OD Method
OPC UA	Open Platform Communication Unified Architecture
OPC UA VC	OPC UA Vision Client

OPC UA ViS	OPC UA Vision Server
OSI	Open Systems Interconnection
OT	Operation Technology
PLC	Programmable Logic Controller
RAM	Read Access Memory
REST	Representational State Transfer
RG	Recipe Generator
RGB	Red Green Blue - Depth
RPC	Remote Procedure Call
SOAP	Simple Object Access Protocol
TCP	Transmission Control Protocol
TSN	Time Sensitive Networking
UDP	User Datagram Protocol
URL	Uniform Resource Locator
UUID	Universally Unique Identifier
VDMA	Verband Deutscher Maschinen- und Anlagenbau (Unternehmen)
VM	Virtual Machine
VS	Vision System
XML	Extended Markup Language

Important Terms in this Thesis

Interface	Enables communication between multiple applications (inter-process communication) on the same or different computer. Examples are DCOM, RPC and REST. Accordingly, in the context of this thesis, "interface" refers to data-oriented instead of functional communication module. The latter is called interface *semantic*.
Object Detection System	A process of camera image acquisition, detecting an object inside that camera image and returning its pose.
Recipe	Describes properties, procedures and parameters for a machine vision job. [1]
Service	A mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description. [2]

Chapter 1

Introduction

1.1 Context and Problem

There is a vast variety of different object detection (OD) algorithms reaching from feature-based to template-based methods, and more recently, also deep-learning-based approaches. Especially the latter have conquered the field very rapidly and consistently improved the state of the art in many established OD benchmarks [3], [4]. However, this improvement has not yet arrived at the shop floors of manufacturing companies. This is in part due to characteristics of the method, such as the large amount of required training data. Moreover, another reason is that the current machine vision infrastructure is not flexible enough to keep track of the dynamic development in the field of OD. Deploying or even only testing a new method typically requires a certain investment in cost and time that has to be justified very well. This workflow can be greatly accelerated by using service-oriented architectures (SOA) [5].

A recent example of how OD can be integrated into manufacturing is depicted in figure 1.1. A robot arm is equipped with two cameras for movement guidance. If the cylindrical objects at the bottom of the picture are to be picked, their poses have to be detected. As object detection methods (ODM) are best suited for a specific class of objects (e.g., rotationally symmetric), it is likely that with a new part type an ODM exchange is necessary. For every new object, new training data has to be gathered. This is a tedious task. Moreover, the software is usually coupled tightly

to the hardware, so the machine manufacturer has to be contacted for a change of software or even hardware.

Figure 1.1: A Fanuc iRVision robot is guided by stereo-vision cameras [6].

Assuming a SOA is used for the robot guidance software, the main challenges are decoupling the software of the hardware, the composition of the services and meeting real-time requirements. Real-time in this context does not mean "as fast as possible" but rather "right on time", i.e., at every discrete timestep the system state must be known.

1.2 Objectives

The goal of this thesis is to simplify the process of programming an object detector and integrating it into the manufacturing process. In fact, the main objective is to avoid any programming – instead, a framework shall be proposed that allows generating an OD service from a single example image or computer-aided design (CAD) file.

The service should detect the object with a specified method and should have

the appropriate interfaces for convenient integration into SOA. The key advantage is that new methods can be tested and deployed without any effort. The manufacturer can generate and deploy a service which has the same interface as the old one but uses an updated method.

The approach of this thesis is to use as many established protocols and semantic standards as possible to achieve maximum interoperability. This is the base for fast testing and deployment of new components. Ultimately, the thesis intends to create a library of reusable "plug-and-work" ODMs and other components. In this context, the open platform communication (OPC) foundation as a conglomerate of key players from different divisions in industry is of great help. It pursues vendor-independent communication between machines. As part of its work, it releases semantic standards, one of which is OPC unified architecture (OPC UA) Vision [1]. It abstracts processes in machine vision systems. While the specification is in draft state, this work adheres to it as much as possible on the concept level and to some extent in the implementation.

One central aspect of OPC UA Vision is recipe management. Recipes describe properties, procedures and parameters for a machine vision job. As they remain vendor specific in OPC UA Vision, one challenge is abstracting ODM procedures. A proposition of two general OD phases is given in this thesis.

Deployment and composition of services should also not be disregarded. Docker is used as virtualization technology and store for services. The composition is simple: camera and OD services (ODS) are called in sequence via the recipe.

1.3 Delimitation

During the design phase of a SOA-based framework (or any other software framework), numerous aspects have to be considered. Two important ones in manufacturing are safety and security [7]. They are not completely excluded in this thesis, but only considered to a very limited extent.

Also, the evaluation of the framework is not yet quantified as it is dependent on the ODM in use and moreover where the components are deployed. To thoroughly test this, a testbed inside an industrial environment must be available.

In operation technology, real-time applications are crucial [8]. As the framework tends to connect information and operation technology domains, it is desirable to illuminate real-time aspects, too. As of now, the standardization process for connecting both domains through, e.g., time-sensitive networking (TSN) is not advanced enough to include it in this work.

1.4 Structure

The thesis is structured as follows: The first chapter explains the problem, its relevance and outlines the objectives of the thesis. Chapter 2 provides the necessary background information for this work and gives an overview of recent research and related work on SOAs and OD. Chapter 3 introduces the concept. The pursued approaches for the proposed framework are described in detail. Chapter 4 is concerned with the implementation. It comprises the rationale of the used programming language and the explanation of the software architecture with the help of sequence, package and class diagrams. In Chapter 5, an evaluation method for SOAs is introduced and applied to the framework. The results are discussed and conceptual weaknesses and possible improvements are considered. The thesis is concluded by Chapter 6, which also points to further improvements and future research directions.

Chapter 2

State of the Art

This chapter introduces the necessary background knowledge to follow the concept and implementation of this thesis. OD illuminates an abstract approach of pose detection and how image processing is currently present in manufacturing. Furthermore, important aspect of SOAs will be introduced, especially service interfaces, semantics and deployment options. Lastly, OPC UA Vision as a provider for an information model and a state machine for vision systems illustrates the current efforts in vendor interoperability in machine vision.

2.1 Object Detection

OD is a computer technology for identifying instances of objects in digital images or videos [7]. As one of the goals of this thesis is to empower manufacturers to exchange object detection methods (ODM) quickly, there are two aspects which need to be considered. First, the current state of OD in manufacturing should be summarized for an understanding of what the proposed framework has to cope with. Second, contemporary object detectors should be reviewed to find a least common denominator for an interface.

2.1.1 Image Processing in Manufacturing

Image processing tasks in manufactoring are categorized into evaluations such as inspection, monitoring, verification and recognition [7]. For a long time, image processing systems were used to identify bad or incomplete parts which were then scrapped. The systems were not used for complex tasks and the result of the process was usually boolean. With progress in OD, value adding processes have enhanced image processing tasks. For example, robots can be guided with depth cameras or a welding process can be not only monitored but also looped back and controlled. OD serves as the eyes and - to some extent - the brain for the robot. Challenges in OD such as bin picking can now be tackled.

Furthermore, integration of an image processing system in manufacturing (and other domains) is a highly specific task. Usually, a machine vision expert has to program and configure the system for every new machine type or even every machine. Small companies cannot afford such experts [7]. They are capable of maintaining the system and other repetitious tasks when the documentation is suitable. Subsequently, system integrators tend to offer rigid turn-key-solutions with a specific user interface. On the inside, the frameworks that system integrators sell are made of reusable applications [7].

2.1.2 Two Phases of Object Detection

Coming from a world of mostly black and white 2D images, OD research has recently advanced to colored depth 3D images through rising computing power [7]. With the added dimension, it is possible to determine the pose of an object in a camera image. Pose detection can be split into two parts: a 3D model of an object serves as training input to generate necessary features or templates (phase 1). After training, the object can be detected in RGB-D images with the help of the training output (phase 2).

Phase 1 is a routine that is necessary for successful pose detection. More than a thousand templates can be generated in this phase. Without the use of automation, the cost/benefit calculation of this phase would never call for using OD. Every template including the angle from which it was acquired would have to be measured and mapped to the template. Subsequently, in recent algorithms, CAD files were

used for this purpose.

An example method of phase 2 is template matching (TM). Fig. 2.1 illustrates the process. In the left image, the face of the man is to be found. The template is the little cut-out in the middle. Pixel by pixel, the template is being convoluted with the original image and rated with a metric. The resulting resolution matrix is depicted on the right. Bright areas indicate potential findings. At the brightest point, the template is rightfully suggested. [9]

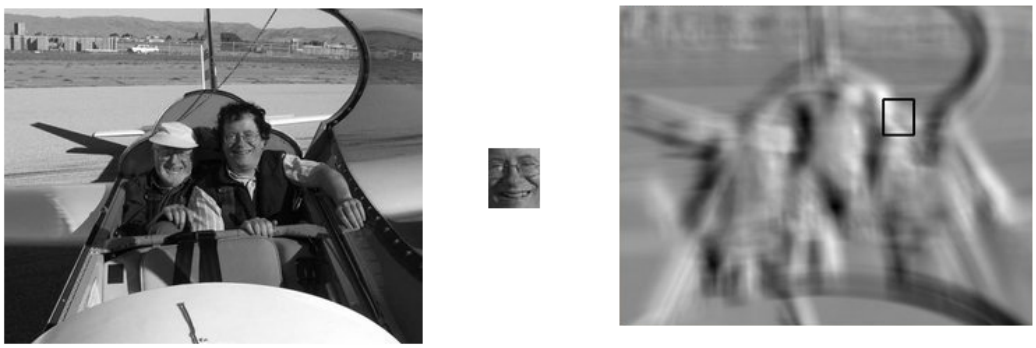


Figure 2.1: In the picture on the left, the template in the middle is to be found. Depicted on the right is the resolution matrix with potential findings indicated with the bright color and the found area of the template in the original image. [9]

Especially in the last decade, 6D pose estimation gained popularity ([10], [11] [12]). In 2018, Hodaň benchmarked 15 ODMs which share the approach of two-phased pose detection [13]. He categorized the methods in template, point-pair-feature, local-feature and learning-based. His evaluation showed that point-pair-feature-based methods currently perform best measured against a pose-error function that deals with pose ambiguities.

To summarize, pose detection is generally split into a preparation and a detection phase. When designing a framework dealing with quickly changing ODMs, this ODM-agnostic process description should be taken into account.

2.2 Service-Oriented Architectures

SOA is a software design paradigm which gained importance towards monolithic approaches. Its main advantages over monolithic approaches are scalability, decoupling of components and simple development, testing and deployment [14]. Services are concise, decoupled components capable of one specific task [15]. The goal is to design services for maximum reusability. Together, either orchestrasted or choreographing, they form an application.

The following subsections focus on how services can be interfaced and deployed.

2.2.1 Service Interfaces

In this section, potential client-server-based interfaces and underlying protocols shall be discussed. The evaluated interfaces are *Advanced Message Queuing Protocol* (AMQP), *Message Queuing Telemetry Transport* (MQTT), *Representational State Transfer* (REST), *Google Remote Procedure Calls* (gRPC), *Graph Query Language* (GraphQL) and *Open Platform Communication Unified Architecture* (OPC UA).

AMQP and MQTT

Both AMQP 0.x and MQTT are broker based protocols specialized for machine-to-machine (M2M) communication. Clients can be sensors, programmable logic controllers, etc.; the server is a broker connecting the clients. A broker is a central instance mediating between parties. Clients can subscribe to various message queues called topics. Telemetry data can then be published and read from these topics handled by the broker. The clients dynamically change between publisher and subscriber. Figure 2.2 illustrates the MQTT architecture. [16]

A commonly used message broker is RabbitMQ which supports AMQP 0.x natively and MQTT via a plugin [18].

AMQP needs to be distinguished between version 0.x and 1.0, as the underlying messaging paradigm has been completely revised. While for 0.x strict publishing/-subscription messaging is required, version 1.0 is based on a peer-to-peer connection where a broker is not required, although possible. Due to the more sophisticated

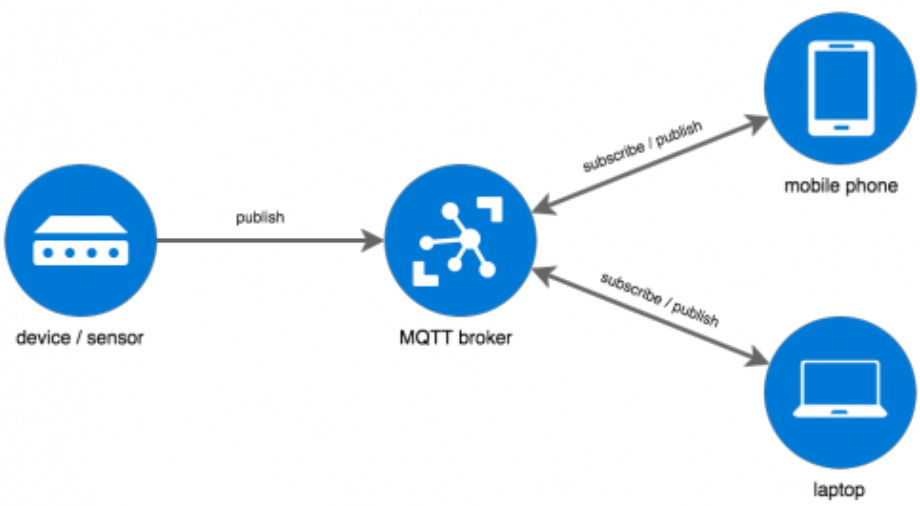


Figure 2.2: MQTT Architecture with the broker in the middle as a mediator between the clients [17].

version of AMQP 1.0, fewer implementations exist. [19]

REST

REST is an architectural paradigm describing how distributed systems can communicate with each other. It consists of five mandatory- and one optional restriction/s. If any of the five mandatory restrictions is violated, an architecture cannot be RESTful. The restrictions are client–server architecture, statelessness, cacheability, layered system, uniform interface and code on demand (optional). Roy Fielding developed REST alongside HTTP/1.1 and although it is not dependent on it, HTTP/1.1 is the primarily used protocol to implement REST. Thus, many web pages fulfill these restrictions naturally. REST messages are usually human-readable JSON files. Unlike MQTT or AMQP 0.x, REST does not rely on a broker. [20]

gRPC

For many cases in the past, it was hard for maintainers to adhere to all REST principles due to its strict nature. Moreover, REST is usually implemented with HTTP/1.1. In 2015, HTTP/2 was released to address the flaws of its predecessor [21]. Among those are the lack of ability of constant data streaming and latency

issues.

gRPC, a remote procedure call technology introduced by Google in 2016, entirely takes advantage of HTTP/2 and thus has some advantages over REST: it allows multiplexing, binary (i.e., quick) data transfer and more. The technology behind gRPC is a remote procedure call, letting the user call remote methods as if they were local, albeit the remote method can be processed on a different hardware or system. gRPC uses protofiles to describe interface semantics. In protofiles, the user specificies service- and method names as well as message types and input/output values. Out of these protofiles, stubs can be generated which are placeholder classes for gRPC clients and servers. Unlike in most implementations of REST, gRPC does not use textual transport data like JSON but relies on Protobuf (short for protocol buffer), a binary buffer [22]. For backwards compatibility towards older clients, there is a gateway available which provides transcoding from HTTP/JSON to gRPC. See figure 2.3 for the concept behind it [23].

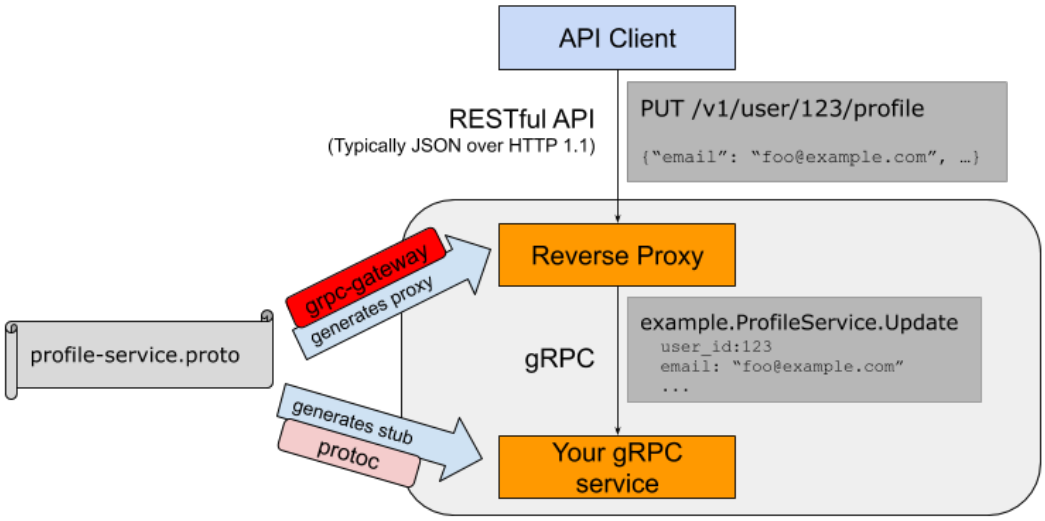


Figure 2.3: gRPC gateway concept [23]. Based on a protofile, it generates a reverse proxy which handles RESTful API client requests and maps them to gRPC methods.

