University of Southern Denmark

A Software Approach to Mitigating Barriers for Smart Grid Integration in the Retail Sector

Hviid, Jakob

DOI:
10.21996/4mb1-ec75

Publication date:
2019

Document version:
Final published version

Citation for pulished version (APA):
Hviid, J. (2019). *A Software Approach to Mitigating Barriers for Smart Grid Integration in the Retail Sector*. [Ph.D. thesis, SDU]. Syddansk Universitet. Det Tekniske Fakultet. https://doi.org/10.21996/4mb1-ec75

Go to publication entry in University of Southern Denmark's Research Portal

Terms of use
This work is brought to you by the University of Southern Denmark.
Unless otherwise specified it has been shared according to the terms for self-archiving.
If no other license is stated, these terms apply:

- You may download this work for personal use only.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying this open access version

If you believe that this document breaches copyright please contact us providing details and we will investigate your claim.
Please direct all enquiries to puresupport@bib.sdu.dk

Download date: 22. May. 2025

A Software Approach to Mitigating Barriers for Smart Grid Integration in the Retail Sector

Jakob Hviid

Center for Energy Informatics
Faculty of Engineering
University of Southern Denmark

August 2019

SUPERVISION

Principal supervisor: Mikkel Baun Kjærgaard

COLOPHON

© Jakob Hviid 2019

Email:	jah@mmmi.sdu.dk	LinkedIn:	jakobhviid
ORCID:	0000-0001-6532-6798	Res. ID:	Q-1061-2018
G. Scholar:	uvLuRRQAAAAJ		

Typeset by the author in TeX Gyre Pagella 11pt, CABIN, Inconsolata and *TeX Gyre Chorus*, using LaTeX and the modernthesis template created by Claudio Mattera.
Printed by the author.

ABSTRACT

The move to green energy is challenging, as the energy grid needs to be kept stable, and traditional power plants help maintain stability. Demand Response (DR) is one solution to this, as it allows fluctuations in the energy grid to be mitigated by the demand-side changing their usage patterns. In Denmark, the retail sector consumed 9.344 TJ in 2013 alone, which equates to about 8% of the total industry sector consumption. This usage makes them compelling for DR integration. This Ph. D. thesis seeks to mitigate the barriers for Smart Grid Integration of the Retail Sector by applying a software approach. To do so, Building Operating Systems (BOSs) are used as a means to integrate said retail stores into the smart grid, to partake in DR activities. Retail stores' motivations to participate in the grid is explored, as well as their concerns to do so. As costs and impact on sales are their main concerns, solutions are explored to mitigate these barriers. First, an Activity-Tracking Service (ATS) is built to create better decision points for DR applications. Second, this is followed by an initiative, in the form of the Service Abstraction Layer (SAL), to increase the portability of services and applications, thereby making certain implementations are made once, instead of being re-implemented for each building. This reduces the cost of DR application implementations. Third, the Service Support System (SSS) and the Ontology based Package Manager (OPM) are built and demonstrated as solutions to reducing deployment costs of BOSs, by creating the tooling for automatic configuration of services, and deployment of drivers, services, and applications. Lastly, the impact of the above systems to the building sector ecosystem is discussed in detail, and highlight several incentives for all stakeholders to participate. The Ph. D. thesis' contributions show significant reductions to BOS related costs, which in turn makes investments in BOS solutions more feasible when considering them for DR.

RESUMÉ

Overgangen til grøn energi er en stor udfordring, da energinettet skal holdes stabilt, og traditionelle kraftværker er med til at bevare denne stabilitet. Fleksibel energi er én løsning på dette, da det gør det muligt at afhjælpe svingninger i energinettet ved at give forbrugerne incitament for at ændre deres brugsmønstre. I Danmark forbrugte detailhandelen 9.344 TJ i 2013, hvilket svarer til ca. 8 % af det samlede industrisektorforbrug. Dette forbrug gør dem interessante for fleksibel energi integration. Denne afhandling forsøger at mindske forhindringerne for integration af detailhandelssektoren i det intelligente elnet, ved hjælp af en softwaretilgang. For at gøre dette bruges bygningsoperativsystemer som et middel til at integrere detailforretningerne i det intelligente elnet, og for at kunne tage del i aktiviteter relateret til fleksibel energi. Detailhandlens motivationer for at deltage i nettet undersøges, såvel som deres bekymringer relateret til at medvirke til fleksibel energi. Da omkostningerne og eventuel påvirkning af deres salg er deres største bekymringer, undersøges eventuelle løsninger der kan afbøde disse barrierer. En af disse løsninger er en service til at følge aktiviteterne i bygningen, som skal bidrage med at skabe bedre beslutningspunkter for applikationer til fleksibel energi. Dette efterfølges af et serviceabstraktionslag der øger portabiliteten af services og applikationer i bygningsoperativsystemer, som hjælper med at sikre at koden kun skal implementeres en gang, og ikke skal ændres for hver bygning det skal operere i. Dette reducerer omkostningerne af applikationsimplementeringer der skal understøtte fleksibel energi. Et pakke deployerings system kaldet OPM, og en service til at understøtte automatisk konfiguration af services, kaldet SSS, bliver bygget og demonstreret som løsninger til at reducere installationsomkostningerne af bygningsoperativsystemerne. Endelig diskuteres hvordan ovenstående systemer påvirker det omlæggende økosystem omkring bygningssektoren, og fremhæver flere incitamenter for at alle interessenter deltager. Afhandlingens bidrag viser betyde-

lige reduktioner til bygningsoperativsystem relaterede omkostninger, hvilket igen gør investeringer i disse løsninger mere gennemførlige, når man overvejer dem i forbindelse med integration i det intelligente elnet.

STRUCTURE OF THE THESIS

The Ph. D. thesis is divided into five separate parts.

Part I of this Ph. D. thesis provides an overview of the thesis. In this part, an introduction, and a summary of the background is provided. It is structured in two chapters:

- Chapter 1: Introduction
- Chapter 2: Background

Part II contains the main contributions of this thesis, which is based on the publications made throughout the Ph. D. project. Chapter 3 explores the current state of retail stores hardware deployments, as well as the motivations and barriers of integrating retail stores into the smart grid. Chapter 4 to 6 explores several software solutions, to mitigate the barriers mentioned above. The topics presented in Part II are as follows:

- Chapter 3: Retail Stores as Flexible Consumers
- Chapter 4: Activity Tracking for better DR Decision Making
- Chapter 5: The Service Abstraction Layer
- Chapter 6: Mitigating BOS Deployment Costs

Part III concludes the thesis, summarizes the conclusions of the thesis, and details thoughts on future research avenues. It is structured in two chapters:

- Chapter 7: Future Research
- Chapter 8: Conclusion

Part IV contains all of the authors first author papers that are part of the Ph. D. project, and that is included as a part of this thesis. The papers are included with only cosmetic adaptations to fit better with the structure and feel of this thesis, but also to make sure they are easier to read. Also, a few spelling mistakes have been fixed. Apart from

these changes, they are included verbatim. Each publication includes its own references, where the numbers correspond with the chapters reference, but also the overall references provided in Part V. This makes it easier for the reader to see what is referenced in the papers, but also finding any reference in the entire thesis as they can be looked up at the end of this book. The papers are structured in the following chapters:

- Chapter 9: The Retail Store as a Smart Grid Ready Building
- Chapter 10: Activity-Tracking Service for Building Operating Systems
- Chapter 11: Service Abstraction Layer for Building Operating Systems
- Chapter 12: Service Portability and Discovery in BOSs using Semantic Modeling
- Chapter 13: Enabling Auto-Configuring Building Services
- Chapter 14: OPM: An Ontology Based Package Manager for BOSs

The final part is Part V, that includes the references for the entire Ph. D. thesis, including the publications references, as explained in the previous paragraph.

PUBLICATIONS

MAIN AUTHOR PUBLICATIONS

The following are the publications, written as first author, included in this thesis.

[1] Jakob Hviid and Mikkel Baun Kjærgaard. 'The Retail Store as a Smart Grid Ready Building: Current Practice and Future Potentials'. In: *Power & Energy Society Innovative Smart Grid Technologies Conference (ISGT), 2018 IEEE*. IEEE. Washington DC, USA, Oct. 2018. DOI: 10.1109/ISGT.2018.8403354. **Published**.

[2] Jakob Hviid and Mikkel Baun Kjærgaard. 'Activity-Tracking Service for Building Operating Systems'. In: *2018 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. IEEE. 2018, pp. 854–859. DOI: 10.1109/PERCOMW.2018.8480362. **Published**.

[3] Jakob Hviid and Mikkel Baun Kjærgaard. 'Service Abstraction Layer for Building Operating Systems: Enabling portable applications and improving system resilience'. In: *2018 IEEE International Conference on Communications, Control, and Computing Technologies for Smart Grids (SmartGridComm)*. IEEE. Aalborg, Denmark, Oct. 2018, pp. 1–6. DOI: 10.1109/SmartGridComm.2018.8587543. **Published**.

[4] Jakob Hviid, Aslak Johansen, Gabe Fierro and Mikkel Kjærgaard. 'Service Portability and Discovery in Building Operating Systems Using Semantic Modeling'. In: *2020 IEEE International Conference on Pervasive Computing and Communications (PerCom) (PerCom 2020)*. ACM. Austin, USA, Mar. 2020. **Submitted**.

[5] Jakob Hviid, Aslak Johansen, Fisayo Caleb Sangogboye and Mikkel Baun Kjærgaard. 'Enabling Auto-Configuring Building Services: The Road to Affordable Portable Applications for Smart Grid Integration'. In: *Proceedings of the Tenth International Conference on Future Energy Systems* (*ACM e-Energy '19*). Phoenix, AZ, USA: ACM, June 2019, pages 68–77. DOI: 10.1145/3307772.3328288. **Published**.

[6] Jakob Hviid, Aslak Johansen, Fisayo Caleb Sangogboye and Mikkel Kjærgaard. 'OPM: an Ontology Based Package Manager for Building Operating Systems'. In: *Proceedings of the International Conference on Internet of Things Design and Implementation* (*IoTDI 2020*). ACM. 2020. **In Submission**.

CO-AUTHOR PUBLICATIONS

The following are secondary publications, contributed as co-author during the Ph. D. project. These are not included as a part of the Ph. D. thesis.

[7] Muhyiddine Jradi, Henrik Engelbrecht Foldager, Rasmus Camillus Jeppesen, Jakob Hviid, Mikkel Ask Rasmussen and Mikkel Baun Kjærgaard. 'Modeling and Performance Simulation of a Retail Store as a Smart Grid Ready Building'. In: *Building Simulation 2019*. International Building Performance Simulation Association. Rome, Italy, 2019. **Accepted**.

Finally, from so little sleeping and so much reading, his brain dried up and he went completely out of his mind.

MIGUEL DE CERVANTES SAAVEDRA, DON QUIXOTE

ACKNOWLEDGMENTS

For as far as i can remember, I have had this view of researchers and prominent people in the world, of being impossibly confident and big perfect icons. Throughout the Ph. D., though, I learned that even the big icons are humans, just like the rest of us, doing what they can to further the knowledge of humanity. They are as insecure and had the same struggles, and all learned to improve who they are, and what they do. This, for me, is what a Ph. D. embodies. The never-ending pursuit for perfection in who you are, what you do, and how you do it.

Doing a Ph. D. can be a lonely, tough, and challenging endeavor. For me, it came with an existential crisis midway, and I was left wondering why I did a Ph. D., and even to some degree questioned everything I knew. Fortunately, while it sometimes felt lonely, I was definitely not alone. I was blessed with a great advisor, Mikkel Baun Kjærgaard, and was received by an exceptionally supporting office consisting of Aslak Johansen, Elena Markoska, and Fisayo Sangogboye. They helped me immensely with feeling welcome from the first day I arrived and discussing the subject matter, but also supported me throughout my doubts and difficulties, and made the entire Ph. D. experience an extraordinary journey. Thanks go to all of you. I would not be handing in my Ph. D. thesis without all of your presences in my life. I also want to thank the rest of my colleagues and fellow Ph. D. students for creating a comfortable environment for us all, and for the time we have spent together. Some of my best friendships have been conceived at our office.

Thanks also go to Tianzhen Hong for accommodating my visit to Lawrence Berkeley National Lab for my change of research environment, and to Bo Nørregaard Jørgensen for giving me the opportunity to do this Ph. D. at all.

Also, I would like to thank my family for the patience, and sincerely ***trying*** to understand my feeble attempts to explain what I am doing

in this Ph. D., even though I seem to keep failing thoroughly in this endeavour.

Last, but most importantly of all, I want to thank my wonderful wife, Annika, that supported me unwaveringly throughout these three years. Even as she disagrees with me doing the Ph. D., she stood by me and my decision, never holding it against me. We got married just before I started my Ph. D., which means we have only experienced marriage and Ph. D. hand in hand. I honestly do not think I knew how much a partner sacrifices for the Ph. D. endeavor. The way that I see it, she is the real hero of these contributions and this Ph. D. thesis. Thank you for everything you bring to my life! I can only hope I will support you with such grace, as you have shown me, throughout the hardships, that life will inevitably bring.

CONTENTS

Abstract i

Resumé iii

Structure of the thesis v

Publications vii

Acknowledgments ix

Contents xv

List of Figures xix

List of Tables xxii

Acronyms xxiv

I Overview 1

1 Introduction 3
- 1.1 The Climate Crisis 3
- 1.2 Energy Production and Stability 4
- 1.3 Supporting a Green and Stable Energy Grid with Demand Response 5
- 1.4 Buildings Role in the Green Movement 6
- 1.5 The FlexReStore Project 7
- 1.6 Research Approach 9
- 1.7 Research Goals and Contributions 10

2 Background 13
- 2.1 Demand Response 13

2.2 Building Management Systems 15
2.3 Building Operating Systems 16
2.4 Communication Busses and Authentication 18
2.4.1 Bosswave 18
2.4.2 Apache Kafka 21
2.5 DR and the Software Space 21
2.6 RDF Ontologies and Metadata 24
2.6.1 Brick 26
2.7 Topic Grouping 27

II Technical Contributions 29

3 Retail Stores as Flexible Consumers 33
3.1 Introduction 33
3.2 Contributions 34
3.2.1 Main contributions in paper [1] 35
3.2.1.1 Contribution 1 35
3.2.1.2 Contribution 2 37
3.2.1.3 Contribution 3 37
3.2.1.4 Contribution 4 38
3.2.1.5 Contribution 5 40
3.3 Conclusions 42

4 Activity Tracking for better DR Decision Making 45
4.1 Introduction 45
4.2 Related Work 46
4.3 Contributions 49
4.3.1 Main Contributions in paper [2] 49
4.3.1.1 Contribution 1 49
4.3.1.2 Contribution 2 51
4.4 Conclusions 54

5 The Service Abstraction Layer 55
5.1 Introduction 56
5.2 Related Work 57
5.3 Contributions 61
5.3.1 Main contributions in paper [3] 61
5.3.1.1 Contribution 1 61

5.3.1.2 Contribution 2 65
5.3.2 Main contributions in paper [4] 68
5.3.2.1 Contribution 1 68
5.3.2.2 Contribution 2 75
5.3.3 Main contributions in paper [6] 78
5.3.3.1 Contribution 1 78
5.4 Conclusions 79

6 Mitigating BOS Deployment Costs 81
6.1 Introduction 81
6.2 Related Work 84
6.3 Contributions 88
6.3.1 Main contributions in paper [5] 88
6.3.1.1 Contribution 1 88
6.3.1.2 Contribution 2 93
6.3.1.3 Contribution 3 97
6.3.2 Main contributions in paper [6] 98
6.3.2.1 Contribution 1 98
6.3.2.2 Contribution 2 101
6.3.2.3 Contribution 3 103
6.4 Conclusions 105

III Conclusions 107

7 Future Research 109

8 Conclusion 113

IV Publications 117

9 The Retail Store as a Smart Grid Ready Building 121
9.1 Abstract 121
9.2 Introduction 122
9.3 Methodology 123
9.4 Screening Results 124
9.5 Screening Conclusions 126
9.6 Leveraging the Demand Response Potential 130
9.7 Conclusions 132

10 Activity-Tracking Service for Building Operating Systems 137
10.1 Abstract 137
10.2 Introduction 138
10.3 Related Work 141
10.4 System design principles 143
10.4.1 Security and Privacy 143
10.4.2 Interoperability 143
10.4.3 Extendability 144
10.4.4 Scalability 144
10.5 System design and Components 144
10.6 Evaluation 148
10.7 Discussion 151
10.8 Conclusion 152

11 Service Abstraction Layer for Building Operating Systems 155
11.1 Abstract 155
11.2 Introduction 156
11.3 Related Work 158
11.4 Concept 160
11.5 Methodology 164
11.6 Evaluation 165
11.7 Results 166
11.8 Conclusion / Discussion 168

12 Service Portability and Discovery in BOSs using Semantic Modeling 175
12.1 Abstract 175
12.2 Introduction 176
12.2.1 Related Work 178
12.2.2 Related Work 180
12.2.3 Approach 183
12.3 Concept 184
12.3.1 Terminology 184
12.3.2 SAL Benefits and Implications 185
12.3.3 SAL Ontology Description and SALI 186
12.4 Evaluation 191
12.4.1 Results 195
12.5 Discussion 196
12.6 Conclusion 197

13 Enabling Auto-Configuring Building Services 203
13.1 Abstract 203
13.2 Introduction 204
13.2.1 Problem 207
13.2.2 Related Work 207
13.2.3 Approach 208
13.3 Software Based DR 209
13.3.1 Occupant Aware DR Case Study 210
13.4 Design 212
13.4.1 Ontologies 215
13.4.2 Service Publication 217
13.4.3 Service Discovery 217
13.4.4 Data Plane / Service Support System 220
13.5 OccuRE Redesign 220
13.6 Ecosystem Impact Assessment 224
13.6.1 The Entrepreneur 227
13.6.2 The Application Developer 228
13.6.3 The Customer 228
13.7 Service Support System Assessment 229
13.8 Conclusion 230

14 OPM: An Ontology Based Package Manager for BOSs 237
14.1 Abstract 237
14.2 Introduction 238
14.3 Related Work 243
14.4 The OPM Concept and Design 245
14.5 Evaluation 251
14.5.1 Evaluation Case: OccuRE 252
14.5.2 Evaluation Parameters 257
14.6 Results 259
14.7 Discussion 260
14.8 Conclusion 262

V Bibliography 269

References 271

LIST OF FIGURES

1.1 Example of Energy Generation and Consumption for Denmark on 2019-07-23 sourced from Energinet [17] 5
1.2 Contribution Timeline Divided into Contribution Goals 11

2.1 A generalized Building Operating System (BOS) overview 16
2.2 Overview of areas touched upon by this Ph. D. thesis, which is repeated in each chapter, underlining the related areas 28

3.1 Overview of Areas examined in this Chapter 33

4.1 Overview of Areas examined in this Chapter 45
4.2 Activity-Tracking Service (ATS) Placement in a BOS Context 49
4.3 Simplified ATS architecture describing the modified State Machine implementation 50
4.4 Evaluation Setup 52
4.5 Testing Results 53

5.1 Overview of Areas examined in this Chapter 55
5.2 BOS Layers overview with Service Abstraction Layer (SAL) 56
5.3 Service Abstraction Layer Class Relationships 63
5.4 Service Abstraction Layer Class Inheritance 64
5.5 Service Abstraction Layer Example: OccuRE 65
5.6 Architecture moved to XBOS and Bosswave 67
5.7 SAL Ontology Parent Classes - Relationships and Properties 71
5.8 SAL Model; Weather Example 74
5.9 Service Dependencies For Each Case Building . . . 76

5.10 The SAL Extensions Made to Support Input Descriptions 79

6.1 Overview of Areas examined in this Chapter 81
6.2 Microservice Ecosystem Example 89
6.3 Request for service publication 91
6.4 Query for listing all published services 92
6.5 The Ontology based Package Manager Concept . . . 99
6.6 Host Overview: Ontology based Package Manager (OPM) required for each swarm or host. Drivers and services can be distributed. Only one Service Support System (SSS) should exist for the setup. Colors correspond with the deployment phases described in Figure 6.8 . . . 100
6.7 BOS environment with color coding for order of deployment. Excluding physical hosts. Colors correspond with the deployment phases described in Figure 6.8 . . . 100
6.8 A typical BOS Deployment Procedure, divided in phases, with associated tasks 101

10.1 The Retail Building and Internet of Things (IoT) Setting for the ATS 140
10.2 ATS Placement in a Building Operating System Context 144
10.3 Simplified ATS architecture describing the modified State Machine implementation 147
10.4 Evaluation Setup 148
10.5 Testing Results 150

11.1 BOS Layer with added SAL 156
11.2 SAL Class Relationships 161
11.3 SAL Class Inheritance 162
11.4 SAL Example: OccuRE 163
11.5 SAL Example: Calendar Scheduler 164
11.6 Architecture moved to XBOS and Bosswave 167

12.1 BOS Layers overview with SAL 177
12.2 SAL Ontology Parent Classes - Relationships and Properties 187
12.3 SAL Model; Weather Example 190
12.4 Evaluation Microservice Ecosystem 192
12.5 Service Dependencies For Each Case Building . . . 193

13.1 Building Stream Configuration 210
13.2 Microservice Ecosystem Example 213
13.3 Request for service publication 218
13.4 Query for listing all published services 219
13.5 sMAP Queries for Resolving Sensor Stream Data . . 221
13.6 SPARQL Protocol and RDF Query Language (SPARQL) Query for Resolving Sensor Stream Data 222
13.7 Sequence of interaction between the PreCount Modules 224

14.1 Ontology based Package Manager (OPM) Concept . 246
14.2 Simplified DB Structure 247
14.3 The SAL Extensions Made to Support Input Descriptions 248
14.4 Simplified Example SAL Model of a Generic Service 249
14.5 Simplified Example of a SAL Requirement Query . 249
14.6 Host Overview: OPM required for each swarm or host. Drivers and services can be distributed. Only one SSS should exist for the setup. Colors correspond with the deployment phases described in Figure 14.8 250
14.7 BOS environment with color coding for order of deployment. Excluding physical hosts. Colors correspond with the deployment phases described in Figure 14.8 . . 250
14.8 A typical BOS Deployment Procedure, divided in phases, with associated tasks 251
14.9 Sequence of interaction between the OccuRE Components without SSS 252
14.10 Building Stream Configuration 253
14.11 sMAP Queries for Resolving Sensor Stream Data . . 254
14.12 Sequence of interaction between the OccuRE, the SSS and Estimation Service 255
14.13 The University of Southern Denmark OU 44 Living Lab used for the Brick Model and for the data used in the OccuRE services 257
14.14 Package Dependencies with Brick requirements . . 258

LIST OF TABLES

2.1 Automated DR methods categorized by DR service type, targeted loads and grid/building roles 22

3.1 Consumption across categories, Total observed consumption, and consumption which is unaccounted for . . 36
3.2 Demand Response Potential (Demand Response Potential (DRP)) 36
3.3 DRP overview based on how they are controlled . . 39
3.4 Demand Response (DR) Mobilization Limitations. . 39

4.1 Comparison of Building Operating Systems (BOSs) and activity modeling 47

5.1 Related Work Comparison 58
5.2 Subclasses for information annotation 72
5.3 Service Portability Evaluation Results 78

6.1 Feature Comparison between Package Managers . . 86
6.2 Costs impacts, benefits and disadvantages associated with each phase, distributed relevant across stakeholders 94
6.3 Comparison between the Original Implementation and the Refactored Code for the Service Support System Adaptation 98
6.4 Comparison between OccuRE and OccuREv2 and PLCount and PLCountv2 respectively 102
6.5 Time in minutes to prepare the BOS and OccuRE, divided into phases and iterations 102

9.1 Consumption across categories, Total observed consumption, and consumption which is unaccounted for . . 125
9.2 Demand Response Potential (DRP) 125

9.3 Demand Response Potential (DRP) overview based on how they are controlled 127
9.4 Demand Response Mobilization Limitations 127

10.1 Comparison of operating systems and activity modeling 139

12.1 Related Work Comparison 181
12.2 Subclasses for information annotation 189
12.3 Service Portability Evaluation Results 196

13.1 Automated DR methods categorized by DR service type, targeted loads and grid/building roles 211
13.2 Results from SPARQL Protocol and RDF Query Language (SPARQL) Queries in Figure 13.6 223
13.3 Comparison between the Original Implementation and the Refactored Code for the Service Support System Adaptation 225
13.4 Costs impacts, benefits and disadvantages associated with each phase, distributed relevant across stakeholders 226

14.1 Feature Comparison between Package Managers . . 242
14.2 Comparison between OccuRE and OccuREv2 and PLCount and PLCountv2 respectively 256
14.3 Time in minutes to prepare the BOS and OccuRE, divided into phases and iterations 259

ACRONYMS

API Application Programming Interface.
ATS Activity-Tracking Service.
AWS Amazon Web Services.

BMS Building Management System.
BOS Building Operating System.
BOSS Building Operating System Services.

DNS Domain Name Server.
DoT Declaration of Trust.
DR Demand Response.
DRP Demand Response Potential.
DSO Distribution System Operator.

GDPR General Data Protection Regulation.
gRPC gRPC Remote Procedure Calls.

HA High Availability.
HAL Hardware Abstraction Layer.
HTML Hypertext Markup Language.
HVAC Heating, Ventilation, and Air Conditioning.

IoT Internet of Things.

JSON JavaScript Object Notation.
JSON-LD JavaScript Object Notation for Linked Data.

OOP Object Oriented Programming.
OPM Ontology based Package Manager.
OWL Web Ontology Language.

RDF Resource Description Framework.
RDFa Resource Description Framework in Attributes.

ROI Return on Investment.

SAL Service Abstraction Layer.
SPARQL SPARQL Protocol and RDF Query Language.
SQL Structured Query Language.
SSS Service Support System.

TTL Time to Live.

URI Uniform Resource Identifier.

WP Work Package.

XBOS eXtensible Building Operating System.

Part I

Overview

CHAPTER

INTRODUCTION 1

This chapter introduces the motivations for the work in this thesis, including the climate crisis, energy production challenges, methods to mitigate these, as well as buildings role in all of the above. Also, this chapter includes a project description for the FlexReStore project that this thesis was a part of, as well as the research approach, goals, and how the contributions of this thesis map to these goals.

1.1 THE CLIMATE CRISIS

The first (1760's), and second (1860's) industrial revolutions sparked a magnificent rise in productivity, and better living conditions for the average citizen. Electricity, infrastructure, cars, planes, light, computers, phones, and so much more spawned from the adventure. These advances were not without cost though, as they also marked the beginning of the climate crisis, also commonly referred to as climate change or global warming. Data [8] shows that before the industrial revolutions, the CO_2 levels where lower than 300 parts per million for several thousand years, but has since risen to levels above 400 parts per million during the year 2000. This signifies a serious problem, as research such as the famous Stern Review [9] shows the CO_2 should be kept below 550 parts per million to avert a significant impact on the global temperature, and the human race.

At this point, the climate crisis evidence is overwhelming. Everything from the economical perspective [9] to the impact on our ecosystem and health [10, 11, 12, 13] have been documented extensively. Thankfully, a movement towards green energy has been established at several levels of society, and recently found its way to the global leadership, accumulating to a small but significant step in the form of the Paris Accords [14]. Even with one of the larger nations going back on their engagement in the accords, the climate crisis has become a significant

topic at the highest level of government worldwide. This trend has also moved into the commercial sector, where the wind turbines and solar energy technology is selling well and is an expanding market. Some of these trends are also driven by companies need to brand themselves as green companies, by creating green initiatives where they can, as the notion supporting the move to green energy is popular among the population. For example, according to Apple's own press releases [15], Apple is running all their server farms on 100% green energy and is actively expanding its green energy capacity to support their production, as well as actively encouraging their suppliers to do so too.

1.2 ENERGY PRODUCTION AND STABILITY

It is difficult to move the entire energy production to green alternatives, as several problems present themselves. Most of the renewable energy sources have unpredictable energy production, because they rely entirely on natural forces, like the wind or the sun. This unpredictability poses a massive problem, as an energy grid has a dynamic upper and lower bound for the amount of energy available in the system [16]. If the boundary is broken in either direction, the grid can become unstable, and thereby cause a brownout. To solve this, traditional energy plants are used as a offset, as they produce a predictable amount of energy. If too much energy is present, the plant can produce less, and if there is not enough energy in the grid, it can produce more to keep the grid stable. This dependence on stable energy grids, and the difficulty of effectively storing massive amounts of energy makes it extremely difficult to shift the entire energy production to green energy. The problem is further amplified by consumer habits. For example, in the evening, most people go home and turn on the stove to make dinner. This generates a relatively sudden spike of energy demand in the energy grid, which is easier to keep up with using traditional power plants. One way of alleviating this problem, is to trade energy with neighboring countries. This kind of trade allows for overproduction or underproduction, while still keeping a stable grid. Prices are based on price negotiation, and unfortunately, neighboring countries often also overproduce energy at the same time, resulting in prices becoming negative, meaning one needs to pay to get rid of the energy that was created. Economically, this creates little incentive to solve the problem

only by trade, effectively keeping a big incentive for energy companies to keep old power plants running. Fortunately, most excess power created, that cannot be sold, can be used for other purposes like district heating.

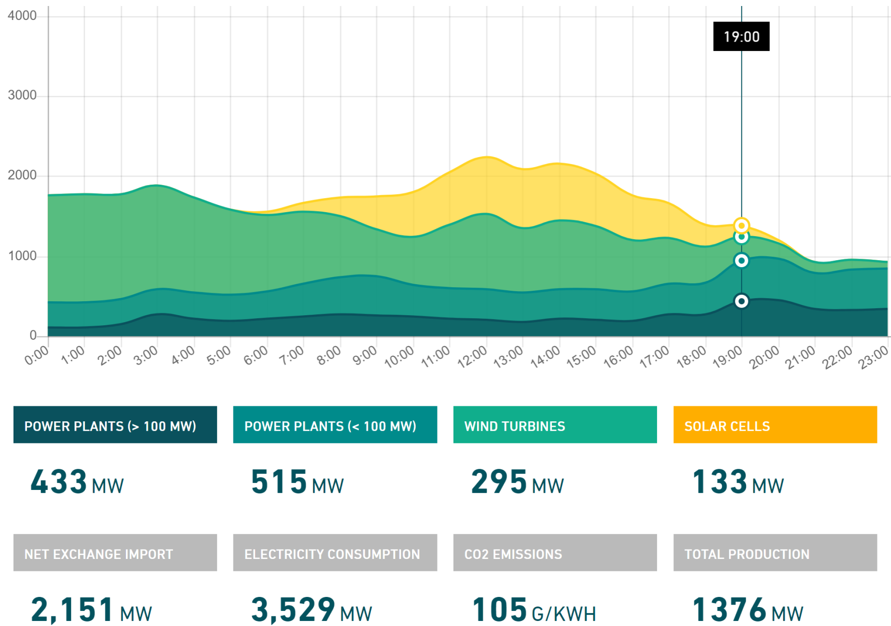

Figure 1.1: Example of Energy Generation and Consumption for Denmark on 2019-07-23 sourced from Energinet [17].

Figure 1.1 is an example from Denmark on 2019-07-23, which shows how power plants take over as the renewable energy sources, Wind Turbines and Solar Cells, stop generating energy, and how energy still needs to be imported from neighboring countries. The numbers are in MegaWatts and are for the 19:00 hours time slot.

1.3 SUPPORTING A GREEN AND STABLE ENERGY GRID WITH DEMAND RESPONSE

Another approach to mitigating the challenge of a stable and green energy grid, is called Demand Response (DR) [18, 19, 20, 21, 22]. DR introduces the concept of demand-side management, whereby the consumers are incentivized to use (or refrain from using) energy when

it is favorable for the energy provider. This can be done in several ways. One example is shifting tasks that could be done at any time, to time slots where energy generation is high, but consumption is low. Another example could be to cool larger frost storage installations a few degrees more, and then turn the cooling off when production is scarce, effectively letting the frost storage installation act as an energy storage facility. DR also covers scenarios for handling emergencies in the grid [19], which is typically implemented using DR events, where participants gain compensation for participating when actively notified to do so. However, the current trend in DR, seems to be moving in the direction of using price signals, effectively creating an incentive for the consumer to use less energy at certain times of the day, or move tasks to cheaper time slots [18]. Unfortunately, currently, most price signaling models seem to have difficulty generating significant enough savings for a consumer to have enough incentive to participate. This can be because of the relatively manual effort of participating in DR, or the relatively high cost of using automated solutions to do so.

1.4 BUILDINGS ROLE IN THE GREEN MOVEMENT

Buildings make up about 40% of the total energy consumption [23, 24]. The retail sector uses a large amount of this energy. In Denmark, the retail sector consumed 9.344 TJ in 2013 [25] alone, which equates to about 8% of the total industry sector consumption. This consumption in itself signifies the importance of the retail sector but is further underscored by the fact that electricity consumption and other energy costs are the highest cost after labor costs.

If buildings are to be integrated into DR programs, and support the transition to green energy, the integration needs to be automated and be economically feasible. Buildings today, are currently most commonly controlled by Building Management Systems (BMSs). These BMSs are rigid, and are often difficult to operate and understand for typical building managers [26]. Also, DR implementations, and settings, are not transferable between installations, making highly configurable DR implementations costly. Especially for retail stores, dynamic configuration to the preferences of the owner is paramount for system acceptance, as lost sales concerns dominate the motivational spectrum for most decision-makers [1]. This makes portability of DR functionality a sig-

nificant factor for the success of DR in the buildings space. Fortunately, the concept of Building Operating Systems (BOSs) [27, 28, 29, 30, 31, 32], have emerged in research in recent years. These systems are built as platforms for buildings, where applications can live on top of these implementations. This is typically achieved by homogenizing the applications interface by using a Hardware Abstraction Layer (HAL). A HAL is a concept used to make sure applications have to interface to one implementation, instead of having to implement functionality for each type of hardware, thereby making the code more portable. Also, BOSs often offer the means for saving historical time-series data and as well as offering some kind of authentication system. These BOSs allow applications a higher portability factor than the BMS solutions, while also allowing for extendable functionality, thus making it a strong candidate for creating smart building functionality and create portable DR applications.

Several studies already address integrating buildings into the smart grid using BOSs and DR [19, 20, 21, 22], but these are not considering retail stores. Some studies are addressing retail stores [33, 18], but are either only looking at a specific sub-system like refrigeration, or fail to address retail stores as a separate building type. Retail stores, in general, are challenging to work with because of how distinctly different they operate compared to other commercial buildings, and how heterogeneous the plug-loads are. Also, the incentive structure and concerns of the retail store owners, are significantly different from other building types, thereby greatly impacting the challenges needed to be solved. This thesis aims to explore and address these technical concerns and barriers for retail stores, with a software approach, to actively participate in DR programs and making DR applications and BOSs cost-effective to deploy across large arrays of buildings.

1.5 The FlexReStore Project

This Ph. D. study is a part of the research project FlexReStore. The name FlexReStore (Flexible Retail Stores), refers to DR integration of the retail store sector. The objective of FlexReStore is, as the name suggests, to pave the way for the retail sector to provide flexible electricity consumption. This was to be done by:

1. Creating an energy guild with the largest and most relevant stake-

holders.

2. Evolve retail store designs to include options for flexible electricity consumption.
3. Develop software-based solutions to operate flexible consumption in retail stores.
4. Study customers' reaction to observable effects of flexible consumption in relation to store owners' profitability.
5. Study retail stakeholders' attitudes towards flexible consumption.

The FlexReStore project consists of six Work Packages (WPs) described as follows:

WP1: Establish a Retail Energy Guild
: This package's goal was to establish an energy guild of retail sector partners. The goal of the energy guild was the collection of electricity consumption data and the establishment of an innovation community to prepare the retail sector for providing flexible consumption.

WP2: Create Retail Store Designs with Flexible Consumption
: This WP's goal was to evolve retail store designs present in Denmark today to new designs that include flexible consumption. The goal was to evolve designs for several store types to include flexible consumption. As cases, the project worked on retail stores operated and owned by the project user group. As options for providing flexible consumption, the project focused on lighting, and other plug-loads. The project had a goal of covering several different store types including at least a hard and a soft goods concept store, a discount store, a supermarket, and a hypermarket.

WP3: Create Software Tools for Flexible Consumption in Retail Stores
: The goal of this WP was to implement software-based solutions to support the operation of flexible consumption in stores. The goal was to provide a software prototype tool that would make it easier for store chains to roll out flexible consumption.

WP4: Collect Stakeholder Reactions to Flexible Consumptions
: Through user-studies and stakeholder involvement, the project developed knowledge of retail stakeholder's views on flexible consumption and technology. In regards to retail customers, the project covered the relation between ambient shopping experience with a focus on lighting and customer shopping behavior.

For this relationship, potentials were considered for gains in a green image towards retail customers. For retail stakeholders, motivational factors and barriers impacting the transition to flexible consumption was studied.

WP5 & WP6: Dissemination & Project Management

Dissemination and exploitation, and project management was defined as separate work packages named WP5 and WP6, respectively, and should require no further explanation.

1.6 RESEARCH APPROACH

This Ph. D. thesis' research was performed within the field of software engineering, and the subfield energy informatics. According to Sommerville [34], the term "software engineering" was first proposed in 1969 at a NATO conference. The term was defined as "an engineering discipline that is concerned with all aspects of software production" [34, p. 6]. Within the software engineering discipline, a small but important field exists called "energy informatics", to which this thesis belongs. Energy informatics covers research, development, and application of information and communication technologies, energy engineering and computer science to address energy challenges [35]. Several research methodologies exist within the field of software engineering and have been already been thoroughly explored [36, 37, 38, 39]. For the purposes of this project, several research approaches were used, based on what the situation called for. The first paper [1] presented by the author of this thesis, uses a case study approach [38] to explore the current state of four retail stores, but also uses a semi-structured interview approach [39] for the interview processes, and generating theories and knowledge from the interviews themselves, by using a variation of grounded theory [39]. Generally, when working with software engineering-related research, several research approaches can be used. One of the most prevalent approaches used is constructive research, which Crnkovic explains like this:

> "The key idea of Constructive Research [...], is the construction, based on the existing knowledge used in novel ways, with possibly adding a few missing links. The construction proceeds trough design thinking that makes projection into the future envisaged solution (theory, artifact)

> and fills conceptual and other knowledge gabs by purposefully tailored building blocks to support the whole construction. Artifacts such as models, diagrams, plans, organization charts, system designs, algorithms and artificial languages and software development methods are typical constructs used in research and engineering" – Crnkovic [36, p. 360]

However, this approach is not commonly used only by itself, but in combination with other approaches. Crnkovic continues:

> "Constructive research takes off from the existing well understood ground and that is why research in Software Engineering often starts with empirical investigations where qualitative (controlled experiment, Survey) or qualitative (Grounded Theory, Case Studies) methods are used prior to the constructive work. Only when sufficient understanding of the research problem and the domain is obtained, one can start addressing a Software Engineering problem by Constructive Research method" – Crnkovic [36, p. 364]

Most of the research performed for this Ph. D. thesis follows the path described by Crnkovic, which is the case for the rest of the papers [2, 3, 4, 5, 6]. They use the constructive research approach when building the software, and case study [38] methods for understanding and defining the scope and requirements of the software being constructed and evaluated. Most of the papers' reasoning and requirements are also based on lessons learned in paper [1] mentioned earlier.

1.7 RESEARCH GOALS AND CONTRIBUTIONS

The goals of this Ph. D. thesis are to implement software-based solutions to support the operation of flexible energy consumption in retail stores, and thereby make it easier to roll out flexible energy functionality in said retail stores. Most of the Ph. D. thesis' content is motivated based on the findings from paper [1].

The set goals are as follows:

1. Identify the current state of the retail store in regards to hardware connectivity and Demand Response Potentials (DRPs).

2. Identify retail store stakeholders motivations and perceived barriers related to DR implementations in their stores.
3. Development of software solutions to mitigate the stakeholders operational and deployment concerns in relation to DR adoption in stores.

Figure 1.2 details the above contributions, as pertains to the goals and the time frame for the six papers of the Ph. D. project. Paper [1], 'The Retail Store as a Smart Grid Ready Building: Current Practice and Future Potentials', touches on goal one and two, as it contributes to both at the same time, while the rest of the papers are confined to the third goal. Also, several topics were defined, that also correspond to the chapters in Part II. These topics were: the mapping of the current state of the retail store and the barriers, as defined with goal 1 and 2; creating better DR decision points; the Service Abstraction Layer (SAL); and finally, mitigation of deployment costs.

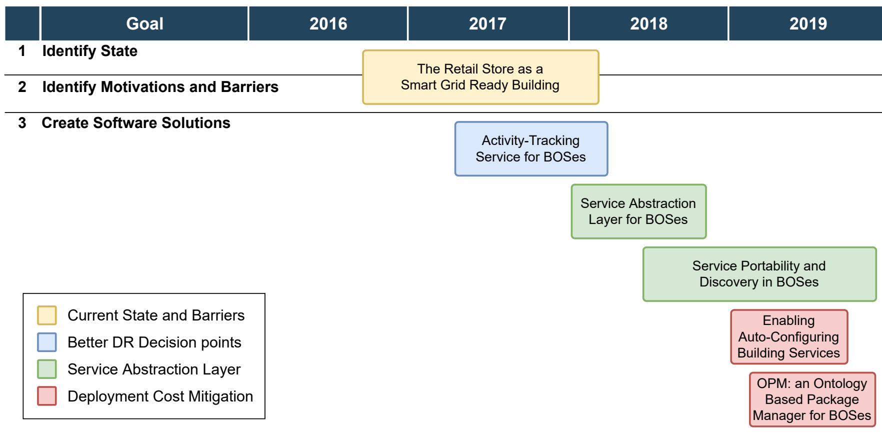

Figure 1.2: Contribution Timeline Divided into Contribution Goals.

Paper [2], named 'Activity-Tracking Service for Building Operating Systems', addressed the third goal by creating an Activity-Tracking Service (ATS) that contributed to better DR decision making, as this was one of the main concerns. In this paper, it became apparent that more overarching problems existed in the BOS space. These problems pertained to service portability, as minimal infrastructure existed in the BOS space to support these. This prompted the creation of the SAL, which also contributes to the third goal, that helps support application

and service portability, as well as service discovery in BOSs. Papers [3] and [4], named 'Service Abstraction Layer for Building Operating Systems: Enabling portable applications and improving system resilience' and 'Service Portability and Discovery in Building Operating Systems Using Semantic Modeling', details the contribution to this space. As paper [1] discovered, retail store owners' motivation and main problem with DR, is the economical aspect. Papers [5] and [6], named 'Enabling Auto-Configuring Building Services: The Road to Affordable Portable Applications for Smart Grid Integration' and 'OPM: an Ontology Based Package Manager for Building Operating Systems', address this cost perspective by preparing the BOS for auto-configurable services, as well as creating a deployment system specialized for deployment of drivers, services, and applications in a BOS context.

CHAPTER

BACKGROUND 2

This chapter, provides the background information needed to understand the rest of the chapters and contributions of this Ph. D. thesis. The Background chapter only clarifies the related technologies that are crosscutting between several chapters and papers, while the chapters in the Technical Contributions part clarifies the related work specific to those topics.

2.1 DEMAND RESPONSE

Demand Response (DR) is a technology that can impact the move to green energy significantly, by either directly or indirectly manipulating the demand-side energy consumption. As the introduction already explained and motivated the need for DR extensively (see page 4 to 6), this section will concentrate on the control types and DR strategies.

Three approaches to DR exist: DR event, DR event negotiation, and price signals. The first two are a form of direct control, where the Distribution System Operator (DSO) actively initiates customer participation in the DR activity. DR events can be automated, but are often performed by physically calling a company, and asking them to lower their energy usage, according to the terms of the contract they have negotiated, leaving the customer with compensation for lost productivity, and typically something on top to give an incentive to participate. DR event negotiations are something that is typically performed through automated means, where participants are asked to deliver a certain amount of energy, and the automated systems will then negotiate an acceptable compromise. Protocols like OpenADR [40] can support these two approaches. The latter approach, price signals, is an alternative where the DSO only signals the state of the network, by regulating the energy pricing. Participants are then supposed to be incentivized by the potential savings made in the changing price. This can be achieved

by several of the strategies described below, such as generating an energy warehouse. However, real savings can be challenging to achieve through price signals, as it requires processes to be supported and automated by multiple components, which at this state can end up costing more than the monetary savings created for the customer.

As mentioned, several strategies exist to implement DR [41, 21, 18]. To provide a better understanding of how these work, a few of the possible strategies are presented here:

Load-Shifting: This is the simple idea of moving non-essential energy using activity to be executed at a later point in time. While this can be as simple as moving the washing of the laundry a few hours, it can also be on a larger scale. One example is using refrigeration warehouses as passive energy storage. The idea is simple; when energy is cheap, cool the entire frost products a few degrees more, and when the energy is expensive, turn off the refrigeration units [42, 43]. This approach leaves a significant amount of energy passively stored as temperature in the warehouse. Another creative approach is using old water towers, that has since been changed to water pumps, as temporary pressure for the water system [44]. The tower would be filled when pricing is low, and the water pumps could then be turned off once the pricing is high, effectively creating an alternative energy storage solution.

Peak-Clipping: Peak clipping is the practice of shaving the top of the energy usage, by sacrificing non-essential energy usage, that is still considered to have a value in normal operations. An example could be as simple as a retail store that wants to make its products, and branding, visible outside of regular opening hours. In itself, the store considers this value, but as it is a non-essential, the lighting brightness could be reduced when outside of specific a specific time-frame and electricity pricing is high [41].

Valley-Filling: As DR is about keeping energy usage within certain thresholds at all times, there will be times when the grid needs more energy to be used when demand is exceptionally low. An example here could be the same as from load-shifting with the frosted storage but seen from the opposite perspective. Use the energy while it is cheap, thereby cooling down the facilities, and then turn it off when pricing increases again.

Fuel-Shift: This strategy is the idea of moving to alternative energy sources while it is demanded. For example, a company could move to an alternative fossil fuel while prices are lower for that resource. Obviously, this will help the energy grid, but not the environment. Fortunately, alternatives exist. One alternative, for example, is moving energy usage to batteries when pricing is high or using electric vehicles integrated into the energy grid [45].

Typical loads within a building, and more specifically retail stores, are lighting, cooling, ventilation, frost, and unsorted plug-loads. Plug-loads are too diverse a category to make good examples of, and frost has already been covered above. However, lighting can be integrated into DR by turning off certain fixtures that are not essential, or by dimming the light. With cooling and ventilation, the intensity can be turned down, or in some cases turned off temporarily. Alternatively, an energy storage can be created, allowing for the system to be turned off for a prolonged time span.

2.2 BUILDING MANAGEMENT SYSTEMS

Building Management Systems (BMSs), are the current industry standard for automated building control. They are typically proprietary implementations from industry vendors, that are built to support their hardware. Though in recent years, larger brands have introduced a certain amount of interoperability between their systems and competitors, as BMSs are being retrofitted in older buildings with other vendor hardware implementations. BMSs are systems that allow for a certain amount of configuration from the building operators. Unfortunately, these systems are often operated from an interface that resembles system design drawings, leaving users confused [26]. Even so, the configurational elasticity of these systems is typically minimal. From experience, some are still being interfaced to by RS232 serial ports, with software stored on 3.5-inch floppy disks, made from companies that have stopped operating several years ago. This type of system tends to be closed to external software, thereby not allowing for expanded functionalities such as DR. Fortunately, many modern BMSs are beginning to allow for external control of the BMS, leaving opportunities for expanding on the use cases a BMS can support.

2.3 BUILDING OPERATING SYSTEMS

In contrast to BMSs, Building Operating Systems (BOSs) are designed to provide the infrastructure for applications to function on top of these. Several BOS platforms and related technologies exist like Building Operating System Services (BOSS) [27], SMAP [46], eXtensible Building Operating System (XBOS)[28], Bosswave [47], Buildingdepot [29], Sensor Andrew [31], Tridium Niagara [32], Volttron [30] and more. While approaches of BOSs are different from each other, they all strive to provide a minimal set of functionality. These often include a standardized interface for the applications on top of them called a Hardware Abstraction Layer (HAL), security features to manage access to the system, a time-series database to store historical data, and more. BOSs are built with different goals in mind, and therefore, do not conform to one another's constructional nature. Some are monolithic systems that only run on one machine, while others like XBOS [28] is distributed in nature, and can span an entire network of machines. Some BOSs that are distributed use a communication bus, which is a well known architectural artifact in the Internet of Things (IoT) space. One such communication bus is Bosswave [47] that in combination with XBOS can operate a single room, a citywide system or even orchestrate on a global scale. For the purpose of this Ph. D. thesis, the terminology used is derived from XBOS. This choice is due to most of the system being based on the XBOS and Bosswave ecosystem.

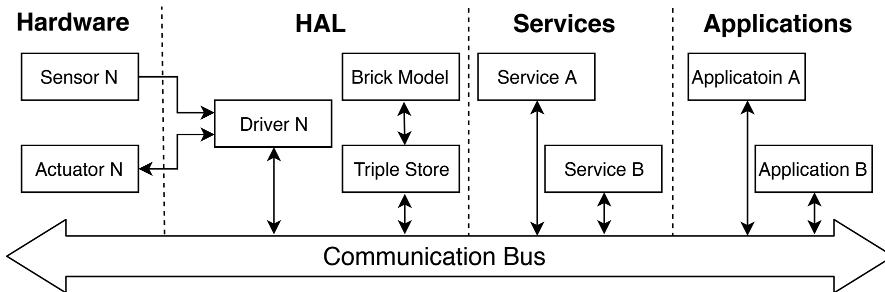

Figure 2.1: A generalized BOS overview.

Figure 2.1 shows a generalized version of the BOS. It contains several components and layers, such as a hardware layer, a HAL, a services layer, an application layer, as well as a communication bus. Below, these

components and their content are detailed.

Hardware Layer: This layer contains the hardware components that make up the eyes, ears, and arms of the BOS. Sensors act as data-gathering devices that can tell what the temperature is, or if a door is open. Actuators, on the other hand, are devices where settings can be changed or commands given, such as ”set temperature to 21 degree celsius”, or ”Open Door”. These devices can be from many different manufacturers, and in some cases, even be BMSs that expose a multitude of both types. Suffice to say; these divides can be extremely heterogeneous, with widely different implementations.

Hardware Abstraction Layer (HAL): The purpose of this layer is to ensure a homogeneous interface for the heterogeneous hardware environment. Thereby, increase the portability of the services and applications hosted on top of the BOS, as they are decoupled from the specific hardware implementation of the building. Also, in the case of XBOS and Brick (see page 26), it provides discoverability of these devices. The common interface is achieved by two components: the drivers and the communication bus. The drivers pick up the data from the specific implementations of hardware, and then modifies it into a standardized format, and pushes this information into the communication bus. If the system has a time-series database, which could be one of the services, it would pick up these messages and store them in the database. The drivers do the same, but reversed, for the actuators. Drivers are, therefore, hardware-specific and need to be written to support a given piece of hardware. Often they are minimal programs, and they can be distributed among any number of physical machines. The component of the common interface is the communication bus. This component provides one standardized method of accessing all information, and in combination with the drivers, both formats are known. The communication bus is explained in more detail on page 18. The last components in this layer are the Triple Store and the Brick model. These are optional, but enables the system to model the physical context of the built environment, and therefore provides important context for the services and applications. The homogenized hardware interfaces can be discovered through these components. For the Triple Store, several implementations

exist, such as HodDB[48], Fuseki [49], and the Service Support System (SSS) contributed later in this Ph. D. thesis [5].

Service and Application Layer: As mentioned, one of the primary purposes of a BOS is to provide a platform for services and applications, which is what is contained in these layers. Often, these layers are depicted as one application layer. There can be several reasons for this. For example, some BOSs expect all services needed for an application to be provided by the BOS, while others do not differentiate between the notion of these two artifacts. For the purpose of this Ph. D. thesis the two artifacts are distinguished by these features; services primary purpose is to serve other services or applications by providing functionality, or added informational value, like forecasts, while applications primary purpose is to orchestrate control of the BOS elements, or to serve the informational or configurational needs of human operators. As these can often be combined, a program in these layers does not necessarily only belong to one of these layers.

2.4 COMMUNICATION BUSSES AND AUTHENTICATION

Several service busses exist in the software architecture space. In this Ph. D. thesis, two implementations are used actively; Bosswave [47] and Kafka [50]. In regards to the contributed research in this thesis, Apache Kafka is only used in paper [6], while Bosswave is used in the papers [2, 3, 4, 5]. The Apache Kafka and Bosswave implementations are detailed below.

2.4.1 BOSSWAVE

Bosswave [47] is a communication bus that is designed to work in the XBOS [28] context, and has in the later iterations of this BOS been implemented as a central component. Bosswave transforms the concept of, BOSs to a potentially globally distributed and cryptographically secured BOS built on top of the blockchain and based on microservices. Bosswave is an overlay network situated on the internet that provides secure communication between IoT devices and building applications. Bosswave uses the blockchain and its inherent smart contracts to encrypt the communication between agents, that maintain the network

and blockchain. These agents, together with the smart contracts, enforce the encryption by using a public key that is published in the blockchain, and a private key that only the creating user has access too. The following concepts and artifacts exist in the Bosswave architecture:

The Blockchain and the Data-plane: Two distinctions are essential to understand about Bosswave; what is maintained in the blockchain, and what is maintained in the data-plane. First, The blockchain is a technology typically known from several types of digital currency [51, 52]. Bosswave itself is based on the Ethereum codebase for the non-data transactional parts of the system, but secures data transmissions in the data-plane with the keys used to secure blockchain entries into the blockchain. Bosswave itself uses Ethereum as its foundation for a public ledger and to control and enforce usage rights and privacy in the Bosswave Communication Bus (also called the Bosswave Syndication Bus). While all Declaration of Trusts (DoTs) (Permissions), namespaces, and other entities are saved in the blockchain, all actual data transmitted over the network only exist on the data-plane, which is managed by the Agents, that also maintain the blockchain itself.

The Bosswave Economy: As Bosswave is based on Ethereum, it also has a monetary component. Therefore, all changes to the blockchain require money, referred to as "fuel", to make a change to the blockchain. Therefore, creating new permission costs fuel, as does creating a user, a namespace, a wallet for the money, or any other action that does not pertain to the data-plane, aside from reading the blockchain. This fuel also needs to be generated somehow. To do this, the Agents also have built-in mining functionality. Note, mining is somewhat of a misnomer, as they help maintain the Bosswave network once this function is enabled, and thereby the mining is compensation for helping to maintain this network. Being able to contribute to the network also enables anyone to join it, as all they need to to help maintain it for some time to generate fuel.

Agents: These Agents are the centerpiece of the decentralized Bosswave network. They act as maintainers of the blockchain, facilitate the data-plane data transmissions, act as routers for the namespaces (think domain name), and also effectively functions as an independent Domain Name Server (DNS). As the Agents control the

data-plane, they effectively facilitate the communication between the drivers, services, and applications that operate on top of Bosswave.

Namespaces and Routing: A namespace is functionally what a domain name is on the internet. Namespaces are used to collect resources into a logical path, just as we know it from a website Uniform Resource Identifier (URI). Namespaces are paid for by fuel and then bound to an encryption key, much like cryptocurrency. This key is the root owner of that namespace and is the only key with access to anything below that namespace. As mentioned, finding resources is done by URIs, such as ”domain.demo/some/path”. These URIs are read/write protected to anyone not possessing the key owning the namespace. It is, however, possible to delegate parts of a URI, for reading or writing, to another key. This action is called a DoT, that effectively ensures URIs have granulated privacy, and at the same time ensures only keys with the right DoT have access to the encrypted data at a given URI. A namespace, much like an internet domain, needs to be hosted by an Agent. Once an Agent is set up to be a Router, the namespace owning key can grant the newly appointed Router rights to administer the data-plane for the given namespace. This Router functionality is saved in the blockchain, that in turn is distributed to all the Agents, effectively implementing DNS functionality. All an Agent need to do to find an agent maintaining a given namespace, is to search the public blockchain.

Entities: The entity concept in Bosswave can be confusing, as it has one form, with many functionalities, but essentially in the blockchain world just signifies a private key. These entities can be representative of a wallet containing fuel, a user of the system, a namespace, and more. Functionally, all of the above are the same type of entity but has different use cases. For example, all entities have wallets attached to them, but could also represent a user with several DoTs associated to them, thereby granting them access to resources, while the same entity might own a namespace. While all of these can be the same, it is strongly encouraged to keep separate entities for each of these use cases.

Declaration of Trust (DoT): As was already preluded to, DoTs are entries into the blockchain, that signifies one key, or entity trusts another. This trust is validated by the key granting permission. DoTs have

the option of allowing a key to trust another to trust the third key, or even deeper, thereby creating a graph. This trust delegation is implemented using the Time to Live (TTL) value of a DoT, where a TTL of 3, means that the key is allowed to delegate trust to three levels below it. DoTs can be revoked in the middle of the graph, which would invalidate all DoTs granted after the revoked DoT. An interesting feature of Bosswave and the DoT system, is that DoTs can be granted across buildings or systems on a global scale. This functionality means one university can give another one access to one specific part of the hardware, or any other part of a URI.

2.4.2 Apache Kafka

Kafka [50] is another type of communication bus. While Bosswave is a research project, Kafka is an industrial-grade software product. Therefore, it implements High Availability (HA) functionality, load balancing features, and allows data published into the bus to be replicated between multiple servers if need be, as well as tracking how far a given service was last time it read from Kafka. All of this is functionality Bosswave does not possess. Also, a Kafka setup is created to be used by a confined set of system owners and is not globally distributed as Bosswave is.

While Bosswave uses URIs, Kafka uses topics. These are fundamentally different, as URIs can impart knowledge of the data being accessed, and the DoT system of Bosswave can grant permissions to anything below a specific part of the URI. With Kafka, information like this would manually need to be embedded into the string-based topic names, and granting permissions to specific streams are more cumbersome, and cannot be used to dynamically grant access to more resources over time, given a specific prefix, as with the URI solution.

2.5 DR and the Software Space

A range of methods has been proposed for how to implement automated DR in software. A main categorization of the methods can be obtained by considering the type of DR service, the targeted loads and the type of grid integration. For the type of automated DR services,

	DR Service Type	Targeted loads	Grid / Building Roles
Vishwanath et al. [53]	energy	space cooling	in-direct price signals
Neupane et al. [54]	energy, flexibility	heat pumps, electric vehicle, household appliances	collaborative direct control
Hajiesmaili et al. [55]	energy, flexibility	electric vehicle, building battery	centralized direct control
Xia et al. [56]	energy, flexibility	residential loads under user control	decentralized in-direct control
Comden et al. [57]	energy, flexibility	residential loads under user control	decentralized in-direct control
Frazetto et al. [54]	energy, flexibility	household appliances	collaborative direct control
Barth et al. [58]	energy, flexibility	industrial processes	decentralized direct control
de Hoog et al. [59]	energy, flexibility, contingency	building battery	centralized in-direct control
Khonji et al. [60]	energy, flexibility	electric vehicles	centralized direct control
Nellemann et al. [20]	energy, flexibility	ventilation	collaborative direct control

Table 2.1: Automated DR methods categorized by DR service type, targeted loads and grid/building roles.

we follow the categorization proposed by Olsen et al. [61]. The categorization includes: **regulation** as a response to random unscheduled deviation; **flexibility** as an additional load following reserve for large un-forecasted wind/solar ramps, **contingency** for rapid and immediate responses to a loss in supply, **energy** to shed or shift energy consumption over time and **capacity** to provide an alternative to generation. The targeted loads can be categorized by their broad type including space cooling, space heating, water heating, electric vehicles, indoor lighting, ventilation, refrigeration, pumps (e.g., pools and wastewater), computing equipment, manufacturing equipment, building battery among others. The main differences among these loads are how fast they can ramp-up/down and if they have a storage capacity. The type of grid integration captures the integration between the role of the grid operator and the building owner. This can at the one extreme be a centralized direct control with the grid operator in charge and at the other end be an in-direct control by price signals where the building owner decides on load activation. In between is different collaborative schemes that optimize the benefits of offering and activating DR between the grid operator and building owner.

