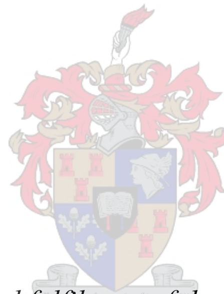
Extending the BASE architecture for complex and reconfigurable Cyber-Physical Systems using holonic principles

by
Daniel Jacobus van Niekerk



Thesis presented in partial fulfilment of the requirements for the degree of Master of Engineering (Mechatronic) in the Faculty of Engineering at Stellenbosch University

Supervisor: Dr Karel Kruger
Co-supervisor: Prof Anton Basson

December 2021

Stellenbosch University https://scholar.sun.ac.za

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date: December 2021

Copyright © 2021 Stellenbosch University

All rights reserved

Stellenbosch University https://scholar.sun.ac.za

Abstract

Extending the BASE architecture for complex and reconfigurable Cyber-Physical Systems using holonic principles

D.J. van Niekerk

Department of Mechanical and Mechatronic Engineering
Stellenbosch University
Private Bag X1, 7602 Matieland, South Africa

Thesis: M.Eng. (Mechatronic Engineering)
December 2021

Industry 4.0 (I4.0) represents the newest technological revolution aimed at optimising industries using drivers such as Cyber-Physical Systems (CPSs), the Internet of Things (IoT) and many more. In the past two decades, the holonic paradigm has become a major driver of intelligent manufacturing systems, making it ideal to advance I4.0.

The objective of this thesis is to extend an existing holonic reference architecture, the Biography-Attributes-Schedule-Execution (BASE) architecture, for complex and reconfigurable CPSs. In the context of this thesis, complex and reconfigurable systems are considered to be systems that are comprised of many diverse, autonomous and interacting entities, and of which the functionality, organization or size is expected to change over time. The thesis applies the principles of holonic systems to manage complexity and enhance reconfigurability of CPS applications.

The BASE architecture is extended for two reasons: to enable it to integrate many diverse entities, and to enhance its reconfigurability. With regards to research on holonic systems, this thesis aims to address two important functions for systems implemented using holonic principles, namely cooperation and cyber-physical interfacing.

The most important extensions made to the architecture were to enable scalability, refine the cooperation between holons, and integrate cyber-physical interfacing services as Interface Holons. These extensions include platform management components (e.g. a service directory) and standardised plugins (e.g. cyber-physical interfacing plugins). The extended architecture was implemented on an educational sheep farm, because of the many heterogeneous resources (sheep, camps, sensors, humans, etc.) on the farm that need to be integrated into a BASE architecture implemented CPS. This case study implementation had to integrate data from different sensors, provide live analysis of observed data and, when required, notify

Stellenbosch University https://scholar.sun.ac.za

the physical world of any problems in the CPS. At the end of the implementation, an evaluation was done using the requirements of a complex, reconfigurable CPS as evaluation criteria. This evaluation involved setting up quantitative and qualitative evaluation metrics for the evaluation criteria, doing the evaluations, and discussing what the results from the different evaluations indicate about the effectiveness and efficiency of the extensions made to the BASE architecture.

The extensions made to the BASE architecture were found to improve robustness and resilience. The use of Erlang was found to play a very important role in the resulting reliability. The extensions also helped to fully address the original BASE architecture's scalability shortcomings and to increase development productivity. Lastly, the extensions show the benefits of using service orientation to enable cooperation between holons and how extracting all cyber-physical interfacing of a system into dedicated Interface Holons reduces development time, improves reusability and enhances diagnosability of interfacing problems.

Stellenbosch University https://scholar.sun.ac.za

Uittreksel

Uitbreiding van die BASE-argitektuur vir komplekse en herkonfigureerbare Kuber-Fisiese Stelsels deur die gebruik van holoniese beginsels

D.J. van Niekerk

Departement van Meganiese and Megatroniese Ingenieurswese
Universiteit Stellenbosch
Privaatsak X1, 7602 Matieland, Suid-Afrika
Tesis: M.Ing. (Megatroniese Ingenieurswese)
Desember 2021

Industrie 4.0 (I4.0) is die nuutste tegnologiese revolusie en dit is daarop gemik om industrieë te optimiseer deur middel van drywers soos Kuber-Fisiese Stelsels (KFSs), die *Internet of Things* (IoT) en vele meer. In die afgelope twee dekades het die holoniese paradigma ŉ belangrike drywer van intelligente vervaardigingstelsels geword, wat dit ideaal maak om I4.0 te bevorder.

Die doel van hierdie tesis is om ‘n bestaande holoniese verwysings argitektuur, die *Biography-Attributes-Schedule-Execution* (BASE-) argitektuur, uit te brei vir komplekse, herkonfigureerbare KFSs. In die konteks van hierdie tesis, word komplekse en herkonfigureerbare stelsels gesien as stelsels wat bestaan uit menige diverse, outonome entiteite wat met mekaar interaksie het en waarvan die funksionaliteit, organisasie en grootte verwag is om te verander met verloop van tyd. Hierdie tesis pas die beginsels van holoniese stelsels toe om die kompleksiteit van KFSs te bestuur en om herkonfigureerbaarheid van KFSs te verbeter.

Die BASE-argitektuur word uitgebrei om twee redes, naamlik om die integrasie van menige diverse entiteite te ondersteun en om die argitektuur se herkonfigureerbaarheid te verbeter. Die studie sal ‘n navorsingsbydrae lewer oor holoniese stelsels deur twee belangrike funksionaliteite van stelsels wat geïmplementeer is deur middel van holoniese stelsels aan te spreek – samewerking tussen holons en kuber-fisiese koppeling.

Die belangrikste uitbreidings wat gemaak is aan die argitektuur was om skaleerbaarheid moontlik te maak, samewerking tussen holons te verfyn en om kuber-fisiese koppelingsdienste te integreer as holons. Hierdie uitbreidings sluit nuwe platformbestuurkomponente en gestandaardiseerde *plugins* in. Die uitgebreide argitektuur is geïmplementeer op ŉ opvoedkundige skaapplaas, omdat die skaapplaas baie heterogene hulpbronne (skape, kampe, sensors, mense, ens.) insluit wat in die BASE-argitektuur geïmplementeerde KFS geïntegreer kon word.

Stellenbosch University https://scholar.sun.ac.za

Hierdie gevallestudie-implementering moes data van verskillende sensors integreer, intydse analises doen van die waargeneemde data en wanneer nodig, ‘n entiteit in die fisiese wêreld inlig van enige probleme in die KFS. Aan die einde van die implementering is ŉ evaluering gedoen deur die vereistes van ŉ komplekse, herkonfigureerbare KFS as evalueringskriteria te gebruik. Die evaluering het bestaan uit die opstel van kwantitatiewe en kwalitatiewe evalueringsmaatreëls, die uitvoer van die evaluerings en ŉ bespreking van wat die evalueringsresultate aandui oor die effektiwiteit en doeltreffendheid van die uitbreidings wat aan die BASE-argitektuur gemaak is.

Dit is bevind dat die uitbreidings wat gemaak is aan die BASE-argitektuur robuustheid en veerkragtigheid verbeter het. Die gebruik van *Erlang* het ŉ groot rol gespeel in die gevolglike betroubaarheid. Die uitbreidings aan die BASE-argitektuur het ook gehelp om die argitektuur volledig skaleerbaar te maak en om ontwikkelingsproduktiwiteit te verbeter. Laastens, bewys die uitbreidings die voordele van diensoriëntasie in die samewerking tussen holons en hoe die gebruik van Koppelings Holons (*Interface Holons)* ontwikkelingstyd verminder, die herbruikbaarheid van programbronkode verbeter en diagnoseerbaarheid van koppelingsprobleme versterk.

Stellenbosch University https://scholar.sun.ac.za

Acknowledgements

First and foremost, I have to thank Doctor Karel Kruger. Thank you for your continuous guidance throughout this thesis. Your passion for engineering, and especially research is contagious. Thank you for all the extra unplanned meetings that helped me to not waste time on things that would not have contributed to my research. Also, the help you gave me with the writing of my thesis definitely lifted it to another level.

To Professor Basson, thank you for making me aware of the power of a single word and how to be cautious with my choice of terminology. Also thank you to both Professor Basson and Dr Kruger for providing me with the required funding for the past two years.

Dale, I have learned so much from you and thank you for all the long voice notes where you explained implementation details about the BASE architecture to me. I appreciate all your help and know you will excel in anything you do in life.

To the rest of the MAD research group, you have made the past two years very enjoyable. Even though we could not come into the MAD offices for a big chunk of these two years, the little time in which we shared lunches and coffee helped to push through when the work seemed unending.

Aan Ma en Pa: Dankie dat julle my vir so lank finansieel ondersteun het om my studies te voltooi. Ma, dankie vir al die kospakkies wat Ma op die vreemdste maniere by my uitgekry het vanaf Bloemfontein en dankie vir al ma se wyse woorde. Pa, op die dae wat my motivering min was kon ek net dink aan hoe hard Pa op 'n daaglikse basis werk en hoeveel ek daarna opkyk, om myself te motiveer om deur te druk.

Alessia, dankie vir al jou liefde en ondersteuning. Dankie dat jy altyd met regte belangstelling na al my verduidelikings oor my werk geluister het. Dankie aan jou ouers wat my altyd laat welkom voel het in julle huis.

Laastens dankie aan God wat my geseën het met my talente. Geen van my prestasies sou moontlik gewees het sonder U ondersteuning elke dag nie.

Stellenbosch University https://scholar.sun.ac.za

Dedication

Glory be to the Father, and the Son, and the Holy Spirit.

Stellenbosch University https://scholar.sun.ac.za

Table of contents

Page

List of figures xii

List of tables xv

List of abbreviations xvi

1 Introduction 1
- 1.1 Background 1
- 1.2 Objectives 2
- 1.3 Motivation 3
- 1.4 Methodology 4

2 Literature review 5
- 2.1 Cyber-Physical Systems 5
  - 2.1.1 Definition 5
  - 2.1.2 Characteristics 5
  - 2.1.3 Applications 10
- 2.2 Reconfigurable Manufacturing Systems 10
- 2.3 Holonic systems 11
  - 2.3.1 Background, definitions and principles 11
  - 2.3.2 Performance metrics of Holonic Control Architectures 12
- 2.4 The BASE architecture 13
  - 2.4.1 Core components and plugins 14
  - 2.4.2 The three-stage activity life cycle 15
  - 2.4.3 An Erlang implementation of the BASE architecture 15
  - 2.4.4 Application of the BASE architecture in industry 16
- 2.5 Conclusion 17

3 Requirements for the extension of the BASE architecture 18
- 3.1 Requirements of a complex, reconfigurable CPS 18
- 3.2 Evaluating the original BASE architecture 19
- 3.3 Extensions required to the BASE architecture 21

4 Platform management components 24
- 4.1 Overview 24
- 4.2 Supervision tree 24
- 4.3 Department of Holon Affairs 25

Stellenbosch University https://scholar.sun.ac.za

4.4 Loggers 26
4.5 User Interfaces 27

5 The refined BASE architecture adminstration shell 28

5.1 Overview 28
5.2 Coordinator 29
5.3 Schedule Gate 30
5.4 Changes to the existing BASE core components 30
5.5 BASE shell component privacy 31
5.6 Plugin configuration 32
5.6.1 Plugin-shell interfacing 32
5.6.2 Plugin configuration file 35
5.7 Activity types 36
5.7.1 Service Provision Activities and Resource Action Activities 36
5.7.2 Resource Data Acquisition Activities 37
5.8 Attribute types 38

6 Cooperation 39

6.1 Business Cards and Service Descriptions 39
6.2 Service discovery 40
6.3 Service Provision Activity data structures 41
6.4 Plugins for standardised service interactions 42
6.4.1 Call For Proposals and service requests 42
6.4.2 Service Provision Activity initiation 43
6.4.3 Service Provision Activity completion 44
6.5 Nested services 45

7 Cyber-physical interfacing 46

7.1 Overview 46
7.2 Interface plugins 47
7.3 Interface Service Descriptions 48
7.3.1 Overview 48
7.3.2 Identifiers 49
7.3.3 Topics 49
7.4 Inter- and intra-holon communications 50
7.4.1 Interface initialisation 50
7.4.2 Observers 51
7.4.3 Informers 53
7.5 Interface Holon activities and attributes 54

Stellenbosch University https://scholar.sun.ac.za

8 Generic implementation of architecture extensions 55
8.1 Storage components 55
8.2 Administrative components 56
8.3 Supervisors 57
8.4 The Erlang gen_server process, gen_server calls and spawn_monitor . 59
8.5 Inter-process communication in a distributed Erlang application 60
8.6 Standard plugins 60

9 Case study 62
9.1 Selection criteria 62
9.2 Description 62
9.3 User requirements 63
9.4 Holon identification 64
9.5 Implementation 65
9.5.1 Platform implementation and integration overview 65
9.5.2 Plugins and activities of case study holons 66

10 Evaluation 71
10.1 Criteria 71
10.2 Quantitative metrics 73
10.2.1 Analysis of implementation source code 74
10.2.2 Analysis of implementation reliability during deployment 75
10.2.3 Development and reconfiguration experiments 76
10.2.4 Scaling experiments 77
10.3 Qualitative evaluation 78
10.4 Discussion 79

11 Conclusion and future work 81

12 References 83

Appendix A Coordinator start-up operations and variables 90

Appendix B Standard activity data structures 92

Appendix C Contract Net Protocol applied to holonic systems 93

Appendix D Feeding sensor Interface Module 94

Appendix E Interface Holon attributes 98

Appendix F Case study holon attributes 101

Stellenbosch University https://scholar.sun.ac.za

Appendix G Python MQTT client for feeding sensor 113

Appendix H User Interface 119

Appendix I Computational requirements evaluation 126

Stellenbosch University https://scholar.sun.ac.za

List of figures

Page

Figure 1: A BASE implementation raising a human to CPS status to facilitate communication with other CPSs (Adapted from Sparrow, 2020) 2

Figure 2: A comprehensive schema of characteristics of CPSs (adapted from Napoleone *et al.* (2020)) 6

Figure 3: Requirements and performance metrics for holonic control implementations (Kruger & Basson, 2018) 13

Figure 4: The BASE architecture (Sparrow, 2021) 14

Figure 5: Shells and data blocks of stage 1, stage 2 and stage 3 activities in the 3SAL model (Sparrow *et al.*, 2020) 15

Figure 6: Generalised structure of a BASE core component data repository implemented in Erlang (Sparrow, 2021) 16

Figure 7: Action time visualisation from captured execution data during the layup activity 17

Figure 8: Relationship matrix showing relationships between the requirements of a complex, reconfigurable CPS and the required BASE architecture extensions 21

Figure 9: Overview of the extended BASE architecture 24

Figure 10: Holon interactions shown on a UI 26

Figure 11: Details of the refined BASE architecture digital administration shell .28

Figure 12: Plugin registrations 33

Figure 13: Schedule Gate informing an EP to start an activity (a) and an activity handler being created after the EP started the activity (b) 34

Figure 14: Updating stage 2 data and finishing an activity (a), and adding post-execution data to it (b) 35

Figure 15: Updating attributes and scheduling new activities 35

Figure 16: Plugin configuration file for all sheep holons 36

Figure 17: RAA vs SPA 37

Figure 18: RDAA used to observe temperature data recorded by a temperature observer holon 38

Figure 19: Data structures of two holons’ services, each with a "Drill" Service Type 40

Figure 20: Service discovery 40

Stellenbosch University https://scholar.sun.ac.za

Figure 21: Inter- and intra-holon interactions for selecting a service provision holon and requesting its services 42
Figure 22: Starting service provision activity..43
Figure 23: Inform, inform-done and failure 44
Figure 24: Nested Services 45
Figure 25: Use of Interface Holons in the BASE Architecture 47
Figure 26: Service Description data structure of cyber-physical interfacing services 48
Figure 27: Interface Holon initialisation 50
Figure 28: Service request to an Observer...51
Figure 29: New data observed by Observer and shared with clients 52
Figure 30: Service request to an Informer 53
Figure 31: Execution of an “Inform” service...54
Figure 32: Observed data of a sheep plotted in UI (adapted image from implementation’s UI with enlarged text)..56
Figure 33: DOHA supervisor process source code 58
Figure 34: Example of a gen_server process handling a call 59
Figure 35: Educational sheep farm setup 63
Figure 36: Implementation setup 65
Figure 37: Relationship matrix showing the relationship between the evaluation criteria and metrics of the extended BASE architecture 71
Figure 38: Logged error found in error log after forcefully killing the attribute reception process of a holon with ID = "Sheep1" 75
Figure 39: Scaling experiment results 77
Figure 40: Contract net protocol - figure adapted from the FIPA CNP diagram (FIPA Contract Net Interaction Protocol Specification, 2021) 93
Figure 41: UI home page 119
Figure 42: UI user administration page 120
Figure 43: UI resources page showing all sheep...121
Figure 44: UI resources page showing all sensors (Observers)..........................121
Figure 45: UI resource details showing a sheep’s attributes 122
Figure 46: UI resource details showing a sheep’s schedule, execution and biography 123
Figure 47: UI resource details showing a sheep’s resource data 124

Stellenbosch University https://scholar.sun.ac.za

Figure 48: Sheep data in downloaded excel file using UI – group sheet 125
Figure 49: Sheep data in downloaded excel file using UI – sheep sheet 125
Figure 50: Implementation's CPU usage before, during and after the addition of 100 holons to the system 126
Figure 51: Implementation's RAM usage before, during and after the addition of 100 holons to the systema...127

Stellenbosch University https://scholar.sun.ac.za

List of tables

Page

Table 1: Evaluation of how well the BASE architecture enables the requirements of a complex, reconfigurable CPS 20

Table 2: Intra-shell communication permissions 32

Table 3: Case study holons with their custom plugins and supported activities ...67

Table 4: Code reuse rate of the case study holons 74

Table 5: Code reuse rate in the service provision plugins 75

Table 6: Development and reconfiguration time of new holons...........................76

Table 7: Data structures of standard activities 92

Table 8: Service initialisation time and computational resource requirements of the implementation for different numbers of holons in the system ..127

Stellenbosch University https://scholar.sun.ac.za

List of abbreviations

3SAL	Three-Stage Activity Lifecycle
AP	Analysis Plugin
API	Application Programming Interface
AR	Augmented Reality
BASE	Biography-Attributes-Schedule-Execution
BC	Business Card
CFP	Call For Proposals
CNP	Contract Net Protocol
CPS	Cyber-Physical System
CPU	Central Processing Unit
DOHA	Department of Holon Affairs
EID	Electronic Identification Number
EP	Execution Plugin
FCR	Feeding Conversion Ratio
HCA	Holonic Control Architecture
HMS	Holonic Manufacturing System
HRH-AS	Human Resource Holon Administration Shell
HTTP	Hypertext Transfer Protocol
I4.0	Industry 4.0
IoT	Internet of Things
MQTT	Message Queuing Telemetry Transport
RAA	Resource Action Activity
RAM	Random Access Memory
RDAA	Resource Data Acquisition Activities
RFID	Radio-Frequency Identification
RP	Reflection Plugin
SBB	State Blackboard

Stellenbosch University https://scholar.sun.ac.za

SLOC	Source Lines Of Code
SoA	Service-oriented Architecture
SoHMS	Service-oriented Holonic Manufacturing System
SP	Scheduling Plugin
SPA	Service Provision Activity
UI	User Interface
URL	Uniform Resource Locator
VR	Virtual Reality

Stellenbosch University https://scholar.sun.ac.za

1 Introduction

This chapter introduces the background and context of the thesis, followed by the specific thesis objectives, the motivation for conducting this research and the research methodology that was applied.

1.1 Background

Industry 4.0 (I4.0) is the most recent industrial revolution following three others and includes technology such as the Internet of Things (IoT), cloud computing and artificial intelligence. Tay, Lee, Hamid & Ahmad (2018) define I4.0 as an aggregation of existing ideas and technologies into a new value chain. This involves connecting systems in a self-organising manner that enables dynamic control within an organisation.

CPSs are systems that use real-time data acquisition from physical components, together with feedback from the cyber parts of these components, to provide intelligent data management, advanced analytics and optimised control in complex systems (Lee, Bagheri & Kao, 2015). CPSs are able to collect data related to themselves and their environment, process and evaluate this data, communicate with other systems, and initiate actions (Thoben, Wiesner & Wuest, 2017).

Derigent, Cardin & Trentesaux (2020) observed that in the past two decades, the holonic systems paradigm has become a major driver of Intelligent Manufacturing Systems. They further show how Holonic Control Architectures (HCAs), which are built on the concept of *holons,* have evolved and can address the needs of I4.0. Koestler (1967) defined a *holon* as an entity with communication and decision-making capabilities that is composed of a set of sub-level holons, yet at the same time is part of a holarchy of higher-level holons. HCAs are composed of multiple autonomous and cooperating holons that are ordered in a holarchy (Bussmann, 1998).

In developing countries such as South Africa, human workers form a big part of most industries. Furthermore, full automation is not possible in most industries, because of the unmatched flexibility of human workers (Özkiziltan & Hassel, 2020). HCA research has considered the integration of humans as holons, but this has not been explored in depth (Sousa, Ramos & Neves, 2007). This is because research related to I4.0 has been predominantly focussed on automation and digital systems.

Sparrow (2021) identified the need for human-centric technology development and subsequently developed a holonic reference architecture, called the Biography-Attributes-Schedule-Execution (BASE) architecture, to elevate humans to CPS

Stellenbosch University https://scholar.sun.ac.za

status. This elevation would help companies exploit the full strengths of their labour force, consequently enhancing their competitiveness (Sparrow 2021).

In the BASE architecture, each human is integrated into the cyber world through their own BASE architecture digital administration shell (henceforth referred to as a BASE shell), which enables its human to interact with other holons through their BASE shells as illustrated in figure 1. This BASE shell stores, processes and communicates information relevant to the human and the services that the human can deliver to other holons.

Figure 1: A BASE implementation raising a human to CPS status to facilitate communication with other CPSs (adapted from Sparrow, 2020)

This thesis forms part of research from the Mechatronics, Automation and Design (MAD) Research Group. This group started with research on Reconfigurable Manufacturing Systems (RMSs), from which two streams of research have been flowing, namely, research on holonic systems (e.g. Kruger & Basson (2018)) and research on digital twins (e.g. Redelinghuys, Basson & Kruger (2019)). Sparrow (2021) identified the need to integrate humans into I4.0 environments and, by addressing this need, built upon the MAD group's research on holonic systems. This thesis aims to show that the BASE architecture is not limited to the integration of human workers, but can be extended to develop complex, reconfigurable CPSs that integrate many different types of resources.

1.2 Objectives

The objective of this thesis is to extend the BASE architecture for complex and reconfigurable CPSs using the principles of holonic systems. This extension includes the identification and development of platform management components (components that manage the platform on which BASE shells are deployed) and refinements to the existing architecture's BASE shells (including the development of standard plugins) to support complex and reconfigurable CPS applications. In the context of this thesis, complex and reconfigurable CPSs are considered to be systems that are comprised of many diverse, autonomous and interacting entities,

Stellenbosch University https://scholar.sun.ac.za

and of which the functionality, organization or size is expected to change over time. The thesis will apply the principles of holonic systems to manage complexity and enhance reconfigurability of CPS applications.

The evaluation of the BASE architecture as performed by Sparrow (2021), is beyond the scope of this thesis. Also, this thesis only focusses on applying the BASE architecture to resources, and the application of the BASE architecture for logistics or coordination processes is not considered.

1.3 Motivation

CPSs promise better integration, smoother interactions, uncertainty handling, better system performance, scalability, flexibility, and faster response times (Bhrugubanda, 2015). Derigent *et al.* (2020) state that holonic systems address nine out of ten key enablers of Industry 4.0, namely: sustainability, real-time capabilities, process virtualization, service orientation, integration, adaptability, big data analysis, autonomous and decentralized decision making and connectivity. Using principles from holonic systems can help to realise the promises of CPSs, and the extension of the BASE architecture aims to illustrate this.

Sparrow (2021) formulated most of the conceptual structures and rules within the BASE architecture and implemented the architecture in industry to show how it can integrate a few human workers in a larger, but not full-scale, application. Sparrow (2021) suggested that refinement of the internal structure of the BASE architecture's core components and attributes can make the architecture capable of integrating more than just human workers. This indicates the potential to use the BASE architecture to create complex and reconfigurable CPSs, consisting of heterogeneous resources. This project aims to prove this by making the necessary extensions to the BASE architecture.

The BASE architecture is extended for two reasons. The first reason is to expand its capabilities beyond the integration of human workers in I4.0, to the integration of many different types of resources. The second reason is to enable reconfigurability in the architecture. Napoleone, Macchi & Pozzetti (2020) identified reconfigurability as an important characteristic that CPSs should have. Koren, Gu & Guo (2018) showed how reconfigurability of RMSs can enhance manufacturing systems' responsiveness.

Many of the characteristics of RMSs can be integrated into the BASE architecture. The BASE architecture already demonstrates two RMS characteristics, modularity and customisability, but the other characteristics of RMSs like scalability and integrability still need to be integrated. To integrate the above-mentioned characteristics, finer details of the BASE architecture like communication between BASE shells, service-discovery, cyber-physical interfacing, user interfacing and diagnostics need to be addressed.

Stellenbosch University https://scholar.sun.ac.za

With regards to research on holonic systems, this thesis aims to address two important functions for systems implemented using holonic principles, namely cooperation and cyber-physical interfacing. The thesis shows how service orientation can enable cooperation between holons, while keeping these holons independent. It also shows that extracting all cyber-physical interfacing of a system into dedicated *Interface Holons* (as discussed in chapter 7) improves development time, reduces system complexity and facilitates better diagnosability of interfacing problems.

1.4 Methodology

To guide the extension of this thesis, research into CPSs was required to understand their most important characteristics. To address reconfigurability and the use of holonic principles, research on RMSs and holonic systems were done. This research was used to formulate the characteristics of complex, reconfigurable CPSs. Subsequently, the BASE architecture was evaluated against these characteristics to formulate a set of required extensions to the architecture.

A top-down approach was used to design the extended BASE architecture, where the first step was to create the infrastructure of the architecture's platform and identify the platform management components. Thereafter, the BASE shell was refined by adding components to it and making any required changes to its existing components. Most of the development time had to be spent on two of the most critical and complex functions of the architecture – cooperation between holons and interfacing between the real and the cyber world. Several standard plugins were developed to address these two functionalities.

Before considering any of the case study specifics, the majority of the development was already performed, since the case study only affected the custom plugins that had to be developed. A case study was selected based on selection criteria that would ensure all of the extensions made in the architecture could be implemented and demonstrated.

After the case study was implemented and deployed, an evaluation of the extensions made to the BASE architecture was performed. For this evaluation, quantitative and qualitative metrics were set up using the requirements of a complex, reconfigurable CPS as evaluation criteria. The evaluation results had to be analysed to determine what they indicated about the extensions, and how much these results were influenced by other factors like the programming language or the author's programming experience. A conclusion is made based on the results from the evaluation that highlights the most important findings, as well as any future work that is still required in the BASE architecture.

Stellenbosch University https://scholar.sun.ac.za

2 Literature review

This chapter presents a review of literature relevant to the thesis. The first three sections cover literature about CPS, RMSs and holonic systems. Hereafter the BASE architecture of Sparrow (2021) is dissected, which served as a starting point for the rest of the thesis.

2.1 Cyber-Physical Systems

The following section starts of by reviewing the most common definitions of CPSs. Thereafter, the section focusses on the characteristics of CPS, which was very important to set up the requirements of a complex, reconfigurable CPS in chapter three. Lastly, the most common applications of CPSs are listed and an example of a CPS architecture to be used in the agriculture industry is discussed.

2.1.1 Definition

Definitions of CPSs are still being changed year by year, as researchers gain new knowledge. In 2013, Kim & Kumar (2013) stated that CPSs require a combination of computing, communication and control. The following year, Gunes, Peter, Givargis & Vahid (2014) integrated the contributions of many articles to define CPSs as systems with embedded computing technology, which are integrated with multi-disciplinary physical components to observe and control these components. Shortly hereafter, Lee (2015) generalized a CPS by stating that it is simply an "orchestration of computers and physical systems".

In 2016, Monostori *et al.* (2016) defined CPSs with slightly different terminology than has been used up to then. They stated that CPSs are systems with collaborating computational entities which have an "intensive connection" with their physical world of interest and its operations, while also using "data-accessing and data-processing services available on the Internet". Two years later, Ribeiro & Bjorkman (2018) defined CPSs as the new generator of embedded systems with advanced artificial intelligence and improved communication capabilities.

One of the newest well cited and more elaborated definitions of CPSs was given by Chen, Wang, Feng, Li & Liu (2019). They stated that in a CPS "the physical system acts as a data access role with sensors and communication systems to collect real-world information and communicate to computation modules, which further analyse and notify the findings to the corresponding physical systems through multiple feedback loops".

2.1.2 Characteristics

Figure 2 shows the technological and operations management characteristics of CPSs according to Napoleone *et al.* (2020) based on 151 CPS articles that they

Stellenbosch University https://scholar.sun.ac.za

considered. A total of 113 characteristics were identified, from which the 19 most cited characteristics (each with at least ten references) were chosen. Napoleone *et al.* (2020) compressed these 19 characteristics into eight lower-level characteristics. Furthermore, they added a predictability as a ninth lower-level characteristic that they felt was important, even though it was not mentioned in the literature that they considered.

These nine lower-level characteristics ("C1" to "C9" in figure 2) were grouped into five higher-level characteristics, namely integration, cooperation, intelligence, reconfigurability and adaptability, and predictability. These five higher-level characteristics (dashed boxes in figure 2) were further split into technological characteristics and operations management characteristics.

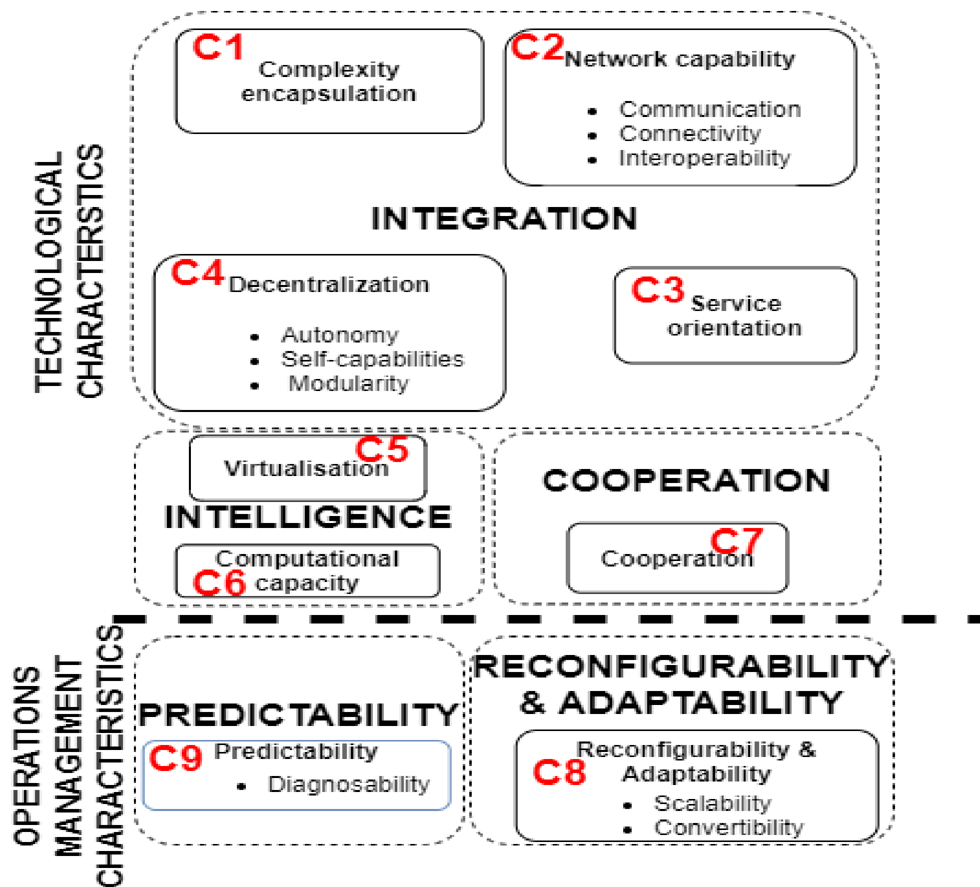


Figure 2: A comprehensive schema of characteristics of CPSs (adapted from Napoleone *et al.* (2020))

Each of the sections from section 2.1.2.1 to 2.1.2.5 discuss one of the higher-level characteristics of CPSs. In each section this is done by looking at the details of the lower-level characteristics that make up the higher-level characteristic being discussed. These sections contain references from some of the most cited CPS papers together with references from Napoleone *et al.* (2020).

Stellenbosch University https://scholar.sun.ac.za

2.1.2.1 Integration

CPSs can be considered as efficiently integrated physical and digital components. The following section will discuss complexity encapsulation, networking capability, service orientation and decentralisation, the first four characteristics of a CPS, which are all required to enable a CPS's integration abilities (the first higher-level characteristic).

CPSs need to integrate several different systems, each with their own communication protocols. The heterogenous nature of different CPSs, integrated into one overarching CPS, is one of the biggest contributors to complexity in these systems (Wan *et al.*, 2013). Thus, Napoleone *et al.* (2020) regard complexity encapsulation as a general design principle of CPSs, and lists it as the first characteristic.

Napoleone *et al.* (2020) state that CPSs are composed of interconnected clusters of computational and physical elements, which are connected through wired and wireless networks. Therefore, they identify network capability, as the second characteristic of CPSs, which includes a CPSs ability to communicate, connect and operate with other CPSs.

The interoperability of a CPSs components needs to be standardized to ensure these components use the same communication protocols, otherwise they would not be able to communicate (Mourtzis & Vlachou, 2018). This standardisation also reduces the costs of manufacturing systems by lessening any customized integrations required for the different components (Ramis Ferrer, Iarovyi, Lobov, Martinez Lastra, & Mohammed, 2020).

CPSs promise real-time data acquisition and information feedback from the digital realm, which requires communication between the digital and physical realms. Communication in CPSs often occurs over the internet and thus IoT is an important enabler of CPSs (Tedeschi *et al.*, 2018).

In summary of the previous three paragraphs, the networking capability of CPSs requires them to be interoperable, connected and able to communicate (Ali *et al.*, 2015), Many papers use the term "network connectivity" to describe CPSs network capability (Jabeur, Sahli & Zeadally, 2015; Khorrami, Krishnamurthy & Karri, 2016).

The third characteristic of CPSs, according to Napoleone *et al.* (2020), is service orientation. In a Service-oriented Architecture (SoA), architectural components provide services to each other over a shared network (Russo, 2021). In CPSs, the principles of SoAs can be used to reduce the complexity of integrating the different components (Iarovyi *et al.*, 2016). The digital services provided by CPS components can be encapsulated and accessed by other CPS components (Tao & Qi, 2019).