The gateway is a plugin of protoc (short for protocol compiler) as part of gRPC's ecosystem. Based on a given proto file, the gRPC-gateway generates a reverse proxy offering a RESTful interface which internally maps REST/HTTP1.1 to gRPC calls. The mapping between gRPC and HTTP is described here with an example protofile directly quoted from Google's API documentation [24]:

```
service Messaging {
  rpc GetMessage(GetMessageRequest) returns (Message) {
    option (google.api.http) = {
        get: "/v1/{name=messages/*}"
    };
  }
}
message GetMessageRequest {
  string name = 1; // Mapped to URL path.
}
message Message {
  string text = 1; // The resource content.
}
```

The option in the protofile enables HTTP REST to gRPC mapping:

HTTP	gRPC
GET /v1/messages/123456	GetMessage(name: "messages/123456")

GraphQL

GraphQL is a data query and manipulation language. It was developed by Facebook and is open-source since 2015. Compared to REST, it has a more flexible and efficient approach. The increase in efficiency over REST is based on faster mobile data access, and more flexibility for the application programming interface (API) to let clients access precisely the data they need, i.e., the server modifies the data with respect to the clients' needs instead of providing one rigid resource. REST allows the user to pass a single set of arguments - the query parameters and URL segments. In GraphQL, every field and nested object can get its own set of arguments. It also lets the user pass arguments in scalar fields allowing for data transformations. An example query directly quoted from GraphQ's documentation is depicted here [25]:

```
{
  human(id: "1000") {
    name
    height(unit: FOOT)
```

```
5     }
6  }
```

The corresponding response from the server is:

```
1   {
2     "data": {
3       "human": {
4         "name": "Luke Skywalker",
5         "height": 5.6430448
6       }
7     }
8  }
```

OPC UA

OPC UA is a *machine-to-machine* (M2M) protocol specialized in vendor independent communication between heterogeneous machines. Until 2018, there were two means of data exchange available for OPC UA: binary data over transmission control protocol (TCP) or extended markup language (XML) data over *Simple Object Access Protocol* (SOAP). Due to the higher performance, the former is primarily used nowadays. [26]

In 2018, the OPC Foundation introduced part 14 of the OPC UA specification stack, OPC UA Publish/Subscribe [27]. It is used to communicate messages between different system components without these components having to know each other's identity. Now it is possible to use more protocols for data exchange: AMQP, MQTT, OPC UA user datagram protocol (UDP) and OPC UA Ethernet. OPC UA UDP can perform multicasts, i.e., one entity can send data to a group with the same effort as sending to a single entity. Multicasts are more performant than brokers, although do not enable time decoupling of services [28]. OPC UA Ethernet is a simple Ethernet based protocol not relying on IP or UDP [27]. As for the payload of the protocols, JSON or UADP are allowed. UADP is specialized for cyclic communication between, e.g., PLCs. The payload is included directly in the transport protocol.

Eckhardt et. al. performed an evaluation based on expert estimates in 2018 on

how the different combinations of data exchange in OPC UA suit for the specific needs at the field, control, human-machine-interface (HMI) and enterprise level [28]. See figure 2.4 for an overview for the evaluated combinations.

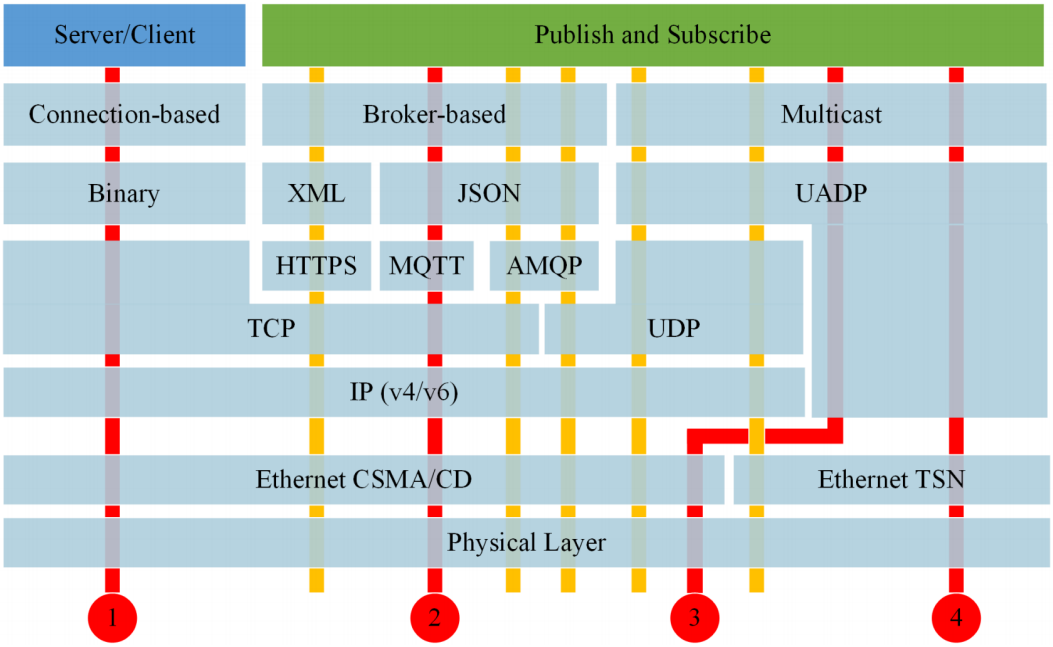


Figure 2.4: OPC UA data exchange variants. The red and yellow vertical lines are possible variants, the red lines are evaluated in detail in [28].

The experts estimated that all combinations are suitable for HMI and enterprise level, but only combinations 3 and 4 are suited for control and field level. Combination 4 uses OPC Ethernet TSN which is OPC UA's approach of eliminating the need for fieldbus protocols [8]. TSN resides on OSI reference model layer 2 as a set of standards[1] to boost ethernet networks with low latency and high availability. So-called **convergent** networks can be implemented with TSN, meaning different components (sensors, PLCs, etc.) can each arrange data streams - from synchronous low-latency to event-based. Streams can be extended or altered at any time. The main components of TSN are time synchronization enabling real-time functionalities, and scheduling and traffic shaping enabling coexistence for multiple streaming classes.
In practice, there are some network adapters and switches available which implement a subset of the TSN standards. The open-source automation development lab

[1] mainly IEEE 802.1Q [29]. See [30] for a full list and detailed description.

which is also responsible for the open-source project open62541[2] currently evaluates performance and prices of the released hardware [31]. While a user currently has to accept a price increase of 1000-2000 % as opposed to conventional hardware, the tests confirm TSN to be as performant as established fieldbus systems [31].

There was also an attempt to create an OPC UA to REST adapter by Ronnhölm in 2018 [32]. Although OPC UA allows HTTP on the application layer, there is an incentive: OPC UA is not RESTful. RESTful HTTP is centered around resources that can be identified by a URL and a message. Any body of data may therefore be manipulated independently of any intermediary application logic. This is not the case with HTTP in OPC UA. Ronnhölm claims ([32], page 26): "To enable exchange with OPC UA servers without the use of an OPC UA stack, a holistic translation must translate both structural and foundational aspects of OPC UA. This means that translation must bypass OPC UA transport protocols, secure channel management, serialization and OPC UA services - all in a way that avoids semantic dependencies and preserves translation transparency." He concludes that a subset of OPC UA services can be transformed to REST when combined with standard create-request-update-delete methods.

2.2.2 Deployment Options

In the last decades, most software applications had a monolithic character which did not focus much on scalability and agile development. With the progress in digitalization, applications had to become more flexible and faster. To address the challenge of deploying applications highly automated, container architectures came about. Unlike virtual machines which need an operating system, runtime and system variables to operate well, Docker and other virtualization technologies are sandbox systems which can imply all the mentioned features and furthermore can run on almost any operating system. In the following, two possible virtualization technologies are briefly introduced, namely Heroku and Docker. They are motivated by pointing out how they are superior to virtual machines in the context of microservices. [33]

Docker is an open-source standard for operating-system-level virtualization. If Docker is installed on an operating system, it is possible to run several applications

[2] An implementation of OPC UA under the Mozilla public license 2.0 as opposed to the dual license model (i.e., proprietary for members, general public license 2.0 for non-members) of the OPC Foundation [31].

on the machine simultaneously, with low start and stop times and little overhead. These applications can rely on different platforms, dependencies, runtimes, etc. The technology behind it is a daemon which shares low-level components with the host operating system. A hypervisor is not necessary (see figure 2.5 for an illustration).

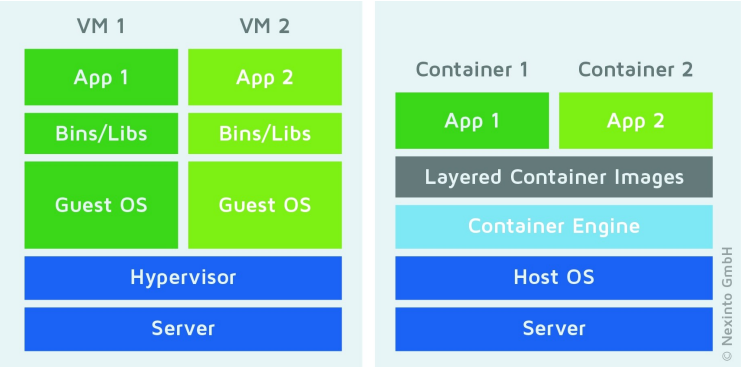


Figure 2.5: Virtual machine architecture on the left versus container architecture on the right. Docker does not rely on a hypervisor. [33]

To virtualize applications, Docker uses containers which are built up on Docker images. Images are read-only templates that are built from a set of instructions written in a dockerfile. They offer one great advantage - a layered file system. This enables the user to easily add or update applications very quickly. Images can be shared in a public/private registry called Docker Hub.
Last but not least, development and operations of applications can be harmonized through a combination of Docker and a continuous integration and continuous delivery platform.

Heroku is a platform as a service provider whose underlying technology shares some core concepts with Docker. E.g., BuildPacks are a set of scripts which are used to set up the final state of an image. The pendant on the Docker side is called dockerfile (see Thurig's blog [34] for a full description of the similarities and table 2.1 for a list of pendants). However, there are also differences between the two alternatives. The main one is the dependency on the Heroku platform on the Heroku side, whereby on the Docker side one is completely flexible in choosing any environment from Raspberry Pi to cloud platform providers like Amazon Web Services. The latter also means a surplus of workload on infrastructure on the Docker side. Also, one is less flexible on the prices. Heroku has a staged price model ranging from 0 to 500 $ per month and dyno. Docker is again more flexible in letting one just

Table 2.1: Similar core concepts of Docker and Heroku. [34]

Docker	Heroku
Dockerfile	BuildPack
Image	Slug
Container	Dyno
Index	Add-Ons
CLI	CLI

paying for the hosting and storaging and leaving the additional features provided by Heroku aside. [35]

2.3 Object Detection Service Interface Semantic

If two humans want to communicate with one another, they need matching channels and need to speak the same language. A channel is a mean of transport for information, e.g., sign language, smoke signs and mobile phones. If one entity tries to call someone via phone if the other does not have a phone or the caller enters the wrong number and the other has its cellphone turned off, they cannot communicate. In case they both have a phone, the called entity answers and both speak a common language (e.g., English), the exchange of information can be achieved. Communication within technical systems faces the same challenges. As for the right channel, models like the open systems interconnection (OSI) basic reference model for information technology standardized by the international organization for standardization (ISO) layer the transfer of information. It ranges from the physical layer consisting of peaks in currents and voltages up until the application layer which includes direct user interaction, resource availability and so forth [36]. This model and the protocols adhering to it ensure that information is delivered safely between communicating entities. However, this model does not imply the semantics of the

payload or the language, as stated in the analogy above.

This section introduces two possible ways of how an ODS service interface semantic can be designed. One resides in the IT and cloud domain whereas the other rather resides in the OT domain.

2.3.1 Google Cloud Vision API

Google Cloud Vision (GCV) API offers a publicly available REST and gRPC interface [37]. OD operations can be executed on the cloud. The upside of this approach is excessive computing power and an attractive pay-per-use price model. The downside is the limitation of the configurability of Google's ODs and the lacking transparency of where and how the data is stored and evaluated. An example gRPC call for annotating images with labels of objects is depicted here:

```
1    rpc AsyncBatchAnnotateFiles(AsyncBatchAnnotateFilesRequest) ↩
   returns (Operation)
```

2.3.2 OPC UA Vision

A currently proposed semantic standard for OD processes is **OPC UA Vision**. It includes a finite state machine abstracting an industrial system from its diverse conditions and transitions. Moreover, it offers an information model covering the administration of recipes, configurations and results. With the help of the state machine, it is defined which information of the information model is retrievable. The content of the three administration objects remains proprietary with the advantage of covering a broad range of OD scenarios.

State Machine

According to the specification, powering up and shutting down a vision system are mandatory processes and thus should be handled in a standardized manner. Also, the handling of errors should be the same for all vision systems. The design of the core operation state, however, shall remain with the manufacturer. Automatic mode as a sub-state of operational mode is one proposed way of designing it. The state

machine for a typical vision system in automatic mode is depicted in figure 2.6. An example of this operation would be a PLC guiding an inspection system for position determination. When powered on, the system enters the preoperational state through loading a configuration marked as active. From there, an operation mode is either automatically chosen by the system or manually triggered. An operation mode is any sub-state machine of the operational state. The automatic mode is chosen and enters the initial state. Then a recipe can be prepared, describing properties, procedures and parameters for a machine vision job. The recipe may include information for a single and/or continuous execution. A single execution would be e.g. determining the pose of an object; a continuous execution could be monitoring and surveillance systems which constantly process and acquire data. When the system is done with an execution or execution step, e.g., taking a picture from one of four angles, it sends results asynchronously to the client. If the system is shut down, it should be put into halt mode first where a safe power-off is assured. From all states, it is possible to enter the error state. Errors are handled aligning with their severity and sometimes need acknowledgment or confirmation by a human before the system can be reset to preoperational state.

2.3.3 Information Model

The information model formally describes all datasets, types, methods, address- and namespaces. See figure 2.8 for an overview and figure 2.7 for an explanation of the notation. Configuration, recipes and result are the three types that have to be dealt with when using OPC UA Vision.

"Even identical vision systems may vary in some details. In order to produce the same results the vision systems have to be adjusted individually e.g., calibrated. Within this document, the set of all parameters that are needed to get the system working is called a configuration. Configurations can be used to align different vision systems that have the same capabilities, so that these systems produce the same results for the same recipes. The ConfigurationManagement handles all configurations that are exposed by the system. Only one configuration can be active at a time. This active configuration affects all recipes used in the machine vision system." ([27], chapter 7.2)

"Properties, procedures and parameters that describe a machine vision task for

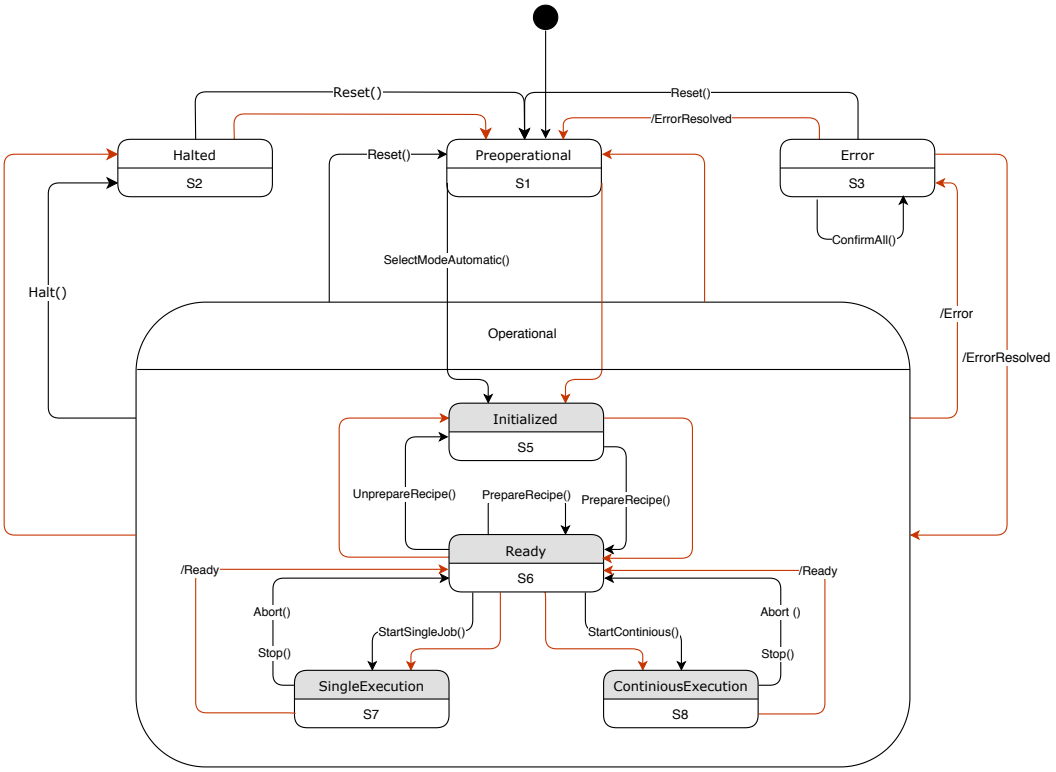


Figure 2.6: OPC UA Vision state machine in automatic operation mode. Red Lines indicate automatic transitions induced by the vision system with optional effects prefixed with a slash. Black lines indicate method induced transitions with the method name as the trigger. The black circle is the entry point of the state machine. All of the states can have optional sub-state machines. States marked in grey are substates. [1]

the vision system are stored in a recipe. Usually there are multiple usable recipes on a vision system. This specification provides methods for activating, loading, and saving recipes. Recipes are handled as binary objects. The interpretation of a recipe is not part of this specification. Recipes are potentially complex entities. A recipe may contain (possibly nested) references to sub-recipes and it may be used for several products. The internal composition of recipes – including the referencing of sub-recipes – is outside the scope of this specification." ([27], appendix 1.2.1 and 1.2.2) For an example recipe life cycle, see appendix B.

"ResultManagementType provides methods to query the results generated by the underlying vision system. Results can be stored in a local result store." ([27], chapter 7.8)

XML nodesets can be used to import information models to an OPC UA server. A screenshot of an example information model uploaded to an OPC UA demo server and viewed by UAExpert is depicted in 2.9. It also shows how methods are called, in this example a simple product of two numbers. An important, more sophisticated method of the OPC UA Vision specification is StartSingleJob (subpart of StateMachineType, not depicted in the information model overview in 2.8). Usually it is called by a PLC to trigger a machine vision task. Its signature consists of following parameters:

StartSingleJob(		
(in)	String	MeasId
(in)	String	PartId
(in)	RecipeIdType	RecipeId
(in)	ProductIdType	ProductId
(out)	String	JobId
(out)	Int32	Error);

In another section of the specification, the data types are defined, e.g. RecipeIdType, which is a structure including an Id, a version and a hash.