Table 2.1 list our classification of recently published methods. The listed methods span the different types of DR services, loads and grid integrations. A particular building owner, to optimize cost benefits of participating in DR, will run several of such methods at the same time to participate in different services. The services will also change over time depending on changing grid situations and thereby economic incentives for delivering different DR services. Considering the proposed methods, this highlights the many different DR services that will be running on future BOS implementations for buildings. Furthermore, each of the listed methods in Table 2.1 utilize a range of other services for accessing real-time and archival building data, executing control, forecasting services, external services (e.g., weather forecasts, energy price signals). However, this is in contrast to how most applications for buildings today are built as monolithic applications where all analysis, processing and GUI functionality is contained in one or few applications.

2.6 RDF ONTOLOGIES AND METADATA

Ontologies have their roots in language and the way we communicate and are used to describe the relationships, and meaning between, and of, concepts. The inheritance becomes abundantly clear when studying the makeup of Resource Description Framework (RDF) [62] that is used to describe these ontologies. RDF consists of triples. Each triple consists of a subject, an object, and a predicate. For example, "John sells Iron", has the subject "John", the object "Iron", and the predicate "sells". From this triple, we can assert that John sells some kind of Iron; it being an Iron for ironing clothing or the material. As this can be confusing and is not useful for machine readability, RDF uses URIs to signify the specific meaning of each concept. For example, Iron could have two separate URIs, "acme.corp/materials#Iron", and "acme.corp/home-appliances#Iron". The URI signifies the specific meaning of a given concept, which is agreed upon by the people using the ontologies. From an RDF perspective, the above triple would look something like this:

- acme.corp/employees#John
- acme.corp/general#Sells
- acme.corp/materials#Iron

Multiple of these triples could be added, and assert several facts about the involved elements, like "John is a Person", "John is an Employee", and "John sells Iron". Effectively, the triples above generate a graph, that represents knowledge of several separate facts, that together create a pattern that can be read by computers. It is essential to understand that RDF concepts do not need to all come from a single Ontology, but that they can be mixed, which is also the intention of these. Compared to typical metadata approaches that tend to use key-value stores [63], the semantic approach of RDF is significantly more expressive.

Several uses of ontologies have been observed, like the semantic web approach, where for example, websites are marked up with ontologies to make them machine-readable and infer meaning to the website for a machine. The common approaches to marking up these websites are technologies like Resource Description Framework in Attributes (RDFa) [64] and JavaScript Object Notation for Linked Data (JSON-LD) [65]. RDFa is used to annotate Hypertext Markup Language (HTML) tags with the ontology constructs, while JSON-LD generates a separate JavaScript Object Notation (JSON) section in the HTML

document, replicating the data present on the site, but keeping the added information together in one place. Both of these approaches allow web crawlers to locate the added data and generate the knowledge graphs for themselves. The knowledge graph can even be extended from several independent websites, where unique identifiers were used to, for example, identify a specific journalist, whom this person works for, and more. Also, several practical examples have been seen implementing JSON-LD, such as Googles Rich Snippets [66, 67], that generate unique visuals when users are searching for their sites, such as Cinema times for Movies, and their ratings. Schema.org [68] is an ontology that is typically used for this kind of website markup, using RDFa and JSON-LD technologies to do so. Therefore, it contains concepts that are commonly used to describe elements found on the internet. Examples of these include an extensive array of company types, opening hours, products, ratings, event descriptions, and more.

As mentioned, RDF based ontologies are created to be machine-readable, and to do so, SPARQL Protocol and RDF Query Language (SPARQL) [69] is most commonly used. SPARQL is a querying language, much like the commonly known Structured Query Language (SQL) language, that is specialized for an RDF context. The language functions differently, as it defines the pattern it is searching for, by asserting several facts first, introduce the variables needed to define what information we want in the output, and more. SPARQL is effective, but queries are more cumbersome to create than traditional SQL.

In some cases, entire abstractions are created on top of RDF. One such abstraction is Web Ontology Language (OWL), that introduces several concepts of which the following are the most important to understand this Ph. D. thesis (Concepts like reasoners are excluded):

Classes: These functionally resemble classes also known from Object Oriented Programming (OOP). They function as the concept for something but also acts as a template for how a copy of it should be modeled. Classes can have Data Properties, Object Properties, and can inherit their properties from other classes.

Inheritance: This is also a concept known well in OOP, which allows Inheritance of functionality, properties, and more. Of course, OWL does not implement functionality but describes it. Therefore, Data Properties, Object properties, and conceptual understanding are inherited. An example of this conceptual understanding

could be the Class Employee inheriting from the Class Person, and therefore, by definition, also inherits all definitions made to the Class Person, thereby implying an Employee is also a Person.

Individuals: An individual would in OOP be regarded as an instance of a Class. An example could be John from earlier, that is an individual of the Class Employee, that again, because of Inheritance, is a Person.

Data Properties: Data properties are simple data attached to the Individual of a Class. For example, John could have the data property Age with the value 42.

Object Properties: In OOP, an object property would most aptly be described as an array of instances of a specific class. Simply put, it refers to other Individuals. It is not this simple, however, as Object Properties also allow for Inheritance as mentioned described above.

In this Ph. D. thesis, the ontologies, and the specific usages of that ontology are typically split up in two. This specific usage of the ontologies is referred to as "the Model". So, the ontology specifies what can be described, but the Model referred to what was described using the given ontology, or a mix of ontologies.

Several related ontologies exist, such as IoT-Lite [70], OWL-S [71], and WSMO [72]. One of these related ontologies, called Brick, is especially important to understand when reading this thesis, as it is referred to often. Brick is presented in the following section.

2.6.1 BRICK

The Brick [73, 74] Ontology is a central component in this Ph. D. thesis for several of the software solutions. It is an RDF based ontology that does not use the abstractions introduced by OWL. Brick is, an ontology used for describing the physical properties of a building, such as hardware, physical locations, and the relationships, thereby capturing the knowledge of how a given building is constructed. The ontology is built keeping BMS in mind, to allow for descriptions that span several types of systems. Applications then use SPARQL to query the model, to discover the building hardware and the context in which it is placed. As the queries are dynamic per building and the associated Brick model, it allows the application to be built for more efficient portability between

buildings. The model, or the specific implementation of Brick, can take a long time to make, and how a building is described similarly each time, is one of the main problems with this approach to building metadata discovery. However, one example where Brick can bring substantial knowledge to the applications, could, for example, be in the case of Heating, Ventilation, and Air Conditioning (HVAC). For HVAC, often complex duct systems with several components are introduced, and the order of which and placement is essential, and directly impacts how the system works. They contain heat exchangers, dampers, valves, condensers, evaporators, air intake, and much more. How these relate to each other, and what setting they are in, relate immensely to its impact on the rooms air quality and temperature. An HVAC system also typically spans a multitude of rooms, and loads can be distributed between several of these, where more than one can end up contributing to one room or area. This is an elaborate description, but it can be essential for some applications to understand how it should be configured. For example, if a service is to predict the temperature of a room a few hours from a given point and with specific HVAC settings, the service could now use a white box model based on the knowledge in the Brick graph to make considerably better predictions.

2.7 TOPIC GROUPING

To make it easier to follow the topics of the Chapters in the Technical Contributions Part of this Ph. D. thesis, each chapter has been associated to different areas. Figure 2.2 details the different areas. Motivations & Barriers relates to both physical and technical barriers, but also to how retail store stakeholders see DR in their stores, and their concerns. Hardware & HAL relates to the physical sensors and actuators that can be deployed in a BOS. Also, it relates to the HAL related technologies, like Drivers, Brick (see page 26), and more. Services & Applications is the code that enables extra functionality on top of the BOSs. The difference between these has already been discussed in the BOS section of this chapter. Application Portability refers to the ability to move service or application code from one implementation of a building to another. Deployment refers to the deployment and maintenance phases of a BOS life-cycle. Supporting Ecosystem refers to the stakeholders involved in the BOS life-cycle. This includes their incentives, impact of

changing environments, or changes to the tooling.

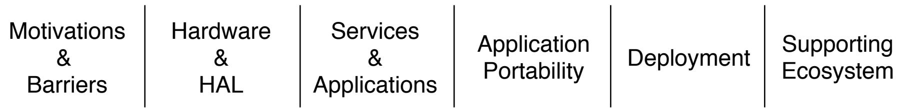

Figure 2.2: Overview of areas touched upon by this Ph. D. thesis, which is repeated in each chapter, underlining the related areas.

Each chapter will have a copy of Figure 2.2 at its beginning, with directly related areas underlined by red, and indirectly related areas marked by yellow. This should allow the reader to better understand where the Chapters contribute to the BOS ecosystem and the Ph. D. thesis' findings.

Part II

Technical Contributions

This part details the contributions made, but summarized in chapters grouping research output into topics. They are designed to create an overview of the different topics and the papers contributions, but leaves some of the details to the papers in Part IV. Note, parts of these chapters come directly from the papers, but are presented without quotation marks.

Chapter 3 details the exploration of four physical retail stores. Chapter 4 details a service created for Building Operating Systems (BOSs) to give DR applications better decision points. Chapter 5 details the Service Abstraction Layer, that helps BOS services and applications to be portable between buildings. And finally, Chapter 6 details two software applications created to mitigate the deployment costs of BOSs in retail stores.

CHAPTER

RETAIL STORES AS FLEXIBLE CONSUMERS

3

This chapter discusses paper [1] (The Retail Store as a Smart Grid Ready Building: Current Practice and Future Potentials). The paper covers the exploration of Demand Response Potentials (DRPs) in retail stores. Also, the paper details an inspection of several types of retail stores to gauge the state, as well as the barriers towards implementing Demand Response (DR) applications in these stores. Section 3.1 introduces and motivates the contributions, and Section 3.2 summarizes the main contributions of the papers as related to this chapters topic.

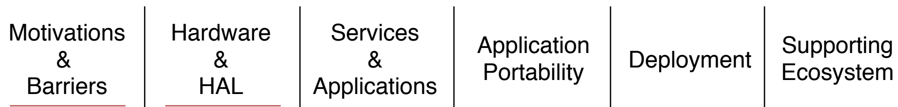

Figure 3.1: Overview of Areas examined in this Chapter.

3.1 INTRODUCTION

To better understand the retail stores, their technical state, and their motivations, this chapter concentrates on exploring these. This chapter reports the basic understanding of the retail store state, that is the basis for the rest of the work in this thesis.

Generally, in the retail sector, DR has been considered from two different perspectives:

1. Retail stores as commercial buildings and using DR strategies known from other types of commercial buildings for light and ventilation control [18].
2. Retail stores with ample cooling and freezing loads and using these for load shifting (e.g. Shafiei et al. [33]).

Therefore, previous work has failed at embracing the retail store as a separate building type for DR and apply a holistic view on the options for flexibility in consumption in the stores at large. For instance, in office buildings the goal is to optimize the indoor environmental conditions so that occupants can work effectively. When examining retail stores, a goal is to optimize the indoor environment to boost sales, e.g., in general, a high level of light and a very high level on premium wares. Highlighting just one of the many differences when comparing retail stores and commercial buildings in general.

This chapter summarizes the results of a holistic study of the flexibility options in retail stores. Instead of a top-down approach looking at the statistics from several stores, a bottom-up approach is applied by screening the stores. Four store types were surveyed:

1. Hypermarket.
2. Supermarket.
3. Garden Center.
4. Hard Goods Store.

A hypermarket is defined as a store ranging from 5000 to 15000 m^2 and have a supermarket and department store section. A supermarket is defined as a store between 1000 and 5000 m^2. These store types were chosen as they represented different sectors with distinct concerns, were willing to put in the time and resources needed for surveying the stores, and giving feedback. All the stores are located in Denmark, and therefore local installation and weather conditions apply. The screening of the stores themselves was done by AURA Energi, which specializes in energy optimization of buildings. Each store has been screened to gather information on each device in the store. The usage and flexibility potential for each type of device has been estimated based on technical documentation, information from store staff and energy metering data.

3.2 CONTRIBUTIONS

This section details the contributions of each paper related to this chapters topic.

3.2.1 Main contributions in paper [1]

3.2.1.1 Contribution 1

The first contribution is an overview of the four retail stores total observed consumption divided into categories. This consumption, can be found in Table 3.1. Units are in MWh per year. The categories used are the largest common groupings found in the screening data. The difference in consumption between the first five categories and total usage is listed in the unidentified load column.

Notably, the electricity usage differs greatly between the retail store types. Particularly the usage percentages between categories. Besides the *hard goods store*, all the other stores have a high percentage of loads placed into the *other* category. These loads are small machines used for various purposes, mostly in the store floor, such as lighting products, aquariums, TVs, and more. These devices will be a hard task to include into the DR activities as they represent a large mass of devices that does not cluster well into a category with similar purposes. The *garden center* is notable in this regard, as about 41% of the loads fall into this category.

The *hard goods store's* unidentified electricity usage is also interesting as the *lighting* and *ventilation* is controlled on a schedule and only few parameters would have an impact on the calculated numbers. This suggests a considerable amount of the unidentified loads would have a good chance of belonging to the *other* category. The above observations point strongly suggest the content of the stores are diverse, and differ immensely from each other based on the category of products the store sells.

The fact that *ovens* ended up being a considerable part of the total usage of the *supermarket* and *hypermarket* is interesting, and makes the *ovens* an obvious candidate to examine further. The DRP, the potential a device can potentially contribute to DR operations, of these devices is considerable, but these ovens are tightly integrated into the processes of the store, which makes it difficult to leverage the potential. Furthermore, using these devices to heat up a store is impractical. Nevertheless, the numbers show a potential candidate for integration if a practical usage pattern can be discerned.

	Lighting	Ventilation	Cooling	Ovens	Other	Unidentified	Total Usage
Garden Center	133 MWh	-	-	-	113 MWh	27 MWh	273 MWh
Hard Goods Store	212 MWh	128 MWh	-	-	12 MWh	205 MWh	557 MWh
Supermarket	478 MWh	141 MWh	648 MWh	364 MWh	154 MWh	116 MWh	1901 MWh
Hypermarket	967 MWh	613 MWh	847 MWh	462 MWh	715 MWh	205 MWh	3809 MWh

Table 3.1: Consumption across categories, Total observed consumption, and consumption which is unaccounted for.

	Lighting	Ventilation	Cooling	Ovens	Total Max Usage
Garden Center	35 kW (100%)	-	-	-	35 kW (100%)
Hard Goods Store	131 kW (56.7%)	100 kW (43.3%)	-	-	231 kW (100%)
Supermarket	77 kW (9.2%)	144 kW (17.1%)	113 kW (13.4%)	508 kW (60.3%)	842 kW (100%)
Hypermarket	176 kW (15.7%)	591 kW (52.7%)	143 kW (12.8%)	211 kW (18.8%)	1.121 kW (100%)

Table 3.2: Demand Response Potential (DRP).

3.2.1.2 Contribution 2

The second contribution pertains to detailing the DRP of the four retail stores. Table 3.2 shows this DRP, divided into the same categories as Table 3.1, with two exceptions, as *other* and *unidentified* are excluded. *Other* is excluded, since enabling all these diverse devices to respond to DR requests would be impractical, complex, and extremely expensive; and *unidentified* because a load needs to be identified to determine its DRP. Devices with unidentified wattage are not included in this table. The DRP is represented by the wattage of a device, or in this case, the accumulated possible wattage used at any given time. To attain a more holistic study, sub-monitoring was decided against as it would be limiting the paper to devices that could be practical to sub-monitor.

In the total wattage column, the table shows the *supermarket* and *hypermarket* are capable of delivering the largest amount of DRP compared to the total usage. The potential is largely attributed to the *cooling*, *ovens*, and *ventilation*. Here the immense overcapacity of the ventilation system of the *hypermarket* stand out, as well as the oven capacity of the *supermarket*. The ovens take up about 60% of the *supermarkets* maximum DRP, dwarfing the rest of the stores' loads. The *hypermarket* is able to deliver about 19% of its maximum DRP from these ovens. These numbers definitely make the category hard to ignore. The ventilation systems, especially of the *supermarket* and *hypermarket*, are also able to deliver substantial DRP when compared to the less flexible *lighting* systems and *cooling* systems. One possible explanation for this discrepancy, though, could be the hugely over-dimensioned ventilation systems compared to the stores' actual needs.

3.2.1.3 Contribution 3

A future system that controls a multitude of devices, like in a retail store, needs programmable access to the devices it has to control. As systems become larger and more complex, interoperability and open access to these Application Programming Interfaces (APIs) become a priority. Therefore, the third contribution, detailed in Table 3.3, shows the DRP in kW based on how devices are controlled. Only devices identified and within the categories listed in Table 3.2 are included. *No Control* means the devices are only controlled by turning a switch on and off. *On-device control* means the individual device is capable of

being configured to respond to the environment in a limited fashion. The *Building Management System (BMS)* columns are divided into two categories. The first, *closed*, is a typically proprietary and closed system that external programs cannot interact with. The second category is for *open* BMSs that provide external integration access via APIs or other means.

When analyzing the available retail stores for DRP, it is evident the current state is found severely lacking in regards to control options. The control distribution in Table 3.3 indicates that only the *hard goods store* actively engages in controlling the building from their BMS. The *hypermarket* and *supermarket* follow behind, but are both terribly behind when considering what their BMSs are capable of doing with these devices. Most of the devices connected to the BMS can only have their status observed. These systems would require extensive retrofits to be leveraged for DR. The garden center is even worse off as 0% is controllable from a BMS. The findings in the control distribution suggest a considerable attitude change is needed when building new stores. Not only should these stores have their loads connected to a BMS or equivalent, but they should also strongly consider the importance of open accessible BMSs that can be interfaced to via programmable and integrable APIs. Not a single store has one single device that can be controlled openly, which effectively leads to the definite conclusion: 0% of the screened stores are compatible with an external signaled DR activation strategy, unless they effectively replace the current BMS. Current research in Building Operating Systems (BOSs), that also facilitates DR, likewise requires open access to the devices. Last but not least, all the stores observed only controlled the lighting in large clusters that did not fit with the floor area. As a consequence, it is hard for a store owner to differentiate certain sections of a store by using the BMS. Better granularity could give the store owner more flexibility. This granularity would also open up for a possibility of introducing DRP, reducing light usage in certain sections of the store where it is less important.

3.2.1.4 Contribution 4

The fourth contribution pertains to DR mobilization limitations. Devices react differently to changes requested by a DR event and can have limits in the frequency and duration, it can be used. Table 3.4 seeks to clarify

	No Control	On-Device Control	BMS - Closed	BMS - Open
Garden Center	449 units / 35 kW	-	-	-
Hard Goods Store	-	-	3886 units / 231 kW	-
Supermarket	315 units / 530 kW	2 units / 113 kW	891 units / 200 kW	-
Hypermarket	980 units / 247 kW	7 units / 153 kW	2868 units / 722 kW	-

Table 3.3: DRP overview based on how they are controlled.

	Lighting				Ventilation				Cooling				Ovens			
	Frequency	Duration	Ramp Up	Ramp Down	Frequency	Duration	Ramp Up	Ramp Down	Frequency	Duration	Ramp Up	Ramp Down	Frequency	Duration	Ramp Up	Ramp Down
Garden Center	Q	U	S	S	-	-	-	-	-	-	-	-	-	-	-	-
Hard Goods Store	Q	U	S	S	Q	U	M	M	-	-	-	-	-	-	-	-
Supermarket	Q	U	S	S	Q	U	M	M	Q	U	M	M	D	H	M	S
Hypermarket	Q	U	S	S	Q	U	M	M	Q	U	M	M	D	H	M	S

Table 3.4: DR Mobilization Limitations..

the DR mobilization limitations found in the inspected systems. The values of the diagram are split into the following possibilities: **S**econds, **M**inutes, **Q**uarter hour, **H**ours, **D**ays, and **U**nlimited. Each value type implies the metric used is lower than the next, so 10 minutes would be M, but 25 would be Q. A notable trend in Table 3.4 is how the categories are aligned over the different retail store types. Response times are within minutes of each other and fall into the same time category.

Table 3.4 details mobilization limits on the systems inspected. The mobilization limits align in each category. This means some generalization in implementation and mobilization can be expected in the sector as this is the case in all four categories. The ramp up and down times listed, are typically not instant due to the physical properties of the devices. The *ovens* is the category that differs most from the other categories as they have several extra limitations. The ovens could theoretically be turned on and off instantly as often as necessary. However, this kind of flexibility is unrealistic as the load would be unpractical to move more than once or twice a day. A baking program can take several hours to complete, and after the dough has raised it needs to be baked relatively quick. Baked bread typically also needs to be ready at specific times of the day. These are natural limits for these ovens as it would otherwise conflict with the retail stores' goals.

3.2.1.5 CONTRIBUTION 5

The fifth contribution is an account of the experiences learned throughout the collaboration and is also discussed in paper [1]. To not inconvenience the reader, the rest of this section is a verbatim copy of the stakeholder discussion, so it is clear what was discussed.

During the screenings, observations, and conversations with staff, it became apparent several factors were different, when creating systems for retail stores compared to homes and offices. To successfully implement DR and leverage DRP in a retail store setting, it is important to understand how it differs from other settings. Especially motivations for controlling the building, and how the environment of the building differs. In this work we combine our own observations with those of Zheng et al. [75] who outline motivations for the retail sector to provide DR based on a literature study.

The focus on sales drives the store owners desire to control the building environment. The purpose is to create an environment that motiv-

ates the customer to buy. Many of the processes that happen in a store are time sensitive and directly related to loads of the store. This can also be said for offices and residential homes. However, a customer will rarely wait ten minutes extra for their purchase to get ready because an oven is turned off. Situations like this results in lost sales, while it would be a minor inconvenience at home or in an office. The same holds true if any load fails for any reason while customers are present. Especially lighting is sensitive and directly impact sales by drawing attention to products. This impact is often leveraged by adding extra lighting around and above a product showcase. The settings and setup of these showcases and floors are often changed when high profile products change. As this behavior directly impacts sales, it is important the store can maintain a differentiated lighting setting. And as such, there is a need for a more dynamic lighting configuration.

All stores agree that schedules and temperature levels are set centrally and typically encompass multiple stores. As mentioned above, the staff needs to be able to change several settings on demand but should have no control over other parameters of the store. As a consequence, a complex security scheme is required, if the loads around the products are to be leveraged for DR purposes. The standard user and groups scheme with granular rights management on specific loads should suffice. Central control should be seen as an opportunity to make DR a central decision for management, to control multiple stores instantly and consistently.

Currently, the size of the regulation zones are large in stores, but there is a clear desire for more granular control, as the sales area frequently changes. The sales floor is typically the largest room with many control zones close to each other. Consideration needs to go into how each control zone impacts the other visually; as one end of the store can often be seen from the other end. Any distraction is a potential lost sale from the perspective of a store owner.

A retail store typically seeks to facilitate an experience where the customer does not need to leave the store. As a consequence, it is usually possible to buy food, as well as to see the products showcased and running. Food preparation and product showcasing use electricity, as does the the butcher's section. These areas use specialized tools and leads to a vast array of machines. When comparing these devices to the other domains, a retail store's load diversity resembles a residential home better than that of an office. If the *other* category is to be leveraged

as potential, a complex task of homogenizing these devices, needs to be solved. This problem is prevalent in both the retail and residential domains.

The occupancy patterns also differ. Residential homes occupancy and usage patterns are dependent on the occupant's work and life patterns, while offices occupancy and usage pattern typically depend on work regulations and the type of work. In retail, the largest loads typically follow schedules and processes in the store. These typically start when the employees arrive at the store and then escalate when customers arrive; new processes start, and food preparation devices and registers are used. These patterns allow for relative predictability in several areas of the store.

If the requirements of the retail store are not taken into account, potential solutions will not seem palatable to retail store stakeholders, as it will make several processes in the store cumbersome. An example mentioned earlier is the simple need to be able to rearrange the floor to accommodate sales of a new premium product. Needing to have personnel from a central entity called in to accommodate these changes is a clear hindrance for adoption in retail. Many retail store loads are intertwined with processes, like using the ventilation system for drying the floors, as well as the ovens. It could prove useful to map these processes and how they are related to the loads. Having a clear overview could potentially help future products leverage loads that would otherwise seem unfeasible to integrate in DR operations. As such, further study should be undertaken into the specific demands and difficulties retail stores have in order to help increase the incentives for stores to invest in these solutions; thereby increasing the potential participants in DR programs.

3.3 CONCLUSIONS

This chapter has provided an overview of the screening of four different types of retail stores. This overview has been detailed from four different perspectives: 1) the yearly usage in a categorized manner, 2) the DRP, 3) how the devices are controlled and at what level, and 4) challenges to leveraging the potential in retail stores.

Some of the important findings from the screenings are as follows. The *Other* category from Table 3.1 shows a relatively large number

of diverse devices. These devices are hard to integrate into the DR as they have very few characteristics in common. The inclusion of *Ovens* in Table 3.2 show a significant potential. While the inclusion of ovens into DR is highly questionable because of their usage patterns and their tight integration into the processes of the retail stores, the immense numbers speak for themselves. These tight integrations are evidenced by the limitations set on the mobilization limits in Table 3.4. The Supermarkets' ovens can produce about 60% of the total DRP. Even in a store with fewer ovens, like the Hypermarket, the ovens provide almost 19% of the total DRP. Currently, several of the loads in a retail store tend to be intertwined deeply with processes happening within the store. To successfully leverage the potential of these loads, a mapping is to be made between the loads and how and when they are used in the processes.

The ventilation system of these stores surveyed also stand out and shows good potential for integration as 43%, 53% and 17% of the potential lies here. The *Supermarket* with 17% is only this low because of the aforementioned *Ovens*. Also, the *Hypermarkets* ventilation system was especially interesting because of the over-dimensioned capacity of 591 kW.

Regarding external control of the devices currently in the store, the picture is rather bleak as not a single device in the screened stores can be controlled externally. There is simply a need for change in how we build systems for buildings. With the direction current research is going in BOSs, future loads need to be accessible externally by default, if they are to be integratable. Also, due to the changing showcases and floor layouts, and desire for central management, it is important to implement a solution that is operable and changeable from the store alone, but also from a central management position. Distributed user and group functionality are a natural consequence of this requirement. Fortunately, retail stores are regularly renovated, and therefore an opportunity exists for retrofit and introducing the required technologies. Of course, this kind of retrofit requires that a business case can be established. However, with some of the mentioned discovered requirements, and by mapping more of this sector, a feasible solution could potentially be found. If found, it should be possible to retrofit retail stores at a faster pace than other building types.

The retail store screenings show tremendous DRP even though there is a multitude of challenges to leveraging this potential. This chapter

has detailed the motivations and difficulties found in the screenings, but much work remains in both charting these challenges, as well as the search for solutions. They are solvable though, and further research in this area will undoubtedly help leverage previously untapped potential.

CHAPTER

ACTIVITY TRACKING FOR BETTER DR DECISION MAKING

4

This chapter discusses paper [2] (Activity-Tracking Service for Building Operating Systems). The paper [2] covers a service for Building Operating Systems (BOSs) that models activities in retail stores using dynamic state machines, thereby creating better decision points for Demand Response (DR) purposes. Section 4.1 introduces and motivates the contributions. In Section 4.2 related work is discussed. Finally, Section 4.3 summarizes the main contributions of the papers as related to this chapters topic.

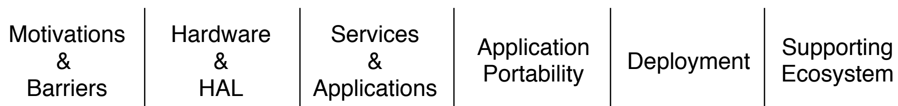

Figure 4.1: Overview of Areas examined in this Chapter.

4.1 INTRODUCTION

This chapter proposes an Activity-Tracking Service (ATS) for BOSs. The purpose of the ATS is to enable developers to write applications that respond programmatically to the activities occurring inside buildings and thereby allow for better and more efficient control of the energy consumption. One example of such control is DR where loads respond to grid-side requests. An exploration into the resistance to DR acceptance in retail stores points to worries about the consequences to the consumer experience. Understanding how staff and customers are using the loads of the store help alleviate these concerns, while also enabling developers to better understand how the store is being used. The first step in this understanding of the human activities inside build-

ings is modeling the activities and making their current state available via Application Programming Interfaces (APIs). Activities could be tracking if a room or area is currently being cleaned, or at what stage of cleaning it is in. Another example could be refrigerators in a store being filled, while the cooling mechanisms optimize when they should cool and how much, based on the state of the cycle. An added benefit of centralizing the understanding and modeling of activities is the added benefit of allowing multiple applications to track a single activity. Additionally, this also allows the domain experts, the store owners or hardware vendors installing the sensors and actuators, to define the related activities. This design would ensure the models accuracy relative to the store's procedures, as this is where the understanding and expertise in relation to the activities in the stores are located. The service is designed to consider the security, privacy, integration, extendability and scalability challenges in the building setting.

4.2 RELATED WORK

Currently BOSs do not consider modeling of the activities happening inside of the building itself. Attempts have been made to model activities within context-aware computing and activity-based computing before as, for instance, manifested in systems like JCAF [76] and CARL [77]. To get a better overview, Table 4.1 details the differences between these systems.

Building Depot 2.0 was developed for office environments and with the goal of being used as a BOS. Therefore it supports a building's infrastructure as a natural consequence. The system is built as a single node, which hampers the scalability of the system as the only option is to scale vertically, or alternatively set up several autonomous systems. For more extensive systems, such as citywide deployments, Building Depot 2.0 would not perform well. To extend the functionality of Building Depot 2.0, a user would have to write code and recompile. However, it is possible to integrate functionality by using a REST API. In regards to activity modeling, Building Depot 2.0 has nothing to offer. This kind of functionality would have to be custom built.

Just like Building Depot 2.0, eXtensible Building Operating System (XBOS) and Bosswave are built for office and teaching environments and has extensive support for the building infrastructure. Compared

	Intended Deployment Area	Support Building Infrastructure	Horizontal Scalability	Service Extendability	Application Functionality	Activity Support
CARL [77]	Residential	–	Single Node	–	–	Label activities
JCAF [76]	Hospitals	–	Single Node, Java	Code Additions	RPC/Java	OO Data Model, Models Activity State
Building Depot 2.0 [29]	Offices	✓	Single Node	Code Additions	REST	–
XBOS + Bosswave [27, 28, 47]	Offices	✓	Cluster of Nodes	Services	Bosswave	–
ATS + XBOS + Bosswave	Retail	✓	Cluster of Nodes	Services, State Machines, Code Additions	Bosswave	Models Activity State

Table 4.1: Comparison of BOSs and activity modeling.

to Building Depot 2.0, the systems are built for a series of nodes that can be vertically and horizontally scaled, as well as be distributed over a larger geographical area. Instead of requiring code additions into existing code and recompilation, the BOS is built to be extendable by using services, and letting applications and services communicate over the Bosswave syndication bus. XBOS and Bosswave do not support the concept of activities and their modeling. Instead, this kind of responsibility is to be implemented by the applications themselves if needed.

Context-aware and activity-based systems CARL and JCAF[76] differs from Building Depot 2.0 and XBOS as they are not actual BOSs. Therefore, the support for the building infrastructure is not considered and is therefore not supported by these systems. Scalability-wise, both are single node systems and are therefore not suited for horizontal scalability. JCAF is an abbreviation for Java Context Awareness Framework, and is intended to be used by other applications that implement context awareness. Just as the proposed ATS, it models the current state of an activity. In contrast to JCAF and ATS, CARL does not label the current state of an activity, but instead only labels certain input as being an activity. The aims of CARL is different as it aims to change the behavior of occupants, where ATS is aimed at providing services for application on top, as well as to enable more loads to operate efficiently and participate in demand response events. Also, CARL is built to infer the current activities of the occupants based on sensor data, but currently it is not acceptable for the retail store stakeholders to guess the activities current state, as a potential DR event could have significant consequences for sales or wares if an activity is not tracked correctly.

The ATS that we propose in this chapter, in contrast, seeks to extend the XBOS/Bosswave combination with an additional service to let the BOS model the activities and their states. The service is intended to be used in a retail setting, which is also where the need for such a system arose. However, it might be relevant for other building types as well. As the ATS is built on top of XBOS and Bosswave, it shares some of the same characteristics such as the support for the building structure, scalability, as well as the Bosswave syndication bus. Where it differs is of cause the main functionality of ATS: support for activities, but also how to extend the functionality of the services. The ATS is extendable in several layers. First, it supports the same kind of service-oriented extendability as its parent system, XBOS, and Bosswave, but

also supports dynamic extensions of state machines at runtime, that is dependent on other state machines, hardware, or sensors. Additionally, code additions can be made if the desired transition types do not exist in the system. The code additions would require a recompilation of the service.

4.3 CONTRIBUTIONS

4.3.1 MAIN CONTRIBUTIONS IN PAPER [2]

4.3.1.1 CONTRIBUTION 1

The first contribution is the design and implementation of the ATS, used for activity tracking in BOSs. To keep this summary short and concise, several parts of the system design were not included, such as the system design principles, but these can be found in the original paper in Chapter 10 on page 137 if they are of interest to the reader.

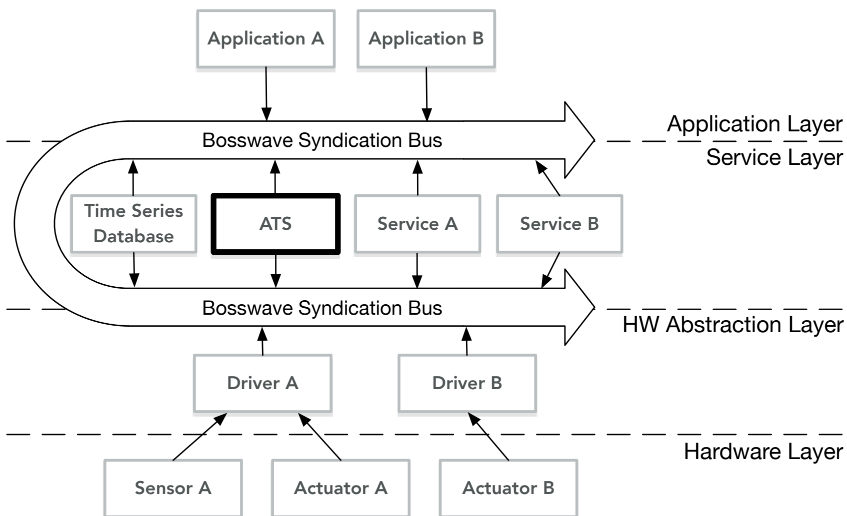

Figure 4.2: ATS Placement in a BOS Context.

Figure 4.2 shows where the ATS is placed in relation to the other components in the BOS. At the bottom sensors and actuators are adapted to Bosswave using a driver. In the service layer several services are placed, a time series database, ATS itself, and other services. These services communicate their respective information or processed data

back into Bosswave. From from this point on applications take over and choose how to act on the information provided by the services, or possibly even the drivers.

Bosswave [47] was chosen to solve the security and privacy considerations, because of how it encrypts all communication. Readers are referred to page 18 for more information on the encryption scheme and how Bosswave works. The Uniform Resource Identifiers (URIs) of Bosswave also solves part of the interoperability considerations, as it is exposing a common way to interact with all information in the BOS. To unify this further, ATS uses JavaScript Object Notation (JSON) to communicate information between services and applications.

An activity is defined as the set of actions and stages that can describe what a person or other entity do to reach a goal. The activities that we would like to model typically involve appliances consuming energy. Examples could be a person in a store filling up a refrigerator with goods, or cleaning personal cleaning the floor of a particular area. To model the activities, an adapted version of the commonly known state machine design pattern was chosen. The reason for this choice was a wish for developers to easily recognize the patterns used in development.

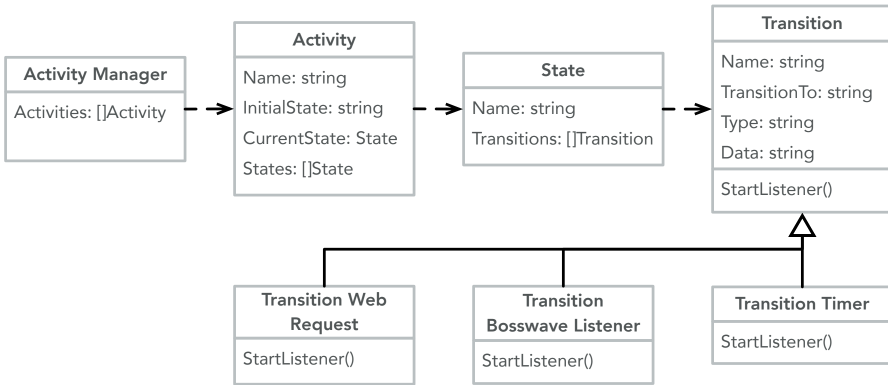

Figure 4.3: Simplified ATS architecture describing the modified State Machine implementation.

A simplified version of the central parts of the ATS can be found in Figure 4.3. The *Activity Manager* has the responsibility to orchestrate the activities. The activities themselves are described using JSON too, and sent to the *Activity Manager* that creates and starts the *Activity*.

Activity is the signifier for a single activity. It holds the current state, as well as all possible states. *State* works similarly as *Activity*, but holds an array of transitions to other states. *Transition* is the part that makes ATS extendable. Its subclasses overrides the *StartListener*() method, and implements their specific functionality. *StartListener*() creates a goroutine (similar to a thread), that waits for the implementation to react to something, and then sends the request to change state into a channel that takes care of the actual state change. The *Data* field in *Transition* is where subclasses are able to draw their specific configurations from. This configuration is stored as JSON and should define the functionality of a given transition. The "Transition Bosswave Lister" for example would need at least a URI to listen to, a key to use for access, and the expected message to listen for. The main difference from a standard state machine is the fact that the machine is dynamically defined using JSON, and extended using custom transitions. It is important to note that an activity state can be built to depend on the state of another activity. In this way, more intricate and interesting logics can be created.

4.3.1.2 CONTRIBUTION 2

The second contribution is the evaluation of the ATS. The following setup, as seen in Figure 4.4 and two activities were created in the ATS. The purpose is to create an example of the possible functionality as simple as possible that still illustrates the complexity of the problem. The example takes a small retail shop as the setting. The first activity lets a sale person in the shop register if they are present or leaving, while the second activity lets cleaning personal register if they are currently cleaning the room or if they finished up. A light regulator application then sets the lighting to 60% if the room is occupied, and 0% if the occupant is not there. If the room is currently being cleaned, the light will automatically increase to 100%, overriding the occupants setting, to increase visibility.

For the hardware layer, Amazon Internet of Things (IoT) Buttons were used, as well as an Amazon Echo. One click on the IoT Button would register cleaning as starting, and two consecutive clicks would send a cleaning finished signal. The Amazon Echo, on the other hand, was configured to react to both requests for cleaning state changes, as well as occupancy state changes. Additionally, a user would verbally

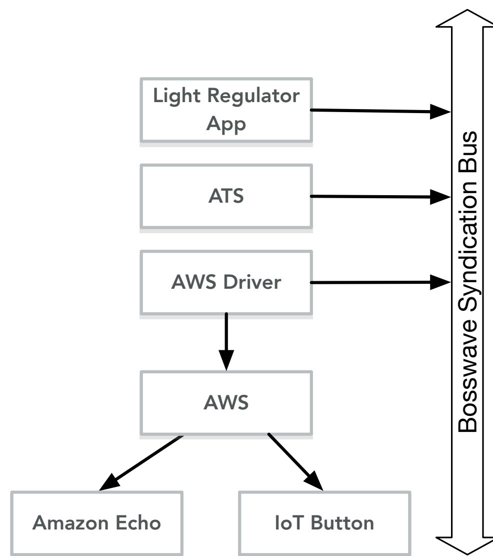

Figure 4.4: Evaluation Setup.

be able to request the current states of the two activities. Both the IoT Buttons and the Amazon Echo device runs functions placed in the Amazon Web Services (AWS) Cloud. These functions relay the requests to the AWS Driver that in turn restructures the request into JSON and publishes the state change request into Bosswave. The ATS then picks up the message, determines if the change is valid, and notifies subscribing applications through Bosswave.

The prototype was tested in an office setting over a week. Both the Amazon echo and the IoT buttons were tested by multiple people in the given time frame. Each person had a short verbal introduction to using the hardware. Testers where told the Amazon Echo could change states of both activities, and given phrases to use to change these activities, as well as to get the current state of the activities. The button's single and double click functionality were demonstrated too.

The hardware chosen for the evaluation had several problems that are not directly relevant for the ATS, and they are therefore not described here. Please refer to Chapter 10 if they are of interest.

Figure 4.5 details the outcome of one of the testing occurrences. As can be seen, in this case, the system behaved as expected. If the Cleaning activity changes to *active*, the light level rises to 100%, and if

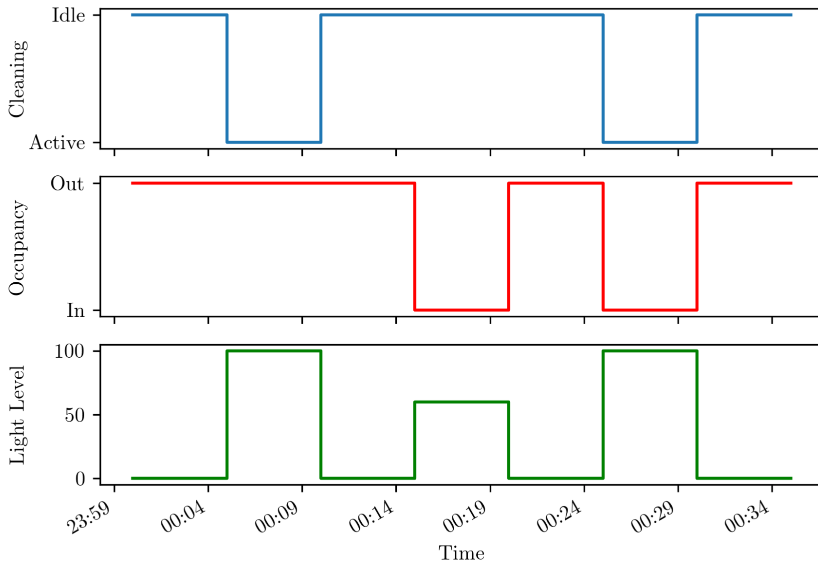

Figure 4.5: Testing Results.

the Cleaning activity is *idle*, and the Occupancy activity is *In*, the light level is set to 60%. Finally, if the activities are set to *in* and *active*, the cleaning activity supersedes occupancy activity and sets the lighting to 100%.

During testing of the ATS, a significant flaw became apparent. If one activity for some reason crashed, the entires ATS crashed. Activities code should be separate from each other, and should also implement a solution to recover the current state of an activity if it crashed. Architecture wise an activity should be considered to be handled more like a separate service that can be stopped, started, restarted and recovered.

Complexity-wise, the ATS is relatively simple and easy to understand. The integral parts are made up of less than 700 lines of code, including three different implementations of a transition. The three transitions are on average 90 lines of code, where on average 25 lines of code is the part a developer would need to implement for a new type of transition. The few lines of code in custom transitions ensure the ATS is easy to extend functionality wise.

The choice of using state machines as the structure of the ATS was found to potentially come with certain drawbacks. A complex state

machine could potentially get lost in a state if the transition failed to notice a change. If this happens, a state machine could get stuck and only move out of this state again once the specific state would be hit again. One way of avoiding this could be introducing a failsafe mechanism developers can implement. This failsafe would allow the activity to reset to a specific state if a particular set of conditions are met. In certain use cases, it could also be beneficial to allow for more flexible state machines that introduce the possibility of estimating what state is currently the most likely to be active out of a given set. Currently, this kind of flexibility is not possible in the implemented rigid understanding of what a state is. An added benefit of this could be allowing an activity to be in several states at once if this use case is needed. In the case of energy analytics involving states, a rigid state machine could also prove unpractical.

4.4 CONCLUSIONS

In this chapter, the ATS was presented and designed for a BOS setting. In conclusion, the designed and implemented ATS performed mostly as expected helped give additional insight into how activities could be modeled, to help leverage hardware entangled in activities for energy savings and DR purposes.

That being said, this exploration into creating a service for a BOS also made it obvious several problems in regards to service portability existed. Simply put, how would the new knowledge being generated by the ATS be discovered by other services or applications? This lead to the Service Abstraction Layer (SAL), which is discussed in Chapter 5.

CHAPTER

THE SERVICE ABSTRACTION LAYER 5

This chapter presents contributions of two semantic service discovery implementations, that enables service portability and discovery between buildings, as well as providing improvements to High Availability (HA) and resilience initiatives in Building Operating Systems (BOSs). The first approach is the topic in Chapter 11 and was published in the paper [3] (Service Abstraction Layer for Building Operating Systems: Enabling portable applications and improving system resilience), while the second implementation is the topic in Chapter 12 and was published in paper [4] (Service Portability and Discovery in Building Operating Systems Using Semantic Modeling). The second implementation has improvements added in Paper [6] (OPM: an Ontology Based Package Manager for Building Operating Systems), that is also examined further in Chapter 14. The extending paper contributions are included, but not motivated, as these are discussed separately in Chapter 6, and the contributions to the Service Abstraction Layer (SAL) are not its primary purpose. The first approach to the SAL presents an ontology that is sparse in its formal description, and expects the creator, and the user of the model, to adhere to more conventions and expectations set when creating the model. For example, that occupancy entries has a specific data model that is by convention. The second approach moved in a different direction, and describes the interface, and the data model too, thereby not requiring prior knowledge of this model. Section 5.1 introduces and motivates the contributions. In Section 5.2 related work is discussed. Finally, Section 5.3 summarizes the main contributions of the papers as related to this chapters topic.

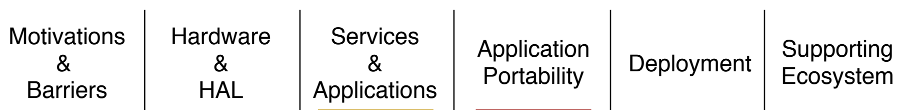

Figure 5.1: Overview of Areas examined in this Chapter.

5.1 INTRODUCTION

And discussed in before, BOSs introduce the concept of a Hardware Abstraction Layer (HAL), that effectively works as a layer on top of the hardware or Building Management Systems (BMSs). The presence of such a layer allows applications and services to be abstracted from the specific implementations of the hardware in the building.

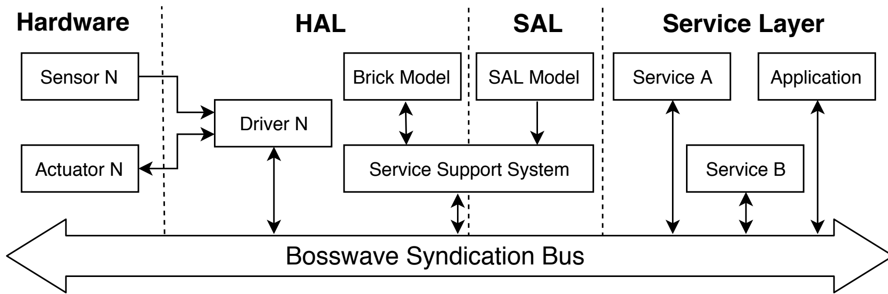

Figure 5.2: BOS Layers overview with SAL.

Figure 5.2 captures the basic concept of the HAL and SAL, that is introduced in this chapter, in the context of a BOS. The components enabling service discovery, effectively implementing the SAL, will henceforth be referred to as just the SAL as a whole. The architecture is based on microservices and includes a communication bus and small services that each solve a single well-specified problem area. Actuators and Sensors, generally regarded as the hardware components are abstracted using the HAL. In this case, the HAL is a combination of Brick, a Brick Model, the Service Support System (SSS) to host the model, and finally, the drivers that hide the specific implementation of the hardware.

In buildings it is important to understand the context, so location, direction, placement, etc., of the data being collected or processed. As covered in the Background section, Brick [74] is an example of a semantic approach to describing buildings, their layout, and how the hardware relates to each other and the building. However, Brick does not describe software services within the building, and how they relate to each other, nor does BOS's HAL provide this. These software services provide a multitude of processed information like fault detection, prediction, occupant modeling, Demand Response (DR) and more [2, 78, 79, 20, 80]. The missing abstraction between services, or between

services and applications, makes it difficult to integrate a DR service on a larger scale between buildings, as service portability between buildings is dependent on specific implementations. Also, the need for re-implementation or reconfiguring services for each building makes the act of participating in inter-building coordination endeavors like DR infeasible from a cost-benefit perspective.

To help achieve large-scale deployment of applications and services, portability is critical. To enable portability, this chapter introduces two different ontology approaches for describing services in BOSs, and thereby enable further abstraction from the building implementation. The service description enables applications and services to interface with other services, without prior knowledge of location, or in the case of the second SAL implementation, also data structure.

5.2 RELATED WORK

In the BOS, Services, and Ontology space several contributions have been made. Table 5.1 compares the related technologies with the SAL approach. Three types of related work is compared:

1. Two BOS specific implementations are detailed: A combination of SMAP [46] and Metafier [63], and Brick [74].
2. Two different service description ontologies: OWL-S [71], and WSMO [72].
3. Two service interface description methods: WSDL [81], and gRPC Remote Procedure Calls (gRPC) [82].

The SAL is designed to function in the BOS space; therefore, exploring existing technologies in this space is relevant. First, the combination of SMAP [46] and Metafier [63] is discussed. Metafier adds metadata to streams of information, thereby allowing the creation of a logical hierarchical structure, but does not explicitly address discovery needs of services and their context. Metafier seeks to add metadata related to hardware devices in a building, by having an expert tell Metafier where resources are located, and what kind of properties the streams have. These annotations are associated with single streams of information, not distinguishing between hardware or services. The metadata allows Metafier to build up a logical hierarchical structure, that signifies the composition of a building, and the relation that sensors and actuators

	SAL	SMAP + Metafier	Brick	OWL-S	WSMO	WSDL	gRPC
Type	Ontology	Stream Metadata	Ontology	Ontology	Ontology	Interface Desc.	Interface Desc.
Service Descriptions	Yes	No	No	Yes	Yes	Yes	Yes
Physical Context Description	Yes	Limited	Yes	No	No	No	No
Queryable	Yes	Limited	Yes	Yes	Yes	No	No
Modality Description	Yes	Limited	Limited	No	No	No	No
Unit Description	Yes	Limited	Limited	Primitives	Primitives	Primitives	Primitives
Temporal Aspect Description	Yes	Yes	No	No	No	No	No
Organizational Description	Yes	No	No	No	Yes	No	No
Multipath Options	Yes	No	No	No	No	No	No
Multipath Priority Options	Yes	No	No	No	No	No	No

Table 5.1: Related Work Comparison.

have to each other in the spatial dimension. Metafier has a limited ability to express the physical context, but only by non-related strings that, for example, can be used to distinguish between the buildings or rooms to which the stream belongs. However, these descriptions do not allow for any description of how one room relates to a building, or if a room is adjacent to another. The SMAP protocol allows for querying for streams and can take into account the metadata created by Metafier. However, these queries are limited by the limited expression of Metafier itself. Also, SMAP is not created with elaborate ontology descriptions in mind and does therefore not support these types of abstractions. Because SMAP is a protocol for retrieval of time series data, it has a precise and detailed query for the temporal aspect of the data. However, this temporal aspect is in the form of a query for data, and not a description of what can be expected of the endpoint. This is due to SMAP also acting as a HAL, and that all requests for data are expected to go through SMAP. Also, the temporal query parameters are describing the time series data retrieved, and not the context of a single property within. Brick [74] is an ontology, that also seeks to enable service and application portability and uses ontologies to describe the hardware and how it relates to locations, and other hardware. The ontology approach has proven itself to be significantly more descriptive than the Metafier approach, capturing significant aspects of a building and how it relates to other components. Also, the SPARQL Protocol and RDF Query Language (SPARQL) language is versatile, allowing for complex questions about the structure of the building to be answered. Brick is specifically designed to describe the physical context of sensors and actuators, as well as how the hardware and rooms inside of a building relate to each other. It therefore not only describes the physical context exceptionally well but is also used as the description of this in the SAL. Brick, however, does not concern itself with services, interfaces, or high availability. Both Brick and Metafier does allow for Modality and Unit descriptions but is currently limited to the types existing in the hardware space, neglecting the more complex types found in the services space. Neither Metafier nor Brick, models services, but only hardware (by design), and therefore fails to capture the output of services, their properties, and context.

Moving on to the ontologies for service descriptions, OWL-S [71] and WSMO [72], they both describe similar perspectives on services. Both are more concerned with control logic, and interfaces, but does

not describe the context of the dataset retrieved from modality, unit, temporal aspects, or physical context. While they do have some unit descriptions, these are limited to more primitive data types like integers, doubles, or strings. WSMO does describe organizational relationships of the service in the form of who created the service, and who owns it. Also, while WSMO does not support multipath HA descriptions, it does have several parameters like Robustness, and Scalability, though these describe more about the properties of the service than actually providing multiple paths. The case could be argued that either of these ontologies could be extended to support the contextual descriptions the SAL sets forth to solve. However, both ontologies are complex and do not describe anything about the content itself and how to read it, but more the interfaces to the services. Also, the ontologies complexity adds overhead with concepts like atomic processes and more, while the SAL sets out to present the core relations needed to solve the issue at hand.

WSDL [81] and gRPC [82] describe the interface, input, and output of a resource. They describe the parameters an RPC endpoint needs to function and the returned values. These technologies describe how to interface with the service, but they do not describe the context of the services they expose, nor do they allow for querying for them, as Table 5.1 states.

None of the alternatives to the SAL presented in Table 5.1 describes the properties needed to support services in a BOS context.

Paper [4] builds on the experiences obtained in Paper [3]. The paper [3] proposes a SAL based on an ontology but unfortunately has several shortcomings. **1**) The implementation of the ontology relies on a simple inheritance strategy of a service endpoint, which is restricted by its limited expressiveness. Specifically, it does not deliver the nuance needed to locate endpoints with specific types of information successfully. **2**) As previous work points out, a service is not able to successfully find the specific information it needs from a transmitted object. This forces the service using the service endpoint to know the specific implementation of the providing service, or for there to exist a detailed specification for how these service endpoints interfaces, based on the service endpoint's inherited class. **3**) previous work also support failover functionality but does not allow for a mechanism to prefer one over another, apart from just choosing the first that was entered into the model. This is a significant shortcoming for a failover implementation.

Paper [4] builds on Paper [3], addresses the above shortcomings, and add upon the functionality.

5.3 CONTRIBUTIONS

5.3.1 MAIN CONTRIBUTIONS IN PAPER [3]

5.3.1.1 CONTRIBUTION 1

The first contribution is the design and implementation of the first version of the SAL, which is used for enabling portable services for BOSs. From the introduction, Figure 5.2 shows how a BOS layered architecture would be constructed but includes the SAL, as well as the SAL Model. The term SAL refers to the layer itself and its functionality, where the SAL Ontology is a part of the implementation, and is the counterpart to Brick, but for services. Also, the SAL Model is the specific implementation of the ontology describing the relation between entities. Again, Brick seeks to abstract hardware from the application, while the SAL model seeks to abstract services from the application. The application will query Brick every time it needs to access hardware and will be provided with Uniform Resource Identifiers (URIs) to the data streams. Similarly, the application would need to query the SAL model if it needs access to services, which will also result in URIs to the specific service endpoints, allowing the application to choose which endpoint to contact.

A concrete example of the usage of a SAL would be the following scenario: An application that plans the assignment of rooms for lectures needs to plan the best distribution of classes. First the application needs to know the lecture rooms that exist in the building. This is achieved by querying the HAL, where a list is returned with all relevant rooms. Instead of reading directly from the hardware and trying to interpret the estimated usage for each room itself, it queries the SAL Model for a service providing occupancy counts for the specific rooms instead. The SAL returns a list of services providing this service, and the application contacts the first service on the list for the information needed. This allows the application to write code that is not specifically written for the specific building, and allows the application to be re-used on top of multiple buildings with other layouts and other services.

Introducing the SAL allows for decoupling and flexibility in the

following scenarios:

1 The microservice Weather Forecast is changed to a service from another provider. The service interface might be different, located on another server, and publish data to a different part of the Bosswave Syndication Bus, but the DR Application keeps working normally as the SAL Model is updated, and SPARQL queries to the model refer to the new location.
2 Removing a service that is depended on by an application, and no substitute is found, will result in an error message telling the technician which services types are missing.
3 Adding a service that delivers information that is already provided by another service, would allow the application to choose which one has the best fitting parameters. This change could allow the application to automatically switch between services, or use the better data source for a specific problem.
4 Building A has one set of services, and building B has another. With the SAL Model, the application could be moved between the two without code changes, as long as the building provides the service types on which the application depends.
5 A building is expanded upon and ends up with two different sets of services for the old section of the building and the new. The application seamlessly interacts with both zones and their set of services using the SAL Model, without code changes.
6 A building with several companies co-located could end up with the need for several extra applications or different services per. company depending on the needs. Again, the application allows for this using the same codebase as the SAL model abstracts from every given service.

The SAL layer consists of a SAL Ontology and a SAL model. The ontology itself describes classes and their relationship with each other, while the model describes the specific services in a building implementation. In classic object oriented programming terminology, the ontology would be the class, and the model would be the instance of that class. The ontology itself is built from five class, and a single class defined in Brick. Figure 5.3 details the relationship between the classes. The first class is named *Service*, which is a representation of the actual service providing one to many *Service Points*. A *Service Point* represents the URI or actual endpoint in which to fetch or send data

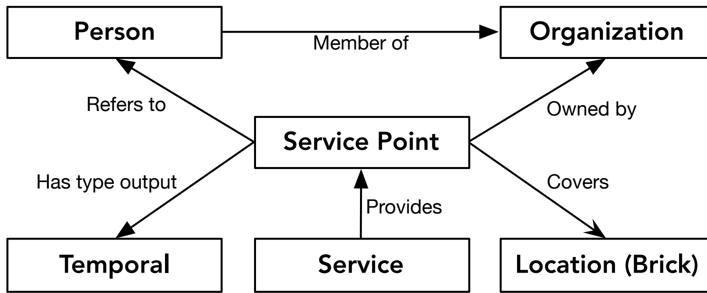

Figure 5.3: Service Abstraction Layer Class Relationships.

to interact with the service. This *Service Point* can be associated with the concept of a *Location*, and thereby enables the service endpoint to be associated to a specific location in a building. *Location* is a concept directly imported from Brick, and introduces several subclasses such as *Room*, what will be used in the examples later. The reason Brick's concept of *Location* and *Room* is reused, is to ensure concepts are not duplicated between several descriptions. Additionally, using the same concept makes the developer aware of how it is constructed, as well as allowing for eventual easier integration into Brick if such is desired in the future. When working with services, the output of a service is often related to predictions or other temporal parameters. Therefore, the *Service Point* has an association to the *Temporal* class. Occupancy prediction services such as OccuRE [80], the context of which types of people that are tracked can be significant. To accommodate this, the concept of *Person* is introduced and associated to the *Service Point*. Last, but not least, the concept of ownership and the owning organizations are introduced. *Organization* has two relationships, one for ownership that is associated with the *Service Point*, and another associated with *Person* signifying the membership of an organization.

Each of the classes mentioned before can have subclasses associated with them. A subclass indicates a more specific type of the superclass. Figure 5.4 details the subclasses identified to be essential for the examples, but is not complete or exhaustive. *Service* is not present as no subclasses where currently identified, and *Location* is part of Brick. *Location* is a superclass, with several subclasses, such as Building, Room, Floor, and more. Refer to the Brick papers [73, 74] for more detail on the *Location* class. The *Temporal* class has several subclasses that are

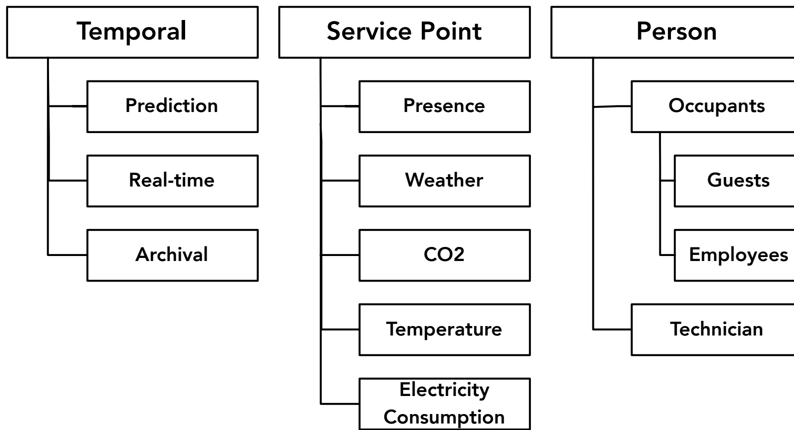

Figure 5.4: Service Abstraction Layer Class Inheritance.

supposed to specify what temporal phase the *Service Point* is describing. Current subclasses are Prediction, Real-time, and Archival. The *Service Point* is one of the hardest classes to create a complete ontology for, and one of the most important for successful large-scale implementation, as the subclasses, specify the actual context of the exposed data. The following subclasses were defined: Presence, Weather, CO_2, Temperature, and Electricity Consumption. All the possible subclasses will effectively be impossible to exhaustively add up front, as all possible service applications are not known, but fortunately, the success and usefulness of the SAL layer do not hinge on the subclasses including every imaginable scenario up front. Finally, the *Person* class has two subclasses. The *Occupants* subclass signifies occupants are modeled with the *Service Point*, but also adds the further specialization of Guest and Employees. The Technician subclass is not used in any of the examples here but is mainly there to represent further relevant additions.