Stellenbosch University https://scholar.sun.ac.za

Modularity, autonomy and self-capabilities are all encapsulated under decentralisation, the fourth characteristic of a CPS according to Napoleone *et al.* (2020). Svetlík (2020) defines a modular system as a system with a flexible set of unified modules which function together to form a higher functional and more complex unit. Heiss *et al.* (2015) show that modularity allows systems to be independent, which make the systems more flexible to react to customer requirements or products that are changed.

CPSs must have a high level of autonomy due to their complex nature (Ribeiro & Bjorkman, 2018). This autonomy is enabled by the ability of CPSs to initiate actions based on the aggregated data received from the physical components in the system (Pirvu, Zamfirescu & Gorecky, 2016). Self-capabilities and autonomy are related terms and Wan, Yan, Suo and Li (2011) show that self-capabilities of components are enabled by the autonomy of these components. Many papers use different terms that refer to these self-capabilities, e.g. self-adaptivity (Chen, Wan, Shu, Li, Mukherjee & Yin, 2018), self-awareness (Lee, Jin & Bagheri, 2017) and self-learning (Scholze, Barata & Stokic, 2017). CPSs are decentralized because their components function independently while working towards a common goal (Ghobakhloo, 2018).

The combination of the lower-level characteristics discussed in this section (complexity encapsulation, network capabilities, service orientation and decentralisation) enable CPSs integration capabilities. Integration, in the context of CPSs, allows new components to be added to existing ones much easier and in a much shorter time frame (Penas, Plateaux, Patalano & Hammadi, 2017).

2.1.2.2 Intelligence

Napoleone *et al.* (2020) state that a CPSs virtualisation and computational capabilities (the fifth and sixth characteristics of CPSs) enable its intelligence, the second higher-level characteristic. Virtualisation is the use of virtual replicas of the physical parts of CPSs to remotely track physical processes (Yuan, Anumba & Parfitt, 2015) and simulate their behaviours (Babiceanu & Seker, 2016). Cimino, Negri and Fumagalli (2019) show that these virtualisations can be extended from just observing and simulating CPS processes, to taking real-time action on the physical resources, based on the simulation results in the virtual world.

Sanderson, Chaplin & Ratchev (2018) refer to virtualization as creating digital twins of CPSs. They also state that accurate virtualisation requires a CPS to have real-time capabilities. Lee, Ryu & Cho (2017) define real-time capability as the capability of a CPS to provide information immediately after new data has been acquired. This allows CPSs to react on changes in the physical realm to increase fault-tolerance and safety in these complex systems (Carreras Guzman, Wied, Kozine & Lundteigen, 2019).

The sixth characteristic of CPSs, computational capability, refers to the data management and analytics carried out by the cyber parts of CPSs (Ghobakhloo,

Stellenbosch University https://scholar.sun.ac.za

2018). These cyber parts take over the tedious computations and actions which previously had to be done manually by humans (Zhou, Zhou, Wang & Zang, 2019). Gai, Qiu, Zhao & Sun (2018) show the advantages of using cloud computing to enable these computational capabilities. In the cloud, web-services can be utilised, and distributed physical resources can connect with each other without needing their cyber parts to run on the same computer (as shown by Caggiano (2018)).

Intelligence of a CPS is measured by how capable the CPS is to sense changes, make decisions and automatically interface with other CPSs (Cheng, Liu, Qiang & Liu, 2016). The intelligence of a higher-level CPS can be distributed among the lower-level CPSs (Cardin, 2019). Intelligence of a CPS relies strongly on its ability to collect information from the physical components in (or close to) real-time and the computational power that is available. Therefore, intelligence is the second higher-level technological characteristic of CPSs, enabled by CPSs' virtualisation and computational capabilities.

2.1.2.3 Cooperation

Napoleone *et al.* (2020) state that cooperation forms both the seventh lower-level characteristic and the third higher-level characteristic of CPSs. Cooperation allows distributed CPSs to autonomously choose which of their components should be used for a specific activity (Etxeberria-Agiriano, Calvo, Noguero & Zulueta, 2012). Tran, Park, Nguyen & Hoang (2019) show that self-organization in a CPS is realised by combining autonomy and cooperation.

2.1.2.4 Reconfigurability and adaptability

According to Napoleone *et al.* (2020), reconfigurability and adaptability can be seen as equivalent and form both the eighth lower-level characteristic and fourth higher-level characteristic of CPSs. Adaptability can be realised by ensuring individual components are independent and do not need to rely on each other to deal with changes in their environment (Ribeiro & Bjorkman, 2018). Adaptability in a CPS requires the CPS to react to frequent changes in requirements by dynamically reconfiguring its components (Otto, Vogel-Heuser & Niggemann, 2018). Therefore, for a system to be adaptable it must be dynamically reconfigurable.

Napoleone *et al.* (2020) added scalability and convertibility as driving characteristics of reconfigurability and adaptability. Scalability refers to a CPSs ability to add or remove participating resources during the CPSs lifecycle (Heiss *et al.*, 2015). Iarovyi *et al.* (2016) describe convertibility as the capability of a CPS to extend their functionalities in a modular way to support new requirements.

2.1.2.5 Predictability

Napoleone *et al.* (2020) added predictability as both the ninth CPS characteristic and the fifth higher-level CPS characteristic and grouped it under CPSs' operations management characteristics. As depicted in figure 2, predictability is enabled by

Stellenbosch University https://scholar.sun.ac.za

diagnosability, which requires that the system's errors and status are transparent. Predictability is required by smart factories to strengthen their production and logistics adaptivity (Facchinetti & Della Vedova, 2011) and to implement predictive maintenance (Shcherbakov, Glotov & Cheremisinov, 2019). Prediction of equipment failures can be used to trigger autonomous maintenance activities.

2.1.3 Applications

Chen (2017) studied 77 CPS publications from 2012 to 2017 in the Scopus database to identify CPS applications. The ten biggest application domains found and reviewed were agriculture, education, energy management, environmental monitoring, intelligent transportation, medical devices and systems, process control, security, smart city and smart home, and smart manufacturing.

One example application of CPSs in agriculture is precision farming. Antonopoulos, Panagiotou, Antonopoulos (2019) proposed a commercial grade CPS platform, called A-FARM, to meet the requirements of multifaceted agriculture cultivation deployments. A-FARM is divided horizontally into an edge and a cloud layer. The edge layer contains sensors and gateways. The cloud layer is split vertically into four layers, namely (from bottom to top): persistence, aggregation, services and web layer. Data is received in the aggregation layer from the edge layer and then stored in the persistence layer (below the aggregation layer), which contains a cluster of databases. The service layer (above the aggregation layer) acts as the entire architecture's orchestrator, while the web layer (above the service layer) contains all the end-user Application Programming Interfaces (APIs). This architecture has not yet been deployed but its authors state that it promises minimisation of water wastage and chemical fertilizer usage.

2.2 Reconfigurable Manufacturing Systems

Koren *et al.* (1999) introduced RMSs to address the unpredictable and fast changes in the market that manufacturing companies have to deal with. RMSs can easily switch production, of a specific product family, by adding and/or removing hardware or software components (Martinsen, Haga, Dransfeld & Watterwald, 2007).

Koren *et al.* (2018) defined six core characteristics of RMSs, as follows:

- Scalability: the ability to modify production capacity by adding or removing resources or changing system components.
- Convertibility: the capability of an existing system to change its functionality to meet new production requirements.
- Diagnosability: an RMS must be designed so that product quality and the reasons for product failures can be diagnosed very quickly.

Stellenbosch University https://scholar.sun.ac.za

- Customisation: the ability to customize a part family.
- Modularity: breaking up operational functions into units that can be altered between different production schemes.
- Integrability: the ability to integrate different modules rapidly and precisely through their hardware and software interfaces.

RMSs promise enhanced system responsiveness to fluctuating markets, improving manufacturing companies' competitiveness. However, real-time operational decision-making in RMSs has not yet been researched properly, most likely because of the complexity of RMSs. Traditional analytical and decision-making algorithms are unable to efficiently deal with this complexity. New intelligent manufacturing techniques like multi-agent systems, cloud manufacturing and CPSs can help address this problem More research on combining RMSs with these techniques would be very valuable. (Koren *et al.*, 2018)

2.3 Holonic systems

2.3.1 Background, definitions and principles

Koestler (1967) proposed the term *holon* to describe an entity or system with self-organizing abilities. He described a holon as being a whole and a part simultaneously. A holon contains an information processing part combined with a physical processing part. In essence, a holon is an entity with communication and decision-making abilities, which may be composed of a set of sub-level holons, but could also form part of a wider organization composed of higher-level holons (Koestler, 1967).

When holons are grouped together they form a holonic system, also referred to as a *holarchy*. Holonic systems are robust when exposed to disturbances and can adapt to changes in the environment (Vyatkin, 2007). Derigent *et al.* (2020) used Giret & Botti (2004) to identify three core principles of holonic systems, namely: decision making, autonomy and cooperation. Each of these principles are discussed in more detail in the following paragraphs.

Each holon needs to be able to make decisions by reducing the number of possibilities and choosing the optimal solution. This decision-making process can be reactive, in response to some stimulus, or pro-active, where the system or an individual holon has a goal towards which it is working. This decision-making process can also be recursive, because of the recursiveness of holarchies in these types of systems. For example, a holon that is made up of many other lower-level holons, might need one/more of its lower-level holons to handle some parts of the decision-making process. (Derigent *et al.* 2020)

Stellenbosch University https://scholar.sun.ac.za

Autonomy, the second principle of holonic systems, stems from the definition of a holon as an independent entity with self-organising capabilities (Koestler, 1967). Giret & Botti (2004) state that autonomy is the degree of freedom with which each holon can make decisions. Each holon's autonomy is defined during the design of the holon, but can be altered by higher-level holons during operation. (Derigent *et al.*, 2020)

The third principle of holonic systems, cooperation, both restricts and expands the autonomy of holons (Derigent *et al.*, 2020). Service-orientated communication, like the Contract-Net Protocol (CNP) of Smith (1980) – discussed in appendix C - is a popular method of enabling cooperation in holonic systems. This can be seen in Service-oriented Holonic Manufacturing Systems (SoHMSs), which allow for flexible and reactive systems (Gamboa *et al.*, 2016).

2.3.2 Performance metrics of Holonic Control Architectures

The informational parts of holonic systems are developed using HCAs (Derigent *et al.*, 2020). Kruger & Basson (2018) proposed evaluation criteria for the implementation of HCAs in manufacturing systems. Their methodology was to first identify the characteristics of HCAs, derive requirements from these characteristics and, finally, formulate quantitative and qualitative performance metrics for HCAs. Figure 3 shows the requirements and performance metrics identified, as well as the dependencies of the requirements on the different performance measures. The rest of this section will only discuss the performance measures and not the requirements.

The six quantitative performance measures identified by Kruger & Basson (2018) are reconfiguration time, development time, code complexity, code extension rate, code re-use rate and computational resource requirements. Reconfiguration time is the time a developer takes to change an existing system (which includes physical and software changes) to integrate a new holon, and development time is the time it takes to develop the new holon's control software. Development and reconfiguration time can both be measured by adding a new type of holon to an existing holonic system and timing the development and the integration separately for the two respective metrics. When measuring the development time, it is crucial that the planning time needed by the developer to understand the problem and the systems involved, is not measured.

Code complexity can be measured using the source lines of code (SLOC) measurement as shown by Cesarini, Pappalardo & Santoro (2008). The code extension rate is the amount of "growth" needed to reconfigure the source code for some reconfiguration and can be calculated by dividing the final configuration's SLOC value by the initial configuration's SLOC value. Reusability is crucial for high productivity and the code reuse rate can be calculated as the initial configuration's SLOC value divided by the new configuration's SLOC value.

Stellenbosch University https://scholar.sun.ac.za

The computational requirements of a holonic control implementation show to what extent its functionality can be supported by resource-limited controllers. To measure an HCA's computational requirements, the Central Processing Unit (CPU) usage and memory usage can be measured on a computer running the HCA implemented system while a production activity is active.

The qualitative measures all aim to analyse the system characteristics to estimate the system's performance for each of these measures. For example, modularity considers the architectural considerations that enable modular implementation and mechanisms to verify the performance of individual modules.

			Characteristics						
			Availability Supportability Development Productivity						
			Requirements						
			Reconfigurability	Robustness	Maintainability	Controller requirements	Complexity	Verification	Reusability
Performance measures	Quantitative	Reconfiguration time	*				*	*	*
		Development time					*	*	*
		Code complexity			*		*		
		Code extension rate	*		*		*		
		Code re-use rate	*		*		*		*
		Computational resource requirements				*			
	Qualitative	Modularity	*		*			*	*
		Integrability	*						*
		Diagnosability	*	*	*			*	
		Convertibility	*		*				
		Fault tolerance		*					
		Distributability				*			
		Developer training requirements			*		*	*	

Figure 3: Requirements and performance metrics for holonic control implementations (Kruger & Basson, 2018)

2.4 The BASE architecture

The development of the BASE architecture was driven by the need to integrate human workers into I4.0 environments. This development was strongly influenced by the requirements and responsibilities of Resource Holons in Holonic Manufacturing Systems (HMSs). Sparrow, Kruger & Basson (2021) proposed the

Stellenbosch University https://scholar.sun.ac.za

BASE shell, a Human Resource Holon Administration Shell (HRH-AS), to raise humans to a CPS level, which would enable them to interact with other CPSs. This BASE shell adds storage, processing, and communication abilities to the human's abilities by making use of Human-Machine Interfaces and Information and Communication Technology. The BASE architecture was created as the reference architecture for this BASE shell, and the details of the BASE architecture and its implementation are discussed in this section. In the context of other popular HCAs like PROSA (Van Brussel, Wyns, Valckenaers, Bogaerts & Peeters, 1998), ADACOR (Leitão & Restivo, 2006) and ARTI (Valckenaers, 2019), the BASE architecture serves as an implementation architecture of the resource (PROSA and ARTI) or operation (ADACOR) holons in these architectures.

2.4.1 Core components and plugins

The BASE architecture consists of five core components and four types of plugins, as shown in figure 4. The BASE architecture's five core components are the Schedule, Execution, Biography, Attributes and Communication Manager. Schedule stores all activities that must still be executed, Execution stores all activities that are being executed and Biography stores all activities that have been completed.

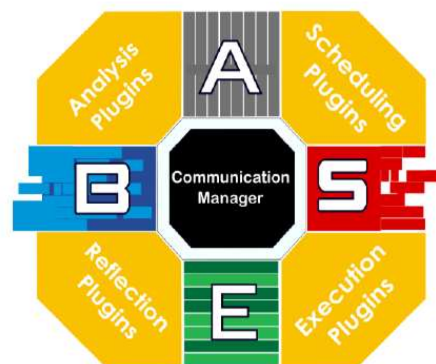


Figure 4: The BASE architecture (Sparrow, 2021)

In addition to its storage component, Execution consists of a State Blackboard (SBB), an Observer and an Informer. The SBB stores the human's current state (physical, mental and biological) and is updated by the Observer. The Observer gathers information about the human from observation services, like sensors and cameras, and delivers the information in Value-Confidence-Timestamp format. The Informer delivers information to the human and uses the Attributes to personalise and optimise this information delivery.

The Attributes component stores a holon's attributes, i.e. the properties of the holon that do not change during activities. The various BASE shells' Communication Managers enable inter-holon communications. The five core components are not case specific and function the same in all BASE shells; however, these components

Stellenbosch University https://scholar.sun.ac.za

do not add any value on their own. Sparrow (2021) created two categories of attributes, namely Personal Attributes and Contextual Attributes. Personal Attributes are persistent data about a resource and its BASE shell and forms the digital model of the resource. Contextual Attributes are application specific and defines the resource within the context it is in.

To add case specific value to a BASE shell, Sparrow (2021) created BASE plugins which can interact with the BASE core components. There are four types of BASE plugins, namely: Scheduling Plugins (SPs), Execution Plugins (EPs), Reflection Plugins (RPs) and Analysis Plugins (APs). A BASE shell's SPs schedule new activities in its Schedule using smart algorithms, consideration of attributes, and interactions with other holons. Scheduled activities are initiated, monitored and driven by EPs, which will update the activities' execution progress in the Execution. When activities are completed, RPs are used to save the activity data in the Biography and add any post-execution data to the activities. APs analyse activity data stored in the Biography in order to update the BASE shell's attributes in the Attributes component.

2.4.2 The three-stage activity life cycle

The BASE architecture uses the Three-Stage Activity Lifecycle (3SAL) model, proposed by Sparrow, Kruger & Basson (2020), for the structuring of its activity data. In the 3SAL model, an activity progresses through three stages: *scheduled*, *in execution* or *completed*. The activity data is structured according to the stage at which the data was created, as shown in figure 5. Schedule data is edited before an activity is started, execution data is edited while an activity is being executed and post-execution data is added after an activity has been completed.

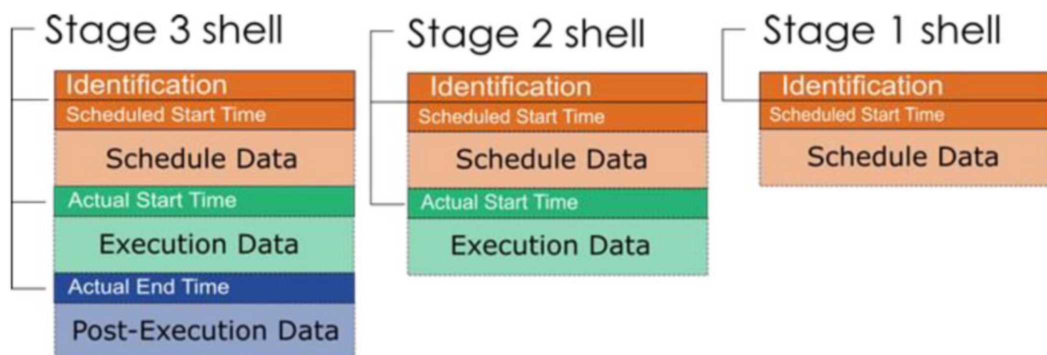


Figure 5: Shells and data blocks of stage 1, stage 2 and stage 3 activities in the 3SAL model (Sparrow *et al.*, 2020)

2.4.3 An Erlang implementation of the BASE architecture

Sparrow (2021) implemented the BASE architecture in Erlang – a highly concurrent programming language. Erlang was selected because the language offers

Stellenbosch University https://scholar.sun.ac.za

robustness, high concurrency and scalability, and the functionality to change the executing code during operation (i.e. without having to stop and restart the application). These characteristics are considered critical when implementing complex and reconfigurable software systems.

The Schedule, Execution, Biography and Attributes components all had the structure shown in figure 6, except that the Attributes component had two storage components, one for personal attributes and one for contextual attributes. Sparrow (2021) used Erlang's Erlang Term Storage (ETS) tables as data repositories, and these are explained in more detail in section 8.1.

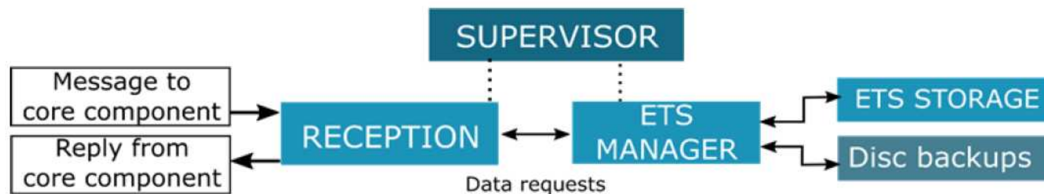


Figure 6: Generalised structure of a BASE core component data repository implemented in Erlang (Sparrow, 2021)

Except for the Execution processes proposed by the BASE architecture, namely the SBB, Observer and Informer, two other processes were added in the implementation, namely the Activity Handler and the Execution Bench. The Activity Handler acted as a gateway between EPs and Execution, and was used to give the next instructions and observe outcomes of activities. Completed activities were moved to the Execution Bench where they awaited further processing from RPs.

2.4.4 Application of the BASE architecture in industry

Sparrow (2021) managed to evaluate the BASE architecture by implementing it in an aerospace composites manufacturing company to address some of the companies' labour-intensive requirements. This case study showed how the BASE shell could be used as a HRH-AS which added value to this composites company by doing real-time automated schedule management, calculating standard work and improving traceability of operations.

An activity that occurs frequently in the aerospace composites company, namely a layup activity, was used for the case study. In this activity a worker fetches a ply pack and mould, unpacks the ply pack and then sequentially places each ply in the mould using plastic spatulas and a heat gun. Core filler, a hardening polymer substance, has to be applied between some plies, using a template showing which areas have to be filled. After all the plies are laid up, the part is placed in a vacuum bag and taken to a vacuuming station, completing the layup activity.

Stellenbosch University https://scholar.sun.ac.za

Figure 7 shows a time sequence visualisation of the duration of the different steps of the layup activity when the BASE shell was used to manage the activity's execution by interfacing with the relevant worker. This data was used to convince the aerospace composites company to deploy BASE shells on their shop floor.

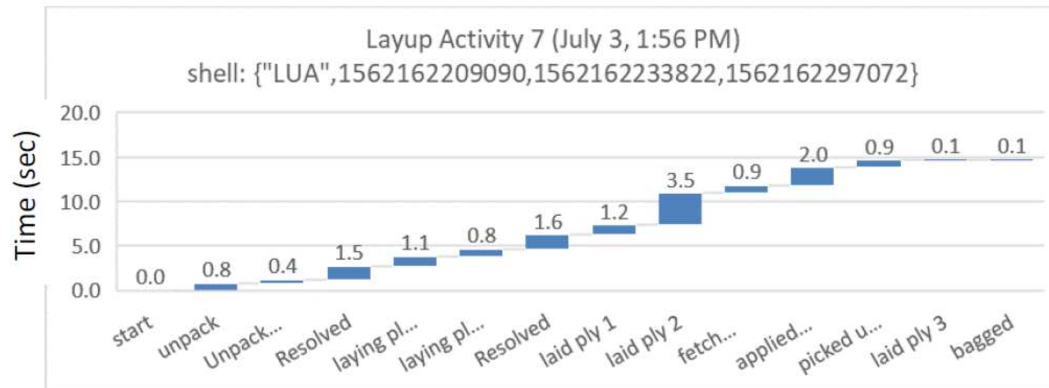


Figure 7: Action time visualisation from captured execution data during the layup activity

2.5 Conclusion

CPSs can be applied in many different industries and promises more accurate reflection of physical systems in the digital realm, as well as greater control over these systems from the digital realm. Reconfigurability within CPSs, can be more effectively realised by integrating the characteristics of RMSs with that of CPSs. Furthermore, the core principles of holonic systems, namely: decision making, autonomy and cooperation can be used to improve decentralisation, cooperation and intelligence within CPSs.

The BASE architecture shows much potential to be extended to more than just the integration of humans into the I4.0 environment. Most of the core components in this architecture are applicable to any type of resource, making it very appropriate for complex CPSs. The BASE architecture's use of plugins greatly enhances its customisability, an important characteristic of reconfigurable CPSs. In conclusion, the BASE architecture shows great potential to be used as a reference architecture for complex, reconfigurable CPSs.

Stellenbosch University https://scholar.sun.ac.za

3 Requirements for the extension of the BASE architecture

This chapter starts by identifying the requirements of a complex, reconfigurable CPS, based on literature. Furthermore, it evaluates the original BASE architecture against these requirements to identify any shortcomings. This evaluation is used to create a list of extensions needed to use the BASE architecture for complex, reconfigurable CPSs.

3.1 Requirements of a complex, reconfigurable CPS

This section determines the requirements of a complex, reconfigurable CPS by transforming the nine characteristics of CPSs, as proposed by Napoleone *et al.* (2020), into requirements. Of the nine CPS characteristics only **decentralisation** is directly used as a requirement. *Complexity encapsulation* and *reconfigurability*, are expanded into more extensive requirements, since these are the two most important CPS characteristics addressed in this thesis. The ten requirements identified in this section can be seen in table 1.

Network capability refers to both the ability of a CPS's cyber and physical components to connect and communicate, and their ability to integrate and interact with other systems outside of their overarching CPS (Napoleone *et al.*, 2020). To clearly distinguish these two abilities within *network capability*, they are split as separate requirements, namely: **cyber-physical interfacing** and **integration**.

Service oriented interactions is a cooperation strategy and thus *service orientation* and *cooperation* are grouped into the same requirement, namely **service-oriented cooperation**. *Virtualisation* is transformed into a requirement by being less specific, and rather using the term **usability**, to refer to a CPS's provision to let users view and reconfigure its details. *Computational ability* cannot be used as a requirement, but will greatly assist in evaluating the scalability of a CPS, since a CPS's scalability can be limited by the computational resources available to it. *Predictability*, although proposed by Napoleone *et al.* (2020) as a characteristic of a CPS, should not be a requirement of a CPS, but rather a potential benefit.

This thesis focusses especially on *complexity encapsulation* and *reconfigurability* as CPS characteristics that the BASE architecture must enable. To encapsulate the complexity of integrating many diverse entities, a CPS must generalise as many functionalities as possible, which is why **generalisation** is added as a requirement. The integration of diverse, autonomous entities can be very prone to errors. Subsequently, the generic parts of a CPS must be robust, while providing resilience for when custom parts of the system fail. This robustness and resilience are encapsulated under the same requirement, namely **reliability**.

Stellenbosch University https://scholar.sun.ac.za

Most research about what *reconfigurability* entails can be found in the field of RMSs. Koren *et al.* (2018) defined six core characteristics of RMSs. *Modularity* is encapsulated within decentralisation and *integrability* has already been mentioned above as a CPS requirement. In the manufacturing domain, *customisation* and *convertibility* refer to different characteristics, but to facilitate many domains these can be seen as the same and the term **customisability** is used as another requirement. The remaining two RMS characteristics, **scalability** and **diagnosability**, and very relevant to reconfigurable CPSs, and are therefore also added as requirements.

3.2 Evaluating the original BASE architecture

Table 1 evaluates the BASE architecture against the requirements of a complex, reconfigurable CPS. These evaluations are focussed on identifying all shortcomings of the BASE architecture as a reference architecture for complex reconfigurable CPSs. Colours are used in the table to indicate which requirements are met fully (green), partially (yellow), or not at all (white).

Stellenbosch University https://scholar.sun.ac.za

Table 1: Evaluation of how well the BASE architecture enables the requirements of a complex, reconfigurable CPS

Requirements	Evaluation of the BASE architecture
Cyber-physical interfacing	The original BASE architecture already uses observer and informer services to enable cyber-physical interfacing. These services need to be integrated as holons and generalised as far as possible to reduce complexity and improve diagnosability.
Integration	External systems can be integrated using plugins.
Service-oriented cooperation	Sparrow (2021) already mentioned service orientation in the BASE architecture, but this still needs to be refined. The services of the BASE architecture's holons need to be standardised and some form of service directory is needed. The Communication Manager already provides the foundation needed for interactions between holons, but standardised service-oriented interaction protocols, encapsulated in plugins, still need to be developed.
Decentralisation	Decentralisation within a single holon is already enabled by the structuring of data (core components) and functionalities (plugins) within each BASE shell. However, decentralisation within an entire CPS with many BASE shells still needs to be addressed.
Scalability	The BASE architecture in its current state can only accommodate a single BASE shell and needs to be extended to facilitate scaling.
Customisability	The use of plugins and editable attributes in each BASE shell meets the requirement of being able to reconfigure a holon's internals.
Diagnosability	This requirement is not addressed yet.
Generalisation	The management of storage and communication functions is handled by the core components. Cyber-physical interfacing and cooperation between holons requires many generic functionalities to reduce complexity.
Reliability (robustness and resilience)	The structure and philosophies of the original BASE architecture already provide a foundation that will make the creation of a robust CPS easier, but these implementations have not yet been addressed. Custom components are prone to errors and thus resilience must also be addressed.
Usability	Interfacing with users of the system still needs to be addressed.

Stellenbosch University https://scholar.sun.ac.za

3.3 Extensions required to the BASE architecture

Table 1 helped to identify where development is still needed in the BASE architecture to use it as a reference architecture for complex, reconfigurable CPSs. Many of these developments are dependent on each other and can be combined into the same extensions/refinements. A matrix showing the requirements, extensions and their relationships is shown in figure 8, and should be used to aid the understanding of the rest of this section.

		Extension												
	THESIS CHAPTER	*4*				*5*						*6*		*7*
		Provision for multiple BASE shells	Supervision	Diagnostic tools	User-interface	Shell privacy	Shell coordination	Refined plugin-shell interfacing	Standardised activity & attribute types	Prevention of data loss	Custom component failure handling	Service directory	Standardised service-oriented interactions	Cyber-physical interfacing holons
Requirement addressed by an extension	Cyber-physical interfacing													x
	Service-oriented cooperation											x	x	
	Decentralisation					x						x		
	Scalability	x			x		x					x		x
	Diagnosability			x	x								x	x
	Generalisation							x	x				x	x
	Reliability		x			x	x			x	x			
	Usability				x									

Figure 8: Relationship matrix showing relationships between the requirements of a complex, reconfigurable CPS and the required BASE architecture extensions

The following list contains the extensions required to the BASE architecture. How each extension is related to the various requirements is also explained in this list.

- **Provision for multiple BASE shells:** The architecture needs to facilitate more than one BASE shell in the same system, enabling scalability.

- **Supervision:** In order to add reliability to the system, the different components need to be supervised, by dedicated supervision processes, so that the failure of these components are detected and isolated, and the components are restarted.

Stellenbosch University https://scholar.sun.ac.za

- **Diagnostic tools:** The architecture requires components dedicated to logging information that can help diagnose problems in the CPS.

- **User Interface:** In order to make the BASE architecture more usable, the architecture needs to make provision for a User Interface (UI) that can enable users to view and reconfigure a BASE architecture implementation. This UI should also enhance diagnostics and allow users to scale the system up and down.

- **Shell privacy:** A very important principle of decentralised architectures are that the different components should be independent. Thus, each BASE shell's internal components must not allow any other components to directly communicate with them. Communications between BASE shells can only happen via their Communication Managers. This also makes the architecture more reliable, since errors are isolated and each holon's data can never be corrupted by other holons' BASE shell components.

- **Shell coordination:** This extension mainly addresses the need for scalability, because when there is more than one BASE shell in the system, each BASE shell's components need to know about each other (be coordinated), otherwise intra-shell communication would not be possible. Each component of a BASE shell might fail, in which case the newly restarted component and the rest of the BASE shell needs to be re-coordinated. Thus, BASE shell coordination is essential to the architecture's reliability as well.

- **Refined plugin-shell interaction:** Refined interactions between plugins and their BASE shells are required to make plugin development more convenient. Extracting the generic functionalities required in plugins and integrating these functionalities into the core components of the BASE architecture also addresses the generalisation requirement.

- **Standardised activity and attribute types:** The integration of different types of resources is destined to result in many different types of activities and attributes, and each developer might use their own terminology to describe the same types of attributes or activities. Therefore, standardised activity and attribute types are needed to encapsulate the most common activities and attributes found in holons' BASE shells.

- **Prevention of data loss:** One of the main functions of BASE shells is to store activity data, which is why BASE shells' core components should be refined to not lose this data. If data gets lost occasionally due to unplanned hardware or network failures, the system would not be reliable.

- **Custom component failure handling:** Any custom component is very prone to errors that could cause failures, which is why it is crucial that the

Stellenbosch University https://scholar.sun.ac.za

core components can handle these custom component failures. This resilience is required to improve the system's reliability.

- **Service directory:** Without a service directory, holons would not be able to find and utilise each other and decentralisation would not be possible. The service directory also allows the system to be scaled up and down with very little reconfigurations, because new holons' services are simply added to the service directory and used whenever another holon requires them.

- **Standardised service-oriented interactions:** In order to reduce complexity, all interactions between holons' BASE shells need to follow some standardised protocol so that all developers build the same communication protocols into the plugins they develop. This extension would also aid the architecture's diagnosability, because there is a predefined list of message types that could be sent between holons.

- **Cyber-physical interfacing holons:** Cyber-physical interfacing services need to be integrated into the BASE platform as holons which provide cyber-physical interfacing services. These holons need to be generalised as far as possible to increase development productivity. Having all cyber-physical interfacing encapsulated in dedicated holons would also improve the scalability of the system, since it would enable system operators to simply add and remove interfaces (as holons) with minimal reconfiguration in other holons. All cyber-physical interfacing problems would also be logged within the respective Interface Holons' Biographies, improving the diagnostics of cyber-physical interfacing errors.

Stellenbosch University https://scholar.sun.ac.za

4 Platform management components

The following chapter introduces the management components required in platforms on which the BASE shells will be deployed. These components are not essential for the BASE shells to operate, but greatly improves the robustness, scalability, cooperation, diagnosability, usability and transparency of the system.

4.1 Overview

Figure 9 shows the platform infrastructure of the extended BASE architecture, which includes the BASE shells together with the platform management components of the platform on which these BASE shells are implemented. The four platform management components illustrated in this figure are the:

- Supervision tree
- Department Of Holon Affairs (DOHA)
- Loggers
- User Interfaces (UIs)

These components are explained in more detail in the rest of this chapter.

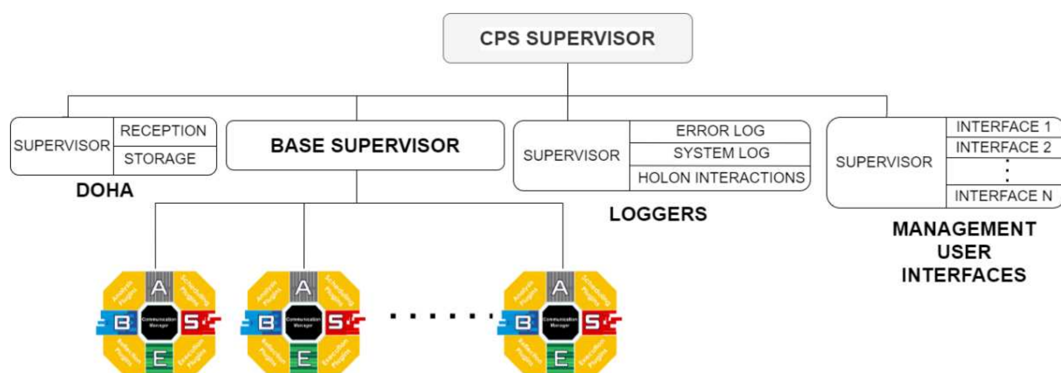


Figure 9: Overview of the extended BASE architecture

4.2 Supervision tree

Software supervision is a complicated topic with little research reported on it. Researchers also have different views of what a software supervisor is, for example Savor & Seviora (1995) defines a software supervisor as a unit that monitors the inputs and outputs of a real-time system to report unexpected behaviours. The developers of Erlang (introduced in section 2.4.4) have a slightly different view of

Stellenbosch University https://scholar.sun.ac.za

software supervisors. In Erlang, supervisors are processes that do nothing else except monitor (and possibly restart) other processes, which might be worker processes (processes doing something other than supervision) or other supervisors (Armstrong, 2003). These supervisors' purpose is to restart the processes that they monitor if one/more of these processes fail. This view of supervisors is adopted for the rest of this thesis.