When the method is called, it triggers transition from state Ready to SingleExecution in the state machine as depicted in 2.6.

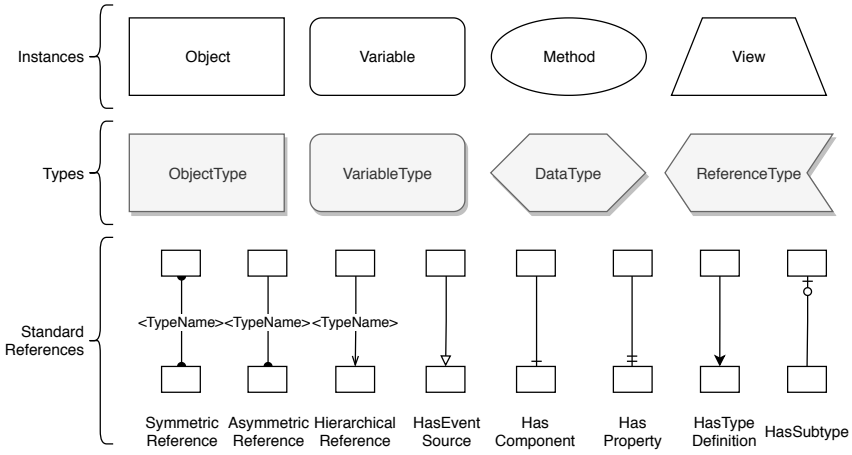


Figure 2.7: OPC UA Vision Information Model Notation.[1]

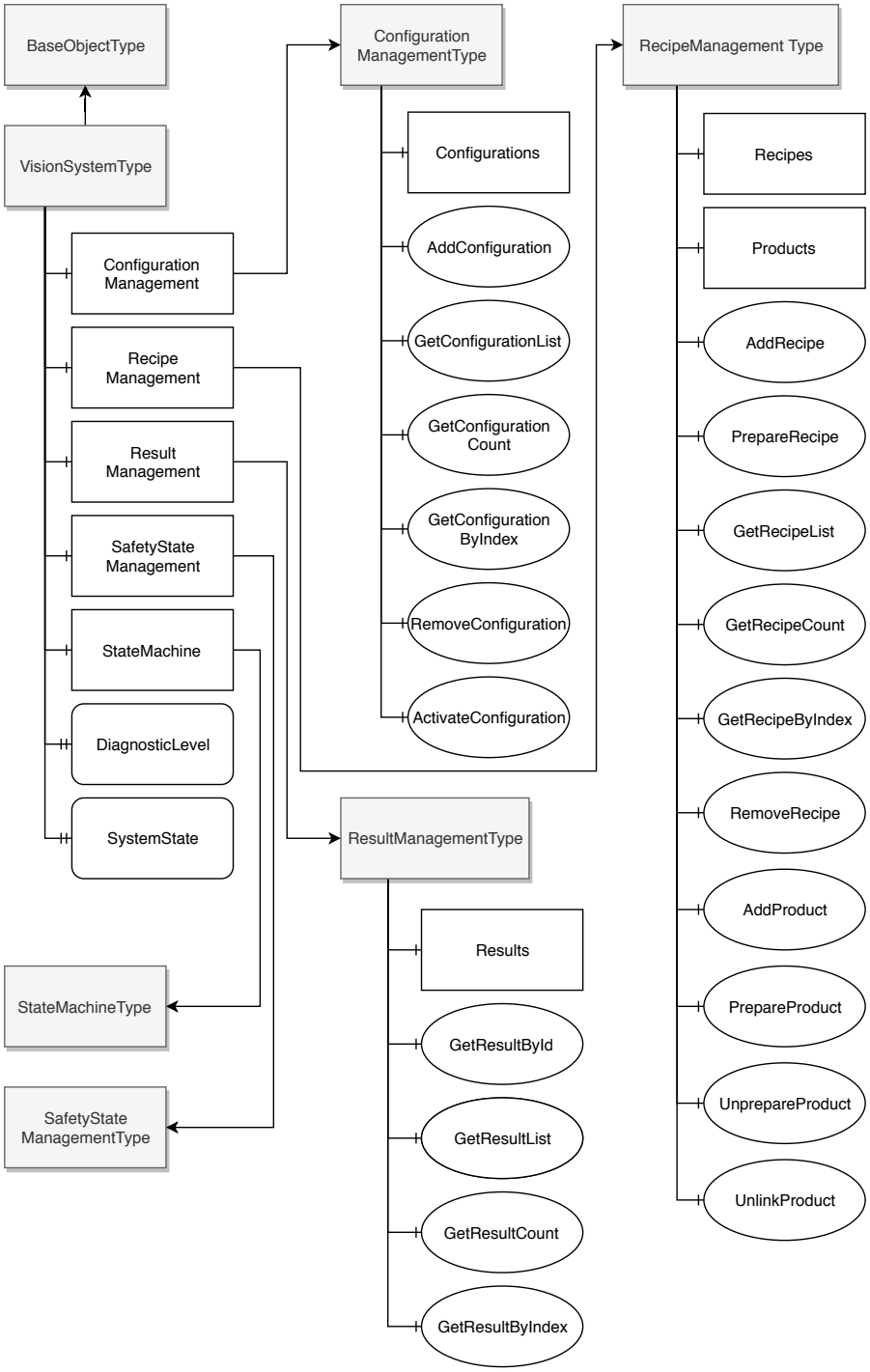


Figure 2.8: OPC UA Vision Information Model Overview. See fig. 2.7 for a description of the notation. [1]

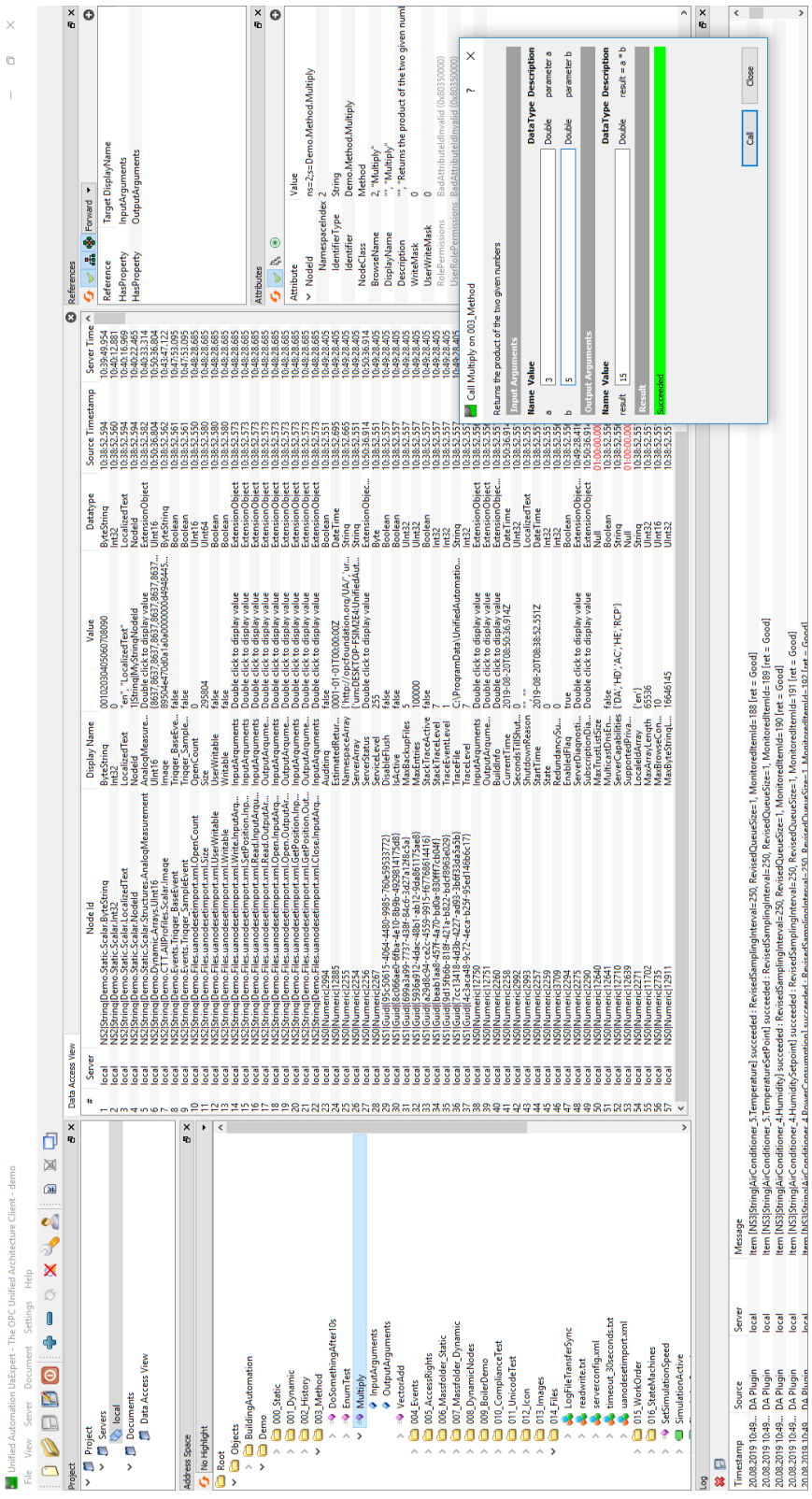


Figure 2.9: Screenshot of UAExpert showing an example information model in the folder hierarchy on the left. The data access view in the middle is a list of nodes and their corresponding current values. Also, a simple method call multiplying two values is depicted on the bottom right corner. The multiply method implies references and attribute which are pointed out on the right.

2.4 Summary

The most important aspects of the state of the art sections are:

- Contemporary ODM use 3D and RGB-D images for pose determination in **two phases**.
- Service interface protocols have to be chosen appropriately for every application.
- Containerization is a convenient way of deploying services.
- OPC UA Vision is based on an information model and a state machine based on which is decided which information is retrievable.
- Recipes are vendor-specific.

Chapter 3

Concept

3.1 Design Variants of an Object Detection Service

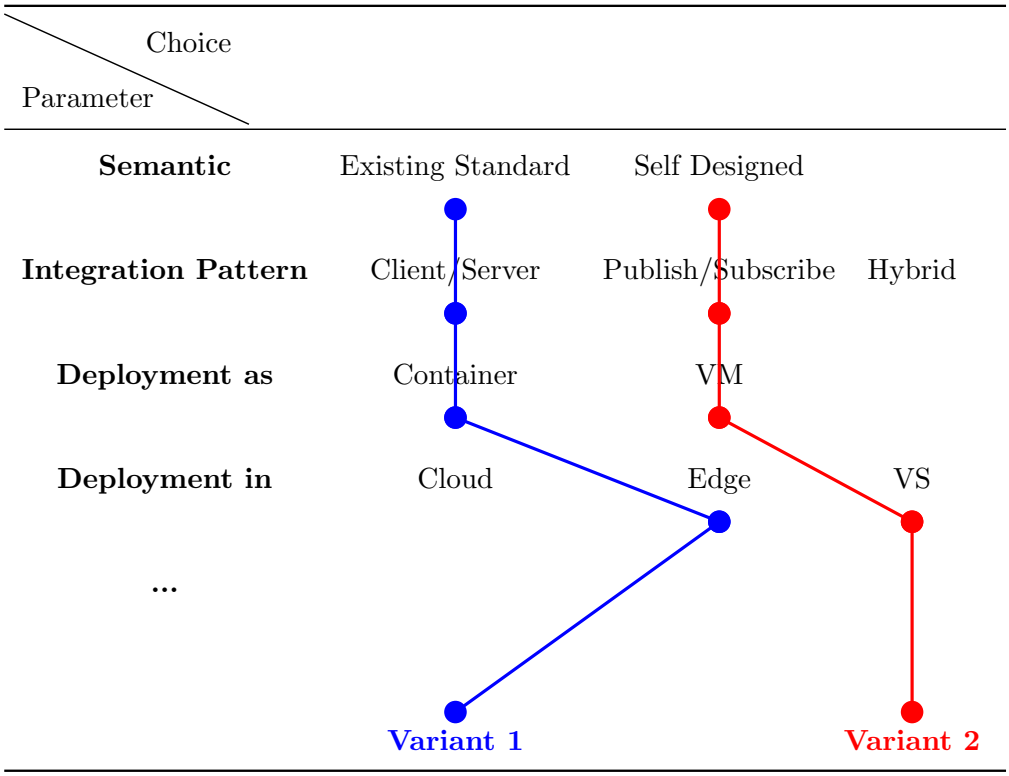
When designing a SOA for ODS, one has to deliberate on a number of tradeoffs. For the purpose of this thesis, ODS should be capable of coping with frequently changing OD algorithms and interfaces. To achieve this, numerous parameters are relevant - semantic, interface, interface adapter, deployment *as* and deployment *in* just to name a few. Several configurations for each of these parameters are possible. See figure 3.1 for a morpholigical box illustrating a number of design variants.

The concept introduces *recipe generator* (RG), a concept for an ODS framework in an industrial environment. It follows the design choices of variant 1.

Semantic defines the functional communication module of the service. It handles the payload of the communication between server and client. The payload depends on definition of methods, resources and possible input/return types. GCV RPC API [22] and to a limited extent[1] OPC UA Vision [1] are two publicly available standards for ODS semantics. In general, resorting to public standards is a good habit due to the already existing user base, detailed documentation, constant development and thorough testing. In contrast to that, one could design the semantic for a service interface on his or her own with the advantage of freedom regarding all design decisions. However, this would mean a constant update of design work

[1] It rather provides a framework for the ODS, but not the semantic (yet).

Table 3.1: Morphological box of the design variants of an ODS. Parameters are marked bold in the first column. Every parameter has possible choices. A combination of choices composes one concept variant, marked as colored lines. Recipe Generator is based on variant 1.



when adapting to new ODM, which is tedious and does not outweigh the advantages of public standards. When comparing OPC UA Vision and GCV RPC API, one can see that the former is meant to be an abstract standard for all kinds of vision systems, the latter is tailored for the GCV service. Through abstraction, the former is more durable than the latter. Also, the OPC Foundation maintains the OPC UA Vision specification and thus focuses on industrial applications and the respective target group. Another way of defining the service would be copying the method signatures and return types of each ODM. This would mean a changing interface towards the client with every new ODM, which is not eligible in the context of the fast-changing world of OD. Instead, the semantic translation between ODM and service methods should be hidden. To yield the aforementioned advantages of the arising industrial standard, this concept relies on OPC UA Vision.

The choice between client/server, publish/subscribe or hybrid integration pattern is driven by factors such as the number of services, real-time requirements, buffering of messages, adoption rate of new technologies and new standards in the industry [38]. In practice, a good integration approach involves starting with a simple client/server pattern and a little number of services and add more sophisticated patterns after critical components have been identified [15]. Due to this reason, RG will also start off with a client/server integration pattern.

Deployment of ODS could be achieved by virtualization in a container, providing a VM or using no encapsulation whatsoever. For a detailed comparison between VM and container, see section 2.2.2. RG utilizes virtualization technologies for its OD and camera services. The main reason for this is the simple exchangeability and enhancement option of containers. Furthermore, they offer very little start-up times compared to VMs. Lastly, native service (i.e., no encapsulation) comes either with high adapting effort for every new service or a lock-in to a specific platform (e.g. Java virtual machine). This could lead in a comparably long time of configuration and getting a new service "running" in the first place. Furthermore, native services imply a huge risk of "dependency hell". For this reason native services are ruled out.

Choosing where to deploy microservices has an impact on security, availability and reliability [15]. In the past year, edge deployment gained popularity. For example, Microsoft introduced Azure Stack which lets the user take advantage of all Azure components locally [39]. This is due to the concern of manufacturers of "offering" their critical data to external companies. Moreover, it tackles the problem of high latency. Deploying software directly on the VS leads to possible high coupling of hardware and software. Also, it is necessary to supply every VS with the necessary computing power, which can mean high costs. RG chooses to deploy its microservices on the edge, i.e., a local factory cloud accepting requests of numerous VSs.

There are many more implementation possibilities such as database integration, orchestration vs. choreography, continuous deployment, service granularity, exception handling, user management and so forth. These aspects are not remarked in the concept phase of this thesis as the main focus is on exchangeability of ODS.

3.2 Recipe Generator

The concept centers around two main components, RG and the OPC UA ViS of the vision system (VS) (see fig. 3.1 for an overview).

Camera providers and detector providers[2] push images to the Docker Hub. The concept does not necessarily rely on Docker, another virtualization technology can be utilized as well. RG trains the detection Docker image, and pushes it back to the Docker Hub. Then, RG generates a recipe. Recipes are scripts which handle camera image acquisition and detection container calling. Camera image acquisition is currently set to be handled between image acquisition containers and cameras directly (or, the container is deployed on the camera itself). This might change shortly. During a phone interview conducted on April 30th, 2019, VDMA member Dr. Reinhard Heister stated that part 2 of the OPC Vision specification would direct the component layer. [40] A component can be a camera that offers an OPC ViS itself. For this task, OPC foundation and EMVA, supervisor of the GenICam interface for cameras [41], will join forces. Hence in a few years' time, it will be possible to have the VS to be interoperable with any camera supporting OPC UA Vision and the images can be acquired via the OPC servers.

The following sections describe the concept in more detail.

3.2.1 Recipe Generation, Transfer and Execution

Generation

The ODS is a trained detector container which is generated with a CAD model and an untrained detector as input. The detector must provide two methods:

```
Train(
    (in)    CADType     CADModel);
    (in)    String      Result);
Test(
    (in)    ImageType   Image
    (out)   PoseType    Pose);
```

[2] For the different roles please look up in annex A.