The first example of applying the SAL Ontology into an actual model is detailed in Figure 5.5. Here the partial SAL Model describes the OccuRE service, which is a service built to track and predict occupancy based on historical data. One of the many *Service Points* exposed by OccuRE is exposed by the *Service Point* subclass; Presence. The Presence subclass refers to a location of which the prediction is made which is a *Location* subclass named Room, as well as specifying that the output is a prediction, by using the relation to the *Temporal* subclass Prediction. The service produces output that tracks Occupants, and all data produced

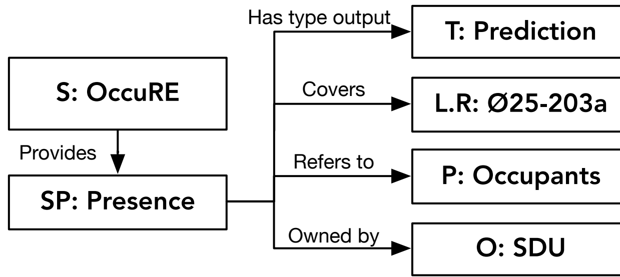

Figure 5.5: Service Abstraction Layer Example: OccuRE.

is owned by SDU (The University of Southern Denmark).

5.3.1.2 CONTRIBUTION 2

The second contribution is a discussion and evaluation of the first version of the SAL. To verify the architecture and SAL concepts, a prototype system is implemented, using the case system described by Nelleman et al. [20]. The case describes a state of the art DR system built on a BOS, that integrates DR decisions with model predictive control, weather data, and occupancy prediction. The goal of the case system was to use model predictive control to deliver comfort compliance, so make sure the building keeps a certain temperature standard, while also integrating with a DR service. The system architecture, was moved from the case's BOS to an eXtensible Building Operating System (XBOS) and Bosswave based system. In the instance of this prototype, the SAL is implemented as a microservice and is separate from the Brick Model. Services not needed are removed from the prototype setup, and some micro-services are shell services serving sample data from the original setup. The external mediator application and the DR Application are connected to an actual system but are a sample application built to simulate the same setup as the original. An The simulation setup will run through the following scenarios for verification:

1. **Application Portability:** Move the application between two buildings with two different sets of services for the same purposes. **Expected result:** No code changes needed.
2. **System Resilience:** Two services expose the same functionality. Remove the current service being used. **Expected result:** Application keeps running.

The architecture in Figure 5.6 is an adaptation of the system discussed by Nelleman et al. [20]. The system in the figure is based on XBOS [28, 27], and Bosswave [47]. The first layer from the left HW (Hardware), contains any number of sensors or actuators. These sensors and actuators are then homogenized by the use of any number of drivers located in the HAL and then published in the Bosswave Syndication Bus. The Brick Model, also located in the HAL, is functionally a semantic representation of the hardware, which allows the service or application to query for the URIs they need to access the data provided by the drivers in the Bosswave Syndication Bus. The combination of the drivers and the Brick Model is what enables the rest of the architecture to work on different buildings without much adaptation. The service layer contains several microservices that read data from the HAL or external services and then publishes them into the Bosswave Syndication Bus. The SAL contains the main counterpart of the Brick Model for the services, which enables applications, and other services, to discover what service types are provided, and if they deliver the kind of information needed for the application to function as intended. Further up the layers, the application layer can be found. This layer is the final part of the BOS and contains the DR application that communicates with the mediator that in turn communicates with the Distribution System Operator (DSO) using the OpenADR [40] protocol.

Working in the context of XBOS, Building Operating System Services (BOSS), and Bosswave, the architecture has the benefits of increased security by encryption, and significantly lowering coupling between all parts of the system. This lower coupling is achieved as drivers, services and applications all integrate with the Bosswave Syndication Bus, and the Brick and SAL Model, and not the specific implementations of each of these types.

The expectation for verifying the architecture that the BOS could be made more resilient by adding several services that supported the same type of output the application on top would need. The tests done in the system above supports worked as intended. The other expectation was that an application could be moved between two separate buildings or SAL Models, and still function as intended. This application move was verified to function as intended. However, the following problem was found during verification of the architecture. If the service is newly introduced to the system, and therefore the SAL Model, the application would need to be restarted for it to query the SAL again. This restart is

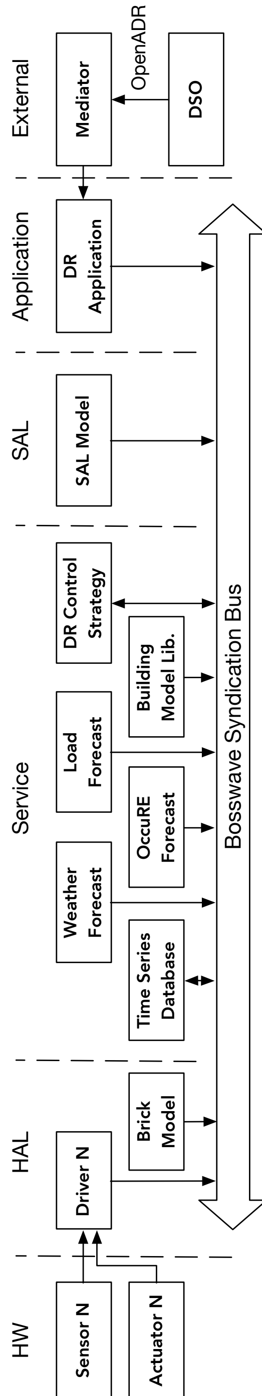

Figure 5.6: Architecture moved to XBOS and Bosswave.

something that can be alleviated by forcing a new query to the SAL if a timeout occurs, or by doing it using time intervals. If both services are present in the SAL Model during the boot process, this will not be needed.

5.3.2 MAIN CONTRIBUTIONS IN PAPER [4]

5.3.2.1 CONTRIBUTION 1

The first contribution is the design and implementation of the second version of the SAL, that is used to enable portable services for BOSs. The second version of the SAL is the product of the simple idea of having information discovery of data context in BOSs, as an application will not have prior knowledge of what services are available, or what interfaces they expose. When applying service discovery to a building, identifying the context of the information exposed is necessary. The first need is to know where the service is located, and what properties are present in the information you can gather from that location. The SAL seeks to expose and describe these properties, with modality, unit, and spatiotemporal aspects. Also, due to the recent initiatives such as EU General Data Protection Regulation (GDPR), an ever-growing interest in ownership of data, the SAL also models ownership of an endpoint's exposed data. This is especially useful in mixed environment buildings encompassing multiple companies or private occupants. Like Brick, the service interfaces are described using an ontology, resulting in a model, which can be queried by other services that have dependencies on information, instead of specific services. The SAL allows for this kind of decoupling from other services, because of the expressiveness of the Resource Description Framework (RDF) modeling method.

The SAL consists of several concepts and elements described below. The *SAL Ontology* is the description of SAL related concepts and their relationships to each other. It is based on RDF [62] and Web Ontology Language (OWL) [83], with references to the Brick Ontology and Schema.org Ontology [68]. The *SAL Instances*, or *SALI*, is a collection of instances of modalities, units, and more, ensuring only one instance of a particular concept exists. The *SAL Model* is the Turtle file containing the modeling specific to the service implementation of a building.

The SAL facilitates several functionalities and benefits. First, the primary purpose of the SAL is the provide a facility to support ser-

vice discovery, thereby allowing applications to integrate with services without prior knowledge of their specifics. The service discovery allows the services consumers, that can be both services or applications, to be portable between buildings without changes in implementation. Second, the change in architecture allows developers to change their services over time, for example, splitting services into multiple interfaces or merging them. This can be done without breaking compatibility with applications that depend on the exposed information, as long as it exposes the same types of information over the new interfaces, with the same context. The above change allows for service developers to move from mostly monolithic service structures to microservices over time or vice versa.

Another area where the SAL contributes is in the area of describing information ownership. This description is especially useful in the case of mixed environment buildings, with multiple companies inhabiting the building, but also if the BOS is serving a larger area with several buildings, or even on a city scale. This usefulness is derived from the SAL's support of ownership description of service endpoints. The SSS mentioned earlier, allows the SAL to be dynamically updated. This system allows services to be installed, query the HAL and SAL, configure itself, and publish its functionality into the SAL. Effectively, the SSS enables plug and play services. Due to the nature of how services are described in the SAL, several paths to similar information can be described. This type of over-provisioning of information, enables services to implement its own HA functionality or load balancing.

Using a microservice-oriented architecture, and the SAL introduces several complexities. Debugging a service architecture where dependencies are loosely defined can be difficult, and generally increases the focus of observability. This change will require developers to be aware of how they expose errors in the system and tell the technician why a service or application is not working. Failing to do so, or as a minimum give examples of what services an application needs, could make it difficult for a technician to diagnose problems. If error messaging is made correctly though, debugging should be relatively easy even with a service architecture this loosely coupled. From the perspective of the developer though, it is easier to detect what component is broken, and fix only that, compared to a monolithic application. This could increase correctness and robustness, as connections between program components are formalized.

This section describes the anatomy of the SAL Ontology, based on the requirements described earlier in this chapter. Figure 5.7 illustrates an overview of the parent classes in the SAL ontology. Subclasses are not present, as the diagram would become too large if it had to encompass all the subclasses. The implemented specific modalities and units are representative of the data streams collected across three buildings by the author's group. On top of this, several additional modalities and units were added, based on the needs of the services developed for the evaluation section.

Service describes a single service and has an object property called *provides*, which defines all ServiceEndpoints created by the service. The service class acts as a starting point for all of its endpoints and is a representation describing an actual service existing in the BOS. It has one data property, *name*, that helps give a meaning to a developer browsing the instance of the class. The *Service Endpoint* class is typically what applications are trying to locate when querying, as it encapsulates the URI for locating the desired information. It has four data properties. First, *Read* and *Write* properties define whether the endpoint expects parameters to function, or if it provides information, or both. The data property *Priority* defines the service's priority compared to other services that provide the same kind of information. This is an indicator for depending services of which order to contact a dependency provider in case of multiple options. Finally, the data property *Uri* defines the actual location where a service will be able to locate the information. This Uri can refer to the Bosswave Syndication Bus, REST or similar technologies.

The Service endpoint has two object properties, *OwnedBy* and *provides*. *OwnedBy* defines the organizational owner of the information provided through the endpoint, while *provides* links to the information parent class. *Information* represents a property inside of the information obtained from the URI residing in the Service Endpoint. One endpoint will typically have a multitude of *information* object properties, that point to separate instances of information. It is important to mention that this structure assumes that the returned object is a fixed structure, and does not change. One data property is present on *Information*, *location*, which is a descriptor of where this single piece of information can be found within the returned object that can be retrieved from the URI described in the ServiceEndpoint. The format of the data property is not formally defined as it depends on the architecture of a given

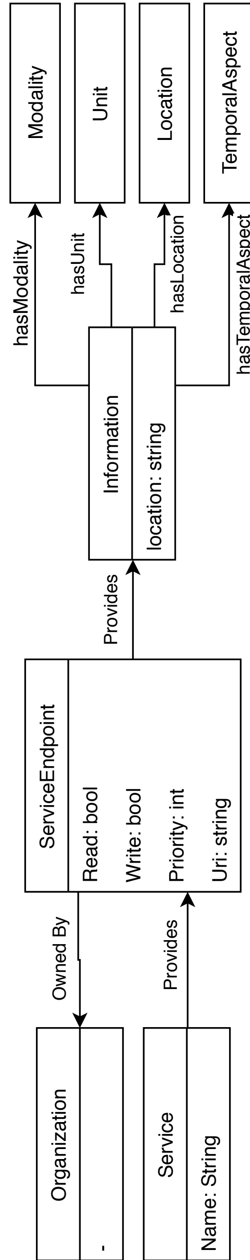

Figure 5.7: SAL Ontology Parent Classes - Relationships and Properties.

BOS. For example, if JavaScript Object Notation (JSON) is used, one possible value could be "[].MyProp" to describe that the property is found in MyProp in each object in the array received. Alternative implementations could be used here using gRPC or other alternatives that suit the specific implementation of the BOS. The information is described by its object properties, *hasModality*, *hasUnit*, *hasLocation*, and *hasTemporalAspect*. Each of these object properties describes a small but significant part of the context of that piece of information. This is an essential part of querying the SAL, as it enables the querying for the context of the information.

Modality	Unit	TemporalAspect
Angle, CO2, Presence, Flow, Illuminance, Power, Pressure, Rain, Humidity, Temperature, Wind, AbsoluteTime, RelativeTime, PowerFlexibility, Performance, Energy, Certainty, Time	Boolean, Count, CubicMeters, CubicMetersPerHour, DegreeCelsius, DegreeFahrenheit, Degrees, GigaByte, KiloByte, Hours, Hertz, Joules, JoulesPerCubicMeter, Kelvin, KiloJoulesPerSquareMeter, KiloMeters, KiloWatts, KiloWattHours, Lux, CubicMeters, CubicMetersPerHour, CubicMetersPerSecond, MilliAmperes, Minutes, MilliMeters, MilliSeconds, MetersPerSecond, MilliVolts, MilliWatts, MilliWattHours, Pascal, Percent, PartsPerMillion, RotationsPerMinute, Volts, Watts, Unitless, Time, DateTime, Date	Prediction RealTime Archival

Table 5.2: Subclasses for information annotation.

The *Modality*, *Unit* and *TemporalAspect* parent classes each consist of several subclasses, none of which have any object or data properties. The subclasses are shown in Table 5.2. These subclasses used in conjunction, add context to the information being described. An example could be the combination of the modality Wind, the Unit MetersPerSecond, and the temporal aspect Prediction. Each has little meaning by itself, but the combination provides an added insight. The lists of modalities and units are on what needed types where observed, but more will need to be added over time. To save the developer time,

each descriptive type have instances defined beforehand in the SAL Instances file. This file is not a requirement, but a convenience. Creating these instances for the developer beforehand ensures cleaner models, as well as only one type of each existing at any given time. All instances are subclass of Modality, Unit, and TemporalAspect.

Imported from Brick, *Location* adds a spatial dimension, describing the concept of building, room, zone, and more. As Brick is already describing the physical properties of a building and uses the same types of descriptive technologies, we consider it well suited for describing the physical context.

Returning to one of the *Organization* class, this class represents a company and its ownership of the information contained at the URI residing in the service endpoint. To not re-engineer a concept already thoroughly explored, the *Organization* parent class is imported from the schema.org [68] ontology. Schema.org has already contributed a significant amount of work modeling organizations, including a multitude of subclasses, as well as the possibility of describing divisions, owners, contact information and more. It is up to the BOS developer to choose the level of detail one wishes to have in these models. This parent class is particularly interesting in buildings with shared spaces between several companies, where a service might only service a single company, and not have permission to process data gathered from other sources. The class does not have any data properties in the figure, as everything is inherited from schema.org.

Figure 5.8 visualizes a limited sample of a SAL Model for a weather prediction service. The example is limited by showing one service endpoint, and one describing property. The service endpoint has a priority set, giving it a weight if more than one weather prediction service is present. The information modeled is owned by the Organization ”Acme”. The information found at ”acme.io/weather/1”, is described to be an array of objects, where each array has a property called ”TempC”, which is a temperature measured in degree Celsius. The value is a predictions about the weather around the ”Headquarters” building. Additional information could be described, which is indicated by the last information box.

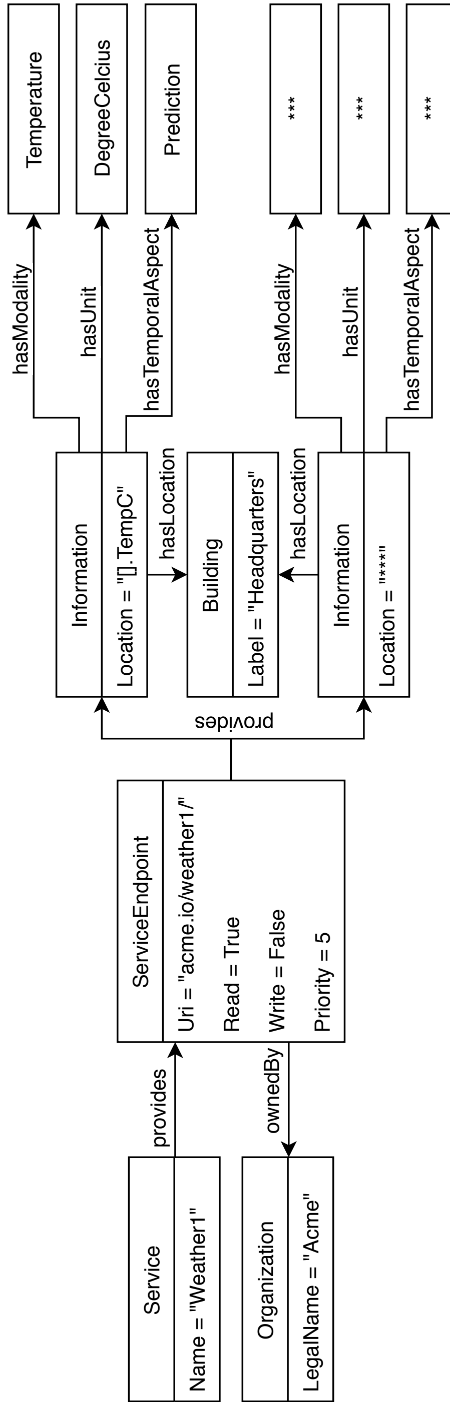

Figure 5.8: SAL Model; Weather Example.

5.3.2.2 CONTRIBUTION 2

The second contribution is a discussion and evaluation of the second version of the SAL. To evaluate the SAL and the components, an evaluation setup is built. The BOS selected and used for the verification implementation is based on XBOS [28] combined with Bosswave [47] and Brick [74]. The ontology is built using, Protégé using OWL and RDF, while the instances of the ontology and the models of the buildings are generated using Python and RDFLib. The SAL Ontology, Instances, and Model files are all saved in the Turtle [84] file format. The SSS [5] is used for hosting the SAL ontology, SAL Instances, and SAL Model, but also Brick and the Brick Model. All queries are in the SPARQL format.

The evaluation is performed on three different SAL Models inspired by actual buildings and their hardware. These represent different types of buildings, namely an office building, a retail store, and an educational building. The services are structured differently for each building, but are using the same implementation of the services, as the portability of these services are critical.

Figure 5.9 shows the dependency between services for each of the buildings. There are six different types of services in the setup, all providing predictions. The flexibility estimator uses information from the solar battery storage prediction service, and the ventilation usage prediction service, to estimate the available flexibility potential. This functionality is used for DR purposes. The energy benchmark service evaluates the performance of the building, and how it changes over time. The ventilation usage prediction service uses the presence prediction and weather service prediction to estimate the need for ventilation in a room. The presence prediction service is based on the OccuRE framework [80] for predicting occupancy. Note that while some rooms are measured as a Boolean indicating presence, others have better sensors allowing for prediction of occupancy counts. This nuance is included in the SAL model. The solar battery prediction is only present in the educational building as that is the only building to have solar panels and battery storage. The service predicts the given capacity of the battery storage at any given time. For the retail store SAL Model, the ventilation usage service, and the presence prediction services are contained in the same service. This is to evaluate if service merges have an impact on the queries needed to locate the information needed.

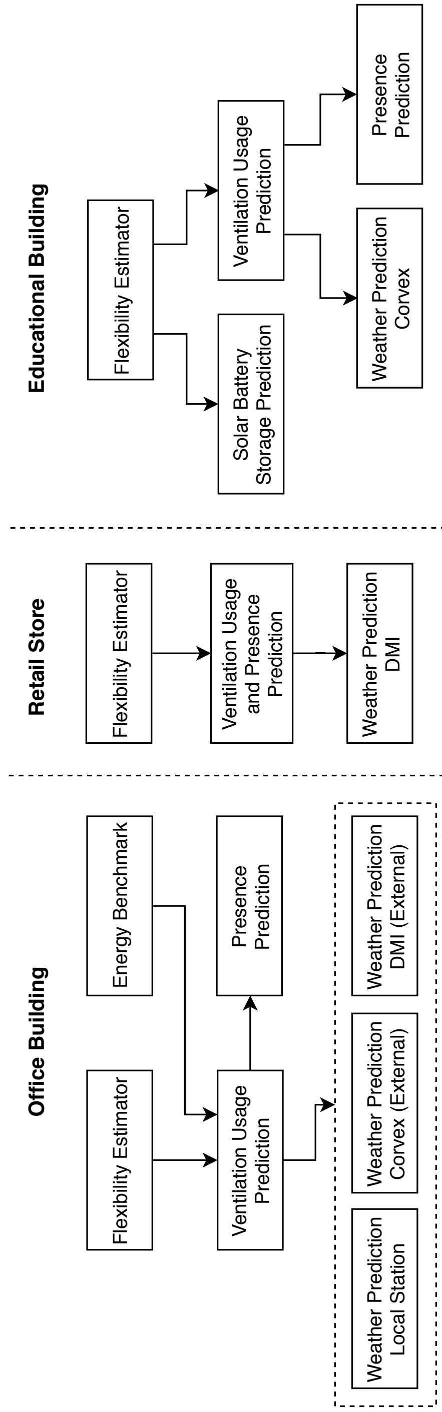

Figure 5.9: Service Dependencies For Each Case Building.

Each BOS service in this experiment is self-configured through queries on the relevant SAL model. The office building has an extra service depending on the ventilation usage prediction service and has several weather prediction services. Two of these are external to the building instrumentation, existing in the cloud, and one that is internal in the form of a service interpreting output from a local weather station. The existence of several weather prediction services allows testing of fail-over functionality. The retail store SAL Model is primarily chosen to allow testing the merging of service functionality, and the impact on the queries needed to find the information. The educational building SAL Model primarily acts as the ideal setup for the flexibility estimator service. So, when the service gets moved to the other SAL Models, it is missing this service and needs to adapt to the sub-optimal environment. Of course, all of the models contribute to testing the implemented services, and thereby queries, across different environments.

The portability of services is demonstrated by using the same queries on all the models for their respective services. As expected, even though the setups are different, the queries and service implementations adapt and work on each building without changes to the code. All data transmitted between services is synthetic data, as real-time data does not benefit the evaluation of the ontology and SAL layer. The infrastructure for high availability is tested using the office SAL Model, as it has multiple weather prediction services available. The model supplies the infrastructure for implementing failover. The ventilation usage prediction service is forced to adapt to changes in the infrastructure when a service is forced offline.

The expectations of the evaluation are as follows: **1**) *Service Portability:* Move the services between the three different SAL Models. *Expected result:* Services are working, and adapts to the changing models. No code changes needed. **2**) *Merging of Services:* Move to a SAL model where one service fulfills two service roles, instead of separate services. *Expected result:* Depend services keep running without code changes. **3**) *System Resilience:* Several services expose the same information type, and have depending services. The used service is then removed. *Expected result:* The depending service keeps functioning correctly.

Table 5.3 shows the results of the evaluation setup, with results split into the three evaluation areas. Section **A** refers to the portability evaluation, where the requirement for a ✓ is that the service ran on the

	Flexibility Est.	Energy Bench.	Vent. Usage Pred.	Presence Pred.	Weather Pred.	Solar Storage Pred.	Merge/Split Services	System Resilience
Office Building	✓	✓	✓	✓	✓	-	✓	✓
Retail Store	✓	-	-	-	✓	-	✓	-
Educational Building	✓	-	✓	✓	✓	✓	✓	-
	A						**B**	**C**

Table 5.3: Service Portability Evaluation Results.

SAL model for the specified building, and could successfully gather the information it needed from the services it depended on. Dashes refer to a situation where the service is not present in the model, meaning there is no evaluation. As the table shows, all portability tests ran successfully, as expected. Section **B** refers to the merging and splitting of service roles, and here a ✓means to a successful merge or split, without impact on services. The table shows all merge or splits of services worked as expected. Section **C** refers to the system resilience aspect of the evaluation, where a ✓means the failover test for the Office Building worked as expected, by arbitrating similar or competing services. Dashes mean that that type of evaluation was not available for that SAL model. All testing was performed at runtime. Based on the observations during the evaluation, the SAL should scale comfortably to several thousand services. This number depends significantly on the implementation of the SSS, and query complexity.

5.3.3 MAIN CONTRIBUTIONS IN PAPER [6]

5.3.3.1 CONTRIBUTION 1

The contribution of this chapter pertains to extending the modeling of the ServiceEndpoints functionality. To support the Ontology based Package Manager (OPM), and the case used in the evaluation (see Chapter 6), the SAL needed to adapted for the specific use case of the OPM, and was therefore extended. ServiceEndpoints are an abstraction for a location where a part of a service can be accessed. The SAL

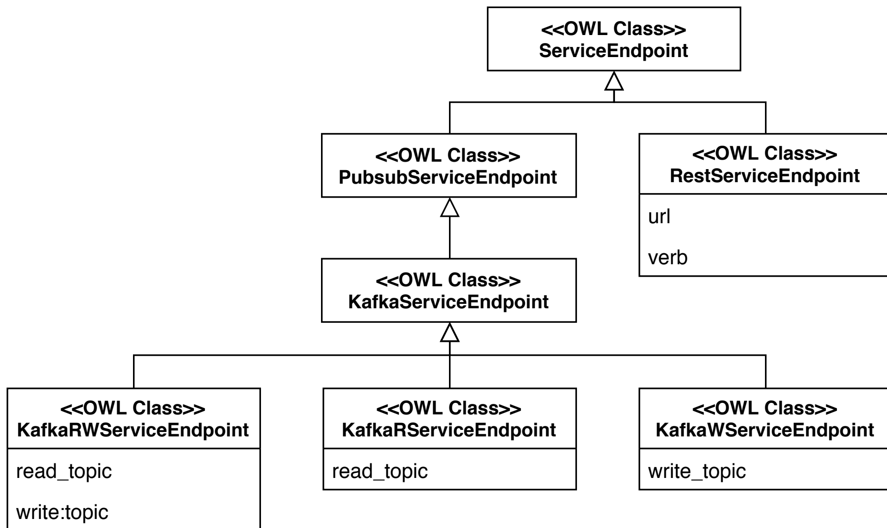

Figure 5.10: The SAL Extensions Made to Support Input Descriptions.

already supported ServiceEndpoints, but did not support the notion of read or write access, nor did it support the different variants of these. Figure 5.10 shows the changes made to the SAL, and how they classes inherit from each other. The service endpoint now has several subclasses that define what type of endpoint it is, and the specific data properties needed to interact with that type of service. This change effectively adds identification of an endpoint is intended for reading, writing, or both.

5.4 CONCLUSIONS

This chapter described an ontology to support service discovery, and enable portability of BOS applications and services, in an ecosystem with no prior knowledge between an application, and the services it uses. The evaluation shows the SAL gives tangible benefits to the goal of enabling portable services. Also, introducing the SAL gives the added benefits of applications being dependent on information, and not specific services, enabling a more loosely coupled and adaptive service landscape. The SAL also enables potential resilience benefits to the BOS as services provide information, and this information can be

provided redundantly. This redundancy enables services to implement failover functionality if services are sensitive to downtime. As the evaluation shows, failover functionality is successfully achieved by arbitrating similar or competing services.

The SAL enables large-scale deployment of services across different types of buildings that change over time, with the same codebase. Also, it allows developers to make standard products, instead of custom implementations for each customer, thereby reducing development costs per customer. For future work, the SAL should be evaluated by running on multiple buildings over a prolonged period of time, to validate it in a running, evolving, and expanding setting with multiple service developers using the ontology. However, the SAL paired with the HAL can have a significant impact on a BOS ecosystem and has the potential to change the return of investment calculations when deciding if a service is going to be profitable to develop.

CHAPTER 6

MITIGATING BOS DEPLOYMENT COSTS

This chapter discusses paper [5] (Enabling Auto-Configuring Building Services: The Road to Affordable Portable Applications for Smart Grid Integration) and paper [6] (OPM: an Ontology Based Package Manager for Building Operating Systems). A few contributions in these papers are detailed in Chapter 5, as they are related to the Service Abstraction Layer (SAL). The papers [5] and [6] covers initiatives to cope with the deployment costs of Building Operating Systems (BOSs). The paper [5] details the Service Support System (SSS) that addresses several BOS shortcomings in regards to automatic configuration of building services. The paper [6] details an ontology based building service deployment system called the Ontology based Package Manager (OPM), to help mitigate the costs of deploying BOSs. Section 6.1 introduces and motivates the contributions. In Section 6.2 related work is discussed. Finally, Section 6.3 summarizes the main contributions of the papers as related to this chapters topic.

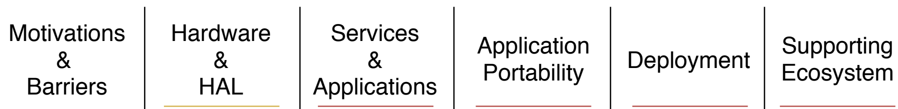

Figure 6.1: Overview of Areas examined in this Chapter.

6.1 INTRODUCTION

Previous work on retail stores on Demand Response (DR) [1], shows the cost of implementing DR functionality in a store would far exceed the cost benefits for the customer. The problem is that retail stores typically work with a relatively short Return on Investment (ROI) of 3 years, while many other building owners make this calculation over 30 years. During the same study, it became apparent that the incentive

for retail stores should be added functionality, that could save them money elsewhere, and cutting deployment costs. An example of added functionality could be the ability to control several retail stores from a central location, and give statistics, while also diagnosing if the building is performing to specifications. This kind of added value could mitigate a substantial amount of the deployment cost, as most buildings currently are left behind after the initial configuration. Once a BOS is deployed, the cost of implementing controls on top of that system that supported DR would be relatively low cost, which would allow for a more reasonable ROI.

Due to the current trend in energy efficiency, and the need for added functionality to reach energy goals, Building Management Systems (BMSs) are found lacking as they cannot adapt to these changing needs. A BOS, in contrast to BMSs, allows for external interfaces and allows applications to extend the functionality of a building. A future system is imagined by several researchers looking into BOS platforms, where buildings are platforms living in an ecosystem with several stakeholders developing and selling components for it. One example of a future where entrepreneurs do not just buy a BMS and deploy, but create functionality in the building for the customer. On top of this vision is another ecosystem of developers creating value for customers by creating applications that work in a building context on top of it. An example of this functionality could be DR.

This chapter concentrates on the cost perspective while leaving the functionality additions mentioned earlier for future work. To solve the issue of deployment costs, a system for deployment of BOSs would need to be developed. Three cost mitigation approaches were identified:

1. Portable applications.
2. Automatic configuration of applications in a BOS.
3. Automatic deployment of applications and dependencies.

Previous research has already addressed the first cost problem of portable applications from several perspectives from a Hardware Abstraction Layer (HAL) [28], metadata [63], ontology descriptions of the hardware structure of the building [73], to service abstraction [3, 4]. For example, in a BOS based on eXtensible Building Operating System (XBOS) [28], Brick can be applied to describe the building using an ontology description of the building and its hardware, only requiring

a query to locate the hardware needed, instead of hard-coding implementations for every specific building. Applications and services can also be abstracted in a similar manner using a SAL [3], only requiring a query, similarly to Brick. Together these technologies allow for building an application that does not need hard-coded references to hardware or software components, thereby enabling portable applications.

The second point of automatic configuration of applications has been mostly enabled by the previous point, portable applications, but lacks several components, specifically, removing the need for BOS maintainers to expand the Brick and SAL model manually. Creating Brick definitions automatically has been made possible, and been implemented by Koh et al. [85], by auto-generating the Brick model from the current BMS data. Removing the need for manual editing of the SAL is a different problem, as the ecosystem, it describes is based on the services needed for that specific building, and not how the building and its sensors and actuators are placed. To solve this problem, a service hosting the ontologies is needed, that can be dynamically updated by the services themselves, as to make sure they can automatically configure themselves, and announce the functionality the service or application provides to the rest of the system. The first paper addressed in this chapter addresses this dynamically updatable model enabled by a service and evaluates cost implications for the different stakeholders in the economic ecosystem, and different phases of the BOS life cycle.

To solve point 3, a package manager is designed that handles the deployment of drivers and services and moves the BOS to a containerized deployment strategy. Creating a package manager for a BOS is not a simple task, as a BOS has several specific obstacles associated with it. For example, several BOSs are distributed, and therefore do not exist on only one host, but several. Also, the requirements for a package to function can be specific hardware instrumentation or services, and these services might have been deployed several decades ago.

Generally, four tasks can potentially take time when deploying, or changing a BOS setup:

1. Time taken to validating the hardware supports the software to be installed on the system.
2. Time taken to learn the dependencies of a service, and how to install them.
3. Time taken to understand how the application works and how it

ties together with other services.
4. Time taken due to packages with broken dependencies as buildings instrumentation and management are deployed for decades without upgrades.

The OPM, addresses all of these four issues, while traditional approaches typically one solve one to two of these. The OPM contributes the following:

- Container-Based Dependency Resolution utilizing the SAL, effectively creating a package manager for container-based applications.
- Deployment of containers into the BOS Ecosystem.
- Allowing not only direct but also loosely defined dependencies based in on the SAL ontology.
- Verifies hardware requirements of the applications.
- Verifies no other pre-existing service satisfies a dependency.

Loose dependencies are an essential part of the OPM design, as it allows the available packages to change over the several decades building instrumentation can last, but still allow the packages to work, by finding alternative solutions for the informational requirements of the package being deployed. Also, validating the hardware requirements is important, as several packages might not be reliable candidates if specific hardware is not present, potentially resulting in a defective dependency graph, and thereby a non-functioning deployment.

The OPM is also evaluated from a cost perspective, by measuring the deployment time for the BOS before implemented changes, and after. Also, the perspective is given on the possible future shapes an OPM can have, and how they can impact the stakeholders of such a system.

6.2 RELATED WORK

This section begins with detailing the related work for paper [5].

Ontologies, like the Brick and the SAL, and the accompanying models, need to be hosted by a service that can handle the queries from the system, and return the metadata requested. HodDB [48] is one such service. It is built specifically for Bosswave and Brick and has very well documented performance scores. Unfortunately, it does currently not support the SAL implementation, that is specifically needed to create

completely auto-configurable services that can publish themselves into the SAL. Furthermore, HodDB currently does not support dynamically changing models. This limitation is specifically an issue, as a service has no way of registering itself in the SAL automatically if HodDB could host it.

In the package management space, multiple solutions exist. Some are designed for deploying binaries, while others are designed for deploying containers. The package managers covered, are the OPM itself, Spawnpoint [86], Docker (with Docker Compose) [87], Kubernetes [88], Spack [89], Snap [90], and traditional package managers that operate at host level such as RPM [91], and APT [92]. Table 6.1 details the features of these packages. There are two overall types of package managers present in the table.

1. **Orchestrators** that are designed to deploy containers into distributed systems. Indirectly, these are also package managers, as they orchestrate how the containers are supposed to work together and get the containers from repositories. These include Spawnpoint, Docker, and Kubernetes.
2. **Package Managers** that are designed to install specific binaries on a host. The last three; Spack, Snap and RPM/APT are examples of these.

Exploring the **Orchestrators**, all of these have strategies for deploying containers on multiple hosts and is built to manoeuver in a distributed systems architecture. Spawnpoint is a secure container deployment software designed to work with Bosswave [47]. It manages the hosts by locating hosts in the blockchain deployed by Bosswave and then sends install commands that include resource restrictions and other information. Destroying, restarting, and deploying containers, can all be done through Spawnpoint. Spawnpoint's intent is to be used in a BOS, just as the OPM. Docker is one of the most popular container orchestrators. It can run in swarm mode and function as a cluster of nodes and can mediate high availability functionality. Docker uses Docker Compose to define what containers should work together and what resources they may use. Kubernetes, to some degree, functions somewhat similar to Docker Swarm, as it functions as an orchestrator for containers, and provides container, resource, and high availability management. Kubernetes can function on top of Docker for container orchestration, but not at the same time as Docker Swarm is deployed.

Feature	OPM	Spawnpoint	Docker	Kubernetes	Spack	Snap	RPM / APT
- Maturity Level	Research	Research	Industry	Industry	Research	Industry	Industry
- Intended Purpose	BOS	BOS	Generalized	Generalized	Generalized	Generalized	Generalized
- Operational Space	Distributed	Distributed	Distributed	Distributed	Distributed	Local	Local
- Type of package managed	Container	Container	Container	Container	Binaries	Container	Binaries
- Multi-host deployment strategy	Yes	Yes	Yes	Yes	Yes	No	No
- Host Node Management Support	Indirect	Yes	Yes	Yes	Yes	No	No
- Resource Management	Indirect	Yes	Yes	Yes	Yes	No	No
- Dependency Resolution	Yes	No	No	No	Yes	No	Yes
- Loose Dependencies	Yes	No	No	No	No	No	No
- Context Specific Dependencies	Yes	No	No	No	No	No	No
- Validate Hardware Requirements	Yes	No	No	No	No	No	No
- Dependency Resolution Takes Currently Installed Packages Into Account	Yes	No	No	No	Yes	Does not apply	Yes
- Dependencies of the Package Manager	Kafka Docker	Bosswave Docker Linux	Linux, Mac OS, or Windows	Linux, Mac OS, or Windows	Linux	Linux	Linux

Table 6.1: Feature Comparison between Package Managers.

The OPM takes many of its features from the three systems above. However, OPM does not manage the notes on which it is installed as the other orchestrators but instead leaves a communication endpoint in the communication bus for each of the hosts available. So, centrally managing the OPM instances is possible, but it is not controlled directly by the OPM. The OPM builds on Docker to deploy its containers, and therefore also inherits the possibility to use Docker Swarm, and gain the high availability functionality for its containers and itself. Spawnpoint, Docker, and Kubernetes all support resource management of the containers. The OPM does not explicitly take care of this, but indirectly supports it as it uses Docker Compose to deploy containers. Traditionally containers do not have dependencies as they should contain them all, or the composition of services should solve this problem. However, dependency resolution still plays a critical role when multiple containers are available, interfaces are complicated, and the user has little to no prior knowledge of the other packages available. Also, as building deployments are in operation for decades, and available packages change over time, loose dependency resolution is required to allow packages to work decades after initial dependencies. The OPM supports direct dependencies, as well as loose dependencies to other containers. Another area where the other orchestrators fall short is validating the hardware requirements of the building instrumentation, but also exploring the physical context of the building. These are all supported by the OPM.

Moving on to the second category, **Package Managers**, Snap stands out as the only candidate that uses containers for deployment. It is used on Ubuntu Linux as a next-generation package manager, that is supposed to solve many of the problems with traditional package managers such as RPM / APT. To do this, Snap adds all of its dependencies inside of the container, making sure it does not need dependency resolution functionality. This works fine on a single host, as packages are not supposed to work together in a distributed system. Snap is different from the orchestrators, as it only deploys containers on one host, and therefore lacks any multi-host functionality. In that regard, it works more like the traditional Linux Package Managers like RPM / APT, as the intent is to install an application on a single host. Spack is a package manager, that seeks to bring the functionality of RPM / APT to high performance computing data centers, enabling deployment of binaries to hundreds of machines at once. It has all of the distributed

functionality as the orchestrators but does not work with containers, but binaries instead. As it works with binaries, just as RPM / APT, it uses dependency resolution to resolve the packages needed. None of the package managers has the ability to validate hardware requirements, have loose dependencies, or explore the context of the building. Spack and RPM / APT does however also check if requirements are already installed, just as the OPM uses the SAL installed in the BOS.

As Table 6.1 demonstrates, no current container orchestrators, nor packages managers, provides the feature set of the OPM. More specifically, no container based system performs dependency resolution between containers, loose or direct, nor do they validate the physical requirements of the containers.

6.3 CONTRIBUTIONS

This section details the contributions of each paper related to this chapters topic.

6.3.1 MAIN CONTRIBUTIONS IN PAPER [5]

6.3.1.1 CONTRIBUTION 1

The first contribution is the design and implementation of the Service Support System, for enabling auto-configurable building services.

Designing a system for exploring the automatic configuration of building services, requires an ecosystem to test and develop in. Figure 6.2 explores this setup, and details the new components in such a setup. Horizontally in the top, the hardware layer, HAL, SAL and Services layers are defined. The hardware layer contains the actual sensors and actuators that make up the instrumentation of the building. This hardware is integrated into the BOS using drivers. These drivers are adapting the hardware interfaces to a common interface that interacts on the Bosswave Syndication Bus. Bosswave has already been touched upon in section 2.4.1 on page 18. On the right of the diagram, the services layer, or application layer is present. This layer contains the three microservices of which two was mentioned earlier, an Archiver, OccuRE, and the DR Orchestrator. The Archiver is a microservice holding historical data from sensors, and potentially also services. In the middle, the system designed to help services auto-configure itself is

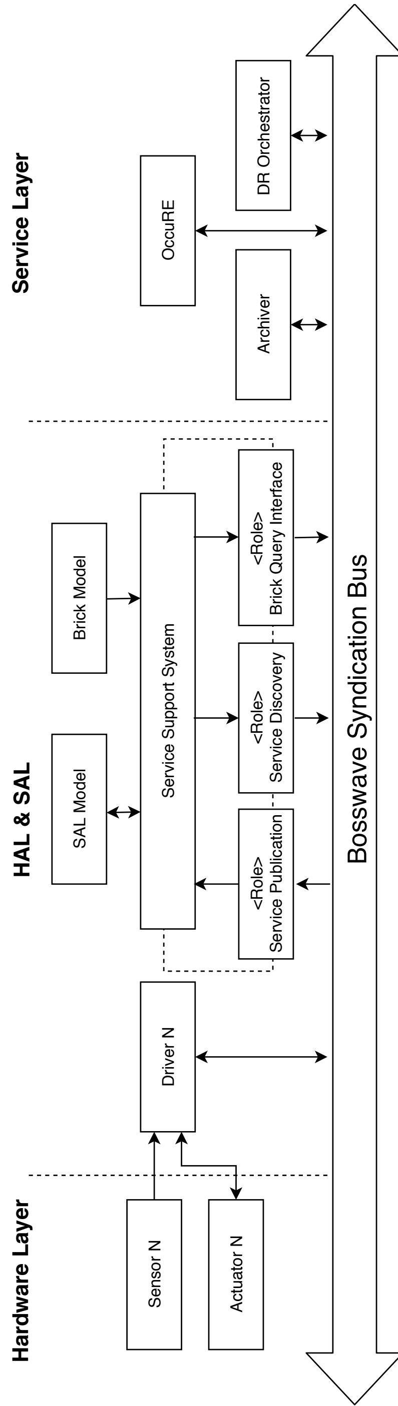

Figure 6.2: Microservice Ecosystem Example.

present. The system consists of several models, and roles. The roles are as follows, Service Publication, Service Discovery, and the Brick Query Interface. Service Publication enables services to publish their endpoints, and make themselves known to the rest of the BOS ecosystem. The Service Discovery Role enables finding services using a SPARQL Protocol and RDF Query Language (SPARQL) Query. Finally, the Brick Query Interface allows for querying for the hardware and location context of the building, also using SPARQL Queries. All of the roles are managed by the SSS, which also hosts the ontologies and models of the SAL and Brick. An ontology is the description of the possible building blocks a model can consist of, while the model is the concrete implementation and use of the ontology. To clarify, this means that the SAL ontology, for example, defines the Service and ServiceEndpoint types, but the SAL model contains the specific instances of Services and ServiceEndpoints.

For the SAL operations, a high-level approach is taken, where the client – who wants to publish a service – describes the service in a JavaScript Object Notation (JSON)-based format. This description spans all relevant parts of the SAL, namely (1) the service individual containing the service name, (2) the endpoint individual representing a single network endpoint, and (3) the information individual detailing which kinds of data are involved.

The JSON is marshaled using a natural nesting which allows on service individual to be associated with any number of endpoint individuals. These, in turn, may be associated with any number of information individuals. A Brick location entity is used for referencing the location of the information individual, and an organization name is used in the endpoint individual to reference a schema.org organization.

To make this work in a Bosswave context, we wrap the publication request in an object indicating a result path for where the result of the publication should be published. By subscribing to this result path, a client will receive a notification of the completion of the operation. Figure 6.3 shows a full example of how the modified version of OccuRE publishes its services.

Discovery is implemented using regular SPARQL queries over the underlying Resource Description Framework (RDF) store. Using a lower abstraction allows greater flexibility while expressing queries, but requires knowledge of the underlying RDF structure and the querying language itself.

```
{
  "result-path": "a20856b9-********",
  "service": {
    "name": "OccuRE",
    "hasServiceEndpoint": [
      {
        "url": "jah.demo/occure/a20937b9-********",
        "read": true,
        "write": false,
        "priority": "1",
        "ownedBy": {
          "legalName": "University of Southern Denmark"
        },
        "providesInformation": [
          {
            "location": "Count",
            "hasModality": "Occupancy",
            "hasUnit": "Count",
            "hasTemporalAspect": "Real-Time",
            "hasLocation": "model:rooms.e20-601b-2"
          }
        ]
      }
    ]
  }
}
```


Figure 6.3: Request for service publication.

```
SELECT ?org_name ?service_name ?sep_url ?loc ?modality ?unit
       ?ta ?bloc
WHERE {
    # main individuals
    ?service rdf:type/rdfs:subClassOf* sal:Service .
    ?sep     rdf:type/rdfs:subClassOf* sal:ServiceEndpoint .
    ?info    rdf:type/rdfs:subClassOf* sal:Information .

    # external individuals
    ?org     rdf:type/rdfs:subClassOf* schema:Organization .
    ?bloc    rdf:type/rdfs:subClassOf* brick:Location .

    # object properties
    ?service sal:hasServiceEndpoint  ?sep .
    ?sep     sal:providesInformation ?info .
    ?sep     sal:ownedBy             ?org .

    # data properties
    ?service sal:name               ?service_name .
    ?org     schema:legalName       ?org_name .
    ?sep     sal:url                ?sep_url .
    ?info    sal:location           ?loc .
    ?info    sal:hasModality        ?modality .
    ?info    sal:hasUnit            ?unit .
    ?info    sal:hasTemporalAspect ?ta .
    ?info    sal:hasLocation        ?bloc .
}
```

Figure 6.4: Query for listing all published services.

To express such a query, knowledge of how the main classes of the SAL are mapped to RDF is needed. The three main classes are Service, ServiceEndpoint and Information. Figure 6.4 illustrates how these – and the external Schema.org Organization and Brick Location – are connected using Web Ontology Language (OWL) data and object properties. This query extracts all published services.

To support the service publication and discovery operations, a server that wraps pythons rdflib module was implemented. It hosts a model spanning both Brick and SAL. The service publication operation is translated to operations on the RDF store of the model. Interfaces for this operation – along with one for a generic SPARQL resolver – are exposed through separate services, each with a corresponding path in Bosswave. The generic SPARQL resolver interface is used for service discovery.

During the translation process of publication, triples have to be added to the store. However, some of those triples may already exist. To avoid duplicate entries, we employ a strategy of first attempting to look up an entity matching what we are about to create. If it exists we reuse it; otherwise, we create it.

6.3.1.2 CONTRIBUTION 2

The second contribution is a discussion of the impact of the SSS on the stakeholder ecosystem.

Based on the system described above, several factors becomes interesting to analyze. First, who are impacted by these changes, and second how do the changes impact the ecosystem and cost perspectives. The three different stakeholders identified in this evaluation are the customer, the entrepreneur, and finally the application developer. The customer is the maintainer of the actual on-site system, and the stakeholder making purchase decisions and sets the goals and rules for a company energy strategy. An entrepreneur is a company assigned to build or to retrofit a building. The application developer is defined as a company entity that develops applications, and services, that goes on top of the BOS. In the current business ecosystem, this stakeholder is mostly nonexistent. An example of an application developer could be an energy company implementing DR applications. Each role is not exclusive, so as an example, an entrepreneur can also be an application developer.

The overall ecosystem expected from a portable application ecosystem, especially also one with package dependency support in the future, is one where third-party developers can build applications for BOSs, just like they can for a phone with an app store. Having packages, or an app store would also change the way entrepreneurs deploy systems.

If a customer stakeholder wants to retrofit or create a new building, an entrepreneur stakeholder builds it for them. As this chapter tries to mitigate the costs of deployment, the costs of the entrepreneur should be reduced, and moved to the application developer. The reason for this; their applications are portable and therefore only implement changes once, and therefore this is a more economical choice for the entire ecosystem. This represents an added value, and thus, makes it easier for customers to buy into the proposed ecosystem. A DR orchestrator can be trivially installed in such an ecosystem as it will take the form of an application, thereby removing the DR-specific cost in the cost-benefit analysis. Also, This change enables an ecosystem of applications developers, and can potentially create new markets.

	Customer	**Entrepreneur**	**App. Developer**
Analysis	- Potential integrations with workflows	- Visible if customer demands can be met - Modularized upselling	- Defined subfield for App. - Make depending modules - Platform benefit, more users same code
Design	- Modularized options	- Less design time used (modules) - Complexity moved to app. dev.	- Added complexity
Implementation	- Cost shifting benefits	- Cost shifting benefits - Less design time - More documentation time for Models	- Added complexity - Application portability implementation
Deployment	- Less deployment time	- Lower deployment time - Potential for containerized deployment	- Can be containerized - Simplify application delivery
Operational	- Less cost to upgrade later - Deep abstracts are more complex - Simpler abstractions	- More aftermarket service options - Less maintenance due to autoconfig modules - Easy upgrade of apps/services	- Easier to market defined functionality modules - More difficult to debug models

Table 6.2: Costs impacts, benefits and disadvantages associated with each phase, distributed relevant across stakeholders.

Table 6.2 shows the different stakeholders mentioned before, as well

as the different phases of a BOS, including the development life-cycle, depending on stakeholder. The phases concentrate on cost impacts, and are defined like this: (1) Analysis: Analyzing the problem, and if a solution should/can be built. (2) Design: Concentrating on designing a viable solution for the problem at hand. (3) Implementation: Building the software, or system needed. (4) Deployment: Deploying the software, be it on-site, or in a software repository, depending on the stakeholder. (5) Operational: Keeping the building and software operational, including potential after-marketing opportunities.

One of the main benefits of automating BOS service and application installations is cost shifting. Cost shifting is the idea of moving costs from one part of the business ecosystem to another, where the overall benefit is better for all, or most, stakeholders involved.

Moving to auto-configured services also introduces ideas of modularizing services and applications, and even containerizing them with options like Docker [87]. This change forces a thought pattern of functionality packages, that are easier to sell for the entrepreneur and to understand for the customer. As a side effect of changing to auto-configurable modules, a potential for a market for third-party developers is created, which can find niche areas, or integrate their product suites cost-effectively into building ecosystems. For both entrepreneur and application developer, this mindset of modularizing creates clear packages of functionality to market.

The Entrepreneur. With a modularized system that is auto-configurable, it is easier for an entrepreneur to know if they can meet customer demands in the analysis phase. This clarity is due to how a modularized and auto-configurable system changes the approach to functionality, and how clear this function is defined. It also brings a different benefit; the potential up-selling. With a modularized system, it is easier to offer specific functionality to a customer, as each function is well defined, and to know the price of implementation and deployment. Also, clear functionality descriptions, for both services and applications, make it possible to create a service and upgrade after-market.

For the entrepreneur implementation costs get drastically lower, as microservices are deployed that does not need configuration, other than Brick which potentially can be auto-generated, by using the tool called Scrabble made by Koh et al. [85]. Koh et al. even provide a benchmark and development framework to help to test and to integrate with the brick model, called Plaster [93]. Fundamentally, a part of

the cost of deployment is shifted to the application developer, as the burden of configuring the application is taken over by the application developer. Also, fewer application developers would need to work at the entrepreneur company to get a viable product, as less configuration and functionality needs to be developed.

The Brick and SAL model could signify a cost increase though, as time is needed to describe the hardware configuration. If the entrepreneur is retrofitting a building, as opposed to creating an entirely new system, a part, or all of the Brick model can be generated automatically, as mentioned earlier. Third-party application developers would develop most of the applications unless the entrepreneur also acts as an application developer.

The Application Developer. For the application developer, there is an initial complexity added to the development process, but also added opportunities to make the entrepreneurs dependent on their services, as less software development expertise is present at the entrepreneur companies. The initial complexity addition is mostly centered around the concepts of SPARQL [69], RDF [62], OWL [83], Brick [74, 73], and the SAL [3]. For a developer though, these concepts are relatively easy to learn. Some cost though, will go into understanding how the entrepreneurs are creating and building the Brick models, so queries to it work across buildings. This understanding is essential for the application developer, as it is what provides the main benefit of targeting multiple buildings at once.

A potential new market that does not exist currently for BMSs creates an opportunity to create services that add specific functionality, like data analysis tools, that other developers can use. This change means that other application developers become possible customers too. To do this effectively, a package manager that can install dependencies would need to be created, as it would support this kind of business ecosystem.

With applications modularized as separate functional services and packaged for auto-configuration, the next obvious move would be to containerize the application for the package manager, for simplified delivery. Containerization is an option for the developer, not a requirement for this work.

The Customer. When analyzing the impacts on the customer, several points are worth mentioning. First, the initial deployment costs should be reduced due to cost shifting, by moving complexity, from

the entrepreneur to the application developer. This cost reduction is achieved as the application developer develops for multiple building types, not for specific buildings. Also, the actual deployment time of the system should be significantly shorter, because it is easier to set up, resulting in lower pricing of the installation.

As addressing a BOS from a modular functionality perspective with automatic configuration is more straightforward for all stakeholders to understand from a business perspective, it is easier to upgrade the BOS functionality in the future, or retrofit old buildings to support specific functionality. For the customer, the abstractions of functionality and modules are easier to understand and buy into, than buying entire new systems. This is where a DR application comes in to play, as it is easier to calculate a defensible ROI.

Adding all the different technologies on top of a traditional BMS, like Brick, SAL, and more, also brings complexity. So, for the customer it is easier to understand the abstractions they need, but more difficult to understand the depth of the system itself if they should need to implement functionality by themselves in the future.

6.3.1.3 Contribution 3

The third contribution is a discussion and evaluation of the SSS. In the approach section of the paper, several factors the SSS should support were defined. The first set of requirements were functional, or evaluative. First is service discovery, which allows for a service or application to find the information it needs; information which is being provided by another service. Second is being able to publish services into the SAL dynamically at runtime. The third is demonstrating the above two works as intended. All three factors were demonstrated by the case study, which implemented a new version of OccuRE, and the SSS.

Other non-functional requirements defined, was that the tooling implementation should not compromise the intended functionality of Brick or the SAL, that a developer should not maintain the RDF concepts related to the SAL, and finally, that publishing to the SAL should not require RDF knowledge. JSON is used to publish, and only requires a simplified understanding of the SAL. It was considered if this kind of abstraction should also be applied to the service discovery role of the SSS, but it was decided against as SPARQL would already be needed to read from the Brick and SAL ontology models. Brick and

SAL were hosted by the SSS and allowed for standard Brick, and SAL queries using SPARQL, thereby upholding the intended functionality of the two ontologies. The SAL was modified but did not compromise the intent of the SAL. Instead, it improved on it and did not change its intended functionality. Also, the SSS maintains the RDF concepts for the SAL, thereby not requiring application developers to maintain these.

Modules	Lines of Code	
	Original Implementation	Service Adaptation
dataservice	274	216
publisher	40	56
predictor	126	126
main	546	112
Total	**986**	**510**

Table 6.3: Comparison between the Original Implementation and the Refactored Code for the Service Support System Adaptation.

The OccuRE implementation demonstrated the effectiveness of the platform, as a 48% reduction in code was achieved, and in a specific case even a 80% reduction as detailed in Table 6.3. Therefore, all of the above requirements were satisfied with the design choices made, except for the above issues. Also, the platform allowed for removing hard-coded data from OccuRE, that was previously not discoverable.

6.3.2 MAIN CONTRIBUTIONS IN PAPER [6]

6.3.2.1 CONTRIBUTION 1

The first contribution is the design and implementation of the OPM, that deploys building services and drivers.

The basic concept of the OPM is to deploy containerized services and drivers with their depending packages, by leveraging ontology-based descriptions of the physical aspects of the building, and the drivers and services contained in the BOS.

Figure 6.5 shows how this OPM conceptually manifests itself. Several sources can request installations; A scriptable command line application, manually by using a web interface, or by using a pre-existing service in the BOS. The request is picked up by the OPM, that recurs-

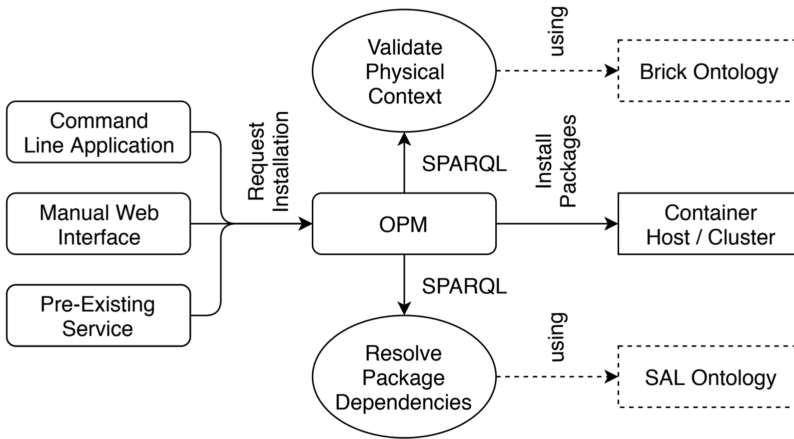

Figure 6.5: The Ontology based Package Manager Concept.

ively validates the package dependencies, and validates the building instrumentation supports the package. Package dependencies are resolved by the OPM, using an internal RDF server, which leverages the SAL [3, 4, 5] ontology. The SAL description for each service in the OPM describes a generalized version of the service that can be installed. This description encompasses all of the endpoints that can be used to interface with the service, and what data types it provides and in what context in relation to Brick. The OPM checks if a service is installed already, as well as if dependencies are already satisfied. These checks are done using the SSS [5]. The physical context is validated for each dependency using a SPARQL [69] query that is executed on a model using the Brick ontology [73, 74]. The SSS hosts both the Brick and SAL model, but the dependency resolution model is stored and handled separately in the OPM. The SSS handles the running states and contexts (SAL and Brick) for the BOS, while the OPM handles potential new packages, and what can be added to the running system.

The OPM is packaged as a Docker container and can exist in several different scenarios, as shown in Figure 6.6. Docker can exist as a Docker Swarm, or as a single Docker host. The OPM is not restricted to either, and as the figure shows, several instances of the service can exist. One for each Docker swarm, or one for each single Docker host. If several swarms are present in the setup, or several hosts, each of these would require an instance of the OPM to be able to install services on that

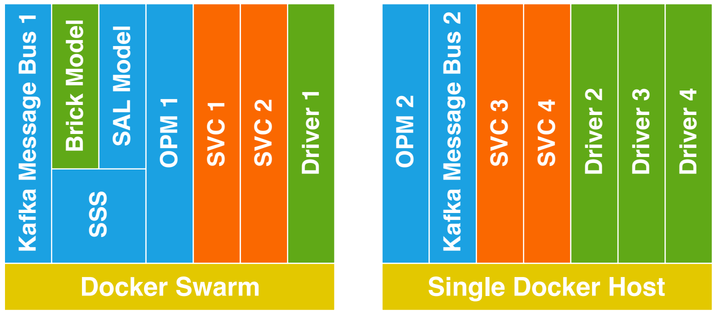

Figure 6.6: Host Overview: OPM required for each swarm or host. Drivers and services can be distributed. Only one SSS should exist for the setup. Colors correspond with the deployment phases described in Figure 6.8.

host. The Kafka Message Bus [50] is the communication bus. Drivers (marked with green) are the adapters between the instrumentation of the building and the Kafka message bus. Services (marked with orange) are the services handled by the OPM. Both services and drivers are not restricted to one host and can be distributed between them as the figure shows. The containerization of the microservices is a deliberate choice, as it forces developers to include all its code dependencies for an application, or expose only a few settings that need to be configured. This choice minimizes complications when someone not familiar with the application needs to deploy it. Also, containerization, in this case with Docker, allows for stored versioning of the applications, easier upgrades between versions.