Sparrow's (2021) use of supervisors in his Erlang implementation of the BASE architecture showed how much robustness supervisors bring to the system. For this reason, supervision is an important functionality that should be integrated in both the platform and the BASE shells running on this platform. Figure 9 and figure 11 (in chapter 5) both show how supervisors must be implemented in the BASE architecture. This hierarchy of supervisors, referred to as a supervision tree, facilitates the start-up, shutdown and failure recovery operations of a BASE architecture implemented CPS.

Each supervisor in the hierarchy monitors its own supervisor/worker processes and when one of them fail, it tries to restart them. When a supervisor cannot restart one of its components or the component keeps on failing, the supervisor terminates all of the components underneath it and then terminates itself. The supervisor that is monitoring this failed supervisor will then attempt to restart the failed supervisor and if it fails, the supervisor above it will continue the recovery attempt until the system can be recovered or fails completely.

When the system is started, the CPS Supervisor will be the only component added to the system and it has the responsibility of starting the four other supervisors underneath it for DOHA, the BASE shells, the loggers and the UIs. Each of these supervisors have their own components that they need to start and monitor. The BASE Supervisor starts and monitors the existing BASE holon's Shell Supervisors, and each Shell Supervisor starts and monitors its internal supervisors which start and monitor their components (figure 11). The shut-down operation happens in the exact opposite order than that of the start-up operation, thus from the bottom up.

4.3 Department of Holon Affairs

DOHA acts as the BASE architecture's service directory (or yellow pages directory) facilitator. Even though Sparrow (2021) did not write about DOHA, he came up with the idea and developed the first code for it. The author of this thesis took the first Erlang code of DOHA to further develop DOHA's architecture. When a holon can provide one or more services, its BASE shell must register these services with DOHA. This enables other holons to search for a service in DOHA and DOHA will give them a list of all holons that can provide the requested service.

A secondary purpose of DOHA, added in this thesis, is to provide a convenient way to know which holons were in the system before it shut down. After a system restart, the list of holons found in DOHA are restarted. DOHA does, however, not help to

Stellenbosch University https://scholar.sun.ac.za

configure the system in its last known state, since the holarchy will self-assemble, based on the relational data stored in the various holons storages.

4.4 Loggers

In order to diagnose a CPS implemented with the BASE architecture, three classes of data or events need to be logged, namely: errors, system information and holon interactions. Errors could include mistakes in developers' code, networking errors or hardware failures. System information could include metrics like CPU, Random Access Memory (RAM) usage and storage usage, and the time it takes to complete certain actions, e.g. the start-up time of the system after power failures or the time it takes to add a new holon in the system. Holon interactions are communications between holons, which in the BASE architecture is service-oriented.

The error, system and holon interactions loggers are added to the BASE architecture's platform management components to improve the platform's diagnosability. The error logger can help platform managers diagnose faults in the platform or the hardware it is using, while the system logger can be used to track the load on the system and to measure its performance in different scenarios.

The service-oriented interactions between holons, recorded by the holon interactions logger aims to help developers diagnose when and why certain interactions between holons (like a cancelled contract) did or did not occur. The functionality of the holon interactions logger is similar to that of the Sniffer agent provided by the Java Agent Development framework (Bellifemine, Caire & Greenwood, 2007). Figure 10 shows the holon interactions logger developed as part of this thesis' generic implementations. In this logger, sequence diagrams are used to show the type of messages exchanged between two or more holons.

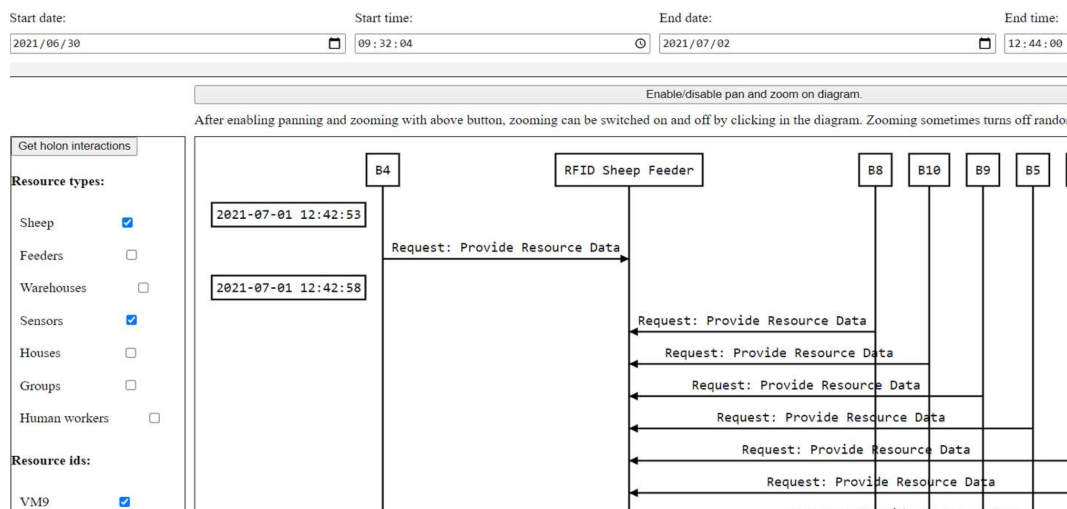


Figure 10: Holon interactions shown on a UI

Stellenbosch University https://scholar.sun.ac.za

4.5 User Interfaces

Users of a BASE architecture implemented CPS typically need to add, remove or change BASE shells, schedule new activities, monitor activity execution and perform diagnostics. Higher-level stakeholders will also require the CPS to be transparent to aid their decision making and planning. Thus, UIs are required as platform components that can provide the following functionalities:

- Display the active holons based on some holon search criteria e.g. holon type/services: This functionality requires that users of the system be able to specify some search criteria and the interface's back-end then be able to find all holons in the system that satisfy this criteria and return this to the interface's front-end. The front-end should then display these holons in an easy-to-understand way so that users can clearly identify the different holons and possibly see other high-level information about each holon like their most important attributes.

- Allow the users to see more detail about each holon: This function entails that users of the system should be able to look through the Attributes, Schedule, Execution (including the SBB), and Biography of each holon. Holons will have many different types of activities and attributes and to make the finding of specific activities or attributes more convenient, users should be able to filter these using search criteria like activity/attribute types or categories.

- Allow users to scale the system: This functionality should allow users to add and remove holons in the existing system, while it is running. When a user wants to add a new holon, the front-end should accept input parameters describing the new holon, like its ID, type, plugins etc. and communicate this to the interface's back-end. In the back-end the supervisor of the new holon's BASE shell should be added under the BASE Supervisor. The new BASE shell's supervisor will initiate the creation of the rest of the BASE shell. To remove holons, the functionality of finding existing holons in the system should be extended so that users can select one or more of the existing holons and indicate that they should be removed.

- Allow the managers or users to reconfigure existing holons: The users should be able to add and remove plugins in existing holons' BASE shells, as well as edit these holons' attributes and scheduled activities through an intuitive interface.

Stellenbosch University https://scholar.sun.ac.za

5 The refined BASE architecture adminstration shell

This chapter gives an overview of the refined BASE shell. This includes introducing new core components, explaining the changes made to the existing core components and showing how independence between BASE shells were realised. The refinements made to plugin-shell interfacing and categories used to group similar activities and attributes in BASE shells are also discussed.

5.1 Overview

Figure 11 shows the structure of a single BASE shell. At the top of the tree structure is a Shell Supervisor, which supervises a Coordinator (discussed in section 5.2) and six other supervisors: a supervisor for each of the core components and one supervisor for the plugins.

The Communication Manager only has a reception process, unlike the four data repository components, which each have a reception process and at least one storage process. The Schedule component also has a Schedule Gate (discussed in section 5.3) and the Execution component has a SBB and a list of Activity Handlers (discussed in section 5.4). Since the core components of the BASE shell were already discussed in section 2.4, this chapter will not reintroduce these components, but rather discuss the changes that were made to some of the components and introduce the Coordinator and Schedule Gate.

Figure 11: Details of the refined BASE architecture digital administration shell

Stellenbosch University https://scholar.sun.ac.za

5.2 Coordinator

The Coordinator was added to the BASE shell's core components, mainly to address the resilience, and in effect the reliability, required in BASE shells. Core components and plugins in a BASE shell are restarted by their supervisors when they fail (as explained in section 4.2). When a component (core component or plugin) is restarted, the Coordinator shares the addresses of its BASE shell's existing components with the restarted component. It also informs existing components about the updated address of the restarted component. To enable this, all core components and plugins register themselves with their BASE shell's Coordinator after being initialised. This allows the Coordinator to keep the updated addresses of its BASE shell's core components and plugins. To facilitate the registrations of a BASE shell's core components and plugins, the Coordinator always needs to be the first component started up underneath its BASE shell's supervisor. When a BASE shell's Coordinator fails, the entire BASE shell must be restarted, which is why the Coordinator must be as robust as possible.

Coordinators also enhance decentralisation in the BASE architecture in two ways. Firstly, each Coordinator provides the coordination needed to split each BASE shell into different core components and plugins, which can still communicate with each other, even after some of these get restarted. Secondly, each Coordinator ensures their BASE shell is independent from other BASE shells. Coordinators enable this by providing their BASE shell's core components and plugins with each other's addresses. This gives a BASE shell's core components and plugins the ability to reject any communication from components that are not part of their BASE shell, since they have a list of addresses of their BASE shell's components. As recommended by Sparrow (2021), the Communication Managers are the only components of BASE shells that communicate with components outside of their BASE shell.

The two main responsibilities of the Coordinator can be summarised as follows:

- Sharing the addresses of its BASE shell's core components and plugins with each other after their initialisation.
- Sharing the address of a restarted core component or plugin (that was restarted by a supervisor because of failure) with the rest of its BASE shell's core components and plugins so that they do not try to communicate with the terminated process, but with the restarted process.

Appendix A explains the Coordinator's start-up procedure and highlights the most important variables that all Coordinators must keep in their state.

Stellenbosch University https://scholar.sun.ac.za

5.3 Schedule Gate

The Schedule Gate was added to the Schedule component to inform EPs when activities, for which they are registered for, are due to start. Consequently, EPs do not have to continually check through the Schedule's storage for activities that need to be started. The Schedule Reception informs the Schedule Gate about the schedule times of newly scheduled activities. The Schedule Gate uses the list of schedule times to determine when it needs to look in the Schedule for activities are due to start, instead of constantly checking through the Schedule.

The Schedule Reception also shares the addresses of EPs that *register* for activity types (this registration is discussed in section 5.6.1) with the Schedule Gate. When the Schedule Gate then finds an activity that must be started, it can use the activity's type to determine which EP must be informed.

5.4 Changes to the existing BASE core components

This section discusses the four most notable changes to the core components of the BASE architecture. The first of these changes is that in the extended BASE architecture, activity's data never have to leave the data repositories of their BASE shell's core components. Plugins only create activities (SPs), start activities (EPs), edit the stage 2 data in active activities (EPs), finish activities (EPs) and add stage 3 data to activities (RPs). The progression of an activity from one data repository component to the next (Schedule to Execution or Execution to Biography) is handled by the core components. For example, when an activity needs to be completed, the activity does not need to be removed from Execution by an RP, to be moved into the Biography. The Execution will will share this activity with the Biography and only when the Biography has successfully received the activity will the Execution remove it from its own storage.

The next change is in the Activity Handlers, which have been refined and generalised for all types of resources, not only humans. Activity Handlers are the gateways between EPs and the Execution component and each activity in the Execution component must have an Activity Handler. The details of how Activity Handlers are used are shown in figures 13, 14 and 15 in section 5.6.1. The functionalities of each Activity Handler can be summed up as follows:

- It stores the most up to date data about an activity in its state so that it can quickly share this data with the relevant EP without needing to go into the Execution component's storage.

- It receives updates about an activity's stage 2 data from EPs and uses this to update the activity in the Execution component's storage.

Stellenbosch University https://scholar.sun.ac.za

- When an EP indicates that it is done executing an activity, the activity's Activity Handler informs the Execution component that an activity is completed. The Execution component then sends the activity to the Biography and removes it from its own storage.

Another notable change is in the Execution component, where the Observer, Informer and Execution Bench components have been removed. Observation of a holon's data and getting information to a physical part of a holon, is discussed in chapter 7. The Execution Bench is considered redundant, since the responsibility of moving activities from Execution to Biography is no longer that of RPs (as explained above).

The last notable change to the core components is in the Communication Manager. Sparrow (2021) stated that it is the responsibility of the Communication Manager to consolidate different communication protocols, but it has been decided to let the Communication Manager only dictate the high-level service-oriented interactions (discussed in chapter 6) and not the protocols or ontologies of the messages. These are handled in the plugins; otherwise the Communication Manager would require frequent adaptation to support new protocols or ontologies.

5.5 BASE shell component privacy

All holons in the CPS need to be independent from each other. This is facilitated by prohibiting communication between the internal component processes of different holons' BASE shells – only allowing communication between BASE shells to occur through the respective Communication Manager components.

The core component processes permitted to communicate with each other within a BASE shell can also be restricted, as shown in table 2. These permissions can be enforced by letting each process store, in its state, the addresses of the component processes that are permitted to communicate with it. The Coordinator and Communication Manager Reception of a BASE shell can receive messages from any core component process or plugin that is part of their BASE shell.

The Schedule Reception, Execution Reception, Biography Reception and Attributes Reception processes allow all plugins to communicate with them, however it is recommended that plugins only communicate with the reception processes of the data repository components to which they are adjacent in figure 4 (e.g. EPs should preferably only communicate with Schedule and Execution reception processes).

The storage processes of the four data repository core components can only receive messages from their core components' reception processes, e.g. the Biography Storage can only receive messages from the Biography Reception. This ensures only one point of entry into the different storage processes, which simplifies the management of storage permissions and the tracking of data changes.

Stellenbosch University https://scholar.sun.ac.za

Table 2: Intra-shell communication permissions

Base shell core component process	Other components in the same BASE shell as this component that are permitted to message/call it
Coordinator	All core component processes and plugins
Communication Manager Reception	All core component processes and plugins
Schedule Reception	All plugins and Schedule Gate
Schedule Gate	Schedule Reception
Execution Reception	Schedule Reception, Activity Handlers and all plugins
Activity Handler	Execution Reception and the single EP that started the activity
Biography Reception	Execution Reception and all plugins
Attributes Reception	All plugins
Storage	The reception process of this component (e.g. the Biography Storage can only receive messages from Biography Reception)

5.6 Plugin configuration

Plugins provide the case specific functionality required in holons' BASE shells. The following section discusses how plugins interact with their BASE shells' core components. It also briefly discusses the concept of plugin configuration files.

5.6.1 Plugin-shell interfacing

In the original BASE architecture, plugins had to poll the receptions of their BASE shells' core components if they wanted to know if there are new activities or changed attributes. This made plugin development slightly more complex, and the plugin processes were also less efficient. As such, functionality is added in all core component receptions to allow plugins to register themselves for all activities or attributes of a certain type. Each core component reception of a BASE shell informs a registered plugin if there are new activities or changed attributes of the type that the plugin registered for.

Figure 12 shows how the different plugins can register for different activity/attribute types. In the figure, two attribute types are indicated as AT1 and

Stellenbosch University https://scholar.sun.ac.za

AT2 and three activity types as T1, T2 and T3. The Attributes and Biography Receptions allow more than one plugin to register for the same attribute/activity type; however, the Schedule and Execution Receptions only allow one plugin per activity type. The reason for permitting only one EP per activity type is because only one EP is allowed to start an activity and update its stage 2 data in Execution. This prevents clashes between EPs over the control of an activity's execution. The reason for permitting only one RP per activity type is to prevent two RPs from overriding each other's changes to an activity's stage 3 data. APs can register for the same activity type, because the activities in Biography are not affected by APs analysis. SPs can register for the same attribute types because Attributes are not affected by SPs that create new activities.

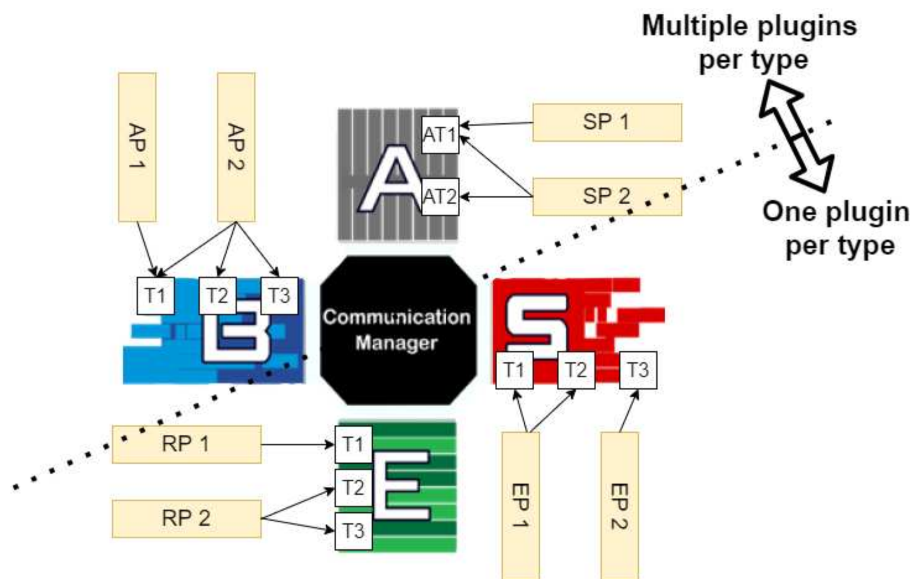


Figure 12: Plugin registrations

Within a BASE shell, plugins must interact with their core components to:

- move activities through their three different stages;
- update attributes based on completed activities' data; and
- analyse changed attributes to potentially schedule other activities.

To support the explanation of these interactions, an example of a robot with a drill tool is considered. The robot in this example can drill more than one part at a time and needs maintenance after a certain period of operation. The interactions between the plugins and core components that are described in this example are, however, applicable to any holon's BASE shell.

Figure 13 indicates the interactions between the core components and an EP in the drill robot's BASE shell, required to start a scheduled activity. Note that figure 13a

Stellenbosch University https://scholar.sun.ac.za

and 13b represent the same BASE shell. In this scenario, Drill EP is registered for activities of type "Drill". The Schedule Gate informs Drill EP about "Act1123 (Drill)" that is due to start (arrow 1, figure 13a). The Schedule Gate does not start the activity, because EPs have the responsibility of starting an activity (this gives plugins control over the timing of activity execution). Drill EP messages Schedule (arrow 2, figure 13a) to start the activity. Schedule sends "Act1123 (Drill)" to Execution and asks it to start an Activity Handler for it (arrow 3, figure 13a). After storing the activity (reference box 4, figure 13b) and starting an Activity Handler for it, Execution replies to Schedule with the Activity Handler's address (arrow 5, figure 13b). Schedule removes the activity in its storage (reference box 6, figure 13b) and sends Drill EP the Activity Handler's address (arrow 7, figure 13b). Drill EP requests the activity data from the Activity Handler (arrow 8, fogire 13b) and starts the execution of the activity.

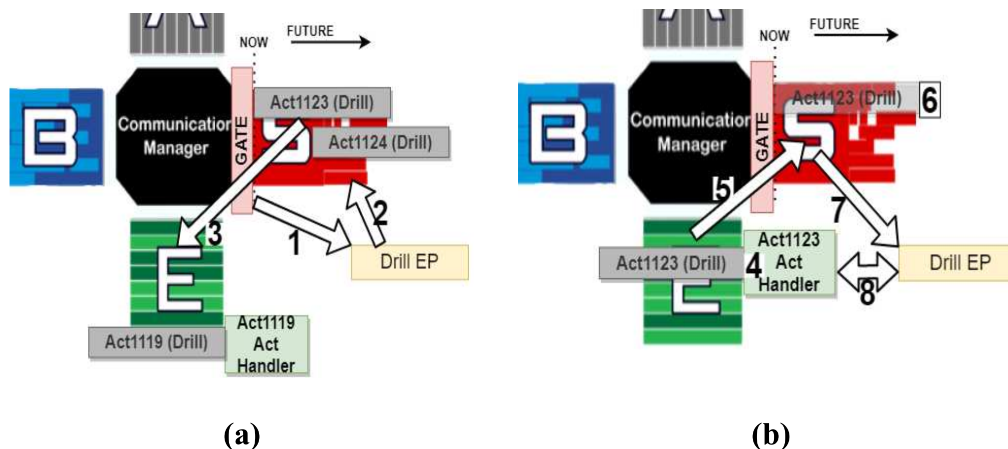


Figure 13: Schedule Gate informing an EP to start an activity (a) and an activity handler being created after the EP started the activity (b)

Figure 14 shows the interactions by which the Drill EP updates the stage 2 data of "Act1123 (Drill)", completes the activity's execution and adds post-execution data to it via an RP. Drill EP sends new stage 2 data to the Activity Handler of "Act1123 (Drill)" (arrow 1, figure 14a) and this Activity Handler updates the activity in its own state and in Execution's storage (arrow 2, figure 14a). Drill EP does this every time it wants to update execution information about the activity that should not get lost if the system were to shut down while the activity was in execution. When Drill EP completes the execution of the activity, it informs the activity's Activity Handler (arrow 3, figure 14a) and the Activity Handler forwards this message to Execution (arrow 4, figure 14a). The Activity Handler subsequently terminates.

Execution sends the activity to the Biography (arrow 5, figure 14a), which, after storing the activity (reference 6, figure 14b), replies to Execution (arrow 7, figure 14b) that it has stored the activity. Before removing the activity from its storage, Execution shares the activity with Drill RP (arrow 8, figure 14b), because Drill RP is registered for activities of type "Drill". Drill RP reflects on the activity (finds post-execution data related to it, e.g. quality control) and updates the activity's stage

Stellenbosch University https://scholar.sun.ac.za

3 data in Biography (arrow 9, figure 14b). This reflection (arrow 9, figure 14b) can happen any time in the future, and does not have to be directly after the activity has been shared with the RP. Note that the activity would still have been in the Biography if Drill RP did not exist, and that Drill RP only updates the existing activity's stage 3 data.

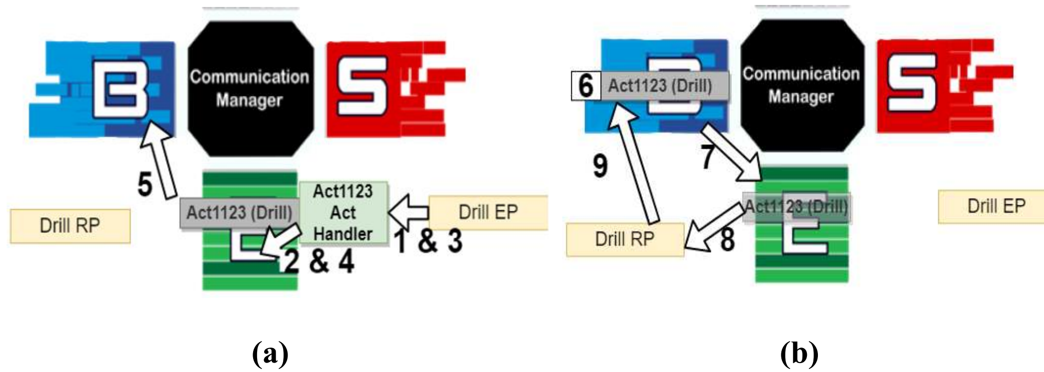


Figure 14: Updating stage 2 data and finishing an activity (a), and adding post-execution data to it (b)

In figure 15, Biography informs Drill AP about the completed "Drill" activity (arrow 1), since Drill AP is registered for activities of type "Drill". Drill AP updates the attribute named "Drill time since last maintenance", by calculating the drilling time from the "Act1123 (Drill)" activity data and adding it to the attribute's current value (arrow 2). The Attributes component informs Drill SP about the updated attribute (arrow 3), as Drill SP is registered for attributes of type "Drill". Drill SP executes an algorithm that determines that this holon's physical part needs maintenance and adds a maintenance activity in Schedule (arrow 4).

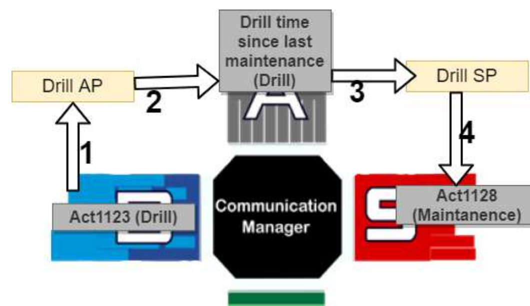


Figure 15: Updating attributes and scheduling new activities

5.6.2 Plugin configuration file

The plugin configuration file is used by a BASE shell's Plugin Supervisor to start up a BASE shell's plugins. The file must contain the list of plugins to start, where each entry in the list states the file/module name (and the directory, if necessary) of the plugin and the plugin's start-up arguments. Start-up arguments can be used

Stellenbosch University https://scholar.sun.ac.za

when one plugin is used for many holons, but the plugin's functionalities can be slightly reconfigured by changing arguments sent to the plugin when it is started up. Each BASE shell must have a plugin configuration file and one file can be shared by many BASE shells. Figure 16 shows a snippet of a plugin configuration file used for all sheep holons in the case study.

```
{
"1":{"module":"resource_data_sp", "args": []},
"2":{"module":"resource_data_ep", "args": []},
"3":{"module":"sheep_aggregation_sp", "args": []},
"4":{"module":"sheep_aggregation_ep", "args": []},
"5":{"module":"sheep_relocation_sp", "args": []},
"6":{"module":"sheep_relocation_ap", "args": []},
"7":{"module":"sheep_eating_rp", "args": []},
"8":{"module":"sheep_fcr_ap", "args": []}
}
```

Figure 16: Plugin configuration file for all sheep holons

5.7 Activity types

Activity types are used to group activities of the same type, for example "Drill", "Grind", "Transport", etc. Multiple activities can have the same type, as long as their identifiers are different. Activity identifiers (activity IDs) are what make activities unique within a holon. The responsibility of ensuring activities have unique IDs has been assigned to the Schedule. When a SP wants to create an activity in the Schedule it only needs to specify the activity type, and the Schedule will ensure that the activity has a unique ID.

There can be an infinite number of activity types within each holon and, to reduce complexity, these are grouped into a finite list of categories. The following three categories have been identified:

- Service Provision Activity (SPA)
- Resource Action Activity (RAA)
- Resource Data Acquisition Activity (RDAA)

Each of these activity categories are discussed in more detail in sections 5.7.1 and 5.7.2 and the summaries of their stage 1 and stage 2 data structures are shown in appendix B.

5.7.1 Service Provision Activities and Resource Action Activities

Any service that a holon provides is always encapsulated in an SPA. SPAs have a very specific data structure in their stage 1 and stage 2 data as explained in section 6.3. RAAs are activities used to address actions required within a resource, and the

Stellenbosch University https://scholar.sun.ac.za

most common examples of RAAs are maintenance and repair activities. An example from the case study implementation, discussed in section 9.5.2, was the "Relocate (RAA)" RAA, which was used when a sheep had to be relocated to another camp. In most cases, RAAs will require other holons' services to execute the required action, in which case their stage 1 data structure contains the *"Service Contract"* and their stage 2 data structure is the same as that of SPAs. These stage 1 and stage 2 data structures of RAAs are summarised in appendix B.

Figure 17 shows how RAAs and SPAs differ. In this figure the service requester requires maintenance, which is why it has "Maintenance (RAA)" on its schedule. Its Maintenance EP starts this RAA (arrow 1) and requests a maintenance service from an appropriate service provider (arrow 2). The service provider accepts the service request and puts "Provide maintenance (SPA)" on its schedule (arrow 3). This SPA contains all the information required for one of the service provider's EPs to drive this service provision (arrow 4). The service requester's RAA will remain in its Execution component until a message is received from the service provider, indicating it has finished the maintenance, at which point the service provider will also finish its SPA (moving it into the Biography). Sometimes RAAs only consist of a resource doing something internally, in which case no service provider is required.

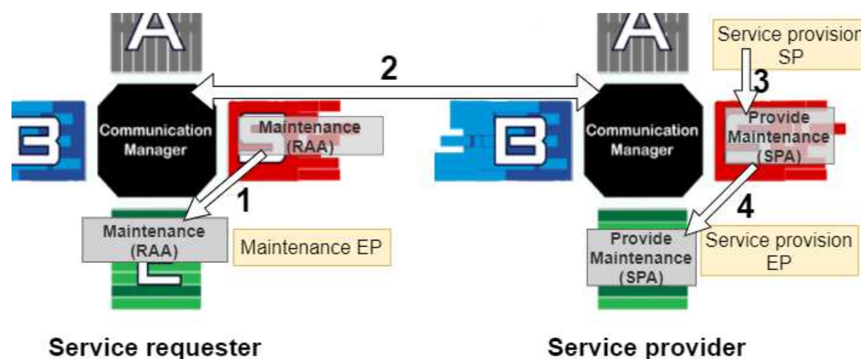


Figure 17: RAA vs SPA

5.7.2 Resource Data Acquisition Activities

RDAAs are used to receive and log observed data about a holon from various Observer holons (as described in section 7.4.2). An RDAA is scheduled in a holon's BASE shell for each Observer holon in the system that can provide data to this holon. RDAAs have a specific stage 1 and stage 2 data structure, which can be seen in appendix B. The stage 1 data has only one field, namely the *"Service Contract"* (discussed in section 6.3) with the Observer holon. When new data is received from an Observer, this data is added to the stage 2 data of the RDAA and the RDAA is moved into the Biography, but it is not removed from Execution. This means the same RDAA is used to continuously receive observed data from an Observer, without having to schedule and start a new RDAA every time data has been

Stellenbosch University https://scholar.sun.ac.za

received. Figure 18 shows an example of how RDAAs (in this case for observed temperature data) are saved to the Biography as new data is received from an Observer, while not removing the original RDAA from Execution.

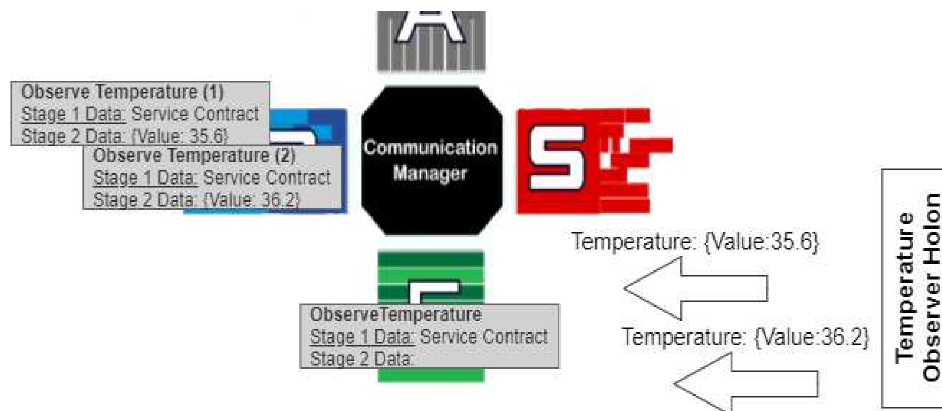


Figure 18: RDAA used to observe temperature data recorded by a temperature observer holon

To ensure previous RDAA entries are not overwritten, the activity ID of the RDAA being moved to the Biography is changed slightly for each new entry into the Biography, as shown in figure 18. When the Observer holon, for which some RDAA was created, is removed from the system or it stops its service provision, the RDAA is finished and the reason for finishing it is also added to its stage 2 data before moving it into the Biography.

5.8 Attribute types

Sparrow (2021) split Attributes into two types, namely *Personal* and *Contextual*. This thesis further divided the Contextual type into three specific types: *Relational*, *Management* and *Condition* Attributes. Relational Attributes represent a holon's relations with other holons in the system and have no value outside of the system. Management Attributes are used to dictate certain behaviours of resources and the values of these attributes are set by users via the UIs. For example, a drilling machine's operating speed or a human's maximum continuous hours of work allowed, can be Management Attributes. Management Attributes are very dependent on the user requirements and are controlled by the users of the system.

Condition Attributes are similar to State Variables, in the sense that they represent the state of the holon. State Variables contain the current state information about a holon (like its last recorded location or temperature) and are updated by EPs, while Condition Attributes can contain anything related to a holon's past state and are updated by APs. The most common State Attributes are statistics derived from RDAAs, SPAs and RAAs, e.g. in this thesis' case study, each sheep's average, maximum and minimum amount eaten per day was part of its Condition Attributes.

Stellenbosch University https://scholar.sun.ac.za

6 Cooperation

One of the most important, but also most complex, characteristics that the BASE architecture should enable is cooperation between holons. This cooperation can be challenging to manage because of the different types of holons integrated into a BASE architecture implemented CPS. A common approach to facilitate this cooperation is by using service-orientated interactions, which will allow each holon to remain independent yet be available for use by other holons. This chapter presents an overview of service-orientated interactions within the extended BASE architecture. An explanation of the architecture's Service Descriptions and *service discovery* – the way in which services are advertised and utilised – is given. Thereafter, the data structures of SPAs (introduced in section 5.7.1) are described. Furthermore, two standard plugins which enable standardised service-oriented interactions between BASE shells are introduced and lastly a brief overview of nested services is given.

6.1 Business Cards and Service Descriptions

Each holon has a Business Card (BC) which is stored in DOHA and is used to advertise the holon's services to other holons. A holon's BC represents who they are (Resource ID), what they are (Resource Type), where they are (BASE shell Address) and what they can do (Services). The Services field contains a hash table (data structure with key-value pairs) with the keys and values being Service Types and Service Descriptions, respectively. When a service requester needs a service provider, it provides the required Service Type to DOHA and DOHA will return all holons that can provide the requested Service Type. The service requester can then filter through all the potential service providers using their Service Descriptions. The Service Description contains more details about a holon's service.

Figure 19 shows an example of two holons' services. Holon A has two Service Types, namely "Drill" and "Mill", while holon B only has a "Drill" Service Type. In figure 19 it can be seen that the Service Descriptions of the two holons' "Drill" services have the same keys ("Maximum RPM" and "Cost (R) per minute"), but different values for each key. This is because in a BASE architecture implemented CPS, it is required that the Service Descriptions of services of the same Service Type, must contain the same data structure. The values used to populate these data structures can differ. This allows a service requester to objectively compare two holons' services (of the same Service Type) to aid the service requester's service filtering algorithms used to filter through service providers before sending out Call For Proposals (CFPs).