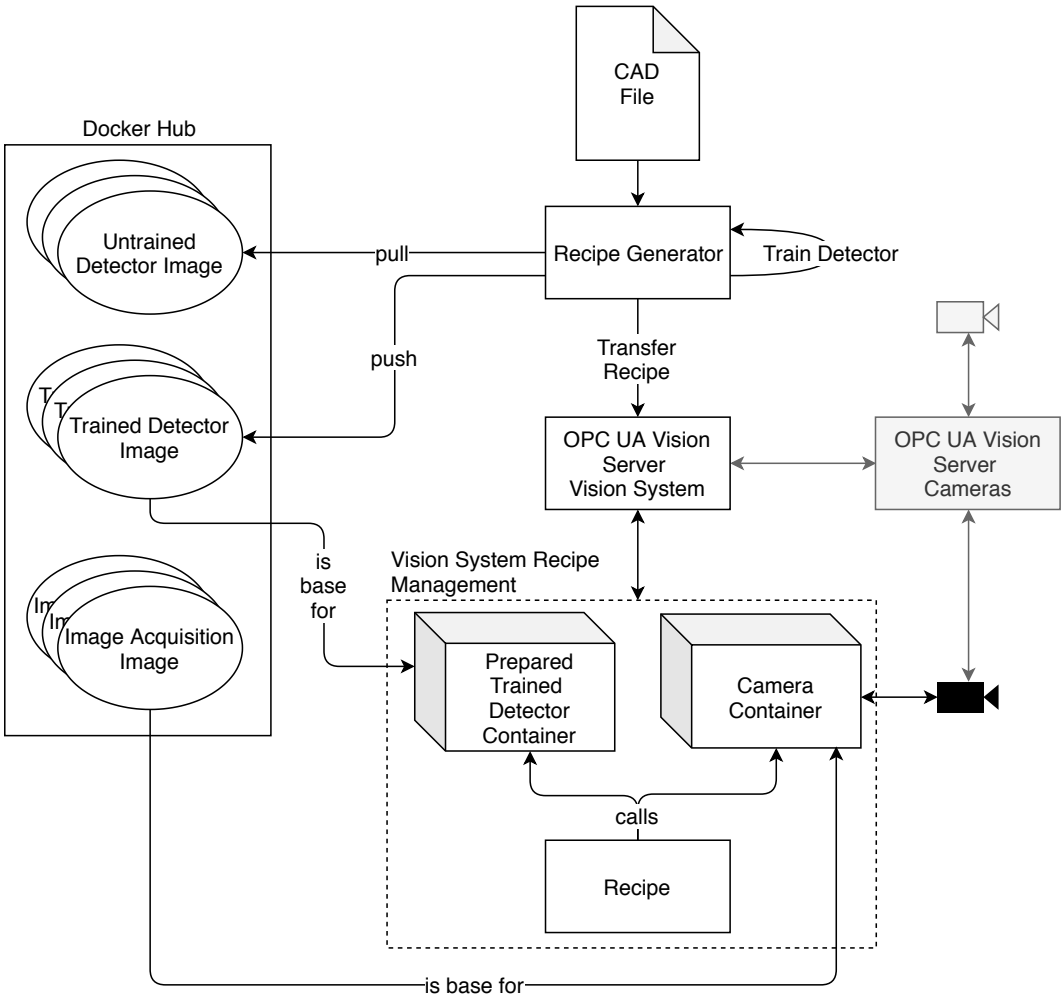


Figure 3.1: Illustration of the industrial ODS concept. The dotted rectangle denotes the scope of the recipe management. The OPC Foundation has not yet released the OPC UA Vision server specification for cameras, hence it is marked grey.

Train is a training routine which generates several views or templates from a CAD model. It should be preconfigured by the detector provider, e.g. learning rates of neural networks have to be preset. After training, the detector container includes generated templates or views and is callable with the Test method. Test requires a camera image for processing the pose of an object aided by the generated templates. Input and output types are proposed in the implementation and annex D but can be versioned and altered if needed. Finally, GenerateRecipe puts image acquisition- and detector container into a sequence. When triggered, an image is grabbed and passed to the Test method of the detector. Test outputs the pose of the object in the image.

Transfer

See figure 3.2 for a sequence diagram. Transfer adheres to the OPC UA Vision specification. The OPC UA client in the diagram is RG in the concept.

- AddRecipe creates a new recipe object and maps an InternalId to the ExternalId. ExternalId is an OPC UA datatype identifying a recipe in the view of the environment, InternalId is an OPC UA data type identifying an instance of the recipe on the VS. InternalId is necessary because recipes can not only be handled via the server but also locally on the VS. For the OPC UA client, only the ExternalId should be callable and internal ambiguities must be handled by the VS. The created recipe object can contain metadata of the recipe and a reference to the recipe itself. As of now, it contains no data.
- GenerateFileForWrite creates a new temporary file object. The GenerateOptions parameter contains the ExternalId of the process. The method returns a FileNodeId and a FileHandle. The temporary file object offers Write and CloseAndCommit methods to transfer binary recipe data. The recipe itself is treated as a file and is referenced in the recipe object.

Execution

See figure 3.3 for a sequence diagram. Execution adheres to the OPC UA Vision specification.

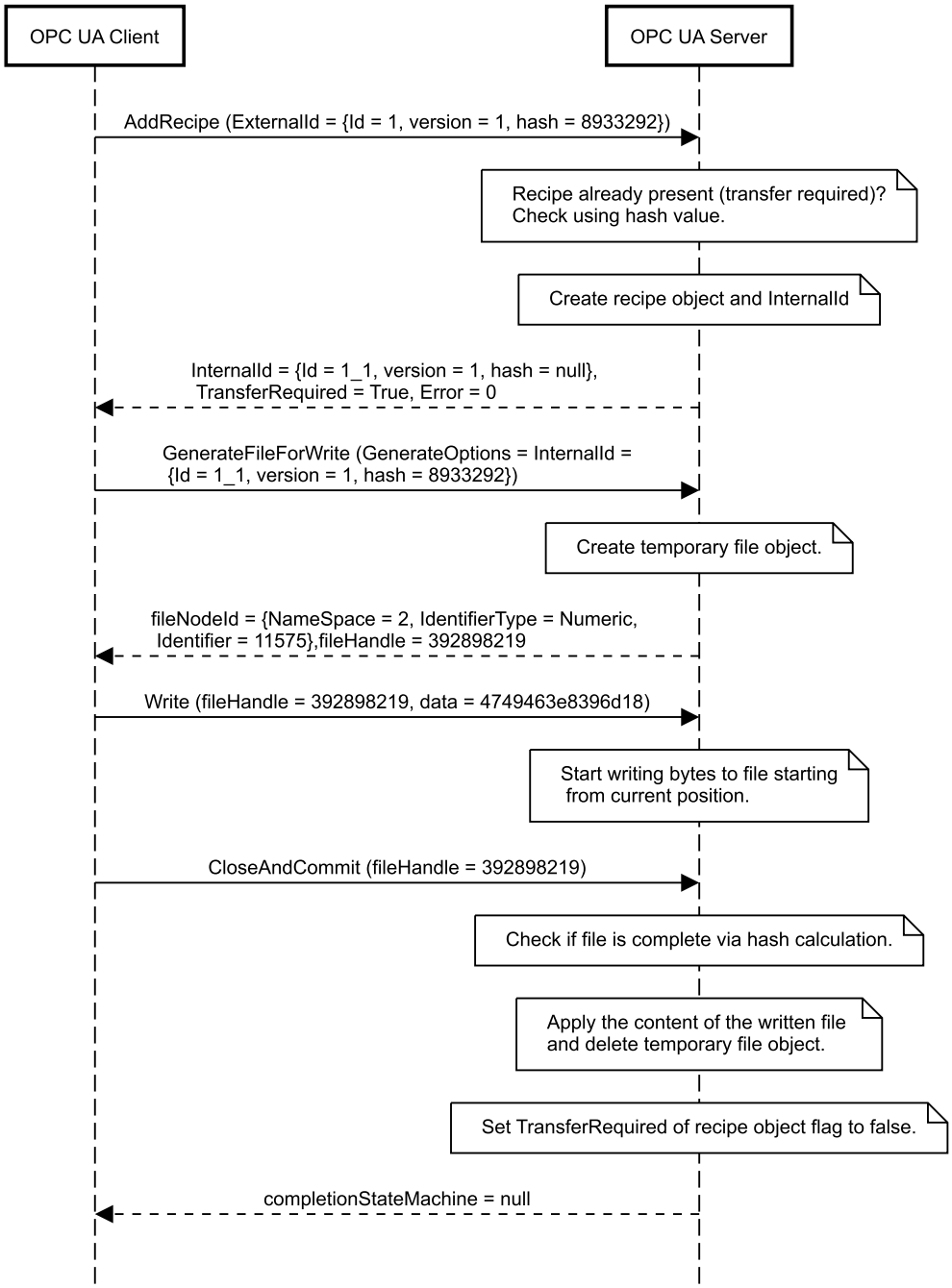


Figure 3.2: Sequence diagram of method based recipe transfer to OPC UA Vision server. Notes on OPC UA server-side denote procedures which are not defined by OPC Vision specification but only described. Not all parameters of method signatures are shown. Example values are provided.

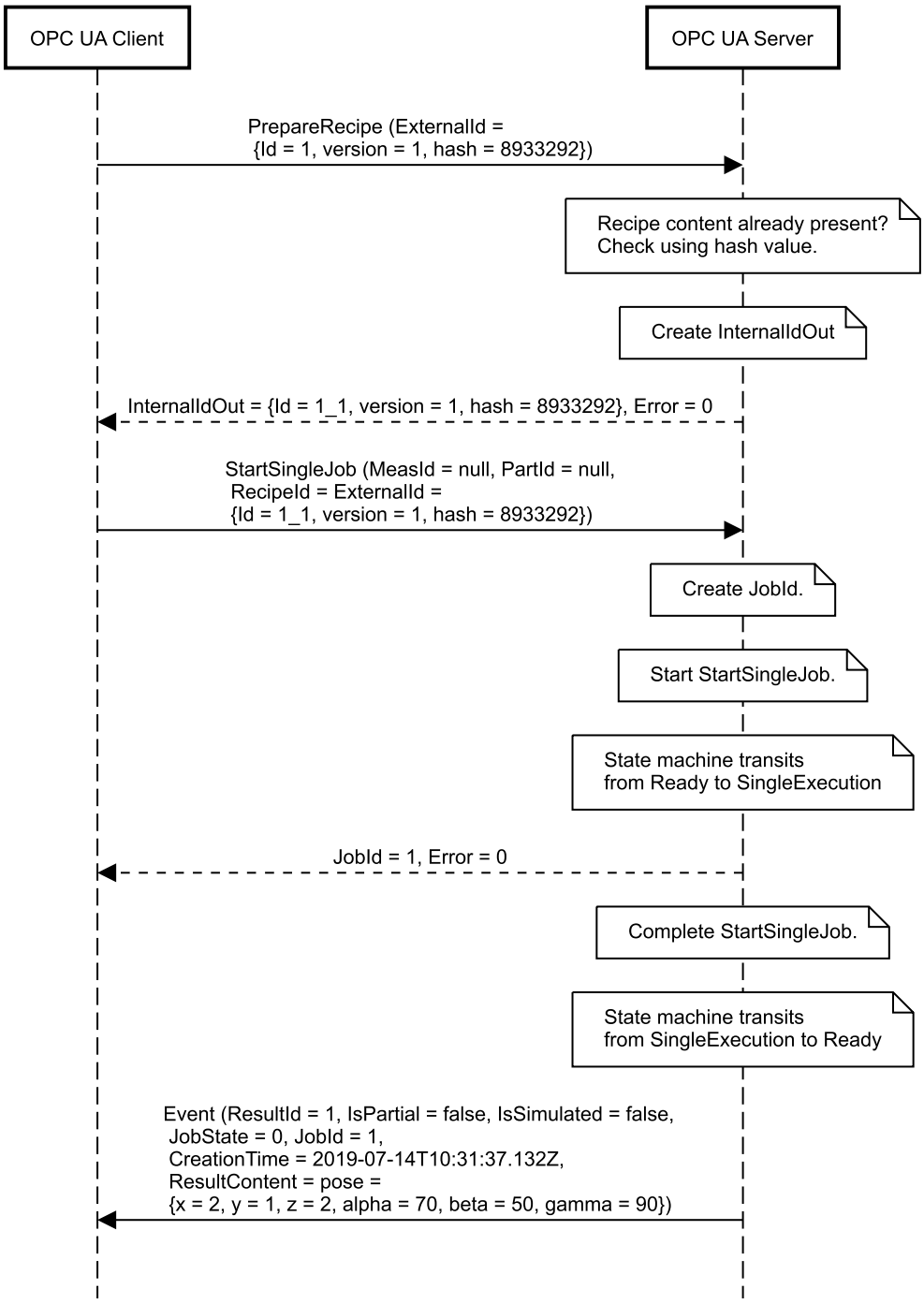


Figure 3.3: Sequence diagram of recipe execution of method based OPC UA Vision server. Notes on OPC UA server-side denote the recipe which is not defined by OPC Vision specification but only described. Not all parameters of method signatures are shown. Example values are provided.

- PrepareRecipe is used to prepare a recipe so that it can be used for starting a job on the vision system. There are several ways the vision system can cope with recipes and their preparation. For example, the vision system can have several recipes prepared for immediate execution or just a single one. PrepareRecipe triggers transition from state Initialized to state Ready of the VS state machine. If the client tries to prepare a detector or image acquisition container, it should return an error.

- StartSingleJob method triggers the transition from state Ready to state SingleExecution. Parameter RecipeId is the ExternalId of the recipe script. The method returns a JobId which has to be used to query any information about the Job later. The Job is finished asynchronously and send back an event which includes the pose of the object.

3.2.2 Deployment

Vision System

In a typical industrial environment, the VS is a production line or a single machine. According to the OPC Vision specification, the application can range from simple light barriers to sophisticated pose detection. The recipes presented here are primarily designed for the latter purpose, however, can be used for the former, too. On the software side, the VS has to cover recipe, configuration and result management. Depending on the size of the system (number of cameras, data storage, CPU cores...) it can be deployed on a single industrial PC or in a larger factory cloud. The detection and image acquisition services in this work are containerized microservices, so the underlying hardware must support the containerization engine. According to an interview with VDMA experts [1], more and more machine vision functionalities shall be transferred to PLCs. This might call for a different approach of deployment for the recipes at a later stage. The OPC client triggering the job/recipe execution is usually a PLC.

Recipe Generator

As for the RG, there are more possibilities. Since multiple vendors can add detectors in a hub, it would make sense to make this available in a (virtual private) cloud. The recipe generation might also require excessive computing power, which can be provided by cloud platforms. Furthermore, mobile networks such as the 5G technology offer bandwidth up to 10.000 MBit/s and latency under 1 ms, so costs on the infrastructure side can be minimized. A possible drawback is the increased security need for data storage and transmission.

The generator might also be deployed as part of the vision system. The advantages of this concept are better data protection possibilities and lower latency. The drawback, in this case, is the lower computing power and tougher access for ODS vendors.

3.2.3 Integration Pattern

The introduced framework relies on a few services with standardized interfaces. Thus, a direct synchronous integration pattern is beneficial due to high performance and simple implementation. In case the framework shall be upgraded to allow for message queueing, user management integration, load balancing and the like, an extension with a service registry might come in handy.

Detector Containers

The detector containers are called during the execution of a recipe. The services might also be called in sequence. For this communication, a standardized interface is of great help. As for the semantic, two methods (Train and Test) are defined. With the Train and Test methods defined, most of the application logic can reside inside the server (detector container); hence changes of ODM do not cause changes for the client (recipe). The connection between recipe, OPC UA Vision server and cameras is implementation-specific.

Recipe Generator

The RG needs an interface for the OPC UA Vision server on the one hand and on the other hand, for the RG client. In case the generator is deployed in an industrial environment, it would be reasonable to configure not only the recipe transfer but also the recipe generation via OPC UA to maintain great interoperability within the shopfloor. For instance, CAD models can be created by cameras and automatically sent to the RG. In case it is deployed in a web environment, an interface rather developed for web technologies is advantageous. For instance, RPC with a RESTful adapter could be used for that purpose.

Image Acquisition

As of now, the image acquisition containers do not have a standardized interface. This is due to the plans of the OPC foundation of fulfilling this task in the near future.

3.2.4 Orchestration of Recipe Management

The recipe, image acquisition- and detector containers combine to recipe management of the OPC UA Vision server. As recipes handle image acquisition and detector container calling, one could say it serves as an orchestrator and is a single point of failure. A possible hazard would be applying too much application logic to the recipe; thus it should stay simple and only run necessary OD steps by calling the containers in sequence with a standardized interface. This maximizes the cohesion of the components. In the case of putting several trained detector containers in sequence, the output of the Test method would have to be abstracted since the second and the following containers would receive a pose of an object as input which is unlikely useful. Abstracting output types would again shift application logic to the recipe which should be minimized. Hence, sequencing OD containers should be scoped in a container which handles the sequencing internally and is externally callable via Train & Test methods.

Chapter 4

Implementation

A proof of concept implementation of the concept introduced in chapter 3 is documented in this chapter[1]. First, it is reasoned which programming language was chosen. Second, the sequence of CAD-to-ODS process is described in detail. Third, the source code and its architecture are explained. Fourth and fifth, it is reasoned which service communication protocol and virtualization technology were chosen respectively.

4.1 Programming Language: Python

For implementing a proof of concept created in 3, a programming language supporting the components Docker, gRPC and OPC UA is highly beneficial. Python not only fulfills all three requirements, but it is also the one of the author's highest expertise.