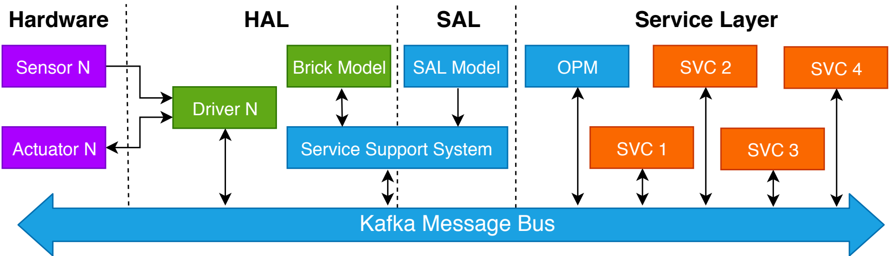

Figure 6.7: BOS environment with color coding for order of deployment. Excluding physical hosts. Colors correspond with the deployment phases described in Figure 6.8.

Figure 6.7 describes the setup from a logical application level. The

figure details how several hardware devices are connected to Kafka by the help of the drivers. The colors of both Figure 6.6 and Figure 6.7 pertain to the phases the component is usually deployed.

6.3.2.2 Contribution 2

The second contribution is an evaluation of the OPM. The evaluation method, mainly focuses on measuring the mitigation of deployment costs. To do so, one must understand the phases for deploying a BOS and what each phase entails. Figure 6.8 shows the phases of a BOS

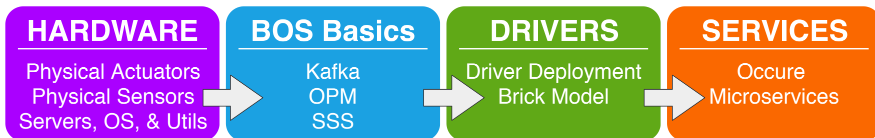

Figure 6.8: A typical BOS Deployment Procedure, divided in phases, with associated tasks.

and the components of the system that are installed and configured for that phase. The figure is color coded to show how each phase pertains to Figure 6.6 and 6.7. Each phase needs to complete before the next begins, as each phase has a dependency on a component in the previous phase. The first phase, hardware, pertains to the physical devices needed for the BOS to function. These are the instrumentation in the form of sensors and actuators, but also the servers where the BOS is going to reside. In this phase, the operating system and utilities needed are also installed. The second phase installs the essential components of the BOS that are common for all deployments of the BOS. In this case, this includes Kafka, the OPM, and the SSS. Once essential components are installed, the drivers are deployed. These drivers are specific to the building instrumentation, and therefore, the number of these, and the choice vary. At this stage, the Brick model is also finalized. If the drivers do not require Brick model validation, or the model is complete, the OPM can install them. The Brick model can take considerable time to create, but could ideally be provided by the entrepreneur installing the building instrumentation, as they possess most of the knowledge needed. Once the brick model is present, the OPM can validate the physical context, and therefore install service type packages with hardware requirements. Once the needed functionality is installed, the BOS

is ready.

Modules	Lines of Code	
	OccuRE	**OccuREv2**
sensor resolver	211	0
strategy resolver	101	79
publisher	40	0
main	254	134
Total	**606**	**213**

Modules	Lines of Code	
	PLCount	**PLCountv2**
strategy handler	0	180
publisher	0	29
strategy	166	166
main	459	0
Total	**625**	**384**

Table 6.4: Comparison between OccuRE and OccuREv2 and PLCount and PLCountv2 respectively.

To evaluate the OPM, OccuRE was restructured to fit into a OPM deployment strategy. Specific details about OccuRE and this restructuring can be found in the original paper in Chapter 14, and it is worth to mention that this is a substantial evolution of the restructuring performed for paper [5]. In Table 6.4, we compare the between the original implementation and refactored the code for both OccuRE framework and the PLCount estimation strategy. For all OccuRE and PLCount modules, we were able to achieve a 64% and 38% code reduction of the original codebase respectively due to the SSS, with SAL and Brick, and the OPM restructuring of how services are created and deployed.

To measure the actual deployment cost benefits, deployment times for the BOS was measured three times for the manual approach, and the OPM based approach. Table 6.5 shows the measured time for the manual install approach and the OPM approach. The time is split into the deployment phases previously discussed in Figure 6.8.

	Manual			**OPM**		
Iteration	**1**	**2**	**3**	**1**	**2**	**3**
Hardware	6	6	6	7	6	6
BOS Basics	69	20	14	1	1	1
Services	18	7	4	1	1	1
Accumulated Time	93	33	24	9	8	8

Table 6.5: Time in minutes to prepare the BOS and OccuRE, divided into phases and iterations.

Table 6.5 shows the manual deployment time takes considerably

longer but decreases with experience when deploying the system. Deploying with the OPM has consistent installation times that are significantly lower than the manual approach, even at the first iteration. Installing the BOS basics was mostly caused by unclear error messages to dependencies that needed to be debugged. These dependencies varied from the Java runtime environment, Python environment versioning, missing packages, and more. While many of the delays in the initial iterations can be automated, if there are standard deployments, it does not solve the issue of different versions of runtimes and dependencies. Moreover, building instrumentation exists for decades, and as systems are maintained and software changes, a technician can often end up with installation or maintenance durations that are closer to the first iteration, than the third. The consistency in deployment duration with the OPM shows how the combination of containerization, dependency resolution, and package manager functionality can drastically minimize costs, and require less prior knowledge of the software being deployed. Also, the evaluation does not provide a complete picture, as the deployment of the BOS was performed with personal having some degree of experience with partial deployment of a BOS. Being new to BOSs, in general, would most likely result in a more significant gap in deployment durations, between the two approaches. At the same time, the physical context of the building is also being validated, which ensures a markedly higher likelihood that the software being deployed works with the instrumentation. This validation results in further time savings, but it has not measured at this time. Also, a typical BOS does not only contain the functionality of OccuRE but can potentially have hundreds of use cases and services ranging from simple control to complicated algorithms. This means that the complexity of a deployments services phase can be insurmountable for someone new to the area. The time savings seen for the deployment of OccuRE should also be multiplied by each service present on the system, resulting in more significant savings.

6.3.2.3 CONTRIBUTION 3

The third contribution is a discussion of the impact of the OPM on the stakeholder ecosystem, also discussed in paper [5].

Several factors change with the introduction of the OPM. The relevant stakeholders are entrepreneurs that deploy and maintain the BOS,

developers that create the software that runs on the BOS platform, and finally the customer, that buys and uses the system in day to day operations.

For entrepreneurs, a few parameters change. First, expertise levels required of the technicians can be of a narrower scope. This change allows for shorter employee training periods, as well as leaving the more experienced technicians to deal with more complex tasks, instead of deployments. Second, as complexity is moved away from the entrepreneurs, and to the developers, an opportunity for creating a more potent value proposition for customers presents itself. This opportunity is due to employees with expertise being free to explore application functionality instead, and exploring functionality to the BOSs; they sell, instead of assimilating in-depth and intricate knowledge about specific services in the system, with little to no direct value to the customer. Third, technicians that require less training tend also to be a less expensive workforce, again creating cheaper deployments that benefit both entrepreneur and customer alike. Fourth, aftermarket maintenance and upgrades for BOSs will be less of an investment for the customers, thereby potentially creating a relationship that generates more sales over time, as upgraded functionality is simple to add.

For application developers, several parameters change as well. Having a repository of software allows developers to have a place to expose potential customers to their software. It also allows for new indirect sales channels, where packaging, distribution, and installation, remove most of the friction of BOS related software. Potentially, customers could also be other developers that need specific problems solved. Having a frictionless point of distribution and deployment is also one of the first steps towards the vision of an application store for buildings, again creating new opportunities. Creating a frictionless distribution channel would also encourage developers to create new and untraditional applications that involve buildings, emulating the exploration of form factor functionality that was last seen in 2008 with mobile phones. However, for this to happen, BOSs need to be more widespread and have a clear path to monetization. An example of a potential application developer could be an energy provider, seeking to make companies integrate to DR initiatives. Having frictionless platforms, as described above, allows for an easier buy-in from customers, as the added functionality is cheap to install, and functions on top of their already deployed BOS.

Finally, the customers. Customers could benefit from an OPM on several levels. First, less costly deployments make an initial investment in a BOS more feasible. Second, a more frictionless installation experience could allow a customer to experiment and explore. Currently, this kind of exploration requires a high level of expertise. Third, cheaper access to upgrading or changing building functionality after initial deployment, potentially allowing companies to more easily integrate buildings functionality into day to day operations.

Generally, the potential impact of a more frictionless and cheap deployment of BOSs is in the interest of all stakeholders involved. Simply put, the OPM has a value proposition for all levels of the stakeholder ecosystem.

6.4 CONCLUSIONS

Tooling for a BOS was created and supports automatic configuration and publishing of services into a BOS based on XBOS, Brick and the SAL. The tooling, in the form of the SSS that implements the service discovery and publication roles, successfully allows services to configure themselves into a new environment, and publish their functionality. Also, a significant reduction in lines of code for services was demonstrated, totaling 48%, or in a specific case 80%. These changes enable entrepreneurs and application developers to reduce deployment and production costs.

Also, the OPM was presented, that deploys containers in a BOS. It introduces time effective and simple deployment of drivers and services, demanding less knowledge to use, while also validating the physical context of the building is supported by the packages. The OPM was used to deploy a BOS several times, and compared to the previous deployment method. Significant time savings were documented, with the OPM consistently having reliable deployment times.

The SSS and OPM impact on the ecosystem was also discussed. The changes are generally favorable for all stakeholders in the ecosystem. The application developer is the stakeholder that has the most cost shifted to them, but the creation of new market opportunities should offset the relatively low addition of cost in work hours.

In the introduction, three areas were identified that would need to be solved to reduce deployment costs of BOSs. The first, Application

Portability, was already mostly solved by the HAL, Brick, and the SAL. The second was auto-configuration of services that was solved by the SSS. The third, and last, automatic deployment of application dependencies was addressed by the OPM.

Part III

Conclusions

CHAPTER

7 FUTURE RESEARCH

This Ph. D. thesis has explored several avenues in relation to its goals, and that of the FlexReStore Project. However, new research paths have opened because of the research performed in the Ph. D. thesis, and several others where left unexplored. This chapter explores future research related to both paths.

The exploration of the retail stores left several avenues open. First, it would be beneficial to have explored more ways to make Building Operating Systems (BOSs) more interesting and financially viable for retail stores. Doing this would open up for new possibilities of making Demand Response (DR) a financially viable option to add to an already pre-existing BOS, given the retail stores relatively short acceptable Return on Investment (ROI) periods of three years. One suggestion already made was creating options for centrally managing the ventilation, cooling, and more from a central location. This would give retail stores the option to actively change settings as need throughout the year, and allow for policies to be made on the settings needed. Currently, most of the Building Management Systems (BMSs) in the buildings are left alone, or only changed if something troublesome disturbs operations, thereby leaving distinct opportunities for operation optimizations by a centrally controlled system. Unfortunately, this type of added functionality only brings tangible savings for larger retail chains or malls. More research should be performed in the direction of motivations to use BOSs in smaller retail stores.

Second, two of the retail stores, specifically the Supermarket and Hypermarket, had a significant Demand Response Potential (DRP) in the unexpected category: ovens. The Supermarkets' ovens could produce about 60% of the total DRP from that store, while the Hypermarket's, ovens provided almost 19%. This is a considerable amount of DRP. While it seems problematic to find an effective method of mobilizing this DRP sensibly, it could be worth the effort to explore.

Third, creating better incentive structures for DR, making it more

profitable to participate could also bring some attention to the technology. This is already a research field in itself, and incentive structures are already being explored.

Moving on to the Activity-Tracking Service (ATS). As part of the ATSs development and its testing, it became apparent that there could be value in exposing other kinds of information than just the activities. Alternative information like opening hours, area/store type, and more, could be useful for developers creating applications on top of the BOS. Further study into what kind of information that could be valuable, should be performed. If valuable information categories are found, the ATS should be expanded to incorporate these new types of information. Information like this would require being stored and structured semantically in a way that is easy to maintain for those that have access to the information, for example, store owners, as application developers would most likely not have this kind of information at hand if they are creating a generalized application.

In regards to the Service Abstraction Layer (SAL), it would benefit from additional maturation and verification from several stakeholders. First, it should be deployed on multiple buildings to validate the setup further. Second, it should be distributed to multiple service and application developers to gather further experience with their needs. This would mature the SAL markedly, and ensure it fits the needs of the developers. Just working with the OccuRE service, the SAL was extended several times, and the same is expected when working with a broader array of services an applications. Not only would the SAL be more matured by these research avenues by collecting practical experience and feedback, but also validate the design further.

Precisely like the SAL, the Service Support System (SSS) and Ontology based Package Manager (OPM) would likewise benefit from the aforementioned maturation process. Additionally, though, the OPM could benefit significantly from automatic package deployments when particular patterns are detected in the physical context of the building. This could result in plug and play functionality for hardware instrumentation, by automatically deploying drivers when particular instrumentation is detected, thereby further reducing deployment costs. Also, more direct integration of resource management into the OPM could be beneficial to catch up to the alternatives in this area.

The SAL, SSS, and OPM all impose changes to how applications are created and deployed, but also how the building space ecosystem

could work in the future. Further research into how the application developers can be encouraged to participate in the BOS ecosystem, would be beneficial, as applications for the BOS ecosystem, is essential for its value proposition to potential customers investing in such a platform.

CHAPTER 8

CONCLUSION

In this chapter, conclusions about the thesis work are drawn. This Ph. D. thesis explored the technical state of retail stores and the motivations and barriers of the stakeholders. Also, this thesis presents several software solutions, on top of Building Operating Systems (BOSs), to mitigate several of the aforementioned barriers.

This Ph. D. thesis has provided an overview of the screening of four different types of retail stores. This overview has been detailed from four different perspectives:

– The yearly usage in a categorized manner.
– The Demand Response Potential (DRP) based on the same categorizations.
– How the devices are controlled, and at what level.
– And finally challenges to leveraging the potential in retail stores, as well as some of their most notable motivations.

The retail store screenings show significant DRP even though there is a multitude of challenges to leveraging this potential. Regarding external control of the devices currently in the store, the picture looks rather bleak as not a single device in the screened stores can be controlled externally. Simply put, there is a need for change in how we build systems for buildings. To better integrate with BOSs, future loads need to be accessible externally by default, if they are to be integratable. Also, due to the changing showcases and floor layouts, and desire for central management, it is essential to implement a solution that is operable and changeable from the store, but also from a central management position. Distributed user and group functionality are a natural consequence of this requirement. Fortunately, retail stores are regularly renovated, and therefore, an opportunity exists for retrofit and introducing the required technologies. Of course, this kind of retrofit requires that a business model can be discerned and that several of the concerns of the stakeholders can be addressed. The following main concerns with

the largest impact on the software developed, were identified during the interviews conducted:

1. The main stakeholders care mostly about short term income; therefore, they are highly concerned about the effect of Demand Response (DR) on store efficiency, and its impact on potential sales. Several of these stakeholders expressed concern if a system could track their routines closely enough, to not initiate DR activities when it could impact store operations.
2. A retail store's future is never certain, which limits the retail store's investments to solutions that can have a tangible return of investment within three years. This is a significant limitation to DR based operations in retail stores, as hardware, software, and deployment is expensive.

The first concern was partially addressed by the Activity-Tracking Service (ATS), as it helps DR applications have better decision points for when a DR event can occur. However, during the development of the ATS, that creates more knowledge from operational models and serves this information to other services and applications, it became apparent that services portability between BOSs had not been considered. This lack in portability would drive up costs for DR applications, as the DR application would need to be implemented for each building. Also, a DR application in itself, would not warrant an entire BOS installation with hardware, as it is too costly compared to the gains, which conflicts with the second main concern identified above.

To mitigate this problem, the Service Abstraction Layer (SAL) was developed. The SAL learned from the experiences of Brick [74]. During its two iterations, that this Ph. D. thesis encompasses, it effectively decoupled the applications and services in the system from other services, thereby allowing for higher portability between buildings. This paves the way for cheaper DR applications, as one codebase can now be moved between buildings more efficiently. The SAL also came with several other benefits, such as the ability to over-provision the access to specific data, generated from different services, effectively enabling resiliency options like High Availability (HA), to create higher up-times for high priority applications. Also, changing interfaces of a service, or substitution of a given service, is now possible without changes to the rest of the system, as long as the types of information that is depended on, is provided by one of the services in the system.

Unfortunately, the BOS did not allow a service to make changes to the setup on the fly, which reduced some of the functionality and future possibilities of the SAL. To change this, the Service Support System (SSS) was developed, creating the needed tooling for auto-configurable building services, effectively solving the last barrier identified for portable applications and services in the BOS. The SSS hosts the SAL and Brick models, and allows services to dynamically make changes to the represented services in the SAL, without restarting services, or depending on static models. Also, the SAL and the SSS demonstrated how the code of service implementations could be significantly simplified.

To further reduce the cost of BOS deployments in retail stores, and thereby addressing the second concern even further, the Ontology based Package Manager (OPM) was developed. The OPM is an ontology-based package manager, that deploys containers in a BOS, and combines characteristics from containerization technologies, package managers, and ontologies. This OPM used the knowledge gained from the SAL research, to create a package manager that can deploy drivers, services, and applications in a BOS. It validates the physical aspects of the building are compatible with the packages to be installed, but also resolves dependencies between these packages using the SAL ontology. Not only did the OPM create significant time savings, but also created reliable deployment times from the first iteration, thereby ensuring the BOS is cheaper to maintain, even if it is old enough for the technician not to have much experience with it.

The contributions in both [5] and [6], discussed in this Ph. D. thesis, also provided an analysis of the potential future impact of these two systems to the stakeholders of the building space ecosystem. These stakeholders are the Customers buying into such a system, the Entrepreneurs deploying the hardware and BOSs, and finally the Application Developers. Potentials for new markets was discussed, as well as how the changes to the ecosystem could generate tangible incentives for all stakeholders in the building space ecosystem.

While the above contributions do not solve all of the challenges associated to making DR implementations attractive to Retail Store Stakeholders, it does make a significant stride toward a more cost-efficient solution, while also creating incentives for all stakeholders involved in the building space ecosystem.

PROJECT RELATED ACKNOWLEDGMENTS

We would like to acknowledge the contributions of Jens Gjesing from AURA Energi, regarding the screening of the stores, as well as the staff and leadership of the participating retail stores, that kindly lend us their expertise and time. This work is supported by Energinet.dk ForskEL, now managed by EUDP, in relation to the project FlexReStore (12413), and was also supported by the SDU ODEx project, that explores Open Data on Human Behavior.

Part IV

Publications

In this part, are reported, one per chapter, the following publications.

[1] Jakob Hviid and Mikkel Baun Kjærgaard.
'The Retail Store as a Smart Grid Ready Building: Current Practice and Future Potentials'. In: *Power & Energy Society Innovative Smart Grid Technologies Conference (ISGT), 2018 IEEE*. IEEE. Washington DC, USA, Oct. 2018.
DOI: 10.1109/ISGT.2018.8403354. **Published**.

[2] Jakob Hviid and Mikkel Baun Kjærgaard.
'Activity-Tracking Service for Building Operating Systems'.
In: *2018 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. IEEE. 2018, pp. 854–859. DOI: 10.1109/PERCOMW.2018.8480362.
Published.

[3] Jakob Hviid and Mikkel Baun Kjærgaard. 'Service Abstraction Layer for Building Operating Systems: Enabling portable applications and improving system resilience'. In: *2018 IEEE International Conference on Communications, Control, and Computing Technologies for Smart Grids (SmartGridComm)*. IEEE. Aalborg, Denmark, Oct. 2018, pp. 1–6.
DOI: 10.1109/SmartGridComm.2018.8587543. **Published**.

[4] Jakob Hviid, Aslak Johansen, Gabe Fierro and Mikkel Kjærgaard. 'Service Portability and Discovery in Building Operating Systems Using Semantic Modeling'.
In: *2020 IEEE International Conference on Pervasive Computing and Communications (PerCom) (PerCom 2020)*.
ACM. Austin, USA, Mar. 2020. **Submitted**.

[5] Jakob Hviid, Aslak Johansen, Fisayo Caleb Sangogboye and Mikkel Baun Kjærgaard.
'Enabling Auto-Configuring Building Services: The Road to Affordable Portable Applications for Smart Grid Integration'.
In: *Proceedings of the Tenth International Conference on Future Energy Systems (ACM e-Energy '19)*.
Phoenix, AZ, USA: ACM, June 2019, pages 68–77.
DOI: 10.1145/3307772.3328288. **Published**.

[6] Jakob Hviid, Aslak Johansen, Fisayo Caleb Sangogboye and Mikkel Kjærgaard. 'OPM: an Ontology Based Package Manager for Building Operating Systems'. In: *Proceedings of the International Conference on Internet of Things Design and Implementation* (*IoTDI 2020*). ACM. 2020. **In Submission**.

Figures and tables were minimally edited to fit the layout of this thesis, and they are otherwise identical to the ones in aforementioned publications. Text was copied verbatim, except for bibliography reference numbers and styles which are kept uniform for the entire thesis, and for minor spelling adaptations and corrections.

A local bibliography is reported at the end of each chapter, listing all references cited in the corresponding publication. All references are also repeated in the global bibliography at the end of this thesis, on page 271.

CHAPTER 9

THE RETAIL STORE AS A SMART GRID READY BUILDING: CURRENT PRACTICE AND FUTURE POTENTIALS

This chapter is a cosmetic adaptation of the following conference paper: Jakob Hviid and Mikkel Baun Kjærgaard. 'The Retail Store as a Smart Grid Ready Building: Current Practice and Future Potentials'. In: *Power & Energy Society Innovative Smart Grid Technologies Conference (ISGT), 2018 IEEE*. IEEE. Washington DC, USA, Oct. 2018. DOI: 10.1109/ISGT.2018.8403354. **Published**

The paper was presented at the Ninth Conference on Innovative Smart Grid Technology (ISGT 2018), sponsored by the IEEE Power and Energy Society (PES), in Washington, DC, USA, on 19 February 2018.

9.1 ABSTRACT

Retail stores use a considerable amount of energy. Therefore, this type of building is important to consider for providing demand-side flexibility services. However, existing work on demand-side flexibility has only considered narrow aspects of the loads and opportunities in retail buildings. This paper details a study on the loads and practices in different types of retail stores, as well as the flexibility potential. Four different retail store types were inspected, cataloged, and analyzed together with their energy usage. The results highlight that large percentages of the store loads are too diverse to group into usable categories. The usual categories are present, like Lighting, Frost, Cooling, Comfort Cooling, Ventilation, and one less common group: Ovens. In the supermarket screened, the ovens took up 60% of the flexibility potential and about 19% of the Hypermarket's potential. Flexibility potential numbers are detailed, as well as the current ability to control the loads. Retail stores are different from other sectors in regards to

their motivations for upgrading installations. Some of the notable requirements, potential challenges, and barriers to realizing the flexibility potential are highlighted in relation to these motivations. In conclusion, the current state of the controllable devices and the granularity with which they can be controlled are found to be lacking severely, if they are to be integrated into a modern Demand Response (DR) system. As retail stores are regularly renovated, an opportunity exists to introduce required technologies across stores at a faster pace than in other building types, if a business case exists.

9.2 INTRODUCTION

One possible solution for handling the fluctuating production by renewable energy sources is DR technologies. DR covers technologies and programs that enable and incentivize electricity consumers to change their load profiles. DR has traditionally been applied to address peak loads on the grid, e.g., hot summer load peaks in California, US [18]. However, the potential of DR also covers scenarios for handling emergency situations in the grid, local load peaks [19] and the integration of renewable electricity generation.

The retail sector is a large consumer of energy, e.g., in Denmark the retail sector consumed 9.344 TJ in 2013 [25] which equates to about 8% of the total industry sector consumption. This consumption in itself signifies the importance for the retail sector but is further underscored by the fact that electricity consumption and other energy costs are the largest cost after labor costs.

Previous work on DR has considered the retail sector with two different perspectives: i) considering retail stores as commercial buildings and using DR strategies known from other types of commercial buildings for light and ventilation control [18]; ii) considering retail stores with ample cooling and freezing loads and using these for load shifting (e.g. Shafiei et al. [33]). Therefore, previous work has failed at embracing the retail store as a separate building type for DR and apply a holistic view on the options for flexibility in consumption in the stores at large. For instance, in office buildings a goal is to optimize the indoor environmental conditions so that occupants can work effectively. When examining retail stores, a goal is to optimize the indoor environment to boost sales, e.g., in general, a high level of light and a very high level on

premium wares. Highlighting just one of the many differences when comparing retail stores and commercial buildings in general.

This paper summarizes the results of a holistic study of the flexibility options in retail stores. Instead of a top-down approach looking at the statistics from several stores, a bottom-up approach is applied by actually screening the stores. The study considers four different types of retail stores which include a hypermarket, supermarket, garden center, and a hard goods store. Each store has been screened to gather information on each device in the store. The usage and flexibility potential for each type of device has been estimated based on technical documentation, information from store staff and energy metering data.

The contributions of the paper are as follows: 1) Detail the devices and consumption of four different store types, using one years data. 2) Present estimates of the flexibility potential for each device type in these stores. 3) Document the level of control (e.g., manual or based on a Building Management System (BMS)) for each type of device in the four different store types. 4) Clarify potential pitfalls when enabling Internet of Things (IoT) and Building Operating Systems (BOSs) in retail.

9.3 METHODOLOGY

Four store types were surveyed: hypermarket, supermarket, garden center, and hard goods store. A hypermarket is defined as a store ranging from 5000 to 15000 m^2 and have a supermarket and department store section. A supermarket is defined as a store between 1000 and 5000 m^2. These store types were chosen as they represented different sectors with distinct concerns, were willing to put in the time and resources needed for surveying the stores, and giving feedback. All the stores are located in Denmark, and therefore local installation and weather conditions apply. The screening of the stores themselves was done by AURA Energi, which specializes in energy optimizations of buildings.

The screening process followed these guidelines: 1) survey the store and write down each piece of electricity consuming device, its wattage and operation hours. To gather information on operation hours several methods were used as appropriate: manual observations, schedules from the BMS [94], and experienced estimations from the store man-

agers and their staff; 2) collect a year of electricity metering data for the store at the available granularity and analyze this [95]; 3) combine the lists of devices and metering data to validate the collected findings, as well as filling out any missing information; 4) technicians access the stores BMS to collect data about which devices can be accessed and controlled from the BMS, and at what levels; 5) collect DR mobilization limitations by inspecting hardware specifications and limitations and consulting with manuals or manufacturers. The parameters for the data collected was derived from Piette et al. [18].

9.4 SCREENING RESULTS

The information collected is presented in four tables, that summarize the current usage, the Demand Response Potential (DRP), the control options, and mobilization limitations. DRP is defined as the maximum potential energy that could potentially be leveraged for DR purposes. The tables aim to answer: What is the DRP across the store types in question? And to which degree can they be controlled using Application Programming Interfaces (APIs)?

The current usage can be found in Table 9.1. Units are in MWh per year. The categories used are the largest common groupings found in the screening data. The loads of Lighting, Ventilation, Ovens, and Other have an estimated load strain based on the data from the BMS, observable patterns, and information given by the employees in the case of manually changeable loads. In the case of cooling (Frost & Comfort Cooling), the data is sourced from the metering data as they were measured separately. The difference in consumption between the first five categories and total usage is listed in the unidentified load column. The *hard goods store* had installed LED lighting within the last year before inspection. Therefore, the *total usage* has been calculated based on the ten months, where LED lighting was in use, by using the average to estimate the two previous months. The *unidentified* column of the *supermarket* and *hypermarket* can be attributed to falsely estimated usage of ventilation and lighting or external weather conditions that impact the ventilation systems power usage immensely.

Table 9.2 shows the DRP of the retail stores and is divided into the same categories as Table 9.1 with two exceptions, as *other* and *unidentified* are excluded. *Other* is excluded, since enabling all these diverse

	Lighting	**Ventilation**	**Cooling**	**Ovens**	**Other**	**Unidentified**	**Total Usage**
Garden Center	133 MWh	-	-	-	113 MWh	27 MWh	273 MWh
Hard Goods Store	212 MWh	128 MWh	-	-	12 MWh	205 MWh	557 MWh
Supermarket	478 MWh	141 MWh	648 MWh	364 MWh	154 MWh	116 MWh	1901 MWh
Hypermarket	967 MWh	613 MWh	847 MWh	462 MWh	715 MWh	205 MWh	3809 MWh

Table 9.1: Consumption across categories, Total observed consumption, and consumption which is unaccounted for.

	Lighting	**Ventilation**	**Cooling**	**Ovens**	**Total Max Usage**
Garden Center	35 kW (100%)	-	-	-	35 kW (100%)
Hard Goods Store	131 kW (56.7%)	100 kW (43.3%)	-	-	231 kW (100%)
Supermarket	77 kW (9.2%)	144 kW (17.1%)	113 kW (13.4%)	508 kW (60.3%)	842 kW (100%)
Hypermarket	176 kW (15.7%)	591 kW (52.7%)	143 kW (12.8%)	211 kW (18.8%)	1.121 kW (100%)

Table 9.2: Demand Response Potential (DRP).

devices to respond to DR requests would be impractical, complex, and extremely expensive; and *unidentified* because a load needs to be identified to determine its DRP. The numbers are measured in kW. Devices with unidentified wattage are not included in this table. The DRP is represented by the wattage of a device, or in this case, the accumulated possible wattage used at any given time. To attain a more holistic study, sub-monitoring was decided against as it would be limiting the paper to devices that could be practical to sub-monitor.

A future system that controls a multitude of devices, like in a retail store, needs programmable access to the devices it has to control. As systems become larger and more complex, interoperability and open access to these APIs become a priority. Therefore, Table 9.3 shows the DRP in kW based on how devices are controlled. Only devices identified and within the categories listed in Table 9.2 are included. *No Control* means the devices are only controlled by turning a switch on and off. *On-device control* means the individual device is capable of being configured to respond to the environment in a limited fashion. The *BMS* columns are divided into two categories. The first, *closed*, is a typically proprietary and closed system that external programs cannot interact with. The second category is for *open* BMSs that provide external integration access via APIs or other means.

Devices react differently to changes requested by a DR event and can have limits in the frequency and duration, it can be used. Table 9.4 seeks to clarify the DR mobilization limitations found in the inspected systems. The values of the diagram are split into the following possibilities: **S**econds, **M**inutes, **Q**uarter hour, **H**ours, **D**ays, and **U**nlimited. Each value type implies the metric used is lower than the next, so 10 minutes would be M, but 25 would be Q. A notable trend in Table 9.4 is how the categories are aligned over the different retail store types. Response times are within minutes of each other and fall into the same time category.

9.5 SCREENING CONCLUSIONS

Notably in Table 9.1, the electricity usage differs greatly between the retail store types. Particularly the usage percentages between categories. Besides the *hard goods store*, all the other stores have a high percentage of loads placed into the *other* category. These loads are small machines

	No Control	**On-Device Control**	**BMS - Closed**	**BMS - Open**
Garden Center	449 units / 35 kW	-	-	-
Hard Goods Store	-	-	3886 units / 231 kW	-
Supermarket	315 units / 530 kW	2 units / 113 kW	891 units / 200 kW	-
Hypermarket	980 units / 247 kW	7 units / 153 kW	2868 units / 722 kW	-

Table 9.3: Demand Response Potential (DRP) overview based on how they are controlled.

	Lighting				**Ventilation**				**Cooling**				**Ovens**			
	Frequency	Duration	Ramp Up	Ramp Down	Frequency	Duration	Ramp Up	Ramp Down	Frequency	Duration	Ramp Up	Ramp Down	Frequency	Duration	Ramp Up	Ramp Down
Garden Center	Q	U	S	S	-	-	-	-	-	-	-	-	-	-	-	-
Hard Goods Store	Q	U	S	S	Q	U	M	M	-	-	-	-	-	-	-	-
Supermarket	Q	U	S	S	Q	U	M	M	Q	U	M	M	D	H	M	S
Hypermarket	Q	U	S	S	Q	U	M	M	Q	U	M	M	D	H	M	S

Table 9.4: Demand Response Mobilization Limitations.

used for various purposes, mostly in the store floor, such as lighting products, aquariums, TVs, and more. These devices will be a hard task to include into the DR activities as they represent a large mass of devices that does not cluster well into a category with similar purposes. The *garden center* is notable in this regard, as about 41% of the loads fall into this category.

The *hard goods store's* unidentified electricity usage is also interesting as the *lighting* and *ventilation* is controlled on a schedule and only few parameters would have an impact on the calculated numbers. This seems to suggest that a considerable amount of the unidentified loads would have a good chance of belonging to the *other* category. The above observations point strongly suggest the content of the stores are diverse, and differ immensely from each other based on the category of products the store sells.

The fact that *ovens* ended up being a considerable part of the total usage of the *supermarket* and *hypermarket* is interesting, and makes the *ovens* an obvious candidate to examine further. The DRP of these devices is considerable, but these ovens are tightly integrated into the processes of the store, which makes it hard to leverage the potential. Furthermore, using these devices to heat up a store is impractical. Nevertheless, the numbers show a potential candidate for integration if a practical usage pattern can be discerned.

In Table 9.2 the total wattage column shows the *supermarket* and *hypermarket* are capable of delivering the largest amount of DRP compared to the total usage in Table 9.1. The potential is largely attributed to the *cooling*, *ovens*, and *ventilation*. Here the immense overcapacity of the ventilation system of the *hypermarket* stand out, as well as the oven capacity of the *supermarket*. The ovens take up about 60% of the *supermarkets* maximum DRP, dwarfing the rest of the stores' loads. The *hypermarket* is able to deliver about 19% of its maximum DRP from these ovens. These numbers definitely make the category hard to ignore. The ventilation systems, especially of the *supermarket* and *hypermarket*, are also able to deliver substantial DRP when compared to the less flexible *lighting* systems and *cooling* systems. One possible explanation for this discrepancy, though, could be the hugely over-dimensioned ventilation systems compared to the stores' actual needs.

When analyzing the available retail stores for DRP, it is evident the current state is found severely lacking in regards to control options. The control distribution in Table 9.3 indicates that only the *hard goods*

store actively engages in controlling the building from their BMS. The *hypermarket* and *supermarket* follow behind, but are both terribly behind when considering what their BMSs are capable of doing with these devices. Most of the devices connected to the BMS can only have their status observed. These systems would require extensive retrofits to be leveraged for DR. The garden center is even worse off as 0% is controllable from a BMS. The findings in the control distribution suggest a considerable attitude change is needed when building new stores. Not only should these stores have their loads connected to a BMS or equivalent, but they should also strongly consider the importance of open accessible BMSs that can be interfaced to via programmable and integrable APIs. Not a single store has one single device that can be controlled openly, which effectively leads to the definite conclusion: 0% of the screened stores are compatible with an external signaled DR activation strategy, unless they effectively replace the current BMS. Current research in Software Defined Buildings [29, 28, 27, 46], that also facilitates DR, likewise requires open access to the devices. Last but not least, all the stores observed only controlled the lighting in large clusters that did not fit with the floor area. As a consequence, it is hard for a store owner to differentiate certain sections of a store by using the BMS. Better granularity could give the store owner more flexibility. This granularity would also open up for a possibility of introducing DRP, reducing light usage in certain sections of the store where it is less important.

Table 9.4 details mobilization limits on the systems inspected. The mobilization limits align in each category. This means some generalization in implementation and mobilization can be expected in the sector as this is the case in all four categories. The ramp up and down times listed, are typically not instant due to the physical properties of the devices. The *ovens* is the category that differs most from the other categories as they have several extra limitations. The ovens could theoretically be turned on and off instantly as often as necessary. However, this kind of flexibility is unrealistic as the load would be unpractical to move more than once or twice a day. A baking program can take several hours to complete, and after the dough has raised it needs to be baked relatively quick. Baked bread typically also needs to be ready at specific times of the day. These are natural limits for these ovens as it would otherwise conflict with the retail stores goals.

9.6 LEVERAGING THE DEMAND RESPONSE POTENTIAL

During the screenings, observations, and conversations with staff, it became apparent several factors were different, when creating systems for retail stores compared to homes and offices. To successfully implement DR and leverage DRP in a retail store setting, it is important to understand how it differs from other settings. Especially motivations for controlling the building, and how the environment of the building differs. In this work we combine our own observations with those of Zheng et al. [75] who outline motivations for the retail sector to provide DR based on a literature study.

The focus on sales drives the store owners desire to control the building environment. The purpose is to create an environment that motivates the customer to buy. Many of the processes that happen in a store are time sensitive and directly related to loads of the store. This can also be said for offices and residential homes. However, a customer will rarely wait ten minutes extra for their purchase to get ready because an oven is turned off. Situations like this results in lost sales, while it would be a minor inconvenience at home or in an office. Especially lighting is sensitive and directly impact sales by drawing attention to products. This is often done by extra lighting around and above the product showcase. The settings and setup of these showcases and floors are often changed when high profile products change. As this behavior directly impacts sales, it is important the store can maintain a differentiated lighting setting. And as such, there is a need for a more dynamic lighting configuration.

All stores agree that schedules and temperature levels are set centrally and typically encompass multiple stores. As mentioned above, the staff needs to be able to change several settings on demand but should have no control over other parameters of the store. As a consequence, a complex security scheme is required, if the loads around the products are to be leveraged for DR purposes. The standard user and groups scheme with granular rights management on specific loads should suffice. Central control should be seen as an opportunity to make DR a central decision for management, to control multiple stores instantly and consistently.

Currently, the size of the regulation zones are large in stores, but there is a clear desire for more granular control, as the sales area frequently changes. The sales floor is typically the largest room with many control

zones close to each other. Consideration needs to go into how each control zone impacts the other visually; as one end of the store can often be seen from the other end. Any distraction is a potential lost sale from the perspective of a store owner.

A retail store typically seeks to facilitate an experience where the customer does not need to leave the store. As a consequence, it is usually possible to buy food, as well as to see the products showcased and running. Food preparation and product showcasing use electricity, as does the the butcher's section. These areas use specialized tools and leads to a vast array of machines. When comparing these devices to the other domains, a retail store's load diversity resembles a residential home better than that of an office. If the *other* category is to be leveraged as potential, a complex task needs to be solved that is prevalent in both the retail and residential domains.

The occupancy patterns also differ. Residential homes occupancy and usage patterns are dependent on the occupant's work and life patterns, while offices occupancy and usage pattern typically depend on work regulations and the type of work. In retail, the largest loads typically follow schedules and processes in the store. These typically start when the employees arrive at the store and then escalate when customers arrive; new processes start, and food preparation devices and registers are used. These patterns allow for relative predictability in several areas of the store.

If the requirements of the retail store are not taken into account, potential solutions will not seem palatable to retail store stakeholders, as it will make several processes in the store cumbersome. An example mentioned earlier is the simple need to be able to rearrange the floor to accommodate sales of a new premium product. Needing to have personnel from a central entity called in to accommodate these changes is a clear hindrance for adoption in retail. Many retail store loads are intertwined with processes, like using the ventilation system for drying the floors, as well as the ovens. It could prove useful to map these processes and how they are related to the loads. Having a clear overview could potentially help future products leverage loads that would otherwise seem unfeasible to integrate in DR operations. As such, further study should be undertaken into the specific demands and difficulties retail stores have in order to help increase the incentives for stores to invest in these solutions; thereby increasing the potential participants in DR programs.

9.7 CONCLUSIONS

This paper has provided an overview of the screening of four different types of retail stores. This overview has been detailed from four different perspectives: 1) the yearly usage in a categorized manner, 2) the DRP, 3) how the devices are controlled and at what level, and 4) challenges to leveraging the potential in retail stores.

Some of the important findings from the screenings are as follows. The *Other* category from Table 9.1 shows a relatively large number of diverse devices. These devices are hard to integrate into the DR as they have very few characteristics in common. The inclusion of *Ovens* in Table 9.2 show a huge potential. While the inclusion of ovens into DR is highly questionable because of their usage patterns and their tight integration into the processes of the retail stores, the immense numbers speak for themselves. These tight integrations are evidenced by the limitations set on the mobilization limits in Table 9.4. The Supermarkets' ovens can produce about 60% of the total DRP. Even in a store with fewer ovens, like the Hypermarket, the ovens provide almost 19% of the total DRP. Currently, several of the loads in a retail store tend to be intertwined deeply with processes happening within the store. To successfully leverage the potential of these loads, a mapping is to be made between the loads and how and when they are used in the processes.

The ventilation system of these stores surveyed also stand out and shows good potential for integration as 43%, 53% and 17% of the potential lies here. The *Supermarket* with 17% is only this low because of the aforementioned *Ovens*. Also, the *Hypermarkets* ventilation system was especially interesting because of the over-dimensioned capacity of 591 kW.

Regarding external control of the devices currently in the store, the picture is rather bleak as not a single device in the screened stores can be controlled externally. There is simply a need for change in how we build systems for buildings. With the direction current research is going in Software Defined Buildings [29, 28, 27, 46], which are also used for DR, future loads need to be accessible externally by default, if they are to be integratable. Also, due to the changing showcases and floor layouts, and desire for central management, it is important to implement a solution that is operable and changeable from the store alone, but also from a central management position. Distributed user

and group functionality are a natural consequence of this requirement. Fortunately, retail stores are regularly renovated, and therefore an opportunity exists for retrofit and introducing the required technologies. Of course, this kind of retrofit requires that a business case can be established. However, with some of the mentioned discovered requirements, and by mapping more of this sector, a feasible solution could potentially be found. If found, it should be possible to retrofit retail stores at a faster pace than other building types.

The retail store screenings show tremendous DRP even though there is a multitude of challenges to leveraging this potential. This paper has detailed the motivations and difficulties found in the screenings, but much work remains in both charting these challenges, as well as the search for solutions. They are solvable though, and further research in this area will undoubtedly help leverage previously untapped potential.

ACKNOWLEDGMENTS

We would like to acknowledge the contributions of Jens Gjesing from AURA Energi, regarding the screening of the stores, as well as the staff of the participating retail stores, that kindly lend us their expertise and time. This work is supported by Energinet.dk ForskEL for the project FlexReStore (12413).

REFERENCES

[1] Jakob Hviid and Mikkel Baun Kjærgaard. 'The Retail Store as a Smart Grid Ready Building: Current Practice and Future Potentials'. In: *Power & Energy Society Innovative Smart Grid Technologies Conference (ISGT), 2018 IEEE*. IEEE. Washington DC, USA, Oct. 2018. DOI: 10.1109/ISGT.2018.8403354. **Published**.

[18] Mary Ann Piette, Sila Kiliccote and Girish Ghatikar. 'Field Experience with and Potential for Multi-time Scale Grid Transactions from Responsive Commercial Buildings'. In: *ACEEE Summer Study on Energy Efficency in Buildings*. 2014.

[19] Mikkel Baun Kjærgaard, Krzysztof Arendt, Anders Clausen, Aslak Johansen, Muhyiddine Jradi, Bo Nørregaard Jørgensen, Peter Nelleman, Fisayo Caleb Sangogboye, Christian T. Veje and Morten Gill Wollsen. 'Demand response in commercial buildings with an Assessable impact on occupant comfort'. In: *SmartGridComm 2016*. 2016, pp. 447–452.

[25] Danmarks Statistik. *ENE3H: Bruttoenergiforbrug i fælles enheder efter branche og energitype [Accessed: 23/2/2017]*. http://www.statistikbanken.dk/ENE3H. 2017.

[27] Stephen Dawson-haggerty, Andrew Krioukov, Jay Taneja, Sagar Karandikar, Gabe Fierro, Nikita Kitaev and David Culler. 'BOSS: building operating system services'. In: *NSDI '13* (2013), pp. 1–15.

[28] Gabriel Fierro and David E Culler. 'XBOS: An Extensible Building Operating System'. In: *Proceedings of the 2nd ACM International Conference on Embedded Systems for Energy-Efficient Built Environments*. ACM. 2015, pp. 119–120.

[29] Thomas Weng, Anthony Nwokafor and Yuvraj Agarwal. 'Buildingdepot 2.0: An integrated management system for building analysis and control'. In: *BuildSys'13*. ACM. 2013.

[33] S. Shafiei, Henrik Rasmussen and Jakob Stoustrup. 'Modeling Supermarket Refrigeration Systems for Demand-Side Management'. In: *Energies* 6.2 (Feb. 2013), pp. 900–920. ISSN: 1996-1073. DOI: 10.3390/en6020900.

[46] Stephen Dawson-Haggerty, Xiaofan Jiang, Gilman Tolle, Jorge Ortiz and David Culler. 'sMAP: a simple measurement and actuation profile for physical information'. In: *SenSys'10* (2010). DOI: 10.1145/1869983.1870003.

[75] Zheng Ma, Joy Dalmacio Billanes, Mikkel Baun Kjaergaard and Bo Nørregaard Jørgensen. 'Energy Flexibility in Retail Buildings: from a Business Ecosystem Perspective'. In: *Proceedings of the 14th International Conference on the European Energy Market*. Dresden, Germany: IEEE, 2017.

[94] E.J. Knibbe. *Building management system*. US Patent 5,565,855. Oct. 1996.
URL: https://www.google.com/patents/US5565855.

[95] Almir Mehanovic, Emil Sebastian Rømer, Jakob Hviid and Mikkel Baun Kjærgaard.
'Clustering and Visualisation of Electricity Data to identify Demand Response Opportunities: Poster Abstract'.
In: *BuildSys 2016*. 2016, pp. 233–234.

CHAPTER 10

ACTIVITY-TRACKING SERVICE FOR BUILDING OPERATING SYSTEMS

This chapter is a cosmetic adaptation of the following conference paper: Jakob Hviid and Mikkel Baun Kjærgaard. 'Activity-Tracking Service for Building Operating Systems'. In: *2018 IEEE International Conference on Pervasive Computing and Communications Workshops* (*PerCom Workshops*). IEEE. 2018, pp. 854–859. DOI: 10.1109/PERCOMW.2018.8480362. **Published**.

The paper was presented at the 2018 IEEE International Conference on Pervasive Computing and Communications Workshop, PerFoT, in Athens, Greece, on 19 March 2018.

10.1 ABSTRACT

Many high consuming electricity loads in retail stores are currently highly intertwined in human activities. Without knowledge of such activities, it is difficult to improve the energy efficiency of the loads operation for sustainability and cost reasons. The increasing availability of Internet of Things (IoT) sensors and devices promise to deliver rich data about human activities and control of loads. However, existing proposals for Building Operating Systems (BOSs) that should combine such data and control opportunities do not provide concepts and support for activity data. In this paper, we propose an Activity-Tracking Service (ATS) for BOSs. The service is designed to consider the security, privacy, integration, extendability and scalability challenges in the building setting. We provide initial findings for testing the system in a proof of concept evaluation using a set of common IoT sensors and devices.

10.2 INTRODUCTION

The retail sector is a large consumer of energy. The consumption in itself signifies the importance for the retail sector to address energy efficiency. Furthermore, this is underscored by the fact that electricity consumption and other energy costs are the most significant cost after labor costs. However, a challenge is that many retail store loads are intertwined in activities within the store. For example, when doing cleaning using the ventilation system for drying the floors or preparing bread for sale at the right time using the oven right after the dough has raised. Without knowledge of such activities, it is difficult to improve the energy efficiency of the loads operation for sustainability and cost reasons.

The increasing availability of IoT sensors and devices promise to deliver rich data about human activities and control of loads. This includes occupancy sensors [96] and new IoT devices, such as the Amazon suite of IoT products with Alexa and IoT Buttons as concrete examples. Figure 10.1 illustrates a store with various loads and IoT devices. The concept of software-defined buildings [27] aims to improve the operation of buildings by providing an information platform for the creation of efficient and human-centered building systems. A core challenge within this vision is the creation of portable software applications to scale deployments to large parts of the building stock. To enable portable software applications for buildings the community has proposed several types of BOSs sandboxing applications from the particular instrumentation of a building. However, existing proposals for BOSs that should combine such data and control opportunities do not provide concepts and support for activity data including Building Operating System Services (BOSS) [27] and eXtensible Building Operating System (XBOS) [28]. Within the pervasive computing community researchers have considered human activities for context-awareness and activity-based computing [76, 77]. However, these systems do not consider the integration with the building instrumentation and building portability for applications as considered by BOSs.

This paper proposes an ATS for BOSs. The purpose of ATS is to enable developers to write applications that respond programmatically to the activities occurring inside buildings and thereby allow better and more efficient control of the energy consumption as illustrated in Figure 10.1. One example of such control is Demand Response

	Intended Deployment Area	Support Building Infrastructure	Horizontal Scalability	Service Extendability	Application Functionality	Activity Support
CARL [77]	Residential	–	Single Node	–	–	Label activities
JCAF [76]	Hospitals	–	Single Node, Java	Code Additions	RPC/Java	OO Data Model, Models Activity State
Building Depot 2.0 [29]	Offices	✓	Single Node	Code Additions	REST	–
XBOS + Bosswave [27, 28, 47]	Offices	✓	Cluster of Nodes	Services	Bosswave	–
ATS + XBOS + Bosswave	Retail	✓	Cluster of Nodes	Services, State Machines, Code Additions	Bosswave	Models Activity State

Table 10.1: Comparison of operating systems and activity modeling.

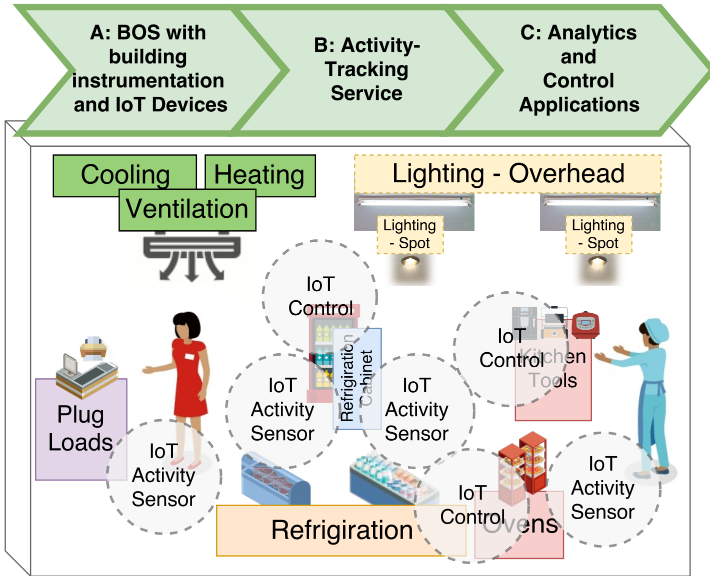

Figure 10.1: The Retail Building and IoT Setting for the ATS.

(DR) [21] where loads respond to grid-side requests. An exploration into the resistance to DR acceptance in retail stores points to worries about the consequences to the consumer experience. Understanding how staff and customers are using the loads of the store help alleviate these concerns, while also enabling developers to better understand how the store is being used. The first step in this understanding of the human activities inside buildings is modeling the activities and making their current state available via Application Programming Interfaces (APIs). Activities could be tracking if a room or area is currently being cleaned, or at what stage of cleaning it is in. Another example could be refrigerators in a store being filled, while the cooling mechanisms optimize when they should cool and how much, based on the state of the cycle. An extra benefit of centralizing the understanding and modeling of activities is the added benefit of allowing multiple applications to track a single activity. Additionally, this also allows the domain experts, the store owners or hardware vendors installing the sensors and actuators, to define the related activities. This design would ensure the models accuracy relative to the store's procedures, as this is where the understanding and expertise in relation to the activities in the stores are located. The service is designed to consider the security,

privacy, integration, extendability and scalability challenges in the building setting.

We provide initial findings for testing the ATS in a proof of concept evaluation. This proof of concept is enabled using a set of common IoT sensors and devices together with the ATS. To illustrate how an application can use this information, a sample application is built to utilize the ATS.

10.3 RELATED WORK

Several BOSs, or parts thereof, has been proposed by several sources like BOSS [27], XBOS [28], BRICK [73], Bosswave [47], Building Depot 2.0 [29] and more. Common for them all is the fact they do not consider modeling of the activities happening inside of the building itself. Attempts have been made to model activities within context-aware computing and activity-based computing before as, for instance, manifested in the systems JCAF [76] and CARL [77]. To get a better overview, Table 10.1 details the differences between these systems.

Building Depot 2.0 was developed for office environments and with the goal of being used as a building operating system. Therefore it supports a building's infrastructure as a natural consequence. The system is built as a single node, which hampers the scalability of the system as the only option is to scale vertically, or alternatively set up several autonomous systems. For more extensive systems, such as citywide deployments, Building Depot 2.0 would not perform well. To extend the functionality of Building Depot 2.0, a user would have to write code and recompile. However, it is possible to integrate functionality by using a REST API. In regards to activity modeling, Building Depot 2.0 has nothing to offer. This kind of functionality would have to be custom built.

Just like Building Depot 2.0, XBOS and Bosswave is built for office and teaching environments and has extensive support for the building infrastructure. Compared to Building Depot 2.0, the systems are built for a series of nodes that can be vertically and horizontally scaled, as well as be distributed over a larger geographical area. Instead of requiring code additions into existing code and recompilation, the BOS is built to be extendable by using services, and letting applications and services communicate over the Bosswave syndication bus. Bosswave

will be explained in more detail later in this paper. XBOS and Bosswave do not support the concept of activities and their modeling. Instead, this kind of responsibility is to be implemented by the applications themselves if needed.

Context-aware and activity-based systems CARL and JCAF differs from Building Depot 2.0 and XBOS as they are not actual BOSs. As a consequence, the support for the building infrastructure is not a considered and is therefore not supported by these systems. Scalability wise, both are single node systems and are therefore not suited for horizontal scalability. JCAF is an abbreviation for Java Context Awareness Framework. It is a Java-based framework that aims to provide a *"Java-based context-awareness infrastructure and programming API for creating context-aware computer applications"* [76]. JCAF is intended to be used by other applications that implement context awareness and has been developed and tested in a hospital environment. JCAF, just as the proposed ATS, models the current state of an activity (or context in the JCAF terminology). In contrast to JCAF and ATS, CARL does not label the current state of an activity, but instead only labels certain input as being an activity. Therefore, the activity tracking is not as detailed tracking as JCAF and ATS. In a retail setting, the CARL approach to labeling activities would also be ineffective, as the system would not be able to distinguish customers from employees, and the amount of entities moving around in the building, and being tracked, is considerably larger compared to the residential setting CARL is supposed to operate in. The aims of CARL is also different as it aims to change the behavior of occupants, where ATS is aimed at providing services for application on top, as well as to enable more loads to operated efficiently and participate in DR events. Also, CARL is built to infer the current activities of the occupants based on sensor data, but currently it is not acceptable for the retail store stakeholders to guess the activities current state, as a potential DR event could have significant consequences for sales or wares if an activity is not tracked correctly. The ATS system could potentially support some of the elements seen in CARL, but it is not currently a goal.

The ATS that we propose in this paper, in contrast, seeks to extend the XBOS/Bosswave combo with an additional service to let the operating system model the activities and their states. The service is intended to be used in a retail setting, which is also where the need for such a system arose. However, it might be relevant for other building types

as well. As the ATS is built on top of XBOS and Bosswave, it shares some of the same characteristics such as the support for the building structure, scalability, as well as the Bosswave syndication bus. Where it differs is of cause the main functionality of ATS: support for activities, but also how to extend the functionality of the services. The ATS is extendable in several layers. First, it supports the same kind of service-oriented extendability as its parent system, XBOS, and Bosswave, but also supports dynamic extensions of state machines at runtime, that is dependent on other state machines, hardware, or sensors. Additionally, code additions can be made if the desired transition types do not exist in the system. The code additions would require a recompilation of the service.

10.4 SYSTEM DESIGN PRINCIPLES

An ATS that is a serious addition to a BOS needs to solve and consider several factors. This section presents some of the main design concerns for such a system.

10.4.1 SECURITY AND PRIVACY

As for every IoT based system, security is a significant concern. BOSs are often distributed, and sometimes over several buildings without internal networks between them. These environments and setups can be unpredictable for a system designer that does not know where the service needs to be deployed. Therefore, all communication to and from sensors, actuators, other services, and applications needs be encrypted and include a security model. ATS needs to be able to delegate access to specific or several activities based on the implemented security scheme.

10.4.2 INTEROPERABILITY

To minimize friction with developers, and to maximize interoperability, an ATS needs to have a single common communication interface for applications that will interact with the system. The concepts for interacting with an ATS should be presented using commonly used patterns developers understand, as to make sure adoption is as frictionless as possible.

10.4.3 EXTENDABILITY

The functionality of an ATS should not be limited to specific hardware or implementations. Therefore, the software needs to be extendable by developers as to make sure specialized solutions can be integrated into the service. Also, applications in an application layer should be able to create custom activity tracking in an ATS and monitor their status.

10.4.4 SCALABILITY

As BOSs can encompass everything from a small apartment to entire cities, an ATS needs to be scalable. Considerations need to be made to ensure high performance is still possible to achieve at a later stage of development, even if the current focus is one or more buildings per service.

10.5 SYSTEM DESIGN AND COMPONENTS

The ATS is designed to fit into a system based on BOSS, XBOS and Bosswave [27, 28, 47]. The following paragraphs and figures addresses the system design principles and architecture for the ATS.

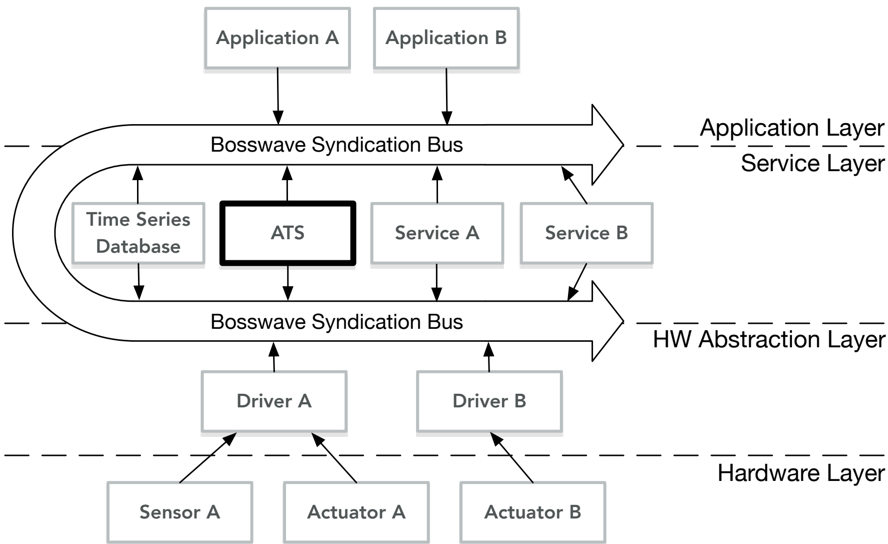

Figure 10.2: ATS Placement in a Building Operating System Context.

Figure 10.2 shows where the ATS is placed in relation to the other components in the building operating system. At the bottom sensors and actuators are adapted to Bosswave using a driver. A driver is a small piece of software that reads the data from the sensors and publishes them into Bosswave in a uniform format, while also listening for requests to change the state of the actuators. The driver also has the responsibility of publishing the metadata that describes where a sensor or actuator is located, and what kind of actuations it can perform. The metadata should follow a standardized format to enable integration of different types of sensors and devices, e.g., following the recently proposed metadata schema for building infrastructures named BRICK [73]. In the service layer several services are placed, a time series database, ATS itself, and other services. These services communicate their respective information or processed data back into Bosswave. From from this point on applications take over and choose how to act on the information provided by the services, or possibly even the drivers.

Bosswave [47] was chosen to solve the security and privacy considerations. Bosswave is an encrypted network based on cryptocurrency known from Bitcoin [51] and Ethereum [52]. Bosswave itself uses Ethereum as its foundation for a public ledger and to control and enforce usage rights and privacy in the Bosswave Syndication Bus (Fig. 10.2). In this network namespaces are created, much like the domains, we know today from the Internet. These domains are paid for by cryptocurrency mined for this specific network, and then bound to an encryption key, much like cryptocurrency is bound to a key. Finding resources is done by Uniform Resource Identifiers (URIs) (e.g. "domain.demo/some/path"). These URIs are read/write protected to anyone not possessing the key owning the name space. It is, however, possible to delegate parts of a URI, for read or write, to another key. This action is called a Declaration of Trust (DoT), that effectively ensures URIs have granulated privacy, and at the same time ensures only keys with the right keys have access to the encrypted data at a given URI.