Stellenbosch University https://scholar.sun.ac.za

Holon A	Holon B

{
"Drill":{
"Maximum RPM":600,
"Cost (R) per minute":7
},
"Mill":{
"Cost (R) per minute":5
}
}
	{
	"Drill":{
	"Maximum RPM":1500,
	"Cost (R) per minute":7
	}
	}

Figure 19: Data structures of two holons' services, each with a "Drill" Service Type

6.2 Service discovery

Having a service directory is one of the fundamental requirements of any SoA. DOHA was introduced in section 4.3 as the BASE architecture's service directory. DOHA contains a database that stores the BCs of all the holons in the CPS. When holons are created, removed or their details change, DOHA is informed so that it can update this database and inform other holons about the changes. When a holon needs to find other holons, it can search for them in DOHA by their name, type or by one of the holon's Service Types.

Holons can also register for specific service types so that they are informed by DOHA when new service providers become available – as is shown in figure 20. It is important to note that it is one of the plugins of the holon's BASE shell that does this registration through its BASE shell's Communication Manager (arrows 1 and 2). When a new holon's BC is added to DOHA (arrow 4), DOHA informs all holons that are registered for the new holon's Service Types, about the new holon (arrow 5). The registered holons' Communication Managers forward this message to their plugins that initiated the registrations (arrow 6).

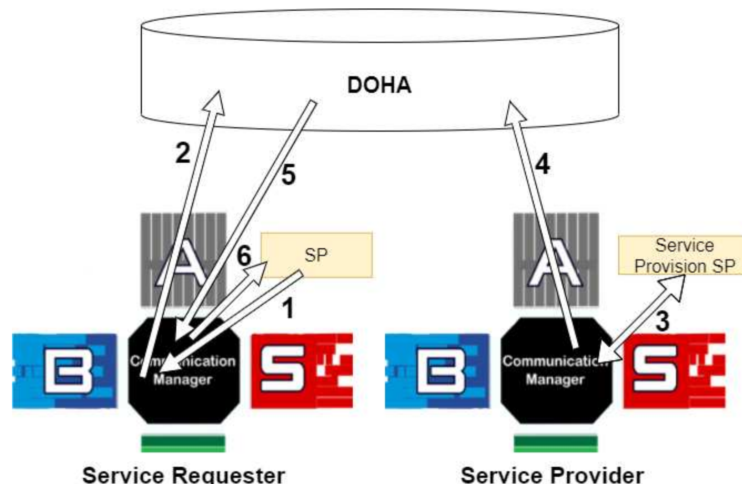


Figure 20: Service discovery

Stellenbosch University https://scholar.sun.ac.za

All holons' Communication Managers are the same and thus services cannot be programmed into the Communication Manager. Each holon's Communication Manager gets its holon's services from the Service Provision SP (arrow 3), adds this to the holon's BC and then registers the holon with DOHA (arrow 4). Service provision SPs are discussed in section 6.4.

6.3 Service Provision Activity data structures

SPAs were introduced in section 5.7.1 as activities that encapsulate a service being provided by a holon. The stage 1 data of SPAs contain the *"Service Contract"*, and the stage 2 data contain the *"Pending Proposals"*, *"Proposals"*, *"Sub-Contracts"* and *"Results"*. A visual representation of SPAs' data structures can be found in appendix B, to aid the explanations of the data fields in the following paragraphs.

"Service Contracts" are used by service requesters and providers to ensure both parties are aware of, and agree to, the details of the service being provided. These contracts contain the "*requester BC", "provider BC", "delivery address", "service type", "request details"* and "*inform template"*. The "*requester BC"* allows the service provider to know to which holon it is providing a service and the "*provider BC"* allows the requester to know which holon is providing the service. These two BCs enable the requester and provider holon to communicate with each other during the execution of the service. The "*delivery address"* is used by the requester's Communication Manager to know to which of its plugins an inform message should be delivered and is populated the first time that a plugin sends CFPs (discussed in section 6.4.1) to acquire a new service. The *"service type"* simply contains the Service Type of the service being provided.

The "*request details"* are constructed by the requester to specify any details about the service it requires and are used by the service provider to fulfil the service requirements of the requester. The "*request details"* will differ for each Service Type, but all service providers with the same Service Type should be able to process the same "*request details"*. In effect, a requester does not need to change the "*request details"* for every unique service provider and only needs to know what details must be provided for every Service Type. The "*inform template"* specifies the format in which the service requester is expecting *inform messages* from the service provider. This allows the service provider to reject the service request if this template does not match the format and structure that it can deliver its *inform messages* in.

"Pending Proposals", "Proposals" and "*Sub-Contracts"* will only be created if a service provider needs other service providers for some service it is providing – the concept of nested services is explained in section 6.5. "*Pending Proposals"* contain the details of CFPs that have been sent out, but for which the proposals are still pending. "*Proposals"* contain the details of the proposals that have been received from service providers. "*Sub-Contracts"* contain the "*Service Contracts"* of the accepted proposals. Note that for SPAs, "*Pending Proposals", "Proposals"* and

Stellenbosch University https://scholar.sun.ac.za

"*Sub-Contracts*" are all part of the stage 2 data of the service provider, not the service requester. The value of the *"Result"* field can be either *"Pending"*, *"Success"*, *{"Failed", Reason}* or *{"Cancelled", Reason}*, where "*Reason*" contains the details of the failure or cancellation. A service cancellation occurs when the holon that requested the service, cancels it before it is completed. The fields explained in this paragraph are also applicable to the stage 2 data of RAAs.

6.4 Plugins for standardised service interactions

The BASE architecture integrates the CNP of Smith (1980) for all of its service-oriented interactions between holons. The CNP allows service requesters (auctioneers) to negotiate with service providers (bidders) in order to select the best service provider and utilise the services they promised. Appendix C gives a more in-depth overview of the CNP, in the context of holonic systems. This section will describe how the BASE architecture's service provision plugins interact with their BASE shell's core components to cooperate with other holons.

6.4.1 Call For Proposals and service requests

Figure 21 shows the various intra- and inter-holon interactions that occur during the first phase of all service-oriented interactions. When a holon's plugin (in this figure an EP, but it can be a plugin from any of the four categories) needs a service of some Service Type, the plugin first acquires from DOHA all the available service providers that can provide this Service Type. The plugin then requests its Communication Manager (arrow 1) to send a CFP to all the potential service providers (arrow 2). All the service providers' Communication Managers forward the request to their Service Provision SP (arrow 3).

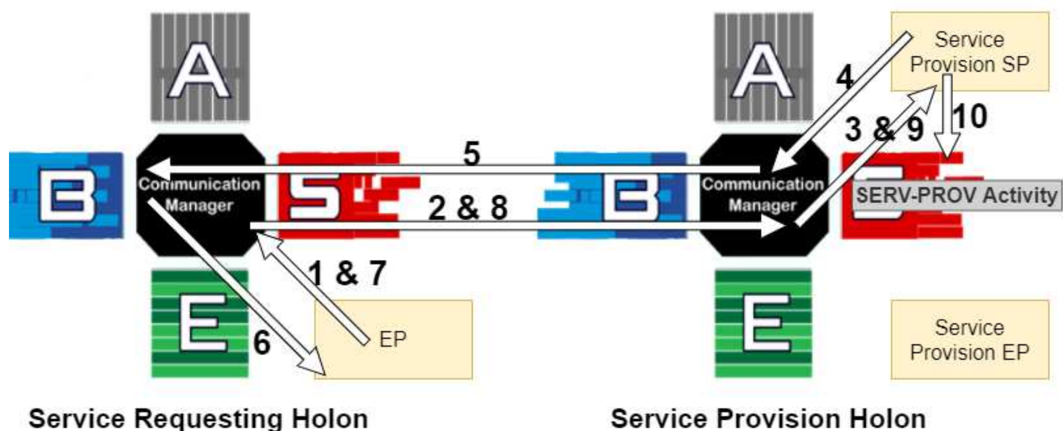


Figure 21: Inter- and intra-holon interactions for selecting a service provision holon and requesting its services

Each Service Provision SP decides between replying with a proposal (arrows 4 and 5) or refusing to make any proposal. All the proposals received by the service

Stellenbosch University https://scholar.sun.ac.za

requesting holon's Communication Manager are forwarded to the plugin that initially sent out the CFP (arrow 6). After selecting a service provider, this plugin sends an *accept proposal* message via its Communication Manager (arrow 7) to the service provision holon (arrow 8). The service provision holon's Communication Manager forwards the accept message to its Service Provision SP (arrow 9), which schedules the SPA, indicated as "SERV-PROV Activity" (arrow 10).

The proposal process can be bypassed if a holon does not need to analyse many proposals – i.e. when a service request is only sent to a single holon. In this scenario, a *service request* message is sent to the service provision holon (arrows 1, 2 and 3) and if the holon accepts the request it schedules the SPA (arrow 10) and replies to confirm acceptance (arrows 4, 5 and 6).

6.4.2 Service Provision Activity initiation

Figure 22 shows the various intra-holon interactions that occur when a service provision holon starts an SPA on its schedule. When an SPA ("SERV-PROV Activity" in figure 22) is due to start, the Service Provision EP is informed by its Schedule (arrow 1), after which it is the responsibility of the Service Provision EP to start the activity (arrow 2). When the Schedule receives a start message for "SERV-PROV Activity", it sends the activity to the Execution (arrow 3) and when Execution acknowledges that it has received and saved the activity, Schedule removes the activity from its list of scheduled activities. The Service Provision EP can update the stage 2 data of this activity in Execution, while it is executing the service (arrow 4).

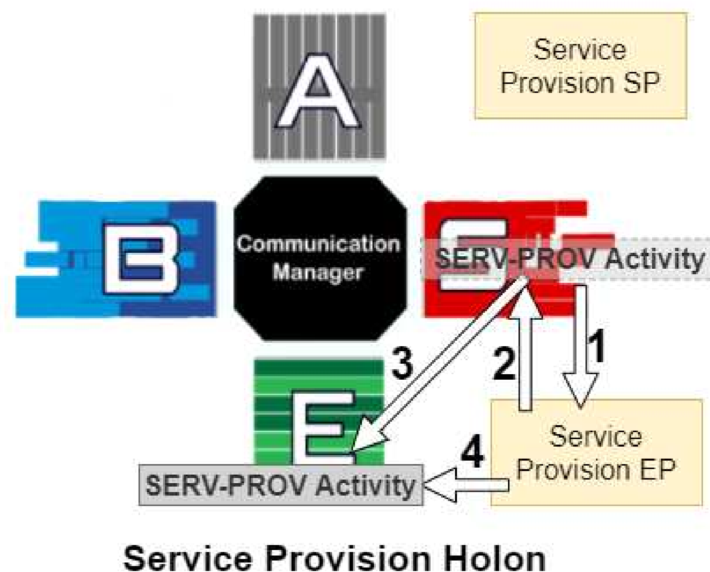


Figure 22: Starting service provision activity

Stellenbosch University https://scholar.sun.ac.za

6.4.3 Service Provision Activity completion

Figure 23 shows the intra- and inter-holon interactions that occur when *inform* messages are sent from a service provider during its service execution, and when the service has been finished or has failed. While executing a service, the service provision holon's Service Provision EP can send *inform* messages via its Communication Manager (arrow 1) to the service requesting holon (arrow 2). The service requesting holon's Communication Manager forwards these *inform* messages to the plugin that initially made the service request (arrow 3). Although not indicated in figure 23, these *inform* messages can also be sent from the service requesting holon to the service provision holon.

When the service provider has finished the service or the service has failed, an *inform-done* or *failure* message, respectively, is sent through the Communication Manager to the service requesting holon (arrows 4 and 5). The service requesting holon's Communication Manager forwards the message to the plugin that initially made the service request (arrow 6). In parallel with both of the scenarios mentioned above, the stage 2 data of the SPA can be updated in the Execution component (arrow 4 in figure 22). After the *inform-done* or *failure* message is sent, the Service provision EP informs the Execution component that the activity is completed (arrow 7). The Execution component then sends the activity to the Biography component and waits for an acknowledge message from the Biography component before removing the activity from its storage (arrow 8).

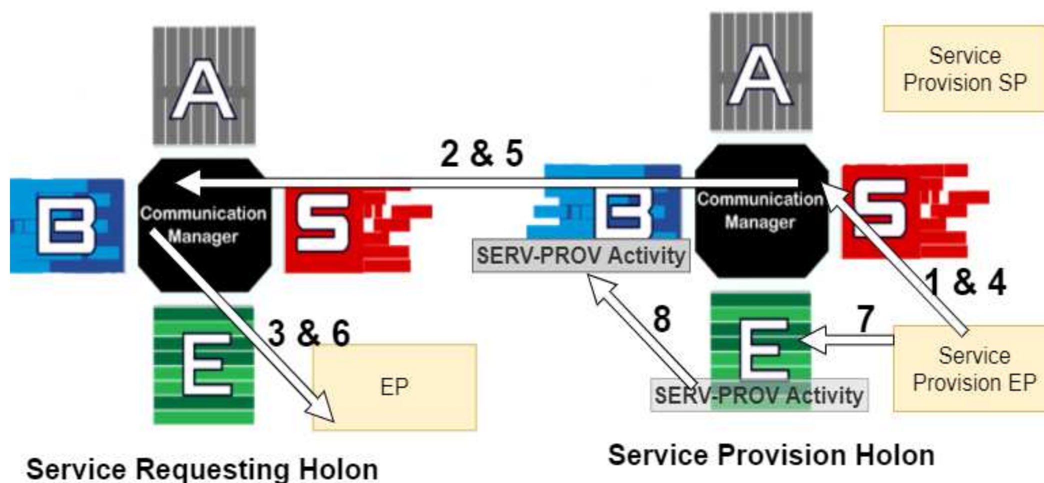


Figure 23: Inform, inform-done and failure

Stellenbosch University https://scholar.sun.ac.za

6.5 Nested services

In the BASE architecture, each service is provided by a single holon. If that holon needs other holons to assist with the service provision, it will need to request their services. This can happen in both the scheduling and execution stages of the service and follows the same sequence of iterations described in sections 6.4.1. to 6.4.3. Figure 24 shows how nested services are supported in the BASE architecture. In this figure a hypothetical situation is illustrated where service provision holons (holons B & C) need services from other service provision holons (holons C & D) in order to execute the services they promised to service requesting holons (holon A & B). When a service provision holon (e.g. Holon B) receives an *inform* message from another service provision holon (e.g. Holon C), this might trigger some action and possibly another *inform* message to its client (e.g. Holon A).

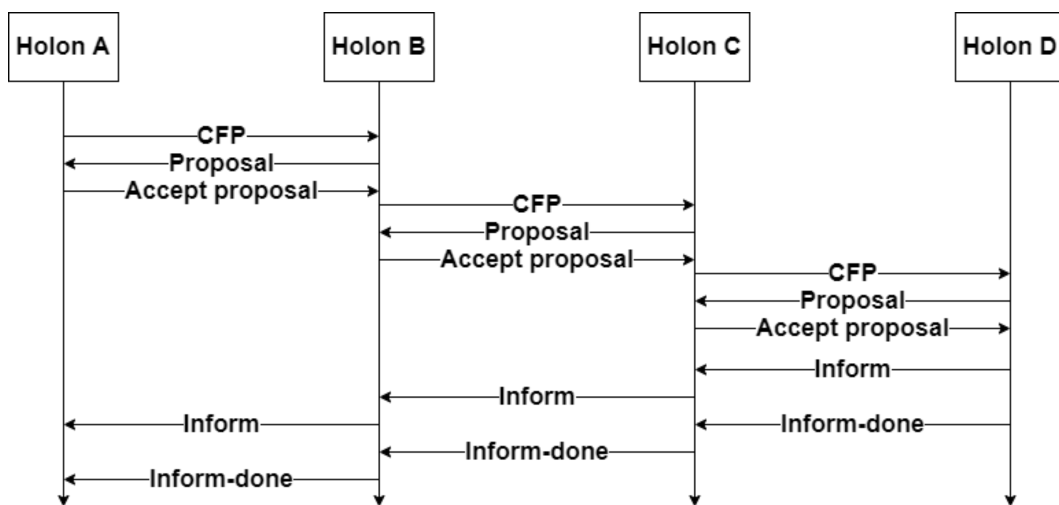


Figure 24: Nested Services

Stellenbosch University https://scholar.sun.ac.za

7 Cyber-physical interfacing

The CPS concept implies the presence of an interface between the physical and cyber elements of a system. This chapter will discuss the extensions that were made to the BASE architecture to enable cyber-physical interfacing with minimal custom development.

7.1 Overview

Devices like temperature sensors, accelerometers, location sensors, loudspeakers, computer monitors, augmented or virtual reality systems, touch screen interfaces, and WhatsApp do not have any other functionalities other than enabling the digital world to communicate with the physical and/or vice versa. These devices can be represented as holons and are henceforth referred to as Interface Holons.

The extended BASE architecture has two standard types of Interface Holons, namely *Observers* and *Informers*. These holons have the responsibility of enabling communication between other holons' BASE shells and physical parts. When a holon's BASE shell needs to exchange data with its physical part, it does not need to manage the cyber-physical interfacing connections and communications and can simply make use of an available Interface Holon.

Observers enable data flow from holons' physical parts to their BASE shells. Common examples of Observers are sensors that can measure properties like temperature, location, heart rate, acceleration, torque, speed, etc. Other holons' BASE shells can create Service Contracts with Observers so that these Observers will share any new observed data about these holons' physical parts, with their BASE shells. Informers enable holons' BASE shells to send messages to their physical parts. Examples of Informers are loudspeakers, screens, warning lights and vibrating devices. Some Interface Holons are both Observers and Informers. The most common examples are touch screen devices like tablets and phones, as well as newer technologies like Virtual Reality (VR) and Augmented Reality (AR) glasses. Observers and Informers have a predefined internal data, functional and communication structure as discussed in section 7.2 – 7.5.

The use of Interface Holons to enable data flow between the physical and cyber worlds is illustrated in figure 25. This figure shows two human workers and five cyber-physical interfaces, each with their own BASE shell. Sparrow (2021) categorized cyber-physical interfaces into two classes: personal and environmental; where a personal interface is dedicated to a single resource and an environmental interface can be used by more than one resource. The factory shown in figure 25 has three environmental interfaces (a camera, a speaker and WhatsApp) and each worker has a personal interface, which measures their heart rate. The camera and two heart rate sensors are examples of Observers, while the speaker is an example of an Informer. The WhatsApp interface is an example of an Observer and an

Stellenbosch University https://scholar.sun.ac.za

Informer combined, since it allows information to flow in both directions. An interesting part to note about the WhatsApp interface is that although each worker has their own phone, the physical part of the WhatsApp interface is shown as one resource. This is because the BASE shell of the WhatsApp interface does not individually communicate with each phone, but uses a third-party service, like Twilio (Twilio, 2021), to communicate with the different phones.

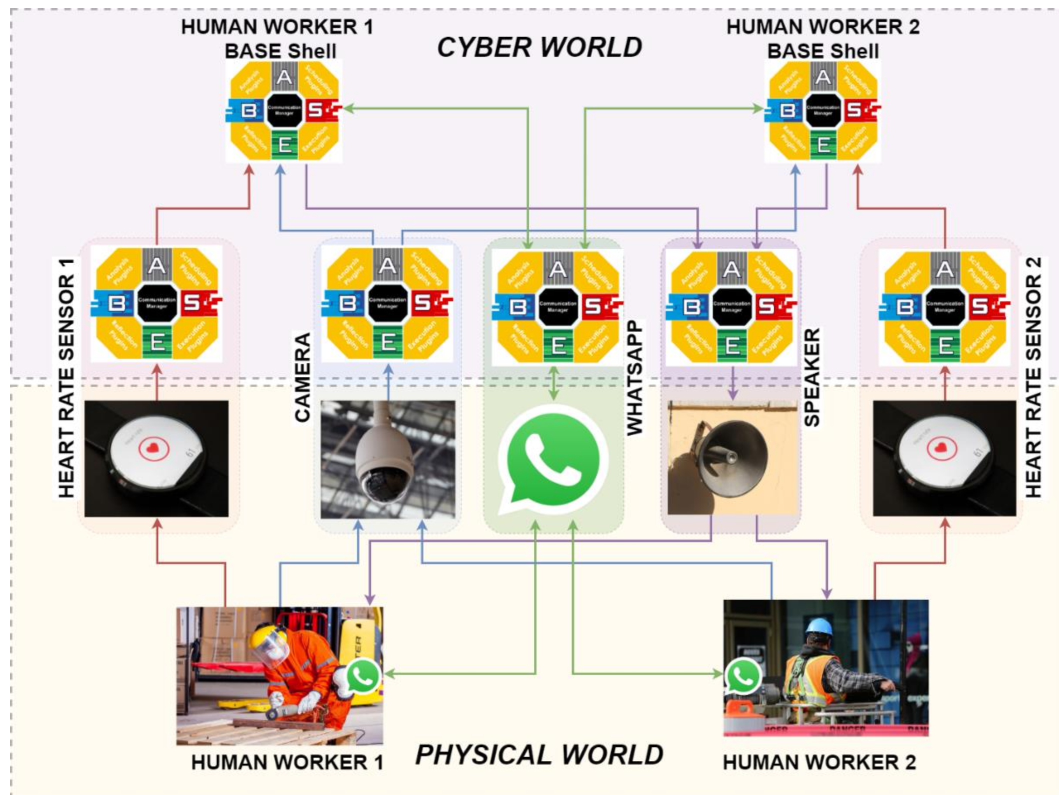


Figure 25: Use of Interface Holons in the BASE Architecture

7.2 Interface plugins

Every Interface Holon needs two standard plugins, namely a Service Provision SP (introduced in section 6.4) and an Observer EP or Informer EP (depending on the interface type), which are both built on top of the Service Provision EP. The Observer EP and Informer EP are generic for all Observers and Informers. The interface specifics are contained in the *Interface Module*, which is simply a container of program code. The Interface Module contains the code required to start the custom *Interface Process* and is also used by the Observer EP to verify Identifiers (discussed in section 7.3.2). When an Interface Holon is initialised, its Observer/Informer EP calls a function in its holon's Interface Module to start the *Interface Process*, where this process is a new thread that starts and manages the connection with the holon's physical device. This Interface Process must be able to send and/or receive messages to and from the Interface EP that started it. Each

Stellenbosch University https://scholar.sun.ac.za

Observer/Informer EP needs to monitor its holon's Interface Process and restart it if it fails.

When new data has been observed about another holon by an Observer's physical device, this physical device sends the data to its BASE shell's Interface Process. The Interface Process forwards this new data to its Observer EP and the Observer EP shares the new data with the relevant holon. When another holon requests an "Inform" service from an Informer, the Informer EP sends a message to its Interface Process to execute the service. This Interface Process then messages its physical device with the custom details required to inform the physical part of the BASE shell that requested the "Inform" service. The Interface Module of the case study's feeding sensor is shown in appendix D.

7.3 Interface Service Descriptions

7.3.1 Overview

Figure 26 shows, in JavaScript Object Notation (JSON) format, the standardised Service Description structure for cyber-physical interfacing services.

Service Description =
{
"Resources":{
    "Resource type 1":**Interfacing Details,**
    "Resource type2": **Interfacing Details,**
    ...
    },
    "Interface description": Anything
}

Interfacing Details = {
"Available identifiers":"all" OR ["Identifier1","Identifier2"...],
"Identifier type":"ID" OR any other string,
"Topics":{
    "Topic1":{"Accuracy":0-100,"Unit":"kg"/"s"/etc.},
    "Topic2":{ "Accuracy":0-100,"Unit":"kg"/"s"/etc.},
    ...
}

Figure 26: Service Description data structure of cyber-physical interfacing services

Observers and Informers have the same Service Description structure, except that the Informer does not have a "Topics" field. The highest level of the Service Description has two fields: "Resources" and "Interface description". The "Interface description" field can be used by developers to add any descriptive information about the interface. The "Resources" field contains all the resource types for which interfacing is possible, and each resource type field contains three fields, namely:

Stellenbosch University https://scholar.sun.ac.za

"Available identifiers", "Identifier type" and "Topics". Identifiers and Topics are discussed in more detail in sections 7.3.2 and 7.3.3.

7.3.2 Identifiers

Identifiers are used by Interface Holons to distinguish between the different service requesters (i.e. other holons) with which they interact. When using Informers, holons do not need to specify an identifier – their ID will always be used, which the Informer can find in their BC. One very important functionality of Informers is that they prevent one holon's BASE shell to communicate directly with another holon's physical part.

Like Informers, Observers can use a holon's ID as Identifier, but Observers can also use one of the holon's attributes, e.g. in the case study the feeding sensor used the Electronic Identification Number (EID), an attribute of each sheep. The type of Identifier that an Interface Holon is able to use is specified in its "Identifier type" field. Note that the identifier type can differ for the different type of resources that an Observer can interface with. There are two ways in which a holon can specify for which Identifier they are making a service request to an Observer:

- The holon can add an "Identifier" field to the request arguments of the Service Contract with one of its attribute's used as the value of this field.

- The holon can specify no Identifier in the Request Arguments, in which case the Interface Holon will use the holon's ID.

In both scenarios, the Observer will verify the Identifier by calling the *valid_identifier* function in its Interface Module. Observers allow more than one holon to use the same Identifier. For example, all the machines in a room can request to observe the ambient temperature of the same room. The Identifier used, might then be something similar to "Room ID", which must be an attribute in all the machines' BASE shells. However, most of the time each Identifier is meant for one holon.

When the "Available identifiers" field is equal to "all", the Interface Holon can (but will not necessarily) enable cyber-physical interfacing for all holons of some resource type, as long as they provide a valid Identifier. When an "Available identifiers" field is not equal to "all", but is a list of identifiers, the Interface Holon can only enable interfacing for the Identifiers specified, and will reject requests from holons with Identifiers that are not in this list.

7.3.3 Topics

Topics are the names/descriptions of observed data, e.g. "Temperature", "RPM", etc., and each Topic specifies the estimated accuracy and unit of the observed data (as shown in figure 26). When making a request to an Observer, a holon can specify

Stellenbosch University https://scholar.sun.ac.za

which Topics' data it wants to receive or not specify anything, in which case all of the Topics' data will be shared with the holon.

7.4 Inter- and intra-holon communications

After an Informer executed an "Inform" service, the Service Contract with the requesting holon is completed. However, "Observe" Service Contracts are not terminated after new observed data has been sent to a client holon, but remain active while the Observer is active and the client has not cancelled the service. Sections 7.4.1 to 7.4.3 show the inter- and intra-holon interactions related to Observers and Informers. The standard plugins, Interface Module and Interface Process used in each of the interactions were introduced in section 7.2.

7.4.1 Interface initialisation

Figure 27 shows the interactions necessary to add a new Interface Holon to a CPS. The initialisation of all Interface Holons follows the same procedure. The Interface EP (Observer EP and Informer EP) calls the *start_interface* function in its holon's Interface Module (arrow 1). The Interface Module starts the Interface Process (arrow 2) and returns the address of this process to the Interface EP (arrow 3), since this EP needs to monitor the Interface Process to restart it if necessary. As mentioned in section 7.2, the Interface Process is different for each Interface Holon and will manage the connection with the holon's physical device or with a gateway to the device (arrow 4).

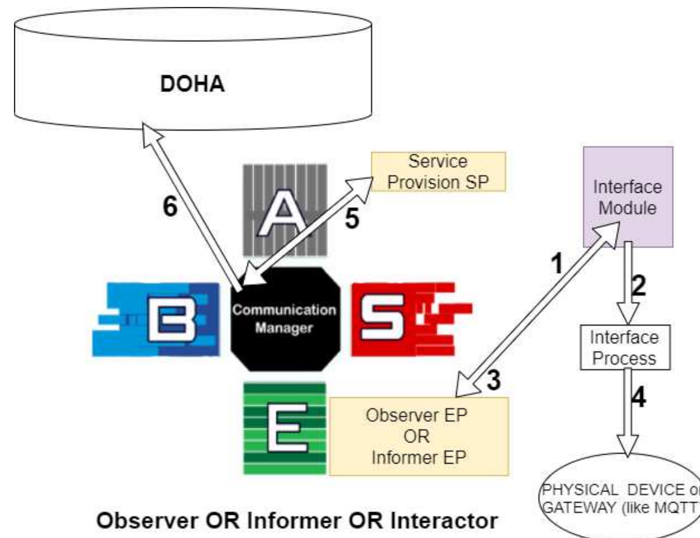


Figure 27: Interface Holon initialisation

Stellenbosch University https://scholar.sun.ac.za

7.4.2 Observers

7.4.2.1 Overview of plugins required by Observers' clients

Any BASE shell that wants to automatically receive observed data relevant to its holon from all available Observers, needs to have two standard plugins, namely: Resource Data SP and Resource Data EP. Together these plugins enable a BASE shell to know about new Observers, request their services and receive observed data. The details of how these plugins interact with their own shell and with Observers are discussed in sections 7.4.2.2 and 7.4.2.3.

7.4.2.2 Service Request

Figure 28 shows the BASE shell of a holon, Resource A, requesting an Observer's "Observe" service, just after the Observer has been added to the system. Before the Observer was added to the system, Resource A's Resource Data SP registered with DOHA for holons that can provide "Observe" services. Section 6.2 showed how plugins register for services with DOHA. Just after the Observer registers with DOHA, DOHA informs Resource A's Resource Data SP (via its Communication Manager) about the Observer (arrow 1). Resource Data SP checks if the new Observer can provide data for its holon (Resource A), by evaluating the Service Description of the Observer (described in section 7.3). If the new Observer can provide data for Resource A, Resource Data SP schedules a new RDAA activity, "Act 2115(RDAA)", with the scheduled start time set to the current time (arrow 2).

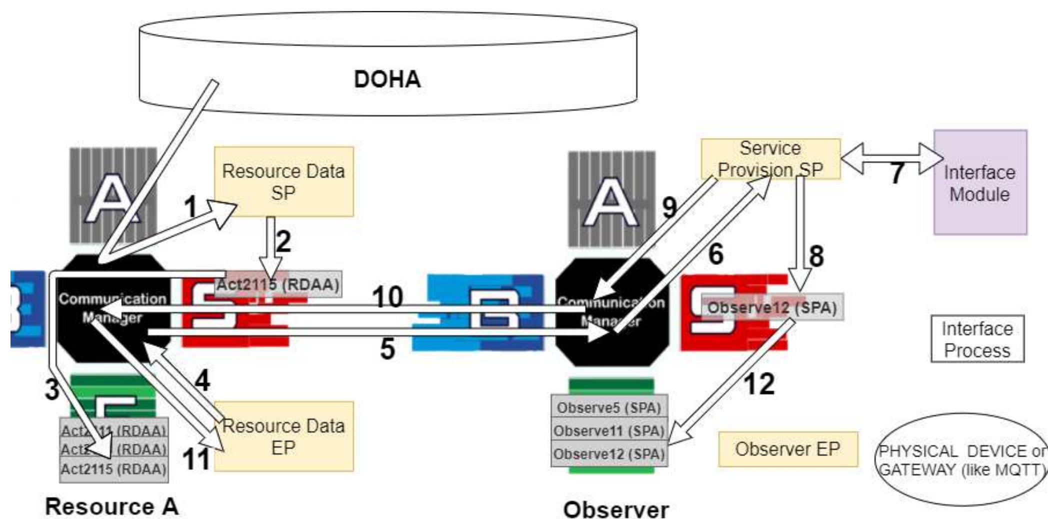


Figure 28: Service request to an Observer

Resource A's Resource Data EP is informed, by its BASE shell's Schedule, about the new RDAA activity and starts it (arrow 3; more details in figure 13). Note that there might be more than one RDAA activity in Execution at the same time – one activity for every Observer in the system that can provide data for this holon.

Stellenbosch University https://scholar.sun.ac.za

Resource Data EP requests its Communication Manager (arrow 4) to forward a service request to the Observer (arrow 5). In this service request's request arguments, no Topics are specified, because Resource Data EP is registering for all available topics. The Observer's Communication Manager forwards this request to its Service Provision SP (arrow 6), which checks two requirements before accepting the service request and scheduling a new SPA. These requirements are (in order):

1. Check if Resource A's resource type is one of the resource types in this Observer's Service Description.

2. Check if the Identifier provided by Resource A is valid (arrow 7) and included in the "Available identifier" list (discussed in section 7.3.1).

If both of the above requirements are met, Service Provision SP schedules a new SPA, "Observe12 (SPA)", with the scheduled time set to the current time (arrow 8), and accepts the service request via its Communication Manager (arrows 9, 10 and 11). Observer EP will be notified about the SPA by its BASE shell's Schedule, triggering it to start this activity (arrow 12) and add Resource A to its list of clients.

7.4.2.3 New data observed and shared by Observer

Figure 29 shows the Observer's Interface Process receiving new data from its real-world part (arrow 1). The Observer's Interface Process sends the new data to its Observer EP (arrow 2), which triggers Observer EP to update the stage 2 data of the relevant SPAs (arrows 3) and share the new data with all the registered clients as inform messages (arrows 4 and 5). The Communication Manager of each client forwards the inform message to their Resource Data EP (arrow 6). Each client's Resource Data EP copies the RDAA in execution, adds the observed data to its stage 2 data and moves the copied RDAA into its biography (arrow 7) without removing the original RDAA from its Execution. RDAAs were explained in more detail in section 5.7.2.

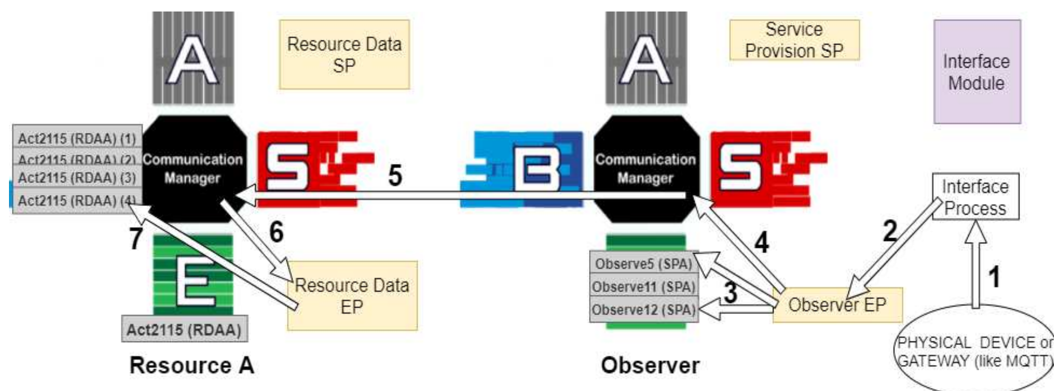


Figure 29: New data observed by Observer and shared with clients

Stellenbosch University https://scholar.sun.ac.za

7.4.3 Informers

7.4.3.1 Overview of plugins required by Informers' clients

Any holon that wants to use an Informer can do so from any of its plugins and does not need any standard plugins. A service request can be sent to Informers at any point in time when their "Inform" service is required, and this service will be executed as soon as possible. Thus, no long-lived Service Contract is required as in the case with Observers.