4.1.1 Support for Docker

There is a Docker SDK available: It lets the user do anything the Docker command does, but from within Python apps – run containers, manage containers, manage Swarms, etc. [42]

[1] The code is available here: `https://gitlab.tubit.tu-berlin.de/nkeuck/masterthesis`

```
import docker

client = docker.from_env()

# You can now run containers:
client.containers.run("ubuntu", "echo hello world")
```

4.1.2 Support for gRPC

Python is one of the supported languages as it is binded to gRPC's C core [43]. Hence it is easy for the RG to import necessary gRPC stubs. The gRPC client can then establish the channel. We can now call the server as shown here with the Train method:

```
# detector/detector_client.py

import grpc
# import stubs containing classes and methods generated from proto
↪  file
import detector_pb2
import detector_pb2_grpc


class DetectorClient:
    def __init__(self):
        # establish a channel based on a TCP http/2 connection
        channel = grpc.insecure_channel("localhost:8000")
        # stub can be used to invoke gRPC methods
        self.stub = detector_pb2_grpc.DetectorStub(channel)

    def Train(self, cadFile, **kwargs):
        config = kwargs.setdefault("config", io.StringIO("dummy
        ↪  config"))
        # generate an iterable object of the cadFile and config
```

```
        # types have to be the same as message types in the proto
        ↪ file
        requestiterator = self.get_file_chunks(cadFile, config)
        # call Train method and print result
        response = self.stub.Train(requestiterator)
        print("Detector client received Train result: {0} \nAnd
        ↪ image: {1}".format(response.Result, response.Image))

# ...
```

4.1.3 Support for OPC UA

There is an open-source Python SDK for OPC UA [44]. Hence, it is easy for the RG to import an OPC UA client, whilst the OPC UA client can easily connect to a server:

```
# opcua/opcua_client.py

from opcua import Client
from opcua import ua

run():
    # populate adress space, invoke methods...

if __name__ == "__main__":
    # ...
    client = Client("opc.tcp://localhost:4840/freeopcua/server/")
    run()
```

In the given example, the main method is illustrated, this is of course only executed if the script is run directly and not via an import.

4.2 From CAD File to Pose of an Object

The subsections are named like the methods one would call during usage. They illustrate the process of a CAD file or camera image to determining the pose of an object inside that file or image through sequence diagrams. The requirement is having a ready to use detector and camera Docker image loaded in Docker hub and a CAD file.

In the diagrams of this section, dashed arrows denote gRPC connections, and dotted arrows denote OPC UA connections. Signatures are hidden in the diagrams for a more concise overview. OPC UA and gRPC methods are written in CamelCase, thus the different naming conventions in the implementation.

4.2.1 GetConfig

This optional method can be used to gather the current configuration of the detector or camera docker image. See figure 4.1.

1. `GetConfig` is called by the RG client which could be a machine operator, a PLC or a production planner.

   GetConfig(
   (out) String Config);

2. To run this method, it is necessary to instantiate an instance of the `DockerApi` class which connects to a Docker registry provided by environment variables. You need to provide an `imageName` when instantiating. Running the `run` method does not require any input, a `port` can be stated optionally (default 8000 for detectors, 8011 for cameras):

   run_container(
   (in) Int port
   (out) ContainerObject container);

   `ContainerObject` will be returned by the nested method call explained in point 4.

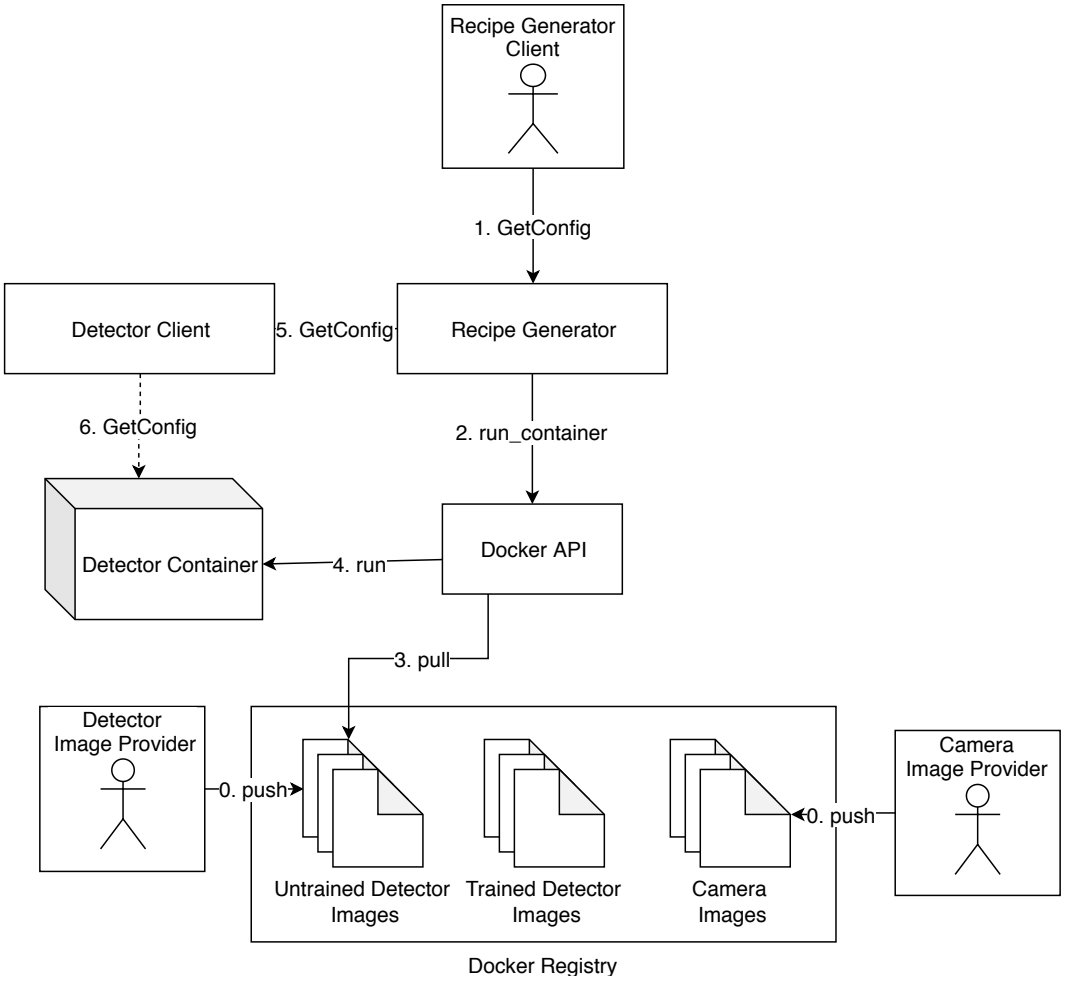


Figure 4.1: Sequence Diagram - GetConfig

3. `Pull` is invoked by the run method in point 4 and needs no call.

4. See [42] for a detailed reference of `run`:

run(			
	(in)	String	imageName
	(in)	boolean	detach
	(in)	boolean	autoremove
	(in)	Dict	ports
	(in)	boolean	stderr
	(in)	boolean	stdout
	(out)	ContainerObject	container);

5. `GetConfig` is a member of `DetectorClient` which is initialized with a gRPC channel to the detector and interfacing stub. GetConfig can be called with no input. For the signature see 1.

6. `GetConfig` gathers the config via gRPC. For the signature see point 1.

4.2.2 Train

See figure 4.2.

1. `Train` uses a CAD file and an optional configuration file as input to train the detector for later usage. It returns the Train result and optional camera image files for result representation.

```
Train(
    (in)    file      cad
    (in)    file      conf
    (out)   file      image
    (out)   String    result);
```

2. See point 2 in 4.2.1.

3. See point 3 in 4.2.1.

4. See point 4 in 4.2.1.

5. `Train` is a member of a `DetectorClient` which is initialized with a gRPC channel to the detector and interfacing stub. See point 1 for the signature.

6. `Train` gathers the config via gRPC. See point 1 for the signature.

7. `commit` creates an image of the running detector container instance and pushes it to a repository. It is a method part of the Docker Python framework described in [42], see a detailed description of the method there. Here, the method pushes the image to the repository defined by environment variables.

```
Train(
    (in)    ContainerObject    containerinstance
    (out)   String             result);
```

8. `create_image` is part of `commit`.

9. `push` is part of `commit`.

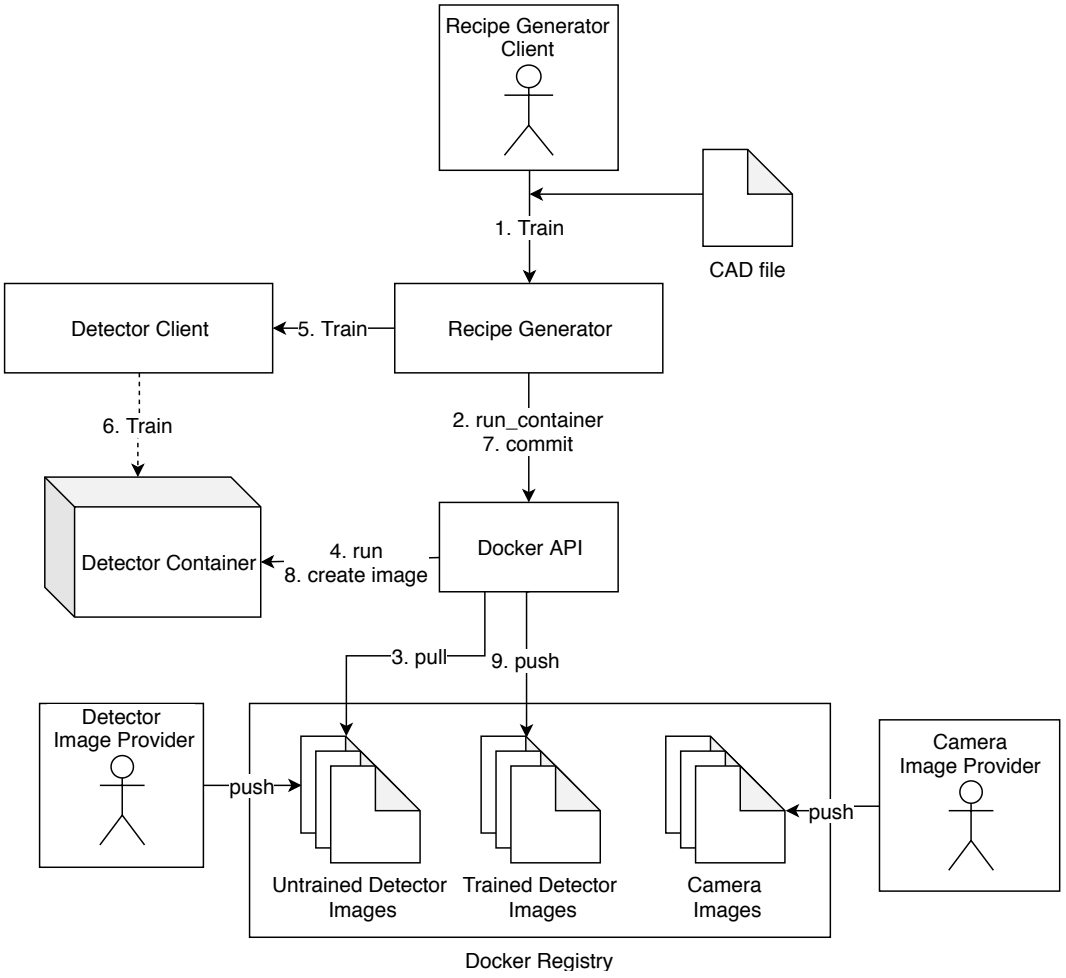


Figure 4.2: Sequence Diagram - Train

4.2.3 GenRecipe

See figure 4.3. `GenRecipe` takes a docker image pair (`i`) and saves it in the recipe storage. The pair consists of one camera and one detector docker image. As of now, the method saves the pair as a file in a folder. In the future, this could be done with a database. The filename is a generated UUID later used as `ExternalId`.

1. `GenRecipe` initializes an instance of `Recipe` with an `ExternalId` and calls its `add` method.

```
GenRecipe(
        (in)        String        i
        (out)       String        ExternalId);
```

2. `add` takes the image pair (`imageList`) as input and `write`s it into recipe/recipes folder with `ExternalId` as the filename.

```
add(
        (in)        String        imageList);
```

3. `write` writes `imageList` as comma-separated String into a file. The method is part of `add`.

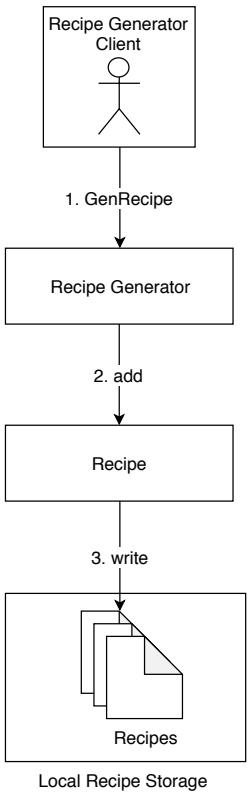


Figure 4.3: Sequence Diagram - GenRecipe

4.2.4 Transfer

`Transfer` fetches the recipe and transmits it over to the OPC UA Vision server recipe management. The server has to be running for a successful method call. In the demo implementation, the recipe storage of the RG and the OPC UA Vision server are the same (see 4.4). A recipe is a pair of docker image names, the list being decorated with an `ExternalId`. The OPC UA Vision client can be a PLC or a human machine operator. See figure 4.4.

1. `Transfer` instantiates an instance of `Recipe` with `extid` and transfers the gathered image list via an OPC UA Client to the server.

   Transfer(
   (in) String extid
   (out) String ExternalId);

2. See point 1.

3. get_recipe(
   (out) Array imageList);

4. add_recipe(
   (in) String ExternalId
   (in) Array imageList
   (out) String Result);

5. OPC UA Client opens a TCP connection to OPC UA Server and then calls `AddRecipe`. See point 4 for the signature.

6. See point 2 in subsection 4.2.3.

7. See point 3 in subsection 4.2.3.

4.2.5 Prepare

`Prepare` runs the two docker images defined by `ExternalId` returns. See figure 4.5.

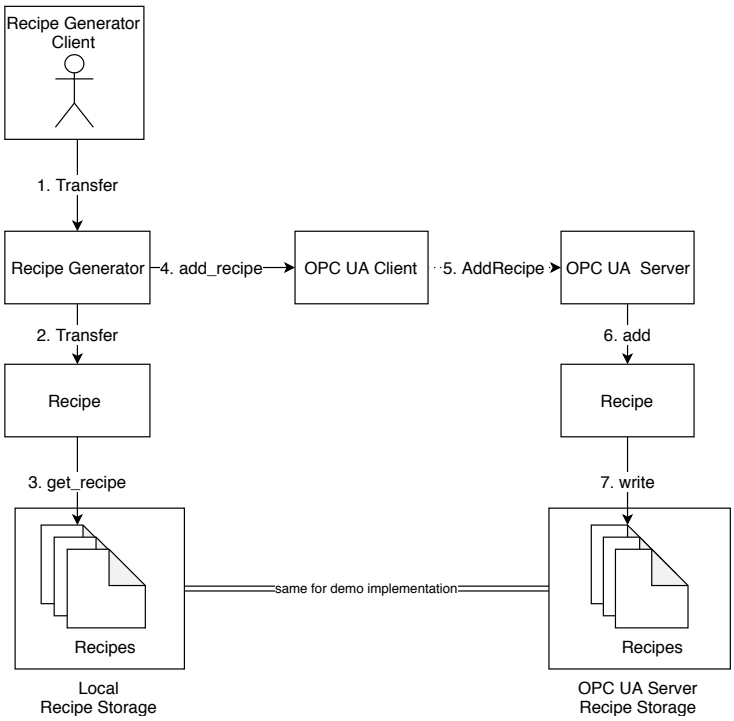


Figure 4.4: Sequence Diagram - Transfer

1. PrepareRecipe(

(in)	String	ExternalId
(out)	String	Result);

2. OPC UA Server imports the same `Recipe` class as the RG. It instantiates an instance of the class and calls the prepare method.

3. From here onward the process is analogous to the methods described before.

4.2.6 Test

See figure 4.6.

1. OPC UA Vision server creates a `JobId` unique for this job and returns it along with the result.

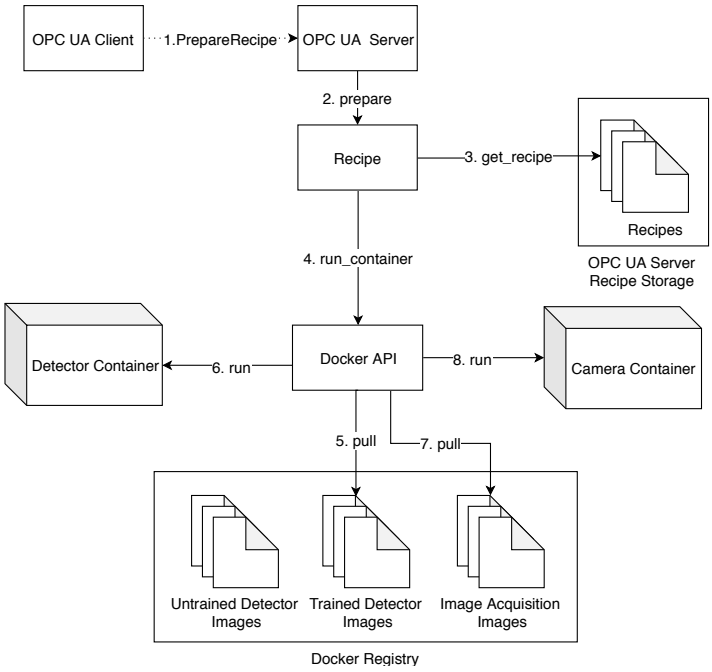


Figure 4.5: Sequence Diagram - Prepare

StartSingleJob(

(in)	String	RecipeId
(out)	Dict	Pose
(out)	String	JobId);

2. GrabImage(

(out)	bytes	image);

3. The camera client calls the camera container located on a camera or camera image storage to fetch and return a camera image. See point 2 for the signature.

4. `Test` uses the grabbed image of point 2 to call detector client's `Test` method. `Test` optionally returns images to illustrate its result.