The URIs of Bosswave also solves part of the interoperability considerations, as it is exposing a common way to interact with all information in the BOS. To unify this further, ATS uses JavaScript Object Notation (JSON) to communicate information between services and applications. JSON is universally easy to read by most programming languages and

technologies.

As for scalability, several factors are important. First, choosing a language that is high performing even though potentially thousands of threads could be running. Second, a database that is scalable enough to handle the potential pressure from several buildings and activities being created deleted, and edited. Regarding the first parameter, language choice, the choice fell on Golang for several reasons. The primary reason is the fact Golang has been built for concurrency and speed. The language can handle several hundreds of threads without problems and has built-in performance testing and concurrency testing. The second reason to choose this language over other high performing languages is the fact that Bosswave is built in Golang and provides pre-built APIs for this language. APIs for other languages are available too, like Java and Python, but they are not kept up to date in the same manner. Bosswave is an integral component of the system, and it makes sense to build ATS on the same technologies, as long as they perform exceptionally well too. The second factor mentioned above was the database. For this prototype, the database speed is not important as it is going to be used in a small setting with few activities. Therefore, SQLite was chosen for simplicity, and to make it easy to move from server to server for testing purposes. Be aware a large scale system would need to consider the database choice. The database load is only high during startup and shutdown of the service, as new activities seldom would be created and deleted in large batches. Database access speed is only important at these times as state history of the activities are maintained by the Time Series Database, as depicted in Fig. 10.2.

For the purposes of this paper, an activity is defined as the set of actions and stages that can describe what a person or other entity do to reach a goal. The activities that we would like to model typically involving appliances consuming energy. Examples could be a person in a store filling up a refrigerator with goods, or cleaning personal cleaning the floor of a particular area. To model the activities, an adapted version of the commonly known state machine design pattern was chosen. The reason for this choice was a wish for developers to easily recognize the patterns used in development. Also, state machines naturally describe activities, as an activity that needs to be described to a system will always be in a state and that state changes based on input from several sources. While it may be possible an activity could have several states at the same time, it was not taken into consideration

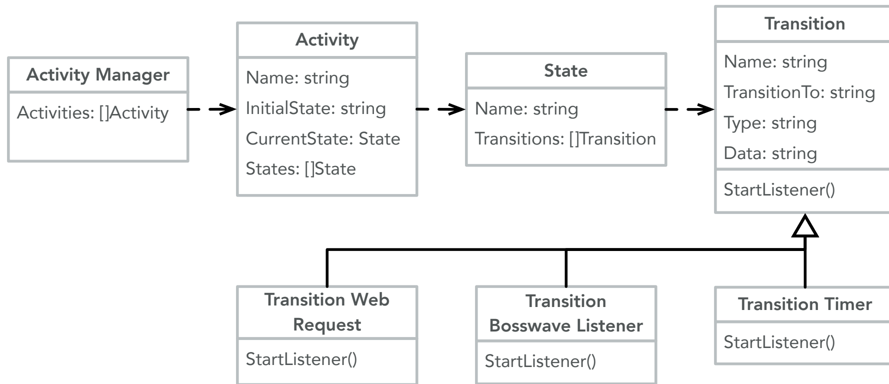

Figure 10.3: Simplified ATS architecture describing the modified State Machine implementation.

for this paper. If multiple active states are needed for a given activity, multiple activities would have to be created in the ATS. One of the other solutions mentioned earlier, CARL [77], is able to label several activities at once from a mass of sensor data, but it is not able to track the state of a given activity. JCAF [76] on the other hand is able to track the state of an activity but, just as CARL, is missing the scalability and service-oriented approach to its architecture.

A simplified version of the central parts of the ATS can be found in fig. 10.3. The *Activity Manager* has the responsibility to orchestrate the activities. The activities themselves are described using JSON too, and sent to the *Activity Manager* that creates and starts the *Activity*. *Activity* is the signifier for a single activity. It holds the current state, as well as all possible states. *State* works similarly as *Activity*, but holds an array of transitions to other states. *Transition* is the part that makes ATS extendable. Its subclasses overrides the *StartListener*() method, and implements their specific functionality. *StartListener*() creates a goroutine (similar to a thread), that waits for the implementation to react to something, and then sends the request to change state into a channel that takes care of the actual state change. The *Data* field in *Transition* is where subclasses are able to draw their specific configurations from. This configuration is preferably stored as JSON and should define the functionality of a given transition. The "Transition Bosswave Lister" for example would need at least a URI to listen to, a key to use for access, and the expected message to listen for. The main difference

from a standard state machine is the fact that the machine is dynamically defined using JSON, and extended using custom transitions. It is important to note that an activity state can be built to depend on the state of another activity. In this way, more intricate and interesting logics can be created.

10.6 EVALUATION

To test the ATS, the following setup, as seen in Fig. 10.4 and two activities were created in the ATS. The purpose is to create an example of the possible functionality as simple as possible that still illustrates the complexity of the problem. The example takes a small retail shop as the setting. The first activity lets a sale person in the shop register if they are present or leaving, while the second activity lets cleaning personal register if they are currently cleaning the room or if they finished up. A light regulator application then sets the lighting to 60% if the room is occupied, and 0% if the occupant is not there. If the room is currently being cleaned, the light will automatically increase to 100%, overriding the occupants setting, to increase visibility.

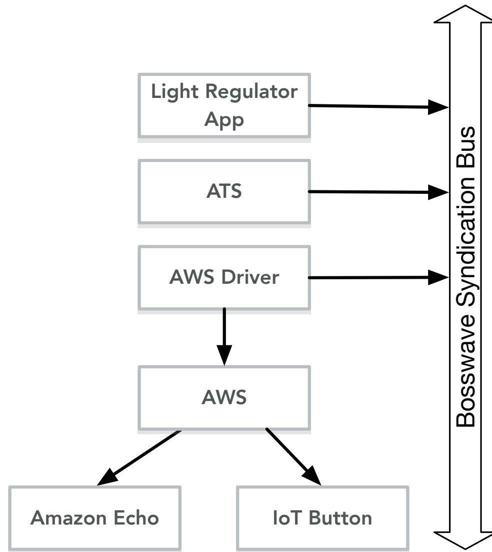

Figure 10.4: Evaluation Setup.

For the hardware layer, Amazon IoT Buttons were used, as well as

an Amazon Echo. One click on the IoT Button would register cleaning as starting, and two consecutive clicks would send a cleaning finished signal. The Amazon Echo, on the other hand, was configured to react to both requests for cleaning state changes, as well as occupancy state changes. Additionally, a user would verbally be able to request the current states of the two activities. Both the IoT Buttons and the Amazon Echo device runs functions placed in the Amazon Web Services (AWS) Cloud. These functions relay the requests to the AWS Driver that in turn restructures the request into JSON and publishes the state change request into Bosswave. The ATS then picks up the message, determines if the change is valid, and notifies subscribing applications through Bosswave.

The prototype was tested in an office setting over a week. Both the Amazon echo and the IoT buttons where tested by multiple people in the given time frame. Each person had a short verbal introduction to using the hardware. Testers where told the Amazon Echo could change states of both activities, and given phrases to use to change these activities, as well as to get the current state of the activities. The button's single click and double click functionality were demonstrated too.

Usage of Amazon Echo yielded mixed results. A few were able to use it to change states of the activities without training and flaws, while a few kept changing the wrong activity state. Up to 50% of the actuations from these persons resulted in a wrong state change being triggered. The added functionality of the Echo, to get the current status of the activities, was beneficial for clearing up the confusion from these flaws. These mis-actuations was later diagnosed to most likely be caused by the voice to text interface trying to match a given command to the pre-programmed possible phrases. Better defined phrases where keywords that are further from each other would alleviate some of the problems.

The IoT buttons, on the other hand, showed significantly better reliability and ease of use. Only a single person had problems understanding the double click functionality, which was solved by a practical demonstration. An alternative if a user has problems using the double click functionality, for example, due to medical conditions, could be to utilize the long press instead, as this press requires very little dexterity from the hands. The only two complaints about the IoT buttons was mentioned. First, that buttons did not give much feedback if the press was actually registered, or if the right command was registered, other than a

small light that would always light green when the command was sent. Second, the button took up to 4 seconds registering the change. If a miss-press happened, a user would have to wait for the button to finish its communication with the server, as the button does not implement queue functionality.

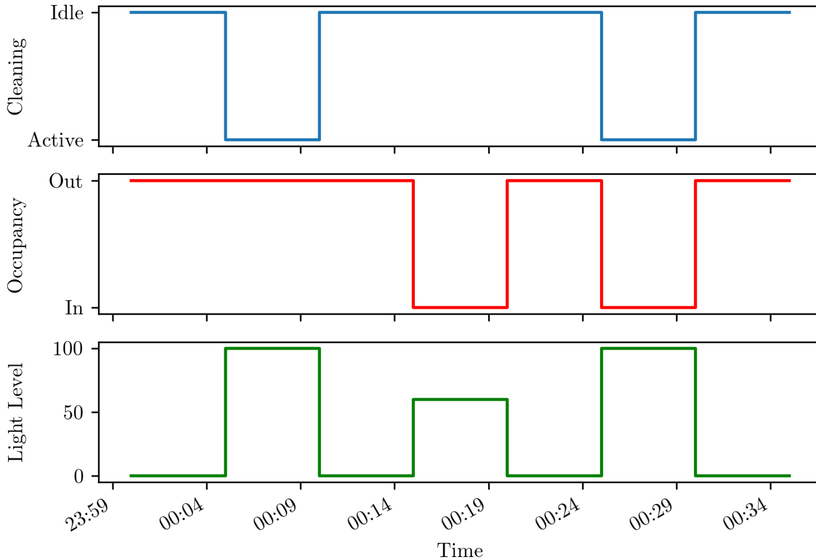

Figure 10.5: Testing Results.

Figure 10.5 details the outcome of one of the testing occurrences. As can be seen, in this case, the system behaved as expected. If the Cleaning activity changes to *active*, the light level rises to 100%, and if the Cleaning activity is *idle*, and the Occupancy activity is *In*, the light level is set to 60%. Finally, if the activities are set to *in* and *active*, the cleaning activity supersedes occupancy activity and sets the lighting to 100%.

During testing of the ATS, a significant flaw became apparent. If one activity for some reason crashed, the entires ATS crashed. Activities code should be separate from each other, and should also implement a solution to recover the current state of an activity if it crashed. Architecture wise an activity should be considered to be handled more like a separate service that can be stopped, started, restarted and recovered.

Complexity-wise, the ATS is relatively simple and easy to understand.

The integral parts are made up of less than 700 lines of code, including three different implementations of a transition. The three transitions are on average 90 lines of code, where on average 25 lines of code is the part that a developer needs to implement to add a new transition. The few lines of code in custom transitions ensure the ATS is easy to extend functionality wise.

10.7 DISCUSSION

Apart from the need of separation between activities if a crash occurs, the ATS performed as expected, as long as the Amazon Echo behaved as intended. The separation is necessary to fix, as several hundred, or more, activities could modeled in the ATS, and therefore a multitude of custom transitions could be responsible for crashes. Another useful addition would be to allow the developers to dynamically post transition functionality into the service, instead of creating a custom implementation of Transition. This addition would reduce the coupling between applications and the ATS, and limit the developer's interaction with the ATS code.

The choice of using state machines as the structure of the ATS was found to potentially come with certain drawbacks. A complex state machine could potentially get lost in a state if the transition failed to notice a change. If this happens, a state machine could get stuck and only move out of this state again once the specific state would be hit again. One way of avoiding this could be introducing a failsafe mechanism developers can implement. This failsafe would allow the activity to reset to a specific state if a particular set of conditions are met. In certain use cases, it could also be beneficial to allow for more flexible state machines that introduce the possibility of estimating what state is currently the most likely to be active out of a given set. Currently, this kind of flexibility is not possible in the implemented rigid understanding of what a state machine is. An added benefit of this could be allowing an activity to be in several states at once if this use case is needed. In the case of energy analytics involving states, a rigid state machine could also prove unpractical.

Amazon Echo presented several complications, but would most likely not be implemented in retail stores. As a result, the problems detected with this actuation method would most properly not happen in an

actual setup. IoT buttons proved to be reliable, and easy to understand and use. It is reasonable the same conclusion would be drawn in a retail setting, as long as clear guidelines are set. Other sources of control should be considered, such as point of sale systems, scraping calendars for information and more.

As part of the ATS development and its testing, it became apparent there could be value in exposing other kinds of information than just the activities. Alternative information like opening hours, area/store type, and more, could be useful for developers creating applications on top of the BOS. Further study into what kind of information that could be valuable should be performed. If valuable information categories are found, the ATS should be expanded to incorporate these new types of information. Information like this would require being stored and structured semantically in a way that is easy to maintain for those that have access to the information, for example, store owners, as application developers would most likely not have this kind of information at hand if they are creating a generalized application.

10.8 CONCLUSION

In this paper, we presented an ATS designed for a BOS setting. In conclusion, the designed and implemented ATS performed mostly as expected helped give additional insight into how activities could be modeled, to help leverage hardware entangled in activities for energy savings and DR purposes.

ACKNOWLEDGMENT

This work is supported by Energinet.dk ForskEL for the project FlexReStore (12413).

REFERENCES

[2] Jakob Hviid and Mikkel Baun Kjærgaard. 'Activity-Tracking Service for Building Operating Systems'. In: *2018 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. IEEE. 2018,

pp. 854–859. DOI: 10.1109/PERCOMW.2018.8480362.
Published.

[21] Naoya Motegi, Mary Ann Piette, David S Watson, Sila Kiliccote and Peng Xu. 'Introduction to commercial building control strategies and techniques for demand response'. In: *Lawrence Berkeley National Laboratory LBNL-59975* (2007).

[27] Stephen Dawson-haggerty, Andrew Krioukov, Jay Taneja, Sagar Karandikar, Gabe Fierro, Nikita Kitaev and David Culler. 'BOSS: building operating system services'. In: *NSDI '13* (2013), pp. 1–15.

[28] Gabriel Fierro and David E Culler. 'XBOS: An Extensible Building Operating System'. In: *Proceedings of the 2nd ACM International Conference on Embedded Systems for Energy-Efficient Built Environments*. ACM. 2015, pp. 119–120.

[29] Thomas Weng, Anthony Nwokafor and Yuvraj Agarwal. 'Buildingdepot 2.0: An integrated management system for building analysis and control'. In: *BuildSys'13*. ACM. 2013.

[47] Michael P Anderson, John Kolb, Kaifei Chen, David E Culler, Randy Katz, Michael Andersen, John Kolb, Kaifei Chen, David E Culler and Randy Katz. 'Democratizing Authority in the Built Environment'. In: *BuildSys*. 2017.

[51] Satoshi Nakamoto. *Bitcoin: A peer-to-peer electronic cash system*. 2008.

[52] Gavin Wood. 'Ethereum: A secure decentralised generalised transaction ledger'. In: *Ethereum Project Yellow Paper* 151 (2014).

[73] Bharathan Balaji, Arka Bhattacharya, Gabriel Fierro, Jingkun Gao, Joshua Gluck, Dezhi Hong, Aslak Johansen, Jason Koh, Joern Ploennigs, Yuvraj Agarwal, Mario Berges, David Culler, Rajesh Gupta, Mikkel Baun Kjærgaard, Mani Srivastava and Kamin Whitehouse. 'Brick: Towards a Unified Metadata Schema For Buildings'. In: *BuildSys*. 2016. DOI: 10.1145/2993422.2993577.

[76] Jakob E Bardram et al. 'The Java Context Awareness Framework (JCAF)-A Service Infrastructure and Programming Framework for Context-Aware Applications.' In: *Pervasive*. Vol. 3468. Springer. 2005, pp. 98–115.

[77] Brian L Thomas and Diane J Cook. 'Activity-Aware Energy-Efficient Automation of Smart Buildings'. In: *Energies* 9.8 (2016), p. 624.

[96] Mikkel Baun Kjærgaard and Fisayo Caleb Sangogboye. 'Categorization framework and survey of occupancy sensing systems'. In: *PMC* 38 (2017), pp. 1–13.

CHAPTER 11

SERVICE ABSTRACTION LAYER FOR BUILDING OPERATING SYSTEMS: ENABLING PORTABLE APPLICATIONS AND IMPROVING SYSTEM RESILIENCE

This chapter is a cosmetic adaptation of the following conference paper: Jakob Hviid and Mikkel Baun Kjærgaard. 'Service Abstraction Layer for Building Operating Systems: Enabling portable applications and improving system resilience'. In: *2018 IEEE International Conference on Communications, Control, and Computing Technologies for Smart Grids (SmartGridComm)*. IEEE. Aalborg, Denmark, Oct. 2018, pp. 1–6. DOI: 10.1109/SmartGridComm.2018.8587543. **Published**.

The paper was presented at the 2018 IEEE International Conference on Communications, Control, and Computing Technologies for Smart Grids, in Aalborg, Denmark, on 28 October 2018.

11.1 ABSTRACT

For large scale implementation of Demand Response (DR) programs, applications enabling DR would need to be portable between buildings. Building Operating Systems (BOSs) are an essential strategic piece for enabling portable applications, which work with several hardware implementations from different vendors. These BOSs can effectively be used to implement large-scale DR implementations. Currently, though, the service layer of BOSs requires direct integration with specific services, and thereby have requirements for specific implementations of services to be present in a specific building. This paper proposes the introduction of a Service Abstraction Layer (SAL) to decouple the application from the specific implementations of services, as well as to introduce the concept of redundancy in service responsibility areas.

These changes would allow for application portability between buildings but also allow for BOS resiliency.

A prototype abstraction layer is implemented and tested. Results show the introduction of a SAL has promising benefits for BOSs, and successfully decouples the applications from the BOS implementation, as well as improving system reliability and resiliency. For smart grid applications, the addition of a service abstraction layer allows for large-scale portable applications, but would still require some form of standardization of communication protocols for different service types.

11.2 INTRODUCTION

Handling the introduction of large-scale green energy production is difficult. Not only because of the logistics of transforming a grids power production, but mostly because producing energy at the right time, and when it is needed is a daunting task when the production is at the mercy of the wind and sun. At the same time storing the excess energy produced is expensive, and comes at the cost of energy loss, as a consequence of energy conversion. DR technologies are one of the solutions aiming to mitigate some of these issues [19]. In California, US [18], the technology is applied to address peak-loads and to help ensure a stable energy grid too. DR helps redistribute loads and encourage customers to use energy at the most favorable time, which naturally is when the energy grid is not under heavy load, and the opposite when it is. In Denmark, the retail sector uses a considerable amount of the energy consumed today. About 8% of the industry sectors energy usage, about 9.344 TJ in 2013 [25], is consumed by retail stores, which makes them a compelling candidate for DR integration.

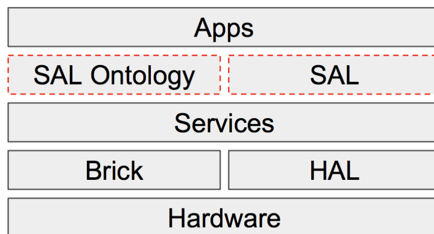

Figure 11.1: BOS Layer with added SAL.

BOSs is one of the promising methods of supporting DR. The reason for this stems from the fact that BOSs can abstract the applications, needed for DR, from the hardware implementation from different vendors. Especially older buildings and building complexes that need to work together as one have different hardware specifications and implementations from each other. For large scale application distribution, applications need to run flawlessly regardless of hardware or service implementations. As a BOSs integrates with every piece of hardware in a building, this is the optimum placement for DR decision making or aggregation. The Distribution System Operator (DSO) would in this context supply an application that integrates into their systems, to either provide price signals or direct control of the BOS, via protocols, such as, OpenADR [40]. The DR application can in other situations be supplied by another developer or stakeholder. To make this kind of DR application, and any other, successfully integrate into large-scale DR initiatives, the application would need to be easily portable between buildings. To support this, Brick [73, 74] was introduced, as a possible contender to abstract the applications from the hardware of the buildings. However, this kind of abstraction has not been introduced for the BOS services.

Fig. 11.1 shows a layered architecture for a BOS using Brick. The red dotted lines signify the main contribution of this paper, the addition of a SAL to enable actual portability of applications, as well as enabling a more resilient building architecture. Currently the services in BOSs have had hard-coded service integration with specific services. Services are currently simply not abstracted from the application implementation in the same way the hardware is abstracted from the applications by using Brick. The resiliency addition is manifested by the possibility of having several services exposing the same type of information, possibly from different sources. An example could be outside weather forecasts that are exposed using a service, but the external service goes down for maintenance. If this would happen, an additional service that integrates with a different provider can be installed, to ensure the system could be running at maximum efficiency, even through downtime periods. As for the portability, an example could be two buildings with different services but exposing the same kind of information. Here installing the same codebase in both buildings will result in an application running as intended without any code or configuration changes. These changes enable large-scale deployment of

applications across different types of buildings with the same codebase and allow for developers to make standard products, instead of custom implementations for each customer.

This paper introduces the idea of a SAL, as well as discusses the benefits and drawbacks of such an addition. A prototype system is implemented to verify the concept. The concept is presented in the following sections.

11.3 RELATED WORK

Several contributions into the field of BOSs have been made, but most notable for this contribution is Building Operating System Services (BOSS) [27], SMAP [46], eXtensible Building Operating System (XBOS) [28], and Bosswave [47]. These papers explain and expand on the idea of BOSs. The first three, BOSS, XBOS and SMAP, contributes to the core understanding, but are confined to typically local networks or controlled networks. Bosswave, on the other hand, transforms the idea of, and the concept of, BOSs to a potentially globally distributed and cryptographically secured operating system built on top of the blockchain and based on microservices. Microservices [97] is a software architecture technique that is based on the service-oriented architecture [98] (SOA). It encourages fine-grained services, hence the micro, and lightweight protocols. A microservice is expected only to solve one problem, and typically communicates over a service bus. The blockchain is the technology typically known from several types of digital currency [51, 52]. Bosswave itself is based on Etherium codebase for the non-data transactional parts of the system but secures data transmissions with the keys also used in the blockchain. Some contributions to BOSs, concerning DR have been made, for example, Hviid and Kjærgaard's [2], which introduces a service to track activities of occupants, to allow for better DR decision making. Other examples can also be found, such as Peffer et al. [22], and Weng et al. [99].

Other contributions to BOSs have been made, such as OMEGA 2.0 [100] and EF-Pi [101]. OMEGA 2.0 still sees the world as being the operating system and the applications on top and neglects to utilize a service ecosystem. Also, OMEGA 2.0 is a Java based system and not a distributed system, and as such implements interfaces for abstraction. This choice of architecture forces the system to run on a single

machine whereas XBOS and Bosswave are distributed systems, where abstractions are needed to overcome the abstraction problem. A SAL in the context of this paper makes little sense in a monolithic java based application setup, where Java based interfaces partly solve this problem. EF-Pi is interesting as it works specifically with the DR space, but the system does not seem to have other purposes than DR, which can make it a difficult value proposition for consumers. This focus also means running generic applications on top of EF-Pi is not the primary function. As such a SAL seems to be less relevant in this case. Both systems can however be easily abstracted into hardware or services in the context of XBOS and Bosswave.

In regards to the concept of portable applications for BOSs, several contributions have been made. Metafier [63] and Brick [73, 74] are two of these additions. Brick itself is especially interesting as it signifies a component that the contribution of this paper builds on. Brick is an ontology by which a model can be built that signifies the hardware setup of a specific building. This ontology enables an application to discover hardware dynamically using a SPARQL Protocol and RDF Query Language (SPARQL) [69] query through a service like HodDB [48] instead of being hardcoded to the specific implementation of a building. However, Brick fails to encompass all of the aspects discovered by Bhattacharya et al. [102], which also points to several other factors not currently covered by Brick, such as services and people.

Other types of operating systems have also faced the problems related to application portability. Typically this is solved through operating system Application Programming Interfaces (APIs), but this approach limits the application to function only on the intended operating system. Other approaches are seen in programming languages, and their attempt to solve this by adding a layer between the operating system and the application. Most notably the Java JVM [103] does this, as well as other languages such as Python and Perl that requires the user to install an interpreter. A reverse example is Microsoft's C#, which seeks to abstract the application implementation language, allowing for several languages to be used for a single application, from the interpreter by introducing the MSIL (Microsoft Intermediate Language) in between the application and operating system. Common for all of these solutions is the introduction of an abstraction layer in between the application layer, and the operating system below.

The following section will detail the concept of the SAL, as well as

give examples of possible usage. Below a methodology is described, as well as the evaluation setup and results.

11.4 CONCEPT

Other BOSs, as mentioned earlier, fail to consider the benefits of introducing a SAL, application portability, and resilience. Fig. 11.1 shows how a traditional BOS layered architecture would be constructed but includes the SAL, as well as the SAL Ontology, marked with the red dotted lines. The SAL is the layer itself, where the SAL Ontology is the counterpart to Brick, but for services, and finally, the SAL Model is the specific implementation of the ontology describing the relation between entities. Again, Brick seeks to abstract hardware from the application, while the SAL model seeks to abstract services from the application. The application will query Brick every time it needs to access hardware and will be provided with URIs to the data streams. Similarly, the application would need to query the SAL model if it needs access to services, which will also result in Uniform Resource Identifiers (URIs) to the specific service endpoints, allowing the application to choose which endpoint to contact.

A concrete example of the usage of a SAL would be the following scenario: An application that plans the assignment of rooms for lectures needs to plan the best distribution of classes. First the application needs to know the lecture rooms that exist in the building. This is achieved by querying the Hardware Abstraction Layer (HAL), where a list is returned with all relevant rooms. Instead of reading directly from the hardware and trying to interpret the estimated usage for each room itself, it queries the SAL Model for a service providing occupancy counts for the specific rooms instead. The SAL returns a list of services providing this service, and the application contacts the first service on the list for the information needed. This allows the application to write code that is not specifically written for the specific building, and allows the application to be re-used on top of multiple buildings with other layouts and other services.

Introducing the SAL allows for decoupling and flexibility in the following scenarios:

1 The microservice Weather Forecast is changed to a service from another provider. The service interface might be different, located

on another server, and publish data to a different part of the Bosswave Syndication Bus, but the DR Application keeps working normally as the SAL Model is updated, and SPARQL queries to the model refer to the new location.

2 Removing a service that is depended on by an application, and no substitute is found, will result in an error message telling the technician which services types are missing.

3 Adding a service that delivers information that is already provided by another service, would allow the application to choose which one has the best fitting parameters. This change could allow the application to automatically switch between services, or use the better data source for a specific problem.

4 Building A has one set of services, and building B has another. With the SAL Model, the application could be moved between the two without code changes, as long as the building provides the service types on which the application depends.

5 A building is expanded upon and ends up with two different sets of services for the old section of the building and the new. The application seamlessly interacts with both zones and their set of services using the SAL Model, without code changes.

6 A building with several companies co-located could end up with the need for several extra applications or different services per. company depending on the needs. Again, the application allows for this using the same codebase as the SAL model abstracts from every given service.

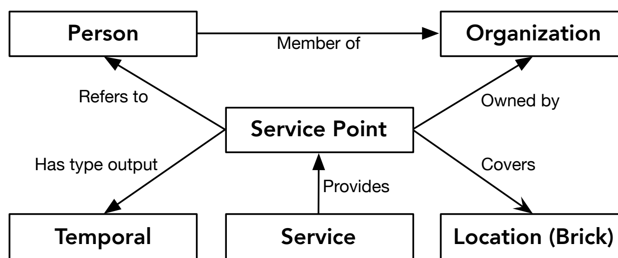

Figure 11.2: SAL Class Relationships.

The SAL layer consists of a SAL Ontology and a SAL model. The ontology itself describes classes and their relationship with each other, while the model describes the actual services in a building implementa-

tion. In simplified programming terminology, the ontology is the class, and the model is the instance of that class. The ontology itself is built from five class, and a single class defined in Brick. Fig. 11.2 details the relationship between the classes. The first class is named *Service*, which is a representation of the actual service providing one to many *Service Points*. A *Service Point* represents the URI or actual endpoint in which to fetch or send data to interact with the service. This *Service Point* can be associated with the concept of a *Location*, and thereby enables the service endpoint to be associated to a specific location in a building. *Location* is a concept directly imported from Brick, and introduces several subclasses such as *Room*, what will be used in the examples later. The reason Brick's concept of *Location* and *Room* is reused, is to ensure concepts are not duplicated between several descriptions. Additionally, using the same concept makes the developer aware of how it is constructed, as well as allowing for eventual easier integration into Brick if such is desired in the future. When working with services, the output of a service is often related to predictions or other temporal parameters. Therefore, the *Service Point* has an association to the *Temporal* class. Occupancy prediction services such as OccuRE [80], the context of which types of people that are tracked can be significant. To accommodate this, the concept of *Person* is introduced and associated to the *Service Point*. Last, but not least, the concept of ownership and the owning organizations are introduced. *Organization* has two relationships, one for ownership that is associated with the *Service Point*, and another associated with *Person* signifying the membership of an organization.

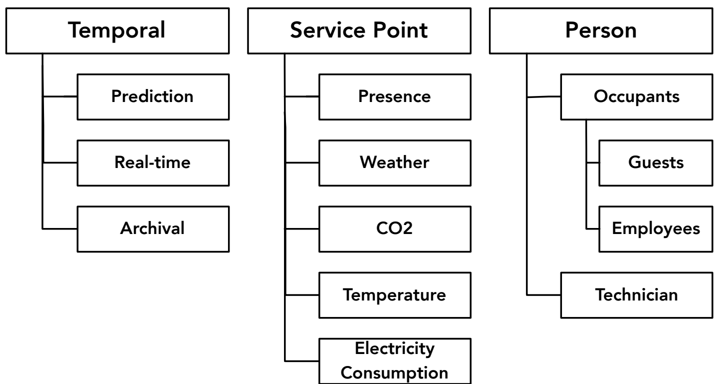

Figure 11.3: SAL Class Inheritance.

Each of the classes mentioned before can have subclasses associated with them. A subclass indicates a more specific type of the superclass. Fig. 11.3 details the subclasses identified to be essential for the examples in this paper but is not complete or exhaustive. *Service* is not present as no subclasses where currently identified, and *Location* is part of Brick. *Location* is a superclass, with several subclasses, such as Building, Room, Floor, and more. Please refer to the Brick papers [73, 74] for more detail on the *Location* class. The *Temporal* class has several subclasses that are supposed to specify what temporal phase the *Service Point* is describing. Current subclasses are Prediction, Real-time, and Archival. The *Service Point* is one of the hardest classes to create a complete ontology for, and one of the most important for successful large-scale implementation, as the subclasses, specify the actual context of the exposed data. For this paper, and the examples used herein, the following subclasses were defined: Presence, Weather, CO_2, Temperature, and Electricity Consumption. All the possible subclasses will effectively be impossible to exhaustively add up front, as all possible service applications are not known, but fortunately, the success and usefulness of the SAL layer do not hinge on the subclasses including every imaginable scenario up front. Finally, the *Person* class has two subclasses. The Occupants subclass signifies occupants are modeled with the *Service Point*, but also adds the further specialization of Guest and Employees. The Technician subclass is not used in any of the examples here but is mainly there to represent further relevant additions.

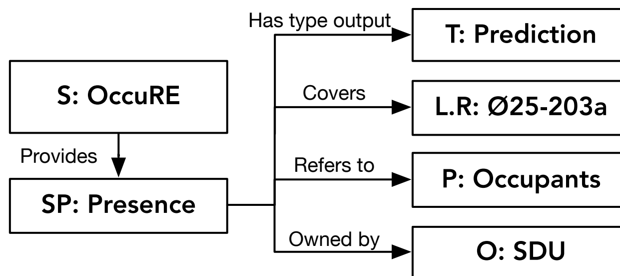

Figure 11.4: SAL Example: OccuRE.

The first example of applying the SAL Ontology into an actual model is detailed in fig. 11.4. Here the partial SAL Model describes the OccuRE service, which is a service built to track and predict occupancy based on historical data. One of the many *Service Points* exposed by

OccuRE is exposed by the *Service Point* subclass; Presence. The Presence subclass refers to a location of which the prediction is made which is a *Location* subclass named Room, as well as specifying that the output is a prediction, by using the relation to the *Temporal* subclass Prediction. The service produces output that tracks Occupants, and all data produced is owned by SDU (The University of Southern Denmark).

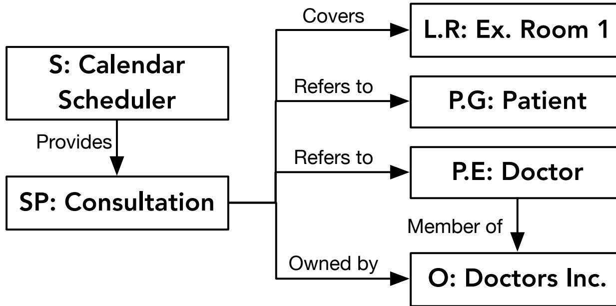

Figure 11.5: SAL Example: Calendar Scheduler.

The second example, is of a Calendar Scheduler in a private practice, which is detailed in fig. 11.5. Here the *Service*, Calendar Scheduler, exposes scheduled consultations for Exam Room 1. The consultation is for patients of the subtype Guest and is associated with a subtype Employee named Doctor. The doctor is employed at the *Organization* Doctors Inc., which also owns all data exposed by the calendar scheduler.

11.5 METHODOLOGY

This paper works with the same case from [20], but moves the entire setup from a SMAP [46] and BOSS[27] based architecture to a setup based on XBOS [28] and Bosswave [47]. The move in architecture shifts it from a traditional layered architecture to a microservice architecture. The main contribution of this paper is the addition of a SAL. The move between the two architectures will be explored in more detail in a later section of this paper.

The technology choices of the SAL, are based on the emerging standard Brick [73, 74]. Brick is the current technology of choice to implement discovery of hardware in the HAL when working with Bosswave.

Brick uses the concept of ontologies, Resource Description Framework (RDF) (Turtle) [62], and SPARQL [69] to decouple the layers above the HAL from the hardware itself. Similarly, the SAL needs to be decoupled from the other services and applications. Creating a new service discovery standard or using a competing service discovery standard would be actively working against the Brick standard which currently seems to be the future successor, and therefore using the same technologies are a logical choice.

The ontology, and model, is built using Protégé [104] and hosted using the Fuseki [49] server. A driver for the Fuseki server is created, effectively integrating the SAL model with Bosswave. All components, like services, the DR application, and the drivers and services are built using the Golang language, and using the official Bosswave bindings [105] (or API) for this language. The services created only expose sample data, as the functionality of the SAL layer is the interesting concept and not the computation. The same is the case for the DR application, that uses the SAL model, as well as the services, but does not transmit actual decisions to the DSOs mediator.

The architecture and SAL Model is in this iteration built based on the expectation that a technician will be setting up the environment and making changes to the SAL Model. Several additional complications would need to be solved before a system like this would be completely plug and play.

11.6 EVALUATION

To verify the architecture and SAL concepts, a prototype system is implemented, using the case system described by Nelleman et al. [20]. The case describes a state of the art DR system built on a BOS, that integrates DR decisions with model predictive control, weather data, and occupancy prediction. The goal of the case system was to use model predictive control to deliver comfort compliance, so make sure the building keeps a certain temperature standard, while also integrating with a DR service. The system architecture in this paper, has moved from the case's BOS to an XBOS and Bosswave based system. In the instance of this prototype, the SAL is implemented as a microservice and is separate from the Brick Model. Services not needed are removed from the prototype setup, and some micro-services are shell services

serving sample data from the original setup. The external mediator application and the DR Application are connected to an actual system but are a sample application built to simulate the same setup as the original. An ontology is created to serve as the basis for the SAL Model. The simulation setup will run through the following scenarios for verification:

1. **Application Portability:** Move the application between two buildings with two different sets of services for the same purposes. **Expected result:** No code changes needed.
2. **System Resilience:** Two services expose the same functionality. Remove the current service being used. **Expected result:** Application keeps running.

11.7 RESULTS

The architecture in fig. 11.6 is an adaptation of the system discussed by Nelleman et al. [20]. The system in the figure is based on XBOS [28, 27], and Bosswave [47]. The first layer from the left HW (Hardware), contains any number of sensors or actuators. These sensors and actuators are then homogenized by the use of any number of drivers located in the HAL and then published in the Bosswave Syndication Bus. The Brick Model, also located in the HAL, is functionally a semantic representation of the hardware, which allows the service or application to query for the URIs they need to access the data provided by the drivers in the Bosswave Syndication Bus. The combination of the drivers and the Brick Model is what enables the rest of the architecture to work on different buildings without much adaptation. The service layer contains several microservices that read data from the HAL or external services and then publishes them into the Bosswave Syndication Bus. The SAL contains the main counterpart of the Brick Model for the services, which enables applications, and other services, to discover what service types are provided, and if they deliver the kind of information needed for the application to function as intended. Further up the layers, the application layer can be found. This layer is the final part of the BOS and contains the DR application that communicates with the mediator that in turn communicates with the DSO using the OpenADR [40] protocol.

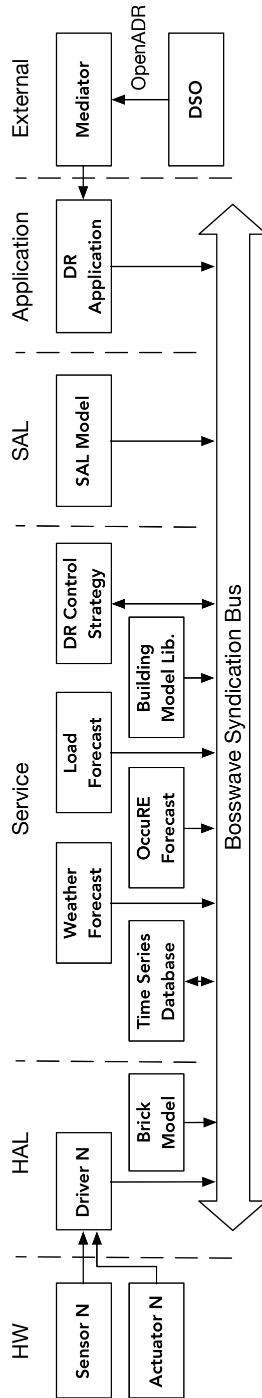

Figure 11.6: Architecture moved to XBOS and Bosswave.

Working in the context of XBOS, BOSS, and Bosswave, the architecture has the benefits of increased security by encryption, and significantly lowering coupling between all parts of the system. This lower coupling is achieved as drivers, services and applications all integrate with the Bosswave Syndication Bus, and the Brick (exposed by HodDB [48]) and SAL Model, and not the specific implementations of each of these types. HodDB is a query processor for Brick which is designed to fit within the Bosswave and XBOS ecosystem.

The expectation for verifying the architecture that the operating system could be made more resilient by adding several services that supported the same type of output the application on top would need. The tests done in the system above supports worked as intended. The other expectation was that an application could be moved between two separate buildings or SAL Models, and still function as intended. This application move was verified to function as intended. However, the following problem was found during verification of the architecture. If the service is newly introduced to the system, and therefore the SAL Model, the application would need to be restarted for it to query the SAL again. This restart is something that can be alleviated by forcing a new query to the SAL if a timeout occurs, or by doing it using time intervals. If both services are present in the SAL Model during the boot process, this will not be needed.

11.8 CONCLUSION / DISCUSSION

Based on the verification results above, the addition of a SAL would add tangible benefits to the goal of enabling portable building applications, but also to improving building resiliency, as losing a service is an actual possibility in a distributed system architecture. Applications would no longer be hard coded to the specific service implementations but to dynamic queries that allow for changes in the underlying hardware and service structure of the BOS. Of cause, not hardcoding comes with a cost. Developers would need to learn SPARQL and work with the abstract of a service type, instead of just integrating with one specific service in the first place. Also, a service developer would need to make changes to the SAL model in order for the service to be registered as an active part of the BOS. As Brick already puts these requirements on the developer, it is not seen as a particularly huge investment.

An important note to add is the fact that neither Brick or the SAL Ontology solves the problem of standardized interfaces on a type of service. The datatypes exposed by services tend to be complex, and not just time series as of a primitive type as is typically the case with the data exposed in the HAL. Some description, or standardization, would be needed to be agreed upon to create genuinely portable services and applications. However, this paper does not seek to solve this predicament, and neither does Brick at this point.

Nevertheless, abstracting the services from the applications, as demonstrated in this paper, allows for applications to be considerably less dependent on specific BOS implementations, and thereby allows DR applications a further reach. It will enable DR applications to be deployed on a larger scale without code changes and to operate in a more trustworthy manner at the same time.

ACKNOWLEDGMENT

This work is supported by Energinet.dk ForskEL for the project FlexReStore (12413).

REFERENCES

[2] Jakob Hviid and Mikkel Baun Kjærgaard. 'Activity-Tracking Service for Building Operating Systems'. In: *2018 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. IEEE. 2018, pp. 854–859. DOI: 10.1109/PERCOMW.2018.8480362. **Published**.

[3] Jakob Hviid and Mikkel Baun Kjærgaard. 'Service Abstraction Layer for Building Operating Systems: Enabling portable applications and improving system resilience'. In: *2018 IEEE International Conference on Communications, Control, and Computing Technologies for Smart Grids (SmartGridComm)*. IEEE. Aalborg, Denmark, Oct. 2018, pp. 1–6. DOI: 10.1109/SmartGridComm.2018.8587543. **Published**.

[18] Mary Ann Piette, Sila Kiliccote and Girish Ghatikar. 'Field Experience with and Potential for Multi-time Scale Grid Transactions from Responsive Commercial Buildings'. In: *ACEEE Summer Study on Energy Efficency in Buildings*. 2014.

[19] Mikkel Baun Kjærgaard, Krzysztof Arendt, Anders Clausen, Aslak Johansen, Muhyiddine Jradi, Bo Nørregaard Jørgensen, Peter Nelleman, Fisayo Caleb Sangogboye, Christian T. Veje and Morten Gill Wollsen. 'Demand response in commercial buildings with an Assessable impact on occupant comfort'. In: *SmartGridComm 2016*. 2016, pp. 447–452.

[20] Peter Nellemann, Mikkel Baun Kjærgaard, Emil Holmegaard, Krzysztof Arendt, Aslak Johansen, Fisayo Caleb Sangogboye and Bo Nørregaard Jørgensen. 'Demand Response with Model Predictive Comfort Compliance in an Office Building'. In: *SmartGridComm'17*. IEEE. 2017.

[22] Therese Peffer, David Auslander, Domenico Caramagno, David Culler, Tyler Jones, Andrew Krioukov, Michael Sankur, Jay Taneja, Jason Trager, Sila Kiliccote et al. 'Deep demand response: The case study of the CITRIS building at the university of California-Berkeley'. In: (2012).

[25] Danmarks Statistik. *ENE3H: Bruttoenergiforbrug i fælles enheder efter branche og energitype* [*Accessed: 23/2/2017*]. http://www.statistikbanken.dk/ENE3H. 2017.

[27] Stephen Dawson-haggerty, Andrew Krioukov, Jay Taneja, Sagar Karandikar, Gabe Fierro, Nikita Kitaev and David Culler. 'BOSS: building operating system services'. In: *NSDI '13* (2013), pp. 1–15.

[28] Gabriel Fierro and David E Culler. 'XBOS: An Extensible Building Operating System'. In: *Proceedings of the 2nd ACM International Conference on Embedded Systems for Energy-Efficient Built Environments*. ACM. 2015, pp. 119–120.

[40] Charles McParland. 'OpenADR open source toolkit: Developing open source software for the smart grid'. In: *Power and Energy Society General Meeting, 2011 IEEE*. IEEE. 2011, pp. 1–7.

[46] Stephen Dawson-Haggerty, Xiaofan Jiang, Gilman Tolle, Jorge Ortiz and David Culler. 'sMAP: a simple measurement and actuation profile for physical information'. In: *SenSys'10* (2010). DOI: 10.1145/1869983.1870003.

[47] Michael P Anderson, John Kolb, Kaifei Chen, David E Culler, Randy Katz, Michael Andersen, John Kolb, Kaifei Chen, David E Culler and Randy Katz. 'Democratizing Authority in the Built Environment'. In: *BuildSys*. 2017.

[48] Gabriel Fierro and David Culler. 'HodDB: Design and Analysis of a Query Processor for Brick.' In: *IEEE BuildSys '17* (Nov. 2017).

[49] Apache Jena. *Apache Jena Fuseki*. https://jena.apache.org/.

[51] Satoshi Nakamoto. *Bitcoin: A peer-to-peer electronic cash system*. 2008.

[52] Gavin Wood. 'Ethereum: A secure decentralised generalised transaction ledger'. In: *Ethereum Project Yellow Paper* 151 (2014).

[62] Graham Klyne and Jeremy J Carroll. 'Resource Description Framework (RDF): Concepts and Abstract Syntax'. In: *W3C Recommendation* 10.October (2004), pp. 1–20. URL: http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/.

[63] Emil Holmegaard, Aslak Johansen and Mikkel Baun Kjærgaard. 'Metafier - a Tool for Annotating and Structuring Building Metadata'. In: *SmartWorld'17*. 2017.

[69] Eric Prud'hommeaux and Andy Seaborne. 'SPARQL Query Language for RDF'. In: *W3C Recommendation* (2008). DOI: citeulike-article-id:2620569.

[73] Bharathan Balaji, Arka Bhattacharya, Gabriel Fierro, Jingkun Gao, Joshua Gluck, Dezhi Hong, Aslak Johansen, Jason Koh, Joern Ploennigs, Yuvraj Agarwal, Mario Berges, David Culler, Rajesh Gupta, Mikkel Baun Kjærgaard, Mani Srivastava and Kamin Whitehouse. 'Brick: Towards a Unified Metadata Schema For Buildings'. In: *BuildSys*. 2016. DOI: 10.1145/2993422.2993577.

[74] Bharathan Balaji, Arka Bhattacharya, Gabriel Fierro, Jingkun Gao, Joshua Gluck, Dezhi Hong, Aslak Johansen, Jason Koh, Joern Ploennigs, Yuvraj Agarwal, Mario Bergés, David Culler, Rajesh K. Gupta, Mikkel Baun Kjærgaard, Mani Srivastava and Kamin Whitehouse. 'Brick : Metadata schema for portable smart building applications'. In: *Applied Energy* (2018). DOI: 10.1016/J.APENERGY.2018.02.091.

[80] Mikkel Baun Kjaergaard, Aslak Johansen, Fisayo Sangogboye and Emil Holmegaard. 'OccuRE: An Occupancy REasoning Platform for Occupancy-Driven Applications'. In: *CBSE'16*. IEEE, 2016. DOI: 10.1109/CBSE.2016.14.

[97] Sam Newman. *Building microservices: designing fine-grained systems*. " O'Reilly Media, Inc.", 2015.

[98] Thomas Erl. *Service-oriented architecture: concepts, technology, and design*. Pearson Education India, 2005.

[99] Thomas Weng, Bharathan Balaji, Seemanta Dutta, Rajesh Gupta and Yuvraj Agarwal. 'Managing plug-loads for demand response within buildings'. In: *BuildSys'11*. ACM. 2011.

[100] Frauenhofer. *Omega 2.0*. https://www.iis.fraunhofer.de/en/ff/lv/ener/proj/ogema-2-0.html.

[101] Flexiblepower-Alliance-Network. *EF-Pi*. https://flexible-energy.eu/ef-pi/.

[102] Arka Bhattacharya, Joern Ploennigs and David Culler. 'Short paper: Analyzing metadata schemas for buildings: The good, the bad, and the ugly'. In: *Proceedings of the 2nd ACM International Conference on Embedded Systems for Energy-Efficient Built Environments*. ACM. 2015, pp. 33–34.

[103] Jon Meyer and Troy Downing. *Java virtual machine*. O'Reilly & Associates, Inc., 1997.

[104] Holger Knublauch, Ray W Fergerson, Natalya F Noy and Mark A Musen. 'The Protégé OWL plugin: An open development environment for semantic web applications'. In: *International Semantic Web Conference*. Springer. 2004, pp. 229–243.

[105] Michael Andersen. *Bindings for BW2*. https://github.com/immesys/bw2bind.

CHAPTER 12

SERVICE PORTABILITY AND DISCOVERY IN BUILDING OPERATING SYSTEMS USING SEMANTIC MODELING

This chapter is a cosmetic adaptation of the following conference paper: Jakob Hviid, Aslak Johansen, Gabe Fierro and Mikkel Kjærgaard. 'Service Portability and Discovery in Building Operating Systems Using Semantic Modeling'. In: *2020 IEEE International Conference on Pervasive Computing and Communications (PerCom)* (*PerCom 2020*). ACM. Austin, USA, Mar. 2020. **Submitted**.

The paper is submitted to the 18th Annual IEEE International Conference on Pervasive Computing and Communications, in Austin, Texas (USA), and if accepted, will be presented on 23 March 2020.

12.1 ABSTRACT

To achieve cost-efficient Building Operating Systems (BOSs), portable building services are needed. This paper presents an ontology-based service discovery mechanism for BOSs. The semantic approach enables interoperability with other semantic models used in the BOS space. The built environment is characterized by extreme heterogeneity; no buildings are entirely alike. Also, Equipment is replaced or updated, control systems and building functionality evolve, as applications, models, forecasters, and controllers improve. Therefore, services deployed in these settings must be robust to change, for them to operate at scale. Describing service interfaces using a semantic model, together with the physical context in a building, enables querying for the services needed, allowing applications to search for the data needed. This change allows applications to depend on an abstract query, instead of specific services. For evaluation, nine services running on models of the service ecosystems of three concrete buildings, are implemented, demonstrated and

evaluated.

12.2 INTRODUCTION

With society's growing concern for the environment, a significant amount of resources is being put into the green energy sector. One of the subfields in this sector seeks to improve the energy efficiency of buildings, but also to support the move to green energy sources through tight integration with the energy grid. This integration is referred to as Demand Response (DR) [19] DR allows for offsetting some of the effects of the inherent unpredictable energy production of green energy sources, thereby facilitating removal of the traditional energy sources that are undesirable for the green energy agenda. DR is achieved by allowing buildings to act as energy storage, by shifting operations, or by turning off non-essential services in response to energy grid demand. In the United States, California [18], DR is additionally used to ensure a more stable energy grid.

Traditionally buildings have been controlled by closed-loop systems, typically produced and maintained by single companies. These systems are called Building Management Systems (BMSs). Over time buildings are expanded and evolve, which means introducing the building changing BMSs and implementations, possibly even introducing multiple of these. These changes make it challenging to integrate these systems into DR operations. Fortunately, one of the growing trends in research on building operations is BOSs [28, 27, 29], that works as a layer on top of the hardware, or their controlling BMS. The presence of such a layer allows applications and services to be abstracted from the specific implementations of the hardware exposed by the BMSs. This abstraction allows the BOS to orchestrate the resources as needed, and on several scales; from a single building to citywide implementations.

Figure 12.1 captures the basic concept of the Hardware Abstraction Layer (HAL) and Service Abstraction Layer (SAL), in the context of a BOS. The components enabling service discovery, effectively implementing the SAL, will henceforth be referred to as just the SAL, as it is a new implementation of the concept introduced by Hviid and Kjærgaard [3]. The architecture is based on microservices and includes a bus for communication and small services that each solve a single well-specified problem area. Actuators and Sensors, generally regarded

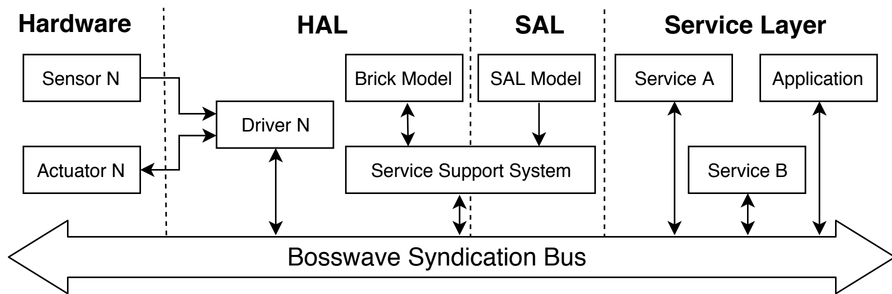

Figure 12.1: BOS Layers overview with SAL.

as the hardware components are abstracted using the HAL. In this case, the HAL is a combination of Brick, a Brick Model, the Service Support System (SSS) to host the model, and finally, the drivers that hide the specific implementation of the hardware.

In buildings it is important to understand the context, so location, direction, placement, etc., of the data being collected or processed. To provide this context, some BOSs are beginning to employ metadata models [106] or semantic models. Brick [74] is an example of a semantic approach to describing buildings, their layout, and how the hardware relates to each other and the building. However, Brick does not describe software services within the building, and how they relate to each other, nor does BOSs HAL provide this. Therefore, the SAL integrates with Brick, and benefits from its strengths. These software services provide a multitude of processed information like fault detection, prediction, occupant modeling, DR and more [2, 78, 79, 20, 80]. The missing abstraction between services, or between services and applications, makes it difficult to integrate a DR service on a larger scale between buildings, as service portability between buildings is dependent on specific implementations. Also, the need for re-implementation or reconfiguring services for each building makes the act of participating in inter-building coordination endeavors like DR infeasible from a cost-benefit perspective.

To help achieve large-scale deployment of applications and services, portability is critical. To enable portability, this paper introduces an ontology for describing services in BOSs, and thereby enable further abstraction from the building implementation. The service description enables applications and services to interface with other services,

without prior knowledge of location, or data structure. In buildings that are maintained over an extended amount of time, multiple changes, and sometimes multiple BMSs are present. This makes it difficult to integrate to only one specific service, as it could end up being different for buildings from a different decade or geographical location. For an application to work successfully in these old and changing buildings, it needs to be assumed that applications do not know what services are present, and vice versa with the services. As a consequence, an application will often be forced to retrieve information from multiple services to achieve a task. Some applications might depend on information not present in the building, which would require the application to degrade gracefully, or an engineer to install the missing service. Furthermore, the ontology, as a minimum, needs to be tied into Brick's semantic model of the building, as the context of what information is being processed by it is paramount for it to be queryable and relevant.

This paper contributes a semantic service discovery implementation, that enables service portability and discovery between buildings, as well as providing improvements to high availability and resilience initiatives in BOSs. The six services are deployed and moved between 3 different case buildings, without any changes to implementation, to evaluate the portability benefits.

12.2.1 Related Work

In the field of BOSs and services, of course, several contributions have been made in a field like Building Operating System Services (BOSS) [27], SMAP [46], eXtensible Building Operating System (XBOS)[28], Bosswave [47], Buildingdepot [29], Sensor Andrew [31], Tridium Niagara [32], Volttron [30] and several others. One of the main contributions of the BOSs above, and the supporting technologies is the addition of a HAL, as described in the introduction. A HAL's role is to adapt the hardware specific interfaces to a standardized interface. This standardized interface enables applications to interact with any vendors hardware in a similar manner, thereby simplifying application implementation. This change allows the application to be written once regardless of which vendor implemented the hardware. Unfortunately, the HAL in itself does not allow for hardware or service discovery, and moving the application to a new building where the hardware structure, besides the vendor, is different without manual configuration

changes. In the intersection of BOS and DR, several contributions have been made, like improved DR decision making by introducing a service that tracks occupant activities [2] and, Nellemann et al. [20] explored automated decision and reasoning concerning DR and BOSs, to enable DR decisions that uphold specific configurable goals. The system controlling these parameters is called Controlium which is presented in the paper. In regards to entire DR systems, EF-Pi [101] is a BOS explicitly designed to serve a single DR-related application. This impacts the value proposition as added functionality is a major selling point for consumers and organizations to install BOSs in the first place and get started towards the path of smart buildings. Other examples can also be found, such as Peffer et al. [22] doing HVAC related DR, and Weng et al. [99] that seeks to include plug-loads into DR.

Service discovery is a broad term, that is used for different purposes at several layers of system architecture. At the lower level, Zeroconf [107], UPnP [108], and SLP [109] exists. Zeroconf's intention is for automatic address configuration and hostname resolution without a DNS server. UPnP is intended as a full system configuration and service discovery protocol suite for computers and devices at a network level. One example of UPnP is an automatic configuration of port forwarding in home routers. SLP is a lightweight service announcement and request protocol, used for locating printers on a local network. SLP supports a limited query language that allows querying for services properties. The three protocols all have in common that they do not support complex service querying for context and that their intended use is network level discovery and not context discovery. Expanding the exploration to higher abstractions of service discovery introduces several other technologies. Among these are Berkeley's Ninja Service Discovery [110] and Apache River [111]. With the above technologies, applications require prior knowledge of a service interface, and the context of the service provided. When exploring the above service discovery techniques, we discover they are not centered around the notion of zero-knowledge between the application and service, but about detecting instances of a service solving distinct tasks such as High Availability (HA), failover functionality, or finding the location of devices with a specific service running. Also, these services have distinct interfaces that the application is still integrating to directly, and has been specifically built around.

12.2.2 RELATED WORK

Table 12.1 compares the related technologies with the SAL approach. Three types of related work is compared:

1. Two BOS specific implementations are detailed: A combination of SMAP [46] and Metafier [63], and Brick [74].
2. Two different service description ontologies: OWL-S [71], and WSMO [72].
3. Two service interface description methods: WSDL [81], and gRPC Remote Procedure Calls (gRPC) [82].

The SAL is designed to function in the BOS space; therefore, exploring existing technologies in this space is relevant. First, the combination of SMAP [46] and Metafier [63] is discussed. Metafier adds metadata to streams of information, thereby allowing the creation of a logical hierarchical structure, but does not explicitly address discovery needs of services and their context. Metafier seeks to add metadata related to hardware devices in a building, by having an expert tell Metafier where resources are located, and what kind of properties the streams have. These annotations are associated with single streams of information, not distinguishing between hardware or services. The metadata allows Metafier to build up a logical hierarchical structure, that signifies the composition of a building, and the relation that sensors and actuators have to each other in the spatial dimension. Metafier has a limited ability to express the physical context, but only by non-related strings that, for example, can be used to distinguish between the buildings or rooms to which the stream belongs. However, these descriptions do not allow for any description of how one room relates to a building, or if a room is adjacent to another. The SMAP protocol allows for querying for streams and can take into account the metadata created by Metafier. However, these queries are limited by the limited expression of Metafier itself. Also, SMAP is not created with elaborate ontology descriptions in mind and does therefore not support these types of abstractions. Because SMAP is a protocol for retrieval of time series data, it has a precise and detailed query for the temporal aspect of the data. However, this temporal aspect is in the form of a query for data, and not a description of what can be expected of the endpoint. This is due to SMAP also acting as a HAL, and that all requests for data are expected to go through SMAP. Also, the temporal query parameters

	SAL	SMAP + Metafier	Brick	OWL-S	WSMO	WSDL	gRPC
Type	Ontology	Stream Metadata	Ontology	Ontology	Ontology	Interface Desc.	Interface Desc.
Service Descriptions	Yes	No	No	Yes	Yes	Yes	Yes
Physical Context Description	Yes	Limited	Yes	No	No	No	No
Queryable	Yes	Limited	Yes	Yes	Yes	No	No
Modality Description	Yes	Limited	Limited	No	No	No	No
Unit Description	Yes	Limited	Limited	Primitives	Primitives	Primitives	Primitives
Temporal Aspect Description	Yes	Yes	No	No	No	No	No
Organizational Description	Yes	No	No	No	Yes	No	No
Multipath Options	Yes	No	No	No	No	No	No
Multipath Priority Options	Yes	No	No	No	No	No	No

Table 12.1: Related Work Comparison.

are describing the time series data retrieved, and not the context of a single property within. Brick [74] is an ontology, that also seeks to enable service and application portability and uses ontologies to describe the hardware and how it relates to locations, and other hardware. The ontology approach has proven itself to be significantly more descriptive than the Metafier approach, capturing significant aspects of a building and how it relates to other components. Also, SPARQL Protocol and RDF Query Language (SPARQL) is versatile, allowing for complex questions about the structure of the building to be answered. Brick is specifically designed to describe the physical context of sensors and actuators, as well as how the hardware and rooms inside of a building relate to each other. It therefore not only describes the physical context exceptionally well but is also used as the description of this in the SAL. Brick, however, does not concern itself with services, interfaces, or high availability. Both Brick and Metafier does allow for Modality and Unit descriptions but is currently limited to the types existing in the hardware space, neglecting the more complex types found in the services space. Neither Metafier or Brick, models services, but only hardware (by design), and therefore fails to capture the output of services, their properties, and context. Moving on to the ontologies for service descriptions, OWL-S [71] and WSMO [72], they both describe similar perspectives on services. Both are more concerned with control logic, and interfaces, but does not describe the context of the dataset retrieved from modality, unit, temporal aspects, or physical context. While they do have some unit descriptions, these are limited to more primitive data types like integers, doubles, or strings. WSMO does describe organizational relationships of the service in the form of who created the service, and who owns it. Also, while WSMO does not support multipath HA descriptions, it does have several parameters like Robustness, and Scalability, though these describe more about the properties of the service than actually providing multiple paths. The case could be argued that either of these ontologies could be extended to support the contextual descriptions the SAL sets forth to solve. However, both ontologies are complex and do not describe anything about the content itself and how to read it, but more the interfaces to the services. Also, the ontologies complexity adds overhead with concepts like atomic processes and more, while the SAL sets out to present the core relations needed to solve the issue at hand. WSDL [81] and gRPC [82] describes the interface, input, and output of a resource.