7.4.3.2 Service Request

Figure 30 shows one of Resource A's EPs, EP-A, using an Informer to send a message to its holon's physical part. EP-A starts by requesting DOHA for all resources with a service of Service Type "Inform", and DOHA returns the BCs of all holons that satisfy this. EP-A goes through all the returned BCs, removes the BCs of Informers that cannot interface with Resource A's physical part, and then selects the preferred Informer. After selecting an Informer, EP-A requests (via its Communication Manager) the Informer to send a message to its holon's physical part (arrows 1 and 2). The Informer's Communications Manager forwards this request to its Service Provision SP (arrow 3). This Service Provision SP checks if it can communicate with Resource A's physical part (by checking the Service Description of its Informer) and, if so, it schedules an SPA, "Inform23 (SPA)" (arrow 4), and accepts Resource A's request (arrows 5 and 6). Resource A's Communication Manager forwards this accept message to EP-A (arrow 7). The Informer's Schedule informs its Informer EP of the SPA that is due to start and, subsequently, Informer EP starts this activity (arrow 8; details shown in figure 13).

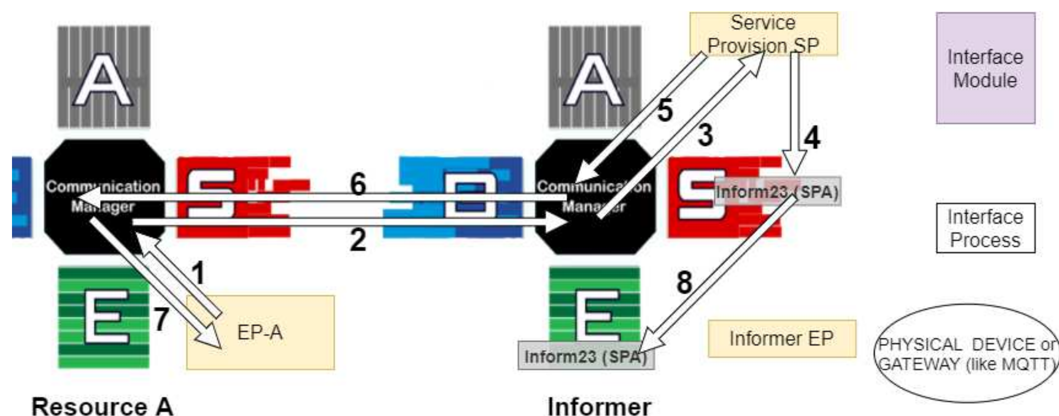


Figure 30: Service request to an Informer

7.4.3.3 Informer executes "Inform" service

Figure 31 shows the interactions between the Informer EP and its Interface Process to execute the "Inform" service requested by Resource A's BASE shell. The

Stellenbosch University https://scholar.sun.ac.za

Informer EP messages its Interface Process, asking it to execute the "Inform" service request (arrow 1). Interface Process communicates with its physical device (arrow 2), to send a message to Resource A's physical part, using this device. The message delivery result is returned to Informer EP (arrow 3), which will send either *inform-done* or *failure* to Resource A (arrows 4, 5 and 6), depending on whether the message was delivered successfully or not. Informer EP writes this result to the stage 2 data of the "Inform23 (SPA)" activity and then finishes this activity (arrow 7 and 8).

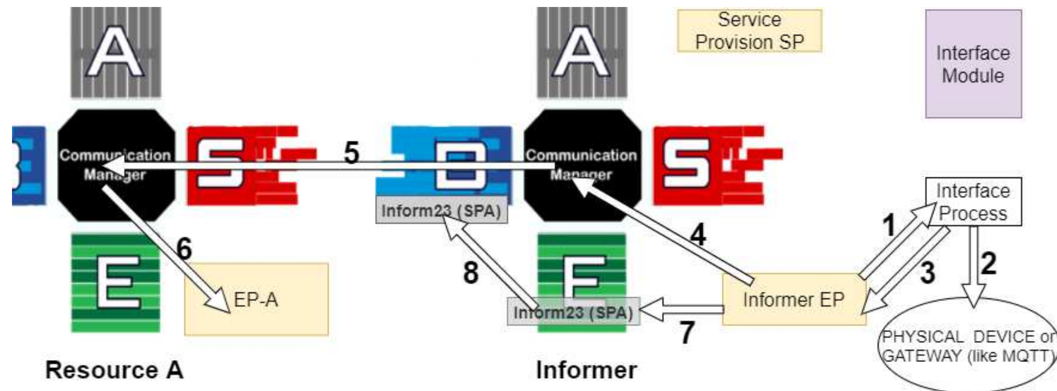


Figure 31: Execution of an "Inform" service

7.5 Interface Holon activities and attributes

As shown in section 7.4, each type of Interface Holon has its own interface activity: "Observe (SPA)" or "Inform (SPA)", depending on the type of interface. In addition, all Interface Holons have three common types of activities: "Device State Change", "Interface State Change" and "Restart interface".

Activities of type "Device State Change" are created when the Interface Process informs its EP that the physical device's state has changed from working to down or vice versa. This provides developers with a mechanism to track the state of an Interface Holon's physical device. However, this functionality requires that the physical device be programmed to communicate its state to the Interface Process.

Activities of type "Interface State Change" are created when the Interface Process fails or has been restarted successfully. An Interface Process's failure is detected by the EP that started it (Observer EP or Informer EP), since this EP monitors the Interface Process after starting it via the Interface Module. It is the responsibility of this EP to start and finish an activity of type "Interface State Change" and to try to restart the interface. If the restart succeeds, the plugin will have to log another activity of type "Interface State Change", with the *Result* set to "Success". Activities of type "Restart interface" can also be scheduled from a UI if a user of the system suspects that an Interface Process is unresponsive. All Interface Holons have the same attributes, as shown in appendix E.

Stellenbosch University https://scholar.sun.ac.za

8 Generic implementation of architecture extensions

This thesis continues to use Erlang to implement the BASE architecture, because of the benefits discussed in section 2.4.4. This chapter will discuss the generic BASE architecture components that were developed and all important implementation strategies.

8.1 Storage components

All data storage in this thesis' BASE architecture implementation was realised using Erlang's ETS tables. Each ETS table in an Erlang application has the same high-level functionalities a Structured Query Language (SQL) table in an SQL database, even though the way they work differ completely. ETS tables are processes in the Erlang runtime system, making access to data in these tables very fast. What is known as rows or data entries in SQL databases is termed objects in ETS tables. Each object is contained in an Erlang tuple and Erlang records can be used as templates for these objects, ensuring new objects are stored in the correct *field order* – the order of the columns in SQL tables. (Erlang -- ets, 2021)

There are four types of ETS tables, namely: set, ordered set, bag and duplicate bag. Sets and ordered sets can only have one object per *key*, while bags and duplicate bags can have many objects per *key*. *Keys* are used to separate objects in an ETS table and can also be used to quickly find or delete objects. Ordered sets return data in the order of their keys' values, which means if keys are auto-incremented or uses the date and time in their constructions, queried data will be returned from oldest to newest. For this reason, ordered sets were the type of ETS table used for all ETS tables in this thesis' BASE architecture implementation.

ETS tables exist in RAM, and thus if an Erlang application with ETS tables is shut down, all data in them are lost. This problem is addressed by using the *ets:tab2file* function to save tables to text files, and the *ets:file2tab* function to create tables from text files. In this thesis' implementation, each table's text file was updated after every create or delete operation in the table, to ensure no data is lost when the system shuts down without warning. However, the tables are retrieved from the text files only at start-up and then kept in memory, otherwise the system would be too slow. If the available RAM on the computer running the Erlang application would be a problem, then the tables should not be kept in memory, but rather read from text files for every operation. ETS tables allow smart read and delete operations using match specifications. These specifications allow data entries to be read/deleted based on the value of any of the entries' properties – similar to what SQL databases allow with their "where" statement.

Stellenbosch University https://scholar.sun.ac.za

Two alternatives to ETS tables were considered, namely DETS tables, which is a disk-based version of ETS tables, and Mnesia, a distributed telecommunications database management system. Both DETS tables and Mnesia are disk-only databases, which may result in longer latencies. However, Mnesia is frequently used in large, distributed Erlang applications and should be considered in future work.

8.2 Administrative components

DOHA's reception component was implemented as an Erlang *gen_server* process and provided all the functionalities required for DOHA as outlined in sections 4.3 and 6.2. DOHA's storage component was implemented as discussed in section 8.1.

The architecture allows for multiple UIs, but for this implementation only one was implemented that can serve multiple users. The implemented UI was a browser-based UI, served by the Erlang application through a web socket using the Cowboy Erlang library (Cámara, 2021). HTML, JavaScript and CSS code was developed for the front-end of this UI. This code had to enable many front-end functionalities to facilitate reconfiguring the system (adding/removing holons, editing management attributes and editing holons' schedules) and viewing live data about holons. This code was also responsible for the communication with the web socket exposed by the Erlang application.

Figure 32 shows a screenshot of the UI where it is used to plot the quantity and duration of a sheep's feeding from the 15th of July to the 19th. The other images of the UI are shown in appendix H. Note that this plot did not require any custom code for the case study and was used for all holons with numeric data in their RDAA activities.

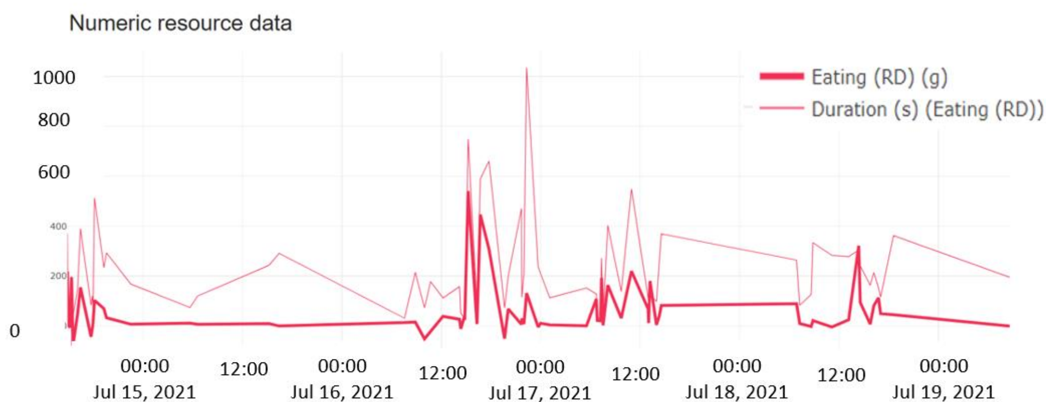


Figure 32: Observed data of a sheep plotted in UI (adapted image from implementation's UI with enlarged text)

Stellenbosch University https://scholar.sun.ac.za

As mentioned in section 4.4, the BASE architecture consists of three loggers, namely: a system logger, an error logger and a holon interactions logger. A system logger was implemented that could record the static state of the system after big changes have been implemented in the system. The system logger was implemented as a *gen_server* process that could receive requests to log the system's state. This call was implemented after big changes in the system, e.g. after a holon was added or removed or after each sheep holon in the system received new weight data. When receiving this call, the system logger waited five seconds to allow the system to settle and then logged the date and time, the number of holons, the number of ETS tables, the number of Erlang processes, the system's overall RAM usage and the RAM usage of the ETS tables.

The error logger could be used by any process in the system to log an error. Each logged error contained the error description, the module name, function and line number where the error was encountered, and the date and time at which the error occurred.

The holon interactions logger was called by all the Communication Managers in the system to indicate when they sent or received any service-oriented messages. When a call was received, the holon interactions logger logged the date and time, sender holon ID, receiver holon ID, message type (CFP, proposal etc.) and the Service Type. This logger was used to debug communications between holons from the UI. JavaScript code was written that was able to decode the logged holon interactions into and display it as sequence diagrams. The front-end of the holon interactions logger was shown in figure 10.

8.3 Supervisors

Erlang has built-in supervisor processes which are responsible for starting, stopping and monitoring their child processes. Erlang supervisors have different restart strategies that a developer must choose from, namely:

- *one_for_one* – only the child process that failed is restarted,
- *one_for_all* – when one child process fails, all child processes are restarted,
- *rest_for_one* – the child process that failed and all child processes that were added after this child process, are restarted,
- *simple_one_for_one* – this is the same as *one_for_one*, except that child processes are added instances of the same process type, thus all child processes are *gen_servers,* supervisors or *gen_fsm*s – Erlang's finite state machines.

One of the core properties of the BASE architecture is the independence between BASE shells and between their internal components. One component's failure

Stellenbosch University https://scholar.sun.ac.za

should not affect any other components, even under the same supervisor. Thus, the *one_for_one* and *simple_one_for_one* restart strategies were the only restart strategies that could be used.

The BASE Supervisor (figure 9) and Plugin Supervisor (figure 11) were implemented with the *simple_one_for_one* restart strategy, because this restart strategy is ideal for situations where the children are not specified beforehand, but change throughout the life of the system. Only a process template needs to be given to this supervisor when it is initialised. When new children need to be added *supervisor:start_child(Sup_PID,[])* can be called, where *Sup_PID* is the address of this supervisor process. No other start up arguments need to be specified, because all children have the same start up arguments.

All other supervisors in the system have fixed children and thus, the *one_for_one* restart strategy was used for these. This restart strategy allows the developer to specify the children for each supervisor in the supervisor's initialisation function, so that these children are started up by the supervisor after it is initialised. DOHA's supervisor code is shown in figure 33 as an example. This code also shows two other properties that need to be specified for all supervisors, namely the *restart intensity* and *period*. These two properties determine the maximum number of times (*restart intensity*) any child is allowed to fail within a certain time period (*period*). Another thing to note from this code is that all children also have individual properties, namely *id*, *start*, *restart*, *shutdown*, *type* and *modules* and the details of this can be found in Erlang's online documentation (Erlang -- supervisor, 2021).

```
-module(doha_sup).
-behaviour(supervisor).
-export([start_link/0, init/1]).

start_link() ->
  supervisor:start_link({local, ?MODULE}, ?MODULE, []).

init([]) ->
  DohaRecep = #{id => 'doha_recep',
    start => {doha_recep, start_link, []},
    restart => temporary,
    shutdown => 2000,
    type => worker,
    modules => [doha_recep]},
  DohaDefStorage = #{id => 'doha_default_storage',
    start => {doha_default_storage, start_link, []},
    restart => temporary,
    shutdown => 2000,
    type => worker,
    modules => [doha_default_storage]},

  {ok, {#{strategy => one_for_one,
    intensity => 5,
    period => 30},
    [DohaDefStorage,DohaRecep]}
  }.
%%SLOC:20
```

Figure 33: DOHA supervisor process source code

Stellenbosch University https://scholar.sun.ac.za

8.4 The Erlang gen_server process, gen_server calls and spawn_monitor

Figure 34 shows an example of the concepts and functions that will be discussed in this section. The figure shows a *gen_server* process (the Biography Reception) handling a *gen_server:call* (*update_s3_data*) by first checking from who the call is received (*security_functions:check_sender*) and then using *spawn_monitor* to create a new process that handles the call.

```
handle_call({update_s3_data,ActId,S3Data},From,State)->
  Recepts = maps:values(Recepts State#bio_state.reception_map),
  FullPermittedList = lists:append(State#bio_state.instance_plugins),
  case security_functions:check_sender(FullPermittedList, From) of
      true->
          case io_lib:printable_list(ActId) of
            true->
               MyPid = self(),
               spawn_monitor(bio_processes,update_s3_data,[ActId,S3Data,MyPid,State,From]),
               {noreply,State};
            _->
               {reply,{error,"ActId must be a string"},State}
         end;
      _->
         {reply,{error,"Only this shell's processes can make this call"},State}
  end;
```

Figure 34: Example of a gen_server process handling a call

Erlang's *gen_server* process was used for all of the core components of each BASE shell. This was done because of *gen_server's* standard set of interface functions, their functionalities for tracing and error logging and the fact that they do not need any extra development to allow Erlang supervisors to monitor and restart them. Erlang has a tool called the Observer, which can show the entire supervision tree of an Erlang application as long as all processes are standard Erlang processes (like *gen_servers* and supervisors). Thus, the utilisation of the Erlang Observer was another motivation for using *gen_server* processes.

In section 5.5, it was mentioned that each BASE shell's internal components do not communicate with other components outside of their BASE shell, except for the Communication Manager. Moreover, within each BASE shell there are restrictions on which components can communicate with each other, as summarised in table 2. There are a few methods of communicating with a *gen_server* process, but only the *gen_server:call* allows the *gen_server* process to see the address of the process that sent the call. Consequently, the *gen_server* can decide to evaluate the call or ignore it, which is why the *gen_server:call* was chosen as communication method with all *gen_servers*. This facilitated all component processes to only respond to calls from components permitted to communicate with them.

Stellenbosch University https://scholar.sun.ac.za

Some actions executed by component processes, in response to calls from other component processes, require much time. If the process that receives the calls also executes these actions, it would sometimes be unavailable to other component processes. Subsequently, component processes were programmed to *spawn* a new process when a call was received, which would execute the time-consuming computations and then reply to the calling component. The problem with this approach was that the spawned process would not be allowed to communicate with other component processes within its BASE shell (because of the shell privacy restrictions discussed in the previous paragraph). This was overcome by using *spawn_monitor*, which allows the component process to spawn and monitor a new process. All component processes were then programmed to check both the address of a calling process, as well as the address of the process monitoring the calling process, to determine if communication should be allowed. This was implemented using Erlang's *process_info* functionality.

8.5 Inter-process communication in a distributed Erlang application

Erlang has a datatype, not found in many programming languages, called an atom. Atoms are literals with their value encapsulated in their name, for example the atom *cat123* has the value *cat123.* In the original implementation of the BASE architecture by Sparrow (2021), all components had atom names, because there was only one BASE shell implemented. In Erlang a process must either have a unique name or not have a name at all (i.e. use the process ID). Erlang also has a maximum number of atoms that can be used in an application. Furthermore, atoms are very memory consuming because of an atom table that is kept in every Erlang application's memory. As such, a decision was made to assign atom names only to DOHA, the loggers and the Communication Managers of the BASE shells – all other processes are identified by their unique process ID.

DOHA and the loggers need to have an atom name, otherwise other components in the system would not be able to find them. The atom names for Communication Managers were used to make it easier to debug the system using the supervision tree of the Erlang Observer.

8.6 Standard plugins

All the standard plugins introduced in chapters 6 and 7 were implemented in the BASE architecture implementation of this thesis. The implemented Observer EP and Informer EP used Interface Modules for their case specific parts as outlined in section 7.2. Functionality was added to the Observer EP to not only wait for data from its Interface Module, but to also register with its BASE shell's Schedule for activities that contain new observed data. This allowed new observed data to be added from the UI to an Observer's Schedule to be shared with other holons. This functionality was used by the sheep scale holon as discussed in section 9.5.2.

Stellenbosch University https://scholar.sun.ac.za

The Resource Data SP and Resource Data EP were implemented completely generic for all holons and did not require any case specific code, because of the way in which RDAAs' stage 2 data is structured (shown in section 5.7.2), and the standardisation of messages from Observer holons. These two plugins should always be implemented in a holon, even if there are not Observers in the system that can provide data for this holon, because new Observers might be added to the system later on. If a holon does not have these plugins, it would not be able to automatically request any "Observe" services from newly added Observers.

Service Provision SP and Service Provision EP, both required a generic and a case-specific code base. The case specific parts of these plugins were extracted into functions that can be called from the generic parts. These functions were put into a module, specific to the holon that the plugin was part of. All of these specific plugin modules were named as follows: "{HolonType}_{PluginName}_mod" – all in lowercase letters, so that each plugin with a case specific part knows how to access its case specific module. For example, the case specific module for the Service Provision SP of any sheep holon was named *sheep_service_provision_sp_mod*.

Three types of faults could occur in case specific modules:

- the module does not exist;
- the relevant function was not implemented; and
- there was a mistake in the function's implementation that caused it to crash.

The Service Provision SP and Service Provision EP both had default actions when a call to a relevant function in their case specific module failed. For example, if the Service Provision EP started an SPA and called its case specific module's *execute* function, but this call failed, it simply finished the SPA and replied to the service requester, stating that it failed to execute the service.

Stellenbosch University https://scholar.sun.ac.za

9 Case study

To demonstrate the benefits and shortcomings of the extended BASE architecture, a real-world case study implementation was required. For this thesis, an educational sheep farm was used as case study. This chapter introduces the criteria that was used to select the case study and, subsequently, describes the selected case study. Thereafter, the user requirements and case study holons are identified. Finally, the implementation details specific to the case study are discussed.

9.1 Selection criteria

The case study implementation is intended to provide a basis on which the different extensions to the BASE architecture can be demonstrated and evaluated. To this end, the following criteria were formulated for the selection of the case study:

- The case study must integrate more than one type of resource into the CPS.
- At least one holon in the system must be an Interface Holon (Observer or Informer).
- At least one example of each activity category (SPA, RAA, RDAA) must be present in the implementation.
- At least one example of each attribute type (Personal, Relational, Management and Condition) must be present in the implementation.
- At least one SP, EP, RP and AP must be implemented in a BASE shell in the implementation.
- At least one service-oriented interaction must be implemented between holons that are not Interface Holons.

9.2 Description

The case study that was selected for this thesis is the Stellenbosch University's educational sheep farm. This farm is used by students in the university's agricultural department for livestock and feed related experiments.

This farm was recently afforded an opportunity to use a sheep scale and feeding data sensor from an agricultural system integrations solution provider. All sheep on the farm were fitted with Radio-Frequency Identification (RFID) tags. Sheep weight data was captured from a scale, which identifies the individual sheep by means of an RFID tag reader. A load cell and RFID tag reader was installed at each feeder, which allowed the feeding sensor to captured feeding data (quantity eaten and duration of eating) of individual sheep.

Stellenbosch University https://scholar.sun.ac.za

Figure 35 shows the farm's setup, which includes the feeding sensor that is connected to the load cells and RFID tag readers at the different feeders, and the sheep scale. The sheep were grouped into camps for experiments.

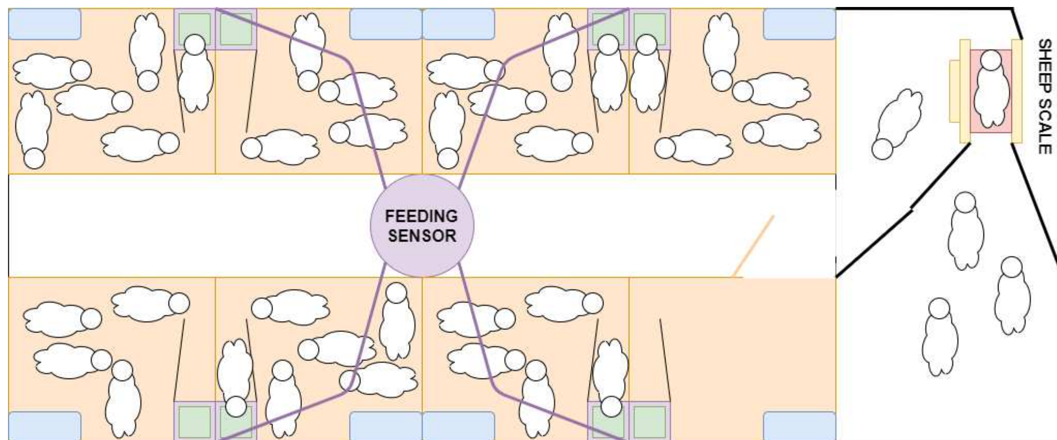


Figure 35: Educational sheep farm setup

The RFID tag reader and load cells of each feeder are connected to a central data acquisition device (Raspberry-Pi), which pushes the recorded data into a cloud database. This Raspberry-Pi sends all recorded data to an SQL Server every 30 minutes. To show the real-time capabilities of the BASE architecture implementation, a method of acquiring this data in near real-time was implemented as discussed in section 9.5.1.

The recorded weight data from the sheep scale can either be integrated into BenguFarm Sheep & Goat (BenguFarm, 2015), if the user owns this software, or downloaded onto a memory stick as a text file. The latter method was used in this case study, since the university's farm does not have access to the expensive BenguFarm software.

9.3 User requirements

With the existing sheep data acquisition system, students have to download data as text files and manually organize the data to enable the desired analysis for each sheep. Furthermore, students have to calculate the average values of the variables of interest within each sheep group – aggregating the data to represent the average sheep of each group. Apart from this tedious and time-consuming process, students cannot determine the status or progress of their experiments until they were completed. The implementation of the extended BASE architecture aimed to improve the monitoring, management and real-time analysis of sheep data by addressing six user requirements:

- Simplify the collection of feeding and weight data for the sheep on the farm.

Stellenbosch University https://scholar.sun.ac.za

- Extract and archive each sheep's data from the farm's collected data so that each sheep's data is grouped together.
- Automatically calculate the overall statistics of the variables of interest.
- Aggregate the variables of interest for each sheep group, since in many experiments the average statistics of the different groups, and not of the individual sheep, are compared.
- Visualise the data and variables of interest in near real-time, so that students can view the progress of their experiments before they are completed.
- Facilitate a notification service to notify students of critical events – e.g. when a feeder has not recorded any data or a sheep has not eaten for some user-defined time period.

In the above-mentioned requirements, the considered variables of interest are daily food intake (kg), sheep mass (kg), sheep mass gain (kg) and Feeding Conversion Ratio (FCR) (mass of feed consumed (kg) over increase in body mass (kg)). Statistics of these variables must be calculated, e.g. average, maximum and minimum values over different time periods.

9.4 Holon identification

Before implementation of the system could start, the resources on the farm that had to be integrated as holons, needed to be identified. The most obvious resources are the **sheep**. As mentioned in section 9.2, the different camps on the farm are used to group sheep in experiments. In these experiments, each camp had different input variables, e.g. each camp might have a different type of food, or each camp contains a different breed of sheep. However, the sheep camps and groups are identified as two different resources, because the **camps** had to do with housing sheep, where the **groups** are used to aggregate sheep data for experiments. Each sheep group could also be relocated to different camps during an experiment and thus a decision was made to let each group and camp be represented as a holon. The students using the system are integrated as **human** holons.

The **sheep scale** and **feeding sensor** are integrated as Observer holons. The physical containers from which the sheep ate are integrated as **feeder** holons and does not form part of the feeding sensor, since these would exist whether or not the feeding sensor is installed. An SMS-based notification service is selected for sending notifications to the users/stakeholders, since not all workers might have smartphones that can run WhatsApp or some Android/Apple application. Vonage (Vonage, 2021) was chosen as service provider for these SMS messages. The **Vonage SMS service** was integrated into the system as an Informer, which can only provide a service to human holons. The attributes of each of these holons can be

Stellenbosch University https://scholar.sun.ac.za

seen in appendix E (Interface Holons) and appendix F (sheep, groups, camps, feeders and humans).

9.5 Implementation

9.5.1 Platform implementation and integration overview

The BASE platform and its integration with other systems and services for the case study is presented in figure 36. The BASE platform, running on an Erlang runtime node, is hosted on a remote PC. The BASE platform interfaces with a browser-based UI, a cloud-based Message Queuing Telemetry Transport (MQTT) broker, a TCP-tunnelling service (ngrok) and an SMS notification service (Vonage).

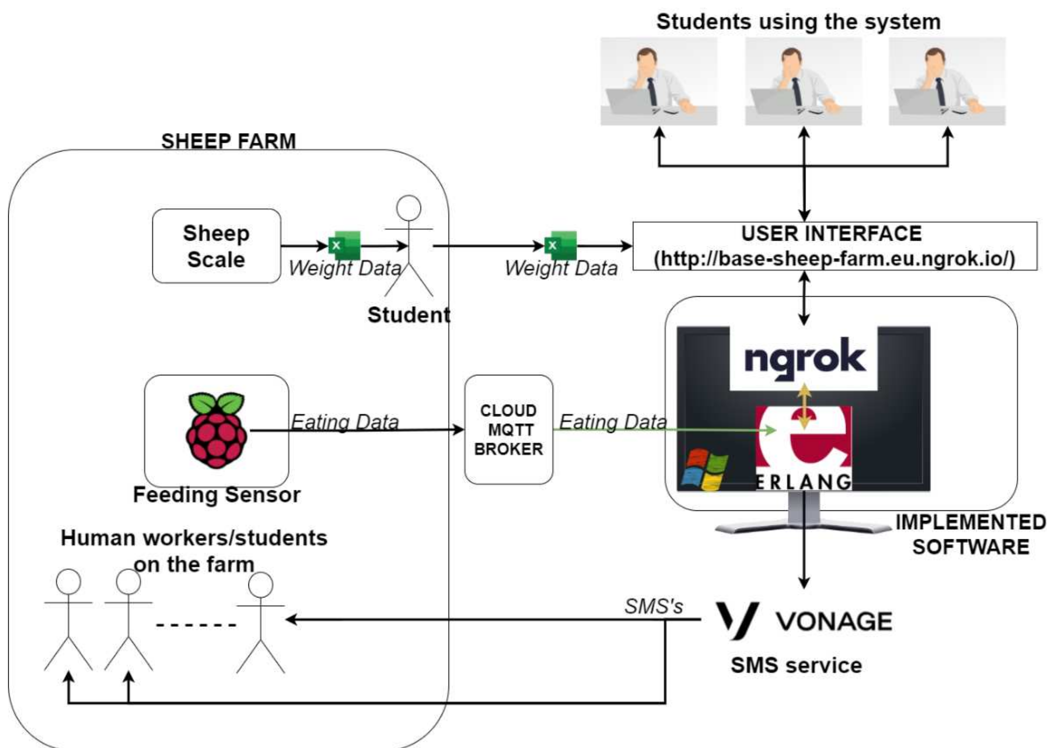


Figure 36: Implementation setup

A browser-based UI was developed for the case study implementation's UI. The UI is served by the BASE platform through a public web socket and consists of HTML, JavaScript and CSS. The web socket is exposed on one of the PCs local TCP ports using the Cowboy Erlang Library (Cámara, 2021). Ngrok, (2021) is used to make this local TCP port public and assign to it a fixed Uniform Resource Locator (URL).

The feeding sensor records how much and for how long a sheep eats every time a sheep is at a feeder. It also adds the ambient temperature to every data packet sent via the MQTT broker, which can be acquired by all camp holons that are in the

Stellenbosch University https://scholar.sun.ac.za

same location as the feeding sensor. The feeding sensor does not have the ability to push the feeding and temperature data to the cloud in real time. Consequently, a Python script that can publish the data in real-time, via a cloud MQTT broker, was added to the feeding sensor.

MQTT is a communication protocol that is built on the TCP transport layer protocol and supports a publish/subscribe architecture (Banks, Briggs, Borgendale & Gupta, 2021). MQTT communications always require at least one broker with any number of clients that publish and subscribe to this broker. The broker is responsible for managing connections, accepting published messages and forwarding published messages to subscribers. The case study implementation required a cloud-based MQTT broker, since the feeding sensor and the BASE platform resided on two different networks. MaQiaTTo (MaQiaTTo, 2021) is a free cloud-based MQTT broker that was used in this implementation.

The source code of the Python script that implements the MQTT client for the feeding sensor is shown in appendix G. The Python script establishes and maintains an MQTT client for the feeding sensor, enabling it to publish new data, via an MQTT broker, to other MQTT clients that subscribe to the data. The Interface Process of the feeding sensor's BASE shell also establishes and maintains an MQTT client that is subscribed to this data. The way in which new data is shared from the Interface Process to the relevant sheep and camp holons was described in section 7.4.2.3.

The sheep scale had no connectivity capabilities and the recorded weight data had to be downloaded to a portable storage device (a USB memory stick). To allow integration of this data into the Erlang application, the students uploaded the data from the portable storage device via the BASE platform's UI.

As mentioned in section 9.4, the SMS API of Vonage (Vonage, 2021) was chosen to send notifications to students that are using the system. Vonage's SMS API can be used by making a REpresentational State Transfer (REST) request containing the API URL, the message text, the phone number to which the message must be delivered, and a unique API key given to a user when they make an account with Vonage.

9.5.2 Plugins and activities of case study holons

Table 4 shows all the case study holon types, except for the Interface Holons (feeding sensor, sheep scale and Vonage SMS service), together with their plugins and the activity types that they support. Each case study holon also has a Resource Data SP and Resource Data EP, even though this is not shown in table 3. Holons provide all their services through their service provision plugins. Table 3 shows that all holons, except the sheep and feeder holons, required service provision plugins. Sections 9.5.2.1 to 9.5.2.6 discuss the functionalities of the custom plugins (indicated in bold in these sections) and activities required for the different case study holons. Some attributes of the case study holons are used in discussions in

Stellenbosch University https://scholar.sun.ac.za

these sections and appendix F can be referenced to see the descriptions of the case study holons' attributes.

Table 3: Case study holons with their custom plugins and supported activities

Holon Type	Plugins	Supported Activities
Sheep	• sheep_aggregation_sp & sheep_aggregation_ep • sheep_relocation_ep & sheep_relocation_ap • sheep_eating_rp & sheep_fcr_ap	• Relocate (RAA) • Aggregate (RAA) • Weight (RDAA) • Eating (RDAA)
Group	• group_service_provision_sp & group_service_provision_ep • group_relocation_ep & group_relocation_ap • group_scaling_ap • group_aggregation_ap	• Relocate (RAA) • Add_Sheep (SPA) • Remove Sheep (RAA)
Camp	• camp_service_provision_sp & camp_service_provision_ep • camp_scaling_ap	• Add_Sheep (SPA) • Add_Group (SPA) • Add_Feeder (SPA) • Temperature (RDAA) • Remove_Sheep (RAA) • Remove_Group (RAA) • Remove_Feeder (RAA)
Feeder	• feeder_relocation_ep & feeder_relocation_ap	• Relocate (RAA)
Human	• human_service_provision_sp & human_service_provision_ep	• Relocate_Sheep (SPA) • Relocate_Feeder (SPA)

9.5.2.1 Interface Holons

Three Interface Holons were implemented, namely the feeding sensor holon (Observer), sheep scale holon (Observer) and Vonage SMS service holon (Informer). The two Observer holons used the Observer EP and the Informer holon used the Informer EP. The only custom development required for each of these holons was in their Interface Modules, which contained the code required for their Interface Processes that communicated with their real-world devices or services. These details were already discussed in section 9.5.1 and appendix D contains the Interface Module of the feeding sensor.

9.5.2.2 Camp holons

Each camp holon provides three services, namely: "Add_Sheep", "Add_Group" and "Add_Feeder". Each camp holon has a management attribute, "Sheep limit", which is set from UIs and is used to determine if a group or sheep can join. A camp

Stellenbosch University https://scholar.sun.ac.za

can add a group if it does not already host a group or some sheep, and if the number of sheep in the group is less than or equal to the sheep limit. A camp can add a sheep (not part of a group) if it already hosts sheep that are not part of a group, and the sheep limit is not yet reached. A camp can host more than one feeder at a time, but in the case study, each camp only hosted one feeder.