Test(

(in)	bytes	image
(out)	Dict	Pose
(out)	bytes	image
(out)	String	JobId);

5. See point 4.

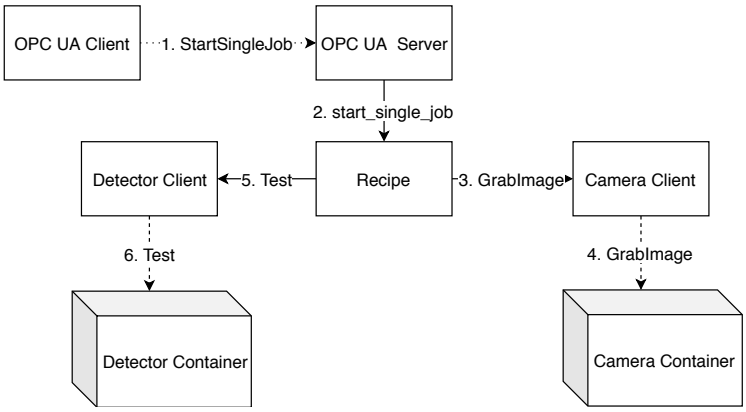


Figure 4.6: Sequence Diagram - Test

4.3 Software Architecture of Recipe Generator

The architecture of RG divides into the same components as presented in the concept in figure 3.1. So does the folder structure:

- camera. Represents the blueprint of a gRPC interface of a camera with the `GrabImage` method.
  - images. Storage for grabbed camera images.
- detector. Represents the blueprint of a gRPC interface of a detector with the `Train` and `Test` method.
- dockerapi. Handles Docker registry administration, container running and more Docker features.
- opc_ua. A Python OPC UA server.
  - recipes. Recipe management storage of the OPC UA server.
- recipe. A template for recipies calling the standardized methods of camera and detector in sequence.
  - recipes. Recipe storage of the RG.

When using RG, only recipe_generator.py and opc_ua_client.py have to be called from a command-line interface. A help is provided, callable with the help flag:

```
py path_to_file/recipe_generator.py --help
```

The source code is provided with comments and docstrings where necessary. All Python modules have no GUI, instead, they are command line based. However, the use of a graphical OPC UA client such as UAExpert is possible.

See figures 4.7 and 4.8 for an illustration of the used classes and packages in the demo implementation. The UML diagrams were generated automatically with Pyreverse by scanning the source code [45].

The classes ending with Servicer and Stub are generated by gRPC. gRPC server classes need to inherit the Servicer class (e.g., Detector inherits DetectorServicer). In case the proto files are enhanced or new files are created, new Servicer and Stubs can be created with a build command similar to:

```
py -m grpc_tools.protoc  -I. --python_out =. //
    --grpc_python_out =. .\ camera.proto
```

Note that the command must be executed from the directory of the respective proto file.

Recipe class is responsible for recipe administration and calling. For example, `prepare()` runs the detector and camera container of the recipe instance.

The package diagram gives an overview of the dependency complexity of the different components. Through the cohesion of the components, methods can be reused effectively and in a way that is following the RG concept. An example is the Docker API (dockerapi.dockerapi) that is used by the RG (recipe.recipe_generator) and the recipe (recipe.recipe).

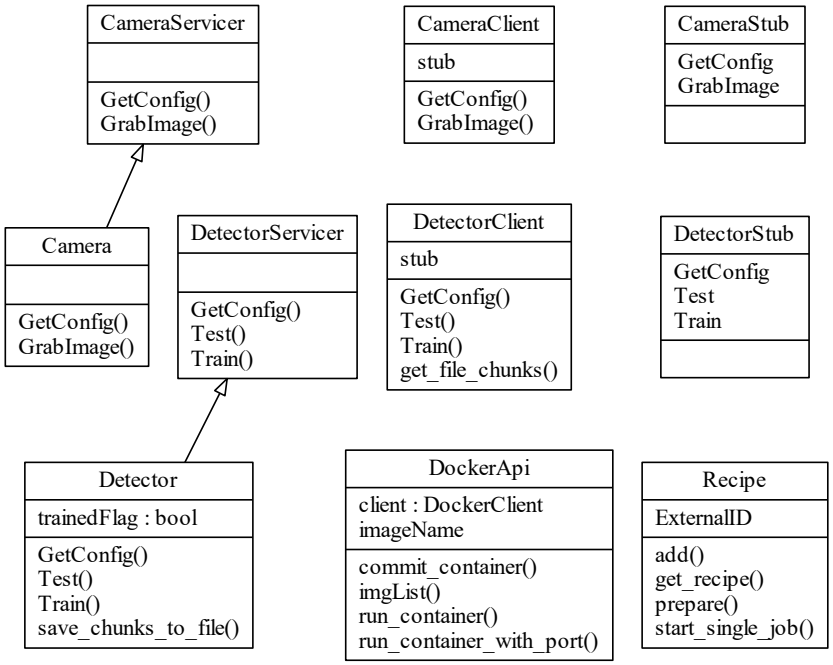


Figure 4.7: Class diagram. Arrows denote inheritance.

4.4 Virtualization Technology: Docker

4.4.1 Pro

Realizing a SOA calls for means of decoupling the components and efficient tooling. Docker is the technology which was used here due to the following reasons:

Dependencies of Components are handled smoothly

Every docker image can pull the packages and system variables it needs as specified in the dockerfile. For example, one detector may depend on openCV 2.7.1 while another detector depends on 1.8. Both docker images that are built using their respective dockerfile are independent of each other. If no virtualization technology was used here, a dependency handling for the whole recipe management would be

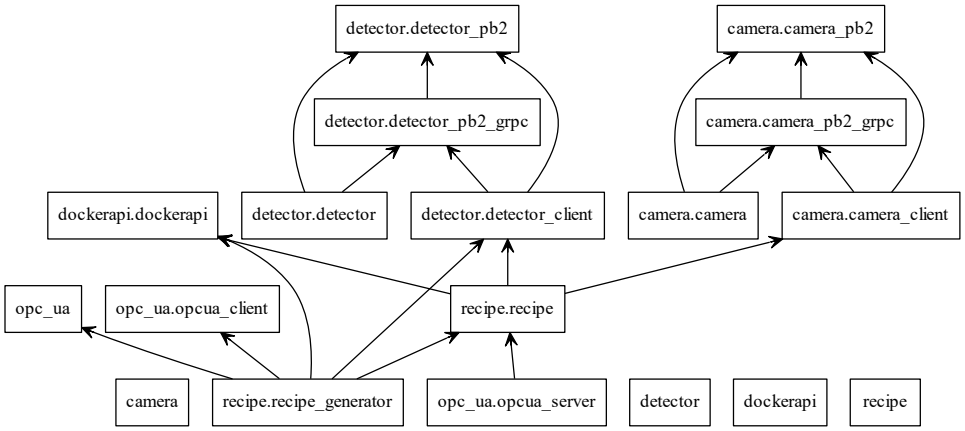


Figure 4.8: Package diagram. Arrows denote imports.

necessary. In the case of two openCV versions, just a slight modification of the framework may be necessary. A more drastic example would be detectors that rely on different .NET frameworks which might not be able to coexist on a system.

Platform Independency

Platform in this context refers to the operating system, e.g. Windows or Linux. The independency is twofold. Firstly, the docker engine runs on Linux (CentOS, Debian, Fedora, Oracle Linux, RHEL, SUSE and Ubuntu) and Windows Server. In an industrial context, both platforms are present and should be supported for maximum flexibility.

Orchestration

Containers can be scaled if more resources are needed, monitoring and load balancing are possible and the network over which the containers communicate can be configured. See e.g. Mandy Waite's contribution to DevFest Ukraine in 2015 for more information. [46]

Docker Hub

Docker offers (semi)-public or private repositories. They can be used by detector or camera providers to push their docker images and for the recipe management to pull them. As base image, there are preconfigured environments available. For this implementation, a fully functioning Python environment was used as base Docker image. To prevent know-how-leaking of ODMs, the accessibility to the images on the hub can be restricted. In this implementation, a local repository was used with full accessibility to the Docker images.

Calculations on Graphical Processing Unit

Some detectors need excessive calculation power. If applicable, the graphical processing unit of the bare metal server the docker engine is running on can be added. It should be kept in mind that with this technique the detector docker image is dependent not only on the docker engine but also on the bare metal hardware. Thus, this option should be used with caution and only if needed.

4.4.2 Contra

Lock-in

The reason against using Docker is making all involved parties introduced in appendix A use docker. Camera and detector docker image providers need to add a dockerfile for building the image and pushing it to the repository. On the recipe management-side a docker engine compatible infrastructure has to be provided.

4.5 Detector and Camera Communication: gRPC

To implement the Train, Test, GetConfig and GrabImage methods defined in 3.2.1 gRPC was used. In the current implementation, the TCP ports on which the detector and camera server listen is static. Detectors listen on port 8000, cameras on port 8011. The approach of gRPC service interfaces comes with certain pros and cons:

4.5.1 Pro

Simple Service Definition

In proto files, the service RPCs and input/output types are defined. With protoc, the protocol buffer compiler, it is possible to generate place holders for clients and servers, called stubs. These place holders are the base for every camera or detector docker image provider. For a complete overview of the created proto files see D.

Multilingual

Idiomatic stubs can be generated in 12 languages [43]. This is a great advantage for every provider since no wrapping or complete rewriting of code in a possibly unknown language has to be done.

Backward Compatibility

According to Sam Newman, it is crucial to be able to deploy new servers without having to deploy clients at the time or otherwise dependently ([15], page 79). This is easily done with gRPC and proto3 syntax - as long as no fields are deleted or the numbers change, older clients will still be able to work with new servers.

REST Gateway

gRPC's ecosystem offers a protobuf/JSON gateway as described in detail in section 2.2.1. In the current implementation, this is not done. Note that as of today, it is not supported to send files as streams via the gateway. As a workaround, files can be converted into base64 encoded strings.

Streaming

gRPC supports HTTP/2. Especially for large files this is an advantage because multiplexed, bi-directional streams are supported. This allows for high-performance file

transfer. In this implementation, a byte stream was used for the Train method of the detector. See D for the proto file and 4.1.2 for an example python implementation.

4.5.2 Contra

Lock-in

As well as for Docker as virtualization technology gRPC might mean extra work for detector and camera providers. Although stubs can be provided, developers still need to adhere to the service definition in the proto file.

Networks will fail

According to Sam Newman, networks can never be trusted ([15], page 76). So, appropriate fail mechanisms have to be provided. For example, a buffering of messages is possible in case of a corrupt gRPC channel. This is not implemented in the demo framework yet.

4.6 Object Detection Methods used

As for the demo framework implementation, only dummy detection methods were used. There is no ODM behind the Train and Test methods, just a simple snippet receiving, saving and returning data. For the config file, an arbitrary string with no further use was sent between services; for the camera image and CAD files, a JPG demo camera image was sent. The relevant sections in source code are marked with comments so the framework architect and Docker image providers know where to include their code snippets.

4.7 Differences to OPC UA Vision Specification

There are major simplifications made of the OPC UA Vision specification. This is mostly due to the fact that there was no **XML nodeset** released by the time of implementation. The nodeset would have allowed to import the entire information

model to an OPC UA server. Instead I had to create the methods and necessary data types manually.

Recipe transfer has been simplified since the OPC UA Vision server and the RG share their recipe storage and Docker registry, it is sufficient to transfer the ExternalId with no additional content required. This might not be the case in a productive environment, where the topology does not allow sharing the same storage and registry. If they share at least the Docker registry, then it is possible to reprogram the ExternalId to a string of concatenated Docker image names, e.g. detector:latest-camera:latest. If they share none of the two, then additional image content transfer via OPC UA has to be done or the OPC UA Vision server Docker registry needs to be managed externally.

According to the OPC UA Vision specification, StartSingleJob is an asynchronous method which triggers recipe execution and the state machine to transit from Ready to SingleExecution (see figures 2.6 for a sequence diagram of the automatic mode of the state machine and 3.3 for a sequence diagram of recipe execution). The entire **state machine** of the OPC UA Vision server was neglected due to the time restrictions of the thesis. As a workaround, all methods were implemented synchronously. In consequence, the event containing the detection result is not sent but instead the pose is directly returned by StartSingleJob.

Chapter 5

Evaluation

This chapter structures in two parts. First, risks and tradeoffs of the concept and implementation are illuminated with an established SOA evaluation method. Second, RG is discussed and possible improvements are pointed out.

5.1 The ATAM Method

> It is better to be vaguely right than exactly wrong.
>
> Carveth Read (1848 – 1931)
> philosopher and logician

In this chapter, an excerpt of the architecture tradeoff analysis method (ATAM, [47]) as conducted in [38] shall be used to evaluate the current implementation and underlying concept. ATAM is a method aimed at illuminating risks in the architecture through the identification of attribute trends, rather than at precise characterizations of measurable quality attribute values [48]. ATAM focuses on the analysis of quality attributes. Quality attributes, also known as **nonfunctional requirements**, include usability, performance, scalability, reliability, security and modifiability. Subsection 5.1.1 provides some generic examples. Subsections 5.1.2 and 5.1.3 introduce the necessary ATAM steps and the goal of finding risks and tradeoffs. For the introduced framework it is an applicable method as it stays high level. Once it is

implemented in an industrial environment, a more quantifiable method is necessary.

ATAM has been introduced by Kazman in 1998 at Carnegie Mellon University in Pennsylvania [47]. It was used on at least 18 architectures in the 2000s [49]. In 2013, Zalewski introduced an updated ATAM method which can be applied at an even earlier stage of architecture evaluation [50].

5.1.1 Quality Attributes

The following example quality attributes are decorated with generic examples which are directly quoted from Appendix A in *Evaluating a Service-Oriented Architecture* [38].

Performance

- A sporadic request for service 'X' is received by the server during normal operation. The system processes the request in less than 'Y' seconds.
- The service provider can process up to 'X' simultaneous requests during normal operation, keeping the response time on the server less than 'Y' seconds.
- The roundtrip time for a request from a service user in the local network to service 'X' during normal operation is less than 'Y' seconds.

Availability

- An improperly formatted message is received by a system during normal operation. The system records the message and continues to operate normally without any downtime.
- An unusually high number of suspect service requests are detected (denial-of-service attack), and the system is overloaded. The system logs the suspect requests, notifies the system administrators, and continues to operate normally.
- Unscheduled server maintenance is required on server 'X.' The system remains operational in degraded mode for the duration of the maintenance.

- A service request is processed according to its specification for at least 99.99 % of all requests.
- A new service is deployed without impacting the operations of the system.
- A third-party service provider is unavailable; modules that use that service respond appropriately regarding the unavailability of the external service; and the system continues to operate without failures.

Security

- A third-party service with malicious code is used by the system. The third-party service is unable to access data or interfere with the operation of the system. The system notifies the system administrators.
- An attack is launched attempting to access confidential customer data. The attacker is not able to break the encryption used in all the hops of the communication and where the data is persisted. The system logs the event and notifies the system administrators.
- A request needs to be sent to a third-party service provider, but the provider's identity can not be validated. The system does not make the service request and logs all relevant information. The third party is notified along with the system administrator.

Testability

- An integration tester performs integration tests on a new version of a service that provides an interface for observing output. 90 % path coverage is achieved within one person-week.

Interoperability

- A new business partner that uses platform 'X' is able to implement a service user module that works with our available services in platform 'Y' in two person-days.

- A transaction of a legacy system running on platform 'X' is made available as a web service to an enterprise application that is being developed for platform 'Y' using the web services technology. The wrapping of the legacy operation as a service with proper security verification, transaction management, and exception handling is done in 10 person-days.

Modifiability

- A service provider changes the service implementation, but the syntax and the semantics of the interface do not change. This change does not affect the service users.
- A service provider changes the interface syntax of a service that is publicly available. The old version of the service is maintained for 12 months, and existing service users are not affected within that period.
- A service user is looking for a service. A suitable service is found that contains no more than 'X' percentage of unneeded operations, so the probability of the service provider changing is reduced.

Reliability

- A sudden failure occurs in the runtime environment of a service provider. After recovery, all transactions are completed or rolled back as appropriate, so the system maintains uncorrupted, persistent data.
- A service becomes unavailable during normal operation. The system detects and restores the service within two minutes.

5.1.2 ATAM Steps

ATAM consists of the following steps (directly quoted from [38]):

1. Present the ATAM: The evaluation team presents a quick overview of the ATAM steps, the techniques used, and the outputs from the process.

2. Present the business drivers: The system manager briefly presents the business drivers and context for the architecture.

3. Present the architecture: The architect presents an overview of the architecture.

4. Identify architectural approaches: The evaluation team and the architect itemize the architectural approaches discovered in the previous step.

5. Generate the quality attribute utility tree: A small group of technically oriented stakeholders identifies, prioritizes, and refines the most important quality attribute goals in a utility tree format.

6. Analyze the architectural approaches: The evaluation team probes the architectural approaches in light of the quality attributes to identify risks, non-risks, and tradeoffs.

7. Brainstorm and prioritize scenarios: A larger and more diverse group of stakeholders creates scenarios that represent their various interests. Then the group votes to prioritize the scenarios based on their relative importance.

8. Analyze architectural approaches: The evaluation team continues to identify risks and tradeoffs while noting the impact of each scenario on the architectural approaches.

9. Present results: The evaluation team recapitulates the ATAM steps, outputs, and recommendations.

These steps are typically carried out in two phases. Phase 1 is architect-centric and concentrates on eliciting and analyzing architectural information. This phase includes a small group of technically oriented stakeholders concentrating on Steps 1 to 6. Phase 2 is stakeholder-centric, elicits points of view from a more diverse group of stakeholders, and verifies the results of the first phase. This phase involves a larger group of stakeholders, builds on the work of the first phase, and focuses on Steps 7 through 9. [51]

5.1.3 ATAM Goals

It is desired to find risks and tradeoffs:

- risks: architectural decisions that might create future problems for some quality attribute. A sample risk: The current version of the Database Management System is no longer supported by the vendor; therefore, no patches for security vulnerabilities will be created.

- tradeoffs: architectural decisions that have an effect on more than one quality attribute. For example, the decision to introduce concurrency improves latency but increases the cost of change for the affected modules.