They describe the parameters an RPC endpoint needs to function and the returned values. These technologies describe how to interface with the service, but they do not describe the context of the services they expose, nor do they allow for querying for them, as Table 12.1 states. None of the alternatives to the SAL presented in Table 12.1 describes the properties needed to support services in a BOS context.

This paper builds on the experiences made by Hviid and Kjærgaard [3], who introduced the term Service Abstraction Layer, or SAL. The paper proposes a SAL based on an ontology but unfortunately has several shortcomings. **1**) The implementation of the ontology relies on a simple inheritance strategy of a service endpoint, which is restricted by its limited expressiveness. Specifically, it does not deliver the nuance needed to locate endpoints with specific types of information successfully. **2**) As previous work points out, a service is not able to successfully find the specific information it needs from a transmitted object. This forces the service using the service endpoint to know the specific implementation of the providing service, or for there to exist a detailed specification for how these service endpoints interfaces, based on the service endpoint's inherited class. **3**) previous work also support failover functionality but does not allow for a mechanism to prefer one over another, apart from just choosing the first that was entered into the model. This is a significant shortcoming for a failover implementation. This paper builds on their work, addresses the above papers shortcomings, and add upon the functionality.

12.2.3 Approach

Solving the described issue first requires framing and boundaries. So, how can application portability and service discovery be achieved given the following restrictions? **1**) The classes of services available at this time, including ventilation usage prediction, energy benchmarking, presence prediction, weather prediction and finally solar battery storage prediction. **2**) No prior knowledge of service interfaces specifications are given, so services are built as general services, as opposed to a specific application needs, or an application built directly on a specific service. **3**) Neither application or service has prior knowledge of each other.

First, given the above restrictions, what relations need to be modeled? Second, what requirements does our class of services need to satisfy

to express their interfaces successfully? Third, how do these expressions relate to the physical aspect of the building the service relates to? Moreover, how do we restructure building services to be general services, and not to be built for specific applications? Currently, most researchers approach building applications as one problem that needs to be solved for only that application. This approach results in large monolithic applications that include all functionality needed to achieve a goal, but as a consequence, it also encloses general solutions to problems, leaving them unreachable for other services in the BOS.

To ensure service portability, an ontology approach was chosen, as prior implementations in related areas, using ontologies has been proven to be successful. These expectations are due to the fact that services and applications have no prior knowledge of each other.

12.3 CONCEPT

The SAL is the product of the simple idea of having service and interface discovery in BOSs, as an application will not have prior knowledge of what services are available, or what interfaces they expose. When applying service discovery to a building, identifying the context of the information exposed is necessary. The first need is to know where the service is located, and what properties are present in the information you can gather from that location. The SAL seeks to expose and describe these properties, with modality, unit, and spatiotemporal aspects. Also, due to the recent initiatives such as EU General Data Protection Regulation (GDPR), an ever-growing interest in ownership of data, the SAL also models ownership of an endpoint's exposed data. This is especially useful in mixed environment buildings encompassing multiple companies or private occupants. Like Brick, the service interfaces are described using an ontology, resulting in a model, which can be queried by other services that have dependencies on information, instead of specific services. The SAL allows for this kind decoupling from other services, because of the expressiveness of the RDF modeling method.

12.3.1 TERMINOLOGY

The SAL consists of several concepts and elements described below. The *SAL Ontology* is the description of SAL related concepts and their

relationships to each other. It is based on RDF [62] and OWL [83], with references to the Brick Ontology and Schema.org Ontology [68]. The *SAL Instances*, or *SALI*, is a collection of instances of modalities, units, and more, ensuring only one instance of a particular concept exists. The *SAL Model* is the Turtle file containing the modeling specific to the service implementation of a building.

12.3.2 SAL benefits and implications

The SAL facilitates several functionalities and benefits. First, the primary purpose of the SAL is the provide a facility to support service discovery, thereby allowing applications to integrate with services without prior knowledge of their specifics. The service discovery allows the services consumers, that can be both services or applications, to be portable between buildings without changes in implementation. Second, the change in architecture allows developers to change their services over time, for example, splitting services into multiple interfaces or merging them. This can be done without breaking compatibility with applications that depend on the exposed information, as long as it exposes the same types of information over the new interfaces, with the same context. The above change allows for service developers to move from mostly monolithic service structures to microservices over time or vice versa.

Another area where the SAL contributes is in the area of describing information ownership. This description is especially useful in the case of mixed environment buildings, with multiple companies inhabiting the building, but also if the BOS is serving a larger area with several buildings, or even on a city scale. This usefulness is derived from the SAL's support of ownership description of service endpoints. The SSS mentioned earlier, allows the SAL to be dynamically updated. This system allows services to be installed, query the HAL and SAL, configure itself, and publish its functionality into the SAL. Effectively, the SSS enables plug and play services. Due to the nature of how services are described in the SAL, several paths to similar information can be described. This type of over-provisioning of information, enables services to implement its own HA functionality or load balancing.

Using a microservice-oriented architecture, and the SAL introduces several complexities. Debugging a service architecture where dependencies are loosely defined can be difficult, and generally increases the

focus of observability. This change will require developers to be aware of how they expose errors in the system and tell the technician why a service or application is not working. Failing to do so, or as a minimum give examples of what services an application needs, could make it difficult for a technician to diagnose problems. If error messaging is made correctly though, debugging should be relatively easy even with a service architecture this loosely coupled. From the perspective of the developer though, it is easier to detect what component is broken, and fix only that, compared to a monolithic application. This could increase correctness and robustness, as connections between program components are formalized.

12.3.3 SAL ONTOLOGY DESCRIPTION AND SALI

This section describes the anatomy of the SAL Ontology, based on the requirements described earlier in this paper. Figure 12.2 illustrates an overview of the parent classes in the SAL ontology. Subclasses are not present, as the diagram would become too large if it had to encompass all the subclasses. The implemented specific modalities and units are representative of the data streams collected across three buildings by the author's group. On top of this, several additional modalities and units were added, based on the needs of the services developed for the evaluation section.

Service describes a single service and has an object property called *provides*, which defines all ServiceEndpoints created by the service. The service class acts as a starting point for all of its endpoints and is a representation describing an actual service existing in the BOS. It has one data property, *name*, that helps give a meaning to a developer browsing the instance of the class. The *Service Endpoint* class is typically what applications are trying to locate when querying, as it encapsulates the Uniform Resource Identifier (URI) for locating the desired information. It has four data properties. First, *Read* and *Write* properties define whether the endpoint expects parameters to function, or if it provides information, or both. The data property *Priority* defines the service's priority compared to other services that provide the same kind of information. This is an indicator for depending services of which order to contact a dependency provider in case of multiple options. Finally, the data property *Uri* defines the actual location where a service will be able to locate the information. This URI can refer to the Bosswave

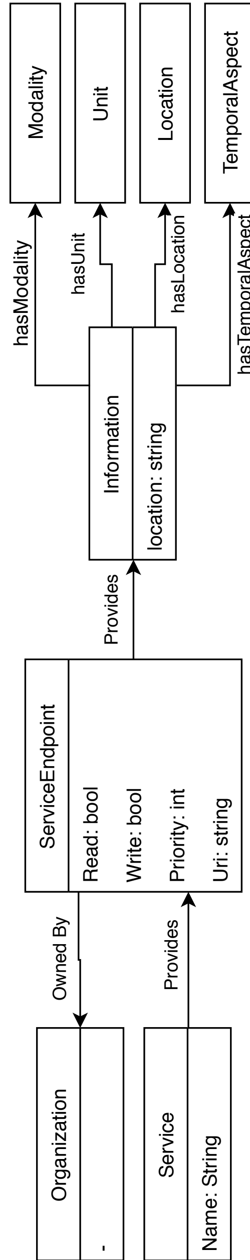

Figure 12.2: SAL Ontology Parent Classes - Relationships and Properties.

Syndication Bus, REST or similar technologies.

The Service endpoint has two object properties, *OwnedBy* and *provides*. *OwnedBy* defines the organizational owner of the information provided through the endpoint, while *provides* links to the information parent class. *Information* represents a property inside of the information obtained from the URI residing in the Service Endpoint. One endpoint will typically have a multitude of *information* object properties, that point to separate instances of information. It is important to mention that this structure assumes that the returned object is a fixed structure, and does not change. One data property is present on Information, *location*, which is a descriptor of where this single piece of information can be found within the returned object that can be retrieved from the URI described in the ServiceEndpoint. The format of the data property is not formally defined as it depends on the architecture of a given BOS. For example, if JavaScript Object Notation (JSON) is used, one possible value could be ”[].MyProp” to describe that the property is found in MyProp in each object in the array received. Alternative implementations could be used here using gRPC or other alternatives that suit the specific implementation of the BOS. The information is described by its object properties, *hasModality*, *hasUnit*, *hasLocation*, and *hasTemporalAspect*. Each of these object properties describes a small but significant part of the context of that piece of information. This is an essential part of querying the SAL, as it enables the querying for the context of the information.

The *Modality*, *Unit* and *TemporalAspect* parent classes each consist of several subclasses, none of which have any object or data properties. The subclasses are shown in Table 12.2. These subclasses used in conjunction, add context to the information being described. An example could be the combination of the modality Wind, the Unit MetersPerSecond, and the temporal aspect Prediction. Each has little meaning by itself, but the combination provides an added insight. The lists of modalities and units are on what needed types where observed, but more will need to be added over time. To save the developer time, each descriptive type have instances defined beforehand in the SAL Instances file. This file is not a requirement, but a convenience. Creating these instances for the developer beforehand ensures cleaner models, as well as only one type of each existing at any given time. All instances are subclass of Modality, Unit, and TemporalAspect.

Imported from Brick, *Location* adds a spatial dimension, describing

Modality	Unit	TemporalAspect
Angle, CO_2, Presence, Flow, Illuminance, Power, Pressure, Rain, Humidity, Temperature, Wind, AbsoluteTime, RelativeTime, PowerFlexibility, Performance, Energy, Certainty, Time	Boolean, Count, CubicMeters, CubicMetersPerHour, DegreeCelsius, DegreeFahrenheit, Degrees, GigaByte, KiloByte, Hours, Hertz, Joules, JoulesPerCubicMeter, Kelvin, KiloJoulesPerSquareMeter, KiloMeters, KiloWatts, KiloWattHours, Lux, CubicMeters, CubicMetersPerHour, CubicMetersPerSecond, MilliAmperes, Minutes, MilliMeters, MilliSeconds, MetersPerSecond, MilliVolts, MilliWatts, MilliWattHours, Pascal, Percent, PartsPerMillion, RotationsPerMinute, Volts, Watts, Unitless, Time, DateTime, Date	Prediction RealTime Archival

Table 12.2: Subclasses for information annotation.

the concept of building, room, zone, and more. As Brick is already describing the physical properties of a building and uses the same types of descriptive technologies, we consider it well suited for describing the physical context.

Returning to one of the *Organization* class, this class represents a company and its ownership of the information contained at the URI residing in the service endpoint. To not re-engineer a concept already thoroughly explored, the *Organization* parent class is imported from the schema.org [68] ontology. Schema.org has already contributed a significant amount of work modeling organizations, including a multitude of subclasses, as well as the possibility of describing divisions, owners, contact information and more. It is up to the BOS developer to choose the level of detail one wishes to have in these models. This parent class is particularly interesting in buildings with shared spaces between several companies, where a service might only service a single company, and not have permission to process data gathered from other sources. The class does not have any data properties in the figure, as everything is inherited from schema.org.

Figure 12.3 visualizes a limited sample of a SAL Model for a weather prediction service. The example is limited by showing one service

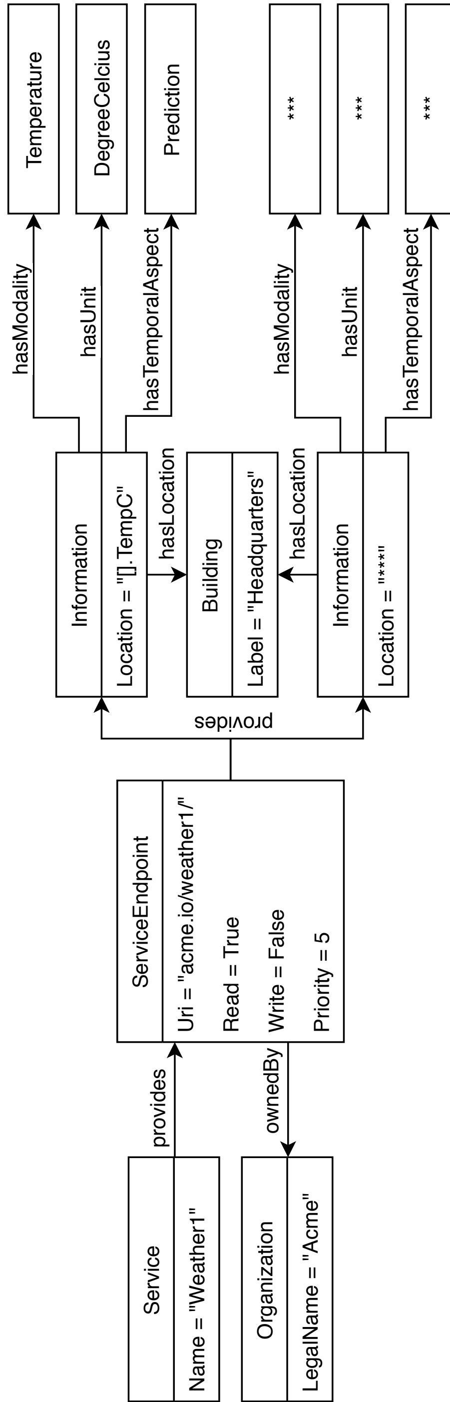

Figure 12.3: SAL Model; Weather Example.

endpoint, and one describing property. The service endpoint has a priority set, giving it a weight if more than one weather prediction service is present. The information modeled is owned by the Organization "Acme". The information found at "acme.io/weather/1", is described to be an array of objects, where each array has a property called "TempC", which is a temperature measured in degree Celsius. The value is a predictions about the weather around the "Headquarters" building. Additional information could be described, which is indicated by the last information box.

12.4 EVALUATION

To evaluate the SAL and the components, an evaluation setup is built. The BOS selected and used for the verification implementation is based on XBOS [28] combined with Bosswave [47] and Brick [74]. The ontology is built using, Protégé using Web Ontology Language (OWL) and Resource Description Framework (RDF), while the instances of the ontology and the models of the buildings are generated using Python and RDFLib. The SAL Ontology, Instances, and Model files are all saved in the Turtle [84] file format. The SSS is used for hosting the SAL ontology, SAL Instances, and SAL Model, but also Brick and the Brick Model. All queries are in the SPARQL format. The experimental setup is described in Figure 13.2. The hardware and HAL are included in the diagram, to support the understanding of how the entire setup is working. Bosswave is providing integrated communication, authentication and permission functionality, which is implemented using the Ethereum Blockchain. All services for a particular building exists in the service layer and communicates through the Bosswave Syndication Bus. An application would query the SAL, and then contacts the resolved provider through Bosswave.

The evaluation is performed on three different SAL Models inspired by actual buildings and their hardware. These represent different types of buildings, namely an office building, a retail store, and an educational building. The services are structured differently for each building, but are using the same implementation of the services, to as the portability of these services are critical.

Figure 12.5 shows the dependency between services for each of the buildings. There are six different types of services in the setup, all

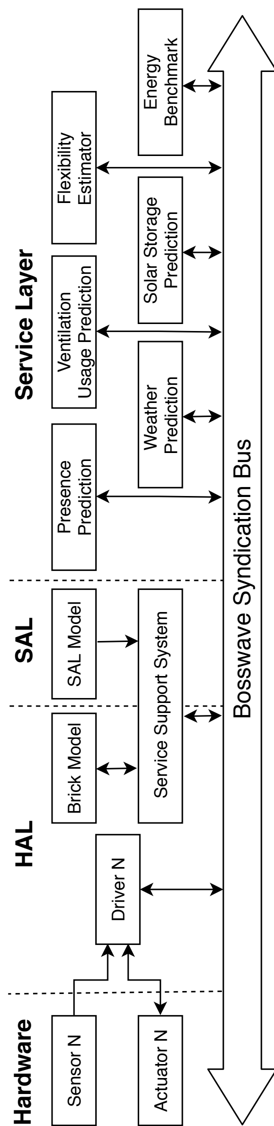

Figure 12.4: Evaluation Microservice Ecosystem.

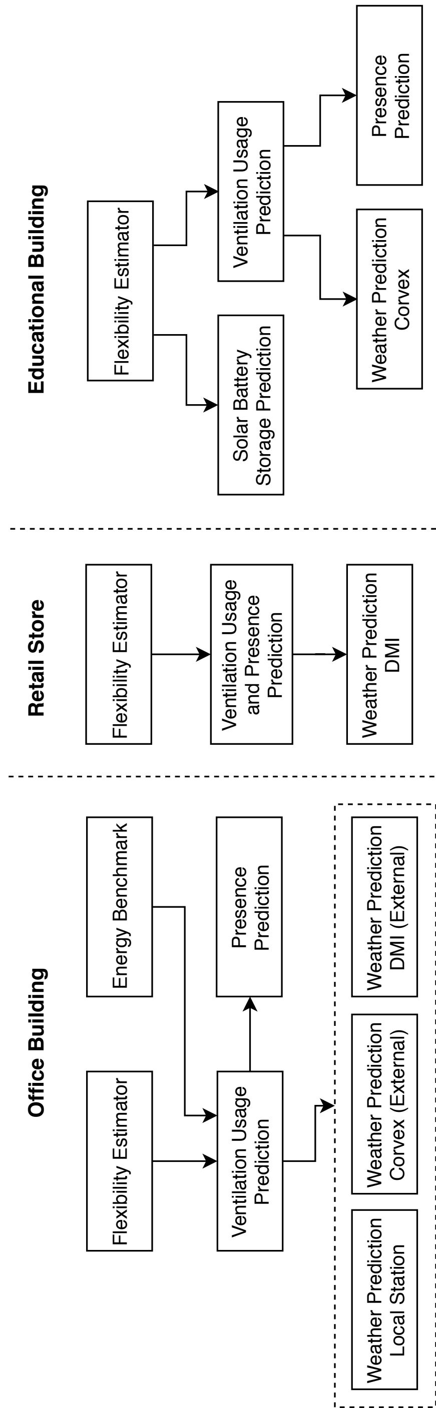

Figure 12.5: Service Dependencies For Each Case Building.

providing predictions. The flexibility estimator uses information from the solar battery storage prediction service, and the ventilation usage prediction service, to estimate the available flexibility potential. This functionality is used for DR purposes. The energy benchmark service evaluates the performance of the building, and how it changes over time. The ventilation usage prediction service uses the presence prediction and weather service prediction to estimate the need for ventilation in a room. The presence prediction service is based on the OccuRE framework [80] for predicting occupancy. Note that while some rooms are measured as a Boolean indicating presence, others have better sensors allowing for prediction of occupancy counts. This nuance is included in the SAL model. The solar battery prediction is only present in the educational building as that is the only building to have solar panels and battery storage. The service predicts the given capacity of the battery storage at any given time. For the retail store SAL Model, the ventilation usage service, and the presence prediction services are contained in the same service. This is to evaluate if service merges have an impact on the queries needed to locate the information needed.

Each BOS service in this experiment is self-configured through queries on the relevant SAL model. The office building has an extra service depending on the ventilation usage prediction service and has several weather prediction services. Two of these are external to the building instrumentation, existing in the cloud, and one that is internal in the form of a service interpreting output from a local weather station. The existence of several weather prediction services allows testing of failover functionality. The retail store SAL Model is primarily chosen to allow testing the merging of service functionality, and the impact on the queries needed to find the information. The educational building SAL Model primarily acts as the ideal setup for the flexibility estimator service. So, when the service gets moved to the other SAL Models, it is missing this service and needs to adapt to the sub-optimal environment. Of course, all of the models contribute to testing the implemented services, and thereby queries, across different environments.

The portability of services is demonstrated by using the same queries on all the models for their respective services. As expected, even though the setups are different, the queries and service implementations adapt and work on each building without changes to the code. All data transmitted between services is synthetic data, as real-time data does not benefit the evaluation of the ontology and SAL layer.

The infrastructure for high availability is tested using the office SAL Model, as it has multiple weather prediction services available. The model supplies the infrastructure for implementing failover. The ventilation usage prediction service is forced to adapt to changes in the infrastructure when a service is forced offline.

The expectations of the evaluation are as follows: **1**) *Service Portability:* Move the services between the three different SAL Models. *Expected result:* Services are working, and adapts to the changing models. No code changes needed. **2**) *Merging of Services:* Move to a SAL model where one service fulfills two service roles, instead of separate services. *Expected result:* Depend services keep running without code changes. **3**) *System Resilience:* Several services expose the same information type, and have depending services. The used service is then removed. *Expected result:* The depending service keeps functioning correctly.

12.4.1 RESULTS

Table 12.3 shows the results of the evaluation setup, with results split into the three evaluation areas. Section **A** refers to the portability evaluation, where the requirement for a ✓ is that the service ran on the SAL model for the specified building, and could successfully gather the information it needed from the services it depended on. Dashes refer to a situation where the service is not present in the model, meaning there is no evaluation. As the table shows, all portability tests ran successfully, as expected. Section **B** refers to the merging and splitting of service roles, and here a ✓ means to a successful merge or split, without impact on services. The table shows all merge or splits of services worked as expected. Section **C** refers to the system resilience aspect of the evaluation, where a ✓ means the failover test for the Office Building worked as expected, by arbitrating similar or competing services. Dashes mean that that type of evaluation was not available for that SAL model. All testing was performed at runtime. Based on the observations during the evaluation, the SAL should scale comfortably to several thousand services. This number depends significantly on the implementation of the SSS, and query complexity.

	Flexibility Est.	Energy Bench.	Vent. Usage Pred.	Presence Pred.	Weather Pred.	Solar Storage Pred.	Merge/Split Services	System Resilience
Office Building	✓	✓	✓	✓	✓	-	✓	✓
Retail Store	✓	-	-	-	✓	-	✓	-
Educational Building	✓	-	✓	✓	✓	✓	✓	-
	A						B	C

Table 12.3: Service Portability Evaluation Results.

12.5 DISCUSSION

The results show service portability, merging and splitting of services, and system resilience, are all functioning as expected. This means the interfaces was successfully expressed using the SAL ontology, as well as the physical context, also allowing for a restructuring of building services, without impact on the depending services.

Several factors could be improved, or explored further. First, BOSs has a steep learning curve. Several technologies, terms, and abstractions that are introduced require the developer to explore technologies like blockchain (due to the Bosswave Syndication Bus in Figure 12.1), ontologies, SPARQL, RDF, turtle and many more. These BOSs and abstractions layers, could benefit from frameworks that encapsulate these in tools that are familiar for developers. A framework that abstracts some of this into a simple package could severely reduce the initial friction when moving into the field of BOSs, and require less prior knowledge. As familiarity increases, the developer can move into the more obscure parts of the systems.

In regards to the SAL Ontology, services requiring input parameters still require some prior knowledge of the service. For example, if a service changes its prediction regularity, the endpoint facilitating these changes cannot currently be described, as the SAL Ontology is created from the needs of the service examples in this case. One solution would be to let the service accept input defining temporal granularity for the output of the service. To alleviate the above restrictions, the ontology needs further expansion. Another area where the SAL ontology needs

expansion is in regards to modalities and unit types. The given types in this paper are not exhaustive, and only by working more with the SAL Ontology and services will these be explored. Fortunately, by nature, RDF ontologies are accessible for a user to expand upon, and will not be restricted by the current modalities and units.

12.6 CONCLUSION

This paper set out to create an ontology that support service discovery, and enable portability of BOS applications and services, in an ecosystem with no prior knowledge between an application, and the services it uses. This goal has been achieved. The evaluation shows the SAL gives tangible benefits to the goal of enabling portable services. Also, introducing the SAL gives the added benefits of applications being dependent on information, and not specific services, enabling a more loosely coupled and adaptive service landscape. The SAL also enables potential resilience benefits to the BOS as services provide information, and this information can be provided redundantly. This redundancy enables services to implement failover functionality if services are sensitive to downtime. As the evaluation shows, failover functionality is successfully achieved by arbitrating similar or competing services.

The SAL enables large-scale deployment of services across different types of buildings that change over time, with the same codebase. Also, it allows developers to make standard products, instead of custom implementations for each customer, thereby reducing development costs per customer. For future work, the SAL should be evaluated by running on multiple buildings over a prolonged period of time, to validate it in a running, evolving, and expanding setting with multiple service developers using the ontology. However, the SAL paired with the HAL can have a significant impact on a BOS ecosystem and has the potential to change the return of investment calculations when deciding if a service is going to be profitable to develop.

REFERENCES

[2] Jakob Hviid and Mikkel Baun Kjærgaard. 'Activity-Tracking Service for Building Operating Systems'. In: *2018 IEEE International Conference on Pervasive Computing*

and Communications Workshops (PerCom Workshops). IEEE. 2018, pp. 854–859. DOI: 10.1109/PERCOMW.2018.8480362. **Published**.

[3] Jakob Hviid and Mikkel Baun Kjærgaard. 'Service Abstraction Layer for Building Operating Systems: Enabling portable applications and improving system resilience'. In: *2018 IEEE International Conference on Communications, Control, and Computing Technologies for Smart Grids (SmartGridComm)*. IEEE. Aalborg, Denmark, Oct. 2018, pp. 1–6. DOI: 10.1109/SmartGridComm.2018.8587543. **Published**.

[4] Jakob Hviid, Aslak Johansen, Gabe Fierro and Mikkel Kjærgaard. 'Service Portability and Discovery in Building Operating Systems Using Semantic Modeling'. In: *2020 IEEE International Conference on Pervasive Computing and Communications (PerCom) (PerCom 2020)*. ACM. Austin, USA, Mar. 2020. **Submitted**.

[18] Mary Ann Piette, Sila Kiliccote and Girish Ghatikar. 'Field Experience with and Potential for Multi-time Scale Grid Transactions from Responsive Commercial Buildings'. In: *ACEEE Summer Study on Energy Efficency in Buildings*. 2014.

[19] Mikkel Baun Kjærgaard, Krzysztof Arendt, Anders Clausen, Aslak Johansen, Muhyiddine Jradi, Bo Nørregaard Jørgensen, Peter Nelleman, Fisayo Caleb Sangogboye, Christian T. Veje and Morten Gill Wollsen. 'Demand response in commercial buildings with an Assessable impact on occupant comfort'. In: *SmartGridComm 2016*. 2016, pp. 447–452.

[20] Peter Nellemann, Mikkel Baun Kjærgaard, Emil Holmegaard, Krzysztof Arendt, Aslak Johansen, Fisayo Caleb Sangogboye and Bo Nørregaard Jørgensen. 'Demand Response with Model Predictive Comfort Compliance in an Office Building'. In: *SmartGridComm'17*. IEEE. 2017.

[22] Therese Peffer, David Auslander, Domenico Caramagno, David Culler, Tyler Jones, Andrew Krioukov, Michael Sankur, Jay Taneja, Jason Trager, Sila Kiliccote et al. 'Deep demand response: The case study of the CITRIS building at the university of California-Berkeley'. In: (2012).

[27] Stephen Dawson-haggerty, Andrew Krioukov, Jay Taneja, Sagar Karandikar, Gabe Fierro, Nikita Kitaev and David Culler. 'BOSS: building operating system services'. In: *NSDI '13* (2013), pp. 1–15.

[28] Gabriel Fierro and David E Culler. 'XBOS: An Extensible Building Operating System'. In: *Proceedings of the 2nd ACM International Conference on Embedded Systems for Energy-Efficient Built Environments*. ACM. 2015, pp. 119–120.

[29] Thomas Weng, Anthony Nwokafor and Yuvraj Agarwal. 'Buildingdepot 2.0: An integrated management system for building analysis and control'. In: *BuildSys'13*. ACM. 2013.

[30] Bora Akyol, Jereme Haack, Brandon Carpenter, Selim Ciraci, Maria Vlachopoulou and Cody Tews. 'Volttron: An agent execution platform for the electric power system'. In: *Third international workshop on agent technologies for energy systems valencia, spain*. 2012.

[31] Anthony Rowe, Mario E Berges, Gaurav Bhatia, Ethan Goldman, Ragunathan Rajkumar, James H Garrett, José MF Moura and Lucio Soibelman. 'Sensor Andrew: Large-scale campus-wide sensing and actuation'. In: *IBM Journal of Research and Development* 55.1.2 (2011), pp. 6–1.

[32] Vykon by Tridium. 'Niagara Networking & Connectivity Guide'. In: *Niagara Release* 2 (), p. 245.

[46] Stephen Dawson-Haggerty, Xiaofan Jiang, Gilman Tolle, Jorge Ortiz and David Culler. 'sMAP: a simple measurement and actuation profile for physical information'. In: *SenSys'10* (2010). DOI: 10.1145/1869983.1870003.

[47] Michael P Anderson, John Kolb, Kaifei Chen, David E Culler, Randy Katz, Michael Andersen, John Kolb, Kaifei Chen, David E Culler and Randy Katz. 'Democratizing Authority in the Built Environment'. In: *BuildSys*. 2017.

[62] Graham Klyne and Jeremy J Carroll. 'Resource Description Framework (RDF): Concepts and Abstract Syntax'. In: *W3C Recommendation* 10.October (2004), pp. 1–20. URL: http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/.

[63] Emil Holmegaard, Aslak Johansen and Mikkel Baun Kjærgaard. 'Metafier - a Tool for Annotating and Structuring Building Metadata'. In: *SmartWorld'17*. 2017.

[68] Open Community. *Schema Org*. https://schema.org/.

[71] David Martin, Mark Burstein, Jerry Hobbs, Ora Lassila, Drew McDermott, Sheila McIlraith, Srini Narayanan, Massimo Paolucci, Bijan Parsia, Terry Payne et al. 'OWL-S: Semantic markup for web services'. In: *W3C member submission* 22.4 (2004).

[72] Jos de Bruijn, Christoph Bussler, John Domingue, Dieter Fensel, Martin Hepp, Uwe Keller, Michael Kifer, Birgitta Konig-Ries, Jacek Kopecky, Ruben Lara, Holger Lausen, Eyal Oren, Axel Polleres, Dumitru Roman, James Scicluna and Michael Stollberg. *Web Service Modeling Ontology (WSMO)*. [Online; accessed 22. Jul. 2019]. Mar. 2014. URL: https://www.w3.org/Submission/WSMO.

[74] Bharathan Balaji, Arka Bhattacharya, Gabriel Fierro, Jingkun Gao, Joshua Gluck, Dezhi Hong, Aslak Johansen, Jason Koh, Joern Ploennigs, Yuvraj Agarwal, Mario Bergés, David Culler, Rajesh K. Gupta, Mikkel Baun Kjærgaard, Mani Srivastava and Kamin Whitehouse. 'Brick : Metadata schema for portable smart building applications'. In: *Applied Energy* (2018). DOI: 10.1016/J.APENERGY.2018.02.091.

[78] Elena Markoska, Muhyiddine Jradi and Bo Nørregaard Jørgensen. 'Continuous commissioning of buildings: A case study of a campus building in Denmark'. In: *iThings, GreenCom, CPSCom, and SmartData*. IEEE. 2016, pp. 584–589.

[79] Krzysztof Arendt, Ana Ionesi, Muhyiddine Jradi, Ashok Kumar Singh, Mikkel Baun Kjærgaard, Christian Veje and Bo Nørregaard Jørgensen. 'A building model framework for a genetic algorithm multi-objective model predictive control'. In: *REHVA World Congress*. 2016.

[80] Mikkel Baun Kjaergaard, Aslak Johansen, Fisayo Sangogboye and Emil Holmegaard. 'OccuRE: An Occupancy REasoning Platform for Occupancy-Driven Applications'. In: *CBSE'16*. IEEE, 2016. DOI: 10.1109/CBSE.2016.14.

[81] Erik Christensen, Francisco Curbera, Greg Meredith, Sanjiva Weerawarana et al. *Web services description language (WSDL) 1.1*. 2001.

[82] Google Inc. and WeWork Companies Inc. *gRPC: A high performance, open-source universal RPC framework*. https://grpc.io/.

[83] W3C. *OWL*. https://www.w3.org/OWL/. 2012.

[84] W3C. *Turtle*. https://www.w3.org/TR/turtle/. 2014.

[99] Thomas Weng, Bharathan Balaji, Seemanta Dutta, Rajesh Gupta and Yuvraj Agarwal. 'Managing plug-loads for demand response within buildings'. In: *BuildSys'11*. ACM. 2011.

[101] Flexiblepower-Alliance-Network. *EF-Pi*. https://flexible-energy.eu/ef-pi/.

[106] David Huynh, David R Karger, Dennis Quan et al. 'Haystack: A Platform for Creating, Organizing and Visualizing Information Using RDF.' In: *Semantic Web Workshop*. Vol. 52. 2002.

[107] Erik Guttman. 'Autoconfiguration for ip networking: Enabling local communication'. In: *IEEE Internet computing* 5.3 (2001), pp. 81–86.

[108] Alan Presser, Lee Farrell, Devon Kemp and W Lupton. 'Upnp device architecture 1.1'. In: *UPnP Forum*. Vol. 22. 2008.

[109] Erik Guttman. 'Service location protocol: Automatic discovery of IP network services'. In: *IEEE Internet Computing* 3.4 (1999), pp. 71–80.

[110] Steven D Gribble, Matt Welsh, Rob Von Behren, Eric A Brewer, David Culler, Nikita Borisov, Steve Czerwinski, Ramakrishna Gummadi, Jason Hill, Anthony Joseph et al. 'The Ninja architecture for robust Internet-scale systems and services'. In: *Computer Networks* 35.4 (2001), pp. 473–497.

[111] Jim Waldo. 'The Jini architecture for network-centric computing'. In: *Communications of the ACM* 42.7 (1999), pp. 76–76.

CHAPTER 13

ENABLING AUTO-CONFIGURING BUILDING SERVICES: THE ROAD TO AFFORDABLE PORTABLE APPLICATIONS FOR SMART GRID INTEGRATION

This chapter is a cosmetic adaptation of the following conference paper: Jakob Hviid, Aslak Johansen, Fisayo Caleb Sangogboye and Mikkel Baun Kjærgaard. 'Enabling Auto-Configuring Building Services: The Road to Affordable Portable Applications for Smart Grid Integration'. In: *Proceedings of the Tenth International Conference on Future Energy Systems* (*ACM e-Energy '19*). Phoenix, AZ, USA: ACM, June 2019, pages 68–77. DOI: 10.1145/3307772.3328288. **Published**.

The paper was presented at the Tenth International Conference on Future Energy Systems, in Phoenix, AZ, USA, on 26 June 2019.

13.1 ABSTRACT

The transition to green energy is imposing new requirements on the energy sector, such as managing an energy flow through the system that is significantly less predictable than traditional energy sources. Technologies like Demand Response (DR) can help influence the consumer's energy use, but is costly to implement, often to a degree that vastly overshadows the benefits for the consumer and grid operator. Reducing initial investment cost for deployments is, therefore, a considerable factor for the success of DR. One approach to DR is using Building Operating Systems (BOSs) to facilitate the functionality. This paper seeks to reduce the cost of deployments of BOSs, for DR purposes, by creating the tooling needed for automatic configuration of BOS services. Building on top of an existing BOS, tooling is created to support services from the first introduction into the BOS environment,

to the exposure of its functionality to the rest of the system, effectively supporting service discovery and publication at runtime. The tooling is evaluated based on how it impacts the BOS ecosystem, but also the business ecosystem around a BOS. Results show that the tooling requires only minor changes to existing services, and that adding new services to the BOS, using the new tooling, can be achieved with little effort compared to traditional installation methods in the ecosystem. Also, shifting implementation complexity and cost from entrepreneurs to application developers, allows for reducing costs significantly, as application developers target multiple buildings for each implementation. The addition of the new tooling allows the them to potentially create new markets and products.

13.2 INTRODUCTION

Climate change has become a growing concern in the world, and focus has shifted towards green energy to solve large parts of the future energy demands. Based on current research, buildings make up about 40% of total energy consumption [23, 24]. Retail store buildings in Denmark specifically, make up about 8% of the industry sector's energy usage [25]. This usage makes buildings very relevant for addressing grid issues due to the increase in renewables.

The shift to green energy also comes with uncertainty, as most energy sources, such as solar and wind turbines, cannot produce energy with the same predictability as, for example, fossil fuels. The added uncertainty can be a significant problem for the energy companies, as a balance needs to be maintained in the network. One technique to help mitigate this uncertainty is called DR, which is a way to indirectly control or incentivize the energy consumers to use energy at the right times of day, to help stabilize the network [18]. In the United States, DR is also used to stabilize the electricity network, to avoid brownouts.

Unfortunately, previous work on retail stores on DR [1], showed it was evident that the costs of implementing DR functionality in a store would far exceed the cost benefits for the customer. The problem is that retail stores typically work with a relatively short Return on Investment (ROI) of 3 years, while many other building owners make this calculation over 30 years. Commonly also in the USA, the ROI calculations on buildings is just one to three years. During the same

study, it became apparent that the incentive for retail stores should be added functionality, that could save them money elsewhere, and cutting deployment costs. An example of added functionality could be the ability to control several retail stores from a central location, and give statistics, while also diagnosing if the building is performing to specifications. This kind of added value could mitigate a substantial amount of the deployment cost, as most buildings currently are left behind after the initial configuration. Once a BOS [28, 27, 46] is deployed, the cost of implementing controls on top of that system that supported DR would be relatively low cost, which would allow for a more reasonable ROI. A BOS is a specialized operating system that provides a Hardware Abstraction Layer (HAL) and thereby provides a common interface for applications built on top of it.

Building control today is implemented in the form of Building Management Systems (BMSs), which in most cases are closed-loop systems, that only allow for limited re-configuration, and tends to be closed systems external software cannot interface to. Due to the current trend in energy efficiency, and the need for added functionality to reach energy goals, BMSs are found lacking as they cannot adapt to these changing needs. A BOS, in contrast to BMSs, allows for external interfaces and allows applications to extend the functionality of a building. A future system is imagined by several researchers looking into BOS platforms [28, 27, 29], where buildings are platforms living in an ecosystem with several stakeholders developing and selling components for it. One example of a future where entrepreneurs do not just buy a BMS and deploy, but create functionality in the building for the customer. On top of this vision is another ecosystem of developers creating value for customers by creating applications that work in a building context on top of it. An example of this functionality could be DR.

This paper concentrates on solving a part of the cost perspective while leaving the functionality additions mentioned earlier for future work. To solve the issue of deployment costs, a system for deployment of BOSs would need to be developed. We have identified three cost mitigation approaches:

1. Portable applications.
2. Automatic configuration of applications in a BOS.
3. Automatic deployment of application dependencies.

Previous research has already addressed the first cost problem of

portable applications from several perspectives from a HAL [28], metadata [63], ontology descriptions of the hardware structure of the building [73], to service abstraction [3]. For example, in a BOS based on eXtensible Building Operating System (XBOS) [28], Brick can be applied to describe the building using an ontology description of the building and its hardware, only requiring a query to locate the hardware needed, instead of hard-coding implementations for every specific building. Applications and services can also be abstracted in a similar manner using a Service Abstraction Layer (SAL) [3], only requiring a query, similarly to Brick. The SAL is essentially a description of all services in a BOS ecosystem, describing the information exposed, the context, and where to find it. This description is essential for making applications and services not depend directly on each other, but only depend on specific kinds of information instead. Together these technologies allow for building an application that does not need hard-coded references to hardware or software components, thereby enabling portable applications.

The second point of automatic configuration of applications has been mostly enabled by the previous point, portable applications, but lacks several components, specifically, removing the need for BOS maintainers to expand the Brick and SAL model manually. Creating Brick definitions automatically has been made possible, and been implemented by Koh et al. [85], by auto-generating the Brick model from the current BMS data. A tool for integration, benchmarking and development is also provided [93]. Removing the need for manual editing of the SAL is a different problem, as the ecosystem, it describes is based on the services needed for that specific building, and not how the building and its sensors and actuators are placed. To solve this problem, a service hosting the ontologies is needed, that can be dynamically updated by the services themselves, as to make sure they can automatically configure themselves, and announce the functionality the service or application provides to the rest of the system. This paper addresses explicitly this dynamically updatable model enabled by a service and evaluates cost implications for the different stakeholders in the economic ecosystem, and different phases of the BOS life cycle. The third point, automatic deployment of application dependencies, is beyond the scope of this paper.

13.2.1 Problem

The problem addressed in this paper proposes the following question: Currently, the ROI does not justify consumer investments in DR, or even only the supporting BOSs. How can we mitigate this cost?

This paper contributes by proposing the tooling needed for automatic configuration and discovery of services, thereby reducing deployment costs. Also, it contributes by exploring the business ecosystem impact, caused by adding auto-configuration and containing functionality in smaller services.

13.2.2 Related work

Related work relevant for this paper include BOSs, software-based platforms and applications for DR and general methods for service-based systems.

As already motivated and discussed in the introduction the proposed concepts add new capabilities, ties together several existing components. These components include a HAL [28], generation of metadata [63, 85, 93], ontology descriptions of the hardware of the building [73], service abstractions [3] and a specific implementation of BOS [28, 27, 29]. The main contribution of this work is to enable the automatic configuration of applications in the BOS ecosystem and demonstrate this for a DR application.

Ontologies, like the Brick and the SAL, and the accompanying models, need to be hosted by a service that can handle the queries from the system, and return the metadata requested. HodDB [48] is one such service. It is built specifically for Bosswave and Brick and has very well documented performance scores. Unfortunately, it does currently not support the SAL implementation, that is specifically needed to create completely auto-configurable services that can publish themselves into the SAL. Furthermore, HodDB currently does not support dynamically changing models. This limitation is specifically an issue, as a service has no way of registering itself in the SAL automatically if HodDB could host it.

There also exist a number of general software frameworks [100, 112] for DR. The problem with these is that they do not address the portability of applications from building to building and automatic configuration of applications. The reason is their monolithic design, e.g.,

by requiring everything to developed in the same programming environment and a single common data model. In terms of examples of software-based systems for DR we cover related work in Section 13.3 when we discuss different examples of applications.

13.2.3 APPROACH

We wish to amortize the cost across an extended feature set by lowering the cost of introducing new features. We will attempt to accomplish this by introducing a Service Support System (SSS). The SSS will live in the ecosystem described in Figure 13.2, with a hardware layer, HAL, SAL, an application layer, and finally the Bosswave Syndication Bus to support communication and authentication between entities. The system is based on a microservice architecture, that encourages functionality to be implemented as small functional modularized entities. Given a SAL and the BOS context, new tooling will need to be constructed to support automatic configuration of services in the ecosystem of interdepending services. The tooling should support the following process: (1) Locate and retrieve the needed information (data, services) that the application needs. (2) Process the data. (3) Publish results back into Bosswave. (4) Publish the service by describing it in the SAL. (5) Another service should now be able to discover the freshly published service. In support of these goals, the following expectations to the system was identified:

1. Solve the needs a service or application needs to discover other services into the system.
2. Be able to handle dynamically updatable ontology models at runtime to support the SAL.
3. Not compromise the intended functionality of Brick or the SAL.
4. Not require a application developer to maintain Resource Description Framework (RDF) concepts in the SAL.
5. Not require the developer to have a deep understanding of ontologies to publish a service to the SAL.

Therefore, the SSS is developed to host and maintain the Brick and SAL ontologies, and models. SSS maintains the RDF objects and concepts and makes sure only one object of a needed concept exists. Effectively this feature means a service only needs to understand itself when publishing the service, thus reducing the complexity of the setup for

the application developer. RDF and ontology concepts are explained further later in this paper.

13.3 SOFTWARE BASED DR

A range of methods has been proposed for how to implement automated DR in software. A main categorization of the methods can be obtained by considering the type of DR service, the targeted loads and the type of grid integration. For the type of automated DR services, we follow the categorization proposed by Olsen et al. [61]. The categorization includes: **regulation** as a response to random unscheduled deviation; **flexibility** as an additional load following reserve for large un-forecasted wind/solar ramps, **contingency** for rapid and immediate responses to a loss in supply, **energy** to shed or shift energy consumption over time and **capacity** to provide an alternative to generation. The targeted loads can be categorized by their broad type including space cooling, space heating, water heating, electric vehicles, indoor lighting, ventilation, refrigeration, pumps (e.g., pools and wastewater), computing equipment, manufacturing equipment, building battery among others. The main differences among these loads are how fast they can ramp-on/off and if they have a storage capacity. The type of grid integration captures the integration between the role of the grid operator and the building owner. This can at the one extreme be a centralized direct control with the grid operator in charge and at the other end be an in-direct control by price signals where the building owner decides on load activation. In between is different collaborative schemes that optimize the benefits of offering and activating DR between the grid operator and building owner.

Table 13.1 list our classification of recently published methods. The listed methods span the different types of DR services, loads and grid integrations. A particular building owner to optimize cost benefits of participating in DR will run several of such methods at the same time to participate in different services. The services will also change over time depending on changing grid situations and thereby economic incentives for delivering different DR services. Considering the proposed methods, this highlights the many different DR services that will be running on future BOS implementations for buildings. Furthermore, each of the listed methods in Table 13.1 utilize a range of other

```
{
    "spatialcoverage": "OU44 Building",
    "temporalgranularity": "1 Minute",
    "sensormodalities": "Camera Counter",
    "spatialgranularity": "Room",
    "informationtype": "Counts",
    "temporalcoverage": "Real-Time"
}
```

Figure 13.1: Building Stream Configuration.

services for accessing real-time and archival building data, executing control, forecasting services, external services (e.g., weather forecasts, energy price signals). However, this is in contrast to how most applications for buildings today are built as monolithic applications where all analysis, processing and GUI functionality is contained in one or few applications.

13.3.1 OCCUPANT AWARE DR CASE STUDY

An example of a DR application is [20] which combines different types of data and models to schedule DR actions. The central element here is a DR Orchestrator role which is to collect the information needed from the BOS, to take action, and decide on how to react to electricity price changes or a DR event. The application used in this paper has been simplified compared to the full system presented by Nellemann et al. [20], as it does not collect all the information needed to orchestrate a real DR event. Instead, it queries for the information provided by the OccuRE implementation, to confirm a functioning auto-configuration setup for the OccuRE microservice. To evaluate a case that fits with how a future ecosystem could be used in practice, the occupancy prediction framework, OccuRE [80] was adapted to work in the context of the BOSs, together with a sample application taking the role of the DR Orchestrator.

The need for occupancy information for different DR applications differs significantly in terms of information type (such as presence, counts, traces, identity, and activities), spatial granularity, temporal granularity, spatial coverage, and temporal coverage [96, 80, 113]. These combinations of specifications also mandate that different sensor modalities for

	DR Service Type	Targeted loads	Grid / Building Roles
Vishwanath et al. [53]	energy	space cooling	in-direct price signals
Neupane et al. [54]	energy, flexibility	heat pumps, electric vehicle, household appliances	collaborative direct control
Hajiesmaili et al. [55]	energy, flexibility	electric vehicle, building battery	centralized direct control
Xia et al. [56]	energy, flexibility	residential loads under user control	decentralized in-direct control
Comden et al. [57]	energy, flexibility	residential loads under user control	decentralized in-direct control
Frazetto et al. [54]	energy, flexibility	household appliances	collaborative direct control
Barth et al. [58]	energy, flexibility	industrial processes	decentralized direct control
de Hoog et al. [59]	energy, flexibility, contingency	building battery	centralized in-direct control
Khonji et al. [60]	energy, flexibility	electric vehicles	centralized direct control
Nellemann et al. [20]	energy, flexibility	ventilation	collaborative direct control

Table 13.1: Automated DR methods categorized by DR service type, targeted loads and grid/building roles.

obtaining relevant occupancy data and processing strategies for computing the necessary occupancy information are required. However, the Achilles heel for proffering the relevant occupancy information or meeting the specifications of a DR application lies in the ability of an occupancy system to resolve the relevant sensor modalities and processing strategies that can satisfy the specification presented by the DR application. A typical example of a specification in the form of a building stream configuration is given in Figure 14.10.

OccuRE [80] is a platform that addresses these concerns, and it provides a unified interface in the form of a REST Application Programming Interface (API) for domain application to query and retrieve relevant occupancy information as part of a BOSss application. OccuRE is comprised of two major system components namely a sensor data resolver and strategy resolver. Given a particular domain specification in the form of a building stream configuration, the responsibility of the sensor resolver is to request Metadata information from a `Metadata broker` about a list of available sensor streams data that are relevant for that information. Also, given a number of registered processing strategies, the role of the strategy resolver is to search a directed graph that links sensor data to occupancy information for the processing strategy best suited for computing the requested configuration. As the OccuRE platform can provide the information needed by DR applications and is currently coded as monolith it is a good case for evaluating the benefits of the proposed solution in this paper.

13.4 DESIGN

Designing a system for exploring the automatic configuration of building services, requires an ecosystem to test and develop in. Figure 13.2 explores this setup, and details the new components in such a setup. The diagram consists of several components. Horizontally in the top, the hardware layer, HAL, SAL and Services layers are defined. The hardware layer contains the actual sensors and actuators that make up the instrumentation of the building. This hardware is integrated into the BOS using drivers. These drivers are adapting the hardware interfaces to a common interface that interacts on the Bosswave Syndication Bus. For now, Bosswave is a communication bus between drivers, and services, but is detailed further below. On the right of

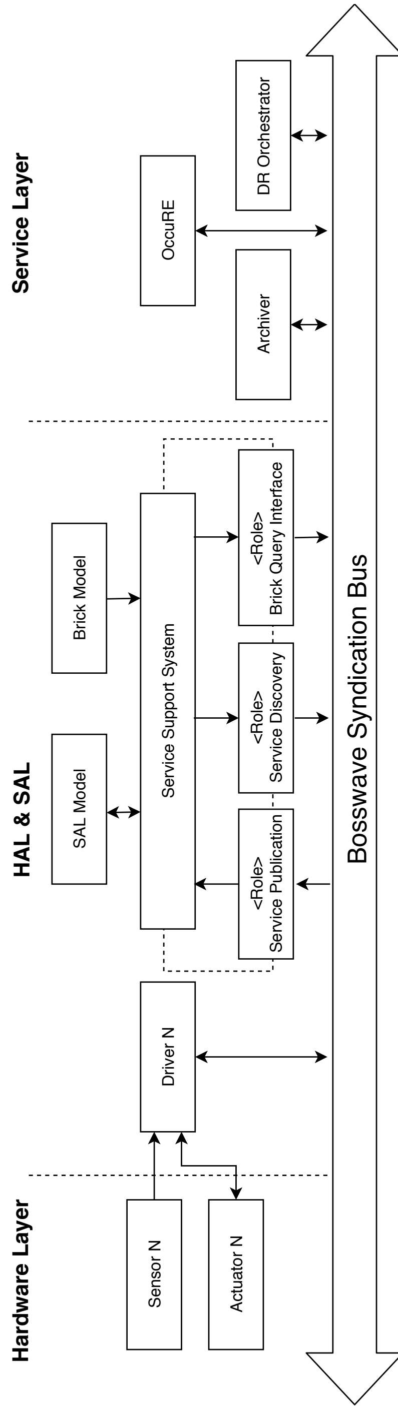

Figure 13.2: Microservice Ecosystem Example.

the diagram, the services layer, or application layer is present. This layer contains the three microservices of which two was mentioned earlier, an Archiver, OccuRE, and the DR Orchestrator. The Archiver is a microservice holding historical data from sensors, and potentially also services. In the middle, the system designed to help services auto-configure itself is present. The system consists of several models, and roles. The roles are as follows, Service Publication, Service Discovery, and the Brick Query Interface. Service Publication enables services to publish their endpoints, and make themselves known to the rest of the BOS ecosystem. The Service Discovery Role enables finding services using a SPARQL Protocol and RDF Query Language (SPARQL) Query. Finally, the Brick Query Interface allows for querying for the hardware and location context of the building, also using SPARQL Queries. All of the roles are managed by the SSS, which also hosts the ontologies and models of the SAL and Brick. An ontology is the description of the possible building blocks a model can consist of, while the model is the concrete implementation and use of the ontology. To clarify, this means that the SAL ontology, for example, defines the Service and ServiceEndpoint types, but the SAL model contains the specific instances of Services and ServiceEndpoints.

When working with BOSs in an XBOS [28] context, and its current state, Bosswave [47] is a technology that is important to understand. Bosswave is an overlay network situated on the internet that provides secure communication between IoT devices and building applications, using technologies like blockchain and service bus. Bosswave uses the blockchain and its inherent smart contracts to encrypt the communication between agents that maintain the network and blockchain. These agents, in turn, facilitate the communication for the devices and services that operate on Bosswave. This encryption is enforced using a public key that is published in the blockchain, and a private key that only the creating user has access too. The blockchain holds entities that can be users, wallets, domain owners and more, and maintains the trust these entities have defined between each other. This trust effectively implements permission functionality, that is enforced when entities want to read or write to the network. All traffic going through Bosswave is communicated through the Bosswave Syndication Bus and is attached to a Uniform Resource Identifier (URI), identifying a specific resource. Bosswave's syndication bus acts as the HAL interface to the application layer, and together with Brick, it decouples applic-

ations from the building hardware. The URI is a path that describes the composition of data streams and their context. The permission sets, or Declaration of Trust (DoT), then define what parts of the URI an application can read or write to, or if they can delegate trust to other entities.

13.4.1 Ontologies

The entire setup is using ontologies to enable the portability of services and applications between buildings. However, what are ontologies in this context? Ontologies like Brick and the SAL is based on the RDF [62]. RDF is a method that can be used to describe relationships between data in a way that can be read easily by a computer. RDF is also the foundation for enabling the semantic web. Brick is an example of an ontology built on RDF and is typically available in the Turtle [84] file format, identifiable by the "ttl" file extension. The SAL also uses RDF and Turtle but additionally uses the W3C Web Ontology Language (OWL) [83] on top of RDF. OWL introduces several new concepts on top of RDF, like classes and individuals. Classes refer closely enough, for the purposes of this paper, to the concept of classes from object-oriented languages, while individuals map closely to the concept of instances of a class.

Brick [73, 74] is, as mentioned before, an ontology describing the physical properties of a building. It was explicitly designed to represent building hardware and relationship description and was not intended as a multipurpose metadata description framework, like for example Haystack [106]. The central goal of Brick is to describe and query for relationships between hardware in a building, and thereby understand the components of the building and how they relate. These queries can then be integrated with applications, allowing for a query it to discover the building hardware and the context in which it is placed. As the queries are dynamic per building and its Brick model, it allows the application to be built for more efficient portability between buildings. The model, or the specific implementation of Brick, can take a long time to make, and how a building is described similarly each time, is one of the main problems with this approach to building metadata discovery.

From the concept of Brick and the recognition that the abstraction from the hardware layer is vital for portable applications, the idea of a SAL [3] evolved. The SAL is solving the same problem for services and

applications, that Brick is solving when abstracting from the hardware, and physical layout of a building. The SAL describes the interfaces and connection points of a service using an ontology, that allows another service or application to query for the information it needs from another unknown service. This functionality enables decoupling of a service or application from other services, as a service can now be discovered through querying. Effectively, it means services can now depend on information, rather than specific services. With the SAL defined by Hviid and Kjærgaard, the description of the service endpoints requires agreement on what a name or topic is, or how it is defined. This agreement is a limitation on how the SAL was implemented. Unfortunately, it does not describe everything a service needs, to read the information, as the data format needs to be agreed upon beforehand. Therefore, the SAL is used as a starting point for the work performed in this paper, but with extended functionality, to support independent service discovery and interfacing.

Figure 13.3 describes a JavaScript Object Notation (JSON) object used to create a new service with one endpoint in the SAL. How this creation of a service works with the SAL is described in the next section called, Service Publication. The description of the SAL is significantly different from the original version by Hviid and Kjærgaard. Just as the previous work, a service has a name, and multiple endpoints, but now the endpoint's contained information is described by a modality, a unit, its temporal aspect, and how it related to a physical location in Brick. Modality examples could be values like Occupancy or Temperature, while Unit examples could be Count or Celsius. Together they describe the context of the information provided. The temporal aspect describes if we are talking about archived data, real-time data, or predictions. The location property of the array of information objects is an indicator of where in the resource described, the information can be found. The example shown in the figure points to a "count" property in the object to be read from Bosswave at the given URL. This property effectively allows for a description on to read the discovered information from the JSON object without having agreed upon a specific naming of properties in the data structure. In this way adding multiple information objects can add to the discoverability of specific parts of the published information. As the JSON object described in Figure 13.3 is a data model of the ontology represented in JSON for a specific task. It does not represent the complete changes to the ontology, and the changes

will not be discussed further in this paper as they are not part of the main contribution, but merely a needed understanding of the context in which the Service Publication role works.

13.4.2 SERVICE PUBLICATION

For the SAL operations, a high-level approach is taken, where the client – who wants to publish a service – describes the service in a JSON-based format. This description spans all relevant parts of the SAL, namely (1) the service individual containing the service name, (2) the endpoint individual representing a single network endpoint, and (3) the information individual detailing which kinds of data are involved.

The JSON is marshaled using a natural nesting which allows on service individual to be associated with any number of endpoint individuals. These, in turn, may be associated with any number of information individuals. A Brick location entity is used for referencing the location of the information individual, and an organization name is used in the endpoint individual to reference a schema.org organization.

To make this work in a Bosswave context, we wrap the publication request in an object indicating a result path for where the result of the publication should be published. By subscribing to this result path, a client will receive a notification of the completion of the operation. Figure 13.3 shows a full example of how the modified version of OccuRE publishes its services.

13.4.3 SERVICE DISCOVERY

Discovery is implemented using regular SPARQL queries over the underlying RDF store. Using a lower abstraction allows greater flexibility while expressing queries, but requires knowledge of the underlying RDF structure and the querying language itself.

To express such a query, knowledge of how the main classes of the SAL are mapped to RDF is needed. The three main classes are Service, ServiceEndpoint and Information. Figure 13.4 illustrates how these – and the external Schema.org Organization and Brick Location – are connected using OWL data and object properties. This query extracts all published services.

```
{
  "result-path": "a20856b9-********",
  "service": {
    "name": "OccuRE",
    "hasServiceEndpoint": [
      {
        "url": "jah.demo/occure/a20937b9-********",
        "read": true,
        "write": false,
        "priority": "1",
        "ownedBy": {
          "legalName": "University of Southern Denmark"
        },
        "providesInformation": [
          {
            "location": "Count",
            "hasModality": "Occupancy",
            "hasUnit": "Count",
            "hasTemporalAspect": "Real-Time",
            "hasLocation": "model:rooms.e20-601b-2"
          }
        ]
      }
    ]
  }
}
```


Figure 13.3: Request for service publication.

```
SELECT ?org_name ?service_name ?sep_url ?loc ?modality ?unit
       ?ta ?bloc
WHERE {
    # main individuals
    ?service rdf:type/rdfs:subClassOf* sal:Service .
    ?sep     rdf:type/rdfs:subClassOf* sal:ServiceEndpoint .
    ?info    rdf:type/rdfs:subClassOf* sal:Information .

    # external individuals
    ?org     rdf:type/rdfs:subClassOf* schema:Organization .
    ?bloc    rdf:type/rdfs:subClassOf* brick:Location .

    # object properties
    ?service sal:hasServiceEndpoint  ?sep .
    ?sep     sal:providesInformation ?info .
    ?sep     sal:ownedBy             ?org .

    # data properties
    ?service sal:name               ?service_name .
    ?org     schema:legalName       ?org_name .
    ?sep     sal:url                ?sep_url .
    ?info    sal:location           ?loc .
    ?info    sal:hasModality        ?modality .
    ?info    sal:hasUnit            ?unit .
    ?info    sal:hasTemporalAspect ?ta .
    ?info    sal:hasLocation        ?bloc .
}
```

Figure 13.4: Query for listing all published services.