Camp holons do not need to give holons permission to leave, which is why "Remove_{holon type} (RAA)" services are not provided by camp holons. When a sheep, group or feeder holon leaves its camp, it informs the camp holon about it and the camp holon removes the relevant holon by scheduling, starting and finishing a "Remove_{holon type} (RAA)" activity. Every time a sheep, group or feeder is added or removed in a camp, the **camp_scaling_ap** updates the "Sheep", "Group" or "Feeder" attribute of the camp.

In the case study, the feeding sensor also recorded the ambient temperature of the shed it was in. Each camp in the case study was in the same location as the feeding sensor, which is why all camp holons received temperature data from the feeding sensor, and logged these temperatures using "Observe Temperature (RDAA)". This ambient temperature data was not utilised for anything, but can potentially be used in future experiments on the sheep farm.

9.5.2.3 Feeder holons

In section 9.4, it was mentioned that feeder holons are required to represent each feeder on the farm, separately from the feeding sensor. For this case study, the feeder holons were developed with limited functionality. Each feeder holon has a primary function of holding (in a management attribute) the product ID of the food inside its physical feeder. Students using the system could use the UI to edit this product ID after refilling a feeder. The product ID is requested by the BASE shells of sheep that are in same camp as a feeder, to be used as post-execution data for their eating data. This allows students to not only see how much and for how long each sheep ate, but also what each sheep ate.

Each feeder can be relocated to a camp, and this is triggered when a new "Relocate (RAA)" RAA is created from the UI in the schedule of the feeder's BASE shell. The **feeder_relocation_ep** is responsible for starting, executing and finishing the feeder's relocation. It does so by first making a service request to the camp to which the feeder needs to be relocated. When this is rejected, the **feeder_relocation_ep** finishes the RAA with the "Result" set to *"Failed"* and the "Reason" set to the reason for the service request being rejected (refer to appendix B on RAA data structures). When the service request is accepted, the **feeder_relocation_ep** requests DOHA for holons that can provide a service of type "Relocate_feeder (SPA)", and sends out CFPs to all these holons (typically human holons). The first holon to respond with a proposal message is selected. When the service is finished, the RAA is completed and the "Result" is set to *"Success"*. When "Relocate (RAA)" is put into the Biography, the **feeder_relocation_ap** updates the feeder's "Camp" attribute to reflect the feeder's new camp.

Stellenbosch University https://scholar.sun.ac.za

9.5.2.4 Sheep holons

The sheep holons were developed with the most detail, since most of the case study requirements could be satisfied in these holons. Each sheep can be relocated to a group or to an empty camp. This relocation is triggered when a new "Relocate (RAA)" activity is created in the sheep's schedule from a UI. The logic executed in the **sheep_relocation_ep** is almost identical to that of the feeder_relocation_ep, except that the sheep can be relocated to either a group that is hosted by a camp, or a camp that hosts sheep without a group. When "Relocate (RAA)" is put into the Biography and was successful, the **sheep_relocation_ap** updates the sheep holon's "Camp", "Feeders" and "Group" attributes. It does so by retrieving the ID of the new camp/group from the RAA's stage 1 data and requesting the other two attributes ("Feeders" and "Group"/ "Camp") from the new camp/group.

Each sheep's resource_data_sp and resource_data_ep ensures each sheep receives and logs any recorded data about the sheep, which in this case study was eating and weight data. The **sheep_eating_rp** was developed to add the details of the food a sheep ate to its "Eating (RDAA)" activities' stage 3 data. It does so by obtaining the sheep's feeder from its "Feeders" attribute and then requesting the feeder holon for the product ID of the food it contains (as explained in section 9.5.2.3). After receiving the product ID, the **sheep_eating_rp** adds this to the stage 3 data of the relevant "Eating (RDAA)" activity.

The **sheep_fcr_ap** was the most complex plugin developed for the case study implementation and implemented many calculations required to satisfy the case study requirements. This AP updates its sheep's state attributes whenever new eating or weight data is put into the Biography of its sheep. All of these state attributes are statistics (over different time periods) about the sheep's daily food intake, mass, mass gain and FCR. These attributes are listed in appendix F.

Whenever the attributes are updated, and the sheep is part of a group, the **sheep_aggregation_sp** creates a new "Aggregate (RAA)" RAA that contains all the updated condition attribute values in its stage 1 data. The **sheep_aggregation_ep** then starts this activity, retrieves the sheep's group ID from its attributes, shares the updated condition attributes with the sheep's group and finishes the activity.

9.5.2.5 Group holons

When a group holon receives data from its sheep, the data is passed to its **group_aggregation_ap**, which updates the group's condition attributes. Each sheep group's condition attributes represent the condition attributes of the average sheep in the group. The steps followed by the **group_relocation_ep** and **group_relocation_ap** to relocate an entire group to another camp are very similar to the steps followed by the two relocation plugins of sheep holons.

Stellenbosch University https://scholar.sun.ac.za

Similar to camps, groups can also host sheep. However, for a group to host sheep, the group itself must be hosted by camp. When a group receives an “Add Sheep” service request, it first needs to check with the camp that it is in if there is any space left before accepting the request. The **group_scaling_ap** updates the group’s “Sheep” attribute when a sheep has been added or removed.

9.5.2.6 Human holons

Due to time constraints, human holons were developed with limited functionality and are only intended to show how the humans might be integrated in a fully developed system. In this case study, human holons can provide two services, namely: “Relocate_Sheep” and “Relocate_Feeder”. In a more completely developed holonic implementation, human holons should have also provided the service of collecting data from the sheep scale and storing it on the computer. In the case study, the only implemented cyber-physical interfacing for human holons was that each holon could send an SMS to its human’s mobile telephone. This was used when a sheep holon needed to inform all humans in the system that the sheep has not eaten for a long time, or when the feeding sensor holon needed to inform all humans in the system that it has not been active for a long time, potentially indicating some failure.

Stellenbosch University https://scholar.sun.ac.za

10 Evaluation

Each of the required BASE architecture extensions in section 3.3 were identified based on shortcomings of the original BASE architecture as a reference architecture for complex, reconfigurable CPSs. The requirements of complex, reconfigurable CPSs then serve as basis for the evaluation of the extended BASE architecture. Building on these requirements, this chapter formulates evaluation criteria comprising quantitative and qualitative metrics. Measurements of the metrics are obtained through the analysis of, and experiments on, the extended BASE architecture – providing the basis for a holistic evaluation and discussion.

10.1 Criteria

To evaluate the extensions presented in this thesis, the CPS requirements that were addressed are used as evaluation criteria. The consideration of the requirements as criteria leads to the identification of a set of quantitative and qualitative metrics, which are used to facilitate the evaluation. The relationships between the requirements and metrics are presented in figure 37.

Sections in which metrics applied

Green columns - 10.2.1
Blue columns - 10.2.2
Gold columns - 10.2.3
Red columns - 10.2.4
White columns - 10.3

	Evaluation metrics												
	Quantitative										Qualitative		
Requirement	Interface Holon code reuse rate	Slowest service initialisation	% messages lost between shells	Service provision code reuse rate	Holon development time	Reconfiguration time after scaling	Overall code reuse rate	Computational requirements	Generic component failure rate	% custom component failures isolated and handled	Error transparency	Modularity	UI Intuitiveness
Cyber-physical interfacing	x		x										
Service-oriented cooperation		x	x	x									
Decentralisation										x		x	
Scalability		x			x	x		x					
Diagnosability											x	x	
Generalisation	x			x	x	x	x						
Reliability			x						x	x			
Usability						x							x

Figure 37: Relationship matrix showing the relationship between the evaluation criteria and metrics of the extended BASE architecture

Stellenbosch University https://scholar.sun.ac.za

As metric, the code reuse rate in Interface Holons evaluates two requirements of the implementation, namely cyber-physical interfacing and generalisation. Section 7.2 showed how the standard plugins of Interface Holons encapsulate the generic functionalities of Interface Holons, allowing developers to only focus on holon-specific implementation details. This generalisation can be evaluated by measuring the amount of code reused for each Interface Holon.

In order for a holon to use a service, it first needs to discover other holons which can provide the desired service, send out service requests to these holons, and receive at least one proposal. The longest time required for a holon to complete these steps is used as a metric to evaluate the implementation's service-oriented cooperation. It is expected that the number of holons in the system would have a direct impact on this time. As such, this time is also used to evaluate the implementation's scalability, because it would indicate how many holons could be added before the performance of the implementation is affected.

Lost messages between BASE shells would cause cooperation errors and make the system unreliable. Thus, the percentage of messages lost between BASE shells evaluates the implementation's service-oriented cooperation and reliability. In addition, this metric evaluates the implementation's cyber-physical interfacing ability. This is because all cyber-physical interfacing messages are communicated via Interface Holons' BASE shells, which must be able to communicate with other BASE shells in the system. In such cases, the loss of messages between BASE shells will also imply a loss of data between the cyber and physical systems.

The code reuse rate of service provision plugins is used as another metric to evaluate the implementation's cooperation ability. Similar to the code reuse rate used for Interface Holons, this metric also evaluates the implementation's generalisation. In this case, the level of generalisation in holons' service provision plugins is measured by measuring the percentage of code reused in holons' service provision plugins.

The time required to develop new holons, and the time required to reconfigure the implementation after adding these holons, have a notable influence on how effectively an implementation can be scaled (Kruger & Basson, 2018). As such, the metrics of development time and reconfiguration time are used to evaluate the scalability of the implementation.

A reduction in development time is expected when generic functionalities are extracted into generic components, since developers do not need to spend time to develop these functionalities. Therefore, development time also evaluates the implementation's generalisation.

Reconfiguration time refers to the time required to add a new holon, edit its attributes and make any other necessary reconfigurations in this holon or any existing holons. In the developed implementation, these reconfigurations need to be done through the UI and, thus, the reconfiguration time also evaluates the

Stellenbosch University https://scholar.sun.ac.za

implementation's usability. Reconfiguration time also evaluates the generalisation of the implementation, because a system with many generic components would need less reconfiguration than a system with very few generic components.

It should be noted that both development and reconfiguration time can be influenced by external and subjective factors, like developer experience or fatigue, and thus more objective metrics are also required to evaluate the implementation's scalability and generalisation. For scalability, the increase in computational and memory requirements associated with the addition of holons to the system is inspected (Kruger & Basson, 2018). For generalisation, the code reuse rate in all holons (not only Interface Holons) is identified as an appropriate metric.

To evaluate the implementation's reliability, both the robustness and the resilience of the implementation need to be considered. To determine the implementation's robustness, the failure rate of its generic components is used as metric. When custom components fail, the implementation must isolate these failures and apply the appropriate fault-handling and/or restart strategy. Therefore, the percentage of custom component failures correctly handled is used to evaluate the implementation's resilience. Furthermore, the metric serves to evaluate the implementation's decentralisation, since the failure of an individual component should not affect the operation of other components in a decentralised system.

Three qualitative evaluation metrics are identified, namely: error transparency, modularity (both used by Kruger & Basson (2018)) and intuitiveness. An error transparency metric is used to evaluate the implementation's diagnosability. Modularity refers to how much a system is broken up into modules, which is why this metric is used to evaluate the implementation's decentralisation. Diagnosability is improved when a system is broken up, and thus modularity also evaluates the implementation's diagnosability. Intuitiveness refers to the ease of using the implementation's UI (from the perspective of a user in the system) and is proposed as a qualitative measure of the implementation's usability.

10.2 Quantitative metrics

This section presents the measurements of the different metrics used for a quantitative evaluation of the implementation. Measurements for several quantitative metrics could be obtained from an analysis of the implementation's source code and from the logged data, recorded while the implementation was deployed. For the measurement of some metrics, development and reconfiguration experiments and scaling experiments had to be performed. The methods and experiments used to obtain measurements of the metrics, along with the obtained results, are discussed in sections 10.2.1 to 10.2.3. However, the discussions about the implications of these metrics are deferred to section 10.4.

Stellenbosch University https://scholar.sun.ac.za

10.2.1 Analysis of implementation source code

An analysis of the source code of the implementation of the extended BASE architecture is performed to obtain measurements for the code reuse rate in the case study holons (which include the Interface Holons) and the service provision plugins. Each of these metrics consider the number of SLOC, as used by Kruger & Basson (2018). The results of the source code analysis of the case study holons are presented in table 4.

Table 4: Code reuse rate of the case study holons

	Sheep	Group	Camp	Feeding sensor	Sheep scale	Vonage SMS service
Custom SLOC	1025	329	691	125	36	32
% code reused	89.7 %	96.42 %	92.77 %	98.61 %	99.6 %	99.64 %

All holons in the BASE architecture have the same generic code base, mostly from the core components of their BASE shells. In the implementation, this generic code base had 8864 SLOC. Table 4 gives a custom SLOC measurement and percentage of code reused for the most important case study holons. The feeder and human holons were not developed with the same detail and completeness as the other holons and are thus left out of this table. The sheep holon was the most complex case study holon type. Nonetheless, only 10.3 % of the sheep holons' entire BASE shell implementation source code is custom code.

The Interface Holons required the least amount of custom code, and the BASE shell of the most complex Interface Holon, the feeding sensor, only required 1.39 % of custom code. The feeding sensor's custom code required an MQTT client, which is why it had much more SLOC than the other two Interface Holons.

The SLOC measurements and percentages of code reused of the case study holons' service provision plugins is shown in table 5. The case study implementation included three holon types providing services (other than cyber-interfacing services), namely the group holons, camp holons and human holons. Group holons provided an "Add_Sheep" service, which required 171 custom SLOC. Camp holons provided an "Add_Sheep", "Add_Group" and "Add_Feeder" service, which required 124 custom SLOC. Human holons provided a "Relocate_Sheep" and "Relocate_Feeder" service, which required 98 custom SLOC. The generic parts of the two service provision plugins required in each service provision holon, had a total SLOC measurement of 1494. As can be seen from these SLOC measurements, the number of services has no clear influence on the code reuse rate in the service

Stellenbosch University https://scholar.sun.ac.za

provision plugins. For example, the group holons service provision plugins needed 1.4 times more SLOC than that of the camp holons, even though the group holons only provided one service and the camp holons provided three services.

Table 5: Code reuse rate in the service provision plugins

	Group	**Camp**	**Human**
Service provision plugins' custom SLOC	171	124	98
% code reused	89.73 %	92.34 %	93.84 %

Another important consideration is that a different case study might have required service provision holons with completely different SLOC values. However, the SLOC values reported are expected to be representative of typical implementations of services provided by a holon in the BASE architecture.

10.2.2 Analysis of implementation reliability during deployment

Before the implementation was deployed, it was confirmed that core component failures are logged if they do occur. This was done by forcefully terminating each of the core components during testing and checking if the failure event is logged in the error log. Figure 38 shows the caption of an entry in the error log after the attribute reception process of a sheep holon's BASE shell terminated. In figure 38, the time of the error (in seconds since 1 January 1970, 00:00), the module in which the error occurred, the line in which the error occurred ("N/A", in this case), the resource ID ("Sheep1") and the error description is shown. No core component or custom component failures were found in the error log at the end of the implementation's deployment, indicating that not a single core component or custom component failed.

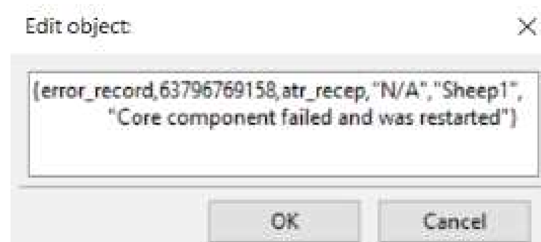


Figure 38: Logged error found in error log after forcefully killing the attribute reception process of a holon with ID = "Sheep1"

The implementation's ability to isolate and handle custom component failures could not be checked during the deployment period because no custom components failed. To force an error, a fault was intentionally added into a sheep holon's AP

Stellenbosch University https://scholar.sun.ac.za

that would cause the plugin to fail the first time an analysis is to be performed on new eating data. The plugin was installed and the first time that it was triggered to analyse new data, it failed and was restarted by its BASE shell's Plugin Supervisor. This test was repeated for each of the plugins and the implementation managed to isolate and log all of the failures and restart all of the plugins appropriately.

10.2.3 Development and reconfiguration experiments

Experiments were conducted to measure the time required to develop new holons, as well as the time required to perform the necessary reconfigurations when new holons were added to the system. In these experiments, the author (acting as the developer) timed himself when developing the custom code for the case study holons and also timed how long it takes to make any necessary reconfigurations. These measurements, for the different case study holons, are shown in table 6. As mentioned in section 10.2.1, the feeder and human holons were not developed with the same detail as the rest and are again left out of this table. While it is not possible in this thesis to compare the measured development times with that of alternative implementations, the measured times provide an indication of the development time required to develop holons in the BASE architecture.

Table 6: Development and reconfiguration time of new holons

	Sheep	Group	Camp	Feeding sensor	Sheep scale	Vonage SMS service
Development time (hh:mm)	68:06	18:58	32:51	05:35	02:05	03:32
Reconfiguration time (hh:mm)	0:12	00:14	0:02	00:01	00:01	00:02

The reconfiguration of the implementation was performed as a two-part process. The first part involved the addition of the holons to the existing implementation and updating their attributes. The second part involved any other reconfigurations required in the new holon or any existing holons. The sheep and group holons had to be scheduled to relocate to a camp when they were added to the system, which is why their reconfiguration times are longer than that of the camp holons, which did not require any reconfiguration. However, this scheduling only required a few minutes because the UI allows users to quickly add or remove activities from holons' schedules. The two Observers (feeding sensor and sheep scale) did not require any reconfiguration, because all BASE architecture holons have a Resource Data EP (introduced in section 7.4.2) that automatically handles any interactions required with new Observers in the system. The Informer (Vonage SMS service) also did not require any reconfigurations, except for its own installation.

Stellenbosch University https://scholar.sun.ac.za

10.2.4 Scaling experiments

To measure the computational load and slowest service initialisation for varying amounts of holons in the system, one sheep scale holon together with varying amounts of sheep holons (0, 9, 99, 499 and 999 sheep holons) were added to the implementation. The sheep scale holon was added to have a service provider that could be requested for its services by the multiple sheep holons. This was required to measure service initialisation time for varying numbers of holons in the system.

The experiment was done on a computer with an Intel i7-8750H CPU (2.20 GHz clock speed) and 16 GB of RAM. To measure CPU usage, the Windows Performance Monitor application was used and to measure the RAM usage of the implementation, the Erlang Observer was used. Appendix I shows plots of the CPU and RAM usage obtained from the Performance Monitor and Erlang Observer when 100 holons were added to the system. CPU time was calculated by multiplying the duration of the holon addition with the average CPU usage percentage. The service initialisation time was measured for single and parallel service initialisations. Single service initialisations involved each sheep holon finding and initialising the service at a different time than other holons. Parallel service initialisation involved all sheep holons finding and initialising the service at the same time.

Figure 39 summarises the results from this experiment. Both the CPU and RAM usage was measured during and after each set of holons were added, but it was found that the CPU usage was negligibly small after the holons were added, even with 1000 holons running on the BASE platform. This is why figure 39 does not show the CPU usage after holons were added. The RAM usage increased continuously while holons were being added and was at its maximum after all the holons were added, which is why the RAM usage shown in figure 39 is the stead state RAM usage after all holons were added. The detailed results can be found in table 7 in appendix I.

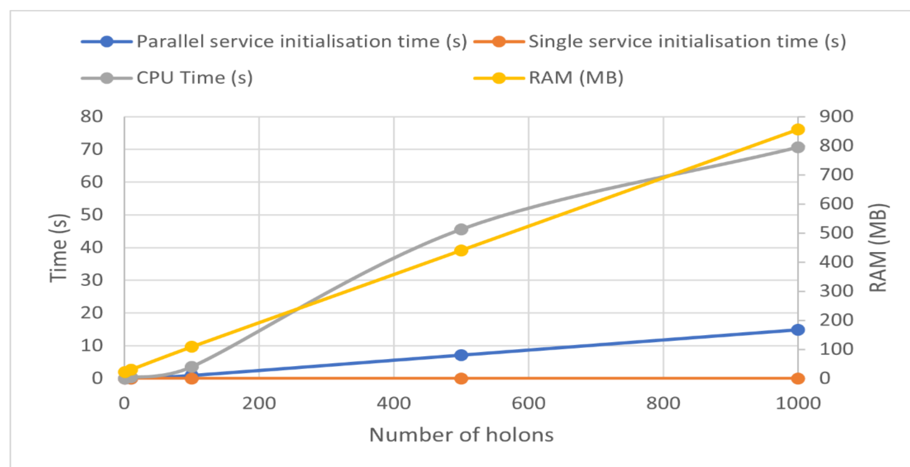


Figure 39: Scaling experiment results

Stellenbosch University https://scholar.sun.ac.za

The CPU time required for every experiment increased at approximately 0.074 s per holon, while the total required RAM usage increased at 0,833 MB per holon. The parallel service-initialisation time increased as more holons were added, and for 1000 holons the longest parallel service-initialisation time measured was just over 14 seconds. However, it is important to note that this scenario - where all holons try to initialise a service with the same service provider at the exact same time – is unlikely to occur. The single service initialisation time barely changed (from zero to 16 milliseconds) as the number of holons increased and is a more accurate representation of how long service initialisation would take.

The total RAM usage includes the RAM usage of the ETS tables, which in each of the experiments was negligibly small. However, as more data gets added to a holon's storage components, these ETS tables will become bigger, but the rate at which they grow is dependent on the type of data that is being stored in them. For example, one holon might have activities which have very large amounts of data stored in them, where another holon's activities might have almost no data stored in them. As mentioned in section 8.1, the ETS tables in the system do not need to be active in RAM when they are not being used, but if they are not the storage operations are slightly slower. It would be up to the developer to decide if they are very concerned about speed of operations or rather want the system to use as little RAM as possible.

10.3 Qualitative evaluation

This section evaluates the implementation's error transparency, modularity and the intuitiveness of its UI. These metrics are difficult to quantify and, as such, are evaluated through a qualitative discussion.

During development and testing of the case study plugins, the author (as developer) had to debug several of his own coding mistakes. The error log helped to detect these mistakes, as well as show in which module and line each mistake occurred. The error log also shows for which holon the error occurred, which enhanced debugging, since many holons used the same Erlang modules. A screenshot of a logged error was shown in figure 38. The holon interactions logger was only developed halfway into the development of the case study holons. After this addition, the debugging of holon interaction errors was much faster. The biggest benefit of the holon interactions logger was to narrow cooperation errors down to the exact step in the CNP, and to know if it was the service provider or receiver that did not react as expected. The system logger made it possible to identify memory leaks (because of spawned processes that did not terminate as expected) and to see which operations took the most CPU time.

The original BASE architecture was already modular within each BASE shell, since it was divided into the Biography, Attributes, Schedule and Execution components together with the four classes of plugins. This thesis extended this modularity to the different BASE shells and the platform management components. DOHA is divided

Stellenbosch University https://scholar.sun.ac.za

into a reception and a storage component – as are the three types of logging, namely error, system and holon interactions.

The newly developed service provision plugins are also modularised. The *service_provision_sp* handles all CFPs and proposal acceptances, while the *service_provision_ep* drives the execution of a service, handles any cancellations and informs service requesters when a service has been completed. Interface Holons are modularised by extracting all generic functionalities into their *observer_ep* or *informer_ep* and allowing developers to add any context-specific details in the Interface Module.

The UI developed for the case study managed to incorporate all the requirements listed in section 4.5. The students who performed research at the sheep farm also managed to use the UI without any major difficulties; however, a thorough explanation of the UI was provided beforehand.

10.4 Discussion

This section will discuss the results from section 10.2 and 10.3, focussing on what these results imply about the benefits and shortcomings of the extensions made to the BASE architecture in this thesis.

The code reuse rates shown in section 10.2.1 indicate that the extensions managed to encapsulate complexity into generic components. It is expected that when fewer custom SLOC are required, development productivity would increase – as confirmed by the short development times reported in section 10.2.3. The number of generic components, together with DOHA, also result in short reconfiguration times when new holons were added to the implementation. The Interface Holons had the fewest SLOC, proving the effectiveness of using an Observer EP or Informer EP for all of the generic functionalities and allowing developers to only focus on the custom code required in the Interface Modules.

The fact that no generic or custom components failed while the implementation was deployed shows a definite increase in reliability, which can mostly be attributed to the supervisors, the Coordinator, and the failure handling capabilities built into the generic components. However, it is expected that if the implementation was not implemented in Erlang the same reliability might not have been possible, but this would need to be confirmed in future work. The deployed implementation also had no messages lost between BASE shells, which is crucial for holons to cooperate efficiently. This also indicates that any cyber-physical interfacing messages received by Observers would always reach the BASE shells of the holons observed, enabling the BASE shells to reflect the state of their holons' physical parts.

Scalability was a fundamental shortcoming of the original BASE architecture, but the results shown in section 10.2.4 indicate a significant improvement in this regard. The results indicate that the only limitation to the implementation's scalability is

Stellenbosch University https://scholar.sun.ac.za

the available memory (RAM) on the computer. The author expects that with future work, the amount of RAM needed per holon can most likely be decreased even more, allowing BASE architecture implementations to be extremely scalable and deployable to resource-constrained controllers (e.g. Raspberry-Pi). In normal situations, service initialisation times between holons are barely influenced by the number of holons in the system. This confirms decentralisation within the BASE architecture and shows one of the benefits of having a decentralised architecture. Again, this is enabled by the inherent concurrency of Erlang.

One possible shortcoming in the implementation's UI is that it was developed for users that have some knowledge on holonic systems and, to a lesser extent, for general users. In future research more time should be spent on finding better ways of navigating through the UI and also incorporating what general users of this system would require of its UI. Intuitiveness is a difficult metric to evaluate, since the user's technical capabilities, experience with the UI and personal preferences can bias the evaluation.

The above discussion highlights two important concepts that can also be used outside of the BASE architecture in other holonic architectures. The first of these findings is that service-oriented communication is the way forward for complex systems that integrate many heterogeneous components. The most notable assistance provided by service-oriented interactions within holonic architectures is that it ensures less effort is spent on reconfiguring an existing system, when a new holon is added to the system. Another advantage of service-oriented communications in holonic systems is that they give more control to individual holons to accept or reject requests from other holons, making them more independent. Independence is one of the fundamental properties of holons (Koestler, 1967).

The second holonic concept applicable outside of the BASE architecture is that interfacing with the real world should be encapsulated in dedicated Interface Holons. The biggest advantage gained by this is that new interfacing devices can be added to a holonic system without requiring a single extra line of code in the rest of the system's holons. It is also easier to diagnose why data transfers between the cyber and physical world fail, because Interface Holons have their own Biographies that can be inspected.

Overall, the extensions made to the BASE architecture managed to address the objective of this thesis, namely encapsulating the complexity of many heterogeneous resources and enabling reconfigurability in the BASE architecture, using holonic principles. The extended BASE architecture was proven to be highly effective in the agricultural case study; however, this architecture would need to be applied in more industries before it can truly be classified as a generic reference architecture for complex, reconfigurable CPSs.

Stellenbosch University https://scholar.sun.ac.za

11 Conclusion and future work

This thesis developed an extended version of the BASE architecture for application in complex and reconfigurable CPSs. To guide the development of the extensions to the architecture, a set of requirements was formulated from a review of literature on CPSs, RMSs and holonic systems and evaluation of the original BASE architecture.

The extensions focussed on four aspects of the BASE architecture: the platform management components, the internal architecture of the administration shell, the cooperation between administration shells and the interfacing of cyber and real-world components. For platform management, a service registration and discovery mechanism, information loggers and a UI was developed. The refinements to the BASE architecture administration shell included new components, standardised plugins and the organisation of attribute and activity information. For the cooperation of resources, service-oriented communication was supported by generic and customisable plugin components. Cyber-physical interfacing was enabled by, and encapsulated within, Interface Holons.

The extensions to the BASE architecture were evaluated by means of a case study implementation. For the case study, the data and operations management of an educational sheep farm was selected as a representative CPS with suitable complexity and requirements for reconfigurability. The evaluation criteria were built on the requirements that guided the development of the extensions and included both quantitative and qualitative metrics. The evaluation revealed important findings, namely:

- Development productivity can be increased by generalising as many components as possible, to reduce the custom code base required.

- The extended BASE architecture is extremely reliable, since no generic or custom components failed and no inter-shell messages were lost while the implementation was deployed. The use of Erlang is considered critical to this reliability.

- The extensions made to the BASE architecture helped to fully address the original BASE architecture's scalability shortcomings. Service-oriented cooperation in the BASE architecture is not affected by the size of the system.

- Service-oriented cooperation and the use of Interface Holons are expected to greatly enhance other holonic reference architectures as well.

While this thesis represents notable progress towards applications of the BASE architecture in complex, reconfigurable CPSs, there is need and potential for further work on the following aspects:

Stellenbosch University https://scholar.sun.ac.za

- **The integration of Logistics Holons** – Logistics Holons, included in the prominent holonic reference architectures (as Activity Holons in ARTI (Valckenaers, 2019), Order Holons in PROSA (Van Brussel *et al.*, 1998) and Task Holons in ADACOR (Leitão & Restivo, 2006)), are responsible for the coordination of services provided by Resource Holons to achieve a specific goal (e.g. the production of a single product instance). If the BASE architecture is to be applied as a control system (i.e. instead of purely an information management system), the integration of Logistics Holons should be explored further.

- **Aggregation** – while the thesis showed a basic example of aggregation by aggregating sheep holons' data into their respective groups, further work is required to achieve enhanced standardisation and value addition with regards to aggregation.

- **Refinement of Interface Holons** – the implications of splitting the Interface Holons into Interface and *Gateway* Holons should be explored. Interface Holons would still be used as the BASE shells for interfacing devices, e.g. sensors. Gateway Holons would be used to represent MQTT brokers, Hypertext Transfer Protocol (HTTP) Servers etc. that are used to communicate with the physical parts of the Interface Holons.

- **Refinement of the UI** – the UI was usable, but it requires users to understand how holonic systems work. Research is required on how a holonic system's UI can make it easier for general users to use the system.

- **Deployment to the cloud** – in some sense the case study was deployed to the cloud, since it was installed on a remote computer. However, many other benefits of the cloud like Big Data, automatic scaling, data security etc. can still be utilised in the BASE architecture. Thus, future work on deploying a BASE architecture implementation in the cloud is required.

- **Comparison with other HCAs** – The BASE architecture as an implementation architecture for holons in PROSA, ADACOR and ARTI should be compared with POLLUX (Jimenez, Bekrar, Zambrano-Rey, Trentesaux & Leitão, 2017) as an implementation architecture for holons in ORCA (Pach, Berger, Bonte & Trentesaux, 2014).

The realisation of the Industry 4.0 vision is expected to rely heavily on the development and integration of CPSs. The BASE architecture, in its extended form as developed in this thesis, leverages holonic principles and a powerful software platform. As such, the BASE architecture holds great potential for the development of complex, reconfigurable CPSs that are expected to emerge in multiple domains.

Stellenbosch University https://scholar.sun.ac.za

12 References

Antonopoulos, K., Panagiotou, C., Antonopoulos, C.P. & Voros, N.S. 2019.A-FARM Precision Farming CPS Platform. in *10th International Conference on Information, Intelligence, Systems and Applications*.

Armstrong, J. (2003). Concurrency oriented programming in Erlang. Prentice Hall.

Babiceanu, R. & Seker, R., 2016. Big Data and virtualization for manufacturing cyber-physical systems: A survey of the current status and future outlook. *Computers in Industry*, 81:128-137.

Banks, A., Briggs, E., Borgendale, K. & Gupta, R., 2021. *MQTT Version 5.0*. [Online]. Available: http://docs.oasis-open.org/mqtt/mqtt/v5.0/csprd02/mqtt-v5.0-csprd02.html [Accessed: 5 July 2021].

Bellifemine, F., Caire, G. & Greenwood, G., 2007. *Developing Multi-Agent Systems with JADE*. John Wiley & Sons, Ltd.

Bengufarm. 2015. [Online] Available: https://www.bengufarm.co.za [Accessed: 23 June 2021].

Bhrugubanda, M., 2015. A Review on Applications of Cyber Physical Systems. *International Journal of Innovative Science, Engineering & Technology*, 2(6):728-730.

Bussmann, S. 1998. An agent-oriented architecture for holonic manufacturing control. in *1st International Workshop on IMS*. Lausanne: 1-12.

Caggiano, A., 2018. Cloud-based manufacturing process monitoring for smart diagnosis services. *International Journal of Computer Integrated Manufacturing*, 31(7):612-623.

Cámara, L., 2021. *Nine Nines*. [Online]. Available: https://ninenines.eu/ [Accessed: 5 July 2021].

Cardin, O., 2019. Classification of cyber-physical production systems applications: Proposition of an analysis framework. *Computers in Industry*, 104:11-21.

Carreras Guzman, N., Wied, M., Kozine, I. & Lundteigen, M., 2019. Conceptualizing the key features of cyber-physical systems in a multi-layered representation for safety and security analysis. *Systems Engineering* 23(2):189-210.

Cesarini, F., Pappalardo, V. & Santoro, C., 2008. A Comparative Evaluation of Imperative and Functional Implementations of the IMAP Protocol. *Proceedings of the 7th ACM SIGPLAN Workshop on Erlang*, 29-40.

Stellenbosch University https://scholar.sun.ac.za

Chen, B., Wan, J., Shu, L., Li, P., Mukherjee, M. & Yin, B., 2018. Smart Factory of Industry 4.0: Key Technologies, Application Case, and Challenges. *IEEE Access*, 6:6505-6519.

Chen, G., Wang, P., Feng, B., Li, Y. & Liu, D., 2019. The framework design of smart factory in discrete manufacturing industry based on cyber-physical system. *International Journal of Computer Integrated Manufacturing*, 33(1):79-101.

Chen, H., 2017. Applications of Cyber-Physical System: A Literature Review. *Journal of Industrial Integration and Management*, 02(03): 1750012.

Cheng, G., Liu, L., Qiang, X. & Liu, Y., 2016. Industry 4.0 Development and Application of Intelligent Manufacturing. *2016 International Conference on Information System and Artificial Intelligence (ISAI),* 407-410

Cimino, C., Negri, E. & Fumagalli, L., 2019. *Review of digital twin applications in manufacturing. Computers in Industry*, 113:103-130.

Derigent, W., Cardin, O. & Trentesaux, D. 2020. Industry 4.0: contributions of holonic manufacturing control architectures and future challenges. *Journal of Intelligent Manufacturing.*

Erlang -- ets. 2021. Available: https://erlang.org/doc/man/ets.html [Accessed: 1 May 2021].

Etxeberria-Agiriano, I., Calvo, I., Noguero, A. & Zulueta, E., 2012. Configurable cooperative middleware for the next generation of CPS. *2012 9th International Conference on Remote Engineering and Virtual Instrumentation*. 1-5.

Facchinetti, T. & Della Vedova, M., 2011. Real-Time Modeling for Direct Load Control in Cyber-Physical Power Systems. *IEEE Transactions on Industrial Informatics*, 7(4):689-698.