5.2 Architecture Evaluation of RG with ATAM Method

In this evaluation section, steps 7 and 8 will be covered as described in subsection 5.1.2. Stakeholders each contribute with possible scenarios to address their goals. For example, RG operators want a framework that is easy to modify. The administrators of a manufacturing company want the framework to be safe and secure.

The scenario list comprising seven entries is claimed to be incomplete and just covers some important aspects that were considered during service design. Also, in practice, ATAM can be iterated which is dropped here. The detailed description of each scenario can be found in table C.1 with the respective analysis in table C.2. This section provides a synthesis of the illuminated risks and tradeoffs.

The main quality attribute considered is modifiability. This goal is reached satisfactory. Possible risks are lock-in effects of the used technology, tradeoffs are on a hardware level: the combination of gRPC + Docker might end up cumbersome without a fitting middleware, thus affecting usability and performance. Also, the more messages that are created for upward compatibility, the more overhead will be sent over the channels. The second most important quality attribute for RG is interoperability. RG is designed for the ease of use with Docker containers, but (the implementation) does not rely on it. Not utilizing Docker leads to a higher implementation effort due to necessary rewrites of the framework. Also, the proposed Train/Test abstraction of the ODM has to be accepted by ODM providers, machine operators and the framework architect. The third most important quality attribute is reliability. Since RG currently relies on strong decoupling of concise services, the foundation for it is laid out. On the downside of the client/server integration pattern

is the lack of quality-of-service delivery of messages.

5.3 Discussion and Possible Improvements

The discussion of RG is structured in the three sub-tasks of this thesis - generating ODS automatically, handling ODS with varying interfaces and handling ODS with varying ODMs. Although the goals intertwine, they are focused on as separately as possible. Every subsection includes means of possible improvements.

5.3.1 Generating Object Detection Services Automatically

This goal has been reached partly. RG enables the user in composing the necessary services by loading them from the service registry, training the ODS, pushing them back to the registry and finally transferring and executing them. What RG does **not** do is creating untrained ODM in the first place. Nor does it contain a smart mechanism to choose which ODM is best for a certain OD task. These are aspects which require a high exptertise.

The idea of a future research direction is a device and/or service catalogue. An example for this is the open-source project Eclipse Vorto [52]. Screenshots are depicted in figures 5.1 and 5.2. The catalogue helps users describe, share and integrate their services. As the project is community-based, every user can upload content such as cameras, ODMs or data types.

Figure 5.1: Screenshot of Eclipse Vorto's device catalogue. Every device has a self-explaining interface semantic. The catalogue is community based, i.e. everyone can add his or her device [52].

Choosing an applicable ODM for a specific task can then be handled by the community through a rating system. If the user base is big enough, it will be possible to evaluate best practices for camera/ODM/machine/part type combinations.

Moreover, ODS can provide (hyperlinks to) tutorials and offer workshops alongside the ODS to better scale their know-how. The ODM source code does not have to be revealed with the advantage of a lower incentive to add content to the catalogue.

The challenge for this catalogue is to reach a critical mass (i.e., active users) to prosper.

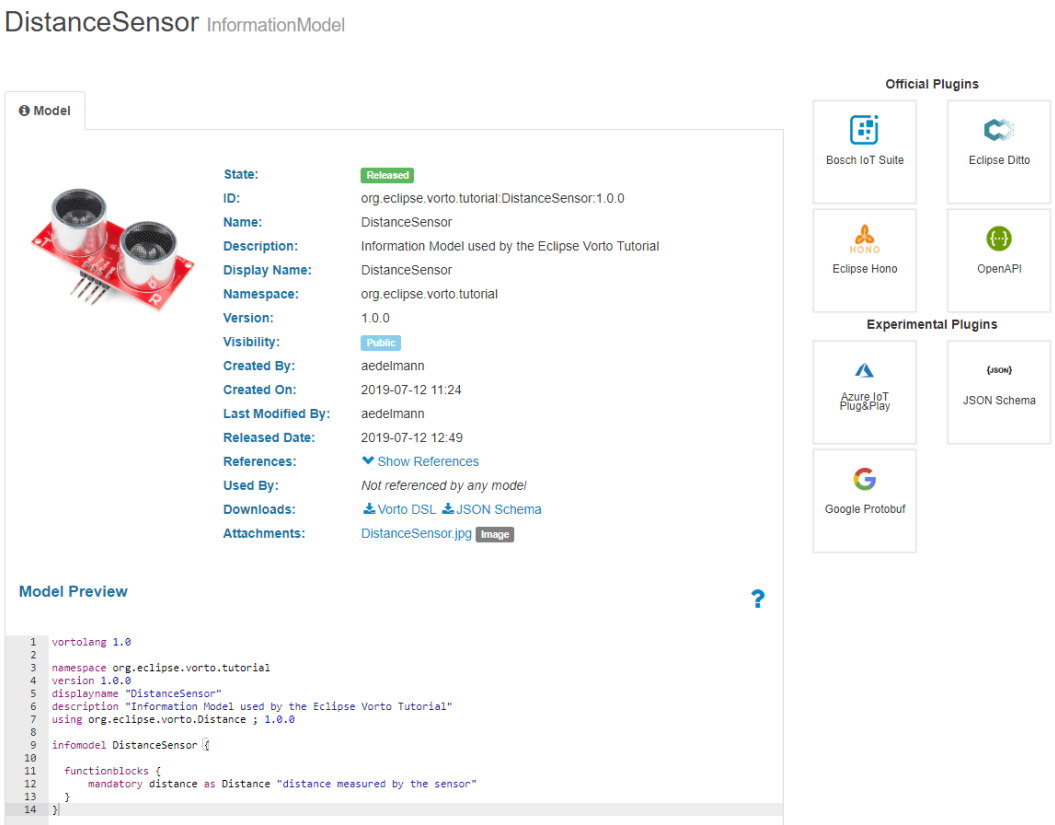


Figure 5.2: Screenshot of a detail view of a distance sensor in Eclipse Vorto's device catalogue. Every device has a self-explaining interface semantic. Data types can be shared between devices [52]. Plugins for mapping to other interface definition languages such as Google Protobuf are available.

5.3.2 Handling Object Detection Services with Varying Interfaces

RG uses gRPC for service communication. There is the possibility to offer a REST gateway [23] as well. This covers a broad range of (web-) services. Still, the framework highly depends on the ODS provider and his ODM configurability to interact with RG. Some ODS providers do not offer a gRPC interface. In this case, the REST gateway would also not find a remedy since it only maps REST to gRPC, not vice versa. Hence, for now RG mainly has to work with open-source ODM.

Also, the services are dockerized. This is a convenient way of deployment for ODS that the RG user owns. RG does not yet however give possibility to utilize cloud-based pay-per-use ODS like Google cloud vision offers [37]. This partly due to the interface semantic abstraction, but also on the protocol. A RG REST client would facilitate RG to work with most cloud-based services, not only the ones that support gRPC like GCV [37].

In the delimitation section of chapter 1 it was stated that this thesis will not cover real-time aspects. Nonetheless, it should not be disregarded in practice. Assuming TSN becomes a predominant technology in the next years, numerous aspects need to be considered. First, new hardware has to be purchased. That can be machines that are TSN-ready, network switches or network adapters. Moreover, the interface protocols need to be exchanged. As gRPC is a protocol designed for information technology, it focuses on being excessively fast - but not right on time. Lastly, the underlying ODsM need to be designed and tested against real-time requirements. To sum up, in the current state the framework can only be used for non time critical processes.

5.3.3 Handling Object Detection Services with Varying Detection Methods

RG's approach of coping with varying ODMs is abstraction. It is shown that modern complex ODMs basically comprise two phases - preparation (Train) and detection (Test). For RG to prosper, one party has to grasp every new ODM to adhere it to the two abstracted methods. That can only happen if the two methods are accepted as generic semantic interfaces. This calls for a standardization organization like the OPC foundation. RG can act as the spark of this fruitful thought. Taken into

account how much cost and lock in effects arouse during the fieldbus war [53], it is particularly important to get together at an early stage.

RG further relies on strong decoupling of hardware and software. The ODMs can be configured with a file for calibration and OD optimization. When it is implemented, the ODM should be as concise as possible.

The granularity of the services is another important domain that needs to be considered. Trading off between performance and reusability is the challenge here. Currently, RG delivers high performance due to coarse granularity [54]. Due to the strict pose output of the Test method, it is not possible to combine multiple ODMs for a greater task externally. However, the fine-granular services (e.g., edge detection) can be hidden behind the external method. As Docker offers a layered file system, new components can be added simply. This also implies that ODM providers to need do not need to expose their know how and can remain a propietary license.

Chapter 6

Conclusion and Future Research Directions

6.1 Conclusion

The aim of this thesis was to automatically generate ODSs with varying detection methods and interfaces. Hence, this results in three challenges:

- Generating ODS automatically
- Handling ODS with varying detection methods
- Handling ODS with varying interfaces

In response, a SOA concept which tackles these challenges was created. Generating ODS (semi-) automatically has been undergird with a supporting framework. It helps to combine the necessary components of camera image acquisition, ODMs and returning the pose in a standardized way. These services must adhere to a versioned semantic "Train" and "Test" definition which is generic enough to comprehend a broad set of ODMs. Expanding the semantic to a new version is of little effort.

The service interfaces are currently limited to gRPC or REST. Albeit this covers two established interfaces, they lack certain functionality such as message queuing and real-time functionalities. A more holistic implementation is desirable for the fu-

ture, e.g. with OPC UA over TSN. This would allow for even greater interoperability and reliability.

6.2 Future Research Directions

Further research directions include fulfilling all OPC UA Vision specification aspects once they are fully released. Especially the drill down to component level and a specification of how to interface cameras will offer the possibility for high-speed implementation of new components.

Also, the framework could be upgraded with smart aspects such as choosing the right camera for a detection task or automated training when a new product type is produced. With the help of these features, recipes could be generated automatically.

A very user-friendly way of extending the framework would be utilizing a catalogue of ODMs and possible components. Examples of these catalogues can be found on Eclipse Vorto [52] or the "yellow pages" for services [55].

Bibliography

[1] VDMA. OPC UA Companion Specification Vision – Part 1: Control, configuration management, recipe management, result management. Draft VDMA 40100-1:2018-11, 2018.

[2] Cm M. MacKenzie, Ken Laskey, Francis McCabe, Peter F. Brown, and Rebekah Metz. Reference Model for Service Oriented Architecture 1.0. OASIS Standard, 2006.

[3] Sebastian Houben, Johannes Stallkamp, Jan Salmen, Marc Schlipsing, and Christian Igel. Detection of traffic signs in real-world images: The German traffic sign detection benchmark. In *The 2013 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 8 2013. ISBN 978-1-4673-6129-3. doi: 10.1109/IJCNN.2013.6706807. URL `http://ieeexplore.ieee.org/document/6706807/`.

[4] Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for autonomous driving? The KITTI vision benchmark suite. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3354–3361. IEEE, 6 2012. ISBN 978-1-4673-1228-8. doi: 10.1109/CVPR.2012.6248074. URL `http://ieeexplore.ieee.org/document/6248074/`.

[5] Martin Rudorfer and Jörg Krüger. Industrial Image Processing Applications as Orchestration of Web Services. *Procedia CIRP*, 76:144–148, 2018. ISSN 22128271. doi: 10.1016/j.procir.2018.01.030. URL `https://linkinghub.elsevier.com/retrieve/pii/S2212827118300441`.

[6] Carlos Gonzalez. Smart Manufacturing Looks to Advances in 3D Machine Vision. Last visited Aug 20th 2019, 2017. URL `https://www.machinedesign.com/iot/smart-manufacturing-looks-advances-3d-machine-vision`.

[7] Alexander Hornberg. *Handbook of Machine Vision*. Wiley, Weinheim, 2nd edition, 7 2017. ISBN 978-3-527-41339-3. URL https://onlinelibrary.wiley.com/doi/book/10.1002/9783527610136.

[8] Robert Wilmes. Zauberwort Konvergenz. *Computer & Automation Sonderheft TSN & OPC UA*, pages 6–10, 4 2019.

[9] OpenCV-Documentation. Template Matching. Last visited Dec 15 2018, 2018. URL https://docs.opencv.org/2.4/doc/tutorials/imgproc/histograms/template_matching/template_matching.html.

[10] Bertram Drost, Markus Ulrich, Nassir Navab, and Slobodan Ilic. Model globally, match locally: Efficient and robust 3D object recognition. In *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 998–1005, San Francisco, 6 2010. IEEE. ISBN 978-1-4244-6984-0. doi: 10.1109/CVPR.2010.5540108. URL http://ieeexplore.ieee.org/document/5540108/.

[11] Martin Sundermeyer, Zoltan-Csaba Marton, Maximilian Durner, Manuel Brucker, and Rudolph Triebel. Implicit 3D Orientation Learning for 6D Object Detection from RGB Images. In *European Conference on Computer Vision*, pages 712–729. Springer, Munich, 2018. ISBN 9783030012304. URL http://link.springer.com/10.1007/978-3-030-01231-1_43.

[12] Stefan Hinterstoisser, Vincent Lepetit, Slobodan Ilic, Stefan Holzer, Gary Bradski, Kurt Konolige, and Nassir Navab. Model Based Training, Detection and Pose Estimation of Texture-Less 3D Objects in Heavily Cluttered Scenes. In *Asian Conference on Computer Vision*, pages 548–562. Springer, 2013. ISBN 9783642373305. URL http://link.springer.com/10.1007/978-3-642-37331-2_42.

[13] Tomáš Hodaň, Frank Michel, Eric Brachmann, Wadim Kehl, Anders Glent Buch, Dirk Kraft, Bertram Drost, Joel Vidal, Stephan Ihrke, Xenophon Zabulis, Caner Sahin, Fabian Manhardt, Federico Tombari, Tae Kyun Kim, Jiří Matas, and Carsten Rother. BOP: Benchmark for 6D object pose estimation. In *European Conference on Computer Vision*, pages 19–35, Munich, 2018. Springer. ISBN 9783030012489.

[14] Mark Richards. *Microservices vs Service Oriented Architecture*. O'Reilly Media, Sebastopol, CA, 2015. ISBN 9781491952429.

[15] Sam Newman. *Building Microservices*. O'Reilly Media, Inc., 2015. ISBN 978-1-491-95035-7.

[16] Andrew Banks and Rahul Gupta. MQTT Version 3.1.1. OASIS Standard. Last visited Dec 15 2018, 2014. URL http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html.

[17] 1Sheeld. Pure-javascript-MQTT-broker. Last visited Dec 15 2018, 2018. URL https://1sheeld.com/mqtt-protocol/pure-javascript-mqtt-broker/.

[18] RabittMQ-Documentation. Which protocols does RabbitMQ support? Last visited Dec 15 2018, 2018. URL https://www.rabbitmq.com/protocols.html.

[19] Jasenka Dizdarević, Francisco Carpio, Admela Jukan, and Xavi Masip-Bruin. A Survey of Communication Protocols for Internet of Things and Related Challenges of Fog and Cloud Computing Integration. *ACM Computing Surveys*, 51(6):1–29, 1 2019. ISSN 03600300. doi: 10.1145/3292674. URL http://dl.acm.org/citation.cfm?doid=3303862.3292674.

[20] Roy Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000. URL https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[21] Gigi Sayfan. REST vs. gRPC: Battle of the APIs. Last visited Dec 15 2018, 2018. URL https://code.tutsplus.com/tutorials/rest-vs-grpc-battle-of-the-apis--cms-30711.

[22] Google-Cloud-Documentation. Cloud Endpoints für gRPC. Last visited Dec 15 2018, 2018. URL https://cloud.google.com/endpoints/docs/grpc/about-grpc.

[23] gRPC-Gateway-Documentation. grpc-gateway. Last visited Dec 15 2018, 2017. URL https://github.com/grpc-ecosystem/grpc-gateway.

[24] Google-API-Documentation. http.proto. Last visited July 11th 2019, 2019. URL https://github.com/googleapis/googleapis/blob/master/google/api/http.proto#L46.

[25] GraphQL-Documentation. Basics Tutorial - Introduction. Last Visited Dec 15 2018, 2018. URL https://www.howtographql.com/basics/0-introduction/.

[26] Miriam Schleipen, Syed-Shiraz Gilani, Tino Bischoff, and Julius Pfrommer. OPC UA & Industrie 4.0 - Enabling Technology with High Diversity and Variability. *Procedia CIRP*, 57:315–320, 2016. ISSN 22128271. doi: 10.1016/j.procir.2016.11.055. URL https://linkinghub.elsevier.com/retrieve/pii/S2212827116312094.

[27] OPC-Foundation. OPC Unified Architecture Part 14: PubSub Release 1.04, 2018.

[28] Andreas Eckhardt, Sebastian Muller, and Ludwig Leurs. An Evaluation of the Applicability of OPC UA Publish Subscribe on Factory Automation use Cases. In *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1071–1074. IEEE, 9 2018. ISBN 978-1-5386-7108-5. doi: 10.1109/ETFA.2018.8502445. URL https://ieeexplore.ieee.org/document/8502445/.

[29] IEEE Standard for Local and metropolitan area networks–Bridges and Bridged Networks. *IEEE Std 802.1Q-2018 (Revision of IEEE Std 802.1Q-2011)*, July: 1–1993, 2018. doi: 10.1109/IEEESTD.2014.6991462.

[30] Dietmar Bruckner, Marius Petru Stanica, Richard Blair, Sebastian Schriegel, Stephan Kehrer, Maik Seewald, and Thilo Sauter. An Introduction to OPC UA TSN for Industrial Communication Systems. *Proceedings of the IEEE*, 2019. ISSN 00189219. doi: 10.1109/JPROC.2018.2888703.

[31] Carsten Emde. Die Frage der Lizenzen. *Computer & Automation Sonderheft TSN & OPC UA*, pages 37–39, 4 2019.

[32] Jesper Rönnholm. Integration of OPC Unified Architecture with IIoT Communication Protocols in an Arrowhead Translator. Master's Thesis, 2018.