13.4.4 Data Plane / Service Support System

To support the service publication and discovery operations, a server that wraps pythons rdflib module was implemented. It hosts a model spanning both Brick and SAL. The service publication operation is translated to operations on the RDF store of the model. Interfaces for this operation – along with one for a generic SPARQL resolver – are exposed through separate services, each with a corresponding path in Bosswave. The generic SPARQL resolver interface is used for service discovery.

During the translation process of publication, triples have to be added to the store. However, some of those triples may already exist. To avoid duplicate entries, we employ a strategy of first attempting to look up an entity matching what we are about to create. If it exists we reuse it; otherwise, we create it.

13.5 OccuRE Redesign

In the original implementation of the OccuRE framework, the sensor resolver module is tightly coupled with the `Metadata Broker`. The `Metadata Broker` uses an `SQL`-like query language to request the list of sensor data over the REST API of a sMAP [46] archiver. The strategy resolver, on the other hand, uses a directed graph that maps the type of sensor stream data to occupancy information for resolving the relevant processing strategy to compute the required occupancy information.

Using the proposed service model, the OccuRE framework can now fully leverage the capabilities of the SAL model for its subsystems and as an orchestrator. First, each of the original OccuRE processing strategies are redesigned and published as microservices. Then, the OccuRE orchestrator utilizes a new service discovery module to resolve the relevant occupancy microservice for a particular building stream configuration. In the current implementation of each occupancy microservice representing an OccuRE strategy, each occupancy microservice queries the Brick Schema for the relevant sensor stream data that can satisfy the occupancy information to be computed. The sensor stream data is subsequently used to retrieve occupancy data from each sensor and/or archival system. As a demonstration, an OccuRE strategy for estimating occupancy count information from camera counters in real-time namely PreCount [114] is adapted for using the SSS. The

original PreCount strategy uses and implements four main modules namely `dataservice`, `publisher`, `predictor`, and the `main` module. The `dataservice` module implements all data communication for the strategy, and it implements a REST API to the `Metadata broker` and the archival system. The `publisher` module also implements a REST API for publishing the computed occupancy count. The `predictor` module implements a series of data transformation and machine learning algorithms for computing occupancy count in real-time. The `main` module implements orchestration functionality that calls the remaining modules and a hard-coded mapping of the cameras sensor streams data.

```
(0)     SELECT  *
        WHERE   Metadata/Form = 'cameracount'
                AND Metadata/Building = 'OU44'


(1)     SELECT  *
        WHERE   Metadata/Value/Type = 'Estimation'
                AND Metadata/SourceName = 'OccuRE Estimation'
                AND Metadata/Building = 'OU44'
```

Figure 13.5: sMAP Queries for Resolving Sensor Stream Data.

Given the building stream configuration in Figure 14.10, the original implementation of PreCount utilizes the sMAP queries given in Figure 14.11 to query all camera count sensors stream data through the `dataservice` module. Subsequently, because of some limitations with regards to the representation of the placement of camera sensors within buildings, the `main` module implements a hard-coded mapping that follows the problem formulation in [115] about the source and destination of the countlines defined in the deployed camera sensors. This hard-coded mapping requires a ground truth knowledge of the directionality of these countlines and thus impedes scalability to unknown buildings. Leveraging the capabilities of the Brick schema, the directionality of the countlines can be obtained without prior knowledge of the ground truth for any given building using the SPARQL query in Figure 13.6. Table 13.2 highlights a pre-processed sensor stream data from the SPARQL query in Figure 13.6 and due to the confidentiality

```
SELECT ?camera_name ?line_name ?smap_uuid ?bw_path
        ?src_name ?dst_name
WHERE {
    ?camera     xovis:name ?camera_name .
    ?line       xovis:name ?line_name .
    ?src        rdfs:label ?src_name .
    ?dst        rdfs:label ?dst_name .

    ?camera     xovis:hasLine ?line .
    ?line       xovis:observes ?connection .
    ?line       xovis:hasDirection ?direction .

    ?src        xovis:feedsOccupants ?direction .
    ?direction  xovis:feedsOccupants ?dst .

    ?direction  bdsmap:hasData/bdsmap:uuid ?smap_uuid .
    ?direction  bdbw:hasData/bdbw:path     ?bw_path .

    ?camera     rdf:type/rdfs:subClassOf* xovis:XovisCamera .
    ?line       rdf:type/rdfs:subClassOf* xovis:CountingLine .
    ?direction  rdf:type/rdfs:subClassOf* xovis:Direction .

    {
        ?src    rdf:type/rdfs:subClassOf* brick:Room .
    } UNION {
        ?src    rdf:type owl:Class .
        ?src    rdfs:label "outside"
    } .

    {
        ?dst    rdf:type/rdfs:subClassOf* brick:Room .
    } UNION {
        ?dst    rdf:type owl:Class .
        ?dst    rdfs:label "outside"
    } .
}
```

Figure 13.6: SPARQL Query for Resolving Sensor Stream Data.

of the `smap_uuids` and `bw_paths`, we have excluded these results from the table.

	camera_name	**line_name**	**src_name**	**dst_name**
1	xovis13	ZoneModel:C:None	Space 1	outside
2	xovis11	ZoneModel:C:None	Space 2	outside
...	...	...	...	...
8	xovis14	ZoneModel:C:None	Space 8	outside
9	xovis13	ZoneModel:C:None	outside	Space 1
10	xovis11	ZoneModel:C:None	outside	Space 2
...	...	...	...	...
16	xovis14	ZoneModel:C:None	outside	Space 8

Table 13.2: Results from SPARQL Queries in Figure 13.6.

Given the returned `bw_paths`, the `dataservice` module can subscribe to the `bw_paths` of the countlines, and it can aggregate the transition data from all countlines to compute both the transition and cumulative counts from the building in real-time. Similarly, the `dataservice` module can query historical count data with the `smap_uuids` to compute the historical transition and cumulative counts. These historical datasets are used by the `predictor` to train a predictive algorithm, and the real-time datasets are used as inputs to estimate the occupancy counts in the building in real-time. Instead of the REST API used in the original `publisher` module, the `publisher` module will publish estimated occupancy counts using the `url` of the service publication request in Figure 13.3.

Figure 14.9 highlights the described sequence of interaction between the modules. In Table 13.3, we present a comparison between the original implementation and the refactored code for the SSS adaptation of PreCount. For all PreCount modules, we achieved a 48% reduction of the original codebase. The various services provided by the SSS provides better tooling for developers to efficiently write more maintainable codes with less overhead. For example, the capabilities afforded by the Brick model in the SSS eliminates the hard-coded mapping representing the ground truth about the directionality of the countline for each building. This elimination resulted in the significant difference of 80% between the original implementation of the `main` module and the implementation using the SSS and consequently, the portability of the PreCount Strategy for any given building.

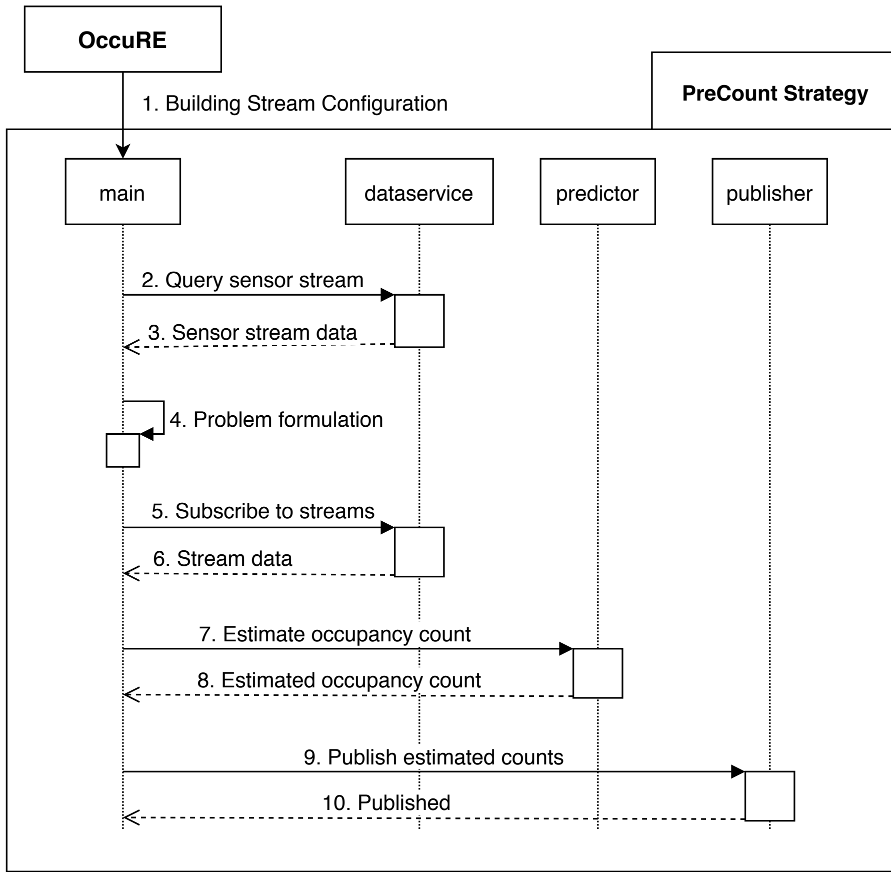

Figure 13.7: Sequence of interaction between the PreCount Modules.

At this stage, when OccuRE is published in the SAL, the DR Orchestrator is able to query the SSS for the information it needs from OccuRE. Afterwards, it subscribes to the Bosswave URI received from the SAL, and begins to orchestrate its DR tasks.

13.6 ECOSYSTEM IMPACT ASSESSMENT

Based on the system described above, several factors becomes interesting to analyze. First, who are impacted by these changes, and second how do the changes impact the ecosystem and cost perspectives. The three different stakeholders identified in this evaluation are the cus-

Modules	Lines of Code	
	Original Implementation	Service Adaptation
dataservice	274	216
publisher	40	56
predictor	126	126
main	546	112
Total	**986**	**510**

Table 13.3: Comparison between the Original Implementation and the Refactored Code for the Service Support System Adaptation.

tomer, the entrepreneur, and finally the application developer. The customer is the maintainer of the actual on-site system, and the stakeholder making purchase decisions and sets the goals and rules for a company energy strategy. An entrepreneur is a company assigned to build or to retrofit a building. The application developer is defined as a company entity that develops applications, and services, that goes on top of the BOS. In the current business ecosystem, this stakeholder is mostly nonexistent. An example of an application developer could be an energy company implementing DR applications. Each role is not exclusive, so as an example, an entrepreneur can also be an application developer.

The overall ecosystem expected from a portable application ecosystem, especially also one with package dependency support in the future, is one where third-party developers can build applications for BOSs, just like they can for a phone with an app store. Having packages, or an app store would also change the way entrepreneurs deploy systems.

If a customer stakeholder wants to retrofit or create a new building, an entrepreneur stakeholder builds it for them. As this paper tries to mitigate the costs of deployment, the costs of the entrepreneur should be reduced, and moved to the application developer. The reason for this; their applications are portable and therefore only implement changes once, and therefore this is a more economical choice for the entire ecosystem. This represents an added value, and thus, makes it easier for customers to buy into the proposed ecosystem. A DR orchestrator can be trivially installed in such an ecosystem as it will take the form of an application, thereby removing the DR-specific cost in the cost-benefit analysis. Also, This change enables an ecosystem of applications

developers, and can potentially create new markets.

	Customer	Entrepreneur	App. Developer
Analysis	- Potential integrations with workflows	- Visible if customer demands can be met - Modularized upselling	- Defined subfield for App. - Make depending modules - Platform benefit, more users same code
Design	- Modularized options	- Less design time used (modules) - Complexity moved to app. dev.	- Added complexity
Implementation	- Cost shifting benefits	- Cost shifting benefits - Less design time - More documentation time for Models	- Added complexity - Application portability implementation
Deployment	- Less deployment time	- Lower deployment time - Potential for containerized deployment	- Can be containerized - Simplify application delivery
Operational	- Less cost to upgrade later - Deep abstracts are more complex - Simpler abstractions	- More aftermarket service options - Less maintenance due to autoconfig modules - Easy upgrade of apps/services	- Easier to market defined functionality modules - More difficult to debug models

Table 13.4: Costs impacts, benefits and disadvantages associated with each phase, distributed relevant across stakeholders.

Table 13.4 shows the different stakeholders mentioned before, as well as the different phases of a BOS, including the development life-cycle, depending on stakeholder. The phases concentrate on cost impacts, and are defined like this: (1) Analysis: Analyzing the problem, and if a solution should/can be built. (2) Design: Concentrating on designing a viable solution for the problem at hand. (3) Implementation: Building the software, or system needed. (4) Deployment: Deploying the software, be it on-site, or in a software repository, depending on the stakeholder. (5) Operational: Keeping the building and software operational, including potential after-marketing opportunities.

One of the main benefits of automating BOS service and application installations is cost shifting. Cost shifting is the idea of moving costs from one part of the business ecosystem to another, where the overall benefit is better for all, or most, stakeholders involved.

Moving to auto-configured services also introduces ideas of modularizing services and applications, and even containerizing them with options like Docker [87]. This change forces a thought pattern

of functionality packages, that are easier to sell for the entrepreneur and to understand for the customer. As a side effect of changing to auto-configurable modules, a potential for a market for third-party developers is created, which can find niche areas, or integrate their product suites cost-effectively into building ecosystems. For both entrepreneur and application developer, this mindset of modularizing creates clear packages of functionality to market.

13.6.1 The Entrepreneur

With a modularized system that is auto-configurable, it is easier for an entrepreneur to know if they can meet customer demands in the analysis phase. This clarity is due to how a modularized and auto-configurable system changes the approach to functionality, and how clear this function is defined. It also brings a different benefit; the potential up-selling. With a modularized system, it is easier to offer specific functionality to a customer, as each function is well defined, and to know the price of implementation and deployment. Also, clear functionality descriptions, for both services and applications, make it possible to create a service and upgrade after-market.

For the entrepreneur implementation costs get drastically lower, as microservices are deployed that does not need configuration, other than Brick which potentially can be auto-generated, by using the tool called Scrabble made by Koh et al. [85]. Koh et al. even provide a benchmark and development framework to help to test and to integrate with the brick model, called Plaster [93]. Fundamentally, a part of the cost of deployment is shifted to the application developer, as the burden of configuring the application is taken over by the application developer. Also, fewer application developers would need to work at the entrepreneur company to get a viable product, as less configuration and functionality needs to be developed.

The Brick and SAL model could signify a cost increase though, as time is needed to describe the hardware configuration. If the entrepreneur is retrofitting a building, as opposed to creating an entirely new system, a part, or all of the Brick model can be generated automatically, as mentioned earlier. Third-party application developers would develop most of the applications unless the entrepreneur also acts as an application developer.

13.6.2 The application developer

For the application developer, there is an initial complexity added to the development process, but also added opportunities to make the entrepreneurs dependent on their services, as less software development expertise is present at the entrepreneur companies. The initial complexity addition is mostly centered around the concepts of SPARQL [69], RDF [62], OWL [83], Brick [74, 73], and the SAL [3]. For a developer though, these concepts are relatively easy to learn. Some cost though, will go into understanding how the entrepreneurs are creating and building the Brick models, so queries to it work across buildings. This understanding is essential for the application developer, as it is what provides the main benefit of targeting multiple buildings at once.

A potential new market that does not exist currently for BMSs creates an opportunity to create services that add specific functionality, like data analysis tools, that other developers can use. This change means that other application developers become possible customers too. To do this effectively, a package manager that can install dependencies would need to be created, as it would support this kind of business ecosystem.

With applications modularized as separate functional services and packaged for auto-configuration, the next obvious move would be to containerize the application for the package manager, for simplified delivery. Containerization is an option for the developer, not a requirement for this work.

13.6.3 The customer

When analyzing the impacts on the customer, several points are worth mentioning. First, the initial deployment costs should be reduced due to cost shifting, by moving complexity, from the entrepreneur to the application developer. This cost reduction is achieved as the application developer develops for multiple building types, not for specific buildings. Also, the actual deployment time of the system should be significantly shorter, because it is easier to set up, resulting in lower pricing of the installation.

As addressing a BOS from a modular functionality perspective with automatic configuration is more straightforward for all stakeholders to understand from a business perspective, it is easier to upgrade the BOS

functionality in the future, or retrofit old buildings to support specific functionality. For the customer, the abstractions of functionality and modules are easier to understand and buy into, than buying entire new systems. This is where a DR application comes in to play, as it is easier to calculate a defensible ROI.

Adding all the different technologies on top of a traditional BMS, like Brick, SAL, and more, also brings complexity. So, for the customer it is easier to understand the abstractions they need, but more difficult to understand the depth of the system itself if they should need to implement functionality by themselves in the future.

13.7 Service Support System Assessment

In the approach section of this paper, several factors the SSS should support were defined. The first set of requirements were functional, or evaluative. First is service discovery, which allows for a service or application to find the information it needs; information which is being provided by another service. Second is being able to publish services into the SAL dynamically at runtime. The third is demonstrating the above two works as intended. All three factors were demonstrated by the case study, which implemented a new version of OccuRE, and the SSS.

Other non-functional requirements defined, was that the tooling implementation should not compromise the intended functionality of Brick or the SAL, that a developer should not maintain the RDF concepts related to the SAL, and finally, that publishing to the SAL should not require RDF knowledge. JSON is used to publish, and only requires a simplified understanding of the SAL. It was considered if this kind of abstraction should also be applied to the service discovery role of the SSS, but it was decided against as SPARQL would already be needed to read from the Brick and SAL ontology models. Brick and SAL were hosted by the SSS and allowed for standard Brick, and SAL queries using SPARQL, thereby upholding the intended functionality of the two ontologies. The SAL was modified but did not compromise the intent of the SAL. Instead, it improved on it and did not change its intended functionality. Also, the SSS maintains the RDF concepts for the SAL, thereby not requiring application developers to maintain these.

The OccuRE implementation demonstrated the effectiveness of the platform, as a 48% reduction in code was achieved, and in a specific case even a 80% reduction. Therefore, all of the above requirements were satisfied with the design choices made, except for the above issues. Also, the platform allowed for removing hard-coded data from OccuRE, that was previously not discoverable.

13.8 CONCLUSION

Tooling for a BOS was created and supports automatic configuration and publishing of services into a BOS based on XBOS, Brick and the SAL. The tooling, in the form of the SSS that implements the service discovery and publication roles, successfully allows services to configure themselves into a new environment, and publish their functionality. Also, Table 13.3 demonstrated a significant reduction in lines of code totaling 48%, or in a specific case 80%. These changes enables entrepreneurs and application developers to reduce deployment costs.

In regards to the ecosystem impacts, the changes are generally favorable for all stakeholders in the ecosystem. The application developer is the stakeholder that has the most cost shifted to them, but the creation of new market opportunities should offset the relatively low addition of cost in work hours.

In the introduction, three areas were identified that would need to be solved to reduce deployment costs of BOSs. The first, Application Portability, was already mostly solved by the HAL, Brick, and the SAL. The second was auto-configuration of services that this paper has addressed. The third, and last, automatic deployment of application dependencies, is still to be addressed. This third addition could be enabled by a package management system, that depends on output types already defined in the SAL. This package manager would essentially evaluate the informational needs of a package, and could recursively install all needed services. A BOS package manager would install a functionality needed and automatically deploy all required subsystems. This functionality, however, is still to be implemented and explored further.

REFERENCES

[1] Jakob Hviid and Mikkel Baun Kjærgaard. 'The Retail Store as a Smart Grid Ready Building: Current Practice and Future Potentials'. In: *Power & Energy Society Innovative Smart Grid Technologies Conference (ISGT), 2018 IEEE*. IEEE. Washington DC, USA, Oct. 2018. DOI: 10.1109/ISGT.2018.8403354. **Published**.

[3] Jakob Hviid and Mikkel Baun Kjærgaard. 'Service Abstraction Layer for Building Operating Systems: Enabling portable applications and improving system resilience'. In: *2018 IEEE International Conference on Communications, Control, and Computing Technologies for Smart Grids (SmartGridComm)*. IEEE. Aalborg, Denmark, Oct. 2018, pp. 1–6. DOI: 10.1109/SmartGridComm.2018.8587543. **Published**.

[5] Jakob Hviid, Aslak Johansen, Fisayo Caleb Sangogboye and Mikkel Baun Kjærgaard. 'Enabling Auto-Configuring Building Services: The Road to Affordable Portable Applications for Smart Grid Integration'. In: *Proceedings of the Tenth International Conference on Future Energy Systems (ACM e-Energy '19)*. Phoenix, AZ, USA: ACM, June 2019, pages 68–77. DOI: 10.1145/3307772.3328288. **Published**.

[18] Mary Ann Piette, Sila Kiliccote and Girish Ghatikar. 'Field Experience with and Potential for Multi-time Scale Grid Transactions from Responsive Commercial Buildings'. In: *ACEEE Summer Study on Energy Efficency in Buildings*. 2014.

[20] Peter Nellemann, Mikkel Baun Kjærgaard, Emil Holmegaard, Krzysztof Arendt, Aslak Johansen, Fisayo Caleb Sangogboye and Bo Nørregaard Jørgensen. 'Demand Response with Model Predictive Comfort Compliance in an Office Building'. In: *SmartGridComm'17*. IEEE. 2017.

[23] European Parliament and Council of the European Union. 'Directive 2010/31/EU of the European Parliament and of the Council on the Energy Performance of Buildings'. In: *Official Journal of the European Union* L153 (May 2010), pp. 13–35. URL: http://eur-lex.europa.eu/eli/dir/2010/31/oj.

[24] U.S. Department of Energy. *2011 Buildings Energy Data Book*. Tech. rep. U.S. Department of Energy, Mar. 2012. URL: https://catalog.data.gov/dataset/buildings-energy-data%20-book.

[25] Danmarks Statistik. *ENE3H: Bruttoenergiforbrug i fælles enheder efter branche og energitype [Accessed: 23/2/2017]*. http://www.statistikbanken.dk/ENE3H. 2017.

[27] Stephen Dawson-haggerty, Andrew Krioukov, Jay Taneja, Sagar Karandikar, Gabe Fierro, Nikita Kitaev and David Culler. 'BOSS: building operating system services'. In: *NSDI '13* (2013), pp. 1–15.

[28] Gabriel Fierro and David E Culler. 'XBOS: An Extensible Building Operating System'. In: *Proceedings of the 2nd ACM International Conference on Embedded Systems for Energy-Efficient Built Environments*. ACM. 2015, pp. 119–120.

[29] Thomas Weng, Anthony Nwokafor and Yuvraj Agarwal. 'Buildingdepot 2.0: An integrated management system for building analysis and control'. In: *BuildSys'13*. ACM. 2013.

[46] Stephen Dawson-Haggerty, Xiaofan Jiang, Gilman Tolle, Jorge Ortiz and David Culler. 'sMAP: a simple measurement and actuation profile for physical information'. In: *SenSys'10* (2010). DOI: 10.1145/1869983.1870003.

[47] Michael P Anderson, John Kolb, Kaifei Chen, David E Culler, Randy Katz, Michael Andersen, John Kolb, Kaifei Chen, David E Culler and Randy Katz. 'Democratizing Authority in the Built Environment'. In: *BuildSys*. 2017.

[48] Gabriel Fierro and David Culler. 'HodDB: Design and Analysis of a Query Processor for Brick.' In: *IEEE BuildSys '17* (Nov. 2017).

[53] Arun Vishwanath, Vikas Chandan, Cameron Mendoza and Charles Blake. 'A Data Driven Pre-cooling Framework for Energy Cost Optimization in Commercial Buildings'. In: *e-Energy 2017*. 2017, pp. 157–167.

[54] Bijay Neupane, Laurynas Siksnys and Torben Bach Pedersen. 'Generation and Evaluation of Flex-Offers from Flexible Electrical Devices'. In: *e-Energy 2017*. 2017, pp. 143–156.

[55] Mohammad Hassan Hajiesmaili, Minghua Chen, Enrique Mallada and Chi-Kin Chau. 'Crowd-Sourced Storage-Assisted Demand Response in Microgrids'. In: *e-Energy 2017*. 2017, pp. 91–100.

[56] Bainan Xia, Hao Ming, Ki-Yeob Lee, Yuanyuan Li, Yuqi Zhou, Shantanu Bansal, Srinivas Shakkottai and Le Xie. 'EnergyCoupon: A Case Study on Incentive-based Demand Response in Smart Grid'. In: *e-Energy 2017*. 2017, pp. 80–90.

[57] Joshua Comden, Zhenhua Liu and Yue Zhao. 'Harnessing Flexible and Reliable Demand Response Under Customer Uncertainties'. In: *e-Energy 2017*. 2017, pp. 67–79.

[58] Lukas Barth, Veit Hagenmeyer, Nicole Ludwig and Dorothea Wagner. 'How much demand side flexibility do we need?: Analyzing where to exploit flexibility in industrial processes'. In: *Proceedings of the Ninth International Conference on Future Energy Systems, e-Energy 2018, Karlsruhe, Germany, June 12-15, 2018*. 2018, pp. 43–62.

[59] Julian de Hoog, Khalid Abdulla, Ramachandra Rao Kolluri and Paras Karki. 'Scheduling Fast Local Rule-Based Controllers for Optimal Operation of Energy Storage'. In: *e-Energy 2018*. 2018, pp. 168–172.

[60] Majid Khonji, Sid Chi-Kin Chau and Khaled M. Elbassioni. 'Challenges in Scheduling Electric Vehicle Charging with Discrete Charging Rates in AC Power Networks'. In: *Proceedings of the Ninth International Conference on Future Energy Systems, e-Energy 2018, Karlsruhe, Germany, June 12-15, 2018*. 2018, pp. 183–186.

[61] Lawrence Berkeley National Laboratory. *Taxonomy for Modeling Demand Response Resources*. Tech. rep. LBNL, 2014.

[62] Graham Klyne and Jeremy J Carroll. 'Resource Description Framework (RDF): Concepts and Abstract Syntax'. In: *W3C Recommendation* 10.October (2004), pp. 1–20. URL: http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/.

[63] Emil Holmegaard, Aslak Johansen and Mikkel Baun Kjærgaard. 'Metafier - a Tool for Annotating and Structuring Building Metadata'. In: *SmartWorld'17*. 2017.

[69] Eric Prud'hommeaux and Andy Seaborne. 'SPARQL Query Language for RDF'. In: *W3C Recommendation* (2008). DOI: citeulike-article-id:2620569.

[73] Bharathan Balaji, Arka Bhattacharya, Gabriel Fierro, Jingkun Gao, Joshua Gluck, Dezhi Hong, Aslak Johansen, Jason Koh, Joern Ploennigs, Yuvraj Agarwal, Mario Berges, David Culler, Rajesh Gupta, Mikkel Baun Kjærgaard, Mani Srivastava and Kamin Whitehouse. 'Brick: Towards a Unified Metadata Schema For Buildings'. In: *BuildSys*. 2016. DOI: 10.1145/2993422.2993577.

[74] Bharathan Balaji, Arka Bhattacharya, Gabriel Fierro, Jingkun Gao, Joshua Gluck, Dezhi Hong, Aslak Johansen, Jason Koh, Joern Ploennigs, Yuvraj Agarwal, Mario Bergés, David Culler, Rajesh K. Gupta, Mikkel Baun Kjærgaard, Mani Srivastava and Kamin Whitehouse. 'Brick : Metadata schema for portable smart building applications'. In: *Applied Energy* (2018). DOI: 10.1016/J.APENERGY.2018.02.091.

[80] Mikkel Baun Kjaergaard, Aslak Johansen, Fisayo Sangogboye and Emil Holmegaard. 'OccuRE: An Occupancy REasoning Platform for Occupancy-Driven Applications'. In: *CBSE'16*. IEEE, 2016. DOI: 10.1109/CBSE.2016.14.

[83] W3C. *OWL*. https://www.w3.org/OWL/. 2012.

[84] W3C. *Turtle*. https://www.w3.org/TR/turtle/. 2014.

[85] Jason Koh, Bharathan Balaji, Dhiman Sengupta, Julian McAuley, Rajesh Gupta and Yuvraj Agarwal. 'Scrabble: transferrable semi-automated semantic metadata normalization using intermediate representation'. In: *Proceedings of the 5th Conference on Systems for Built Environments*. ACM. 2018.

[87] Docker Inc. *Docker Containers*. https://docker.com/. 2019.

[93] Jason Koh, Dezhi Hong, Rajesh Gupta, Kamin Whitehouse, Hongning Wang and Yuvraj Agarwal. 'Plaster: an integration, benchmark, and development framework for metadata normalization methods'. In: *Proceedings of the 5th Conference on Systems for Built Environments*. ACM. 2018, pp. 1–10.

[96] Mikkel Baun Kjærgaard and Fisayo Caleb Sangogboye. 'Categorization framework and survey of occupancy sensing systems'. In: *PMC* 38 (2017), pp. 1–13.

[100] Frauenhofer. *Omega 2.0*. https://www.iis.fraunhofer.de/en/ff/lv/ener/proj/ogema-2-0.html.

[106] David Huynh, David R Karger, Dennis Quan et al. 'Haystack: A Platform for Creating, Organizing and Visualizing Information Using RDF.' In: *Semantic Web Workshop*. Vol. 52. 2002.

[112] OSGP Community. *OSGP: Open Smart Grid Platform*. https://opensmartgridplatform.org/.

[113] Fisayo Caleb Sangogboye and Mikkel Baun Kjærgaard. 'PROMT: predicting occupancy presence in multiple resolution with time-shift agnostic classification'. In: *Computer Science-Research and Development* 33.1-2 (2018), pp. 105–115.

[114] Fisayo Caleb Sangogboye and Mikkel Baun Kjærgaard. 'PreCount: a predictive model for correcting real-time occupancy count data'. In: *Energy Informatics* 1.1 (2018), p. 12.

[115] Fisayo Caleb Sangoboye and Mikkel Baun Kjærgaard. 'Plcount: A probabilistic fusion algorithm for accurately estimating occupancy from 3d camera counts'. In: *Proceedings of the 3rd ACM International Conference on Systems for Energy-Efficient Built Environments*. ACM. 2016, pp. 147–156.

CHAPTER 14

OPM: AN ONTOLOGY BASED PACKAGE MANAGER FOR BUILDING OPERATING SYSTEMS

This chapter is a cosmetic adaptation of the following conference paper: Jakob Hviid, Aslak Johansen, Fisayo Caleb Sangogboye and Mikkel Kjærgaard. 'OPM: an Ontology Based Package Manager for Building Operating Systems'. In: *Proceedings of the International Conference on Internet of Things Design and Implementation* (*IoTDI 2020*). ACM. 2020. **In Submission**.

The paper is to be submitted to the ACM/IEEE International Conference on Internet of Things Design and Implementation (IoTDI'20), in Sydney, Australia, and if accepted, will be presented on 21 April 2020.

14.1 ABSTRACT

The energy sector is experiencing new challenges with the move to green energy. One of these challenges is keeping a stable energy grid when transitioning the production to unpredictable energy generation from green energy sources. Demand Response (DR) can mitigate some of the unpredictability by influencing the consumer's load profile. Unfortunately, the cost of DR implementation, and the required infrastructure, vastly overshadows the benefits for the consumer, thereby negating the incentive to invest. Therefore, reducing the initial cost of investment is a critical factor for the success of DR. Building Operating Systems (BOSs) is one possible avenue to achieve DR functionality in buildings. This paper seeks to reduce initial investment costs of BOSs, by introducing an Ontology based Package Manager (OPM), that dynamically resolves dependencies by installing needed services. An ontology-based approach to dependency resolution allows for loosely defined dependencies but also takes the context of the service into

account, as well as requirements in terms of sensor availability and physical layout of the building. The OPM leverages previous ontologies like Brick, Service Abstraction Layer (SAL), and Schema.org to describe these contexts. The OPM is evaluated by deploying a BOS and accompanying services for occupancy prediction. By significantly reducing deployment complexity, results show considerable time savings, and thereby cost reductions, on deployment and maintenance activities.

14.2 INTRODUCTION

Due to the effects of climate change, there are growing movements in the world, focusing on the transition towards green energy. Current research shows that retail store buildings in Denmark, make up about 8% of the industry sector's energy usage [25], and at a broader perspective, buildings, in general, make up about 40% of the total energy consumption [23, 24]. This usage makes buildings very relevant for addressing grid issues due to the increase in renewables.

Solar power and wind turbines tend to be obvious target candidates for the transition, but these also present new uncertainties for the energy grid, as their production is difficult to predict compared to traditional energy sources like fossil fuels. Also, energy providers have a responsibility to keep a balance on the network, or potential brownouts can occur. DR [18] is a technique used to directly or indirectly control or incentivize energy customers to use energy at the most favorable times of the day, using a multitude of strategies. DR thereby helps mitigate the uncertainties introduced by green energy initiatives.

However, previous research shows that the costs of implementing DR functionality in a retail store [1] would exceed the cost benefits for the customer. This is also the case for other types of buildings. Retail stores typically work with a short Return on Investment (ROI) cycles of about three years, as the future is always uncertain, while this number can dwindle to 1 year in the case of investments in the USA. In specific cases, like with malls owned by a third party, the ROI can be up to 30 years. During the study, it became apparent a viable approach to incentivize retail stores was to let systems have added functionality outside of DR, and to reduce deployment costs. Added functionality could be the ability to centrally control the buildings operational parameters, create statistics, and detecting whether the building was performing within

specifications. These functionalities would mitigate perceived cost and ROI calculations, as operational costs would decrease. One approach to enabling the above functionalities, and enabling DR, would be BOSs [28, 27, 46]. Once a BOS is deployed, adding additional functionality like DR, becomes economically feasible. This allows for a more sensible ROI calculation when pitching DR solutions to the demand side. BOSs are specialized operating systems, that control and manage buildings using a diverse array of services to control everything from an HVAC to predicting how many people are present in a building in case of emergency. BOSs, also provide a Hardware Abstraction Layer (HAL), enabling services and applications to have one common interface to the sensors and actuators in the building.

Today, buildings tend to be controlled using Building Management System (BMS), which are mostly closed loop systems, that only allow for limited adaptation of functionality. Also, most deployed retail stores have BMSs installed, that do not provide external interfaces for other software to integrate to [1]. There is a tendency for newer BMSs to allow for external integration, but these are few, more expensive, and are not present in most buildings as BMSs are not changed often in a buildings life cycle. BMSs does not fit well with the current green energy shift, as they do not possess the functionality needed to enable DR functionality, nor are they adaptable enough. BOSs, on the other hand, allow for external interfaces, and extendability to add new functionality as requirements change.

Several researchers are working on a future where buildings become platforms for applications, where multiple stakeholders would create and sell components for it, based on the HAL provided by the BOS platforms [28, 27, 29, 30, 32, 31]. For example, entrepreneurs would no longer only buy and deploy a BMS, but sell added functionality, as mentioned earlier, as a part of the system, and at the same time allow for the customer to get access to a plethora of applications and functionality. These applications would be created by another new stakeholder to the ecosystem, developers, essentially creating a potential new market for future functionality of buildings [5]. One possible example of added functionality from these application developers could be DR applications.

For DR to be feasible from a customer standpoint, initial costs of deployment need to be addressed. Solving the issues of deployment costs would require solutions to the following three areas:

1. Portable applications.
2. Automatic configuration of applications in a BOS.
3. Automatic deployment of application dependencies.

Previous research has already addressed point 1 from several perspectives; a HAL [28], metadata [63] descriptions, ontology-based descriptions of the hardware and physical structure of the building (Brick) [73], and a SAL [3]. For example, the HAL provides a common interface for applications, Brick provides a queryable discovery method for discovering the physical aspects of the building, and the SAL provides discoverability and publishability of services. The SAL describes all the services installed in the BOS, their interfaces, and in what context each of these is working. An example could be, a service providing predictions about how many occupants are going to be present in a specific room. The SAL can be queried using SPARQL Protocol and RDF Query Language (SPARQL) for the path to this information, and instructions on how to read it. Together, these effectively addresses point 1, portable applications, as the abstractions allow an application to be implemented once, and moved between building implementations, as their requirements are abstracted away from both the building instrumentation and the supporting software. Point 2 is addressed by Hviid et al. [5], by readying the BOS infrastructure for automatic configuration of applications and allow for the SAL to be updated at runtime. However, point 3 is still left largely unaddressed. To solve point 3, this paper proposes a package manager that handles the deployment of drivers and services and moves the BOS to a containerized deployment strategy. Creating a package manager for a BOS is not a simple task, as a BOS has several specific obstacles associated with it. For example, several BOSs are distributed, and therefore do not exist on only one host, but several. Also, the requirements for a package to function can be specific hardware instrumentation or services, and these services might have been deployed several decades ago.

Generally, four tasks can potentially take time when deploying, or changing a BOS setup:

1. Time taken to validating the hardware supports the software to be installed on the system.
2. Time taken to learn the dependencies of a service, and how to install them. Typically most of these concerns can be solved by

containerization using applications such as Docker [87], as all dependencies are contained within, and installation is mostly the same procedure for all containers.

3. Time taken to understand how the application works and how it ties together with other services. With containers, this is done with composition tools like Docker-Compose, but it requires an understanding of how these containers work together. On host systems, this is typically solved by dependency resolution, but dependency resolution is currently not supported by any of the containerization distributions.
4. Time taken due to packages with broken dependencies as buildings instrumentation and management are deployed for decades without upgrades. Currently, there is no solution for this, other than letting a repository version for a specific distribution exist for decades, or never removing packages.

The OPM, proposed in this paper, addresses all of these four issues, while traditional approaches typically one solve one to two of these. The OPM contributes the following:

– Container-Based Dependency Resolution utilizing the SAL, effectively creating a package manager for container-based applications.
– Deployment of containers into the BOS Ecosystem.
– Allowing not only direct but also loosely defined dependencies based in on the SAL ontology.
– Verifies hardware requirements of the applications.
– Verifies no other pre-existing service satisfies a dependency.

These loose dependencies are an essential part of the OPM design, as it allows the available packages to change over the several decades building instrumentation can last, but still allow the packages to work, by finding alternative solutions for the informational requirements of the package being deployed. Also, validating the hardware requirements is important, as several packages might not be reliable candidates if specific hardware is not present, potentially resulting in a defective dependency graph, and thereby a non-functioning deployment.

The OPM is evaluated from a cost perspective, by measuring the deployment time for the BOS before implemented changes, and after. Also, the perspective is given on the possible future shapes an OPM can have, and how they can impact the stakeholders of such a system.

Feature	OPM	Spawnpoint	Docker	Kubernetes	Spack	Snap	RPM / APT
- Maturity Level	Research	Research	Industry	Industry	Research	Industry	Industry
- Intended Purpose	BOS	BOS	Generalized	Generalized	Generalized	Generalized	Generalized
- Operational Space	Distributed	Distributed	Distributed	Distributed	Distributed	Local	Local
- Type of package managed	Container	Container	Container	Container	Binaries	Container	Binaries
- Multi-host deployment strategy	Yes	Yes	Yes	Yes	Yes	No	No
- Host Node Management Support	Indirect	Yes	Yes	Yes	Yes	No	No
- Resource Management	Indirect	Yes	Yes	Yes	Yes	No	No
- Dependency Resolution	Yes	No	No	No	Yes	No	Yes
- Loose Dependencies	Yes	No	No	No	No	No	No
- Context Specific Dependencies	Yes	No	No	No	No	No	No
- Validate Hardware Requirements	Yes	No	No	No	No	No	No
- Dependency Resolution Takes Currently Installed Packages Into Account	Yes	No	No	No	Yes	Does not apply	Yes
- Dependencies of the Package Manager	Kafka Docker	Bosswave Docker Linux	Linux, Mac OS, or Windows	Linux, Mac OS, or Windows	Linux	Linux	Linux

Table 14.1: Feature Comparison between Package Managers.

14.3 RELATED WORK

In the package management space, multiple solutions exist. Some are designed for deploying binaries, while others are designed for deploying containers. The package managers covered in this paper are the OPM itself, Spawnpoint [86], Docker (with Docker Compose) [87], Kubernetes [88], Spack [89], Snap [90], and traditional package managers that operate at host level such as RPM [91], and APT [92]. Table 14.1 details the features of these packages. There are two overall types of package managers present in the table.

1. **Orchestrators** that are designed to deploy containers into distributed systems. Indirectly, these are also package managers, as they orchestrate how the containers are supposed to work together and get the containers from repositories. These include Spawnpoint, Docker, and Kubernetes.
2. **Package Managers** that are designed to install specific binaries on a host. The last three; Spack, Snap and RPM/APT are examples of these.

Exploring the **Orchestrators**, all of these have strategies for deploying containers on multiple hosts and is built to manoeuver in a distributed systems architecture. Spawnpoint is a secure container deployment software designed to work with Bosswave [47]. It manages the hosts by locating hosts in the blockchain deployed by Bosswave and then sends install commands that include resource restrictions and other information. Destroying, restarting, and deploying containers, can all be done through Spawnpoint. Spawnpoint's intent is to be used in a BOS, just as the OPM. Docker is one of the most popular container orchestrators. It can run in swarm mode and function as a cluster of nodes and can mediate High Availability (HA) functionality. Docker uses Docker Compose to define what containers should work together and what resources they may use. Kubernetes, to some degree, functions somewhat similar to Docker Swarm, as it functions as an orchestrator for containers, and provides container, resource, and HA management. Kubernetes can function on top of Docker for container orchestration, but not at the same time as Docker Swarm is deployed. The OPM takes many of its features from the three systems above. However, OPM does not manage the notes on which it is installed as the other orchestrators but instead leaves a communication endpoint in the communication

bus for each of the hosts available. So, centrally managing the OPM instances is possible, but it is not controlled directly by the OPM. The OPM builds on Docker to deploy its containers, and therefore also inherits the possibility to use Docker Swarm, and gain the HA functionality for its containers and itself. Spawnpoint, Docker, and Kubernetes all support resource management of the containers. The OPM does not explicitly take care of this, but indirectly supports it as it uses Docker Compose to deploy containers. Traditionally containers do not have dependencies as they should contain them all, or the composition of services should solve this problem. However, dependency resolution still plays a critical role when multiple containers are available, interfaces are complicated, and the user has little to no prior knowledge of the other packages available. Also, as building deployments are present for decades, and available packages change over time, loose dependency resolution is required to allow packages to work decades after initial dependencies. The OPM supports direct dependencies, as well as loose dependencies to other containers. Another area where the other orchestrators fall short is validating the hardware requirements of the building instrumentation, but also exploring the physical context of the building. These are all supported by the OPM.

Moving on to the second category, **Package Managers**, Snap stands out as the only candidate that uses containers for deployment. It is used on Ubuntu Linux as a next-generation package manager, that is supposed to solve many of the problems with traditional package managers such as RPM / APT. To do this, Snap adds all of its dependencies inside of the container, making sure it does not need dependency resolution functionality. This works fine on a single host, as packages are not supposed to work together in a distributed system. Snap is different from the orchestrators, as it only deploys containers on one host, and therefore lacks any multi-host functionality. In that regard, it works more like the traditional Linux Package Managers like RPM / APT, as the intent is to install an application on a single host. Spack is a package manager, that seeks to bring the functionality of RPM / APT to high performance computing data centers, enabling deployment of binaries to hundreds of machines at once. It has all of the distributed functionality as the orchestrators but does not work with containers, but binaries instead. As it works with binaries, just as RPM / APT, it uses dependency resolution to resolve the packages needed. None of the package managers has the ability to validate hardware require-

ments, have loose dependencies, or explore the context of the building. Spack and RPM / APT does however also check if requirements are already installed, just as the OPM uses the SAL installed in the BOS.

As Table 14.1 demonstrates, no current container orchestrators, nor packages managers, provides the feature set of the OPM. More specifically, no container based system performs dependency resolution between containers, loose or direct, nor do they validate the physical requirements of the containers.

14.4 THE OPM CONCEPT AND DESIGN

This paper will attempt to reduce the deployment costs of BOSs. This cost reduction will be attempted by introducing an ontology-based package manager. The system will work in the context of the BOS ecosystem described later. The BOS is based on a microservice architecture, with a communication bus based on Kafka [50]. Given a SAL, Brick and the BOS context, the OPM will need to support the deployment of services and drivers, and take existing services, drivers, and instrumentation into account when deploying. The following functionalities for the OPM were identified:

1. Resolve packages needed for the package to function.
2. Check if a package's loose requirements are already satisfied by other packages in the BOS.
3. Support direct dependencies on other packages.
4. Validate if the physical parameters, hardware, and spatial, are satisfactory for the package to function.
5. Function within a distributed BOS space.
6. Function in a containerized environment, and be containerized itself.
7. Install the packages identified to satisfy the missing informational needs.

To support these functionalities, the OPM was developed to act as a package manager for the BOS used in this paper.

The basic concept of the OPM is to deploy containerized services and drivers with their depending packages, by leveraging ontology-based descriptions of the physical aspects of the building, and the drivers and services contained in the BOS.

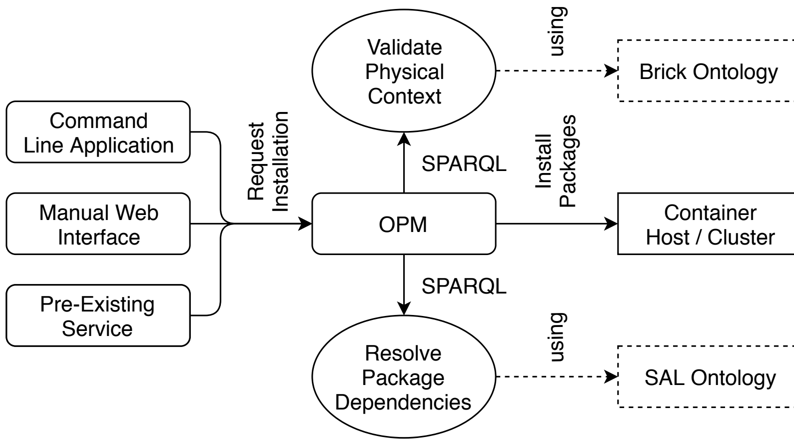

Figure 14.1: Ontology based Package Manager (OPM) Concept.

Figure 14.1 shows how this OPM conceptually manifests itself. Several sources can request installations; A scriptable command line application, manually by using a web interface, or by using a pre-existing service in the BOS. The request is picked up by the OPM, that recursively validates the package dependencies, and validates the building instrumentation supports the package. Package dependencies are resolved by the OPM, using an internal Resource Description Framework (RDF) [62] server, which leverages the SAL [3, 5] ontology. SAL is an ontology that seeks to create abstraction layer from specific services, enabling information needs to be collaboratively provided by several services, effectively adding resiliency, multiple pathways, and service discovery in the relevant physical context. The SAL description for each service in the OPM describes a generalized version of the service that can be installed. This description encompasses all of the endpoints that can be used to interface with the service, and what data types it provides and in what context in relation to Brick. The OPM checks if a service is installed already, as well as if dependencies are already satisfied. These checks are done using the Service Support System (SSS) [5]. The physical context is validated for each dependency using a SPARQL [69] query that is executed on a model using the Brick ontology [73, 74]. Brick is an ontology created using RDF and is used for describing buildings. The ontology especially excels at describing relationships between the physical areas in the building, and the in-

strumentation, but also how one part of the instrumentation relates to another. Brick's approach also has a sharp uptake in adoption, and will also be included in the upcoming 223P ASHRAE Standard [116]. The SSS hosts both the Brick and SAL model, but the dependency resolution model is stored and handled separately in the OPM. The SSS handles the running states and contexts (SAL and Brick) for the BOS, while the OPM handles potential new packages, and what can be added to the running system.

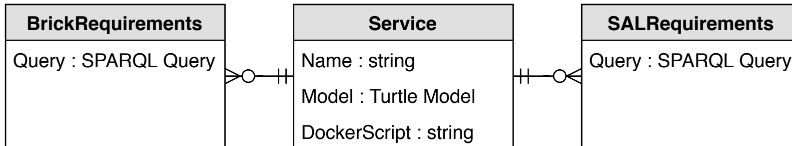

Figure 14.2: Simplified DB Structure.

Figure 14.2 shows a simplified version of what type of data is stored for each service that can be installed. The service table has a name, a SAL model describing what the service can provide (in the turtle [84] format), and finally a Docker Compose [117] script that defines how the service will be installed. A few environment variables, like what communication bus to connect to, will be added to this description by the OPM. The service has two other tables associated with it; BrickRequirements, and SALRequirements. These are optional and can have multiple entries per service. The query field in each of these tables is SPARQL queries that define what type of information is needed to function as intended. These are the queries used to validate the physical context, check if a service is already installed, and finally creating the dependency tree. When the OPM starts, it loads all the turtle models in the database into an RDF store, and afterward utilizes this store, to resolve what package satisfies a specific dependency.

To support the OPM, and the case used in the evaluation, the SAL needed to adapted for the specific use case. ServiceEndpoints are an abstraction for a location where a part of a service can be accessed. The SAL already supports ServiceEndpoints, but does not support the notion of read or write access, nor does it support different variants of these. Figure 14.3 shows the changes made to the SAL, and how they inherit from each other. The service endpoint now has several subclasses that define what type of endpoint it is, and the specific data

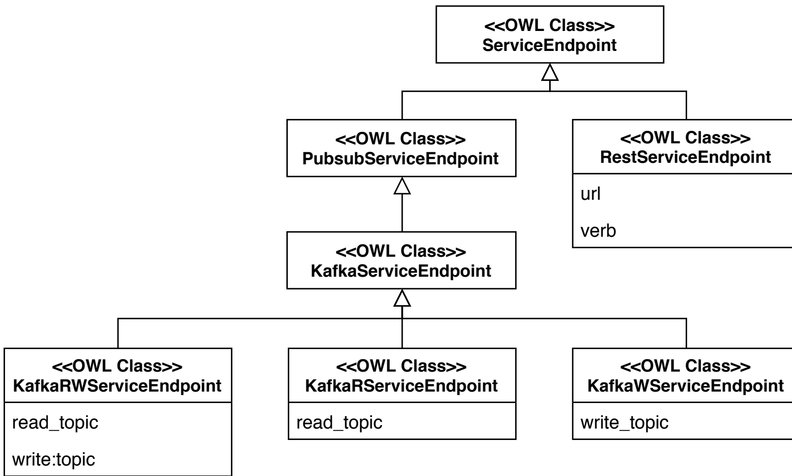

Figure 14.3: The SAL Extensions Made to Support Input Descriptions.

properties needed to interact with that type of service. This change effectively adds identification of an endpoint is intended for reading, writing, or both. The extensions are using Web Ontology Language (OWL) [83] and are saved in the turtle format, just as the SAL.

The turtle models describing the available services that can be installed, as shown in Figure 14.2, are described using a SAL model saved in the Turtle format. A simplified example of this model with only a single service endpoint is detailed in Figure 14.4. Also, to be able to make a requirement in the OPM to the sample service, it needs to be queried. Figure 14.5 shows a simplified example of this query.

The OPM is packaged as a Docker container and can exist in several different scenarios, as shown in Figure 14.6. Docker can exist as a Docker Swarm, or as a single Docker host. The OPM is not restricted to either, and as the figure shows, several instances of the service can exist. One for each Docker Swarm, or one for each single Docker host. If several swarms are present in the setup, or several hosts, each of these would require an instance of the OPM to be able to install services on that host. The Kafka Message Bus [50] is the communication bus used by the BOS. It is a Publish/Subscribe architecture, and is used for all communication between the SSS, OPM, drivers, and services. Kafka can have multiple instances, or just one, and is not required for each host. Drivers (marked with green) are the adapters between the

```
@base <https://achmecorp.com/sal.ttl>.
@prefix sal: <https://achmecorp.com/sal.ttl#>.
@prefix sali: <https://achmecorp.com/sali.ttl#>.
@prefix brick: <http://brickschema.org/ttl/Brick.ttl#>.
@prefix schema: <http://schema.org/version/latest/schema.ttl#>.
@prefix model: <https://achmecorp.com/salmodel.ttl#>.

model:Information_1 a sal:Information;
        sal:hasLocation model:ACME;
        sal:hasModality sali:AbsoluteTime;
        sal:hasTemporalAspect sali:Archival;
        sal:hasUnit sali:DateTime;
        sal:location "Time".
model:KafkaRServiceEndpoint_1 a sal:KafkaRServiceEndpoint;
        sal:ownedBy model:organization_acme;
        sal:priority "1";
        sal:providesInformation model:Information_1;
        sal:read_topic "ReadTopicInKafka".
model:ACME a brick:Location.
model:Service_1 a sal:Service;
        sal:hasKafkaRServiceEndpoint model:KafkaRServiceEndpoint_1;
        sal:name "Service Example".
model:organization_acme schema:legalName "Acme Corp";
        a schema:Organization.
```

Figure 14.4: Simplified Example SAL Model of a Generic Service.

```
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
prefix sal: <https://achmecorp.com/sal.ttl#>
prefix sali: <https://achmecorp.com/sali.ttl#>
prefix brick: <http://brickschema.org/ttl/Brick.ttl#>

SELECT ?service_name
WHERE {
    ?service rdf:type/rdfs:subClassOf* sal:Service .
    ?sep     rdf:type/rdfs:subClassOf* sal:KafkaServiceEndpoint .
    ?info    rdf:type/rdfs:subClassOf* sal:Information .

    ?servtype  rdfs:subPropertyOf sal:hasKafkaServiceEndpoint .
    ?service   ?servtype          ?sep .
    ?direction rdfs:subPropertyOf sal:directionalInformation .
    ?sep       ?direction         ?info .

    ?service sal:name              ?service_name .
    ?info    sal:hasModality       sali:AbsoluteTime .
    ?info    sal:hasUnit           sali:DateTime .
    ?info    sal:hasTemporalAspect sali:Archival .
}
```

Figure 14.5: Simplified Example of a SAL Requirement Query.

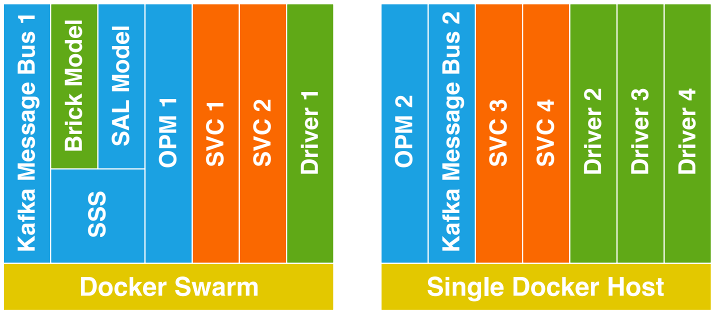

Figure 14.6: Host Overview: OPM required for each swarm or host. Drivers and services can be distributed. Only one SSS should exist for the setup. Colors correspond with the deployment phases described in Figure 14.8.

instrumentation of the building and the Kafka message bus. Services (marked with orange) are the services handled by the OPM. Both services and drivers are not restricted to one host and can be distributed between them as the figure shows. The containerization of the microservices is a deliberate choice, as it forces developers to include all its code dependencies for an application, or expose only a few settings that need to be configured. This choice minimizes complications when someone not familiar with the application needs to deploy it. Also, containerization, in this case with Docker, allows for stored versioning of the applications, easier upgrades between versions.

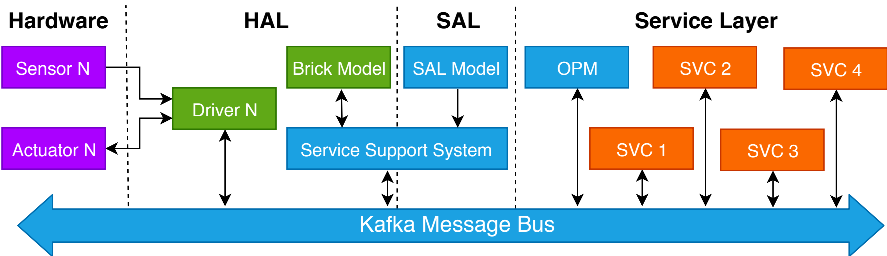

Figure 14.7: BOS environment with color coding for order of deployment. Excluding physical hosts. Colors correspond with the deployment phases described in Figure 14.8.

Seeing the system from a logical application level; the landscape looks similar but communicates a better understanding of how the

components work together. Figure 14.7 describes this BOS environment. The figure details how several hardware devices are connected to Kafka by the help of the drivers. Also, it shows the multiple Kafka instances work together as one communication bus. The OPM, SSS, and all the services communicate through Kafka. The colors of both Figure 14.6 and Figure 14.7 pertain to the phases the component is usually deployed. Further information on this can be found in the evaluation.

14.5 EVALUATION

The evaluation method for this paper mainly focuses on measuring the mitigation of deployment costs. To do so, one must understand the phases for deploying a BOS and what each phase entails. Figure 14.8 shows the phases of a BOS and the components of the system that are installed and configured for that phase. The figure is color coded to show how each phase pertains to Figure 14.6 and 14.7. Each phase needs to complete before the next begins, as each phase has a dependency on a component in the previous phase. The first phase, hardware, pertains to the physical devices needed for the BOS to function. These are the instrumentation in the form of sensors and actuators, but also the servers where the BOS is going to reside. In this phase, the operating system and utilities needed are also installed. The second phase installs the essential components of the BOS that are common for all deployments of the BOS. In this case, this includes Kafka, the OPM,

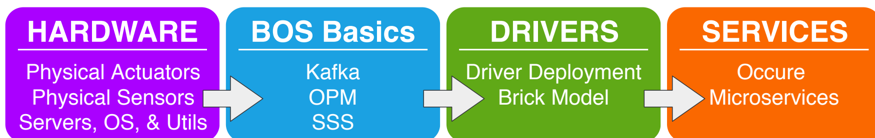

Figure 14.8: A typical BOS Deployment Procedure, divided in phases, with associated tasks.

and the SSS. Once essential components are installed, the drivers are deployed. These drivers are specific to the building instrumentation, and therefore, the number of these, and the choice vary. At this stage, the Brick model is also finalized. If the drivers do not require Brick model validation, or the model is complete, the OPM can install them. The Brick model can take considerable time to create, but could ideally be

provided by the entrepreneur installing the building instrumentation, as they possess most of the knowledge needed. Once the brick model is present, the OPM can validate the physical context, and therefore install service type packages with hardware requirements. Once the needed functionality is installed, the BOS is ready.