FIPA Contract Net Interaction Protocol Specification. 2021. Available: http://www.fipa.org/specs/fipa00029/XC00029F.html. [Accessed: 10 May 2021].

Gai, K. Qiu, M., Zhao, H., Sun, X. 2018. Resource management in sustainable cyber-physical systems using heterogeneous cloud computing. *IEEE Transactions on Sustainable Computing,* 3(2):60-72.

Ghobakhloo, M., 2018. The future of manufacturing industry: a strategic roadmap toward Industry 4.0. *Journal of Manufacturing Technology Management*, 29(6):910-936.

Giret, A. & Botti, V., 2004. Holons and agents. *Journal of Intelligent Manufacturing*, 15(5):645-659.

Stellenbosch University https://scholar.sun.ac.za

Gunes, V., Peter, S., Givargis, T. & Vahid, F., 2014. A Survey on Concepts, Applications, and Challenges in Cyber-Physical Systems. *KSII Transactions on Internet and Information Systems*, 8(12):134-159.

Heiss, M., Oertl, A., Sturm, M., Palensky, P., Vielguth, S. & Nadler, F., 2015. Platforms for industrial cyber-physical systems integration: contradicting requirements as drivers for innovation. *2015 Workshop on Modeling and Simulation of Cyber-Physical Energy Systems (MSCPES)*, 1-8.

Iarovyi, S., Mohammed, W., Lobov, A., Ferrer, B. & Lastra, J., 2016. Cyber–Physical Systems for Open-Knowledge-Driven Manufacturing Execution Systems. Proceedings of the IEEE, 104(5):1142-1154.

Jabeur, N., Sahli, N. & Zeadally, S., 2015. Enabling Cyber Physical Systems with Wireless Sensor Networking Technologies, Multiagent System Paradigm, and Natural Ecosystems. *Mobile Information Systems*, 2015:1-15.

Jimenez, J.F., Bekrar, A., Zambrano-Rey, G., Trentesaux, D., Leitão P., 2017, Pollux: a dynamic hybrid control architecture for flexible job shop systems, *International Journal of Production Research,* 55(15):4229-4247.

Khorrami, F., Krishnamurthy, P. & Karri, R., 2016. Cybersecurity for Control Systems: A Process-Aware Perspective. IEEE Design & Test, 33(5):75-83.

Kim, K. & Kumar, P., 2013. An overview and some challenges in cyber-physical systems. *Journal of the Indian Institute of Science*, 93(3):341-352.

Koestler, A., 1967. *The Ghost in the Machine*. London: Arkana Books.

Koren, Y., Gu, X. & Guo, W., 2018. Reconfigurable manufacturing systems: Principles, design, and future trends. *Frontiers of Mechanical Engineering*, 13(2):121-136.

Koren, Y., Heisel, U., Jovane, F., Moriwaki, T., Pritschow, G., Ulsoy, G. & Van Brussel, H., 1999. Reconfigurable Manufacturing Systems. *CIRP Annals*, 48(2):527-540.

Kruger, K. & Basson, A.H., 2018. Evaluation criteria for holonic control implementations in manufacturing systems. *International Journal of Computer Integrated Manufacturing*, 32(2):148-158.

Lee, E.A. 2015. The past, present and future of cyber-physical systems: A focus on models. *Sensors (Basel).* 15(3):4837–4869.

Lee, H., Ryu, K. & Cho, Y., 2017. A Framework of a Smart Injection Molding System Based on Real-time Data. *Procedia Manufacturing*, 11:1004-1011.

Stellenbosch University https://scholar.sun.ac.za

Lee, J., Bagheri B. & Kao, H.A. 2015. A cyber-physical systems architecture for industry 4.0-based manufacturing systems. *Manufacturing Letters*, 3:18–23.

Lee, J., Jin, C. & Bagheri, B., 2017. Cyber physical systems for predictive production systems. *Production Engineering*, 11(2):155-165.

Leitao, P., Colombo, A. & Restivo, F., 2005. ADACOR: A Collaborative Production Automation and Control Architecture. *IEEE Intelligent Systems*, 20(1):58-66.

MaQiaTTo. 2021. [Online] Available: https://www.maqiatto.com/ [Accessed: 5 July 2021].

Martinsen, K., Haga, E., Dransfeld, S., & Watterwald, L.E., 2007. Robust, Flexible and Fast Reconfigurable Assembly System for Automotive Air-brake Couplings. *Intelligent Computation in Manufacturing Engineering*, 6.

Monostori, L., Kádár, B., Bauernhansl, T., Kondoh, S., Kumara, S., Reinhart, G., Sauer, O., Schuh, G., Sihn, W. & Ueda, K., 2016. *Cyber-physical systems in manufacturing. CIRP Annals*, 65(2):621-641.

Mourtzis, D. & Vlachou, E., 2018. A cloud-based cyber-physical system for adaptive shop-floor scheduling and condition-based maintenance. *Journal of Manufacturing Systems*, 47:179-198.

Napoleone, A., Macchi, M. and Pozzetti, A., 2020. A review on the characteristics of cyber-physical systems for the future smart factories. *Journal of Manufacturing Systems*, 54:305-335.

Ngrok. 2021. [Online] Available: https://ngrok.com/ [Accessed: 5 July 2021].

Otto, J., Vogel-Heuser, B. & Niggemann, O., 2018. Automatic Parameter Estimation for Reusable Software Components of Modular and Reconfigurable Cyber-Physical Production Systems in the Domain of Discrete Manufacturing. *IEEE Transactions on Industrial Informatics*, 14(1):275-282.

Özkiziltan, D. & Hassel, A., 2020. Humans versus Machines: An Overview of Research on the Effects of Automation of Work. *SSRN Electronic Journal* [Electronic]. Available: https://dx.doi.org/10.2139/ssrn.3789992 [Accessed: 3 March 2021].

Pach, C., Berger, T., Bonte, T., Trentesaux, D., 2014. ORCA-FMS: a dynamic architecture for the optimized and reactive control of flexible manufacturing scheduling. *Computers in Industry,* 65:706-720.

Stellenbosch University https://scholar.sun.ac.za

Penas, O., Plateaux, R., Patalano, S. & Hammadi, M., 2017. Multi-scale approach from mechatronic to Cyber-Physical Systems for the design of manufacturing systems. *Computers in Industry*, 86:52-69.

Pirvu, B., Zamfirescu, C. & Gorecky, D., 2016. Engineering insights from an anthropocentric cyber-physical system: A case study for an assembly station. *Mechatronics*, 34:147-159.

Ramis Ferrer, B.R., Iarovyi, S., Lobov, A., Martinez Lastra, J. & Mohammed, W., 2020. Exemplifying the Potentials of Web Standards for Automation Control in Manufacturing Systems. *International Journal of Simulation: Systems, Science & Technology*, 17(33):262-268

Redelinghuys, A.J.H., Basson, A.H. & Kruger, K., 2019. A six-layer architecture for the digital twin: a manufacturing case study implementation. *Journal of Intelligent Manufacturing*, 31(6):1383-1402.

Ribeiro, L. & Bjorkman, M., 2018. Transitioning From Standard Automation Solutions to Cyber-Physical Production Systems: An Assessment of Critical Conceptual and Technical Challenges. *IEEE Systems Journal*, 12(4):3816-3827.

Russo, S., 2021. Service-Oriented Architecture. [Online] Umsl.edu. Available: https://www.umsl.edu/~sauterv/analysis/F2015/Service%20Oriented%20Architecture.html [Accessed: 5 March 2021].

Sanderson, D., Chaplin, J. & Ratchev, S., 2018. Conceptual Framework for Ubiquitous Cyber-Physical Assembly Systems in Airframe Assembly. *IFAC-PapersOnLine*, 51(11):417-422.

Savor, T. & Seviora, R.E. 1995. Improving the Efficiency of Supervision by Software. in Proceedings of the Real-Time Technology and Applications Symposium (RTAS '95). IEEE Computer Society, USA.

Scholze, S., Barata, J. & Stokic, D., 2017. Holistic Context-Sensitivity for Run-Time Optimization of Flexible Manufacturing Systems. *Sensors*, 17(3):455.

Shcherbakov, M., Glotov, A. & Cheremisinov, S., 2019. Proactive and Predictive Maintenance of Cyber-Physical Systems. *Studies in Systems, Decision and Control*, 263-278.

Smith, R. G., 1980. The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver. *Trans. Computers*, 29(12):1104-1113.

Sousa, P., Ramos, C. & Neves, J., 2007. Scheduling in Holonic Manufacturing Systems. *Springer Series in Advanced Manufacturing*, 167-190.

Stellenbosch University https://scholar.sun.ac.za

Sparrow, D. 2021. An architecture for the integration of human workers into an Industry 4.0 manufacturing environment. Unpublished doctoral dissertation. Stellenbosch: University of Stellenbosch.

Sparrow, D., Kruger, K. & Basson, A.H. 2021. An architecture to facilitate the integration of human workers in Industry 4.0 environments. *International Journal of Production Research*, 1-19

Sparrow, D. 2020. An architecture for the integration of human workers into an industry 4.0 manufacturing environment: Lab Report. [Online]. Available: https://sun.ac.za/mad [Accessed: 8 June 2020].

Sparrow, D., Kruger, K., Basson, A.H., 2020. Activity Lifecycle Description for Communication in Human-Integrated Industry 4.0 Environments. in *Service Oriented, Holonic and Multi-agent Manufacturing Systems for Industry of the Future*.

Svetlík, J., 2020. *Modularity of Production Systems*. in Ľubomír Šooš & Jiri Marek. Machine Tools. Rijeka: IntechOpen.

Tao, F. & Qi, Q., 2019. New IT Driven Service-Oriented Smart Manufacturing: Framework and Characteristics. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 49(1):81-91.

Tay, S.I., Lee,T.C., Hamid, N.A. & Ahmad, A.N., 2018. An Overview of Industry 4.0: Definition, Components, and Government Initiatives. *Journal of Advanced Research in Dynamical and Control Systems*, 10(14):1379-1387.

Tedeschi, S., Rodrigues, D., Emmanouilidis, C., Erkoyuncu, J., Roy, R. & Starr, A., 2018. A cost estimation approach for IoT modular architectures implementation in legacy systems. *Procedia Manufacturing*, 19:103-110.

Thoben, K., Wiesner, S. & Wuest, T., 2017. "Industrie 4.0" and Smart Manufacturing – A Review of Research Issues and Application Examples. *International Journal of Automation Technology*, 11(1):4-16.

Tran, N., Park, H., Nguyen, O. & Hoang, T., 2019. Development of a Smart Cyber-Physical Manufacturing System in the Industry 4.0 Context. *Applied Sciences*, 9(16):3325.

Twilio. 2021. *Twilio - Communication APIs for SMS, Voice, Video and Authentication*. [Online]. Available: https://www.twilio.com/ [Accessed: 1 June 2021].

Valckenaers, P. 2019. ARTI Reference Architecture – PROSA Revisited. in *Service Orientation in Holonic and Multi-Agent Manufacturing*, 803.

Stellenbosch University https://scholar.sun.ac.za

Van Brussel, H., J. Wyns, P. Valckenaers, L. Bongaerts & Peeters. P. 1998. Reference Architecture for Holonic Manufacturing Systems: PROSA. *Computers in Industry* 37: 255–274.

Vonage. 2021. [Online] Available: https://www.vonage.com/ [Accessed: 5 July 2021].

Vyatkin, V., 2007. *IEC 61499 Function Blocks for Embedded and Distributed Control Systems Design*. North Carolina: Instrumentation, Systems and Automation Society, ISA.

Wan, J., Yan, H., Suo, H. & Li, F., 2011. Advances in Cyber-Physical Systems Research. *KSII Transactions on Internet and Information Systems*, 5(11).

Yuan, X., Anumba, C. & Parfitt, K., 2015. Review of the Potential for a Cyber-Physical System Approach to Temporary Structures Monitoring. *International Journal of Architectural Research: ArchNet-IJAR*, 9(3):26.

Zhou, J., Zhou, Y., Wang, B. & Zang, J., 2019. Human–Cyber–Physical Systems (HCPSs) in the Context of New-Generation Intelligent Manufacturing. *Engineering*, 5(4):624-636.

Stellenbosch University https://scholar.sun.ac.za

Appendix A Coordinator start-up operations and variables

This appendix discusses the start-up operations and variables of the Coordinator, the new core component added to the BASE architecture digital administration shell.

Start-up operations

The order of steps implemented by the Coordinator at start-up are:

1. **Start core components**: Start the BASE shell's internal supervisors. Each supervisor will start up its reception and storage component, except for the Communication Manager's supervisor which will only start its reception, and the Plugin Supervisor which will not be started yet.

2. **Inform core components about each other:** Wait for each reception and storage component to register themselves (including their unique process address/name) with the Coordinator, and then share these addresses with the core components so that they know of each other.

3. **Start plugins:** Start up the plugin supervisor, which will start up all the plugins in the BASE shell's plugin configuration file (section 5.6.2). Each plugin must register itself with the Coordinator to be able to interact with the rest of the BASE shell and will receive the addresses of the BASE shell's five core receptions upon registration. The plugin supervisor would inform the Coordinator if a plugin failed to start so that the coordinator does not wait for that plugin.

4. **Share plugin addresses with the core components:** Once all the plugins have either registered themselves or failed to start, the Coordinator shares all the plugin addresses with all the core components so that the core components know which processes are allowed to communicate with them (concept of BASE shell privacy discussed in section 5.5).

Stellenbosch University https://scholar.sun.ac.za

Variables

CO1. **core_addresses** – contains the addresses of the core components in a key-value format with the keys being bio_recep, bio_storage, atr_recep, atr_storage, sched_recep, sched_storage, exe_recep, exe_storage, comms_recep, and the values being the process, goroutine or thread unique addresses/names.

CO2. **plugins** – contains the addresses of the active plugins in a key-value format with the keys being the plugin module names and the values being either a unique address/name or "failed" for when the active plugin could not be started or failed too many times.

CO3. **instance_sup** – the unique address/name of the instance's supervisor process.

CO4. **plugins_sup –** the unique address/name of the plugin supervisor process under which all active plugins are spawned.

Stellenbosch University https://scholar.sun.ac.za

Appendix B Standard activity data structures

Table 7 contains a summary of the data structures of standard activities in the extended BASE architecture. These activities were discussed in more detail in section 5.7. Each bullet point represents a data field. The numbers in brackets next to some of the data fields indicate the section numbers in which the data fields are explained in more detail.

Table 7: Data structures of standard activities

	Stage 1 Data	Stage 2 Data
SPA	• Service Contract (6.3)	• Pending Proposals (6.3) • Proposals (6.3) • Sub-Contracts (6.3) • Results (6.3)
RAA	• Case specific details	• Pending Proposals (6.3) • Proposals (6.3) • Sub-Contracts (6.3) • Results (6.3)
RDAA	• Observer ID • Interfacing Details ➢ Identifier Type (7.3.2) ➢ Topics (7.3.3)	• Value • Extra information

Stellenbosch University https://scholar.sun.ac.za

Appendix C Contract Net Protocol applied to holonic systems

Figure 40 shows a sequence diagram, illustrating how the CNP is applied in a holonic system. The CNP is initialised by a service requesting holon, who sends a CFP to all possible service provision holons. These service provision holons can send a proposal if they are interested in providing the service or refuse if they are not interested. The service requesting holon selects the most suitable service provision holon among the proposals and accepts its proposal, while rejecting all the other proposals. Hereafter the service provision holon can send as many *inform-result* messages as is necessary. When the service provision holon fails to provide the service it sends a *failure* message. When the service provision holon has successfully completed providing a service, it sends an *inform-done* message.

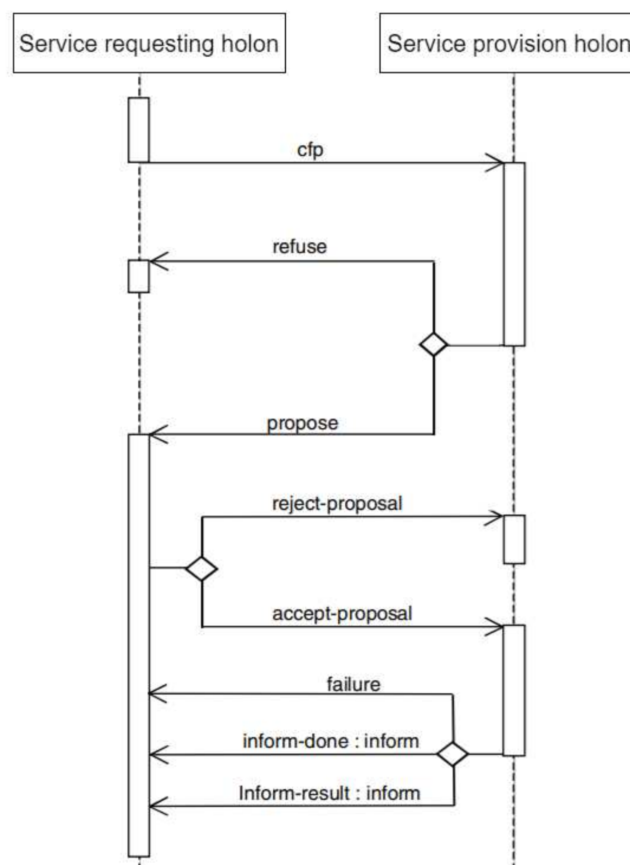


Figure 40: Contract net protocol - figure adapted from the FIPA CNP diagram (FIPA Contract Net Interaction Protocol Specification, 2021)

Stellenbosch University https://scholar.sun.ac.za

Appendix D Feeding sensor Interface Module

This appendix shows the source code of the feeding sensor's Interface Module, which had to connect as an MQTT client to a cloud MQTT broker to receive data from the physical feeding sensor on the educational sheep farm.

```
-module(rfid_sheep_feeder_interface_mod).
-define(TOPIC,<<"djvniekerk@sun.ac.za/pi_temp">>).%change to <<"djvniekerk@sun.ac.za/rfid_sheep_feeder">> for real thing. pi_temp just for testing
with own raspberry pi
-define(REPLY_TOPIC,<<"djvniekerk@sun.ac.za/bf_ack">>).
-define(BROKER,"maqiatto.com").
-define(PORT,1883).
%% ====================================================================
%% API functions
%% ====================================================================
-export([get_gateway_details_and_min_time_between_failures/1,start_interface/3,valid_identifier/2,get_interface_info/1]).
%% ====================================================================
%% Internal functions
%% ====================================================================
get_gateway_details_and_min_time_between_failures(_Id)->%%used by sensor_ep
        {ok,#{<<"Interface type">>=>"MQTT",<<"Broker">>=>?BROKER,<<"TCP Port">>=>?PORT},10000}.
get_interface_info(_Id)->%%used by sensors_serv_prov_mod
        {ok,Topics} = cp_interface_creation:add_observer_topic(#{}, "Eating", 95, "g"),
        {ok,Resources} = cp_interface_creation:add_to_resources(observer, #{}, "Sheep", all, "EID", Topics),
        {ok,Topics2} = cp_interface_creation:add_observer_topic(Topics, "Ambient temperature", 99, "C"),
        {ok,Resources2} = cp_interface_creation:add_to_resources(observer,Resources, "Houses", [<<"RFID Feeder Building">>], "Location", Topics2),
        {"Primary:Measures how much individual sheep eat. Secondary:Measures ambient temperature of building containing rfid feeding
sensor",Resources2}.

valid_identifier("EID",Identifier)->
        case custom_erlang_functions:is_binary_string(Identifier) of
                true->
                        true;
                _->
                        false
        end;
valid_identifier("Location",Identifier)->
        case custom_erlang_functions:is_binary_string(Identifier) of
```

Stellenbosch University https://scholar.sun.ac.za

```
                true->
                        true;
                _->
                        false
        end.

start_interface(MyPlugin,_MyId,ReceptionMap)->
        {Pid,_Ref} = spawn_monitor(fun()->simulate(MyPlugin,100)end),
        {ok,Pid}.

%%Internal
start_emqtt(MyPlugin,ReceptionMap,MessageSent)->
        io:format("\nStarting emqtt"),
        {ok, ConnPid} = emqtt:start_link([{host,?BROKER},{username,"djvniekerk@sun.ac.za"},{password,"BASE_meets_daisy"}]),
        unlink(ConnPid),%do not want this process to fail with ConnPid
        ok = try_to_connect(ConnPid,MyPlugin,ReceptionMap,MessageSent),

        SubOpts = [{qos, 1}],
        {ok, _Props, _ReasonCodes} = emqtt:subscribe(ConnPid, #{}, [{<<"djvniekerk@sun.ac.za/pi_temp">>, SubOpts}]),
        receive_data(ConnPid,MyPlugin,base_time:now(),ReceptionMap,MessageSent).

try_to_connect(ConnPid,MyPlugin,ReceptionMap,MessageSent)->
        io:format("\nTrying to connect to ~p",[?BROKER]),
        %io:format("\nConnPid alive ~p",[is_process_alive(ConnPid)]),
        case is_process_alive(ConnPid) of
                true->
                        Res = emqtt:connect(ConnPid),
                        case Res of
                                {ok,_Props}->
                                        io:format("\nConnected"),
                                        custom_erlang_functions:myGenServCall(MyPlugin, sensor_running),
                                        ok;
                                _->
                                        io:format("\nRes when trying to connect to ~p: ~p",[?BROKER,Res]),
                                        timer:sleep(5000),
                                        io:format("\nGonna restart"),
                                        try_to_connect(ConnPid,MyPlugin,ReceptionMap,MessageSent)
                        end;
                _->
                        start_emqtt(MyPlugin,ReceptionMap,MessageSent)
        end.
%60
```

Stellenbosch University https://scholar.sun.ac.za

```
remove_all_spaces(RawEID)->
        NewS = re:replace(RawEID, "(\\s)", "", [global,{return,list}]),
        list_to_binary(NewS).

receive_data(ConnPid,MyPlugin,LastT,ReceptionMap,MessageSent)->
        %io:format("\nWaiting for packages from maqiatto"),
        receive
            {disconnect, ReasonCode, Properties} ->
                io:format("\nNETW ERROR: Recv a DISCONNECT packet from ~p for rfid_feeder - ReasonCode: ~p, Properties: ~p~n",
[?BROKER,ReasonCode, Properties]),
                        custom_erlang_functions:myGenServCall(MyPlugin, {sensor_down,lists:flatten(io_lib:format("~p", [ReasonCode]))}),
                        try_to_connect(ConnPid,MyPlugin,ReceptionMap,MessageSent),
                        SubOpts = [{qos, 1}],
                        {ok, _Props, _ReasonCodes} = emqtt:subscribe(ConnPid, #{}, [{<<"djvniekerk@sun.ac.za/pi_temp">>, SubOpts}]),
                        receive_data(ConnPid,MyPlugin,LastT,ReceptionMap,MessageSent);
            {publish, PUBLISH} ->
                        PayLoad = maps:get(payload,PUBLISH),
                %io:format("\nRecv a PUBLISH payload: ~p~n", [PayLoad]),
                        PayLoadMap = jsone:decode(PayLoad,[{object_format, map}]),
                        RawEID = maps:get(<<"eid">>,PayLoadMap),
                        EID = remove_all_spaces(binary_to_list(RawEID)),
                        StartDateAndTime = binary_to_list(maps:get(<<"start_date">>,PayLoadMap)),
                        [StartDate|[StartTime]] = string:split(StartDateAndTime, " "),
                        StartInt = base_time:date_and_time_strings_to_base_time(StartDate, StartTime),
                        EndDateAndTime = binary_to_list(maps:get(<<"end_date">>,PayLoadMap)),
                        [EndDate|[EndTime]] = string:split(EndDateAndTime, " "),
                        EndInt = base_time:date_and_time_strings_to_base_time(EndDate, EndTime),
                        case EndInt of
                                LastT->
                                        FinalT = EndInt+1;
                                _->
                                        FinalT = EndInt
                        end,
                        FinalDateAndT = base_time:base_time_to_date_and_time_string(FinalT),
                        [FinalDate|[FinalTime]] = string:split(FinalDateAndT, " "),
                        Duration = EndInt-StartInt,
                        AmountEaten = maps:get(<<"eaten">>,PayLoadMap),
                        DataMap = #{"Eating"=>#{<<"Date">>=>FinalDate,<<"Time">>=>FinalTime,<<"Value">>=>AmountEaten,<<"Duration">>=>Duration}},
                        Temperature = maps:get(<<"temp">>,PayLoadMap),
                        TempDataMap = #{"Ambient temperature"=>#{<<"Date">>=>FinalDate,<<"Time">>=>FinalTime,<<"Value">>=>Temperature}},
                        case is_process_alive(MyPlugin) of
                                true->
```

Stellenbosch University https://scholar.sun.ac.za

```
                                        ok = custom_erlang_functions:myGenServCall(MyPlugin,{new_data,EID,DataMap}),
                                        ok = custom_erlang_functions:myGenServCall(MyPlugin,{new_data,<<"RFID Feeder Building">>,TempDataMap}),
                                        ok = emqtt:publish(ConnPid, ?REPLY_TOPIC, #{}, PayLoad, [{qos, 0}]),%to inform sensor that data has been
received
                                        receive_data(ConnPid,MyPlugin,FinalT,ReceptionMap,false);
                                _->
                                        exit(ConnPid,kill)
                        end;
                _->
                        io:format("\nEMQTT ERROR:Received unknown from emqtt"),
                        receive_data(ConnPid,MyPlugin,LastT,ReceptionMap,MessageSent)
        after 60000->
                case is_process_alive(MyPlugin) of
                        true->
                                case is_process_alive(ConnPid) of
                                        true->
                                                %%check if human holon needs to be informed
                                                {ok,MaxMinutesWithNoActivity} = atr_api:get_attribute_value(maps:get(atr,ReceptionMap), "Max
minutes with no activity"),
                                                case (base_time:now()>=(LastT+MaxMinutesWithNoActivity*60)) and not(MessageSent) of
                                                        true->
                                                                {ok,Humans} = doha_api:get_bcs_by_types(["Human workers"]),

        try_to_inform_each_human(Humans,maps:get(comms,ReceptionMap),MaxMinutesWithNoActivity),
                                                                NewMessageSent=true;
                                                        _->
                                                                NewMessageSent = MessageSent
                                                end,
                                                receive_data(ConnPid,MyPlugin,LastT,ReceptionMap,NewMessageSent);
                                        _->
                                                start_emqtt(MyPlugin,ReceptionMap,MessageSent)
                                end;
                        _->%%if plugin dead kill this process and ConnPid
                                case is_process_alive(ConnPid) of
                                        true->
                                                exit(ConnPid,kill);
                                        _->
                                                ok
                                end
                end
        end.
```

Stellenbosch University https://scholar.sun.ac.za

Appendix E Interface Holon attributes

Attribute ID	Attribute Type	Attribute Context	Attribute Value (Example)	Usage
Gateway details	Personal	Key-value pairs with at least the interface type as key and any other details as separate keys	"{"Interface type": "MQTT", "Broker": "Maqiatto", "Port":1883}"	Provides details about the method of communication between the BASE shell and its physical device.
Start-up date and time	Personal	Date and time	"2020-05-23 14:02:34"	Used to have a reference point of how long the device has been deployed when looking at its other attributes like "Nr of device failures".
Attached to resource with id	Relational	BASE ID	"Camp 12"	An interfacing device's operation might be influenced by the resource it is attached to.
Clients (only Observers)	Relational	Key-value pairs with client identifiers as keys and contracts as values	"{"Sheep14": Sheep14Contract, "Sheep23": Sheep23Contract,	Used to deliver messages from the physical world to the correct BASE shells.

Stellenbosch University https://scholar.sun.ac.za

			“Camp12”: Camp12Contract}”	
Maximum minutes with no activity	Management	Can be used by the Interface Process to inform any available humans in the system that there might be a problem with the device	360	Can be used by the Interface Process to inform any available humans in the system that there might be a problem with the device
Interfacing rate (messages entries per hour)	Communication	Every message shared to/from the physical world is regarded as a message	284.52	Used to track how busy an interfacing device is.
Nr messages delivered (including rejected)	Communication	Every message shared to/from the physical world is regarded as a message.	2746	Used to see how well an interfacing device is performing
Nr deliveries rejected	Communication	Rejected messages are messages that were delivered successfully, but after which a “rejected” or “error” reply was received	12	
Last delivery rejected	Communication	Date and time	“2021-03-04 02:04:08”	Used to easily diagnose the last error that occurred if there was any.

Stellenbosch University https://scholar.sun.ac.za

<table>
<tr><td>Device status</td><td>Device state</td><td>“Running”, “Down” or “pending”</td><td>“Running”</td><td>Gives a live update of the status of the device so that it can be addressed if necessary.</td></tr>
<tr><td>Nr of device failures</td><td>Device state</td><td>A failure is recorded when a message is received from the device that it is down or when the device cannot be reached.</td><td>3</td><td>Gives an overview of the device’s reliability so that it can be replaced if necessary.</td></tr>
<tr><td>Last device failure</td><td>Device state</td><td>Date and time</td><td>“2021-04-23 12:34:41”</td><td>Used to see how long ago the device had a failure, because even if it might have many failures, all of them might be from very long ago, making them less relevant.</td></tr>
<tr><td>Interface process status</td><td>Interface process state</td><td>“Running”, “Down” or “pending”</td><td>“Running”</td><td rowspan="3">These three attributes are used to represent the state of the Interface Process. The Interface Process is the only part that needs to be custom developed for every Interface Holon and can thus be very prone to errors. These three attributes help diagnose these errors and see if any major improvements are necessary to this Interface Process.</td></tr>
<tr><td>Nr interface process failures</td><td>Interface process state</td><td>A failure is recorded when a message is received from the device that it is down or when the device cannot be reached.</td><td>3</td></tr>
<tr><td>Last interface process failure</td><td>Interface process state</td><td>Date and time</td><td>“2021-04-23 12:34:41”</td></tr>
</table>

Stellenbosch University https://scholar.sun.ac.za

Appendix F Case study holon attributes

Sheep

Attribute ID	Attribute Type	Attribute Context	Attribute Value	Usage
Date of birth	Personal	“Date”	“2017-08-23”	Represents the sheep’s age
EID	Personal	“Electronic ID of the tag attached to this resource”	“964001031059312”	Used by any RFID scanning equipment like the RFID Feeding Sensor
Gender	Personal	“Ram, ewe or pending”	“Ewe”	These five attributes can potentially be used to make predictions based on the sheep’s genetic data. This was not done in this case study, but the attributes were included nonetheless.
Dam	Personal	“Database name” or “BASE ID” or “pending”	“Sheep 63”	
Sire	Personal	“Database name” or “BASE ID” or “pending”	“Sheep 86”	

Stellenbosch University https://scholar.sun.ac.za

Offspring	Personal	“Key-value pairs with database names as keys and ids as values”	“Sheep 6”	
Breed	Personal	“Dorper, Merino, etc.”	“Dorper”	
Camp	Relational	“BASE ID”	“Camp 12”	These four attributes represent the relations the sheep has, i.e., in what camp it is, what group it is part of, from which feeders it eats and what sensors are attached to it.
Group	Relational	“BASE ID”	“Group 12”	
Feeders	Relational	“List of BASE IDs”	[“Feeder 1”, “Feeder 4”]	
Attached Sensors	Relational	“List of BASE IDs”	[“Sensor 1”, “Sensor 2”]	
Max hours between eating activities	Management	Used to notify all humans in the system when this sheep has not eaten for a long time	12	Used to notify all humans in the system when this sheep has not eaten for a long time

Stellenbosch University https://scholar.sun.ac.za

Case studies	Management	Stores the start and end dates of the different case studies that the sheep is part of	"{"Case study X":CaseStudyDetails, "Case study Y": CaseStudyDetails}	Stores the start and end dates of the different case studies that the sheep is part of
Healthy	Health	" 'false' or date since sick"	"2021-07-25"	These four attributes were not used in the case study, but are more examples of the type of properties of a sheep that must be included in its attributes.
On heat	Reproduction	"false or date since on heat"	"false"	
Pregnant	Reproduction	" 'false' or date since sick"	"2021-07-25"	
In labor	Reproduction	"false or date since on heat"	"false"	
Overall weight stats	Weight (kg)	Calculated from "Weight (RDAA)" activities	"{max:50, min:25, avg: 47}"	These attributes were all specific to the case study and used to show the data about each sheep relevant to the case study, i.e., statistics about its eating, weight, weight gain and FCR.
Weight stats since last relocation	Weight (kg)	Calculated from "Weight (RDAA)" activities	"{max:50, min:25, avg: 47}"	

Stellenbosch University https://scholar.sun.ac.za

Overall gain stats	Gain (kg/day)	Calculated from “Weight (RDAA)” activities	“{max:1.2, min:-0.7, avg: 0.3}”	Continue: These attributes were all specific to the case study and used to show the data about each sheep relevant to the case study, i.e., statistics about its eating, weight, weight gain and FCR.
Weight stats since last relocation	Gain (kg/day)	Calculated from “Weight (RDAA)” activities	“{max:1.2, min:-0.7, avg: 0.3}”	
Last 24h eating	Eating (g)	Calculated from “Eating (RDAA)” activities	2434	
Overall eating stats	Eating (g/day)	Calculated from “Eating (RDAA)” activities	“{max:3600, min:250, avg: 2100}”	
Eating stats since last relocation	Eating (g/day)	Calculated from “Eating (RDAA)” activities	“{max:3600, min:250, avg: 2100}”	
Overall eating	Eating duration (s)	Calculated from “Eating	“{max:360, min:25, avg: 125}”	

Stellenbosch University https://scholar.sun.ac.za

duration stats		(RDAA)" activities		
Overall eating frequency stats	Eating frequency (meals per day)	Calculated from "Eating (RDAA)" activities	"{max:9, min:2, avg: 6}"	Continue: These attributes were all specific to the case study and used to show the data about each sheep relevant to the case study, i.e., statistics about its eating, weight, weight gain and FCR.
Overall eating stats	FCR (kg_eaten/kg_gained)	Calculated from "Eating (RDAA)" and "Weight (RDAA)" activities	"{max:42, min:1.4, avg: 8.4}"	
Eating stats since last relocation	FCR (kg_eaten/kg_gained)	Calculated from "Eating (RDAA)" and "Weight (RDAA)" activities	"{max:42, min:1.4, avg: 8.4}"	

Stellenbosch University https://scholar.sun.ac.za

Group:

Attribute ID	**Attribute Type**	**Attribute Context**	**Attribute Value**	**Usage**
Camp	Relational	"BASE ID"	"Camp 12"	These four attributes represent the relations the group has, i.e., in what camp it is, what group it is part of, from which feeders it eats and what sensors are attached to it.
Sheep	Relational	"List of BASE IDs"	["Feeder 1", "Feeder 4"]	
Feeders	Relational	"List of BASE IDs"	["Feeder 1", "Feeder 4"]	
Number of sheep	Relational	Integer	23	
Age boundaries (months)	Management	Restricts which sheep are allowed in this group	[12, 48]	These four attributes are used to restrict what sheep are allowed in this group based on their attributes
Avg food intake boundaries (grams)	Management	Restricts which sheep are allowed in this group	[700, 3200]	

Stellenbosch University https://scholar.sun.ac.za

Genders allowed	Management	Restricts which sheep are allowed in this group	"all" or "Rams" or "Ewes"	
Weight boundaries (kg)	Management	Restricts which sheep are allowed in this group	[30, 45]	
Case studies	Management	Stores the start and end dates of the different case studies that the group is part of	"{"Case study X":CaseStudyDetails, "Case study Y": CaseStudyDetails}	Stores the start and end dates of the different case studies that the group (and its sheep) is part of
Overall weight stats	Average Sheep's Weight (kg)	Calculated from "Weight (RDAA)" activities	"{max:50, min:25, avg: 47}"	These attributes were all specific to the case study and used to show the data about each group's sheep relevant to the case study, i.e., statistics about the average sheep's eating, weight, weight gain and FCR.
Weight stats since last relocation	Average Sheep's Weight (kg)	Calculated from "Weight (RDAA)" activities	"{max:50, min:25, avg: 47}"	

Stellenbosch University https://scholar.sun.ac.za

Overall gain stats	Average Sheep's Gain (kg/day)	Calculated from "Weight (RDAA)" activities	"{max:1.2, min:-0.7, avg: 0.3}"	Continue: These attributes were all specific to the case study and used to show the data about each group's sheep relevant to the case study, i.e., statistics about the average sheep's eating, weight, weight gain and FCR.
Weight stats since last relocation	Average Sheep's Gain (kg/day)	Calculated from "Weight (RDAA)" activities	"{max:1.2, min:-0.7, avg: 0.3}"	
Last 24h eating	Average Sheep's Eating (g)	Calculated from "Eating (RDAA)" activities	2434	
Overall eating stats	Average Sheep's Eating (g/day)	Calculated from "Eating (RDAA)" activities	"{max:3600, min:250, avg: 2100}"	
Eating stats since last relocation	Average Sheep's Eating (g/day)	Calculated from "Eating (RDAA)" activities	"{max:3600, min:250, avg: 2100}"	
Overall eating	Average Sheep's Eating duration (s)	Calculated from "Eating	"{max:360, min:25, avg: 125}"	

Stellenbosch University https://scholar.sun.ac.za

duration stats		(RDAA)" activities		
Overall eating frequency stats	Average Sheep's Eating frequency (meals per day)	Calculated from "Eating (RDAA)" activities	"{max:9, min:2, avg: 6}"	
Overall eating stats	Average Sheep's FCR (kg_eaten/kg_gained)	Calculated from "Eating (RDAA)" and "Weight (RDAA)" activities	"{max:42, min:1.4, avg: 8.4}"	
Eating stats since last relocation	Average Sheep's FCR (kg_eaten/kg_gained)	Calculated from "Eating (RDAA)" and "Weight (RDAA)" activities	"{max:42, min:1.4, avg: 8.4}"	

Stellenbosch University https://scholar.sun.ac.za

Camp:

Attribute ID	Attribute Type	Attribute Context	Attribute Value	Usage
Size	Personal	“LxW (m)”	“6x5m”	These five attributes were not used for anything during the case study, but were still included to give an example of the type of attributes that a camp holon would have.
Floor material	Personal	“Text description”	“Wood with grass”	
Roof	Personal	“true/false”	“true”	
Solid walls	Personal	“true/false”	“false”	
Location	Personal	“A GPS coordinate or a description”	“-32.456234 28.837452”	
EID	Personal	“Electronic tag number in string format if the camp has one”	“9434385729534”	Sometimes camps have RFID tags attached to their gates to easily identify them when walking around with an RFID scanner
Sheep limit	Management	“The maximum number of sheep allowed in this camp”	25	This is used to manually or intelligently (based on size) limit the number of sheep a camp can have.
Sheep	Relational	“List of BASE IDs”	[“Sheep 1”, “Sheep 2”]	

Stellenbosch University https://scholar.sun.ac.za

Feeders	Relational	“List of BASE IDs”	[“Feeder 1”, “Feeder 2”]	Relational attributes of the camp showing what sheep, feeders and sensors are in it. Continue: Relational attributes of the camp showing what sheep, feeders and sensors are in it.
Sensors	Relational	“List of BASE IDs”	[“Sensor 1”, “Sensor 2”]	
Group	Relational	“BASE ID”	“Group 3”	
Number of sheep	Relational	Integer	22	

Feeders:

Attribute ID	**Attribute Type**	**Attribute Context**	**Attribute Value**	**Usage**
EID	Personal	“Electronic tag number in string format if the feeder has one”	“9434385729534”	Sometimes feeders have RFID tags attached to their gates to easily identify them when walking around with an RFID scanner
Current Food Product ID	Management	“ID of one of the products in the sheep food warehouse”	“Meadow X345”	This is used by sheep to add to the post-execution data of their recorded eating data and must be changed via the UI whenever new food is put into a feeder.
Camp	Relational	“BASE ID”	“Camp 3”	Represents the camp that the feeder is in.