[33] Peter Wurbs. Docker versus VM: Wie Container die heutige IT veraendern. Last visited Dec 15 2018., 2017. URL https://jaxenter.de/docker-vs-vm-54816.

[34] Thomas Thurig. Docker vs. Heroku. Last visited Dec 15 2018, 2014. URL https://tuhrig.de/docker-vs-heroku/.

[35] Tozzi Chris. Why and How to Move from Heroku to Docker. Last visited Dec 15 2018, 2017. URL https://codefresh.io/docker-tutorial/move-heroku-docker/.

[36] International Organization For Standardization. ISO/IEC 7498-1:1994 Information Technology - Open Systems Interconnection - Basic Reference Model: The Basic Model (2nd ed.), 1996. ISSN 10754210.

[37] Google-Cloud-Documentation. Google Cloud Vision RPC API. Last visited 2019-04-26, 2019. URL https://cloud.google.com/vision/docs/reference/rpc/.

[38] Philip Bianco, Rick Kotermanski, and Paulo F Merson. Evaluating a Service-Oriented Architecture. Technical report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, 2007. URL http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=8443.

[39] Azure-Documentation. Was ist Azure Stack? Last visited August 11 2019, 2019. URL https://azure.microsoft.com/de-de/overview/azure-stack/.

[40] Reinhard Heister. The Future of OPC UA Vision. Phone Interview, 2019.

[41] EMVA. GenICam Interface Standard. Last visited May 4th 2019, 2019. URL https://www.emva.org/standards-technology/genicam/.

[42] Docker-Py-Documentation. Docker SDK for Python. Last visited May 4th 2019, 2019. URL https://docker-py.readthedocs.io/en/stable/.

[43] gRPC-Documentation. Last visited May 4th 2019, 2019. URL https://grpc.io/docs/.

[44] FreeOpcUa-Documentation. OPC UA Python. Last visited May 4th 2019, 2019. URL https://github.com/FreeOpcUa/python-opcua.

[45] Emile Anclin. Pyreverse : UML Diagrams for Python. Last visited Aug 21st 2019, 2008. URL https://www.logilab.org/blogentry/6883.

[46] Mandy Waite. Scalable Microservices with gRPC, Kubernetes, and Containers. Ukraine, 2015. URL https://speakerdeck.com/googlecloudplatform/scalable-microservices-with-grpc-kubernetes-and-containers-devfest-ukraine?slide=2.

[47] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere. The architecture tradeoff analysis method. In *Proceedings. Fourth IEEE International Conference on Engineering of Complex Computer Systems (Cat. No.98EX193)*, pages 68–78. IEEE Comput. Soc, 1998. ISBN 0-8186-8597-2. doi: 10.1109/ICECCS.1998.706657. URL `http://ieeexplore.ieee.org/document/706657/`.

[48] Rick Kazman, Mario Barbacci, Mark Klein, S. Jeromy Carrière, and Steven G. Woods. Experience with performing architecture tradeoff analysis. In *Proceedings of the 21st international conference on Software engineering - ICSE '99*, pages 54–63, New York, New York, USA, 1999. ACM Press. ISBN 1581130740. doi: 10.1145/302405.302452. URL `http://portal.acm.org/citation.cfm?doid=302405.302452`.

[49] Len Bass, Robert Nord, William Wood, and David Zubrow. Risk Themes Discovered through Architecture Evaluations. In *2007 Working IEEE/IFIP Conference on Software Architecture (WICSA'07)*, pages 1–1. IEEE, 1 2007. ISBN 0-7695-2744-2. doi: 10.1109/WICSA.2007.37. URL `http://ieeexplore.ieee.org/document/4077018/`.

[50] Andrzej Zalewski and Szymon Kijas. Beyond ATAM: Early architecture evaluation method for large-scale distributed systems. *Journal of Systems and Software*, 86(3):683–697, 3 2013. ISSN 01641212. doi: 10.1016/j.jss.2012.10.923. URL `https://linkinghub.elsevier.com/retrieve/pii/S0164121212003032`.

[51] Lawrence Jones and Anthony Lattanze. *Evaluate a Wargame Simulation System: A Case Study*. Carnegie Mellon University, Pittsburgh, 2001. URL `http://www.sei.cmu.edu/publications/documents/01.reports/01tn022.html`.

[52] Eclipse. Vorto RepositoryBETA. Last visited Aug 21st 2019, 2019. URL `https://vorto.eclipse.org/#/`.

[53] Max Felser and Thilo Sauter. The fieldbus war: history or short break between battles? In *4th IEEE International Workshop on Factory Communication Systems*, pages 73–80. IEEE, 2002. ISBN 0-7803-7586-6. doi: 10.1109/WFCS.2002.1159702. URL `http://ieeexplore.ieee.org/document/1159702/`.

[54] Martin Rudorfer, Tessa J Pannen, and Jörg Krüger. A Case Study on Granularity of Industrial Vision Services. In *Proceedings of the 2Nd International Sym-*

posium on Computer Science and Intelligent Control, ISCSIC '18, pages 59:1–59:6, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-6628-1. doi: 10.1145/3284557.3284713. URL http://doi.acm.org/10.1145/3284557.3284713.

[55] F. Kretschmer and A. Lechler. Teilnehmerverwaltung und -zuordnung innerhalb einer cloudbasierten Steuerungsplattform. Technical report, ISW Stuttgart, Stuttgart, 2014.

Appendix A

Roles

Recipe Generator Client	The instance calling recipe generator methods Train, Test, Transfer, GenRecipe and GetConfig. It needs to have a certain intelligence to decide when to call which methods with the appropriate parameters. In the current state, this will most likely be a production engineer. It is assumed that adding or altering a recipe does not happen very frequently. In later use where more business intelligence resides inside recipe generator, the client could be a component, e.g. a PLC.
Detector Image Provider	The instance developing new ODMs. This could be research teams, community projects like OpenCV or OD specialists of the company using recipe generator. For every new ODM, it is necessary to adhere to the recipe generator API or, for a new pattern of ODMs, update the recipe generator API.
Camera Image Provider	The instance providing new camera hardware and capable software of interacting with recipe generator.

Framework Architect	The author or contributor of recipe generator. He or she is responsible for the SOA design, continuous development and integration and updated proto files. Often the same person as the recipe genetor client.

Appendix B

Example of a Recipe Life Cycle

To illustrate recipe management, here is a possible life cycle of a recipe taken from the OPC UA Vision specification appendix [1]. Note that the method signatures are not necessarily exact here.

1. A recipe for ProductId-m is created externally (often centrally).
2. The recipe is pushed to the vision system with ExternalId-1, ProductId-m (using AddRecipe())
    - It is stored there with ExternalId-1, InternalId-11.
    - It is linked to ProductId-m on the vision system
3. There are further possible actions on the recipe without any particular order.
    - The recipe may be edited locally later, keeping its ExternalId-1 and receiving InternalId-12.
    - A (binary) different version of the recipe with the same ExternalId-1 may be pushed to the vision system later, receiving InternalId-13.
    - The recipe may be linked later to ProductId-n on the vision system. Note that the external recipe management does not concern itself with the InternalIds of the recipes on different vision systems. If there are, due to one of these operations, several recipes on the vision system with identical ExternalIds but different InternalIds, the vision system/server combination has no means of telling which of these was targeted by the

environment. It may choose to link all of them, or the newest one, or the latest one pushed (ignoring internally edited ones). This is outside the scope of this specification.

4. The automation system is undergoing a change-over process to a specific product, namely ProductId-m. It will re-tool the vision system
   - by calling PrepareRecipe(ExternalId-1); the vision system then selects one of the existing recipes with ExternalId-1 based on internal rules.
   - by calling PrepareRecipe(ProductId-m); the vision system then selects one of the existing recipes linked to ProductId-m based on internal rules.
5. The vision system is commanded to process a specific product
   - by calling StartJob(ExternalId-1); the vision system then starts processing with the recipe prepared for ExternalId-1.
   - by calling StartJob(ProductId-m); the vision system then starts processing with the recipe prepared for ProductId-m.
   - by calling StartJob(ExternalId-2); the vision system then throws an error because no such recipe has been added or prepared.
   - by calling StartJob(ProductId-p); the vision system then throws an error because no such recipe has been added or prepared.
6. If there is no error, the vision system carries out its task, going through the Executing state to return to state Ready waiting for further instructions. There are many other possibilities of errors, e.g. trying to prepare a recipe which is actually a sub-recipe, i.e., not capable of being processed by itself.

Appendix C

ATAM Evaluation

C.1 Quality Attribute Scenarios

For an explanaition of the sources (i.e., roles), see annex A.

Table C.1: Quality Attribute Scenarios

Number	**Quality Attribute**	**Scenario**	
1	Modifiability	Source:	Detector Docker Image Provider
		Stimulus:	Add a new detector
		Artifact:	Docker Registry
		Environment:	Detector provider familiar with Docker, gRPC and detector proto file
		Response:	New detector is added

Continued on next page

Table C.1 – *Continued*

Number	**Quality Attribute**	**Scenario**	
		Response Measure:	No more than five person-days of detector provider team effort is required for the implementation (legal and financial agreements are not included).
2	Modifiability	Source:	Camera Docker Image Provider
		Stimulus:	Add a new camera to the vision system
		Artifact:	Camera
		Environment:	Camera provider familiar with Docker, gRPC and camera proto file
		Response:	New camera is added
		Response Measure:	No more than four person-days of camera provider team effort is required for the implementation (legal and financial agreements are not included).
3	Modifiability	Source:	RG Client
		Stimulus:	Add a new proto file version
		Artifact:	Camera or Detector
		Environment:	A new version of a proto file is necessary due to missing message types
		Response:	New proto file is added

Continued on next page

Table C.1 – *Continued*

Number	**Quality Attribute**	**Scenario**	
		Response Measure:	No more than one person-day of effort is necessary. All existing camera and detector service remain functional.
4	Interoperability	Source	Framework Architect
		Stimulus:	Replace or scratch Docker
		Artifact:	Detector and Camera Image Docker Containers
		Environment:	Docker is not allowed or not possible to use
		Response:	Docker is replaced or scratched
		Response Measure:	No more than fifteen person-days of framework architect effort is required for the implementation
5	Performance	Source	Detector Docker Image Provider
		Stimulus:	Send a file to detector
		Artifact:	Detector Docker Container
		Environment:	CAD file input not larger than 10 MB sent to detector
		Response:	File to detector sent
		Response Measure:	The trip time for a request from detector client in the local network to detector during normal operation is less than 2 seconds.

Continued on next page

Table C.1 – *Continued*

Number	**Quality Attribute**	**Scenario**	
6	Reliability	Source	Framework Architect
		Stimulus	Service failure
		Artifact	All components
		Environment	A sudden failure occurs in the runtime environment of a service.
		Response	After recovery, all transactions are completed or rolled back as appropriate, so the system maintains uncorrupted, persistent data.
		Response Measure:	Manual test if data is corrupt or inconsistent.
7	Security	Source	A possible attacker
		Stimulus	A malware is included in the system
		Artifact	Camera or Detector Docker Container
		Environment	We have to face a denial-of-service attack.
		Response	The malware is detected quickly, best before entering the framework. Necessary information to possibly identify the attacker is logged.
		Response Measure:	Our system is not operable for not more than an hour.

C.2 Architectural Analysis of Scenarios

Table C.2: Architectural Analysis of Scenarios

Analysis for Scenario 1	
Scenario Summary	New Detector added to Docker Registry in no more than five person-days
Business Goal(s)	Extending the available object detection possibilities
Quality Attribute	Modifiability
Architectural Approaches and Reasoning	Detector communication depends on gRPC which covers many programming languages and a REST gateway. Docker registry can be made easily accessible for the provider and is a well-known technology.
Risks	Possible too much of a lock-in effect of both technologies
Tradeoffs	The static Train/Test interface might not (yet) be suitable for all detection purposes, although once implemented it is very easy to use and might become more popular as a versioned standard.
Analysis for Scenario 2	
Scenario Summary	New camera added to vision system in no more than four person-days
Business Goal(s)	Possibility of heterogeneous camera ecosystem
Quality Attribute	Modifiability

Continued on next page

Table C.2 – *Continued*

Architectural Approaches and Reasoning	Docker container including or alternatively just gRPC server running on camera which allows for many programming languages
Risks	The camera firmware can be unmodifiable, thus a middleware is likely to be needed.
Tradeoffs	In theory, the same underlying technology (Docker + gRPC) can be conveniently used for communication between recipe and camera, but in practice, at least for the camera, this might end up cumbersome without a fitting middleware.
Analysis for Scenario 3	
Scenario Summary	A new proto file version is added in no more than on person-day.
Business Goal(s)	Staying up to date with current ODMs
Quality Attribute	Modifiability
Architectural Approaches and Reasoning	gRPC offers a convenient way of expanding proto files. As long as no messages are deleted or overwritten, the system remains downward compatible.
Risks	None
Tradeoffs	The more messages are created, the more overhead will be sent over the channels.
Analysis for Scenario 4	
Scenario Summary	Replace or scratch Docker in no more than 15 person-days.
Business Goal(s)	Permit easy integration with new business partners.

Continued on next page

Table C.2 – *Continued*

Quality Attribute	Interoperability
Architectural Approaches and Reasoning	Detector and Camera gRPC servers/clients do not need Docker for successful communication. It offers a huge convenience for sharing them, starting them, handling dependencies, etc. A switch to e.g. Heroku would require a partial rewrite of the framework.
Risks	Too much effort included rewriting the framework. Instead, an alternative framework might be used.
Tradeoffs	If Docker is available, the ease of service administration is very high. If not, the flexibility is higher but requires more effort.
Analysis for Scenario 5	
Scenario Summary	A 10 MB file is sent to detector in less than 2 seconds.
Business Goal(s)	Compatible message transmission speed.
Quality Attribute	Performance
Architectural Approaches and Reasoning	gRPC sends its data via protobufs which are a binary format specialized in high-performance serialization and deserialization. If an image has to be transferred between OPC UA Server and Client, then the emerging Time Sensitive Networking will help facing real-time transmission challenges.
Risks	If too slow, the process might be the bottleneck in production.
Tradeoffs	None

Continued on next page

Table C.2 – *Continued*

Analysis for Scenario 6	
Scenario Summary	Uncorrupted, persistent data remains on the system after a service failure.
Business Goal(s)	Industrial-grade reliability
Quality Attribute	Reliability
Architectural Approaches and Reasoning	The framework relies on service choreography and client/server communication. Thus, every component has to ensure uncorrupted and persistent data. In the future, a messaging broker could be introduced for better reliability.
Risks	Currently, no reliability is granted, so a service failure will result in a failure of most components of the system
Tradeoffs	Adding a message broker adds reliability but also complexity.
Analysis for Scenario 7	
Scenario Summary	A denial-of-service attack causes no more than an hour of downtime of the framework.
Business Goal(s)	Offering a secure framework with little downtime.
Quality Attribute	Security

Continued on next page

Table C.2 – *Continued*

Architectural Approaches and Reasoning	The Docker Hub is intended to be accessible by many vendors to ensure a variety of detectors and cameras. Every new vendor has to ensure its integrity. New detectors and cameras have to be tested thoroughly for a possible malware function. Through the standardized interface, a malfunctioning service can best be replaced quickly by an equivalent one offered by another vendor.
Risks	A service that is relied on in production reveals itself as malware can be hard to replace in the given time of one hour. Redundant services by multiple vendors can help out in this case.
Tradeoffs	Due to possibly high numbers of services and creative attackers, testing every new vendor and service can be a tedious task.

Appendix D

Proto Files

D.1 Detector

```
syntax = "proto3";
//Build command for gRPC python stubs goes something like:
//py -m grpc_tools.protoc -I. --python_out=. --grpc_python_out=. .\detector.proto

// The detector service definition
service Detector {
  // trains the detector with CADModel
  rpc Train (stream DetectorInput) returns (TrainResult) {
}
  // runs object detection and returns Pose
  rpc Test (stream DetectorInput) returns (TestResult) {
  }
  // returns the detector configuration file
  rpc GetConfig (Empty) returns (Config){
  }
}

message DetectorInput {
  // CAD file, Image...
  Image Content = 1;
  // Configuration of detector
  string Config = 2;
}

```

```
25 message TrainResult {
26   // OK, NOK, created 10k templates...
27   string Result = 1;
28   // Image(s) as illustration of Train Result. Optional.
29   repeated Image Image = 2;
30 }
31
32 message TestResult {
33   // OK, NOK, Unknown...
34   string Result = 1;
35   // Pose(s) of detected object(s)
36   repeated Pose Pose = 2;
37   // Image(s) as illustration of Test Result.
38   repeated Image Image = 3;
39 }
40
41 message Image {
42   bytes Content = 1;
43   // Prominent parts of the image
44   repeated Highlight Highlights = 2;
45 }
46
47 // e.g. bounding box, feature etc. two points are specified
48 message Highlight{
49    uint32  top = 1;
50    uint32  left = 2;
51    uint32  bottom = 3;
52    uint32  right = 4;
53 }
54
55 // 3D pose of an object relative to a fixed point
56 message Pose {
57   double x = 1;
58   double y = 2;
59   double z = 3;
60   double alpha = 4;
61   double beta = 5;
62   double gamma = 6;
63 }
64
65 message Config {
66   string Config = 1;
67 }
68
```

```
69 message Empty {}
```

D.2 Camera

```
1  syntax = "proto3";
2  //Build command for gRPC python stubs goes something like:
3  //py -m grpc_tools.protoc -I. --python_out=. --grpc_python_out=. ↩
       .\camera.proto
4
5  // The camera service definition
6  service Camera {
7    // takes an image with an optional camera config
8    rpc GrabImage (CamConfig) returns (CamImage) {
9    }
10   // returns the camera configuration file
11   rpc GetConfig (Empty_) returns (CamConfig){
12   }
13 }
14
15
16 message CamImage {
17   bytes Content = 1;
18 }
19
20 message CamConfig {
21   string Config = 1;
22 }
23
24 message Empty_ {}
```