14.5.1 Evaluation Case: OccuRE

OccuRE [80] is an occupancy provisioning framework that supports the different information types (such as presence, counts, traces, identity, and activities) for enabling DR applications. Moreso, OccuRE can provision this occupancy information in multiple spatial resolutions and coverages, and in multiple temporal coverages [96, 80, 113]. Fig-

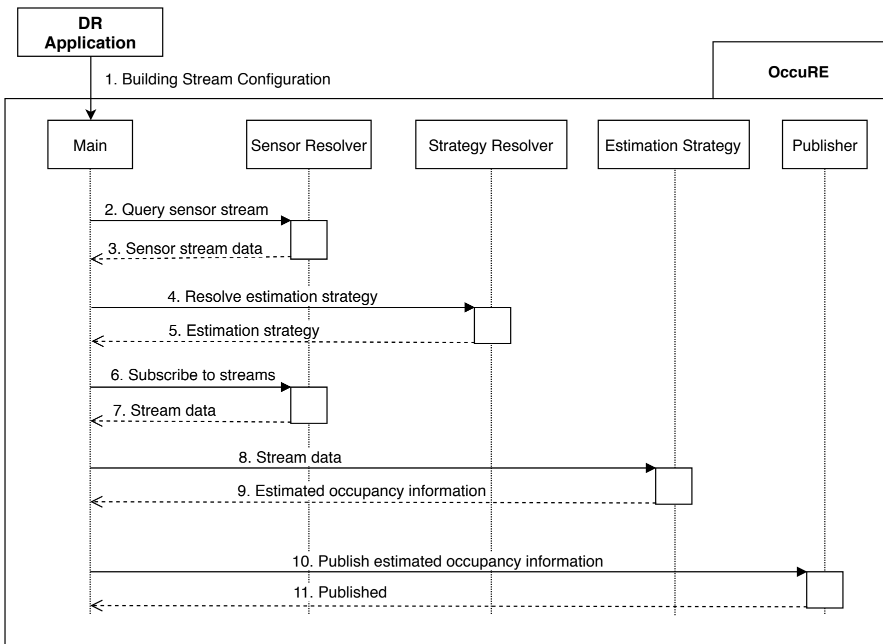

Figure 14.9: Sequence of interaction between the OccuRE Components without SSS.

ure 14.9 provides an overview of the interaction between the various components of OccuRE. The original implementation of OccuRE is comprised of several components for extracting occupancy requests from DR applications. These components include a sensor resolver, a

strategy resolver, a number of estimation strategies and a publisher. The sensor resolver identifies the relevant sensor modalities for provisioning the requested information while the strategy resolver for identifying the necessary estimation strategy for computing the occupancy information from the sensor data. Lastly, the publisher provides a unified interface for provisioning the computed occupancy information. Figure 14.10 is an example of a typical REST request from a DR application requesting occupancy counts in a building in real-time. Given such a request, OccuRE utilizes its sensor resolver method to request Metadata information from a `Metadata Broker` about a list of available sensor streams data for that particular request. Subsequently, the strategy resolver module is used to search for all available strategies that can map the requested occupancy information with the available sensors. Lastly, a strategy is selected for computing the occupancy information using the datasets from the resolved sensor streams, and the computed information is relayed to the DR application through the OccuRE publisher.

```
{
    "spatialcoverage": "Building A",
    "temporalgranularity": "1 Minute",
    "sensormodalities": "Camera Counter",
    "spatialgranularity": "Room",
    "informationtype": "Counts",
    "temporalcoverage": "Real-Time"
}
```

Figure 14.10: Building Stream Configuration.

The challenge with the current OccuRE implementation includes its tightly coupled components which provide little flexibility for developing new novel occupancy strategies for provisioning requested occupancy information. For instance, OccuRE utilizes sMAP [46] archiver as a `Metadata Broker` which enables an SQL-like query language to query available building instrumentation and stream datasets. Hence, given the building stream configuration in Figure 14.10, OccuRE formulates the sMAP query provided in Figure 14.11 to query the available sensor data in the `Metadata Broker`. In order to formulate these queries, OccuRE requires a technical know-how of the Metadata structure for a particular building in order to resolves the available sensor streams.

Also, it is practically impossible to provision occupancy information for multiple buildings with differing Metadata structure especially in cases where a DR event is scheduled for a site. Secondly, OccuRE's tightly coupled structure with the `Metadata Broker` which provides both Metadata information and data streams, limits the scope of potential occupancy strategies that can be implemented especially when the required data streams are not supported in the `Metadata Broker` - sMAP. Thirdly, OccuRE implements a custom abstract class that are extended by each concrete strategy to provide a uniform order of execution and for extracting necessary strategy metadata for the strategy resolver. This uniform order of execution limits the expressiveness of these strategies especially for multimodal occupancy strategies which may require a number of sensor stream data for estimating the requested occupancy information. Lastly, OccuRE provides no mechanism for defining dependencies between various occupancy strategies. For example, an occupancy strategies for predicting occupancy counts in buildings may require occupancy estimates from a strategy proffering occupancy counts in real-time.

```
SELECT  *
WHERE   Metadata/Form = 'cameracount'
        AND Metadata/Building = 'Building A'
```

Figure 14.11: sMAP Queries for Resolving Sensor Stream Data.

OccuRE can now fully leverage the implementation of the SSS and OPM to address the aforementioned bottlenecks, and we hereby refer to SSS and OPM implementation of OccuRE as OccuREv2. Firstly, each of the original OccuRE estimation strategies are redesigned and published as estimation services. Each estimation service is comprised of a service handler, the strategy, and a publisher. The strategy handler registers the service, receives each occupancy request, and it manages a number of running strategies on the service. Figure 14.12 provides an overview of the interaction between OccuRE, the SSS, and the estimation service. On deploying an estimation service, this services registers itself on the SSS to provides SAL information about its ServiceEndpoints. Subsequently, given a similar occupancy request as provided in Figure 14.10, OccuREv2 utilizes its strategy resolver to query the SSS for all ServiceEndpoints. The returned ServiceEndpoints

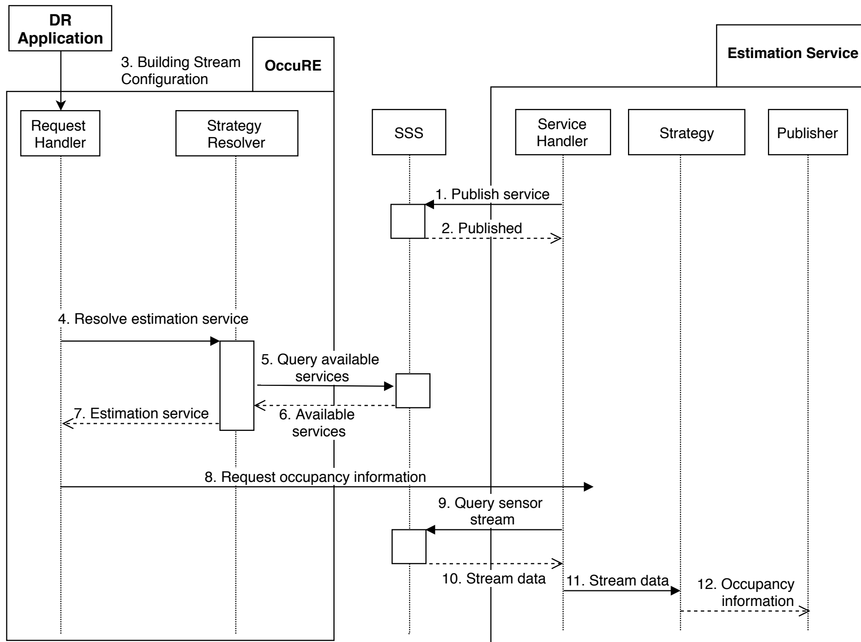

Figure 14.12: Sequence of interaction between the OccuRE, the SSS and Estimation Service.

are resolved by the strategy resolver to determine the relevant service for proffering the requested occupancy information. Given the relevant service, OccuREv2 sends a message through the Kafka bus requesting the occupancy information from the selected estimation service. On receipt of the message by the estimation service, the estimation service queries the SSS for the required sensor streams and data for computing the occupancy information. The estimation service subsequently spawns a strategy for computing the occupancy information, which is subsequently published by the publisher.

With the current architecture, the capabilities of Brick defined in SSS can be leveraged to develop both generic and specialized services that are agnostic to incoherent Metadata structures. Secondly, each estimation service has full flexibility for leveraging different and custom deployments of `Metadata Brokers` without any constraints for using a particular broker. This flexibility also extends to the manner in which an estimation service requests for sensor stream data for computing the requested occupancy information. Lastly, the OPM can be lever-

aged to define dependencies between various estimation services while OccuREv2 serves as an orchestrator for the estimation services. In order to illustrate these capabilities, we have deployed both OccuREv2 and an estimation service for estimating past occupancy counts using stereo-vision camera sensors. This estimation service implements the PLCount [115] as a strategy. Also, this estimation service provides a basis for other depending estimation services for estimating occupancy count both in real-time (current time) and in the future (prediction). Upon receipt of an occupancy request, the PLCount estimation service utilizes the SPARQL query to obtain information about available sensor streams from the SSS. Using this query, the directionality of the count-lines defined in the camera sensors can be obtained without prior knowledge of the ground truth for any given building using a SPARQL query. This, in contrast to the implementation in OccURE, where there exist some limitations with regards to the representation of the placement of camera sensors within buildings. This limitation was overcome by implementing an hard-coded mapping that follows the problem formulation in [115]. It should be noted that this hard-coded mapping requires a ground truth knowledge of the directionality of these count-lines and thus impedes scalability to unknown buildings.

Modules	Lines of Code	
	OccuRE	OccuREv2
sensor resolver	211	0
strategy resolver	101	79
publisher	40	0
main	254	134
Total	**606**	**213**

Modules	Lines of Code	
	PLCount	PLCountv2
strategy handler	0	180
publisher	0	29
strategy	166	166
main	459	0
Total	**625**	**384**

Table 14.2: Comparison between OccuRE and OccuREv2 and PLCount and PLCountv2 respectively.

Given the returned Kafka topics from the SPARQL query, the `service handler` module of the estimation service can subscribe to the Kafka topics of the count-lines defined on the camera sensors, and it can aggregate the transition data from all count-lines to compute both the transition and cumulative counts from the building in the past. The computed occupancy information is subsequently published using `publisher` module to Kafka topic representing the instance of the spawned strategy in the estimation service. In Table 14.2, we compare

the between the original implementation and refactored the code for both OccuRE framework and the PLCount estimation strategy. For all OccuRE and PLCount modules, we were able to achieve a 64% and 38% code reduction of the original codebase respectively.

14.5.2 Evaluation parameters

To evaluate time saved from the OPM installation procedure, the time it takes to deploy the entire BOS is measured. First, the original BOS is installed and measured 3 times, and after this, the process is repeated with the OPM changes. Each iteration, was performed by the same person with no prior experience with deploying the software. The person has the opportunity to learn from the mistakes from each iteration, including what dependencies the software package has, making the next iteration easier to install.

Figure 14.13: The University of Southern Denmark OU 44 Living Lab used for the Brick Model and for the data used in the OccuRE services.

For the evaluation of this paper, the sensors, actuators, data, and Brick model, was obtained from our live living lab [118], OU44, at at the University of Southern Denmark as pictured in Figure 14.13.

This means that the deployment time of the sensors, actuators, drivers, and the time to create the Brick model is not included in the measurements for this evaluation, effectively skipping the driver deployment phase, and only leaving parts of the hardware deployment. Both tests followed these parameters:

1. Excluded from the Hardware Phase; Sensor and actuator deployment, as they are pre-deployed.
2. Included in the Hardware Phase: Host operating system installation time. Ubuntu 18.04 LTS is the operating system of choice in this case.
3. General: All binaries/containers needed are placed locally, so internet connection or repository placement is not a factor.
4. General: All services need to start automatically again if the host is rebooted.

The manual installation followed these parameters:

1. Included in the Hardware Phase: Packages needed for the software to run: Python packages using pip.
2. Included in the BOS Basics Phase: One instance of Kafka (Java Application), the SSS (Python Application), and the SAL model.
3. Included in the Services Phase: Deployment of OccuRE (Python Application).

The OPM based installation followed these parameters:

1. Included in the Hardware Phase: Packages needed for the software to run: Docker, Docker Compose.
2. Included in the BOS Basics Phase: One instance of Kafka (Docker Image), the SSS (Docker Image), the SAL model, and one instance of the OPM (Docker Image).
3. Included in the Services Phase: Deployment of OccuRE (Docker Image) and the depending packages (3 Docker Images) using the OPM. Package dependencies detailed below.

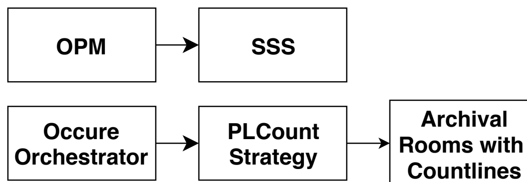

Figure 14.14: Package Dependencies with Brick requirements.

Figure 14.14 shows the dependencies between the packages described in the OPM based evaluation setup, as well as the validation needs

in Brick for the physical context in the building. The OPM itself only depends on the SSS, while the OccuRE orchestrator depends on the PLCount Strategy package. The PLCount Strategy depends on past data being available with count-lines data. The figure shows the initial services intended to solve the informational needs, but are all described loosely using the SAL ontology, so, if a package is no longer available, the requirements might be satisfied by other services providing the same information, or parts of it. Testing this partial information functionality was already covered by Hviid et al. [5], and will therefore not be covered in this evaluation.

14.6 RESULTS

After building the system, and executing the evaluation methodology, results were collected. Table 14.3 shows the measured time for the manual install approach and the OPM approach. The time is split into the deployment phases previously discussed in Figure 14.8.

	Manual			OPM		
Iteration	**1**	**2**	**3**	**1**	**2**	**3**
Hardware	6	6	6	7	6	6
BOS Basics	69	20	14	1	1	1
Services	18	7	4	1	1	1
Accumulated Time	93	33	24	9	8	8

Table 14.3: Time in minutes to prepare the BOS and OccuRE, divided into phases and iterations.

Table 14.3 shows the manual deployment time takes considerably longer but decreases with experience when deploying the system. Deploying with the OPM has consistent installation times that are significantly lower than the manual approach, even at the first iteration. Installing the BOS basics was mostly caused by unclear error messages to dependencies that needed to be debugged. These dependencies varied from the Java runtime environment, Python environment versioning, missing packages, and more.

14.7 DISCUSSION

As mentioned in the results, Table 14.3 shows consistent deployment duration with the OPM, while the manual approach varied greatly in time, but shows marked improvements with each iteration. While many of the delays in the initial iterations can be automated, if there are standard deployments, it does not solve the issue of different versions of runtimes and dependencies. Moreover, building instrumentation exists for decades, and as systems are maintained and software changes, a technician can often end up with installation or maintenance durations that are closer to the first iteration, than the third. The consistency in deployment duration with the OPM shows how the combination of containerization, dependency resolution, and package manager functionality can drastically minimize costs, and require less prior knowledge of the software being deployed. Also, the evaluation does not provide a complete picture, as the deployment of the BOS was performed with personal having some degree of experience with partial deployment of a BOS. Being new to BOSs, in general, would most likely result in a more significant gap in deployment durations, between the two approaches. At the same time, the physical context of the building is also being validated, which ensures a markedly higher likelihood that the software being deployed works with the instrumentation. This validation can result in further time savings not measured in this paper. Also, a typical BOS does not only contain the functionality of OccuRE but can potentially have hundreds of use cases and services ranging from simple control to complicated algorithms. This means that the complexity of a deployments services phase can be insurmountable for someone new to the area. The time savings seen for the deployment of OccuRE should also be multiplied by each service present on the system, resulting in more significant savings.

When taking into account the stakeholders involved in a BOS, several things change with the introduction of the OPM. The relevant stakeholders are entrepreneurs that deploy and maintain the BOS, developers that create the software that runs on the BOS platform, and finally the customer, that buys and uses the system in day to day operations. For entrepreneurs, a few parameters change. First, expertise levels required of the technicians can be of a narrower scope. This change allows for shorter employee training periods, as well as leaving the more experienced technicians to deal with more complex tasks,

instead of deployments. Second, as complexity is moved away from the entrepreneurs, and to the developers, an opportunity for creating a more potent value proposition for customers presents itself. This opportunity is due to employees with expertise being free to explore application functionality instead, and exploring functionality to the BOSs; they sell, instead of assimilating in-depth and intricate knowledge about specific services in the system, with little to no direct value to the customer. Third, technicians that require less training tend also to be a less expensive workforce, again creating cheaper deployments that benefit both entrepreneur and customer alike. Fourth, aftermarket maintenance and upgrades for BOSs will be less of an investment for the customers, thereby potentially creating a relationship that generates more sales over time, as upgraded functionality is simple to add. For application developers, several parameters change as well. Having a repository of software allows developers to have a place to expose potential customers to their software. It also allows for new indirect sales channels, where packaging, distribution, and installation, remove most of the friction of BOS related software. Potentially, customers could also be other developers that need specific problems solved. Having a frictionless point of distribution and deployment is also one of the first steps towards the vision of an application store for buildings, again creating new opportunities. Creating a frictionless distribution channel would also encourage developers to create new and untraditional applications that involve buildings, emulating the exploration of form factor functionality that was last seen in 2008 with mobile phones. However, for this to happen, BOSs need to be more widespread and have a clear path to monetization. An example of a potential application developer could be an energy provider, seeking to make companies integrate to DR initiatives. Having frictionless platforms, as described above, allows for an easier buy-in from customers, as the added functionality is cheap to install, and functions on top of their already deployed BOS. Finally, the customers. Customers could benefit from an OPM on several levels. First, less costly deployments make an initial investment in a BOS more feasible. Second, a more frictionless installation experience could allow a customer to experiment and explore. Currently, this kind of exploration requires a high level of expertise. Third, cheaper access to upgrading or changing building functionality after initial deployment, potentially allowing companies to more easily integrate buildings functionality into day to

day operations. Generally, the potential impact of a more frictionless and cheap deployment of BOSs is in the interest of all stakeholders involved. Simply put, the OPM has a value proposition for all levels of the stakeholder ecosystem.

14.8 CONCLUSION

OPM is a an ontology based package manager, that deploys containers in a BOS. It introduces time effective and simple deployment of drivers and services, demanding less knowledge to use, while also validating the physical context of the building is supported by the packages. The OPM was used to deploy a BOS several times, and compared to the previous deployment method. Significant time savings were documented, with the OPM consistently having reliable deployment times. The OPM impact on the ecosystem was also discussed.

Possible future avenues for this work could include automatic package deployments when certain patters are detected in the physical context of the building. This could result in plug and play functionality for hardware instrumentation, by automatically deploying drivers when a certain instrumentation is detected, thereby further reducing deployment costs in a deployment phase that was only sparsely explored in this paper. Also, more direct integration of resource management into the OPM could be beneficial to catch up to the alternatives in this area.

Even without the above functionality, the OPM has demonstrated a significant cost reduction to the deployment costs, by using less time on deploying or maintaining the BOS. This was achieved by resolving dependencies on other packages, and containing its binary dependencies within containers, thereby significantly reducing deployment complexity for the technician deploying the system.

REFERENCES

[1] Jakob Hviid and Mikkel Baun Kjærgaard. 'The Retail Store as a Smart Grid Ready Building: Current Practice and Future Potentials'. In: *Power & Energy Society Innovative Smart Grid Technologies Conference (ISGT), 2018 IEEE*.

IEEE. Washington DC, USA, Oct. 2018.
DOI: 10.1109/ISGT.2018.8403354. **Published**.

[3] Jakob Hviid and Mikkel Baun Kjærgaard. 'Service Abstraction Layer for Building Operating Systems: Enabling portable applications and improving system resilience'. In: *2018 IEEE International Conference on Communications, Control, and Computing Technologies for Smart Grids (SmartGridComm)*. IEEE. Aalborg, Denmark, Oct. 2018, pp. 1–6.
DOI: 10.1109/SmartGridComm.2018.8587543. **Published**.

[5] Jakob Hviid, Aslak Johansen, Fisayo Caleb Sangogboye and Mikkel Baun Kjærgaard.
'Enabling Auto-Configuring Building Services: The Road to Affordable Portable Applications for Smart Grid Integration'.
In: *Proceedings of the Tenth International Conference on Future Energy Systems (ACM e-Energy '19)*.
Phoenix, AZ, USA: ACM, June 2019, pages 68–77.
DOI: 10.1145/3307772.3328288. **Published**.

[6] Jakob Hviid, Aslak Johansen, Fisayo Caleb Sangogboye and Mikkel Kjærgaard. 'OPM: an Ontology Based Package Manager for Building Operating Systems'.
In: *Proceedings of the International Conference on Internet of Things Design and Implementation (IoTDI 2020)*. ACM. 2020.
In Submission.

[18] Mary Ann Piette, Sila Kiliccote and Girish Ghatikar.
'Field Experience with and Potential for Multi-time Scale Grid Transactions from Responsive Commercial Buildings'.
In: *ACEEE Summer Study on Energy Efficency in Buildings*. 2014.

[23] European Parliament and Council of the European Union.
'Directive 2010/31/EU of the European Parliament and of the Council on the Energy Performance of Buildings'. In: *Official Journal of the European Union* L153 (May 2010), pp. 13–35.
URL: http://eur-lex.europa.eu/eli/dir/2010/31/oj.

[24] U.S. Department of Energy. *2011 Buildings Energy Data Book*. Tech. rep. U.S. Department of Energy, Mar. 2012.
URL: https://catalog.data.gov/dataset/buildings-energy-data%20-book.

[25] Danmarks Statistik. *ENE3H: Bruttoenergiforbrug i fælles enheder efter branche og energitype* [*Accessed: 23/2/2017*]. http://www.statistikbanken.dk/ENE3H. 2017.

[27] Stephen Dawson-haggerty, Andrew Krioukov, Jay Taneja, Sagar Karandikar, Gabe Fierro, Nikita Kitaev and David Culler. 'BOSS: building operating system services'. In: *NSDI '13* (2013), pp. 1–15.

[28] Gabriel Fierro and David E Culler. 'XBOS: An Extensible Building Operating System'. In: *Proceedings of the 2nd ACM International Conference on Embedded Systems for Energy-Efficient Built Environments*. ACM. 2015, pp. 119–120.

[29] Thomas Weng, Anthony Nwokafor and Yuvraj Agarwal. 'Buildingdepot 2.0: An integrated management system for building analysis and control'. In: *BuildSys'13*. ACM. 2013.

[30] Bora Akyol, Jereme Haack, Brandon Carpenter, Selim Ciraci, Maria Vlachopoulou and Cody Tews. 'Volttron: An agent execution platform for the electric power system'. In: *Third international workshop on agent technologies for energy systems valencia, spain*. 2012.

[31] Anthony Rowe, Mario E Berges, Gaurav Bhatia, Ethan Goldman, Ragunathan Rajkumar, James H Garrett, José MF Moura and Lucio Soibelman. 'Sensor Andrew: Large-scale campus-wide sensing and actuation'. In: *IBM Journal of Research and Development* 55.1.2 (2011), pp. 6–1.

[32] Vykon by Tridium. 'Niagara Networking & Connectivity Guide'. In: *Niagara Release* 2 (), p. 245.

[46] Stephen Dawson-Haggerty, Xiaofan Jiang, Gilman Tolle, Jorge Ortiz and David Culler. 'sMAP: a simple measurement and actuation profile for physical information'. In: *SenSys'10* (2010). doi: 10.1145/1869983.1870003.

[47] Michael P Anderson, John Kolb, Kaifei Chen, David E Culler, Randy Katz, Michael Andersen, John Kolb, Kaifei Chen, David E Culler and Randy Katz.

‘Democratizing Authority in the Built Environment’. In: *BuildSys*. 2017.

[50] KMM Thein. ‘Apache kafka: Next generation distributed messaging system’. In: *International Journal of Scientific Engineering and Technology Research* 3.47 (2014), pp. 9478–9483.

[62] Graham Klyne and Jeremy J Carroll. ‘Resource Description Framework (RDF): Concepts and Abstract Syntax’. In: *W3C Recommendation* 10.October (2004), pp. 1–20. URL: http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/.

[63] Emil Holmegaard, Aslak Johansen and Mikkel Baun Kjærgaard. ‘Metafier - a Tool for Annotating and Structuring Building Metadata’. In: *SmartWorld’17*. 2017.

[69] Eric Prud’hommeaux and Andy Seaborne. ‘SPARQL Query Language for RDF’. In: *W3C Recommendation* (2008). DOI: citeulike-article-id:2620569.

[73] Bharathan Balaji, Arka Bhattacharya, Gabriel Fierro, Jingkun Gao, Joshua Gluck, Dezhi Hong, Aslak Johansen, Jason Koh, Joern Ploennigs, Yuvraj Agarwal, Mario Berges, David Culler, Rajesh Gupta, Mikkel Baun Kjærgaard, Mani Srivastava and Kamin Whitehouse. ‘Brick: Towards a Unified Metadata Schema For Buildings’. In: *BuildSys*. 2016. DOI: 10.1145/2993422.2993577.

[74] Bharathan Balaji, Arka Bhattacharya, Gabriel Fierro, Jingkun Gao, Joshua Gluck, Dezhi Hong, Aslak Johansen, Jason Koh, Joern Ploennigs, Yuvraj Agarwal, Mario Bergés, David Culler, Rajesh K. Gupta, Mikkel Baun Kjærgaard, Mani Srivastava and Kamin Whitehouse. ‘Brick : Metadata schema for portable smart building applications’. In: *Applied Energy* (2018). DOI: 10.1016/J.APENERGY.2018.02.091.

[80] Mikkel Baun Kjaergaard, Aslak Johansen, Fisayo Sangogboye and Emil Holmegaard. ‘OccuRE: An Occupancy REasoning Platform for Occupancy-Driven Applications’. In: *CBSE’16*. IEEE, 2016. DOI: 10.1109/CBSE.2016.14.

[83] W3C. *OWL*. https://www.w3.org/OWL/. 2012.

[84] W3C. *Turtle*. https://www.w3.org/TR/turtle/. 2014.

[86] John Kolb. 'Spawnpoint: Secure Deployment of Distributed, Managed Containers'. PhD thesis. Master's thesis. EECS Department, University of California, Berkeley ..., 2018.

[87] Docker Inc. *Docker Containers*. https://docker.com/. 2019.

[88] Cloud Native Computing Foundation. *Kubernetes Container Orchestrator*. https://kubernetes.io/. 2019.

[89] Todd Gamblin, Matthew LeGendre, Michael R Collette, Gregory L Lee, Adam Moody, Bronis R de Supinski and Scott Futral. 'The Spack package manager: bringing order to HPC software chaos'. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM. 2015, p. 40.

[90] Canonical. *Snapcraft*. https://snapcraft.io/. 2019.

[91] RPM. *RPM Package Manager*. https://rpm.org. 2019.

[92] Debian. *APT*. https://github.com/Debian/apt. 2019.

[96] Mikkel Baun Kjærgaard and Fisayo Caleb Sangogboye. 'Categorization framework and survey of occupancy sensing systems'. In: *PMC* 38 (2017), pp. 1–13.

[113] Fisayo Caleb Sangogboye and Mikkel Baun Kjærgaard. 'PROMT: predicting occupancy presence in multiple resolution with time-shift agnostic classification'. In: *Computer Science-Research and Development* 33.1-2 (2018), pp. 105–115.

[115] Fisayo Caleb Sangoboye and Mikkel Baun Kjærgaard. 'Plcount: A probabilistic fusion algorithm for accurately estimating occupancy from 3d camera counts'. In: *Proceedings of the 3rd ACM International Conference on Systems for Energy-Efficient Built Environments*. ACM. 2016, pp. 147–156.

[116] Ashrae. *ASHRAE's BACnet Committee, Project Haystack and Brick Schema Collaborating to Provide Unified Data Semantic Modeling Solution*. https://www.ashrae.org/about/news/2018. Feb. 2018.

[117] Adrian Mouat.
Using Docker: Developing and deploying software with containers.
" O'Reilly Media, Inc.", 2015.

[118] Muhyiddine Jradi, Fisayo Caleb Sangogboye,
Claudio Giovanni Mattera, Mikkel Baun Kjærgaard,
Christian Veje and Bo Nørregaard Jørgensen. 'A world class
energy efficient university building by Danish 2020 Standards'.
In: *Energy Procedia* 132 (2017), pp. 21–26.

Part V

Bibliography

REFERENCES

[1] Jakob Hviid and Mikkel Baun Kjærgaard. 'The Retail Store as a Smart Grid Ready Building: Current Practice and Future Potentials'. In: *Power & Energy Society Innovative Smart Grid Technologies Conference (ISGT), 2018 IEEE*. IEEE. Washington DC, USA, Oct. 2018. DOI: 10.1109/ISGT.2018.8403354. **Published** (cit. on pp. 6, 9–12, 33, 35, 40, 81, 121, 204, 238, 239).

[2] Jakob Hviid and Mikkel Baun Kjærgaard. 'Activity-Tracking Service for Building Operating Systems'. In: *2018 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. IEEE. 2018, pp. 854–859. DOI: 10.1109/PERCOMW.2018.8480362. **Published** (cit. on pp. 10, 11, 18, 45, 49, 56, 137, 158, 177, 179).

[3] Jakob Hviid and Mikkel Baun Kjærgaard. 'Service Abstraction Layer for Building Operating Systems: Enabling portable applications and improving system resilience'. In: *2018 IEEE International Conference on Communications, Control, and Computing Technologies for Smart Grids (SmartGridComm)*. IEEE. Aalborg, Denmark, Oct. 2018, pp. 1–6. DOI: 10.1109/SmartGridComm.2018.8587543. **Published** (cit. on pp. 10, 12, 18, 55, 60, 61, 82, 83, 96, 99, 155, 176, 183, 206, 207, 215, 228, 240, 246).

[4] Jakob Hviid, Aslak Johansen, Gabe Fierro and Mikkel Kjærgaard. 'Service Portability and Discovery in Building Operating Systems Using Semantic Modeling'. In: *2020 IEEE International Conference on Pervasive Computing and Communications (PerCom) (PerCom 2020)*. ACM. Austin, USA, Mar. 2020. **Submitted** (cit. on pp. 10, 12, 18, 55, 60, 61, 68, 82, 99, 175).

[5] Jakob Hviid, Aslak Johansen, Fisayo Caleb Sangogboye and Mikkel Baun Kjærgaard. 'Enabling Auto-Configuring Building Services: The Road to Affordable Portable Applications for Smart Grid Integration'. In: *Proceedings of the Tenth International Conference on Future Energy Systems (ACM e-Energy '19)*. Phoenix, AZ, USA: ACM, June 2019, pages 68–77. DOI: 10.1145/3307772.3328288. **Published** (cit. on pp. 10, 12, 18, 75, 81, 84, 88, 99, 102, 103, 115, 203, 239, 240, 246, 259).

[6] Jakob Hviid, Aslak Johansen, Fisayo Caleb Sangogboye and Mikkel Kjærgaard. 'OPM: an Ontology Based Package Manager for Building Operating Systems'. In: *Proceedings of the International Conference on Internet of Things Design and Implementation (IoTDI 2020)*. ACM. 2020. **In Submission** (cit. on pp. 10, 12, 18, 55, 78, 81, 98, 115, 237).

[7] Muhyiddine Jradi, Henrik Engelbrecht Foldager, Rasmus Camillus Jeppesen, Jakob Hviid, Mikkel Ask Rasmussen and Mikkel Baun Kjærgaard. 'Modeling and Performance Simulation of a Retail Store as a Smart Grid Ready Building'. In: *Building Simulation 2019*. International Building Performance Simulation Association. Rome, Italy, 2019. **Accepted**.

[8] Institute for Atmospheric and Climate Science (IAC). *Historical CO2 Datasets*. [Online; accessed 29. Jul. 2019]. July 2019. URL: https://www.co2.earth/historical-co2-datasets (cit. on p. 3).

[9] Nicholas Stern. *The Economics of Climate Change: The Stern Review*. Cambridge University Press, Oct. 2006. ISBN: 9780521700801 (cit. on p. 3).

[10] William W Kellogg. *Climate change and society: consequences of increasing atmospheric carbon dioxide*. Routledge, 2019 (cit. on p. 3).

[11] Terry P Hughes, James T Kerry, Andrew H Baird, Sean R Connolly, Andreas Dietzel, C Mark Eakin, Scott F Heron, Andrew S Hoey, Mia O Hoogenboom,

Gang Liu et al.
'Global warming transforms coral reef assemblages'.
In: *Nature* 556.7702 (2018), p. 492 (cit. on p. 3).

[12] Shilong Piao, Philippe Ciais, Yao Huang, Zehao Shen, Shushi Peng, Junsheng Li, Liping Zhou, Hongyan Liu, Yuecun Ma, Yihui Ding et al. 'The impacts of climate change on water resources and agriculture in China'.
In: *Nature* 467.7311 (2010), p. 43 (cit. on p. 3).

[13] Günther Fischer, Mahendra Shah, Francesco N. Tubiello and Harrij Van Velhuizen. 'Socio-economic and climate change impacts on agriculture: an integrated assessment, 1990–2080'.
In: *Philosophical Transactions of the Royal Society B: Biological Sciences* 360.1463 (2005), pp. 2067–2083 (cit. on p. 3).

[14] Coral Davenport.
'Nations approve landmark climate accord in Paris'.
In: *New York Times* 12 (2015) (cit. on p. 3).

[15] Keri Fulton.
Apple now globally powered by 100 percent renewable energy.
[Online; accessed 29. Jul. 2019]. June 2019. URL:
https://www.apple.com/newsroom/2018/04/apple-now-globally-powered-by-100-percent-renewable-energy
(cit. on p. 4).

[16] Bruce Perlstein, Lindsay Battenberg, Erik Gilbert, Robin Maslowski, Frank Stern, Stuart Schare, Karin Corfee and Ryan Firestone.
'Potential Role of Demand Response Resources in Maintaining Grid Stability and Integrating Variable Renewable Energy under California's 33 percent Renewable Portfolio Standard'.
In: *prepared for California's Demand Response Measurement & Evaluation Committee, July* (2012) (cit. on p. 4).

[17] Energinet.dk. *The Energy System Right Now: Historic Data*.
[Online; accessed 28. Jul. 2019]. July 2019.
URL: https://en.energinet.dk (cit. on p. 5).

[18] Mary Ann Piette, Sila Kiliccote and Girish Ghatikar.
'Field Experience with and Potential for Multi-time Scale Grid Transactions from Responsive Commercial Buildings'.

In: *ACEEE Summer Study on Energy Efficency in Buildings*. 2014 (cit. on pp. 5–7, 14, 33, 122, 124, 156, 176, 204, 238).

[19] Mikkel Baun Kjærgaard, Krzysztof Arendt, Anders Clausen, Aslak Johansen, Muhyiddine Jradi, Bo Nørregaard Jørgensen, Peter Nelleman, Fisayo Caleb Sangogboye, Christian T. Veje and Morten Gill Wollsen. 'Demand response in commercial buildings with an Assessable impact on occupant comfort'. In: *SmartGridComm 2016*. 2016, pp. 447–452 (cit. on pp. 5–7, 122, 156, 176).

[20] Peter Nellemann, Mikkel Baun Kjærgaard, Emil Holmegaard, Krzysztof Arendt, Aslak Johansen, Fisayo Caleb Sangogboye and Bo Nørregaard Jørgensen. 'Demand Response with Model Predictive Comfort Compliance in an Office Building'. In: *SmartGridComm'17*. IEEE. 2017 (cit. on pp. 5, 7, 22, 56, 65, 66, 164–166, 177, 179, 210, 211).

[21] Naoya Motegi, Mary Ann Piette, David S Watson, Sila Kiliccote and Peng Xu. 'Introduction to commercial building control strategies and techniques for demand response'. In: *Lawrence Berkeley National Laboratory LBNL-59975* (2007) (cit. on pp. 5, 7, 14, 140).

[22] Therese Peffer, David Auslander, Domenico Caramagno, David Culler, Tyler Jones, Andrew Krioukov, Michael Sankur, Jay Taneja, Jason Trager, Sila Kiliccote et al. 'Deep demand response: The case study of the CITRIS building at the university of California-Berkeley'. In: (2012) (cit. on pp. 5, 7, 158, 179).

[23] European Parliament and Council of the European Union. 'Directive 2010/31/EU of the European Parliament and of the Council on the Energy Performance of Buildings'. In: *Official Journal of the European Union* L153 (May 2010), pp. 13–35. URL: http://eur-lex.europa.eu/eli/dir/2010/31/oj (cit. on pp. 6, 204, 238).

[24] U.S. Department of Energy. *2011 Buildings Energy Data Book*. Tech. rep. U.S. Department of Energy, Mar. 2012. URL: https://catalog.data.gov/dataset/buildings-energy-data%20-book (cit. on pp. 6, 204, 238).

[25] Danmarks Statistik. *ENE3H: Bruttoenergiforbrug i fælles enheder efter branche og energitype [Accessed: 23/2/2017]*. http://www.statistikbanken.dk/ENE3H. 2017 (cit. on pp. 6, 122, 156, 204, 238).

[26] Elena Markoska and Sanja Lazarova-Molnar. 'Usability Requirements for Smart Buildings' Performance Testing Solutions: A Survey'. In: *The Third IEEE International Workshop on Smart Cities Systems Engineering (IEEE SCE 2019)*. IEEE. 2019 (cit. on pp. 6, 15).

[27] Stephen Dawson-haggerty, Andrew Krioukov, Jay Taneja, Sagar Karandikar, Gabe Fierro, Nikita Kitaev and David Culler. 'BOSS: building operating system services'. In: *NSDI '13* (2013), pp. 1–15 (cit. on pp. 7, 16, 47, 66, 129, 132, 138, 139, 141, 144, 158, 164, 166, 176, 178, 205, 207, 239).

[28] Gabriel Fierro and David E Culler. 'XBOS: An Extensible Building Operating System'. In: *Proceedings of the 2nd ACM International Conference on Embedded Systems for Energy-Efficient Built Environments*. ACM. 2015, pp. 119–120 (cit. on pp. 7, 16, 18, 47, 66, 75, 82, 129, 132, 138, 139, 141, 144, 158, 164, 166, 176, 178, 191, 205–207, 214, 239, 240).

[29] Thomas Weng, Anthony Nwokafor and Yuvraj Agarwal. 'Buildingdepot 2.0: An integrated management system for building analysis and control'. In: *BuildSys'13*. ACM. 2013 (cit. on pp. 7, 16, 47, 129, 132, 139, 141, 176, 178, 205, 207, 239).

[30] Bora Akyol, Jereme Haack, Brandon Carpenter, Selim Ciraci, Maria Vlachopoulou and Cody Tews. 'Volttron: An agent execution platform for the electric power system'. In: *Third international workshop on agent technologies for energy systems valencia, spain*. 2012 (cit. on pp. 7, 16, 178, 239).

[31] Anthony Rowe, Mario E Berges, Gaurav Bhatia, Ethan Goldman, Ragunathan Rajkumar, James H Garrett, José MF Moura and Lucio Soibelman. 'Sensor Andrew: Large-scale campus-wide sensing and actuation'. In: *IBM Journal of Research and Development* 55.1.2 (2011), pp. 6–1 (cit. on pp. 7, 16, 178, 239).

[32] Vykon by Tridium.
'Niagara Networking & Connectivity Guide'.
In: *Niagara Release* 2 (), p. 245 (cit. on pp. 7, 16, 178, 239).

[33] S. Shafiei, Henrik Rasmussen and Jakob Stoustrup.
'Modeling Supermarket Refrigeration Systems for Demand-Side Management'.
In: *Energies* 6.2 (Feb. 2013), pp. 900–920. ISSN: 1996-1073.
DOI: 10.3390/en6020900 (cit. on pp. 7, 33, 122).

[34] Ian Sommerville. *Software Engineering*. 9th. Pearson, 2011.
ISBN: 978-0-13-705346-9 (cit. on p. 9).

[35] Bo Nørregaard Jørgensen, Mikkel Baun Kjærgaard, Sanja Lazarova-Molnar, Hamid Reza Shaker and Christian T Veje.
'Challenge: Advancing energy informatics to enable assessable improvements of energy performance in buildings'.
In: *Proceedings of the 2015 ACM Sixth International Conference on Future Energy Systems*. ACM. 2015, pp. 77–82.
DOI: 10.1145/2768510.2770935 (cit. on p. 9).

[36] Gordana Dodig Crnkovic. 'Constructive research and info-computational knowledge generation'.
In: *Model-Based Reasoning in Science and Technology*.
Springer, 2010, pp. 359–380 (cit. on pp. 9, 10).

[37] John W Creswell and J David Creswell. *Research design: Qualitative, quantitative, and mixed methods approaches*.
Sage publications, 2017 (cit. on p. 9).

[38] Per Runeson and Martin Höst. 'Guidelines for conducting and reporting case study research in software engineering'.
In: *Empirical software engineering* 14.2 (2009), p. 131
(cit. on pp. 9, 10).

[39] Charlotte Baarts, Morten Frederiksen, Merete Watt Boolsen, Kennet Lynggaard, Birger Steen Nielsen, Pirkko Raudaskoski, Martyn Hammersley, Thomas Szulevicz, Peter Dahler-Larsen, Louise Jane Phillips, Bo Jacobsen, Judy Gammelgaard, Henrik Kaare Nielsen, Casper Bruun Jensen, Lene Tanggaard, Dorte Marie Søndergaard, Benny Karpatschof, Bent Flyvbjerg, Kurt Aagaard Nielsen, Bente Halkier, Lis Højgaard, Kirsten Hastrup, Hans Hauge, Jakob Steensig, Norman Denzin,

Søren Kristiansen, Svend Brinkmann and Barbara Czarniawska. *Kvalitative metoder*. Hans Reitzels, Jan. 2015. ISBN: 978-874125904-8 (cit. on p. 9).

[40] Charles McParland. 'OpenADR open source toolkit: Developing open source software for the smart grid'. In: *Power and Energy Society General Meeting, 2011 IEEE*. IEEE. 2011, pp. 1–7 (cit. on pp. 13, 66, 157, 166).

[41] IP Byriel, Helle Justesen, A Beck, J Borup Jensen, Kl Rosenfeldt Jakobsen and Steen Hartvig Jacobsen. 'Energy 2007. Research, development, demonstration; Energi 07. Forskning, udvikling, demonstration'. In: (2007) (cit. on p. 14).

[42] Joy E Altwies and Douglas T Reindl. 'Passive thermal energy storage in refrigerated warehouses'. In: *International Journal of refrigeration* 25.1 (2002), pp. 149–157 (cit. on p. 14).

[43] Sasank Goli, Aimee McKane and Daniel Olsen. *Demand response opportunities in industrial refrigerated warehouses in california*. Tech. rep. Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 2011 (cit. on p. 14).

[44] Gratien Bonvin, Sophie Demassey, Claude Le Pape, Nadia Maizi, Vincent Mazauric and Alfredo Samperio. 'A convex mathematical program for pump scheduling in a class of branched water networks'. In: *Applied energy* 185 (2017), pp. 1702–1711 (cit. on p. 14).

[45] Hugo Morais, Tiago Sousa, João Soares, Pedro Faria and Zita Vale. 'Distributed energy resources management using plug-in hybrid electric vehicles as a fuel-shifting demand response resource'. In: *Energy conversion and management* 97 (2015), pp. 78–93 (cit. on p. 15).

[46] Stephen Dawson-Haggerty, Xiaofan Jiang, Gilman Tolle, Jorge Ortiz and David Culler. 'sMAP: a simple measurement and actuation profile for physical information'. In: *SenSys'10* (2010). DOI: 10.1145/1869983.1870003 (cit. on pp. 16, 57, 129, 132, 158, 164, 178, 180, 205, 220, 239, 253).

[47] Michael P Anderson, John Kolb, Kaifei Chen, David E Culler, Randy Katz, Michael Andersen, John Kolb, Kaifei Chen, David E Culler and Randy Katz. 'Democratizing Authority in the Built Environment'. In: *BuildSys*. 2017 (cit. on pp. 16, 18, 47, 50, 66, 75, 85, 139, 141, 144, 145, 158, 164, 166, 178, 191, 214, 243).

[48] Gabriel Fierro and David Culler. 'HodDB: Design and Analysis of a Query Processor for Brick.' In: *IEEE BuildSys '17* (Nov. 2017) (cit. on pp. 18, 84, 159, 168, 207).

[49] Apache Jena. *Apache Jena Fuseki*. https://jena.apache.org/ (cit. on pp. 18, 165).

[50] KMM Thein. 'Apache kafka: Next generation distributed messaging system'. In: *International Journal of Scientific Engineering and Technology Research* 3.47 (2014), pp. 9478–9483 (cit. on pp. 18, 21, 100, 245, 248).

[51] Satoshi Nakamoto. *Bitcoin: A peer-to-peer electronic cash system*. 2008 (cit. on pp. 19, 145, 158).

[52] Gavin Wood. 'Ethereum: A secure decentralised generalised transaction ledger'. In: *Ethereum Project Yellow Paper* 151 (2014) (cit. on pp. 19, 145, 158).

[53] Arun Vishwanath, Vikas Chandan, Cameron Mendoza and Charles Blake. 'A Data Driven Pre-cooling Framework for Energy Cost Optimization in Commercial Buildings'. In: *e-Energy 2017*. 2017, pp. 157–167 (cit. on pp. 22, 211).

[54] Bijay Neupane, Laurynas Siksnys and Torben Bach Pedersen. 'Generation and Evaluation of Flex-Offers from Flexible Electrical Devices'. In: *e-Energy 2017*. 2017, pp. 143–156 (cit. on pp. 22, 211).

[55] Mohammad Hassan Hajiesmaili, Minghua Chen, Enrique Mallada and Chi-Kin Chau. 'Crowd-Sourced Storage-Assisted Demand Response in Microgrids'. In: *e-Energy 2017*. 2017, pp. 91–100 (cit. on pp. 22, 211).

[56] Bainan Xia, Hao Ming, Ki-Yeob Lee, Yuanyuan Li, Yuqi Zhou, Shantanu Bansal, Srinivas Shakkottai and Le Xie. 'EnergyCoupon: A Case Study on Incentive-based Demand Response in Smart Grid'. In: *e-Energy 2017*. 2017, pp. 80–90 (cit. on pp. 22, 211).

[57] Joshua Comden, Zhenhua Liu and Yue Zhao. 'Harnessing Flexible and Reliable Demand Response Under Customer Uncertainties'. In: *e-Energy 2017*. 2017, pp. 67–79 (cit. on pp. 22, 211).

[58] Lukas Barth, Veit Hagenmeyer, Nicole Ludwig and Dorothea Wagner. 'How much demand side flexibility do we need?: Analyzing where to exploit flexibility in industrial processes'. In: *Proceedings of the Ninth International Conference on Future Energy Systems, e-Energy 2018, Karlsruhe, Germany, June 12-15, 2018*. 2018, pp. 43–62 (cit. on pp. 22, 211).

[59] Julian de Hoog, Khalid Abdulla, Ramachandra Rao Kolluri and Paras Karki. 'Scheduling Fast Local Rule-Based Controllers for Optimal Operation of Energy Storage'. In: *e-Energy 2018*. 2018, pp. 168–172 (cit. on pp. 22, 211).

[60] Majid Khonji, Sid Chi-Kin Chau and Khaled M. Elbassioni. 'Challenges in Scheduling Electric Vehicle Charging with Discrete Charging Rates in AC Power Networks'. In: *Proceedings of the Ninth International Conference on Future Energy Systems, e-Energy 2018, Karlsruhe, Germany, June 12-15, 2018*. 2018, pp. 183–186 (cit. on pp. 22, 211).

[61] Lawrence Berkeley National Laboratory. *Taxonomy for Modeling Demand Response Resources*. Tech. rep. LBNL, 2014 (cit. on pp. 23, 209).

[62] Graham Klyne and Jeremy J Carroll. 'Resource Description Framework (RDF): Concepts and Abstract Syntax'. In: *W3C Recommendation* 10.October (2004), pp. 1–20. URL: http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/ (cit. on pp. 24, 68, 96, 165, 185, 215, 228, 246).

[63] Emil Holmegaard, Aslak Johansen and Mikkel Baun Kjærgaard. 'Metafier - a Tool for Annotating and Structuring Building Metadata'. In: *SmartWorld'17*. 2017 (cit. on pp. 24, 57, 82, 159, 180, 206, 207, 240).

[64] Ben Adida and Mark Birbeck. 'RDFa primer'. In: *Bridging the Human and Data Webs* (2008) (cit. on p. 24).

[65] Manu Sporny, Dave Longley, Gregg Kellogg, Markus Lanthaler and Niklas Lindström. 'JSON-LD 1.0'. In: *W3C Recommendation* 16 (2014), p. 41 (cit. on p. 24).

[66] Thomas Steiner, Raphaël Troncy and Michael Hausenblas. 'How google is using linked data today and vision for tomorrow'. In: (2010) (cit. on p. 25).

[67] Google. *Enable Rich Snippets with Structured Data*. [Online; accessed 03. Aug. 2019]. July 2019. URL: https://developers.google.com/search/docs/guides/mark-up-content (cit. on p. 25).

[68] Open Community. *Schema Org*. https://schema.org/ (cit. on pp. 25, 68, 73, 185, 189).

[69] Eric Prud'hommeaux and Andy Seaborne. 'SPARQL Query Language for RDF'. In: *W3C Recommendation* (2008). DOI: citeulike-article-id:2620569 (cit. on pp. 25, 96, 99, 159, 165, 228, 246).

[70] Maria Bermudez-Edo, Tarek Elsaleh, Payam Barnaghi and Kerry Taylor. 'IoT-Lite: a lightweight semantic model for the Internet of Things'. In: *2016 Intl IEEE Conferences on Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld)*. IEEE. 2016, pp. 90–97 (cit. on p. 26).

[71] David Martin, Mark Burstein, Jerry Hobbs, Ora Lassila, Drew McDermott, Sheila McIlraith, Srini Narayanan, Massimo Paolucci, Bijan Parsia, Terry Payne et al. 'OWL-S: Semantic markup for web services'.

In: *W3C member submission* 22.4 (2004)
(cit. on pp. 26, 57, 59, 180, 182).

[72] Jos de Bruijn, Christoph Bussler, John Domingue, Dieter Fensel, Martin Hepp, Uwe Keller, Michael Kifer, Birgitta Konig-Ries, Jacek Kopecky, Ruben Lara, Holger Lausen, Eyal Oren, Axel Polleres, Dumitru Roman, James Scicluna and Michael Stollberg. *Web Service Modeling Ontology (WSMO)*. [Online; accessed 22. Jul. 2019]. Mar. 2014. URL: https://www.w3.org/Submission/WSMO (cit. on pp. 26, 57, 59, 180, 182).

[73] Bharathan Balaji, Arka Bhattacharya, Gabriel Fierro, Jingkun Gao, Joshua Gluck, Dezhi Hong, Aslak Johansen, Jason Koh, Joern Ploennigs, Yuvraj Agarwal, Mario Berges, David Culler, Rajesh Gupta, Mikkel Baun Kjærgaard, Mani Srivastava and Kamin Whitehouse. 'Brick: Towards a Unified Metadata Schema For Buildings'. In: *BuildSys*. 2016. DOI: 10.1145/2993422.2993577 (cit. on pp. 26, 63, 82, 96, 99, 141, 145, 157, 159, 163, 164, 206, 207, 215, 228, 240, 246).

[74] Bharathan Balaji, Arka Bhattacharya, Gabriel Fierro, Jingkun Gao, Joshua Gluck, Dezhi Hong, Aslak Johansen, Jason Koh, Joern Ploennigs, Yuvraj Agarwal, Mario Bergés, David Culler, Rajesh K. Gupta, Mikkel Baun Kjærgaard, Mani Srivastava and Kamin Whitehouse. 'Brick : Metadata schema for portable smart building applications'. In: *Applied Energy* (2018). DOI: 10.1016/J.APENERGY.2018.02.091 (cit. on pp. 26, 56, 57, 59, 63, 75, 96, 99, 114, 157, 159, 163, 164, 177, 180, 182, 191, 215, 228, 246).

[75] Zheng Ma, Joy Dalmacio Billanes, Mikkel Baun Kjaergaard and Bo Nørregaard Jørgensen. 'Energy Flexibility in Retail Buildings: from a Business Ecosystem Perspective'. In: *Proceedings of the 14th International Conference on the European Energy Market*. Dresden, Germany: IEEE, 2017 (cit. on pp. 40, 130).

[76] Jakob E Bardram et al. 'The Java Context Awareness Framework (JCAF)-A Service Infrastructure and Programming Framework for Context-Aware Applications.' In: *Pervasive*. Vol. 3468. Springer. 2005, pp. 98–115 (cit. on pp. 46–48, 138, 139, 141, 142, 147).

[77] Brian L Thomas and Diane J Cook. 'Activity-Aware Energy-Efficient Automation of Smart Buildings'. In: *Energies* 9.8 (2016), p. 624 (cit. on pp. 46, 47, 138, 139, 141, 147).

[78] Elena Markoska, Muhyiddine Jradi and Bo Nørregaard Jørgensen. 'Continuous commissioning of buildings: A case study of a campus building in Denmark'. In: *iThings, GreenCom, CPSCom, and SmartData*. IEEE. 2016, pp. 584–589 (cit. on pp. 56, 177).

[79] Krzysztof Arendt, Ana Ionesi, Muhyiddine Jradi, Ashok Kumar Singh, Mikkel Baun Kjærgaard, Christian Veje and Bo Nørregaard Jørgensen. 'A building model framework for a genetic algorithm multi-objective model predictive control'. In: *REHVA World Congress*. 2016 (cit. on pp. 56, 177).

[80] Mikkel Baun Kjaergaard, Aslak Johansen, Fisayo Sangogboye and Emil Holmegaard. 'OccuRE: An Occupancy REasoning Platform for Occupancy-Driven Applications'. In: *CBSE'16*. IEEE, 2016. DOI: 10.1109/CBSE.2016.14 (cit. on pp. 56, 63, 75, 162, 177, 194, 210, 212, 252).

[81] Erik Christensen, Francisco Curbera, Greg Meredith, Sanjiva Weerawarana et al. *Web services description language (WSDL) 1.1*. 2001 (cit. on pp. 57, 60, 180, 182).

[82] Google Inc. and WeWork Companies Inc. *gRPC: A high performance, open-source universal RPC framework*. https://grpc.io/ (cit. on pp. 57, 60, 180, 182).

[83] W3C. *OWL*. https://www.w3.org/OWL/. 2012 (cit. on pp. 68, 96, 185, 215, 228, 248).

[84] W3C. *Turtle*. https://www.w3.org/TR/turtle/. 2014 (cit. on pp. 75, 191, 215, 247).

[85] Jason Koh, Bharathan Balaji, Dhiman Sengupta, Julian McAuley, Rajesh Gupta and Yuvraj Agarwal. 'Scrabble: transferrable semi-automated semantic metadata normalization using intermediate representation'. In: *Proceedings of the 5th Conference on Systems for Built Environments*. ACM. 2018 (cit. on pp. 83, 95, 206, 207, 227).

[86] John Kolb. 'Spawnpoint: Secure Deployment of Distributed, Managed Containers'. PhD thesis. Master's thesis. EECS Department, University of California, Berkeley ..., 2018 (cit. on pp. 85, 243).

[87] Docker Inc. *Docker Containers*. https://docker.com/. 2019 (cit. on pp. 85, 95, 226, 241, 243).

[88] Cloud Native Computing Foundation. *Kubernetes Container Orchestrator*. https://kubernetes.io/. 2019 (cit. on pp. 85, 243).

[89] Todd Gamblin, Matthew LeGendre, Michael R Collette, Gregory L Lee, Adam Moody, Bronis R de Supinski and Scott Futral. 'The Spack package manager: bringing order to HPC software chaos'. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM. 2015, p. 40 (cit. on pp. 85, 243).

[90] Canonical. *Snapcraft*. https://snapcraft.io/. 2019 (cit. on pp. 85, 243).

[91] RPM. *RPM Package Manager*. https://rpm.org. 2019 (cit. on pp. 85, 243).

[92] Debian. *APT*. https://github.com/Debian/apt. 2019 (cit. on pp. 85, 243).

[93] Jason Koh, Dezhi Hong, Rajesh Gupta, Kamin Whitehouse, Hongning Wang and Yuvraj Agarwal. 'Plaster: an integration, benchmark, and development framework for metadata normalization methods'. In: *Proceedings of the 5th Conference on Systems for Built Environments*. ACM. 2018, pp. 1–10 (cit. on pp. 95, 206, 207, 227).

[94] E.J. Knibbe. *Building management system*. US Patent 5,565,855. Oct. 1996.
URL: https://www.google.com/patents/US5565855
(cit. on p. 123).

[95] Almir Mehanovic, Emil Sebastian Rømer, Jakob Hviid and Mikkel Baun Kjærgaard.
'Clustering and Visualisation of Electricity Data to identify Demand Response Opportunities: Poster Abstract'.
In: *BuildSys 2016*. 2016, pp. 233–234 (cit. on p. 124).

[96] Mikkel Baun Kjærgaard and Fisayo Caleb Sangogboye.
'Categorization framework and survey of occupancy sensing systems'. In: *PMC* 38 (2017), pp. 1–13
(cit. on pp. 138, 210, 252).

[97] Sam Newman.
Building microservices: designing fine-grained systems.
" O'Reilly Media, Inc.", 2015 (cit. on p. 158).

[98] Thomas Erl.
Service-oriented architecture: concepts, technology, and design.
Pearson Education India, 2005 (cit. on p. 158).

[99] Thomas Weng, Bharathan Balaji, Seemanta Dutta, Rajesh Gupta and Yuvraj Agarwal.
'Managing plug-loads for demand response within buildings'.
In: *BuildSys'11*. ACM. 2011 (cit. on pp. 158, 179).

[100] Frauenhofer. *Omega 2.0*.
https://www.iis.fraunhofer.de/en/ff/lv/ener/proj/ogema-2-0.html (cit. on pp. 158, 207).

[101] Flexiblepower-Alliance-Network. *EF-Pi*.
https://flexible-energy.eu/ef-pi/ (cit. on pp. 158, 179).

[102] Arka Bhattacharya, Joern Ploennigs and David Culler.
'Short paper: Analyzing metadata schemas for buildings: The good, the bad, and the ugly'.
In: *Proceedings of the 2nd ACM International Conference on Embedded Systems for Energy-Efficient Built Environments*.
ACM. 2015, pp. 33–34 (cit. on p. 159).

[103] Jon Meyer and Troy Downing. *Java virtual machine*.
O'Reilly & Associates, Inc., 1997 (cit. on p. 159).

[104] Holger Knublauch, Ray W Fergerson, Natalya F Noy and Mark A Musen. 'The Protégé OWL plugin: An open development environment for semantic web applications'. In: *International Semantic Web Conference*. Springer. 2004, pp. 229–243 (cit. on p. 165).

[105] Michael Andersen. *Bindings for BW2*. https://github.com/immesys/bw2bind (cit. on p. 165).

[106] David Huynh, David R Karger, Dennis Quan et al. 'Haystack: A Platform for Creating, Organizing and Visualizing Information Using RDF.' In: *Semantic Web Workshop*. Vol. 52. 2002 (cit. on pp. 177, 215).

[107] Erik Guttman. 'Autoconfiguration for ip networking: Enabling local communication'. In: *IEEE Internet computing* 5.3 (2001), pp. 81–86 (cit. on p. 179).

[108] Alan Presser, Lee Farrell, Devon Kemp and W Lupton. 'Upnp device architecture 1.1'. In: *UPnP Forum*. Vol. 22. 2008 (cit. on p. 179).

[109] Erik Guttman. 'Service location protocol: Automatic discovery of IP network services'. In: *IEEE Internet Computing* 3.4 (1999), pp. 71–80 (cit. on p. 179).

[110] Steven D Gribble, Matt Welsh, Rob Von Behren, Eric A Brewer, David Culler, Nikita Borisov, Steve Czerwinski, Ramakrishna Gummadi, Jason Hill, Anthony Joseph et al. 'The Ninja architecture for robust Internet-scale systems and services'. In: *Computer Networks* 35.4 (2001), pp. 473–497 (cit. on p. 179).

[111] Jim Waldo. 'The Jini architecture for network-centric computing'. In: *Communications of the ACM* 42.7 (1999), pp. 76–76 (cit. on p. 179).

[112] OSGP Community. *OSGP: Open Smart Grid Platform*. https://opensmartgridplatform.org/ (cit. on p. 207).

[113] Fisayo Caleb Sangogboye and Mikkel Baun Kjærgaard. 'PROMT: predicting occupancy presence in multiple resolution with time-shift agnostic classification'. In: *Computer Science-Research and Development* 33.1-2 (2018), pp. 105–115 (cit. on pp. 210, 252).

[114] Fisayo Caleb Sangogboye and Mikkel Baun Kjærgaard. 'PreCount: a predictive model for correcting real-time occupancy count data'. In: *Energy Informatics* 1.1 (2018), p. 12 (cit. on p. 220).

[115] Fisayo Caleb Sangoboye and Mikkel Baun Kjærgaard. 'Plcount: A probabilistic fusion algorithm for accurately estimating occupancy from 3d camera counts'. In: *Proceedings of the 3rd ACM International Conference on Systems for Energy-Efficient Built Environments*. ACM. 2016, pp. 147–156 (cit. on pp. 221, 256).

[116] Ashrae. *ASHRAE's BACnet Committee, Project Haystack and Brick Schema Collaborating to Provide Unified Data Semantic Modeling Solution*. https://www.ashrae.org/about/news/2018. Feb. 2018 (cit. on p. 247).

[117] Adrian Mouat. *Using Docker: Developing and deploying software with containers*. " O'Reilly Media, Inc.", 2015 (cit. on p. 247).

[118] Muhyiddine Jradi, Fisayo Caleb Sangogboye, Claudio Giovanni Mattera, Mikkel Baun Kjærgaard, Christian Veje and Bo Nørregaard Jørgensen. 'A world class energy efficient university building by Danish 2020 Standards'. In: *Energy Procedia* 132 (2017), pp. 21–26 (cit. on p. 257).