Stellenbosch University https://scholar.sun.ac.za

Humans:

Attribute ID	Attribute Type	Attribute Context	Attribute Value	Usage
Date of birth	Personal	"Date"	"1997-11-18"	These four attributes were not used for anything in the case study and are just examples of what type of attributes a human holon would have.
Gender	Personal	"Male, Female or pending"	"Male"	
Height (cm)	Personal	"cm"	181	
Weight (kg)	Personal	"kg"	66	
Cell phone number	Management	"27*******"	"27836566942"	This attribute is used by the human holon to tell the Vonage SMS holon to which cell phone number an SMS must be sent if the human holon needs to communicate with its physical human.

Stellenbosch University https://scholar.sun.ac.za

Appendix G Python MQTT client for feeding sensor

The appendix shows the Python code used to enable the feeding sensor to share its data over an MQTT broker. The comments in the code can be used to understand how this was done.

```
#This version waits for published messages to be received by the base factory, not just by #the broker
Explanation: When new data is recorded the pi will try and publish the data. #The client subscribes for a 'bf_ack' that will be published by the base factory when it #receives the data. If this bf_ack is not received after 5 seconds because of a broker #error or because the factory is offline the data is kept in the que and the publish is #not successful. If not successful the data is anyways kept in a que in a text file which #is only removed once the data has #been published successfully. A new thread will be #created for an alarm that will go off after 60 seconds to que a retry. This will continue #until data can be published. When new data comes in with old data still in que. The new #data is just added to the que and the pi again retries to publish all data in que. If #unsuccessful it is not needed to start a new alarm. There will never exist more than one #retry alarm thanks to the global variable BFRetryAlarm. If an alarm goes off when there #is no data in que anymore because it was successfully published when new data came in #which triggered a retry, the alarm is just ignored. One more important feature is that no #more than one client will ever be created. When new data comes in while the system is #busy trying to publish other data, the new data is added to NewBFDataQue. Once the #current publish is done it will see if any new datas came in while it was busy trying to #publish. If there did, it will add any unpublished data left over with the new data and #make another publish attempt. The scenario in this last paragraph is not expected to #happen often. The only scenario where this might happen is when two different feeders are #sending data to the pi just after each other, but still this will not be a problem.

#How to use:
#1. Call start when pi is started up (can be seperate thread if you want)
#2. Call #new_data(EID::string,StartDate::string,EndDate::string,Temp::number,Eaten::number)
#when new data available (can be seperate thread if you want)
import paho.mqtt.client as mqtt
import time
import json
import os
import threading

#GLOBAL VARIABLES - BF used to seperate from other global variables maybe already
implemented to prevent global variable clashes
BFClient = "pending"
NewBFDataQue = []
BFRetryAlarm = False #if true a second alarm won't be started
BFPackageReceived = "none"
```

Stellenbosch University https://scholar.sun.ac.za

```
#---

#HELPER FUNCTIONS:
def is_json(myjson):
  try:
    json_object = json.loads(myjson)
  except ValueError as e:
    return False
  return True
#---

#retry_alarm is started as a seperate thread when data could not be published
#successfully. When new data comes in before this alarm goes off the unsuccessfull data is #published
before the new data. If the publish fails again new data is just added to the #que
and another retry_alarm is started
def retry_alarm():
    global BFRetryAlarm
    BFRetryAlarm = True
    #print("New alarm started")
    time.sleep(60)
    BFRetryAlarm = False
    #print("Alarm going off")
    new_data(0,0,0,0,0)

#This function is automatically called when the system manages to connect
def on_connect(client, userdata, flags, rc):
    if rc==0:
        client.connected_flag=True
        #print("connected")
    else:
        #print("Could not connect. Return code = ",rc)
        client.bad_connection_flag=True

 #This function is automatically called when the system receives a published message
def on_message(client,userdata,message):
    global BFPackageReceived
    if (BFClient==client)and(message.topic=="djvniekerk@sun.ac.za/bf_ack"):
        BFPackageReceived = message.payload.decode("utf-8")
        #print("payload rec: ",message.payload.decode("utf-8"))

def GetQue():
    Que = []
    if os.path.isfile("base_factory_que.txt") ==True:
        with open('base_factory_que.txt','r') as filehandle:
            filecontents = filehandle.readlines()
```

```
            for line in filecontents:
                if is_json(line):
                    Que.append(line.strip())
            filehandle.close()
    else:
        #print("Que file does not exist yet. Gonna create now")
        f = open('base_factory_que.txt','x')
    return Que

def SaveQue(Que):
    with open('base_factory_que.txt','w') as filehandle:
        filehandle.writelines("%s\n"% e for e in Que)
        filehandle.close()

#Called when system booted up to send any old data still in the que file.
def start():
    global NewBFDataQue
    Que = GetQue()
    if len(Que)>0:
        NewBFDataQue = ["start"]
        try_to_publish_que(Que)

#Called when new data available in the system
def new_data(EID,StartDate,EndDate,Temp,Eaten):
    global NewBFDataQue
    if EID==0:#if EID==0 the call was from the retry alarm and there is not any new data
        NewDataJson = "none"
    else:
        NewData = {"eid":EID,"start_date":StartDate,"end_date":EndDate,"temp":Temp,"eaten":Eaten}
        NewDataJson = json.dumps(NewData)
    if len(NewBFDataQue)==0:
        NewBFDataQue.append(NewDataJson)
        if not(EID==0):#if EID==0 the call was from the retry alarm and there is not any new data
            Que = GetQue()
            Que.append(NewDataJson)
            SaveQue(Que)#save before starting publish
            try_to_publish_que(Que)
        else:#call from alarm
            Que = GetQue()
            if len(Que)>0:
                try_to_publish_que(Que)
            else:
                NewBFDataQue.pop(0)
                if len(NewBFDataQue)>0:#in case another new data came in while alarm was processed
                    NewQue = NewBFDataQue.copy()
```

Stellenbosch University https://scholar.sun.ac.za

```
                    NewBFDataQue = ["busy"]
                    #print("NewQue")
                    #(NewQue)
                    try_to_publish_que(NewBFDataQue)
    else:
        #print("Still busy publishing data. Queing this request")
        #not saving data in file to ensure no clashes between two threads trying to open or save
        #to a file
        if EID==0:#if the call was from retry_alarm, do not append to NewBFDataQue
            global BFRetryAlarm
            if not(BFRetryAlarm):#new alarm only started if no alarms exists yet.
            #One alarm triggers old past failed data to be pushed
                tRetry = threading.Thread(target = retry_alarm,args = [])
                tRetry.start()
        else:
            NewBFDataQue.append(NewDataJson)

def try_to_publish_que(Que):
    SQue = Que.copy()
    #print("Starting que:")
    #print(Que)
    mqtt.Client.connected_flag = False
    mqtt.Client.bad_connection_flag = False
    broker = "maqiatto.com"
    port = 1883
    client = mqtt.Client()
    global BFClient
    BFClient = client
    client.on_connect = on_connect
    client.on_message = on_message
    client.loop_start()
    #print("Connecting to broker",broker)
    client.username_pw_set(username = "djvniekerk@sun.ac.za",password="BASE_meets_daisy")
    nrConnectionAttempts = 0
    ConnectionAttemptMade = False
    while nrConnectionAttempts<3:
        try:
            client.connect(broker,port)
            ConnectionAttemptMade = True
            break
        except:
            #print("Can't connect. No internet connection or broker down")
            nrConnectionAttempts = nrConnectionAttempts+1
            time.sleep(1)
```

Stellenbosch University https://scholar.sun.ac.za

```
    if ConnectionAttemptMade:    #only waiting for acknowledge if connection attempt was made
        WaitCount=0
        Publish = True
        while not client.connected_flag :
            #print("waiting")
            WaitCount = WaitCount+1
            if (WaitCount>5):
                #print("Did not get connection acknowledge from broker after 5 seconds")
                Publish = False
                break
            else:
                time.sleep(1)
                if client.bad_connection_flag:
                    #print("Some connection error.")
                    Publish = False
                    break

        if Publish==True:  #only trying to publish if connection attempt was made and an
        #acknowledge was received
            client.subscribe("djvniekerk@sun.ac.za/bf_ack",qos=0)
            #print("publishing data in que")
            Count=0
            for d in Que:
                if (Count>10):#no point in trying to publish if previous failed
                    break
                msg_info = client.publish("djvniekerk@sun.ac.za/pi_temp",d,qos=2)
                global BFPackageReceived
                Count=0
                #print("Tried to publish ",d)
                while True:
                    if(BFPackageReceived==d):
                        #print("BASE received package")
                        SQue.remove(d)
                        break
                    else:
                        time.sleep(1)
                        Count=Count+1
                        if (Count>10):
                            #print("BASE did not receive package")
                            break
            client.unsubscribe("djvniekerk@sun.ac.za/bf_ack")
        client.loop_stop()#if disconnected after connection was already established and
data already published this will take 2mins to execute
        #print("stopped loop")
        try:
```

Stellenbosch University https://scholar.sun.ac.za

```
            #print("trying to disconnect")
            client.disconnect()
            #print("done")
        except:
            pass
            #print("Was not needed to disconnect")
    else:
        #print("Could not connect to broker after 3 connection attempts. Giving up now")
        client.loop_stop()
        try:
            client.disconnect()
        except:
            pass
            #print("Was not needed to disconnect")

    if len(SQue)>0:
        global BFRetryAlarm
        if not(BFRetryAlarm):#new alarm only started if no alarms exists yet. One alarm triggers
old past failed data to be pushed
            tRetry = threading.Thread(target = retry_alarm,args = [])
            tRetry.start()

    #print("Ending que")
    #print(SQue)
    global NewBFDataQue
    NewBFDataQue.pop(0)
    if len(NewBFDataQue)>0: #starting pending 'new datas' together with failed publishes
        for e in NewBFDataQue:
            SQue.append(e)
        NewBFDataQue = ["busy"]
        #print("NewQue")
        #print(SQue)
        SaveQue(SQue)
        try_to_publish_que(SQue)
    else:
        SaveQue(SQue)#always saving que at end
```

Stellenbosch University https://scholar.sun.ac.za

Appendix H User Interface

Figures 41 to 47 show screenshots from the BASE architecture implementation's UI. Figures 48 and 49 show examples of Excel files that could be downloaded by students for their experiments on the sheep farm

Note to examiners: This user interface can be used to access the BASE architecture implementation by going to http://base-sheep-farm.eu.ngrok.io/ and entering either "internal" or "external" (depending on type of examiner) as both the username and password.

H.1 Screen shots of the user interface

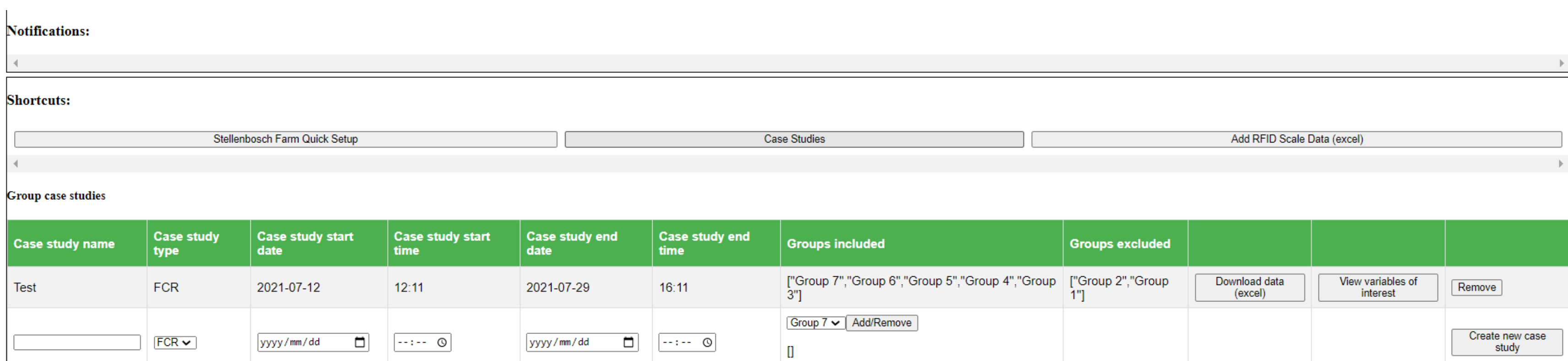


Figure 41: UI home page

Figure 41 shows the home page after the *Case Studies* Button was clicked. The *Stellenbosch Farm Quick Setup* button was used to quickly add the sheep, groups, feeders, camps and Interface Holons from an Excel file. The *Add RFID Scale Data (excel)* was used to quickly add new weight data to the sheep scale holon's schedule (to share with sheep holons) without needing to go to Resources -> Sensors -> RFID Sheep Scale. The *Case Studies* button opens a table (with the green header in figure 41) where students can add and

Stellenbosch University https://scholar.sun.ac.za

remove case studies, view live data in the browser about the variables of interest for each case study, and download data about each case study. The three buttons on the home page under *Shortcuts* and their functionalities were the only custom development required for the sheep farm. The rest of the functionalities of the UI is completely generic. Figure 42 shows the *Users* page where users of the system can be viewed, added and removed by administrators (permission level = 1), and where each user can change their password.

Figure 42: UI user administration page

Stellenbosch University https://scholar.sun.ac.za

igure 43 shows the UI after clicking on *Resources->Sheep->Get all.* The UI shows all the sheep resources, including their attributes. The headers in the table can be clicked to order the resources according to an attribute (will order alphabetically or numerically). Figure 44 shows all the Observers in the system, i.e., the sheep scale and the feeding sensor.

Home | Users | Resources | Holon Interactions | Sheep

Get all | Get by ids | Add new

	Personal							Relational				Management			Weight gain (kg/day)
IDs	Offspring	Date of birth	EID	Dam	Gender	Sire	Breed	Sensors	Feeders	House	Group	Set food content	Case studies	Max hours between eating activities	Gain stats since last relocation
VM9	{}	pending	964001031059312	pending	pending	pending	SAVM	[]	["Feeder 6"]	House 6	Group 6	{}	{}	24	{"avg":"pending","max":"pending","min":
VM8	{}	pending	964001031058998	pending	pending	pending	SAVM	[]	["Feeder 6"]	House 6	Group 6	{}	{}	24	{"avg":"pending","max":"pending","min":
VM7	{}	pending	964001031059972	pending	pending	pending	SAVM	[]	["Feeder 6"]	House 6	Group 6	{}	{}	24	{"avg":"pending","max":"pending","min":

Figure 43: UI resources page showing all sheep

	Personal		Relational		Management	BC	Data delivery				Interface state			Sensor state		
IDs	Interface details	Startup date and time	Attached to resource with id	Clients	Max minutes with no activity	BC	Data delivery rate (data entries per hour)	Last data delivery rejected	Nr data packages delivered (including rejected)	Nr deliveries rejected	Interface status	Last interface failure	Nr interface failures	Last sensor failure	Nr sensor failures	Sensor status
RFID Sheep Scale	View	2021-07-01 16:47:21	none	View	360	View	0.31823845654348576	{}	136	0	Running	{}	0	{}	0	pending
RFID Sheep Feeder	View	2021-07-01 16:47:21	none	View	360	View	46.84349636282834	{}	20027	0	Running	{}	0	{}	0	Running

Figure 44: UI resources page showing all sensors (Observers)

Stellenbosch University https://scholar.sun.ac.za

The way in which each resource's data is displayed is shown in figures 45, 46 and 47. Figure 45 shows how each resource's attributes are displayed, categorized under personal, relational, management and state attributes. Note that the personal and management attributes can be edited, but not the relational. Each state attribute has a checkbox under it which can be used to display the RDs used to calculate the attribute's value.

Attributes:

Personal

Personal						
Offspring	Date of birth	EID	Dam	Gender	Sire	Breed
{}	pending	964001031059972	pending	pending	pending	SAVM
Edit	Edit	Edit	Edit	Edit	Edit	Edit

Relational

Relational			
Sensors	Feeders	House	Group
[]	["Feeder 6"]	House 6	Group 6

Management

Management		
Set food content	Case studies	Max hours between eating activities
{}	{}	24
Edit	Edit	Edit

State

Weight gain (kg/day)			Reproduction			Eating (g/day)			Weight (kg)			FCR (kg_eaten/kg_gained)		
Gain stats since last relocation	Current daily gain	Overall gain stats	On heat	Pregnant	In labor	Eating stats since last relocation	Current daily intake	Overall eating stats	Current weight	Overall weight stats	Weight stats since last relocation	Current FCR	Overall FCR stats	FCR stats since last rel
{"avg":0,"max":0,"min":0}	0	{"avg":0,"max":0,"min":0}	false	false	false	View	804.454	View	45	{"avg":45,"max":45,"min":45}	{"avg":45,"max":45,"min":45}	3592.91	{"avg":3592.91,"max":3592.91,"min":3592.91}	{"avg":3592.91,"max":359

Figure 45: UI resource details showing a sheep's attributes

Stellenbosch University https://scholar.sun.ac.za

Figure 46 shows how the UI displays the schedule, execution and biography of each resource. Under *Schedule, Get schedule->Schedule new acts* was clicked, followed by clicking on the dropdown box under *Activity type,* which gives the possible activities that can be scheduled for this resource (sheep). When activities under *Execution* or *Biography* contain data with more than 100 characters, a view button is generated on the UI, which can be clicked to display the data in the button. The same button can again be clicked to hide this.

Figure 46: UI resource details showing a sheep's schedule, execution and biography

Stellenbosch University https://scholar.sun.ac.za

Figure 47 shows how the Resource Data of each resource is displayed in the UI. If the data is numeric it will be plotted. The thicker line in figure 47 shows the eating data of a sheep over a few days and the thinner line in the plot shows how long the sheep ate every time it had eaten.

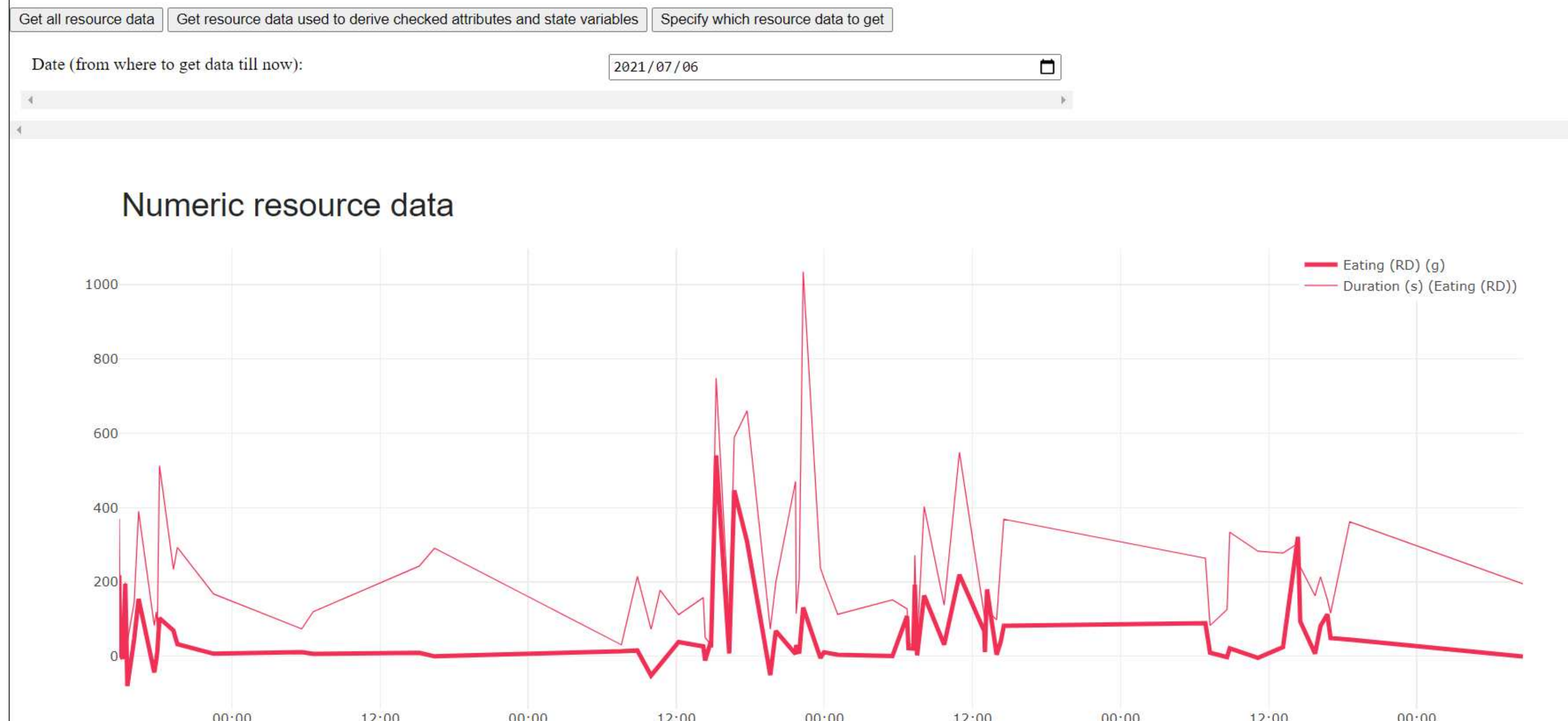


Figure 47: UI resource details showing a sheep's resource data

Stellenbosch University https://scholar.sun.ac.za

H.2 Downloaded FCR data for student experiments

Figures 48 and 49 show an example of the Excel files that could be downloaded by students for their experiments on the sheep farm. For every case study, an excel file would be downloaded for each group. In each file there is a *Group* sheet which contains the average FCR of the entire group as shown in figure 48. There is also a sheet for every sheep, showing all the data and variables of interest as shown in figure 49.

Date	Group Avg FCR (kg_eaten/kg_gained)
25/05/2021	9,908006955
27/05/2021	3,701021509
30/05/2021	31,12107978

Figure 48: Sheep data in downloaded excel file using UI – group sheet

	Date	Time	Eating (g)	Weight (kg)	Last 24h eaten (kg)	Total eaten (kg)	Total gain (kg)	Overall FCR (kg_eaten/kg_gained)
Starting weight	24/05/2021	08:36:55		57				
Case study start	24/05/2021	07:00:00						
	24/05/2021	07:49:31	260,4439163		0,260443916	0,260443916		
	24/05/2021	08:36:55		57	0,260443916	0,260443916	0	
	24/05/2021	08:59:26	448,5235112		0,708967427	0,708967427		
	24/05/2021	10:09:54	701,0658881		1,410033316	1,410033316		
	24/05/2021	11:20:07	660,7718327		2,070805148	2,070805148		
	24/05/2021	12:30:03	749,5520732		2,820357222	2,820357222		
	24/05/2021	13:39:54	709,7745237		3,530131745	3,530131745		
	24/05/2021	14:50:14	213,7846633		3,743916409	3,743916409		
	24/05/2021	16:00:18	491,5089451		4,235425354	4,235425354		
	24/05/2021	17:10:24	791,4789674		5,026904321	5,026904321		
	24/05/2021	18:20:06	754,5401918		5,781444513	5,781444513		
	24/05/2021	19:13:47	718,8773932		6,500321906	6,500321906		
	24/05/2021	20:24:17	630,9604642		7,13128237	7,13128237		
	24/05/2021	21:34:14	706,6542838		7,837936654	7,837936654		
	24/05/2021	22:43:58	292,4677127		8,130404367	8,130404367		
	24/05/2021	23:54:22	313,4010085		8,443805375	8,443805375		
	25/05/2021	01:04:21	612,7127736		9,056518149	9,056518149		
	25/05/2021	02:14:37	670,9337958		9,727451945	9,727451945		
	25/05/2021	03:24:30	354,6128422		10,08206479	10,08206479		
	25/05/2021	04:34:34	313,4253771		10,39549016	10,39549016		
	25/05/2021	05:44:21	268,5735659		10,66406373	10,66406373		
	25/05/2021	06:54:41	769,9437368		11,43400747	11,43400747		
	25/05/2021	08:04:43	282,7961859		11,45635974	11,71680365		
	25/05/2021	08:36:55		58	11,45635974	11,71680365	1	11,71680365
	25/05/2021	09:14:43	564,7007781		11,572537	12,28150443		
	25/05/2021	10:24:50	433,7975666		11,30526868	12,715302		
	25/05/2021	11:35:06	387,9950896		11,03249194	13,10329709		
	25/05/2021	12:45:21	248,5548816		10,53149475	13,35185197		
	25/05/2021	13:55:12	749,8215979		10,57154182	14,10167357		

Figure 49: Sheep data in downloaded excel file using UI – sheep sheet

Stellenbosch University https://scholar.sun.ac.za

Appendix I Computational requirements evaluation

Figures 50 and 51 show the CPU usage and RAM usage plots when the computational resource requirements of the BASE architecture implementation were measured as part of the evaluation of the extended BASE architecture (section 10.2.4). Figure 50 shows the CPU usage measured using Windows's Performance Monitor and figure 51 shows the CPU measured using Erlang's Observer. Both of these figures show the plots for the experiment where 100 holons were added to the system.

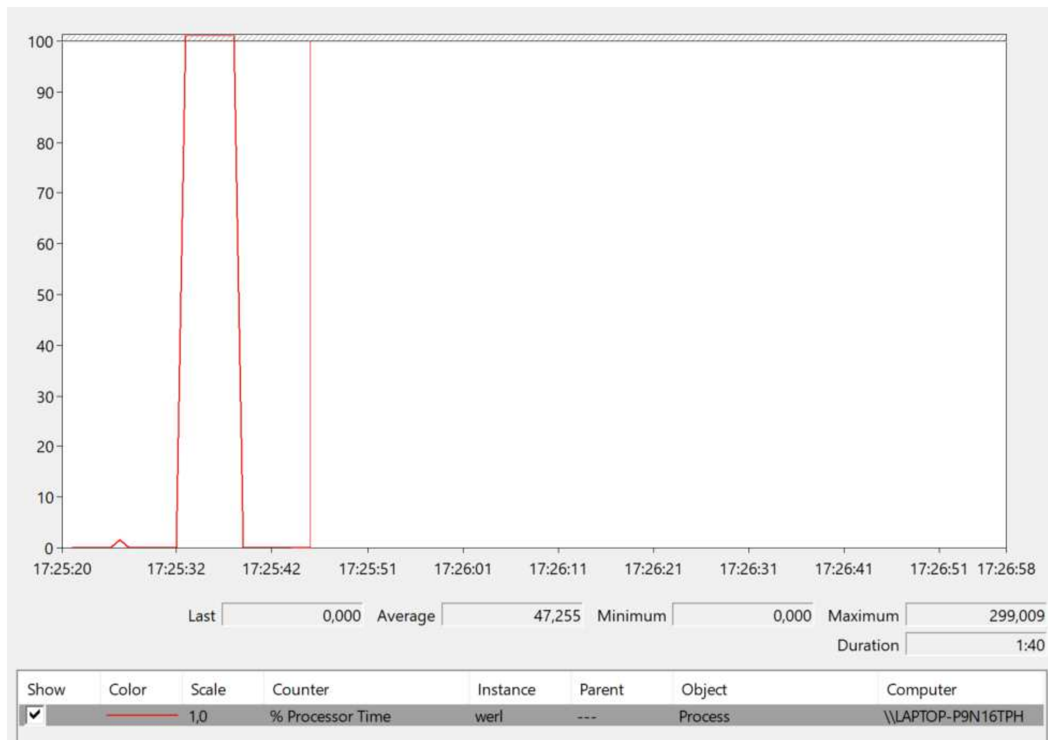


Figure 50: Implementation's CPU usage before, during and after the addition of 100 holons to the system

Stellenbosch University https://scholar.sun.ac.za

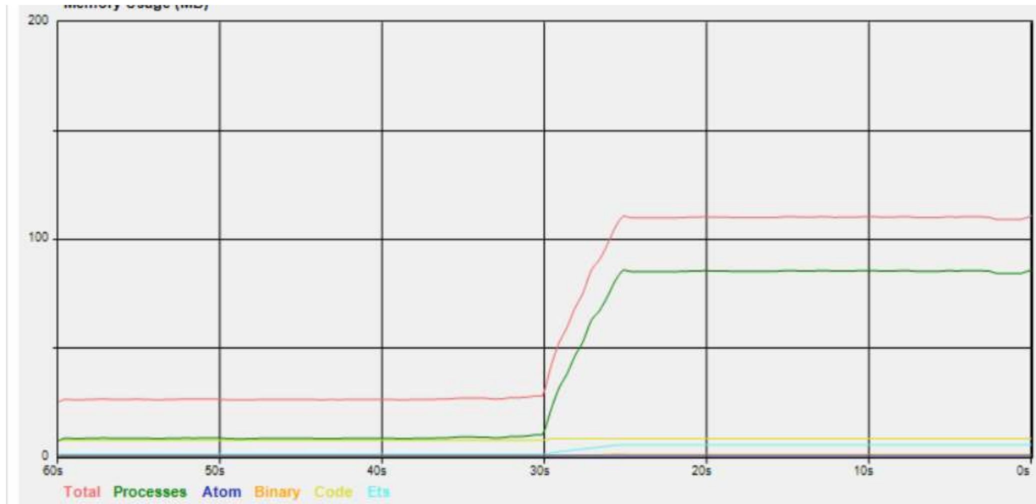


Figure 51: Implementation's RAM usage before, during and after the addition of 100 holons to the systema

Table 8 shows the detailed results of the computational requirement experiment discussed in section 10.2.4.

Table 8: Service initialisation time and computational resource requirements of the implementation for different numbers of holons in the system

Number of holons	Service initialisation time (milli-seconds)	While holons were being added			After holons were added	
		Average CPU Usage (% of total)	Time to add holons (s)	CPU Time (s)	Total RAM Usage (MB)	Average CPU Usage (% of total)
1	0	45	0.016	0.007	22.1	0.17
10	15	100	0.359	0.359	30.2	0.15
100	933	100	3.563	3.563	108.6	0.19
500	7109	100	45.547	45.547	440.1	0.18
1000	14812	100	70.594	70.594	855.6	0.18